ISO/IEC CD 23090-38:2024(E)

ISO/IEC JTC 1/SC 29/WG 7

Secretariat: JISC

**Information technology — Coded representation of immersive media — Part 38: Enhanced geometry-based point cloud compression**

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.

© 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

Fax: +41 22 749 09 47

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

Contents

[Foreword ix](#_Toc184422901)

[Introduction x](#_Toc184422902)

[1 Scope 1](#_Toc184422903)

[2 Normative references 1](#_Toc184422904)

[3 Terms and definitions 1](#_Toc184422905)

[3.1 General terms 1](#_Toc184422906)

[3.2 High-level syntax and entropy coding terms 5](#_Toc184422907)

[3.3 Tree structure terms 7](#_Toc184422908)

[3.4 Geometry coding terms 8](#_Toc184422909)

[3.5 Attribute coding terms 9](#_Toc184422910)

[3.6 Fine granularity slices terms 10](#_Toc184422911)

[4 Abbreviated terms 10](#_Toc184422912)

[4.1 Acronyms 10](#_Toc184422913)

[4.2 Mnemonics 11](#_Toc184422914)

[5 Conventions 11](#_Toc184422915)

[5.1 General 11](#_Toc184422916)

[5.2 Symbolic names 11](#_Toc184422917)

[5.3 Numerical representation 12](#_Toc184422918)

[5.4 Arithmetic operators 12](#_Toc184422919)

[5.5 Logical operators 12](#_Toc184422920)

[5.6 Relational operators 12](#_Toc184422921)

[5.7 Bit-wise operators 13](#_Toc184422922)

[5.8 Assignment operators 13](#_Toc184422923)

[5.9 Range notation 13](#_Toc184422924)

[5.10 Mathematical functions 14](#_Toc184422925)

[5.10.1 General 14](#_Toc184422926)

[5.10.2 IntAtan2 15](#_Toc184422927)

[5.10.3 IntCos and IntSin 15](#_Toc184422928)

[5.10.4 IntSqrt 16](#_Toc184422929)

[5.10.5 IntRecipSqrt 16](#_Toc184422930)

[5.10.6 IntRecip 17](#_Toc184422931)

[5.10.7 Div 18](#_Toc184422932)

[5.10.8 Morton 18](#_Toc184422933)

[5.10.9 FromMorton 18](#_Toc184422934)

[5.11 Order of operation precedence 19](#_Toc184422935)

[5.12 Named expressions 19](#_Toc184422936)

[5.12.1 General 19](#_Toc184422937)

[5.12.2 Scope 20](#_Toc184422938)

[5.12.3 Arguments of named expressions 20](#_Toc184422939)

[5.12.4 Sub-expressions 21](#_Toc184422940)

[5.12.5 Definitions with multiple statements 21](#_Toc184422941)

[5.12.6 Textual definitions 21](#_Toc184422942)

[5.13 Variables, syntax elements and tables 21](#_Toc184422943)

[6 Point cloud format and relationship to coded and output representations 22](#_Toc184422944)

[6.1 General format 22](#_Toc184422945)

[6.2 Attributes 22](#_Toc184422946)

[6.2.1 General 22](#_Toc184422947)

[6.2.2 Colour 22](#_Toc184422948)

[6.2.3 Opacity 23](#_Toc184422949)

[6.2.4 Reflectance 23](#_Toc184422950)

[6.2.5 Normal vector 23](#_Toc184422951)

[6.2.6 Material identifier 23](#_Toc184422952)

[6.2.7 Frame number/index 23](#_Toc184422953)

[6.2.8 User defined attributes 24](#_Toc184422954)

[6.3 Codec-derived attributes 24](#_Toc184422955)

[6.3.1 General 24](#_Toc184422956)

[6.3.2 Slice identifier 24](#_Toc184422957)

[6.3.3 Slice tag 24](#_Toc184422958)

[6.3.4 Canonical point order 24](#_Toc184422959)

[6.3.5 Point Morton order 24](#_Toc184422960)

[6.4 Coded point cloud format 25](#_Toc184422961)

[6.4.1 Sequence coordinate system 25](#_Toc184422962)

[6.4.2 Coding coordinate system 26](#_Toc184422963)

[6.4.3 Coded point cloud sequence 27](#_Toc184422964)

[6.4.4 Coded point cloud frame 27](#_Toc184422965)

[6.4.5 Slice of a coded point cloud frame 27](#_Toc184422966)

[6.4.6 Repetition of slices 28](#_Toc184422967)

[6.4.7 Relationship between tiles and slices 28](#_Toc184422968)

[6.4.8 Parameter sets 29](#_Toc184422969)

[6.5 Output point cloud format 30](#_Toc184422970)

[6.5.1 General 30](#_Toc184422971)

[6.5.2 Coordinate system 30](#_Toc184422972)

[6.5.3 Fixed-point conformance output 30](#_Toc184422973)

[6.5.4 Attributes 30](#_Toc184422974)

[6.5.5 Output point cloud sequence 30](#_Toc184422975)

[6.5.6 Output point cloud frame 30](#_Toc184422976)

[7 Syntax and semantics 31](#_Toc184422977)

[7.1 Method of specifying syntax in tabular form 31](#_Toc184422978)

[7.2 Specification of syntax functions and descriptors 32](#_Toc184422979)

[7.3 Syntax in tabular form 32](#_Toc184422980)

[7.3.1 General 32](#_Toc184422981)

[7.3.2 Parameter sets, ancillary data and byte alignment 32](#_Toc184422982)

[7.3.3 Geometry data unit 42](#_Toc184422983)

[7.3.4 Attribute data unit 50](#_Toc184422984)

[7.3.5 Defaulted attribute data unit syntax 53](#_Toc184422985)

[7.4 Semantics 54](#_Toc184422986)

[7.4.1 General 54](#_Toc184422987)

[7.4.2 Parameter sets, ancillary data and byte alignment 54](#_Toc184422988)

[7.4.3 Geometry data unit 70](#_Toc184422989)

[7.4.4 Attribute data unit 72](#_Toc184422990)

[7.4.5 Defaulted attribute data unit semantics 73](#_Toc184422991)

[8 Decoding process 73](#_Toc184422992)

[8.1 General decoding process 73](#_Toc184422993)

[8.2 Frame decoding processes 74](#_Toc184422994)

[8.2.1 General 74](#_Toc184422995)

[8.2.2 Frame counter 74](#_Toc184422996)

[8.3 Slice decoding processes 74](#_Toc184422997)

[8.3.1 General 74](#_Toc184422998)

[8.3.2 State variables 74](#_Toc184422999)

[8.3.3 Geometry decoding process 75](#_Toc184423000)

[8.3.4 Default attribute values 75](#_Toc184423001)

[8.3.5 Attribute decoding process 75](#_Toc184423002)

[8.3.6 At the end of a slice 75](#_Toc184423003)

[8.3.7 Update of inter prediction buffer 75](#_Toc184423004)

[9 Slice geometry 76](#_Toc184423005)

[9.1 General 76](#_Toc184423006)

[9.2 Occupancy tree 76](#_Toc184423007)

[9.2.1 General 76](#_Toc184423008)

[9.2.2 Coded occupancy tree 76](#_Toc184423009)

[9.2.3 Occupancy tree syntax element semantics 79](#_Toc184423010)

[9.2.4 Node dimensions per tree level 79](#_Toc184423011)

[9.2.5 State representation 80](#_Toc184423012)

[9.2.6 Occupancy tree node coding 80](#_Toc184423013)

[9.2.7 Occupied neighbourhood patterns 86](#_Toc184423014)

[9.2.8 Neighbourhood-permuted node occupancy bitmap 101](#_Toc184423015)

[9.2.9 Dictionary coding of occupancy\_byte 102](#_Toc184423016)

[9.2.10 Bitwise occupancy coding 107](#_Toc184423017)

[9.2.11 Planar occupancy coding 116](#_Toc184423018)

[9.2.12 Direct nodes 125](#_Toc184423019)

[9.2.13 Angular coding 131](#_Toc184423020)

[9.2.14 Subtree scaling 142](#_Toc184423021)

[9.2.15 Reference frame generation 147](#_Toc184423022)

[9.3 Predictive tree 150](#_Toc184423023)

[9.3.1 General 150](#_Toc184423024)

[9.3.2 Syntax element semantics 151](#_Toc184423025)

[9.3.3 Tree traversal for reconstruction of point positions 153](#_Toc184423026)

[9.3.4 Reconstruction of point coordinates 159](#_Toc184423027)

[9.4 TriSoup 162](#_Toc184423028)

[9.4.1 General 162](#_Toc184423029)

[9.4.2 Syntax element semantics 165](#_Toc184423030)

[9.4.3 TriSoup decoding process 169](#_Toc184423031)

[10 Slice attributes 188](#_Toc184423032)

[10.1 General 188](#_Toc184423033)

[10.2 Point coordinates 188](#_Toc184423034)

[10.2.1 General 188](#_Toc184423035)

[10.2.2 Conversion to scaled angular coordinates 189](#_Toc184423036)

[10.3 Syntax element semantics 189](#_Toc184423037)

[10.4 Raw attribute decoding 190](#_Toc184423038)

[10.5 Attribute decoding using the region-adaptive hierarchical transform 190](#_Toc184423039)

[10.5.1 General 190](#_Toc184423040)

[10.5.2 Transform tree 191](#_Toc184423041)

[10.5.3 Coefficient order 194](#_Toc184423042)

[10.5.4 Coefficient scaling 196](#_Toc184423043)

[10.5.5 Transform domain prediction 198](#_Toc184423044)

[10.5.6 Sample domain prediction and last component prediction 206](#_Toc184423045)

[10.5.7 Inverse transform 208](#_Toc184423046)

[10.5.8 Reconstructed attribute values 211](#_Toc184423047)

[10.6 Attribute decoding using levels of detail 211](#_Toc184423048)

[10.6.1 General 211](#_Toc184423049)

[10.6.2 Syntax element semantics 212](#_Toc184423050)

[10.6.3 Reconstruction process 212](#_Toc184423051)

[10.6.4 State variables 212](#_Toc184423052)

[10.6.5 Levels of detail 213](#_Toc184423053)

[10.6.6 Predictor search 220](#_Toc184423054)

[10.6.7 Reconstruction of attribute values 230](#_Toc184423055)

[10.6.8 Prediction mode coding 232](#_Toc184423056)

[10.6.9 Scaling 233](#_Toc184423057)

[10.6.10 Coefficient prediction 234](#_Toc184423058)

[10.6.11 Transform coefficient weights 235](#_Toc184423059)

[10.6.12 Transform 235](#_Toc184423060)

[10.6.13 Point quantization weights 236](#_Toc184423061)

[10.7 Attribute quantization parameters 237](#_Toc184423062)

[10.7.1 Syntax element semantics 237](#_Toc184423063)

[10.7.2 Per-point regional QP offset 238](#_Toc184423064)

[10.7.3 Attribute coefficient QP 238](#_Toc184423065)

[10.7.4 Definition of *AttrQstep* 239](#_Toc184423066)

[10.8 Reference slice generation for attribute 239](#_Toc184423067)

[10.8.1 General 239](#_Toc184423068)

[10.8.2 Reconstructed frame generation 239](#_Toc184423069)

[10.8.3 Reference frame generation 239](#_Toc184423070)

[10.8.4 Reference slice generation 241](#_Toc184423071)

[11 Parsing process 241](#_Toc184423072)

[11.1 General 241](#_Toc184423073)

[11.2 Data unit buffer 244](#_Toc184423074)

[11.2.1 General 244](#_Toc184423075)

[11.2.2 State 244](#_Toc184423076)

[11.2.3 Initialization at the start of parsing a data unit 244](#_Toc184423077)

[11.2.4 Initialization at the start of parsing a geometry data unit footer 244](#_Toc184423078)

[11.2.5 Definition of *DuNextBit* 245](#_Toc184423079)

[11.3 Chunked bytestream parsing 245](#_Toc184423080)

[11.3.1 General 245](#_Toc184423081)

[11.3.2 Chunk syntax 245](#_Toc184423082)

[11.3.3 Chunk semantics 246](#_Toc184423083)

[11.3.4 State 246](#_Toc184423084)

[11.3.5 Span of chunked bytestream data within a data unit 246](#_Toc184423085)

[11.3.6 The chunk buffer 246](#_Toc184423086)

[11.3.7 State update at the start of every CBS 247](#_Toc184423087)

[11.3.8 Unpacking a single chunk 247](#_Toc184423088)

[11.3.9 Definition of *ChunkNextAeBit* 247](#_Toc184423089)

[11.3.10 Definition of *ChunkNextBpBit* 247](#_Toc184423090)

[11.3.11 Boundary between spliced chunked bytestreams 247](#_Toc184423091)

[11.3.12 Location of chunked bytestream boundaries 248](#_Toc184423092)

[11.4 General inverse binarization processes 249](#_Toc184423093)

[11.4.1 Parsing unsigned fixed-length codes (FL) 249](#_Toc184423094)

[11.4.2 Parsing signed fixed-length codes (FL+S) 249](#_Toc184423095)

[11.4.3 Parsing 𝑘-th order exp-Golomb codes (EGk) 249](#_Toc184423096)

[11.4.4 Parsing concatenated truncated unary and 𝑘-th order exp-Golomb codes (TU+EGk) 249](#_Toc184423097)

[11.4.5 Parsing truncated unary codes (TU) 250](#_Toc184423098)

[11.4.6 Mapping process for signed codes 250](#_Toc184423099)

[11.4.7 Parsing ASN.1 object identifiers 250](#_Toc184423100)

[11.5 CABAC parsing processes 251](#_Toc184423101)

[11.5.1 Initialization 251](#_Toc184423102)

[11.5.2 Definition of *AeReadBin* 251](#_Toc184423103)

[11.5.3 Contextual probability models 251](#_Toc184423104)

[11.5.4 Arithmetic decoding engine 257](#_Toc184423105)

[11.6 Parsing state memorization and restoration 259](#_Toc184423106)

[11.6.1 General 259](#_Toc184423107)

[11.6.2 Geometry data units 259](#_Toc184423108)

[11.6.3 Attribute data units 260](#_Toc184423109)

[11.6.4 Defaulted attribute data units 260](#_Toc184423110)

[12 OBUF parsing process 260](#_Toc184423111)

[12.1 General 260](#_Toc184423112)

[12.2 Creation of an OBUF instance 261](#_Toc184423113)

[12.2.1 Creation and initialization of OBUF trees 261](#_Toc184423114)

[12.2.2 Creation and initialization of an array of OBUF ACPMs 262](#_Toc184423115)

[12.3 Creation of a buffer of OBUF tree leaves 262](#_Toc184423116)

[12.4 Call and update of an OBUF instance 263](#_Toc184423117)

[12.4.1 Decode and update according to OBUF trees 263](#_Toc184423118)

[12.4.2 Decode and update according to an element of the buffer of OBUF tree leaves 265](#_Toc184423119)

[12.5 Decode of a bin based on an OBUF ACPM 265](#_Toc184423120)

[12.5.1 enProbability correction based on probability bounds 266](#_Toc184423121)

[12.5.2 Decoding of a bin using CABAC 266](#_Toc184423122)

[Annex A (normative) Profiles and levels 267](#_Toc184423123)

[A.1 Permitted ranges for syntax elements 267](#_Toc184423124)

[Annex B (normative) Type-length-value encapsulated bytestream format 272](#_Toc184423125)

[B.1 General 272](#_Toc184423126)

[B.2 Syntax and semantics 272](#_Toc184423127)

[B.2.1 Syntax 272](#_Toc184423128)

[B.2.2 Semantics 272](#_Toc184423129)

[B.3 Parsing process 273](#_Toc184423130)

[Annex C (informative) Arithmetic encoding engine 274](#_Toc184423131)

[C.1 General 274](#_Toc184423132)

[C.2 State variables 274](#_Toc184423133)

[C.3 Initial state 274](#_Toc184423134)

[C.4 Encoding process for a single binary symbol 274](#_Toc184423135)

[C.5 Arithmetic encoder state renormalization process 275](#_Toc184423136)

[C.6 Arithmetic encoding engine termination process 275](#_Toc184423137)

[Annex D (normative) Partial decoding and spatial scalability 276](#_Toc184423138)

[D.1 General 276](#_Toc184423139)

[D.2 Partial slice geometry 276](#_Toc184423140)

[D.2.1 General 276](#_Toc184423141)

[D.2.2 Selection of partial point positions 276](#_Toc184423142)

[D.2.3 Partial point positions 277](#_Toc184423143)

[D.2.4 Output points 277](#_Toc184423144)

[D.3 Partial attribute decoding 277](#_Toc184423145)

[D.3.1 General 277](#_Toc184423146)

[D.3.2 The finest detail level 277](#_Toc184423147)

[Annex E (normative) Fine granularity slices 278](#_Toc184423148)

[E.1 General 278](#_Toc184423149)

[E.2 Coded point cloud format 278](#_Toc184423150)

[E.2.1 Fine granularity slices 278](#_Toc184423151)

[E.2.2 Layer-group structure 278](#_Toc184423152)

[E.3 Syntax and semantics for fine granularity slices 279](#_Toc184423153)

[E.3.1 Syntax in tabular form 279](#_Toc184423154)

[E.3.2 Semantics 284](#_Toc184423155)

[E.4 Decoding process 288](#_Toc184423156)

[E.4.1 General 288](#_Toc184423157)

[E.4.2 Fine granularity slice decoding process 288](#_Toc184423158)

[E.5 Fine granularity slice geometry 290](#_Toc184423159)

[E.5.1 General 290](#_Toc184423160)

[E.5.2 Partial tree of occupancy tree 290](#_Toc184423161)

[E.6 Fine granularity slice attributes 294](#_Toc184423162)

[E.6.1 General 294](#_Toc184423163)

[E.6.2 Point coordinates 294](#_Toc184423164)

[E.6.3 Attribute decoding using levels of detail 294](#_Toc184423165)

[E.6.4 Parent FGS generation for attributes 299](#_Toc184423166)

[E.7 Parsing process 300](#_Toc184423167)

[E.7.1 General 300](#_Toc184423168)

[E.7.2 Data unit buffer 300](#_Toc184423169)

[E.7.3 CABAC parsing processes 300](#_Toc184423170)

[E.7.4 Parsing state memorization and restoration 300](#_Toc184423171)

[E.8 Partial decoding 302](#_Toc184423172)

[E.8.1 General 302](#_Toc184423173)

[E.8.2 Partial density decoding 303](#_Toc184423174)

[E.8.3 Partial region decoding 304](#_Toc184423175)

[Bibliography 305](#_Toc184423176)

[Index of named expressions and variables 306](#_Toc184423177)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

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 document 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 and IEC 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)) or the IEC list of patent declarations received (see [patents.iec.ch](http://patents.iec.ch/)).

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 Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO/IEC 23090 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

Advancements in 3D capturing and rendering technologies are enabling new applications and services in the fields of assisted and autonomous driving, cartography, cultural heritage, industrial processes, immersive real-time communication, and Virtual/Augmented/Mixed reality (VR/AR/MR) content creation, transmission and communication. Point clouds have arisen as one of the main representations for such applications. A point cloud frame consists of a set of 3D points. Every point, in addition to having a 3D position, may also be associated with numerous other attributes such as colour, transparency, reflectance, timestamp, surface normal and classification. Such representations require a large amount of data, which can be costly in terms of storage and transmission. This document provides the method for efficiently compressing point cloud representations.

**Information technology — Coded representation of immersive media — Part 38: Enhanced geometry-based point cloud compression**

# Scope

This document specifies enhanced geometry-based point cloud compression.

# 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.

*Recommendation ITU‑T T.35, Procedure for the allocation of ITU‑T defined codes for non-standard facilities*

*ISO/IEC 8825‑1 (Rec. ITU‑T X.690), Information technology — ASN.1 encoding rules — Part 1: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

*ISO/IEC 9834‑1 (Rec. ITU‑T X.660), Information technology — Procedures for the operation of object identifier registration authorities — Part 1: General procedures and top arcs of the international object identifier tree*

*ISO/IEC 9834‑8 (Rec. ITU-T X.667), Information technology — Procedures for the operation of object identifier registration authorities — Part 8: Generation of universally unique identifiers (UUIDs) and their use in object identifiers*

*ISO/IEC 23091‑2, Information technology — Coding-independent code points — Part 2: Video*

# Terms and definitions

For the purposes of this document, the following terms and definitions apply.

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

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

## General terms

point

fundamental element of a *point cloud* (3.1.2) comprising a position specified as *Cartesian coordinates* (3.1.9) and zero or more *attributes* (3.1.20)

point cloud

unordered list of *points* (3.1.1)

point cloud sequence

sequence of one or more *point clouds* (3.1.2)

point cloud frame

*point cloud* (3.1.2) in a *point cloud sequence* (3.1.3)

coded point cloud frame

coded representation of a *point cloud frame* (3.1.4)

reference point cloud frame

a coded point cloud frame (3.1.5) which contains points (3.1.1) that may be used for inter prediction (3.1.28) in the decoding process of subsequent point cloud frames (3.1.4) in decoding order

canonical point order  
canonical decoding order

order of *points* (3.1.1) decoded from a *slice* (3.1.22) according to the decoding and parsing processes specified in this document

bounding box

axis-aligned cuboid defining a spatial region that bounds a set of *points* (3.1.1)

coordinates

<Cartesian> three scalar multiples of respective orthogonal *XYZ* (3.1.12) unit vectors with finite precision and bounds that specify a position relative to a fixed reference

coordinates

<angular> a position specified as the radial distance 𝜌 from the V axis, an azimuth angle 𝜑 in the S-T plane and an indexed elevation

coordinates

<attribute> either *STV* (3.1.13) or scaled *RPI* (3.1.14) point coordinates used to code an attribute

XYZ (axes)

X, Y and Z axes, in that order, used to represent *Cartesian coordinates* (3.1.9)

STV (axes)

S, T and V axes, in that order, that are a sequence-dependent permutation of the *XYZ axes* (3.1.12); used to represent the coded *geometry* (3.1.19)

RPI (axes)

R, P and I axes, in that order, used to represent *angular coordinates* (3.1.10)

sequence coordinate system

scaled and translated application-specific coordinate system that applies to an entire coded *point cloud sequence* (3.1.3), and in which all *points* (3.1.1) have non-negative, fixed-point coordinates

coding coordinate system

scaled *sequence coordinate system* (3.1.15) that applies for an entire coded *point cloud sequence* (3.1.3), and in which all *points* (3.1.1) have non-negative integer coordinates

slice coordinate system

translated *coding coordinate system* (3.1.16) that applies for a single *slice* (3.1.22), and in which all *points* (3.1.1) in the *slice* have non-negative integer coordinates

beam

sampler of point positions using angular coordinates by rays cast with a fixed elevation and from a point on and rotating around the V axis at the angular origin

geometry

*point positions* (3.4.1) associated with a set of *points* (3.1.1)

attribute

scalar or vector property associated with each *point* (3.1.1) in a *point cloud* (3.1.2)

EXAMPLE Colour, reflectance, frame index, etc.

position

<bit> bit in a binary string or value, representing the factor

EXAMPLE The LSB has bit position 0.

slice

geometry and attributes for part of, or an entire, *coded point cloud frame* (3.1.5)

Note 1 to entry: the bounding boxes of any two slices can intersect.

tile

set of *slices* (3.1.22) identified by a common slice\_tag *syntax element* value (3.2.15) whose *geometry* (3.1.19) should be contained within a *bounding box* (3.1.8) specified in a tile inventory data unit

prediction

an embodiment of the *prediction process* (3.1.25)

prediction process

the use of a *predictor* (3.1.26) to provide an estimate of the data element currently being decoded

predictor

a combination of specified values or previously decoded data elements used in the decoding process of subsequent data elements

intra prediction

a *prediction* (3.1.24) derived from only data elements (e.g., *point positions* (3.4.1) or *attributes* (3.1.20)) of the same decoded *slice* (3.1.22)

inter prediction

a *prediction* (3.1.24) derived in a manner that is dependent on data elements (e.g., *point positions* (3.4.1) or *attributes*(3.1.20)) of one or more *reference point cloud frames* (3.1.6)

uni-prediction

an inter *prediction* (3.1.28) derived in a manner that is dependent on data elements (e.g., *point positions* (3.4.1) or *attributes*(3.1.20)) of one *reference point cloud frame* (3.1.6)

bi-prediction

an inter *prediction* (3.1.28) derived in a manner that is dependent on data elements (e.g., *point positions* (3.4.1) or *attributes*(3.1.20)) of two *reference point cloud frames* (3.1.6)

I-slice  
intra slice

a *slice* (3.1.22) that is decoded using *intra prediction* (3.1.27) without referring to a *reference point cloud frame* (3.1.6)

reference slice

a coded *slice* (3.1.22) which contains points (3.1.1) that may be used for inter prediction (3.1.28) in the decoding process of subsequent *slices* (3.1.22) in decoding order

I-frame  
intra frame

a *point cloud frame* (3.1.4) that is decoded using *intra prediction* (3.1.27) without referring to a *reference point cloud frame* (3.1.6)

P-frame  
predictive frame

a *point cloud frame* (3.1.4) that is decoded using *intra prediction* (3.1.27) or *inter prediction* (3.1.28) from at most one *reference point cloud frame* (3.1.6)

B-frame  
bi-predictive frame

a *point cloud frame* (3.1.4) that is decoded using *intra prediction* (3.1.27) or *inter prediction* (3.1.28) from at most two *reference point cloud frames* (3.1.6)

Morton code

non-negative integer obtained by interleaving the bits of three integers

Morton order

elements ordered according to their *Morton code* (3.1.36)

sparse array

array with fewer set elements than total addressable elements; unset elements can have an inferred value when accessed

temporal ID

An identifier associated with a point cloud frame which may be used to support temporal scalability.

## High-level syntax and entropy coding terms

ASN.1  
abstract syntax notation one

notation specified by Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1 that is used for the definition of data types, values and constraints on data types

[SOURCE: Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1]

bin

binary symbol (bit) of the *binarized* (3.2.3) representation of a *syntax element* value (3.2.15)

binarization

specification of a *syntax element*'s value (3.2.15) as a sequence of *bins* (3.2.2)

bypass

<symbol> a static, equiprobable probability model

bypass

<stream> *bypass symbols* (3.2.4) that are not encoded in an arithmetic-coded *bitstream* (3.2.6)

bitstream

<data> sequence of bits

bitstream

<coded sequence> sequence of bits, in the form of encapsulated *data units* (3.2.12), that represents a coded *point cloud sequence* (3.1.3)

set bit

bit with the value 1

unset bit

bit with the value 0

byte

sequence of 8 bits, typeset with the most significant bit on the left and the least significant bit on the right.

Note 1 to entry: When represented in a bitstream, the most significant bit of a byte is first.

byte aligned

*bitstream* (3.2.6) position that is an integer multiple of eight bits from the position of the first bit in the bitstream

data unit  
DU

sequence of *bytes* (3.2.10) conveying a single *syntax structure* (3.2.16) of known length

data unit header

parameters, located from the start of a *data unit* (3.2.12)

data unit footer

parameters, located from the end of a *data unit* (3.2.12)

syntax element

element of data represented in the *bitstream* (3.2.6)

syntax structure

zero or more *syntax elements* (3.2.15) present together in the *bitstream* (3.2.6) in a specified order

parameter set

collection of parameters that apply when activated

sequence parameter set  
SPS

parameters for an entire coded *point cloud sequence* (3.1.3), conveyed by an SPS *data unit* (3.2.12) and activated when referenced by a geometry data unit

geometry parameter set  
GPS

parameters for the coding of *slice* (3.1.22) geometry, conveyed by a GPS *data unit* (3.2.12) and activated when referenced by a geometry data unit

attribute parameter set  
APS

parameters for the coding of a *slice* (3.1.22) attribute, conveyed by an APS *data unit* (3.2.12) and activated when referenced by an attribute data unit

object identifier  
OID

<ASN.1> ordered list of primary integer values from the root of the *international object identifier tree* (3.2.22) to a node, which unambiguously identifies that node

[SOURCE: Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1]

international object identifier tree

tree whose root corresponds to Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1 and whose nodes correspond to *registration authorities* (3.2.24) responsible for allocating arcs from a parent node

[SOURCE: Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1]

registration

<object identifier> assignment of an unambiguous name to an object in a way which makes the assignment available to interested parties

[SOURCE: Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1]

registration authority

<international object identifier tree> an entity such as an organization, a standard or an automated facility that performs *registration* (3.2.23) of one or more types of objects

[SOURCE: Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1]

application specific

defined by an application or an application standard

unspecified

when used in subclauses specifying values of a particular *syntax element*, indicates that the values have no specified meaning in this document and will not have a specified meaning in future versions of this document

## Tree structure terms

tree

recursive structure of *nodes* (3.3.7) without loops, and containing a single *root node* (3.3.5)

top

<tree> *tree level* (3.3.4) with *depth* of 0 (3.3.8), consisting of the *root node* (3.3.5)

bottom

<tree> *tree level* (3.3.4) with the greatest *depth* (3.3.8)

tree level

set of *nodes* (3.3.7) at the same *depth* (3.3.8) in a *tree* (3.3.1)

root node

<tree> *node* (3.3.7) without a *parent node* (3.3.10)

leaf node

terminal *node* (3.3.7) without any *child nodes* (3.3.9)

node

<tree> element of a *tree* (3.3.1)

depth

<node> number of descendent hops from the *root node* (3.3.5) to a *node* (3.3.7)

child node

direct descendent of a *node* (3.3.7)

parent node

direct ancestor of a *node* (3.3.7)

grandparent node

direct ancestor of a *node*'s (3.3.7) *parent node* (3.3.10)

great-grandparent node

direct ancestor of a *node*'s (3.3.7) *grandparent node* (3.3.11)

sibling nodes

*nodes* (3.3.7) that are *child nodes* (3.3.9) of the same *parent node* (3.3.10)

subtree

part of a *tree* (3.3.1) comprising a *subtree root node* (3.3.15) and all its descendents over all subsequent *tree levels* (3.3.4)

root node

<subtree> single *node* (3.3.7) of a *subtree* (3.3.14) from which all other nodes in the same subtree are descendents

TriSoup node

occupied *leaf node* (3.3.6) of an *occupancy tree* (3.4.2) representing a sub-volume of the 3D space (or volume) containing at least one point of the *point cloud* (3.1.2)

## Geometry coding terms

position

<point> three-dimensional coordinates of a *point* (3.1.1)

occupancy tree

eight-ary *tree* (3.3.1) of *occupancy tree nodes* (3.4.4) representing the *geometry* (3.1.19) of a *slice* (3.1.22)

predictive tree

*tree* (3.3.1) of *predictive tree nodes* (3.4.5) representing the *geometry* (3.1.19) of a *slice* (3.1.22)

node

<occupancy tree> *node* (3.3.7) of an *occupancy tree* (3.4.2) representing a sub-volume of the 3D space (or volume) containing the *point cloud* (3.1.2)

node

<predictive tree> *node* (3.3.7) of a *predictive tree* (3.4.3) representing a single *position* (3.4.1) for one or more *points* (3.1.1)

direct node

<occupancy tree> terminal *node* (3.4.4) that codes one or more *point positions* (3.4.1)

occupancy bitmap

8-bit bitmap for an occupancy tree *node* (3.4.4) whose bits indicate the existence of *child nodes* (3.3.9) at particular locations in the next *tree level* (3.3.4)

occupied neighbourhood pattern

indicates the existence and arrangement of the six possible occupancy tree *nodes* (3.4.4) that share faces with a central node

TriSoup

representation of the geometry of the *point cloud* (3.1.2) in a *TriSoup node* (3.3.16) by a set of triangles

TriSoup edge

any edge belonging to a cuboid volume associated with a *TriSoup node* (3.3.16)

TriSoup edge vertex

a *TriSoup vertex* (3.4.13) located on a *TriSoup edge* (3.4.10)

TriSoup face vertex

a *TriSoup vertex* (3.4.13) located on a face shared by two adjacent *TriSoup* nodes (3.3.16)

TriSoup vertex

a vertex of *TriSoup triangle* (3.4.14) including *TriSoup edge vertex* (3.4.11) and *TriSoup face vertex* (3.4.12)

TriSoup triangle

a triangle belonging to a *TriSoup node* (3.3.16) and whose vertex are defined from the *TriSoup vertices* (3.4.13) belonging to the cuboid volume associated with said TriSoup node

voxelization (of a TriSoup triangle)

process of transforming a *TriSoup* triangle (3.4.14) into a set of points (3.1.1)

## Attribute coding terms

primary attribute component

first, or only, attribute component, identified by the index 0

secondary attribute component

attribute component other than the first component, identified by an index greater than 0

detail level

set of *points* (3.1.1) that represent a subsampled version of the slice *geometry* (3.1.19)

refinement list

set of *points* (3.1.1) present in one *detail level* (3.5.3) that are not present in the next coarsest *detail level*

refinement point

*point* (3.1.1) in a *refinement list* (3.5.4)

predictor set

set of neighbouring *points* (3.1.1) from which an *attribute* (3.1.20) value is predicted

## Fine granularity slices terms

fine granularity slices

a subset of a slice which carries a geometry or an attribute of a subgroup (3.6.3) in a layer-group (3.6.2)

layer-group

a group of consecutive tree levels (3.3.4) of an occupancy tree (3.4.2)

subgroup

a spatial subset of a layer-group (3.6.2) where the bounding box of a subgroup shall not overlap with other subgroups in the same layer-group

root layer-group

a layer-group (3.6.2) which contains a root node (3.3.5) of the occupancy tree (3.4.2)

parent subgroup

a layer-group (3.6.2) which contains a root node (3.3.5) of the occupancy tree (3.4.2)

child subgroup

a subgroup (3.6.3) in a layer-group (3.6.2) adjacent to the maximum depth of the current subgroup, where the bounding box of the child subgroup is a subset of the bounding box of the current subgroup

# Abbreviated terms

## Acronyms

APS Attribute parameter set

ADU Attribute data unit

CBS Chunked bytestream

CPM Contextual probability model

DADU Dependent attribute data unit

DGDU Dependent geometry data unit

DU Data unit

FBDU Frame boundary marker data unit

FGS Fine granularity slice

FSAP Frame-specific attribute properties

GDU Geometry data unit

GOF Group of frames

GPS Geometry parameter set

G-PCC Geometry-based point cloud compression

LoD Level(s) of detail

LSB Least significant bit

MSB Most significant bit

NA Not applicable

QP Quantization parameter

RAHT Region adaptive hierarchical transform

SPS Sequence parameter set

## Mnemonics

EGk Exponential Golomb code of order 𝑘

FL Fixed-length code

FL+S Fixed-length code plus conditional sign bit

TU Truncated unary code

attr Attribute

cnt Count

geom Geometry

idx Index

occtree Occupancy tree

occ Occupancy tree node

ptree Predictive tree

ptn Predictive tree node

ti Tile inventory

tlv Type-length-value

seq Sequence

# Conventions

## General

1. The mathematical operators used in this document are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are defined more precisely, and additional operations are defined, such as exponentiation and real-valued division. Numbering and counting conventions generally begin from 0.

## Symbolic names

Variables and expressions use the following case-insensitive naming conventions to indicate their use:

* 𝑖, 𝑗: general loop or index variable
* 𝑘: component of an XYZ/STV/RPI position, coordinate or location
* 𝑐: component of an attribute
* qc: quantization-parameterized component: 0 – primary; 1 – secondary
* dpth: the depth of a node or level in a tree
* lvl: tree level or detail level, counted from the bottom of a tree or hierarchy
* 𝑚: Morton-coded location
* ns, nt, nv: node coordinates
* nsc, ntc, nvc: child node coordinates
* nsp, ntp, nvp: parent node coordinates
* ptIdx: index of a point in canonical decoding order
* rfmtIdx: index of a point in an array of LoD refinement points
* ni: index for an element in a point's predictor set

## Numerical representation

|  |  |
| --- | --- |
| binary | typeset as 'X…XX' where each base 2-digit X is 0 or 1 |
| octal | typeset as X…XX8 where each base 8-digit X is 0 to 7 |
| decimal | typeset as X…XX where each base 10-digit X is 0 to 9 |
| hexadecimal | typeset as 0xX…XX where each base 16-digit X is 0 to 9 or A to F |

## Arithmetic operators

|  |  |
| --- | --- |
| + | Addition |
| − | Subtraction (as a two-argument operator) or negation (as a unary prefix operator) |
| × | Multiplication |
|  | Exponentiation. Specifies 𝑥 to the power of 𝑦. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation. |
| / | Integer division with truncation of the result toward zero. For example, 7 / 4 and −7 / −4 are truncated to 1 and −7 / 4 and 7 / −4 are truncated to −1. |
| ÷ | Division where no truncation or rounding is intended |
|  | Division in mathematical equations where no truncation or rounding is intended |
|  | Summation of 𝑓( 𝑖 ) with 𝑖 taking all integer values from 𝑥 up to and including 𝑦 |
| 𝑥 % 𝑦 | Modulus, remainder of 𝑥 divided by 𝑦, defined only for integers 𝑥 and 𝑦 with 𝑥 ≥ 0 and 𝑦 > 0 |

## Logical operators

|  |  |
| --- | --- |
| 𝑥 && 𝑦 | Conditional boolean logical "and" of 𝑥 and 𝑦; the operand 𝑦 is only evaluated if 𝑥 is true. |
| 𝑥 || 𝑦 | Conditional boolean logical "or" of 𝑥 and 𝑦; the operand 𝑦 is only evaluated if 𝑥 is false. |
| ¬ | Boolean logical "not" |
| 𝑥 ? 𝑦 : 𝑧 | If 𝑥 is true or not equal to 0, evaluates to 𝑦; otherwise, evaluates to 𝑧 |

## Relational operators

|  |  |
| --- | --- |
| > | Greater than |
| ≥ | Greater than or equal to |
| < | Less than |
| ≤ | Less than or equal to |
| == | Equal to |
| ≠ | Not equal to |

## Bit-wise operators

|  |  |
| --- | --- |
| & | Bit-wise "and". When operating on integer arguments, operates upon a two's complement representation of the integer value. When operating upon a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding MSBs equal to 0. |
| | | Bit-wise "or". When operating on integer arguments, operates upon a two's complement representation of the integer value. When operating upon a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding MSBs equal to 0. |
| ^ | Bit-wise "exclusive or". When operating on integer arguments, operates upon a two's complement representation of the integer value. When operating upon a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding MSBs equal to 0. |
| 𝑥 >> 𝑦 | Arithmetic right shift as specified by DivExp2Floor( 𝑥, 𝑦 ). It is equivalent to shifting a two's complement integer representation of 𝑥 by 𝑦 binary digits. This operator is defined only for non-negative integer values of 𝑦. |
| 𝑥 << 𝑦 | Arithmetic left shift of a two's complement integer representation of 𝑥 by 𝑦 binary digits. This operator is defined only for non-negative integer values of 𝑦. Bits shifted into the LSBs as a result of the left shift have a value equal to 0. |

1. According to the rules of precedence (5.11), the expressions 𝑎 + 𝑏 << 𝑐 + 𝑑 and 𝑎 | 𝑏 << 𝑐 | 𝑑 are identical to ( 𝑎 + 𝑏 ) << ( 𝑐 + 𝑑 ) and 𝑎 | ( 𝑏 << 𝑐 ) | 𝑑, respectively, and not 𝑎 + ( 𝑏 << 𝑐 ) + 𝑑 or ( 𝑎 | 𝑏 ) << ( 𝑐 | 𝑑 ).

## Assignment operators

|  |  |
| --- | --- |
| = | Assignment operator |
| := | Expression definition (5.12) |
| ++ | Increment, i.e. 𝑥++ is equivalent to 𝑥 = 𝑥 + 1; when used in an array index, evaluates to the value of the variable prior to the increment operation |
| −− | Decrement, i.e. 𝑥−− is equivalent to 𝑥 = 𝑥 − 1; when used in an array index, evaluates to the value of the variable prior to the decrement operation |
| ×= | Multiply by amount specified and update, i.e. 𝑥 ×= 3 is equivalent to 𝑥 = 𝑥 × 3 |
| += | Increment by amount specified, i.e. 𝑥 += 3 is equivalent to 𝑥 = 𝑥 + 3, and 𝑥 += (−3) is equivalent to 𝑥 = 𝑥 + (−3) |
| −= | Decrement by amount specified, i.e. 𝑥 −= 3 is equivalent to 𝑥 = 𝑥 − 3, and 𝑥 −= (−3) is equivalent to 𝑥 = 𝑥 − (−3) |
| >>= | Arithmetic right shift by amount specified, i.e. 𝑥 >>= 1 is equivalent to 𝑥 = 𝑥 >> 1 |
| <<= | Arithmetic left shift by amount specified, i.e. 𝑥 <<= 1 is equivalent to 𝑥 = 𝑥 << 1 |

## Range notation

|  |  |
| --- | --- |
| 𝑥 = 𝑎 .. 𝑏 | 𝑥 takes on monotonically increasing integer values starting from 𝑎 and proceeding to 𝑏, inclusive, with 𝑥, 𝑎 and 𝑏 being integer numbers |

## Mathematical functions

### General

|  |  |
| --- | --- |
| ArcSin( 𝑥 ) | Trigonometric arc sine function |
| Abs( 𝑥 ) |  |
| Bit( 𝑥, 𝑖 ) | ( 𝑥 >> 𝑖 ) & 1 |
| Ceil( 𝑥 ) | Lowest integer greater than or equal to 𝑥 |
| Clip3( min, max, 𝑥 ) |  |
| CrossProduct( *v1*, *v2* ) |  |
| DivExp2Floor( 𝑥, 𝑝 ) |  |
| DivExp2Fz( 𝑥, 𝑝 ) |  |
| DivExp2Tz( 𝑥, 𝑝 ) | 𝑥 / Exp2( 𝑝 ) |
| DivExp2Up( 𝑥, 𝑝 ) |  |
| DivExp2Inf( 𝑥, 𝑝 ) |  |
| Exp2( 𝑝 ) |  |
| Floor( 𝑥 ) | Greatest integer less than or equal to 𝑥 |
| Gcd( 𝑎, 𝑏 ) | Greatest integer that is a factor of both 𝑎 and 𝑏 |
| InnerProduct( *v1*, *v2* ) |  |
| IntLog2( 𝑥 ) | Floor( Log( 𝑥 ) ÷ Log( 2 ) ) |
| Log( 𝑥 ) | Natural logarithm of the argument 𝑥 |
| Min( 𝑥, 𝑦 ) |  |
| MinVec( *v*) | Min( *v*[0], Min( *v*[1], *v*[2] ) ) where *v* is a three-dimensional vector |
| Max( 𝑥, 𝑦 ) |  |
| MaxVec( *v*) | Max( *v*[0], Max( *v*[1], *v*[2] ) ) where *v* is a three-dimensional vector |
| PopCnt( 𝑥 ) | Number of set bits present in the binary representation of 𝑥 |
| RoundFz( 𝑥 ) | ( 2 × 𝑥 + Sign( 𝑥 ) ) / 2 |
| RoundUp( 𝑥 ) | ( 2 × 𝑥 + 1 ) / 2 |
| Sign( 𝑥 ) |  |
| Sin( 𝑥 ) | Trigonometric sine function |
| Sqrt( 𝑥 ) |  |

### IntAtan2

The function theta = IntAtan2( 𝑦, 𝑥 ) is a 20-bit fixed-point approximation of the arc tangent of that accounts for the Cartesian quadrant of the parameters. Its:

* parameters 𝑥 and 𝑦 are integer, Cartesian coordinates;
* result shall be equal to the value of the expression intAtan2[ 𝑦 ][ 𝑥 ].

The expression sineThetaI is for a right-angled triangle with catheti adj and opp in the first octant.

sineThetaI := opp × IntRecipSqrt(adj × adj + opp × opp) >> 20  
 where  
 opp := Min(Abs(y), Abs(x))  
 adj := Max(Abs(y), Abs(x))

The angle thetaI is derived by interpolating between values of ArcSinFp for a 9-bit approximation of sine theta.

thetaI := ArcSinFp[idx0] + (alpha × (ArcSinFp[idx1] − ArcSinFp[idx0]) >> 11)  
 where  
 idx0 := sineThetaI >> 11  
 idx1 := Min(362, idx0 + 1)  
 alpha = sineThetaI % Exp2(11)

The expression ArcSinFp[ 𝑥 ] specifies the 20-bit fixed-point approximation for arc sine.

The result is obtained by mapping thetaI to the correct octant according to the signs of the parameters.

intAtan2[y][x] := Sign(y) × thetaIh  
 where thetaIh :=  
 x > 0 && Abs(x) ≥ Abs(y) ? thetaI :  
 x > 0 && Abs(x) < Abs(y) ? 1647099 − thetaI :  
 x < 0 && Abs(x) < Abs(y) ? 1647100 + thetaI :  
 x < 0 && Abs(x) ≥ Abs(y) ? 3294177 − thetaI : 0

### IntCos and IntSin

The functions 𝑥 = IntCos( 𝜃, piBits ) and 𝑦 = IntSin( 𝜃, piBits ) are 24-bit fixed-point approximations of the cosine and sine of 𝜃. Their:

* parameters 𝜃 and piBits specify an angle measured in units of half turns;
* result shall be equal to the value of the expression cathetus.

The fixed-point cathetus for the unit circle is calculated by interpolating between values of SinFp. The values of sgn and theta are determined from 𝜃 for the corresponding function as specified by Table 1. The variable pi, equal to Exp2( piBits ), represents one half turn.

cathetus := sgn × DivExp2Up(iFrac0 × SinFp[idx0] + iFrac1 × SinFp[idx1], fracBits)  
 where  
 fracBits := piBits − 11  
 iFrac1 := theta − ((theta >> fracBits) << fracBits)  
 iFrac0 := Exp2(fracBits) − iFrac1  
 idx0 := Min(1024, theta >> fracBits)  
 idx1 := Min(1024, idx0 + 1)

Table 1 — Values of sgn and theta for functions IntCos and IntSin

| Domain | IntCos | | IntSin | |
| --- | --- | --- | --- | --- |
| sgn | theta | sgn | theta |
| 𝜃 ≤ −pi | −1 | pi/2 | 0 | 0 |
| −pi < 𝜃 < −pi/2 | −1 | −𝜃 − pi/2 | −1 | pi + 𝜃 |
| −pi/2 ≤ 𝜃 < 0 | 1 | 𝜃 + pi/2 | −1 | −𝜃 |
| 0 ≤ 𝜃 < pi/2 | 1 | −𝜃 + pi/2 | 1 | 𝜃 |
| pi/2 ≤ 𝜃 < pi | −1 | 𝜃 − pi/2 | 1 | pi − 𝜃 |
| 𝜃 ≥ pi | −1 | pi/2 | 0 | 0 |

The expression SinFp[ 𝑥 ] specifies the 24-bit fixed-point approximation of sine.

### IntSqrt

The function 𝑟 = IntSqrt( 𝑥 ) is an integer approximation of the principal square root of 𝑥. Its:

* parameter 𝑥 is a non-negative integer;
* result shall be equal to the value of the expression intSqrt[ 𝑥 ].

It is specified in terms of the fixed-point reciprocal square root. If the parameter 𝑥 is greater than or equal to , the calculation uses a quantized value of 𝑥 to ensure computability using 64-bit arithmetic.

1. IntSqrt( 0 ) is 1.

intSqrt[x] := x ≤ Exp2(46)  
 ? 1 + (x × IntRecipSqrt(x) >> 40)  
 : 1 + (x8 × IntRecipSqrt(x8) >> 32)  
 where  
 x8 := DivExp2(x, 16) + 1

### IntRecipSqrt

The function rRecip = IntRecipSqrt( 𝑥 ) is a 40-bit fixed-point approximation of the reciprocal square root of 𝑥. Its:

* parameter 𝑥 is a non-negative integer;
* result shall be equal to the value of the expression intRecipSqrt[ 𝑥 ].

1. IntRecipSqrt( 0 ) is 0.

The parameter 𝑥 is scaled to be in the range in the expression xScaled by multiplying or dividing by a power of four. The expression xScaleLog4 is the log4 scale factor.

xScaled := Floor(x × Exp2(2 × xScaleLog4))  
xScaleLog4 := 15 − IntLog2(x) / 2

The reciprocal square root shall be determined by two rounds of the Newton–Raphson method. The initial approximation for the scaled parameter 𝑥 is specified by the expression approxR0. Table 2 specifies the initial approximants over the domain of approxR with 18 fractional bit precision.

approxR0 := approxR[xScaled >> 25]

The second approximation from the first round of the Newton–Raphson method is specified by the expression approxR1.

approxR1 := threeR0[approxR0] − (rCubed0[approxR0] × xScaled >> 32)  
threeR0[r] := 3 × DivExp2Fz(r, 18) << 22  
rCubed0[r] := DivExp2Fz(r × r × r, 54) << 8

The third approximation from the second round of the Newton–Raphson method is specified by the expression approxR2.

approxR2 := threeR1[approxR1] − rCubed1[approxR1] >> 32  
threeR1[r] := r × 3 << 28  
rCubed1[r] := r × (r × (r × pInScaled >> 32) >> 32)

The result is obtained by scaling the third approximation by the square root of the initial scale factor.

intRecipSqrt[x] := x > 0 ? Floor(approxR2 × Exp2(xScaleLog4 − 3)) : 0

Table 2 — Initial approximations approxR[ 𝑖 + 𝑗 ] for IntRecipSqrt( 𝑖 + 𝑗 << 25 ). Values are typeset in hexadecimal form without the 0x prefix.

| 𝑗 | 𝑖 | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **32** | 3F7FFDA | 3E7FFB7 | 3DBFFBD | 3CC0013 | 3BFFFEE | 3AFFFE1 | 3A3FFDE | 397FF96 |
| **40** | 38FFFDE | 383FFC0 | 3780063 | 36FFFCD | 3640014 | 35BFFFA | 34FFF8B | 3480010 |
| **48** | 3400039 | 3380042 | 3300008 | 328002B | 3200046 | 317FFC2 | 3100012 | 307FFDB |
| **56** | 303FFC5 | 2FC004F | 2F3FFE0 | 2EFFF93 | 2E7FF91 | 2E3FF83 | 2DC0037 | 2D7FFB6 |
| **64** | 2D0000B | 2CBFF96 | 2C7FF66 | 2C00017 | 2BC0053 | 2B7FFA4 | 2AFFF43 | 2AC004B |
| **72** | 2A80061 | 2A3FF6A | 2A00032 | 29BFF3B | 297FF74 | 293FFC9 | 28C001E | 287FF6D |
| **80** | 283FF9E | 27FFF93 | 27BFFB1 | 27BFFE8 | 277FF3C | 2700056 | 2700000 | 26C0069 |
| **88** | 26800CE | 26400CD | 25FFF5A | 25BFFA8 | 25BFFD6 | 258008C | 25400B9 | 2500020 |
| **96** | 24BFF93 | 24C00B6 | 24800E6 | 24400B3 | 2400011 | 2400054 | 23C0049 | 237FF98 |
| **104** | 2380104 | 233FFFC | 2300047 | 2300024 | 22C0029 | 2280000 | 2280012 | 223FF79 |
| **112** | 223FF54 | 21FFF56 | 21BFFDE | 21C0078 | 2180111 | 217FF3D | 2140023 | 20FFF0F |
| **120** | 21000F1 | 20C0019 | 20C0137 | 2080015 | 2080091 | 204004F | 20400F7 | 200006B |

### IntRecip

The function ( recip, *fracBits* ) = IntRecip(divisor ) returns a fixed-point approximation of 1 ÷ divisor and the number of fractional bits in the fixed-point approximation. Its:

* parameter divisor is a non-null integer;
* result is the tuple specified by the value of the expressions recip and *fracBits* .

The number of fractional bits in the fixed-point approximation is provided by fracBits expression.

fracBits := (31 << 1) – fracBitsOffset

Where the fracBitsOffset expression is determined by

fracBitsOffset := 30 – IntLog2(divisor)

1. fracBitsOffset is determined to maximise the number of bits during the computation of recip while letting the possibly of computing it into 64 bits signed integer registers without any risk of overflow.

The parameter divisor is scaled to be in the range in the expression scaledDivisor by multiplying or dividing by a power of two.

scaledDivisor := Floor(divisor × Exp2(fracBitsOffset))

The approximation of 1 ÷ divisor shall be determined by one iteration of the Newton–Raphson division approximation method. The initial approximation for the method is specified by the expression recip0 and corresponds to a first order approximation of 1 ÷ divisor.

recip0 := (Round(48 ÷ 17 × Exp2(28)) << 31) - Round(32 ÷ 17 × Exp2(28)) × scaledDivisor

The recip expression corresponds to the final approximation and is specified from the first round of the Newton–Raphson method.

recip := recip0 × ((1 << 31) - (scaleDivisor × recip0 >> 31)) >> 31

### Div

The function quotient = Div( dividend, divisor, fracBits ) is a fixed-point approximation of dividend ÷ divisor. Its:

* parameters dividend and divisor are integers;
* parameter fracBits is the number of fractional bits in the fixed-point result;
* result is specified by the value of the expression quotient.

quotient := dividend × recipDivisor[idx] >> 16 + excess − fracBits  
 where  
 idx := DivExp2Fz(divisor, excess)  
 excess := Max(0, IntLog2(divisor) − 7)  
 recipDivisor[idx] := RoundFz(Exp2(16) ÷ idx) − 1

### Morton

The function 𝑚 = Morton( 𝑠, 𝑡, 𝑣 ) converts its parameters to a 3D Morton code. Its:

* parameters 𝑠, 𝑡 and 𝑣 are non-negative integers;
* result is specified by the value of the expression morton.

The conversion interleaves the bits of each parameter 𝑣, 𝑡 and 𝑠; in that order, starting from the LSBs. The LSB of 𝑣 is the LSB of 𝑚. Table 3 illustrates the construction of 3D Morton codes from the bit string representation of the parameters 𝑠, 𝑡 and 𝑣.

The expression Morton[ expr ] performs the same conversion for an expression expr that takes an argument 𝑘, 𝑘 ∈ { 0, 1, 2 }.

Morton[expr] := Morton(expr[0], expr[1], expr[2])

Table 3 — Construction of 3D Morton codes 𝑚 from the tuple ( 𝑠, 𝑡, 𝑣 )

| Bit string form | | | | Decimal form |
| --- | --- | --- | --- | --- |
| 𝑠 | 𝑡 | 𝑣 | 𝑚 | 𝑚 |
| '0 0' | '0 0' | '0 0' | '0 0 0  0 0 0' | 0 |
| '0 0' | '0 0' | '0 1' | '0 0 0  0 0 1' | 1 |
| '0 1' | '1 1' | '1 0' | '0 1 1  1 1 0' | 30 |
| '0 1' | '1 1' | '1 1' | '0 1 1  1 1 1' | 31 |
| '1 0' | '0 1' | '1 0' | '1 0 1  0 1 0' | 42 |
| '1 0' | '0 1' | '1 1' | '1 0 1  0 1 1' | 43 |
| '1 1' | '1 0' | '0 0' | '1 1 0  1 0 0' | 52 |
| '1 1' | '1 0' | '0 1' | '1 1 0  1 0 1' | 53 |
| … | … | … | … | … |

### FromMorton

The function ( 𝑠, 𝑡, 𝑣 ) = FromMorton( 𝑚 ) is the inverse of Morton( 𝑠, 𝑡, 𝑣 ). Its:

* parameter 𝑚 is a non-negative, integer, 3D Morton code;
* result is the tuple specified by the value of the expressions 𝑠, 𝑡 and 𝑣.

The conversion deinterleaves the bits of 𝑣, 𝑡 and 𝑠; in that order, starting from the LSB. The LSB of 𝑚 is the LSB of 𝑣.

## Order of operation precedence

When order of precedence in an expression is not indicated explicitly by round brackets, the following rules apply:

* Operations of a higher precedence are evaluated before any operation of a lower precedence.
* Operations of the same precedence are evaluated sequentially from left to right.

Table 4 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

1. For those operators that are also used in the C programming language, the order of precedence used in this document is the same as used in the C programming language.

Table 4 — Operation precedence from highest (at top of table) to lowest (at bottom of table)

| Operations (with operands 𝑥, 𝑦 and 𝑧) |
| --- |
| 𝑥++, 𝑥−− |
| ¬𝑥, −𝑥 (as a unary prefix operator) |
|  |
| 𝑥 × 𝑦, 𝑥 / 𝑦, 𝑥 ÷ 𝑦, , 𝑥 % 𝑦 |
| 𝑥 + 𝑦, 𝑥 − 𝑦 (as a two-argument operator), |
| 𝑥 << 𝑦, 𝑥 >> 𝑦 |
| 𝑥 < 𝑦, 𝑥 ≤ 𝑦, 𝑥 > 𝑦, 𝑥 ≥ 𝑦 |
| 𝑥 == 𝑦, 𝑥 ≠ 𝑦 |
| 𝑥 & 𝑦 |
| 𝑥 ^ 𝑦 |
| 𝑥 | 𝑦 |
| 𝑥 && 𝑦 |
| 𝑥 || 𝑦 |
| 𝑥 ? 𝑦 : 𝑧 |
| 𝑥 .. 𝑦 |
| 𝑥 = 𝑦, 𝑥 += 𝑦, 𝑥 −= 𝑦, 𝑥 ×= 𝑦, 𝑥 <<= 𝑦, 𝑥 >>= 𝑦 |

## Named expressions

### General

Operations and values in this document are sometimes specified in the form of named expressions. Exemplar named expressions are described in Table 5.

A named expression is a named macro-like statement. Every occurrence of a named expression is substituted by its definition when evaluated. The definition is provided by the ≔ operator.

Substitution is atomic. For example, the substitution for 3 × ExAPlusB is equivalent to 3 × ( 𝑎 + 𝑏 ), not ( 3 × 𝑎 ) + 𝑏.

The definition for a named expression is immutable. For example, ExTwo is equivalent to the value 2; unlike a variable it cannot be modified. All instances of ExTwo could be substituted by the numeric value 2.

The substitution for a named expression can be a variable. The substituted variable in such cases can be modified. For example, ExVar++ increments the variable Var.

Table 5 — Examples of named expressions

| Example | Remarks |
| --- | --- |
| ExTwo := 2 | ExTwo is equivalent to the value 2 |
| ExAPlusB := a + b | ExAPlusB is equivalent to ( 𝑎 + 𝑏 ) |
| ExTwoIndirect := ExTwo | ExTwoIndirect is equivalent to ExTwo |
| Var = 2  ExVar := Var | ExVar is equivalent to (an alias of) the variable Var |
| ExTimesTwo[i] := 2 × i | ExTimesTwo[ 𝑗 + 1 ] is equivalent to 2 × ( 𝑗 + 1 ) |
| ExSquared[i] := i × i | ExSquared[ ExVar++ ] is equivalent to Var × Var, with Var incremented after the evaluation of ExSquared |
| for (Var = 0; Var ≤ 10; Var++)  sum += ExTimesTwo[Var] | sum is incremented, in total, by 110 |
| ExWhere[i] := ExTimesTwo[inner]  where  inner := i + 1 | ExWhere[ 𝑗 ] is equivalent to 2 × ( 𝑗 + 1 ) |
| ExSumA[i] := i > 0  ? i + ExSumA[i − 1]  : 0 | Recursive definition.  ExSumA[ 10 ] evaluates to 55 |
| ExSumB[i] :=  ExSumB = 0  for (; i > 0; i−−)  ExSumB += i | Imperative definition using multiple statements.  ExSumB[ 10 ] evaluates to 55 |
| ExSum10[expr] :=  ExSum10 = 0  for (i = 0; i ≤ 10; i++)  ExSum10 += expr[i] | ExSum10 applies expr to each 𝑖, 𝑖 ∈ 0 .. 10, summing the result.  ExSum10[ ExprTimesTwo ] evaluates to 110  ExSum10[ ExprSquared ] evaluates to 385 |

### Scope

The scope of a named expression is not affected by the relative order of its definition and use; a named expression can be referenced earlier in the document than its definition.

Named expressions identified by a capital initial are "global" definitions that apply to the whole document. They may be directly referenced in other subclauses.

Named expressions identified by a lower-case initial are "local" definitions that apply to the subclause in which they are defined.

If a global definition references a local definition in the same subclause, that local definition is used when the global definition is referenced in another subclause.

### Arguments of named expressions

The definition of an expression can be in terms of one or more parameters. Each parameter is enclosed in square brackets. For example, the definition ExTimesTwo[ 𝑖 ] has a single parameter 𝑖.

A named expression can be applied to one or more arguments. When the definition is substituted for a named expression, every instance of each parameter is replaced by the text of the corresponding argument.

Replacements are atomic. For example, ExTimesTwo[ 𝑗 + 1 ] is equivalent to 2 × ( 𝑗 + 1 ), not ( 2 × 𝑗 ) + 1.

### Sub-expressions

A definition can contain a where-clause that defines further named expressions. They apply only to the definition containing the where-clause. For example, the definition of ExWhere[ 𝑖 ] defines the sub-expression inner.

### Definitions with multiple statements

Some definitions cannot be succinctly expressed by a single statement. In such cases, a definition can consist of multiple statements. The evaluated value for the whole definition is specified by assignments or modifications to a variable with the same name as the named expression. For example, ExSumB.

### Textual definitions

Some definitions are provided by a descriptive equivalence in textual or tabular form. For example:

* "The expression Ex[ 𝑖 ] is specified by Table X (Value for Ex[ 𝑖 ])."
* "The value for the expression Ex is specified by Table X for each axis 𝑘."
* "The expression Ex is equivalent to the following [procedural code]."

## Variables, syntax elements and tables

Syntax elements in the bitstream are represented in bold type. Each syntax element is described by its name (all lower-case letters with underscore characters) and one descriptor for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e. not bold) type.

In some cases the syntax tables use the values of variables derived from other syntax elements' values. Such variables appear in the syntax tables, or text, named by a mixture of lower- and upper-case letters and without any underscore characters. Variables with a capital initial are valid for the decoding of the current syntax structure and all dependent syntax structures. They may be used in the decoding process for later syntax structures without mentioning their origin. Variables with a lower-case initial are only used within the clause in which they are derived.

1. The syntax is described in a manner that closely follows the C language syntactic constructs.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions (7.2). These functions assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round brackets including zero or more parameter names (for definition) or arguments (for usage), separated by commas (if more than one).

Functions that are not syntax functions (including mathematical functions specified in 5.10) are described by their names, which start with a capital initial, contain a mixture of lower- and upper-case letters without any underscore characters and end with left and right round brackets surrounding zero or more parameter names (for definition) or arguments (for usage), separated by commas (if more than one).

Arrays are sequences of values identified by a common name. Both syntax elements and variables can be arrays. Subscripts or square brackets are used to index an array.

Boolean true and false values are interchangeable with the integers 1 and 0, respectively; non-zero integers are equivalent to true.

# Point cloud format and relationship to coded and output representations

## General format

A point cloud is an unordered list of points representing geometry, optional attribute information and associated metadata. Geometry information describes the location of points in a three-dimensional Cartesian coordinate system. Attributes are typed properties of each point, such as its colour or reflectance. Metadata is information used to interpret the point cloud, the point geometry and the attribute data.

Each point in a point cloud is a tuple of a three-dimensional position and attribute values for every attribute present in the point cloud. All points shall have the same number of attributes in the same order.

Point cloud metadata may describe, for example, a geometric transformation used to map points to another coordinate system, spatial regions (tiles) within a point cloud, the identification of attribute types and how attribute values are interpreted.

An *N*-point point cloud is symbolically illustrated in Figure 1. Rows of are points. Each point comprises a position with components and values for the components of each attribute of to .

文本

中度可信度描述已自动生成

Figure 1 — A symbolic representation of a point cloud.

Two point clouds are identical if there exists a one-to-one mapping between points in the two point clouds. For example, a permutation of points (rows in ) preserves identity.

## Attributes

### General

An attribute comprises one or more components.

A point cloud, unless otherwise specified, may contain more than one instance of a particular attribute. The significance or interpretation of multiple instances of the same attribute is unspecified.

Metadata can be associated with each attribute instance. Attribute metadata conveys sequence level characteristics such as an attribute label identifier or a frame level interpretation such as an attribute scale and offset information.

### Colour

The colour attribute specifies the colour of a point. The attribute shall comprise one of the following configurations:

* A luma () component (monochrome).
* A luma component and two chroma components ( or ).
* Green, blue and red components (, also known as *RGB*).
* Other unspecified monochrome or tri-stimulus colour systems (e.g., *YZX*, also known as *XYZ*).

The actual colour representation method in use is described using ISO/IEC 23091‑2 coding-independent video code points and is indicated using syntax specified in 7.3.2.7.

The ordering of attribute components is specified by Table 6.

Table 6 — Relationship between colour components and attribute components

| Colour representation | Attribute component index | | |
| --- | --- | --- | --- |
| 0 | 1 | 2 |
| Monochrome |  | – | – |
|  |  |  |  |
|  |  |  |  |
| or RGB |  |  |  |
| YZX or XYZ | *Y* | *Z* | *X* |

### Opacity

Opacity is a single component attribute. When normalized to the interval [ 0, 1 ], the value 0 indicates that a point is completely transparent and the value 1 indicates complete opacity. The opacity attribute may be used to control colour blending when rendering a colour attribute.

1. Opacity is often called an alpha channel or transparency.

### Reflectance

Reflectance is a single component attribute that represents the ratio of incident light reflected by a point; it is a dimensionless quantity. Values are bounded by a minimum that indicates complete absorption and a maximum that indicates complete reflection or saturation.

### Normal vector

A normal vector is a three-component attribute representing a vector perpendicular to the surface tangent plane at an associated point. The axes identification of the normal vectors is identical to that of the STV axes for the coded point cloud geometry. The length of a normal vector is not required to be one.

Normal vectors may be used when rendering a point cloud. A point's appearance may be modified according to the difference between the incident light direction and its normal vector.

### Material identifier

A material identifier is a single component attribute that associates a point with a material from a range of materials. Points with a common material identifier share a characteristic that may be used to identify an object or type of object. Materials are not specified by this document.

### Frame number/index

The frame number and frame index attributes are single component attributes that indicate how a point cloud frame may be partitioned into one or more ordered sub-frames. Each sub-frame is a partial representation of a point cloud frame, comprising points with the same frame number/index attribute value.

1. Sub-frame partitioning does not form part of the decoding or output processes specified by this document.

A point cloud sequence shall contain no more than one instance of a frame number/index attribute. A point cloud sequence shall not contain both frame number and frame index attributes.

The frame number attribute may be used to order all sub-frames over the entire point cloud sequence. Points from different point cloud frames shall not have the same value for the frame number attribute.

The frame index attribute may be used to order the sub-frames of a single point cloud frame.

An example of the relationship between frames, sub-frames and their ordering is shown in Table 7. The point cloud frames *a*, *b* and *c* are partitioned into sub-frames. Sub-frame orders are shown for the cases where the attribute is a frame number or a frame index.

Table 7 — Example partitioning of three consecutive frames a, b and c into sub-frames

|  | Frame | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | | | b | | c | | |
| **Frame number attribute** | | | | | | | | |
| Sub-frame attribute value | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 7 |
| Sub-frame presentation order | *a*0 | *a*1 | *a*2 | *b*3 | *c*4 | *b*5 | *c*6 | *c*7 |
| **Frame index attribute** | | | | | | | | |
| Sub-frame attribute value | 0 | 2 | 1 | 0 | 1 | 0 | 1 | 3 |
| Sub-frame presentation order | *a*0 | *a*1 | *a*2 | *b*0 | *b*1 | *c*0 | *c*1 | *c*3 |

### User defined attributes

The point cloud format supports attributes other than those specified in this document. A user defined attribute shall be identified by an international object identifier. The international object identifier shall either be assigned by a registration authority in accordance with Rec. ITU‑T X.660﻿ |​ ISO/IEC 9834‑1, or generated without registration using a universally unique identifier (UUID) as specified by Rec. ITU‑T X.667﻿ |​ ISO/IEC 9834‑8.

## Codec-derived attributes

### General

Codec-derived attributes represent values that are determined as side-effects of a processes specified in this document.

A decoder may, but is not required to, output one or more codec-derived attributes. Any codec-derived attributes output by a decoder shall conform to the definitions in 6.3.

### Slice identifier

The slice identifier attribute shall be a single component attribute that identifies the slice from which a point is decoded. Identification shall use the slice\_id syntax element value.

### Slice tag

The slice tag attribute shall be a single component attribute that identifies the group of slices from which a point is decoded. Identification shall use the slice\_tag syntax element value.

### Canonical point order

The canonical point order attribute shall be a single component attribute that specifies the order within a slice in which points are decoded by the geometry decoder as specified in this document.

Values of the point decoding order attribute shall be equal to ptIdx of the corresponding point PointPos[ ptIdx ] in a slice.

### Point Morton order

The point Morton order attribute shall be a single component attribute that specifies the order of points within a slice according to ascending values of Morton-coded STV slice position (i.e. prior to 8.3.6).

The Morton order shall be equivalent to the order of points in the finest detail level specified in 10.6.5.2 as if both attr\_canonical\_order\_enabled and attr\_coord\_conv\_enabled are both 0.

For example, if three points 𝑎, 𝑏 and 𝑐 in canonical point order are ordered { 𝑎, 𝑐, 𝑏 } in the finest detail level, then the respective values for the Morton order attribute are 0, 2 and 1.

## Coded point cloud format

### Sequence coordinate system

The sequence coordinate system is specified by the position of its origin in an externally defined application-specific coordinate system and by the length of its unit vectors.

All points in a coded point cloud sequence shall have non-negative coordinates in the sequence coordinate system.

A position in the sequence coordinate system is related to the position in the application coordinate system by the sequence origin SeqOrigin and the unit vector length SeqUnit specified by the active SPS:

The maximum bound on the sequence coordinate system depends upon the level to which the coded point cloud sequence conforms, as specified in Annex A.

An example sequence coordinate system (marked SCS) is illustrated in Figure 2. A point with an 𝑥-coordinate of 75 in the sequence coordinate system has a position in the application-specific coordinate system of 105 SeqUnit. If SeqUnit is 0,8 AppUnit, the 𝑥-coordinate of the point in the application-specific coordinate system (marked ACS) is 4,2.

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| ACS | Application-specific coordinate system |
| CCS | Coding coordinate system |
| SCS | Sequence coordinate system |
| 𝑎:𝑏 | Ratio of SCS to CCS |

Figure 2 — Relationship between application, sequence, and coding coordinate systems.

### Coding coordinate system

The coding coordinate system is a non-negative integer coordinate system used to code point positions. It is either identical to, or a geometric contraction of the sequence coordinate system. Its origin is coincident with the sequence coordinate system origin.

A position in the coding coordinate system is related to the position in the sequence coordinate system by the binary fixed-point scale factor SeqCodedScale:

An example coding coordinate system (marked CCS) is illustrated in Figure 2. A point with an 𝑥-coordinate of 48 in the coding coordinate system has an 𝑥-coordinate, , of 75 when scaled by the scale factor .

Point position components in the coding coordinate system shall satisfy the following level dependent (Annex A) constraint:

Where SeqCodedScaleN and SeqCodedScaleD are the numerator and denominator, respectively, of SeqCodedScale when represented as an irreducible fraction:

1. This constraint guarantees that conversion from the coding coordinate system to the sequence coordinate system can be performed using 32-bit arithmetic.

### Coded point cloud sequence

The coded representation of a point cloud sequence comprises one or more point cloud frames encoded as a sequence of encapsulated DUs that convey syntax structures as specified in 7.3. An encapsulation format is specified by Annex B. Alternative encapsulation formats may be specified by the application.

The coded point cloud sequence shall include:

* An SPS that enumerates the attributes present in the coded point cloud format and conveys both metadata and decoding parameters that pertain to the whole coded point cloud sequence.
* Any GPSs that convey parameters required for the decoding of geometry data.
* Any APSs that convey parameters required for the decoding of attribute data.
* The slices comprising each coded point cloud frame.

### Coded point cloud frame

A coded point cloud frame comprises a sequence of zero or more slices with the same value of a notional frame counter FrameCtr (8.2.2). An optional frame boundary marker data unit explicitly signals the end of a frame.

It is a requirement of bitstream conformance that:

* Every coded point cloud frame shall have a unique value of FrameCtr within the sequence.
* Coded point cloud frames shall be ordered such that the notional frame counter increases for each successive coded point cloud frame when biprediction\_enabled is equal to 0.
* The value of of gdu\_temporal\_id of each GDU in the frame and adu\_temporal\_id of each ADU in the frame is equal to the same value.
* The decoding of current frame shall not depend on any point cloud frame F in the bitstream where the temporal ID of frame F is greater than the temporal ID of the current frame.

An empty frame shall be signalled by a frame boundary marker data unit without any preceding slices with the same value of FrameCtr.

A coded point cloud frame independently codes a single point cloud frame without dependencies upon any previous or subsequent point cloud frame.

A decoded point cloud frame is the concatenation of all points in all constituent slices of the frame.

Unless prohibited by an SPS constraint, coincident points in a point cloud frame may arise from:

* points coded in a single slice with a non-zero duplicate point count;
* distinct points with the same position in a single slice; or
* the concatenation of multiple slices.

### Slice of a coded point cloud frame

Every slice shall include a GDU that codes the slice geometry and ADUs or defaulted attribute DUs that code the slice attributes. A slice is identified by the GDU slice\_id.

The slice geometry is coded in the slice's coordinate system. The bounding boxes of slices may intersect, including within a single frame.

A slice shall start with a GDU. This GDU may be followed by optional redundant GDUs that duplicate the slice geometry. ADUs and defaulted attribute DUs shall occur after all GDUs in the slice. DUs belonging to different slices shall not be interleaved.

Within a slice, other DUs may be present. For example, an APS can occur within a slice to convey parameters for attribute decoding.

It is a requirement of bitstream conformance that:

* All GDUs present in a slice shall reconstruct the same geometry in the same canonical point order.
* Every slice shall have a corresponding ADU or defaulted attribute DU for every attribute enumerated in the SPS.
* All ADUs present in a slice with the same value of adu\_sps\_attr\_idx shall reconstruct the same attribute values.

Only one GDU in a slice shall be decoded; all others shall be ignored when decoding (removed from the bitstream and discarded). A decoder shall choose which GDU is decoded.

ADU parsing depends upon certain GDU header parameters. ADU decoding depends upon the reconstructed slice geometry.

Slices are either independent or dependent. An independent slice does not require any other slice to be decoded first. A dependent slice requires that the immediately preceding slice in bitstream order is decoded first. A slice shall be directly depended upon by no more than a single dependent slice.

A dependent slice shall not depend upon a slice in a different point cloud frame.

### Repetition of slices

Slices may be repeated within a coded point cloud frame. Repetition shall not change the value of slice\_id.

A slice set is the set of slices with the same value of slice\_id within a coded point cloud frame.

It is a requirement of bitstream conformance that all slices in each slice set shall reconstruct the same points in the same canonical order.

From each slice set, only one slice shall be decoded; all others shall be ignored for decoding (removed from the bitstream and discarded). A decoder shall choose which slice is decoded.

### Relationship between tiles and slices

A group of slices can be identified by a common slice tag identifier (slice\_tag).

The tile inventory DU provides a means to associate a bounding box with a group of slices. Each tile comprises a single bounding box and an identifier (tileId). Tile bounding boxes may overlap. Implementations can use a tile inventory to aid spatial random access.

When a tile inventory is present in the bitstream, slice\_tag shall identify a tile by its tileId. Otherwise, the use of slice tags is application specific.

When a slice tag identifies a tile, a dependent slice should not depend upon a slice in a different tile. To do otherwise can prevent decoding of individual tiles (for example, in spatial random access decoding).

Tile information is not used by the decoding processes specified in this document.

图示, 工程绘图

描述已自动生成

Key

|  |  |
| --- | --- |
|  | Slice 𝑛, associated with tile 𝑡 |
|  | Bounding box of tile 𝑡 |

Figure 3 — Example arrangement of tiles and slices.

An example arrangement of tiles and slices within a coded point cloud frame is shown in Figure 3. Slices and are associated with tile and slices , and are associated with tile ; the bounding box of does not include . A decoder that performs spatial random access to decode a region (not shown) can use the tile inventory to determine tile IDs for the set of tiles that intersect . Only slices with matching slice tags would need to be decoded. Since the slice is not included in the bounding box of tile , if intersects but not , the slice is not discoverable using the tile inventory. However, in the case that and intersect, would have a matching slice tag.

### Parameter sets

#### Activation of parameter sets

The parameters contained in an SPS, GPS or APS shall not have any effect until the activation of the respective parameter set.

At most one SPS, GPS and APS are active at any given moment during the decoding process. The activation of a parameter-set shall deactivate any previously active parameter set of the same type.

At the start of a coded point cloud sequence, no parameter sets are active.

An SPS shall be activated by the parsing of a GDU. Once activated, it shall remain active for the whole of the coded point cloud sequence.

A GPS shall be activated by the parsing of a GDU.

An APS shall be activated by the parsing of an ADU.

1. Other DUs that contain references to SPS, GPS or APS DUs do not cause the referenced parameter-set to be activated.

#### Order of parameter sets

DUs shall be conveyed to a decoder in an order such that any parameter-set to be activated is available prior to the point of activation.

#### Duplication of parameter sets

Parameter-set DUs may be repeated at any point in the coded point cloud sequence.

All parameter-set DUs with the same parameter-set identifier shall be identical for the duration of the coded point cloud sequence.

1. Parameter-set identifiers are distinct for each type of parameter set.

## Output point cloud format

### General

Point cloud frames decoded from a G-PCC bitstream shall be output in the output point cloud format (6.5).

### Coordinate system

A decoder shall output points in the sequence coordinate system.

The output point cloud format shall indicate the sequence origin SeqOrigin and the sequence unit SeqUnit as point cloud metadata.

### Fixed-point conformance output

A decoder that is configured to output 𝑛-fractional-bit fixed-point positions shall round half-values of away from zero prior to output as :

### Attributes

Attribute values shall be interpreted according to the semantics of the attribute type and any per-sequence or frame-specific attribute properties. For example, if a frame-specific scale and offset property is present for an attribute, the output attribute values for that frame would be interpreted according to 7.4.2.2.5.

### Output point cloud sequence

Decoding a conforming G-PCC bitstream generates a sequence of output point cloud frames. Output point clouds frames are output in the order of FrameCtr of each point cloud frame within the sequence.

### Output point cloud frame

Each output point cloud frame is specified in terms of the following state variables:

* The variable RecCloudPointCnt, the cumulative number of points in the output point cloud frame.
* The array RecCloudPos of decoded point positions; RecCloudPos[ ptIdx ][ 𝑘 ] is the 𝑘-th coordinate of the ptIdx-th output point in the coding coordinate system.
* The array RecCloudAttr of decoded point attributes; RecCloudAttr[ ptIdx ][ attrIdx ][ 𝑐 ] is the 𝑐-th component of the identified attribute for the ptIdx-th point. Attributes are identified by the index attrIdx into the active SPS attribute list.

Decoder implementations may output points in a different order to the canonical order specified by this document.

Immediately prior to outputting the decoded point cloud frame, point positions shall be converted to the sequence coordinate system.

Each decoded point cloud frame shall be stored in buffer. One decoded point cloud frame stored in buffer shall be output when it is the first decoded point cloud frame or the FrameCtr of it is equal to the FrameCtr of the previously one output point cloud frame plus 1. The buffer for one decoded point cloud frame shall be released when the point cloud frame does not serve as the reference frame for any other point cloud frames to be decoded.

# Syntax and semantics

## Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed bitstreams. Additional constraints on the syntax may be specified, either directly or indirectly, in other clauses.

The following table lists examples of pseudo code used to describe the syntax. When syntax\_element appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

|  |  |
| --- | --- |
|  | Descriptor |
| /\* A statement can be a syntax element with an associated descriptor or can be a statement used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples \*/ |  |
| syntax\_element | ue(v) |
| statement |  |
|  |  |
| /\* A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. \*/ |  |
| { |  |
| statement |  |
| statement |  |
| … |  |
| } |  |
|  |  |
| /\* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true \*/ |  |
| while( condition ) |  |
| statement |  |
|  |  |
| /\* A "do … while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true \*/ |  |
| do { |  |
| statement |  |
| } while( condition ) |  |
|  |  |
| /\* An "if … else" structure specifies a test of whether a condition is true, and if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed \*/ |  |
| if( condition ) |  |
| primary statement |  |
| else |  |
| alternative statement |  |
|  |  |
| /\* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. \*/ |  |
| for( initial statement; condition; subsequent statement ) |  |
| primary statement |  |

## Specification of syntax functions and descriptors

The functions presented here are used in the syntactical description. These functions are expressed in terms of the value of the bitstream pointer DataUnitReadIdx that indicates the position of the next bit to be read from the bitstream by the decoding process.

byte\_aligned( ) is specified as:

* If the next bit in the bitstream is the first bit in a byte (DataUnitReadIdx % 8 == 0), the value of byte\_aligned( ) is true.
* Otherwise, the value of byte\_aligned( ) is false.

more\_data\_in\_data\_unit( ) is specified as:

* If parsing of the DU is incomplete (DataUnitReadIdx / 8 < DataUnitLength), the value of more\_data\_in\_data\_unit( ) is true.
* Otherwise, the value of more\_data\_in\_data\_unit( ) is false.

Length( 𝑥 ) is the length in bits of the coded syntax element 𝑥 as measured by the change in DataUnitReadIdx between the start and end of the syntax element.

The following descriptors specify the parsing process of every syntax element. The parsing processes are specified in Clause 11.

* ae(v): adaptive arithmetic entropy-coded syntax element.
* de(v): dictionary coded syntax element.
* oid(v): an ASN.1 object identifier.
* s(𝑛): signed integer using an 𝑛-bit magnitude and a sign bit.
* se(v): signed integer 0-th order Exp-Golomb-coded syntax element.
* u(𝑛): unsigned integer using 𝑛 bits. When 𝑛 is "v" in the syntax table, the number of bits varies in a manner dependent upon the value of other syntax elements.
* ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element.

## Syntax in tabular form

### General

The syntax structures and the syntax elements within these structures are specified in 7.3.2. Any values that are not specified in the tables shall not be present in the bitstream unless otherwise specified in this document.

### Parameter sets, ancillary data and byte alignment

#### Sequence parameter set data unit syntax

|  |  |
| --- | --- |
| seq\_parameter\_set( ) { | Descriptor |
| simple\_profile\_compliant | u(1) |
| dense\_profile\_compliant | u(1) |
| predictive\_profile\_compliant | u(1) |
| main\_profile\_compliant | u(1) |
| reserved\_profile\_18bits | u(18) |
| slice\_reordering\_constraint | u(1) |
| unique\_point\_positions\_constraint | u(1) |
| level\_idc | u(8) |
| sps\_seq\_parameter\_set\_id | u(4) |
| frame\_ctr\_lsb\_bits | u(5) |
| slice\_tag\_bits | u(5) |
| seq\_origin\_bits | ue(v) |
| if( seq\_origin\_bits) { |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| seq\_origin\_xyz[ 𝑘 ] | s(v) |
| seq\_origin\_log2\_scale | ue(v) |
| } |  |
| seq\_bbox\_size\_bits | ue(v) |
| if( seq\_bbox\_size\_bits ) |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| seq\_bbox\_size\_minus1\_xyz[ 𝑘 ] | u(v) |
| seq\_unit\_numerator\_minus1 | ue(v) |
| seq\_unit\_denominator\_minus1 | ue(v) |
| seq\_unit\_is\_metres | u(1) |
| seq\_coded\_scale\_exponent | ue(v) |
| seq\_coded\_scale\_mantissa\_bits | ue(v) |
| seq\_coded\_scale\_mantissa | u(v) |
| num\_attributes | ue(v) |
| for( attrIdx = 0; attrIdx < num\_attributes; attrIdx++ ) { |  |
| attr\_components\_minus1[ attrIdx ] | ue(v) |
| attr\_instance\_id[ attrIdx ] | ue(v) |
| attr\_bitdepth\_minus1[ attrIdx ] | ue(v) |
| attr\_label\_known[ attrIdx ] | u(1) |
| if( attr\_label\_known[ attrIdx ] ) |  |
| attr\_label[ attrIdx ] | ue(v) |
| else |  |
| attr\_label\_oid[ attrIdx ] | oid(v) |
| attr\_property\_cnt | ue(v) |
| byte\_alignment( ) |  |
| for( 𝑗 = 0; 𝑗 < attr\_property\_cnt; 𝑗++ ) |  |
| attribute\_property(attrIdx ) |  |
| } |  |
| geom\_axis\_order | u(3) |
| bypass\_stream\_enabled | u(1) |
| entropy\_continuation\_enabled | u(1) |
| sps\_extension\_present | u(1) |
| if( sps\_extension\_present ) { |  |
| if(num\_attributes > 1) |  |
| cross\_attr\_prediction\_enabled | u(1) |
| bypass\_bin\_coding\_prob\_update\_disabled | u(1) |
| inter\_frame\_prediction\_enabled | u(1) |
| if(inter\_frame\_prediction\_enabled) |  |
| inter\_entropy\_continuation\_enabled | u(1) |
| fgs\_layer\_group\_enabled | u(1) |
| while( more\_data\_in\_data\_unit( ) ) |  |
| sps\_extension\_data | u(1) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute property syntax

|  |  |
| --- | --- |
| attribute\_property( attrIdx ) { | Descriptor |
| attr\_prop\_type | u(8) |
| attr\_prop\_len | u(8) |
| AttrPropDataLen = attr\_prop\_len |  |
| if( attr\_prop\_type == 0 ) { |  |
| attr\_prop\_itu\_t\_t35\_country\_code | u(8) |
| AttrPropDataLen−− |  |
| if( attr\_prop\_itu\_t\_t35\_country\_code == 255 ) { |  |
| attr\_prop\_itu\_t\_t35\_country\_code\_extension\_byte | u(8) |
| AttrPropDataLen−− |  |
| } |  |
| attribute\_property\_data( attrIdx, AttrPropDataLen ) |  |
| } else if( attr\_prop\_type == 1 ) { |  |
| attr\_prop\_oid | oid(v) |
| AttrPropDataLen −= Length( attr\_prop\_oid ) / 8 |  |
| attribute\_property\_data( attrIdx, AttrPropDataLen ) |  |
| } else if( attr\_prop\_type == 2 ) { |  |
| attr\_cicp\_colour\_primaries[ attrIdx ] | ue(v) |
| attr\_cicp\_transfer\_characteristics[ attrIdx ] | ue(v) |
| attr\_cicp\_matrix\_coeffs[ attrIdx ] | ue(v) |
| attr\_cicp\_video\_full\_range[ attrIdx ] | u(1) |
| } else if( attr\_prop\_type == 3 ) { |  |
| attr\_offset\_bits | ue(v) |
| attr\_offset[ attrIdx ] | s(v) |
| attr\_scale\_bits | ue(v) |
| attr\_scale\_minus1[ attrIdx ] | u(v) |
| attr\_frac\_bits[ attrIdx ] | ue(v) |
| } else if( attr\_prop\_type == 4 ) { |  |
| for( 𝑐 = 0; 𝑐 ≤ attr\_components\_minus1[ attrIdx ]; 𝑐++ ) |  |
| attr\_default\_value[ attrIdx ][ 𝑐 ] | u(v) |
| } else |  |
| attribute\_property\_data( attrIdx, attr\_prop\_len ) |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute property data syntax

|  |  |
| --- | --- |
| attribute\_property\_data( attrIdx, numBytes ) { | Descriptor |
| for( 𝑖 = 0; 𝑖 < numBytes; 𝑖++) |  |
| attr\_prop\_byte[ 𝑖 ] | u(8) |
| } |  |

#### Tile inventory data unit syntax

|  |  |
| --- | --- |
| tile\_inventory( ) { | Descriptor |
| ti\_seq\_parameter\_set\_id | u(4) |
| ti\_frame\_ctr\_lsb\_bits | u(5) |
| ti\_frame\_ctr\_lsb | u(v) |
| tile\_cnt | u(16) |
| if( tile\_cnt > 0 ) { |  |
| tile\_id\_bits | u(5) |
| tile\_origin\_bits\_minus1 | u(8) |
| tile\_size\_bits\_minus1 | u(8) |
| for( tileIdx = 0; tileIdx < tile\_cnt; tileIdx++ ) { |  |
| tile\_id[ tileIdx ] | u(v) |
| tileId = tile\_id\_bits ? tile\_id[ tileIdx ] : tileIdx |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| tile\_origin\_xyz[ tileId ][ 𝑘 ] | s(v) |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| tile\_size\_minus1\_xyz[ tileId ][ 𝑘 ] | u(v) |
| } |  |
| ti\_origin\_bits\_minus1 | ue(v) |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| ti\_origin\_xyz[ 𝑘 ] | s(v) |
| ti\_origin\_log2\_scale | ue(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Geometry parameter set data unit syntax

|  |  |
| --- | --- |
| geometry\_parameter\_set( ) { | Descriptor |
| gps\_geom\_parameter\_set\_id | u(4) |
| gps\_seq\_parameter\_set\_id | u(4) |
| slice\_geom\_origin\_scale\_present | u(1) |
| if( ¬slice\_geom\_origin\_scale\_present ) |  |
| gps\_geom\_origin\_log2\_scale | ue(v) |
| geom\_dup\_point\_counts\_enabled | u(1) |
| geom\_tree\_type | u(1) |
| if( geom\_tree\_type == 0 ) { |  |
| occtree\_point\_cnt\_list\_present | u(1) |
| occtree\_direct\_coding\_mode | u(2) |
| if( occtree\_direct\_coding\_mode ) |  |
| occtree\_direct\_joint\_coding\_enabled | u(1) |
| occtree\_coded\_axis\_list\_present | u(1) |
| occtree\_neigh\_window\_log2\_minus1 | u(3) |
| if( occtree\_neigh\_window\_log2\_minus1 > 0 ) { |  |
| occtree\_adjacent\_child\_enabled | u(1) |
| occtree\_intra\_pred\_max\_nodesize\_log2 | ue(v) |
| } |  |
| occtree\_bitwise\_coding | u(1) |
| occtree\_planar\_enabled | u(1) |
| if( occtree\_planar\_enabled ) { |  |
| for( 𝑖 = 0; 𝑖 < 3; 𝑖++) |  |
| occtree\_planar\_threshold[ 𝑖 ] | ue(v) |
| if( occtree\_direct\_coding\_mode == 1 ) |  |
| occtree\_direct\_node\_rate\_minus1 | u(5) |
| } |  |
| } |  |
| geom\_angular\_enabled | u(1) |
| if( geom\_angular\_enabled ) { |  |
| slice\_angular\_origin\_present | u(1) |
| if( ¬slice\_angular\_origin\_present ) { |  |
| gps\_angular\_origin\_bits\_minus1 | ue(v) |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| gps\_angular\_origin\_xyz[ 𝑘 ] | s(v) |
| } |  |
| if( geom\_tree\_type == 1 ) { |  |
| ptree\_ang\_azimuth\_pi\_bits\_minus11 | ue(v) |
| ptree\_ang\_azimuth\_step\_minus1 | ue(v) |
| ptree\_ang\_radius\_scale\_log2 | ue(v) |
| } |  |
| num\_beams\_minus1 | ue(v) |
| beam\_elevation\_init | se(v) |
| beam\_voffset\_init | se(v) |
| if( geom\_tree\_type == 0 ) |  |
| beam\_steps\_per\_rotation\_init\_minus1 | ue(v) |
| for( 𝑖 = 1; 𝑖 ≤ num\_beams\_minus1; 𝑖++ ) { |  |
| beam\_elevation\_diff[ 𝑖 ] | se(v) |
| beam\_voffset\_diff[ 𝑖 ] | se(v) |
| if( geom\_tree\_type == 0 ) |  |
| beam\_steps\_per\_rotation\_diff[ 𝑖 ] | se(v) |
| } |  |
| if( occtree\_planar\_enabled ) |  |
| occtree\_planar\_buffer\_disabled | u(1) |
| } |  |
| geom\_scaling\_enabled | u(1) |
| if( geom\_scaling\_enabled ) { |  |
| geom\_qp | ue(v) |
| geom\_qp\_mul\_log2 | u(2) |
| if( geom\_tree\_type == 1 ) |  |
| ptree\_qp\_period\_log2 | ue(v) |
| else if( occtree\_direct\_coding\_mode ) |  |
| occtree\_direct\_node\_qp\_offset | se(v) |
| } |  |
| gps\_extension\_present | u(1) |
| if( gps\_extension\_present ) { |  |
| if(geom\_tree\_type == 0) { |  |
| trisoup\_enabled | u(1) |
| if(trisoup\_enabled) { |  |
| trisoup\_non\_cubic\_node\_start\_edge\_present | u(1) |
| trisoup\_non\_cubic\_node\_end\_edge\_present | u(1) |
| } |  |
| } |  |
| if(geom\_tree\_type == 0 && geom\_angular\_enabled) |  |
| occtree\_angular\_extension\_enabled | u(1) |
| if(geom\_tree\_type == 0 || geom\_angular\_enabled) |  |
| inter\_prediction\_enabled | u(1) |
| if(inter\_prediction\_enabled) { |  |
| global\_motion\_enabled | u(1) |
| if(geom\_tree\_type == 1) { |  |
| inter\_azim\_scale\_log2 | ue(v) |
| resampling\_enabled | u(1) |
| max\_points\_per\_entry\_minus1 | ue(v) |
| if(max\_points\_per\_entry\_minus1 > 0) |  |
| down\_sampling\_range | ue(v) |
| } |  |
| biprediction\_enabled | ue(v) |
| if(biprediction\_enabled > 0) |  |
| frame\_merge\_enabled | u(1) |
| } |  |
| if( geom\_tree\_type == 0 && geom\_angular\_enabled && occtree\_direct\_coding\_mode && inter\_prediction\_enabled) |  |
| occtree\_inter\_angular\_direct\_coding\_enabled | u(1) |
| if(occtree\_planar\_enabled && geom\_angular\_enabled && occtree\_direct\_coding\_mode) |  |
| geo\_planar\_idcm\_angular\_disabled | u(1) |
| if( geom\_tree\_type == 1 && geom\_angular\_enabled) { |  |
| ptree\_sec\_resid\_disabled | u(1) |
| ptree\_ang\_azimuth\_scaling\_enabled | u(1) |
| if(ptree\_ang\_azimuth\_scaling\_enabled) { |  |
| ptree\_ang\_max\_pred\_index | ue(v) |
| ptree\_ang\_pred\_list\_radius\_resid\_threshold | ue(v) |
| ptree\_ang\_radius\_resid\_context\_qphi\_threshold\_present | u(1) |
| if(ptree\_ang\_radius\_resid\_context\_qphi\_threshold\_present) { |  |
| ptree\_ang\_radius\_resid\_context\_qphi\_threshold | ue(v) |
| } |  |
| } |  |
| } |  |
| if(occtree\_planar\_enabled && ¬geom\_angular\_enabled ) |  |
| octree\_planar\_neigh\_prediction\_enabled | u(1) |
| while( more\_data\_in\_data\_unit( ) ) |  |
| gps\_extension\_data | u(1) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute parameter set data unit syntax

|  |  |
| --- | --- |
| attribute\_parameter\_set( ) { | Descriptor |
| aps\_attr\_parameter\_set\_id | u(4) |
| aps\_seq\_parameter\_set\_id | u(4) |
| attr\_coding\_type | ue(v) |
| attr\_primary\_qp\_minus4 | ue(v) |
| attr\_secondary\_qp\_offset | se(v) |
| attr\_qp\_offsets\_present | u(1) |
| if( attr\_coding\_type == 0 ) { |  |
| raht\_prediction\_enabled | u(1) |
| if( raht\_prediction\_enabled ) { |  |
| raht\_prediction\_subtree\_min | ue(v) |
| raht\_prediction\_samples\_min | ue(v) |
| } |  |
| } else if( attr\_coding\_type ≤ 2 ) { |  |
| pred\_set\_size\_minus1 | ue(v) |
| pred\_inter\_lod\_search\_range | ue(v) |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |
| pred\_dist\_bias\_minus1\_xyz[ 𝑘 ] | ue(v) |
| if( attr\_coding\_type == 2 ) |  |
| last\_comp\_pred\_enabled | u(1) |
| lod\_scalability\_enabled | u(1) |
| if( lod\_scalability\_enabled ) |  |
| pred\_max\_range\_minus1 | ue(v) |
| else { |  |
| lod\_max\_levels\_minus1 | ue(v) |
| if( ¬lod\_max\_levels\_minus1 ) |  |
| attr\_canonical\_order\_enabled | u(1) |
| else { |  |
| lod\_decimation\_mode | ue(v) |
| if( lod\_decimation\_mode > 0 ) |  |
| for( lvl = 0; lvl < lod\_max\_levels\_minus1; lvl++ ) |  |
| lod\_sampling\_period\_minus2[ lvl ] | ue(v) |
| lod\_initial\_dist\_log2 | ue(v) |
| lod\_dist\_log2\_offset\_present | u(1) |
| } |  |
| } |  |
| if( attr\_coding\_type == 1 ) { |  |
| pred\_direct\_max\_idx\_plus1 | ue(v) |
| if( pred\_direct\_max\_idx\_plus1 ) { |  |
| pred\_direct\_threshold | u(8) |
| pred\_direct\_avg\_disabled | u(1) |
| } |  |
| pred\_intra\_lod\_search\_range | ue(v) |
| if( pred\_intra\_lod\_search\_range ) |  |
| pred\_intra\_min\_lod | ue(v) |
| inter\_comp\_pred\_enabled | u(1) |
| pred\_blending\_enabled | u(1) |
| } |  |
| } else if( attr\_coding\_type == 3 ) |  |
| raw\_attr\_width\_present | u(1) |
| if( ¬lod\_scalability\_enabled ) |  |
| attr\_coord\_conv\_enabled | u(1) |
| if( attr\_coord\_conv\_enabled ) |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) { |  |
| attr\_coord\_conv\_scale\_bits\_minus1[ 𝑘 ] | u(5) |
| attr\_coord\_conv\_scale[ 𝑘 ] | u(v) |
| } |  |
| aps\_extension\_present | u(1) |
| if( aps\_extension\_present ) { |  |
| if(cross\_attr\_prediction\_enabled) { |  |
| cross\_attr\_prediction\_enabled\_this\_type | u(1) |
| if( cross\_attr\_prediction\_enabled\_this\_type) |  |
| refAttrIdx | ue(v) |
| } |  |
| if( attr\_coding\_type == 0 ) { |  |
| lossless\_coding\_enabled | u(1) |
| raht\_last\_comp\_pred\_enabled | u(1) |
| } |  |
| if( attr\_coding\_type == 1) |  |
| for( *i* = 0;  *i* <= pred\_set\_size\_minus1;  *i*++ ) |  |
| quant\_neigh\_weight[ i ] | ue(v) |
| attr\_inter\_prediction\_enabled | u(1) |
| if( attr\_inter\_prediction\_enabled) |  |
| if( attr\_coding\_type == 0 ) { |  |
| raht\_inter\_layer\_depth\_minus1 | ue(v) |
| if(lossless\_coding\_enabled == 0 ) { |  |
| raht\_send\_inter\_filters | u(1) |
| raht\_inter\_skip\_layers | ue(v) |
| } |  |
| raht\_inter\_layer\_code\_enabled | u(1) |
| } |  |
| else |  |
| attr\_inter\_prediction\_search\_range | ue(v) |
| if( (attr\_coding\_type == 1 || attr\_coding\_type == 2) && ¬lod\_scalability\_enabled && ¬lod\_max\_levels\_minus1) |  |
| max\_points\_per\_sort\_log2\_plus1 | ue(v) |
| if( (attr\_coding\_type == 1 || attr\_coding\_type == 2) && pred\_set\_size\_minus1 >= 2) |  |
| prediction\_with\_distribution\_enabled | u(1) |
| if( attr\_coding\_type == 0 ) |  |
| raht\_buffer\_extension\_flag | u(1) |
| if( attr\_coding\_type == 0 && raht\_prediction\_enabled) { |  |
| raht\_subnode\_prediction\_enabled | u(1) |
| raht\_intra\_layer\_code\_enabled | u(1) |
| if( raht\_subnode\_prediction\_enabled) |  |
| for( *i* = 0;  *i* < 5;  *i* ++ ) |  |
| raht\_prediction\_weights[ i ] | ue(v) |
| raht\_prediction\_search\_range | ue(v) |
| } |  |
| if( (attr\_coding\_type == 1 || attr\_coding\_type == 2) && fgs\_layer\_group\_enabled) |  |
| fgs\_attr\_parameter( ) |  |
| while( more\_data\_in\_data\_unit( ) ) |  |
| aps\_extension\_data | u(1) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Frame-specific attribute properties data unit syntax

|  |  |
| --- | --- |
| frame\_specific\_attribute\_properties( ) { | Descriptor |
| fsap\_seq\_parameter\_set\_id | u(4) |
| fsap\_frame\_ctr\_lsb\_bits | u(5) |
| fsap\_frame\_ctr\_lsb | u(v) |
| fsap\_sps\_attr\_idx | ue(v) |
| fsap\_num\_props | ue(v) |
| byte\_alignment( ) |  |
| for( 𝑖 = 0; 𝑖 < fsap\_num\_props; 𝑖++ ) |  |
| attribute\_property( fsap\_sps\_attr\_idx ) |  |
| } |  |

#### Frame boundary marker data unit syntax

|  |  |
| --- | --- |
| frame\_boundary\_marker( ) { | Descriptor |
| fbdu\_frame\_ctr\_lsb\_bits | u(5) |
| fbdu\_frame\_ctr\_lsb | u(v) |
| byte\_alignment( ) |  |
| } |  |

#### User data data unit syntax

|  |  |
| --- | --- |
| userdata\_data\_unit( ) { | Descriptor |
| user\_data\_oid | oid(v) |
| while( more\_data\_in\_data\_unit( ) ) |  |
| user\_data\_byte | u(8) |
| } |  |

#### Byte alignment syntax

|  |  |
| --- | --- |
| byte\_alignment( ) { | Descriptor |
| while( ¬byte\_aligned( ) ) |  |
| alignment\_bit\_equal\_to\_zero /\* equal to 0 \*/ | u(1) |
| } |  |

### Geometry data unit

#### Geometry data unit syntax

|  |  |
| --- | --- |
| geometry\_data\_unit( ) { | Descriptor |
| geometry\_data\_unit\_header( ) |  |
| if( geom\_tree\_type == 0 ) { |  |
| if(fgs\_layergroup\_enabled) |  |
| fgs\_occupancy\_tree(startDepth, endDepth   ) |  |
| else |  |
| occupancy\_tree(occtreeMaxDepthMinus1   ) |  |
| if(trisoup\_enabled) |  |
| trisoup( ) |  |
| } else if( geom\_tree\_type == 1 ) |  |
| predictive\_tree( ) |  |
| geometry\_data\_unit\_footer(occtreeMaxDepthMinus1 ) |  |
| } |  |

#### Geometry data unit header syntax

|  |  |  |
| --- | --- | --- |
| geometry\_data\_unit\_header( ) { | Descriptor | Semantics |
| gdu\_geometry\_parameter\_set\_id | u(4) | 7.4.3.2 |
| gdu\_temporal\_id | u(3) | 7.4.3.2 |
| slice\_id | ue(v) | 7.4.3.2 |
| slice\_tag | u(v) | 7.4.3.2 |
| frame\_ctr\_lsb | u(v) | 7.4.3.2 |
| if( entropy\_continuation\_enabled ) { |  |  |
| slice\_entropy\_continuation | u(1) | 7.4.3.2 |
| if( slice\_entropy\_continuation ) |  |  |
| prev\_slice\_id | ue(v) | 7.4.3.2 |
| } |  |  |
| if( slice\_geom\_origin\_scale\_present ) |  |  |
| slice\_geom\_origin\_log2\_scale | ue(v) | 7.4.3.2 |
| slice\_geom\_origin\_bits\_minus1 | ue(v) | 7.4.3.2 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| slice\_geom\_origin\_xyz[ 𝑘 ] | u(v) | 7.4.3.2 |
| if( slice\_angular\_origin\_present ) { |  |  |
| slice\_angular\_origin\_bits\_minus1 | ue(v) | 7.4.3.2 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| slice\_angular\_origin\_xyz[ 𝑘 ] | s(v) | 7.4.3.2 |
| } |  |  |
| if( geom\_tree\_type == 0 ) { |  |  |
| occtree\_depth\_minus1 | ue(v) | 9.2.3 |
| if( occtree\_coded\_axis\_list\_present ) |  |  |
| for( dpth = 0; dpth ≤ occtreeMaxDepthMinus1; dpth++ ) |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| occtree\_coded\_axis[ dpth ][ 𝑘 ] | u(1) | 9.2.3 |
| occtree\_stream\_cnt\_minus1 | ue(v) | 9.2.3 |
| } |  |  |
| if( geom\_scaling\_enabled ) { |  |  |
| slice\_geom\_qp\_offset | se(v) | 7.4.3.2 |
| if( geom\_tree\_type == 1 ) |  |  |
| slice\_ptree\_qp\_period\_log2\_offset | se(v) | 9.3.2.1 |
| } |  |  |
| if(trisoup\_enabled ) |  |  |
| trisoup\_params(  ) |  |  |
| if( geom\_tree\_type == 1 ) { |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| ptn\_resid\_abs\_log2\_bits[ 𝑘 ] | u(3) | 9.3.2.1 |
| if( geom\_angular\_enabled ) |  |  |
| ptn\_radius\_min | ue(v) | 9.3.2.1 |
| } |  |  |
| if( inter\_prediction\_enabled) { |  |  |
| slice\_inter\_prediction | u(1) | 7.4.3.2 |
| if( slice\_inter\_prediction) { |  |  |
| if( global\_motion\_enabled) |  |  |
| global\_motion\_params(0) |  |  |
| if(biprediction\_enabled) |  |  |
| slice\_biprediction | u(1) | 7.4.3.2 |
| if( slice\_biprediction && global\_motion\_enabled) |  |  |
| global\_motion\_params(1) |  |  |
| } |  |  |
| } |  |  |
| if( slice\_inter\_prediction &&  inter\_entropy\_continuation\_enabled) { |  |  |
| slice\_inter\_entropy\_continuation | u(1) | 7.4.3.2 |
| if( slice\_inter\_entropy\_continuation ) { |  |  |
| prev\_inter\_entropy\_frame\_ctr\_lsb | u(v) | 7.4.3.2 |
| prev\_inter\_entropy\_slice\_id | ue(v) | 7.4.3.2 |
| } |  |  |
| } |  |  |
| if(fgs\_layer\_group\_enabled) |  |  |
| if( geom\_tree\_type == 0 ) |  |  |
| fgs\_geometry\_data\_unit\_parameter() |  |  |
| byte\_alignment( ) |  |  |
| } |  |  |

#### Geometry data unit footer syntax

|  |  |  |
| --- | --- | --- |
| geometry\_data\_unit\_footer(occtreeMaxDepthMinus1 ) { | Descriptor | Semantics |
| byte\_alignment( ) |  |  |
| if( occtree\_point\_cnt\_list\_present ) |  |  |
| for( dpth = 1; dpth < occtreeMaxDepthMinus1; dpth++ ) |  |  |
| occtree\_lvl\_point\_cnt\_minus1[ dpth ] | u(24) | 9.2.3 |
| slice\_num\_points\_minus1 | u(24) | 7.4.3.3 |
| } |  |  |

#### Occupancy tree syntax

|  |  |  |
| --- | --- | --- |
| occupancy\_tree(occtreeMaxDepthMinus1) { | Descriptor | Semantics |
| if( slice\_inter\_prediction && global\_motion\_enabled && MotionPartitionType == 1) |  |  |
| for(idx = 0; idx < NumMotionBlocks; idx++) |  |  |
| gm\_comp\_partition\_block[idx] | ae(v) | 9.2.15.2.2 |
| OccQpSubtreeDepth = occtreeMaxDepthMinus1 + 1 |  | 9.2.14.4 |
| for( Dpth = 0; Dpth ≤ occtreeMaxDepthMinus1; Dpth++ ) { |  |  |
| occupancy\_tree\_level( Dpth ) |  |  |
| if( Dpth + 1 > OcctreeEntropyStreamDepth ) |  | 9.2.3 |
| occtree\_end\_of\_entropy\_stream | ae(v) | 9.2.3 |
| } |  |  |
| } |  |  |

#### Occupancy tree level syntax

|  |  |  |
| --- | --- | --- |
| occupancy\_tree\_level( dpth ) { | Descriptor | Semantics |
| if( geom\_scaling\_enabled && dpth < OccQpSubtreeDepth ) |  |  |
| occ\_subtree\_qp\_offset\_present | ae(v) | 9.2.14.3 |
| if( occ\_subtree\_qp\_offset\_present ) |  |  |
| OccQpSubtreeDepth = dpth |  |  |
| for( NodeIdx = 0; NodeIdx < OccNodeCnt[ dpth ]; NodeIdx++ ) |  |  |
| occupancy\_tree\_node( dpth, NodeIdx ) |  |  |
| } |  |  |

#### Occupancy tree node syntax

|  |  |  |
| --- | --- | --- |
| occupancy\_tree\_node( dpth, nodeIdx ) { | Descriptor | Semantics |
| if( occ\_subtree\_qp\_offset\_present ) { |  |  |
| occ\_subtree\_qp\_offset\_abs[ Ns ][ Nt ][ Nv ] | ae(v) | 9.2.14.3 |
| if( occ\_subtree\_qp\_offset\_abs[ Ns ][ Nt ][ Nv ] ) |  |  |
| occ\_subtree\_qp\_offset\_sign[ Ns ][ Nt ][ Nv ] | ae(v) | 9.2.14.3 |
| } |  |  |
| if(occtree\_direct\_coding\_mode && DirectNodePresent && geo\_planar\_idcm\_angular\_disabled) |  |  |
| occ\_direct\_node | ae(v) | 9.2.12.2 |
| if( occtree\_planar\_enabled ) { |  |  |
| if(gps\_extension\_present ) { |  |  |
| if(AllowPlanarCopyMode ) |  | 9.2.11.10 |
| planar\_copy\_mode | ae(v) | 9.2.11.2 |
| if((¬planar\_copy\_mode && MultiPlanarEligible ) |  | 9.2.11.8 |
| multi\_planar\_flag | ae(v) | 9.2.11.2 |
| } |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| if( PlanarEligible[ 𝑘 ] ) { |  | 9.2.11.5 |
| if( ¬gps\_extension\_present || ¬PlanarInferred[ 𝑘 ]) |  | 9.2.11.9 |
| occ\_single\_plane[ 𝑘 ] | ae(v) | 9.2.11.2 |
| if( occ\_single\_plane[ 𝑘 ]  || ¬PlanarPosInferred[ 𝑘 ]) |  | 9.2.11.11 |
| occ\_plane\_pos[ 𝑘 ] | ae(v) | 9.2.11.2 |
| } |  |  |
| } |  |  |
| if( occtree\_direct\_coding\_mode && DirectNodePresent && ¬geo\_planar\_idcm\_angular\_disabled ) |  | 9.2.12.3.2 |
| occ\_direct\_node | ae(v) | 9.2.12.2 |
| if( occ\_direct\_node ) |  |  |
| occupancy\_tree\_direct\_node( ) |  |  |
| else { |  |  |
| if( OccMaybeSingleChild ) |  | 9.2.6.8 |
| occ\_single\_child | ae(v) | 9.2.6.2 |
| if( occ\_single\_child ) |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| if( OccFreeAxis[ 𝑘 ] ) |  | 9.2.6.6 |
| occupancy\_idx[ 𝑘 ] | ae(v) | 9.2.6.2 |
| if( OccMapPresent ) |  | 9.2.6.9 |
| if( occtree\_bitwise\_coding ) { |  |  |
| for( 𝑖 = 0; 𝑖 < 8; 𝑖++ ) |  |  |
| if( OccBitPresent[ 𝑖 ] ) |  | 9.2.10.3 |
| occupancy\_bit[ 𝑖 ] | ae(v) | 9.2.6.2 |
| } else |  |  |
| occupancy\_byte | de(v) | 9.2.6.2 |
| if( TerminalNode && geom\_dup\_point\_counts\_enabled ) |  | 9.2.6.5 |
| for( child = 0; child < OccChildCnt; child++ ) |  |  |
| occ\_dup\_point\_cnt[ child ] | ae(v) | 9.2.6.2 |
| } |  |  |
| } |  |  |

#### Direct node syntax

|  |  |  |
| --- | --- | --- |
| occupancy\_tree\_direct\_node( ) { | Descriptor | Semantics |
| direct\_point\_cnt\_eq2 | ae(v) | 9.2.12.2 |
| if( geom\_dup\_point\_counts\_enabled && ¬direct\_point\_cnt\_eq2 ) |  |  |
| direct\_dup\_point\_cnt | ae(v) | 9.2.12.2 |
| if( occtree\_direct\_joint\_coding\_enabled && direct\_point\_cnt\_eq2 ) |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| if( ¬geom\_angular\_enabled || 𝑘 == ( 1 ^ AzimuthAxis ) ) { |  | 9.2.13.3 |
| direct\_joint\_prefix[ 𝑘 ] | ae(v) | 9.2.12.2 |
| if( DnJointDiffBitPresent[ 𝑘 ] ) |  | 9.2.12.5.4 |
| direct\_joint\_diff\_bit[ 𝑘 ] | ae(v) | 9.2.12.2 |
| } |  |  |
| for( dnPt = 0; dnPt ≤ direct\_point\_cnt\_eq2; dnPt++ ) |  |  |
| if( geom\_angular\_enabled ) { |  |  |
| direct\_rem[ dnPt ][ 1 ^ AzimuthAxis ] | ae(v) | 9.2.12.2 |
| beam\_idx\_resid\_abs[ dnPt ] | ae(v) | 9.2.12.2 |
| if( beam\_idx\_resid\_abs[ dnPt ] ) |  |  |
| beam\_idx\_resid\_sign[ dnPt ] | ae(v) | 9.2.12.2 |
| direct\_rem\_st\_ang[ dnPt ] | ae(v) | 9.2.12.2 |
| if( occtree\_angular\_extension\_enabled) { |  |  |
| if(DnBitsAfterPlanar[2] > 0) |  |  |
| direct\_v\_ang\_resid\_abs[ dnPt ] | ae(v) | 9.2.12.2 |
| if( direct\_v\_ang\_resid\_abs[ dnPt ] > 0) |  |  |
| direct\_v\_ang\_resid\_sign[ dnPt ] | ae(v) | 9.2.12.2 |
| } else |  |  |
| direct\_rem\_v\_ang[ dnPt ] | ae(v) | 9.2.12.2 |
| } else |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| direct\_rem[ dnPt ][ 𝑘 ] | ae(v) | 9.2.12.2 |
| } |  |  |

#### Predictive tree syntax

|  |  |  |
| --- | --- | --- |
| predictive\_tree( ) { | Descriptor | Semantics |
| PtnCnt = 0 |  |  |
| do { |  |  |
| predictive\_tree\_node( 0, PtnCnt ) |  |  |
| ptree\_end\_of\_slice | ae(v) | 9.3.2.1 |
| } while( ¬ptree\_end\_of\_slice ) |  |  |
| } |  |  |

#### Predictive tree node syntax

|  |  |  |
| --- | --- | --- |
| predictive\_tree\_node( dpth, nodeIdx ) { | Descriptor | Semantics |
| PtnCnt++ |  |  |
| if( geom\_scaling\_enabled && ¬( nodeIdx % PtnQpInterval ) ) { |  |  |
| ptn\_qp\_offset\_abs[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if( ptn\_qp\_offset\_abs[ nodeIdx ] ) |  |  |
| ptn\_qp\_offset\_sign[ nodeIdx ] | ae(v) | 9.3.2.2 |
| } |  |  |
| if( geom\_dup\_point\_counts\_enabled ) |  |  |
| ptn\_dup\_point\_cnt[ nodeIdx ] | ae(v) | 9.3.2.2 |
| ptn\_child\_cnt\_xor1[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if(slice\_inter\_prediction && dpth) |  |  |
| ptn\_inter\_flag[nodeIdx] | ae(v) | 9.3.2.2 |
| if(slice\_inter\_prediction && slice\_biprediction && dpth) |  |  |
| ptn\_pred\_direction[nodeIdx] | ae(v) | 9.3.2.2 |
| if(ptn\_inter\_flag[nodeIdx]) |  |  |
| ptn\_inter\_pred\_mode[nodeIdx] | ae(v) | 9.3.2.2 |
| else |  |  |
| if( ¬ptree\_ang\_azimuth\_scaling\_enabled) |  |  |
| ptn\_pred\_mode[ nodeIdx ] | ae(v) | 9.3.2.2 |
| else |  |  |
| ptn\_pred\_idx[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if( geom\_angular\_enabled ) { |  |  |
| ptn\_phi\_mul\_abs\_prefix[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if( ptn\_phi\_mul\_abs\_prefix[ nodeIdx ] == 2 ) |  |  |
| ptn\_phi\_mul\_abs\_minus2[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if( ptn\_phi\_mul\_abs\_minus2[ nodeIdx ] == 7 ) |  |  |
| ptn\_phi\_mul\_abs\_minus9[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if( ptn\_phi\_mul\_abs\_prefix[ nodeIdx ] ) |  |  |
| ptn\_phi\_mul\_sign[ nodeIdx ] | ae(v) | 9.3.2.2 |
| } |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) { |  |  |
| if( 𝑘 == 0 && ptree\_ang\_azimuth\_scaling\_enabled) { |  |  |
| **ptn\_radius\_resid\_abs**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if(ptn\_radius\_resid\_abs[ nodeIdx ] > 0) |  |  |
| **ptn\_radius\_resid\_sign**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| } |  |  |
| else if( 𝑘 == 1 && ptree\_ang\_azimuth\_scaling\_enabled) { |  |  |
| **ptn\_phi\_resid\_abs\_gt0**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if(ptn\_phi\_resid\_abs\_gt0[ nodeIdx ]) { |  |  |
| **ptn\_phi\_resid\_sign**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| **ptn\_phi\_resid\_abs\_gt1**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| if(ptn\_phi\_resid\_abs\_gt1[ nodeIdx ]) |  |  |
| **ptn\_phi\_resid\_abs\_rem**[ nodeIdx ] | ae(v) | 9.3.2.2 |
| } |  |  |
| } |  |  |
| else if( 𝑘 < 2 || ¬geom\_angular\_enabled || num\_beams\_minus1 ) { |  |  |
| ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| if( ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ] ) { |  |  |
| ptn\_resid\_abs\_log2[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| ptn\_resid\_abs\_rem[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| if( 𝑘 || ptn\_pred\_mode[ nodeIdx ] ) |  |  |
| ptn\_resid\_sign[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| } |  |  |
| } |  |  |
| } |  |  |
| if( geom\_angular\_enabled  && ¬ptree\_sec\_resid\_disabled) |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) { |  |  |
| ptn\_sec\_resid\_abs[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| if( ptn\_sec\_resid\_abs[ nodeIdx ][ 𝑘 ] ) |  |  |
| ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] | ae(v) | 9.3.2.2 |
| } |  |  |
| for( 𝑖 = 0; 𝑖 < ( ptn\_child\_cnt\_xor1[ nodeIdx ] ^ 1 ); 𝑖++ ) |  |  |
| predictive\_tree\_node( dpth + 1, PtnCnt ) |  |  |
| } |  |  |

#### TriSoup syntax

|  |  |  |
| --- | --- | --- |
| trisoup(  ) { | Descriptor | Semantics |
| for( edgeIdx = 0; edgeIdx <  numberTriSoupEdges; edgeIdx++ ) { |  |  |
| vertex\_present[edgeIdx] | ae(v) | 9.4.2.2 |
| if( vertex\_present[edgeIdx]) |  |  |
| for( bit = 0; bit <  trisoup\_vertex\_number\_bits; bit++ ) |  |  |
| vertex\_position[edgeIdx][bit] | ae(v) | 9.4.2.2 |
| } |  |  |
| for( nodeIdx = 0; nodeIdx <  numberTriSoupNodes; nodeIdx++ ) { |  |  |
| if( trisoup\_centroid\_vertex\_residual\_present &&  trisoup\_sampling\_value\_minus1 ≤ 3 && numVertex > 3) { |  |  |
| centroid\_residual\_is\_zero[nodeIdx] | ae(v) | 9.4.2.3 |
| if(¬centroid\_residual\_is\_zero[nodeIdx]) { |  |  |
| if( highBound[nodeIdx] && lowBound[nodeIdx] ) |  |  |
| centroid\_residual\_sign[nodeIdx] | ae(v) | 9.4.2.3 |
| magBound = (centroid\_residual\_sign ? highBound[nodeIdx] : lowBound[nodeIdx]) - 1 |  |  |
| bit = 0 |  |  |
| while( magBound > 0) { |  |  |
| centroid\_residual\_magnitude[nodeIdx] [*bit*] | ae(v) | 9.4.2.3 |
| if( centroid\_residual\_magnitude[nodeIdx][*bit*] ) |  |  |
| Break |  |  |
| magBound-- |  |  |
| bit++ |  |  |
| } |  |  |
| magBits[nodeIdx] = bit |  |  |
| } |  |  |
| } |  |  |
| } |  |  |
| if(trisoup\_face\_vertex\_enabled) |  |  |
| for( nodeIdx = 0; nodeIdx <  numberTriSoupNodes; nodeIdx++ ) |  |  |
| for(fvIdx = 0; fvIdx < 3; fvIdx++) |  |  |
| if(FaceEligible[nodeIdx][fvIdx]) |  | 9.4.2.4 |
| has\_face\_vertex[nodeIdx][fvIdx] | ae(v) | 9.4.2.3 |
| } |  |  |

#### TriSoup parameters syntax

|  |  |  |
| --- | --- | --- |
| trisoup\_params(  ) { | Descriptor | Semantics |
| trisoup\_node\_size\_log2\_minus2 | u(3) | 9.4.2.1 |
| trisoup\_sampling\_value\_minus1 | u(8) | 9.4.2.1 |
| trisoup\_num\_unique\_segments\_bits\_minus1 | ue(v) | 9.4.2.1 |
| trisoup\_num\_unique\_segments\_minus1 | u(v) | 9.4.2.1 |
| trisoup\_vertex\_number\_bits | u(3) | 9.4.2.1 |
| trisoup\_centroid\_vertex\_residual\_present | u(1) | 9.4.2.1 |
| if( trisoup\_centroid\_vertex\_residual\_present) |  |  |
| trisoup\_face\_vertex\_enabled | u(1) | 9.4.2.1 |
| trisoup\_halo\_enabled | u(1) | 9.4.2.1 |
| if( trisoup\_halo\_enabled) |  |  |
| trisoup\_adaptive\_halo\_enabled | u(1) | 9.4.2.1 |
| trisoup\_vertex\_merge\_enabled | u(1) | 9.4.2.1 |
| if( trisoup\_non\_cubic\_node\_start\_edge\_present) { |  |  |
| trisoup\_slice\_bb\_pos\_bits | ue(v) | 9.4.2.1 |
| if( trisoup\_slice\_bb\_pos\_bits) { |  |  |
| trisoup\_slice\_bb\_pos\_log2\_scale | ue(v) | 9.4.2.1 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| trisoup\_slice\_bb\_pos\_xyz[*k*] | u(v) | 9.4.2.1 |
| } |  |  |
| } |  |  |
| if( trisoup\_non\_cubic\_node\_end\_edge\_present) { |  |  |
| trisoup\_slice\_bb\_width\_bits | ue(v) | 9.4.2.1 |
| if( trisoup\_slice\_bb\_width\_bits) { |  |  |
| trisoup\_slice\_bb\_width\_log2\_scale | ue(v) | 9.4.2.1 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| trisoup\_slice\_bb\_width\_xyz[*k*] | u(v) | 9.4.2.1 |
| } |  |  |
| } |  |  |
| } |  |  |

#### Global motion parameters syntax

|  |  |  |
| --- | --- | --- |
| global\_motion\_params(idx ) { | Descriptor | Semantics |
| if( geom\_tree\_type == 1 ) |  |  |
| slice\_inter\_frame\_ref\_gmc[ *idx*] | u(1) | 7.4.3.2 |
| if( geom\_tree\_type == 0 ||  slice\_inter\_frame\_ref\_gmc[ *idx*] ) { |  |  |
| for(*i* = 0; *i* < 3;*i*++ ) |  |  |
| for( *j* = 0; *j* < 3; *j*++ ) |  |  |
| gm\_matrix[ *idx*][*i* ][*j* ] | se(v) | 7.4.3.2 |
| for(*j* = 0; *j* < 3; *j*++ ) |  |  |
| gm\_trans[ *idx*][*j* ] | se(v) | 7.4.3.2 |
| } |  |  |
| if( geom\_tree\_type == 0 && idx == 0) { |  |  |
| motion\_partition\_type | u(1) | 7.4.3.2 |
| motion\_zero\_origin | u(1) | 7.4.3.2 |
| if(MotionPartitionType == 1) |  |  |
| for(*k* = 0; *k* < 3; *k*++ ) |  |  |
| motion\_block\_size[ *k*] | ue(v) | 7.4.3.2 |
| } |  |  |
| if( MotionPartitionType == 0 || slice\_inter\_frame\_ref\_gmc[ *idx*]) { |  |  |
| gm\_thres\_top[ *idx*] | se(v) | 7.4.3.2 |
| gm\_thres\_bot[ *idx*] | se(v) | 7.4.3.2 |
| } |  |  |
| } |  |  |

### Attribute data unit

#### Attribute data unit syntax

|  |  |
| --- | --- |
| attribute\_data\_unit( ) { | Descriptor |
| attribute\_data\_unit\_header( ) |  |
| if( attr\_coding\_type ≠ 3 ) { |  |
| if( raht\_inter\_layer\_code\_enabled || raht\_intra\_layer\_code\_enabled) |  |
| raht\_layer\_pred\_modes( ) |  |
| attribute\_coeffs( ) |  |
| } |  |
| else |  |
| attribute\_raw( ) |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute data unit header syntax

|  |  |  |
| --- | --- | --- |
| attribute\_data\_unit\_header( ) { | Descriptor | Semantics |
| adu\_attr\_parameter\_set\_id | u(4) | 7.4.4.2 |
| adu\_temporal\_id | u(3) | 7.4.4.2 |
| adu\_sps\_attr\_idx | ue(v) | 7.4.4.2 |
| adu\_slice\_id | ue(v) | 7.4.4.2 |
| if( lod\_dist\_log2\_offset\_present  || attr\_inter\_prediction\_enabled) |  |  |
| lod\_dist\_log2\_offset | se(v) | 10.6.2 |
| if( last\_comp\_pred\_enabled && AttrDim == 3) |  |  |
| for( dpth = 0; dpth ≤ lod\_max\_levels\_minus1; dpth++ ) |  |  |
| last\_comp\_pred\_coeff\_diff[ dpth ] | se(v) | 10.6.10.1 |
| if( inter\_comp\_pred\_enabled ) |  |  |
| for( dpth = 0; dpth ≤ lod\_max\_levels\_minus1; dpth++ ) |  |  |
| for( 𝑐 = 1; 𝑐 < AttrDim; 𝑐++) |  |  |
| inter\_comp\_pred\_coeff\_diff[ dpth ][ 𝑐 ] | se(v) | 10.6.10.1 |
| if( attr\_qp\_offsets\_present ) |  |  |
| for( qc = 0; qc < Min( 2, AttrDim ); qc++) |  |  |
| attr\_qp\_offset[ qc ] | se(v) | 10.7.1 |
| attr\_qp\_layers\_present | u(1) | 10.7.1 |
| if( attr\_qp\_layers\_present ) { |  |  |
| attr\_qp\_layer\_cnt\_minus1 | ue(v) | 10.7.1 |
| for( dpth = 0; dpth ≤ attr\_qp\_layer\_cnt\_minus1; dpth++ ) |  |  |
| for( qc = 0; qc < Min( 2, AttrDim ); qc++ ) |  |  |
| attr\_qp\_layer\_offset[ dpth ][ qc ] | se(v) | 10.7.1 |
| } |  |  |
| attr\_qp\_region\_cnt | ue(v) | 10.7.1 |
| if( attr\_qp\_region\_cnt ) |  |  |
| attr\_qp\_region\_bits\_minus1 | ue(v) | 10.7.1 |
| for( 𝑖 = 0; 𝑖 < attr\_qp\_region\_cnt; 𝑖++ ) { |  |  |
| if( ¬attr\_coord\_conv\_enabled ) { |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| attr\_qp\_region\_origin\_xyz[ 𝑖 ][ 𝑘 ] | u(v) | 10.7.1 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| attr\_qp\_region\_size\_minus1\_xyz[ 𝑖 ][ 𝑘 ] | u(v) | 10.7.1 |
| } else { |  |  |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| attr\_qp\_region\_origin\_rpi[ 𝑖 ][ 𝑘 ] | u(v) | 10.7.1 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| attr\_qp\_region\_size\_minus1\_rpi[ 𝑖 ][ 𝑘 ] | u(v) | 10.7.1 |
| } |  |  |
| for( ps = 0; ps < Min( 2, AttrDim ); ps++) |  |  |
| attr\_qp\_region\_offset[ 𝑖 ][ ps ] | se(v) | 10.7.1 |
| } |  |  |
| if(attr\_coding\_type == 0 && lossless\_coding\_enabled == 0) { |  |  |
| attr\_AC\_qp\_offset\_present | u(1) | 10.7.1 |
| if(attr\_AC\_qp\_offset\_present){ |  |  |
| attr\_AC\_qp\_layer\_cnt\_minus1 | ue(v) | 10.7.1 |
| for( dpth = 0; dpth < attr\_AC\_qp\_layer\_cnt\_minus1; dpth++) |  |  |
| for( ACcompidx = 0; ACcompidx < 7; ACcompidx++) |  |  |
| for( qc = 0; qc < Min( 2, AttrDim ); qc++) |  |  |
| attr\_AC\_qp\_offset[dpth][qc ][ACcompidx] | se(v) | 10.7.1 |
| } |  |  |
| } |  |  |
| if(attr\_inter\_prediction\_enabled) { |  |  |
| slice\_attr\_inter\_prediction | u(1) | 7.4.4.2 |
| if(slice\_biprediction) |  |  |
| slice\_attr\_inter\_prediction2 | u(1) | 7.4.4.2 |
| if( slice\_attr\_inter\_prediction && raht\_send\_inter\_filters &&  ¬raht\_inter\_layer\_code\_enabled) { |  |  |
| num\_inter\_filters | ue(v) | 7.4.4.2 |
| for( filteridx = 0; filteridx < num\_inter\_filters; filteridx++) |  |  |
| raht\_inter\_filter\_qidx[filteridx] | se(v) | 7.4.4.2 |
| } |  |  |
| if( raht\_inter\_layer\_code\_enabled && slice\_attr\_inter\_prediction) { |  |  |
| **layer\_code\_depth** | ue(v) | 10.7.1 |
| if( raht\_send\_inter\_filters) |  |  |
| for( filteridx = 0; filteridx < layer\_code\_depth + 1 – raht\_inter\_skip\_layer; filteridx++) |  |  |
| if( filteridx == 0 && raht\_inter\_skip\_layer == 0) |  |  |
| raht\_inter\_filter\_qidx[filteridx] | se(v) | 7.4.4.2 |
| else if( slice\_raht\_inter\_layer\_code\_mode[filteridx + raht\_inter\_skip\_layer - 1]) |  |  |
| raht\_inter\_filter\_qidx[filteridx] | se(v) | 7.4.4.2 |
| } |  |  |
| } |  |  |
| if(fgs\_layer\_group\_enabled) |  |  |
| if(attr\_coding\_type == 1 || attr\_coding\_type == 2) |  |  |
| fgs\_attribute\_data\_unit\_parameter() |  |  |
| byte\_alignment( ) |  |  |
| } |  |  |

#### Attribute data unit coefficients syntax

|  |  |  |
| --- | --- | --- |
| attribute\_coeffs( ) { | Descriptor | Semantics |
| for( 𝑖 = 0; 𝑖 < PointCnt; 𝑖++ ) { |  |  |
| zero\_run\_length\_prefix | ae(v) | 10.3.1.1 |
| if( zero\_run\_length\_prefix == 3 ) { |  |  |
| zero\_run\_length\_minus3\_div2 | ae(v) | 10.3.1.1 |
| if( zero\_run\_length\_minus3\_div2 < 4) |  |  |
| zero\_run\_length\_minus3\_mod2 | ae(v) | 10.3.1.1 |
| else |  |  |
| zero\_run\_length\_minus11 | ae(v) | 10.3.1.1 |
| } |  |  |
| 𝑖 += ZeroRunLength |  |  |
| if( 𝑖 < PointCnt ) |  |  |
| attribute\_coeff\_tuple( 𝑖 ) |  |  |
| } |  |  |
| } |  |  |

#### Attribute coefficient tuple syntax

|  |  |  |
| --- | --- | --- |
| attribute\_coeff\_tuple( coeffIdx ) { | Descriptor | Semantics |
| for( 𝑐 = 0, inferLastComp = 1; 𝑐 < AttrDim; 𝑐++ ) { |  |  |
| coeff\_abs[ 𝑐 ] | ae(v) | 10.3.1.2 |
| if( coeff\_abs[ 𝑐 ] || ( 𝑐 == AttrDim – 1 && inferLastComp ) ) |  |  |
| coeff\_sign[ 𝑐 ] | ae(v) | 10.3.1.2 |
| inferLastComp &= coeff\_abs[ 𝑐 ] == 0 |  |  |
| } |  |  |
| } |  |  |

#### Raw attribute value syntax

|  |  |  |
| --- | --- | --- |
| attribute\_raw( ) { | Descriptor | Semantics |
| for( ptIdx = 0; ptIdx < PointCnt; ptIdx++ ) |  |  |
| for( 𝑐 = 0; 𝑐 < AttrDim; 𝑐++ ) { |  |  |
| if( raw\_attr\_width\_present ) |  |  |
| raw\_attr\_component\_length | u(8) | 10.3.1.3 |
| raw\_attr\_value[ ptIdx ][ 𝑐 ] | u(v) | 10.3.1.3 |
| } |  |  |
| } |  |  |

#### Region-adaptive hierarchical layer prediction mode syntax

|  |  |  |
| --- | --- | --- |
| raht\_layer\_pred\_modes( ) { | Descriptor | Semantics |
| start = RahtRootLvl |  |  |
| for( lvl = start; lvl ≥ 0; lvl--) { |  |  |
| if(  raht\_inter\_layer\_code\_enabled) |  |  |
| slice\_raht\_inter\_layer\_code\_mode[ lvl ] | ae(v) | 7.4.4.2 |
| if( raht\_intra\_layer\_code\_enabled ) |  |  |
| slice\_raht\_intra\_layer\_code\_mode[ lvl ] | ae(v) | 7.4.4.2 |
| } |  |  |
| } |  |  |

### Defaulted attribute data unit syntax

|  |  |  |
| --- | --- | --- |
| defaulted\_attribute\_data\_unit( ) { | Descriptor | Semantics |
| defattr\_seq\_parameter\_set\_id | u(4) | 7.4.5 |
| defattr\_reserved\_zero\_3bits | u(3) | 7.4.5 |
| defattr\_sps\_attr\_idx | ue(v) | 7.4.5 |
| defattr\_slice\_id | ue(v) | 7.4.5 |
| for( 𝑐 = 0; 𝑐 < AttrDim; 𝑐++ ) |  |  |
| defattr\_value[ 𝑐 ] | u(v) | 7.4.5 |
| byte\_alignment( ) |  |  |
| } |  |  |

## Semantics

### General

The semantics associated with the syntax structures and with the syntax elements within these structures are specified either in 7.4 or in the subclause identified by the semantics column of the syntax table.

When the semantics of a syntax element are specified in tabular form, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

General constraints on syntax element values are specified in Annex A.

### Parameter sets, ancillary data and byte alignment

#### Sequence parameter set data unit semantics

##### General

The parameters specified by an SPS shall apply to any DU where that SPS is activated.

simple\_profile\_compliant specifies whether (when 1) or not (when 0) the bitstream conforms to the Simple profile.

dense\_profile\_compliant specifies whether (when 1) or not (when 0) the bitstream conforms to the Dense profile.

predictive\_profile\_compliant specifies whether (when 1) or not (when 0) the bitstream conforms to the Predictive profile.

main\_profile\_compliant specifies whether (when 1) or not (when 0) the bitstream conforms to the Main profile.

reserved\_profile\_18bits shall be equal to 0 in bitstreams conforming to this version of this document. Other values for reserved\_profile\_18bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of reserved\_profile\_18bits.

slice\_reordering\_constraint specifies whether (when 1) or not (when 0) the bitstream is sensitive to the reordering or removal of slices within a coded point cloud frame. If slices are reordered or removed when slice\_reordering\_constraint is 1, the resulting bitstream might not be fully decodable.

unique\_point\_positions\_constraint equal to 1 specifies that in each coded point cloud frame, all points shall have unique positions. unique\_point\_positions\_constraint equal to 0 specifies that in any coded point cloud frame, two or more points may have the same position.

* 1. Even if the points in each slice have unique positions, points from different slices in the same frame can be coincident. In this case, unique\_point\_positions\_constraint would be set to 0.
  2. Points with identical positions in the same frame are prohibited when unique\_point\_positions\_constraint is 1 even if they have different values of the frame index/number attribute.

level\_idc specifies the level to which the bitstream conforms as specified in Annex A. Bitstreams shall not contain values of level\_idc other than those specified in Annex A. Other values of level\_idc are reserved for future use by ISO/IEC.

sps\_seq\_parameter\_set\_id identifies the SPS for reference by other DUs. sps\_seq\_parameter\_set\_id shall be 0 in bitstreams conforming to this version of this document. Other values of sps\_seq\_parameter\_set\_id are reserved for future use by ISO/IEC.

frame\_ctr\_lsb\_bits specifies the length in bits of the syntax element frame\_ctr\_lsb.

slice\_tag\_bits specifies the length in bits of the syntax element slice\_tag.

bypass\_stream\_enabled specifies whether bypass bins for arithmetic-coded syntax elements are conveyed in a separate data stream. When equal to 1, the two data streams are multiplexed using a sequence of fixed-length chunks (11.3). When equal to 0, bypass bins form part of the arithmetic-coded bitstream.

cross\_attr\_prediction\_enabled specifies whether (when 1) or not (when 0) attribute values shall be coded using correlations across different types of attributes when num\_attributes is greater than 1. When cross\_attr\_prediction\_enabled is not present, it shall be inferred to be 0.

bypass\_bin\_coding\_prob\_update\_disabled specifies whether (when 1) or not (when 0) probability update for coding bypass bins shall be disabled. When bypass\_stream\_enabled is 0, bypass\_bin\_coding\_prob\_update\_disabled shall be applied. When bypass\_bin\_coding\_prob\_update\_disabled is not present, it shall be inferred to be 0.

entropy\_continuation\_enabled specifies whether (when 1) or not (when 0) the entropy parsing of a DU may depend upon the final entropy parsing state of a DU in the preceding slice. It is a requirement of bitstream conformance that entropy\_continuation\_enabled shall be 0 when slice\_reordering\_constraint is 0.

inter\_frame\_prediction\_enabled equal to 1 specifies that inter prediction may be used to derive the positions and attributes in a DU. inter\_frame\_prediction\_enabled equal to 0 specifies that inter prediction is not used to derive the positions and attributes in a DU.

inter\_entropy\_continuation\_enabled specifies whether (when 1) or not (when 0) the entropy parsing of a DU may depend upon the final entropy parsing state of a DU in the preceding frame in bitstream order. When inter\_entropy\_continuation\_enabled is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that inter\_frame\_prediction\_enabled shall be 0 and inter\_entropy\_continuation\_enabled shall be 0 when slice\_reordering\_constraint is 0.

sps\_extension\_present specifies whether (when 1) or not (when 0) sps\_extension\_data syntax elements are present in the SPS syntax structure. sps\_extension\_present shall be 0 in bitstreams conforming to this version of this document. The value of 1 for sps\_extension\_present is reserved for future use by ISO/IEC.

sps\_extension\_data may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders shall ignore all sps\_extension\_data syntax elements.

fgs\_layer\_group\_enabled equals to 1 specifies that a slice comprises multiple fine granularity slices of partial slice geometry or partial slice attribute. fgs\_layer\_group\_enabled equals to 0 specifies that a slice is not comprised by fine granularity slices. When fgs\_layer\_group\_enabled equals to 1, partially decoded occupancy tree can be reconstructed as specified in Annex E. When fgs\_layer\_group\_enabled is not present, fgs\_layer\_group\_enabled is inferred to 0.

It is a requirement for bitstream conformance that fgs\_layer\_group\_enabled shall be 0 under any of the following conditions:

* geom\_tree\_type is 1, or
* occtree\_coded\_axis\_list\_present is 1, or
* geom\_scaling\_enabled is 1 and geom\_qp\_mul\_log2 is not 3, or
* geom\_angular\_enabled is 1, or
* inter\_prediction\_enabled is 1.

##### Coordinate systems

seq\_origin\_bits specifies the length in bits of each seq\_origin\_xyz syntax element exclusive of any sign bit.

seq\_origin\_xyz[ 𝑘 ] and seq\_origin\_log2\_scale together specify the XYZ origin of the sequence and coding coordinate systems in units of the sequence coordinate system from the application-specific coordinate system origin. When seq\_origin\_bits is 0, seq\_origin\_xyz[ 𝑘 ] and seq\_origin\_log2\_scale shall be inferred to be 0. The 𝑘-th XYZ component of the origin is specified by the expression SeqOrigin[ 𝑘 ].

SeqOrigin[k] := seq\_origin\_xyz[k] << seq\_origin\_log2\_scale

seq\_bbox\_size\_bits specifies the length in bits of each seq\_bbox\_size\_minus1\_xyz syntax element.

seq\_bbox\_size\_minus1\_xyz[ 𝑘 ] plus 1 specifies the 𝑘-th XYZ component of the coded volume dimensions in the sequence coordinate system. When seq\_bbox\_size\_bits is 0, the coded volume dimensions are unspecified.

seq\_unit\_numerator\_minus1, seq\_unit\_denominator\_minus1 and seq\_unit\_is\_metres together specify the length represented by the unit vectors of the sequence coordinate system.

seq\_unit\_is\_metres equal to 1 specifies that the sequence unit vectors have a length in metres equal to:

seq\_unit\_is\_metres equal to 0 specifies that the sequence unit vectors have a length relative to the application-specific coordinate system unit vector length, AppUnit, equal to:

seq\_coded\_scale\_exponent, seq\_coded\_scale\_mantissa\_bits andseq\_coded\_scale\_mantissa together specify the scale factor that converts the coding coordinate system to the sequence coordinate system. The scale factor is represented by the syntax elements as a normalized binary floating-point value that is greater than or equal to 1. seq\_coded\_scale\_mantisssa\_bits specifies the length in bits of the syntax element seq\_coded\_scale\_mantissa. The scale factor is specified by the expression SeqCodedScale.

geom\_axis\_order specifies the correspondence between the XYZ axes and the STV axes of the coded point cloud in accordance with Table 8.

Syntax elements ending in "\_xyz" are specified using the XYZ axes. The expression StvToXyz[ 𝑘 ] is the component index of the XYZ axis that corresponds to 𝑘-th STV component. Values for StvToXyz[ 𝑘 ] are specified for every geom\_axis\_order in Table 8.

Table 8 — Definition of StvToXyz[ 𝑘 ] according to the value of geom\_axis\_order

| geom\_axis\_order | Axis (𝑘) label | | | StvToXyz[ 𝑘 ] | | |
| --- | --- | --- | --- | --- | --- | --- |
| 0 (S) | 1 (T) | 2 (V) | 0 (S) | 1 (T) | 2 (V) |
| 0 or 4 | Z | Y | X | 2 | 1 | 0 |
| 1 or 7 | X | Y | Z | 0 | 1 | 2 |
| 2 | X | Z | Y | 0 | 2 | 1 |
| 3 | Y | Z | X | 1 | 2 | 0 |
| 5 | Z | X | Y | 2 | 0 | 1 |
| 6 | Y | X | Z | 1 | 0 | 2 |

##### Attributes

Attributes are identified by their index into the SPS.

num\_attributes specifies the number of attributes enumerated by the SPS attribute list.

The expressions AttrDim, AttrBitDepth and AttrMaxVal specify the number of components, the bit depth and the maximum value respectively of the attribute identified by the variable AttrIdx. The decoding of an attribute data unit sets AttrIdx.

AttrDim := attr\_components\_minus1[AttrIdx] + 1  
  
AttrBitDepth := attr\_bitdepth\_minus1[AttrIdx] + 1  
  
AttrMaxVal := Exp2(AttrBitDepth) − 1

attr\_components\_minus1[ attrIdx ] plus 1 specifies the number of components of the identified attribute.

* 1. Attributes with more than three components can only be coded as raw attribute data (attr\_coding\_type = 3).

attr\_instance\_id[ attrIdx ] specifies the instance identifier for the identified attribute.

* 1. The value of attr\_instance\_id can be used to differentiate between attributes with identical attribute labels. For example, a point cloud might have multiple colour attributes sampled from different view points. In this case, attr\_instance\_id can be used by an application to discriminate between the view points.

attr\_bitdepth\_minus1[ attrIdx ] plus 1 specifies the bit depth of every component of the identified attribute.

attr\_label\_known[ attrIdx ], attr\_label[ attrIdx ] and attr\_label\_oid[ attrIdx ] together identify the type of data conveyed by the identified attribute. attr\_label\_known[ attrIdx ] specifies whether (when 1) the attribute is an attribute specified in this document by the value of attr\_label[ attrIdx ], or (when 0) an externally specified attribute identified by the object identifier attr\_label\_oid[ attrIdx ].

Attribute types identified by attr\_label are specified in Table 9. It is a requirement of bitstream conformance that an attribute identified by attr\_label shall have only as many components as specified as valid. Values of attr\_label not specified are reserved for future use by ISO/IEC. A decoder should decode attributes with reserved values of attr\_label.

Attribute types identified by attr\_label\_oid are not specified in this document. attr\_label\_oid specifies an ASN.1 object identifier value in the international object identifier tree. The international object identifier shall either be assigned by a registration authority in accordance with Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1 or generated without registration using a universally unique identifier (UUID) as specified by Rec. ITU‑T X.667﻿ |‌ ISO/IEC 9834‑8.

Table 9 — Identification of attribute type by attr\_label

| attr\_label | Attribute type | Valid component counts |
| --- | --- | --- |
| 0 | Colour | 1 or 3 |
| 1 | Reflectance | 1 |
| 2 | Opacity | 1 |
| 3 | Frame index | 1 |
| 4 | Frame number | 1 |
| 5 | Material identifier | 1 |
| 6 | Normal vector | 3 |

attr\_property\_cnt specifies the number of attribute\_property syntax structures present in the SPS for the attribute.

#### Attribute property semantics

##### Identification of an attribute property

An attribute\_property( attrIdx ) syntax structure specifies a property of the attribute identified by attrIdx.

attr\_prop\_type specifies the attribute property type according to Table 10. The interpretation of attribute properties identified as attribute specific are specified in accordance with the registration of attr\_label\_oid.

Table 10 — Identification of attribute parameter type by attr\_prop\_type

| attr\_prop\_type | Description |
| --- | --- |
| 0 | ITU‑T T.35 user defined |
| 1 | G-PCC user defined |
| 2 | ISO/IEC 23091‑2 video code points |
| 3 | Attribute scale and offset |
| 4 | Default attribute value |
| 5 .. 127 | Reserved for future use by ISO/IEC |
| 128 .. 255 | Attribute specific |

attr\_prop\_len shall be the length in bytes of the attribute\_property syntax structure excluding the syntax elements attr\_prop\_type and attr\_prop\_len.

##### ITU‑T T.35 user defined attribute properties

ITU‑T T.35 user defined properties contain user data registered in accordance with Rec. ITU‑T T.35. The user data are not specified by this document.

attr\_prop\_itu\_t\_t35\_country\_code is a byte having a value specified as a country code by Annex A of Rec. ITU‑T T.35.

attr\_prop\_itu\_t\_t35\_country\_code\_extension\_byte is a byte having a value specified as a country code by Annex B of Rec. ITU‑T T.35.

The ITU‑T T.35 terminal provider code and terminal provider oriented code shall be contained in the initial bytes of attr\_prop\_byte[ ], in the format specified by the administration that issued the terminal provider code. Any remaining attr\_prop\_byte data shall be data having syntax and semantics as specified by the entity identified by the ITU‑T T.35 country code and terminal provider code.

##### G-PCC user defined attribute properties

G-PCC user defined properties contain user data identified by an ASN.1 object identifier. The user data are not specified by this document.

attr\_prop\_oid specifies an ASN.1 object identifier value in the international object identifier tree in accordance with Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1.

Any attr\_prop\_byte data present shall be data having syntax and semantics as specified in accordance with the registration of the object identifier.

##### ISO/IEC 23091‑2 video code points

ISO/IEC 23091‑2 video code points establish properties of a video representation.

attr\_cicp\_colour\_primaries[ attrIdx ] specifies the chromaticity coordinates of the attribute's colour primaries in accordance with the ColourPrimaries code point in ISO/IEC 23091‑2.

attr\_cicp\_transfer\_characteristics[ attrIdx ] specifies, in accordance with the TransferCharacteristics code point in ISO/IEC 23091‑2, either the:

* reference opto-electronic transfer characteristic function of the attribute as a function of a source input, linear, optical intensity with a nominal real-valued range of 0 to 1; or
* inverse of the reference electro-optical transfer characteristic function as a function of an output, linear, optical intensity with a nominal real-valued range of 0 to 1.

attr\_cicp\_matrix\_coeffs[ attrIdx ] describes the matrix coefficients used to derive the attribute's luma and chroma signals from the green, blue and red, or *Y*, *Z* and *X* primaries in accordance with the MatrixCoefficients code point in ISO/IEC 23091‑2.

attr\_cicp\_video\_full\_range[ attrIdx ] specifies the black level and range of the attribute's luma and chroma signals as derived from , and , or , and real-valued component signals in accordance with the VideoFullRangeFlag code point in ISO/IEC 23091‑2.

##### Scale and offset properties

Attribute scale and offset parameters specify how to interpret the range of output attribute values.

1. The decoding process in this document does not scale attribute values prior to output.

attr\_offset\_bits is the length in bits of the subsequent attr\_offset[ attrIdx ] syntax element exclusive of any sign bit.

attr\_scale\_bits is the length in bits of the subsequent attr\_scale\_minus1[ attrIdx ] syntax element.

attr\_offset[ attrIdx ], attr\_scale\_minus1[ attrIdx ] and attr\_frac\_bits[ attrIdx ] together specify how coded attribute values shall be interpreted. When present, the external interpretation of each coded attribute value shall be:

##### Default attribute value

A default attribute value property specifies the value for an attribute that is not otherwise determined by an ADU.

attr\_default\_value[ attrIdx ][ 𝑐 ] specifies the default value of the 𝑐-th component of the identified attribute. The length in bits of each syntax element shall be attr\_bitdepth\_minus1[ attrIdx ] + 1.

#### Attribute property data semantics

attr\_prop\_byte[ 𝑖 ] is a byte containing data having syntax and semantics not specified in this document.

#### Tile inventory data unit semantics

A tile inventory, when present, contains metadata that defines the spatial region of each enumerated tile. Each tile is identified by either an implicit or explicit tile id.

A tile inventory shall apply from the next coded point cloud frame that follows the tile inventory data unit. It shall remain valid until it is replaced by another tile inventory.

A tile inventory DU shall occur before the first GDU of the coded point cloud frame from which it applies. It shall not occur before the last DU of any coded point cloud frame that precedes that from which it applies in data unit order.

ti\_seq\_parameter\_set\_id identifies the active SPS by its sps\_seq\_parameter\_set\_id.

ti\_frame\_ctr\_lsb\_bits specifies the length in bits of the syntax element ti\_frame\_ctr\_lsb. It is a requirement of bitstream conformance that ti\_frame\_ctr\_lsb\_bits shall be equal to frame\_ctr\_lsb\_bits of the active SPS.

ti\_frame\_ctr\_lsb should be the ti\_frame\_ctr\_lsb\_bits LSBs of FrameCtr for the next coded point cloud frame.

tile\_cnt specifies the number of tiles enumerated by the tile inventory.

tile\_id\_bits specifies the length in bits of each tile\_id syntax element. tile\_id\_bits equal to 0 specifies that tiles shall be identified by the index tileIdx.

tile\_origin\_bits\_minus1 plus 1 specifies the length in bits of each tile\_origin\_xyz syntax element exclusive of any sign bit.

tile\_size\_bits\_minus1 plus 1 specifies the length in bits of each tile\_size\_minus1\_xyz syntax element.

tile\_id[ tileIdx ] specifies the identifier of the tileIdx-th tile in the tile inventory. When tile\_id\_bits is 0, the value of tile\_id[ tileIdx ] shall be inferred to be tileIdx. It is a requirement of bitstream conformance that all values of tile\_id shall be unique within a tile inventory.

tile\_origin\_xyz[ tileId ][ 𝑘 ] and tile\_size\_minus1\_xyz[ tileId ][ 𝑘 ] indicate a bounding box in the sequence coordinate system encompassing slices identified by slice\_tag equal to tileId.

tile\_origin\_xyz[ tileId ][ 𝑘 ] specifies the 𝑘-th XYZ coordinate of the tile bounding box's lower corner relative to the tile inventory origin.

tile\_size\_minus1\_xyz[ tileId ][ 𝑘 ] plus 1 specifies the 𝑘-th XYZ dimension of the tile bounding box.

ti\_origin\_bits\_minus1 plus 1 specified the length in bits of each ti\_origin\_xyz syntax element exclusive of any sign bit.

ti\_origin\_xyz[ 𝑘 ] and ti\_origin\_log2\_scale together indicate the XYZ origin of the sequence coordinate system specified by seq\_origin\_xyz[ 𝑘 ] and seq\_origin\_log2\_scale. The values of ti\_origin\_xyz[ 𝑘 ] and ti\_origin\_log2\_scale should be equal to seq\_origin\_xyz[ 𝑘 ] and seq\_origin\_log2\_scale, respectively.

The tile inventory's 𝑘-th XYZ origin coordinate is specified by the expression TileInventoryOrigin[ 𝑘 ].

TileInventoryOrigin[k] := ti\_origin\_xyz[k] << ti\_origin\_log2\_scale

#### Geometry parameter set data unit semantics

##### General parameters

The parameters specified by a GPS shall apply to any DU where that GPS is activated.

gps\_geom\_parameter\_set\_id identifies the GPS for reference by other DUs.

gps\_seq\_parameter\_set\_id identifies the active SPS by its sps\_seq\_parameter\_set\_id.

slice\_geom\_origin\_scale\_present specifies whether (when 1) or not (when 0) slice\_geom\_origin\_log2\_scale is present in the GDU header. slice\_geom\_origin\_scale\_present equal to 0 specifies that the slice origin scale is specified by gps\_geom\_origin\_log2\_scale.

gps\_geom\_origin\_log2\_scale specifies the scale factor used to derive the slice origin from slice\_geom\_origin\_xyz when slice\_geom\_origin\_scale\_present is 0.

geom\_dup\_point\_counts\_enabled specifies whether (when 1) or not (when 0) duplicate points can be signalled in a GDU by a per-point duplication count.

1. geom\_dup\_point\_counts\_enabled equal to 0 does not prohibit the coding of the same point position multiple times within a single slice by means other than the direct\_dup\_point\_cnt, occ\_dup\_point\_cnt or ptn\_dup\_point\_cnt syntax elements.

geom\_tree\_type equal to 0 specifies that slice geometry is coded using an occupancy tree (7.3.3.4). geom\_tree\_type equal to 1 specifies that slice geometry is coded using a predictive tree (7.3.3.8).

gps\_extension\_present specifies whether (when 1) or not (when 0) gps\_extension\_data syntax elements are present in the GPS syntax structure. gps\_extension\_present shall be 0 in bitstreams conforming to this version of this document. The value of 1 for gps\_extension\_present is reserved for future use by ISO/IEC.

gps\_extension\_data may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders shall ignore all gps\_extension\_data syntax elements.

##### Angular coding parameters

geom\_angular\_enabled specifies whether (when 1) or not (when 0) slice geometry is coded using information about a set of beams located along and rotating around the V axis of the angular origin. When enabled, point positions are assumed to have been sampled along a ray cast by a beam.

The angular origin AngularOrigin, the apparent V-axis offset BeamOffsetV, the elevation angle 𝜃 of emitted rays and the rotation step angle 𝜑 advanced between ray emissions are illustrated for a single beam in Figure 4.

**图示

描述已自动生成**

Figure 4 — Origin, elevation angle and azimuth step for a beam.

slice\_angular\_origin\_present specifies whether (when 1) or not (when 0) a slice-relative angular origin is signalled in the GDU header. slice\_angular\_origin\_present equal to 0 specifies that the angular origin is gps\_angular\_origin\_xyz. When slice\_angular\_origin\_present is not present, it shall be inferred to be 0.

gps\_angular\_origin\_bits\_minus1 plus 1 specifies the length in bits of each gps\_angular\_origin\_xyz syntax element.

gps\_angular\_origin\_xyz[ 𝑘 ] specifies the 𝑘-th XYZ coordinate of the angular origin in the coding coordinate system.

num\_beams\_minus1 plus 1 specifies the number of beams enumerated by the GPS.

beam\_elevation\_init and beam\_elevation\_diff[ 𝑖 ] together specify beam elevations as gradients above the S-T plane. The elevation gradient for the 𝑖-th beam is specified by the expression BeamElev[ 𝑖 ]. It is a binary fixed-point value with 18 fractional bits.

BeamElev[i] :=  
 i == 0 ? beam\_elevation\_init :  
 i == 1 ? beam\_elevation\_init + beam\_elevation\_diff[1]  
 : 2 × BeamElev[i – 1] – BeamElev[i − 2] + beam\_elevation\_diff[i]

It is a requirement of bitstream conformance that values of BeamElev[ 𝑖 ], 𝑖 ∈ 1 .. num\_beams\_minus1, shall be greater than BeamElev[ 𝑖 – 1 ].

beam\_voffset\_init and beam\_voffset\_diff[ 𝑖 ] together specify the V-axis offsets of the enumerated beams from the angular origin. The offset is specified in units of the coding coordinate system. The offset for the 𝑖-th beam is specified by the expression BeamOffsetV[ 𝑖 ].

BeamOffsetV[i] :=  
 i == 0 ? beam\_voffset\_init  
 : BeamOffsetV[i − 1] + beam\_voffset\_diff[i]

beam\_steps\_per\_rotation\_init\_minus1 and beam\_steps\_per\_rotation\_diff[ 𝑖 ] specify the number of steps made per revolution by the rotating beams. The value for the 𝑖-th beam is specified by the expression BeamStepsPerRev[ 𝑖 ].

BeamStepsPerRev[i] :=  
 i == 0 ? beam\_steps\_per\_rotation\_init\_minus1 + 1  
 : BeamStepsPerRev[i – 1] + beam\_steps\_per\_rotation\_diff[i]

It is a requirement of bitstream conformance that values of BeamStepsPerRev[ 𝑖 ], 𝑖 ∈ 0 .. num\_beams\_minus1, shall not be 0.

ptree\_ang\_azimuth\_pi\_bits\_minus11 plus 11 specifies the number of bits that represent half a turn of a beam around the V axis. One half-turn is 𝜋 radians.

ptree\_ang\_radius\_scale\_log2 specifies a factor used to scale a point's radial angular coordinate during conversion to Cartesian coordinates.

ptree\_ang\_azimuth\_step\_minus1 plus 1 specifies the expected change in azimuth angle of the rotating beams between coded points. Azimuth prediction residuals used in angular predictive tree coding can be coded as a multiple of ptree\_ang\_azimuth\_step\_minus1 + 1 and a remainder.

occtree\_angular\_extension\_enabled specifies whether (when 1) or not (when 0) angular coding extension is enabled with occupancy tree coding. When occtree\_angular\_extension\_enabled is not present, it shall be inferred to be 0.

octree\_planar\_neigh\_prediction\_enabled specifies whether (when 1) or not (when 0) neighbour prediction coding is enabled with planar occupancy coding. When octree\_planar\_neigh\_prediction\_enabled is not present, it shall be inferred to 0.

geo\_planar\_idcm\_angular\_disabled equal to 1 specifies that if the geom\_angular\_enabled is equal to 1, the planar mode is disabled for the IDCM coded nodes. If geom\_angular\_enabled is equal to 0, geo\_planar\_idcm\_angular\_disabled is set equal to 0. When geo\_planar\_idcm\_angular\_disabled is equal to 1, occ\_direct\_node should be derived before deciding per axis planar eligibility.

ptree\_sec\_resid\_disabled specifies whether (when 1) or not (when 0) second coordinate-prediction residual coding is disabled. When ptree\_sec\_resid\_disabled is not present, it shall be inferred to be 0.

ptree\_ang\_azimuth\_scaling\_enabled specifies whether (when 1) or not (when 0) predictive geometry azimuth angle residuals shall be scaled according to the radius. The effect is to provide an adaptive quantization step size of the predictive geometry azimuth angle residuals resulting in a uniform quantization of circular arcs for any radius. When ptree\_ang\_azimuth\_scaling\_enabled is not present, it shall be inferred to be 0.

ptree\_ang\_max\_pred\_index specifies the maximum predictor index being usable in the prediction list for angular coordinates.

ptree\_ang\_pred\_list\_radius\_resid\_threshold specifies a threshold value on the absolute value of predictive geometry radius residual. This threshold is used during the dynamic update process of the prediction list for angular coordinates (9.3.3.7).

ptree\_ang\_radius\_resid\_context\_qphi\_threshold\_presentspecifies whether (when 1) or not (when 0) the threshold of the number of azimuthal angle steps used in selecting the context of the decoding radius residual is present in ptree\_ang\_redius\_resid\_context\_qphi\_threshold. When ptree\_ang\_radius\_resid\_context\_qphi\_threshold\_present is not present, it shall be inferred to be 0.

ptree\_ang\_redius\_resid\_context\_qphi\_threshold specifies the threshold of the number of azimuthal angle steps used in selecting the context of the decoding radius residual. When ptree\_ang\_redius\_resid\_context\_qphi\_threshold is not present, it shall be inferred to be 0.

The value thQphi specify the threshold used in the table of values of CtxTbl and CtxIdx for binarized ae(v) coded GDU syntax elements (11.5.3.4).

thQphi := ptree\_ang\_redius\_resid\_context\_qphi\_threshold

##### Occupancy tree parameters

occtree\_point\_cnt\_list\_present specifies whether (when 1) or not (when 0) the GDU footer enumerates the number of points in each occupancy tree level. When occtree\_point\_cnt\_list\_present is not present, it shall be inferred to be 0.

occtree\_direct\_coding\_mode greater than 0 specifies that point positions may be coded by eligible direct nodes of the occupancy tree. occtree\_direct\_coding\_mode equal to 0 specifies that direct nodes shall not be present in the occupancy tree.

1. Larger values for occtree\_direct\_coding\_mode generally increase the rate of direct node eligibility.

occtree\_direct\_joint\_coding\_enabled specifies whether (when 1) or not (when 0) direct nodes that code two points shall jointly code their positions according to a specific ordering of the points.

occtree\_coded\_axis\_list\_present equal to 1 specifies that the GDU header contains occtree\_coded\_axis syntax elements that are used to derive the node size for each occupancy tree level. occtree\_coded\_axis\_list\_present equal to 0 specifies that occtree\_coded\_axis syntax elements are not present in the GDU syntax and that the occupancy tree represents a cubic volume specified by the tree depth.

occtree\_neigh\_window\_log2\_minus1 plus 1 specifies the number of occupancy tree node locations that form each availability window within a tree level. Nodes outside a window are unavailable to any process related to nodes within the window. occtree\_neigh\_window\_log2\_minus1 equal to 0 specifies that only sibling nodes shall be considered available to the current node.

occtree\_adjacent\_child\_enabled specifies whether (when 1) or not (when 0) the adjacent children of neighbouring occupancy tree nodes are used in bitwise occupancy contextualization. When occtree\_adjacent\_child\_enabled is not present, it shall be inferred to be 0.

occtree\_intra\_pred\_max\_nodesize\_log2 minus 1 specifies the maximum size of an occupancy tree node that is eligible for intra-slice occupancy prediction. When occtree\_intra\_pred\_max\_nodesize\_log2 is not present, it shall be inferred to be 0.

occtree\_bitwise\_coding specifies whether the node occupancy bitmap is coded using (when 1) occupancy\_bit syntax elements or (when 0) the dictionary coded syntax element occupancy\_byte.

occtree\_planar\_enabled specifies whether (when 1) or not (when 0) the coding of node occupancy bitmaps is performed, in part, by the signalling of occupied and unoccupied planes. When occtree\_planar\_enabled is not present, it shall be inferred to be 0.

occtree\_planar\_threshold[ 𝑖 ] specify thresholds used in part to determine the per-axis eligibility for planar occupancy coding. The thresholds are specified from the most (𝑖 = 0) to the least (𝑖 = 2) probable planar axis. Each threshold specifies the minimum likelihood for an eligible axis that occ\_single\_plane is expected to be 1. The range [ 8, 120 ] for occtree\_planar\_threshold corresponds to the likelihood interval [ 0, 1 ).

occtree\_direct\_node\_rate\_minus1 specifies, when present, that of every 32 eligible nodes, only occtree\_direct\_node\_rate\_minus1 + 1 are permitted to be coded as direct nodes.

occtree\_planar\_buffer\_disabled specifies whether (when 1) or not (when 0) the contextualization of per-node occupied plane locations using the plane locations of previously coded nodes shall be disabled. When occtree\_planar\_buffer\_disabled is not present, it shall be inferred to be 0.

##### Scaling parameters

geom\_scaling\_enabled specifies whether (when 1) or not (when 0) the coded geometry shall be scaled during the geometry decoding process.

geom\_qp specifies the geometry QP prior to the addition of per slice and per-node offsets.

geom\_qp\_mul\_log2 specifies the scale factor to be applied to the geometry QP. There are Exp2( 3 − geom\_qp\_mul\_log2 ) QP values for every doubling of the scaling step size.

ptree\_qp\_period\_log2 specifies the period in nodes at which the predictive tree node QP offset is signalled. The period is one in every Exp2( ptree\_qp\_period\_log2 ) nodes.

occtree\_direct\_node\_qp\_offset specifies an offset relative to the slice geometry QP for scaling direct node coded point positions.

##### Inter prediction parameters

inter\_prediction\_enabled specifies whether (when 1) or not (when 0) inter prediction may be used to code the points of the point cloud. When inter\_prediction\_enabled is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that when inter\_frame\_prediction\_enabled is 0, inter\_prediction\_enabled shall be 0.

biprediction\_enabled specifies whether (when 1 or 2) or not (when 0) bi-prediction may be used to code the points of the point cloud. When biprediction\_enabled is not present, it shall be inferred to be 0.

frame\_merge\_enabled specifies whether (when 1) or not (when 0) the two reference frames of bi-prediction may be merged into a merged reference frame. When frame\_merge\_enabled is not present, it shall be inferred to be 0.

global\_motion\_enabled specifies whether (when 1) or not (when 0) global motion compensation is applied to the reference frame used for inter prediction. When global\_motion\_enabled is not present, it shall be inferred to be 0.

inter\_azim\_scale\_log2 specifies a scale factor to be applied to azimuth coordinates used to obtain azimuth look-up values during inter prediction. The values MaxQAzim and MinQAzim specify the maximum and minimum azimuth look up values.

MaxQAzim [i] := 1 << ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11 – inter\_azim\_scale\_log2 - 1

MinQAzim [i] := -(1 << ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11 – inter\_azim\_scale\_log2)

resampling\_enabled specifies a whether (when 1) or not (when 0) the radii of one or more points in a reference frame is updated using motion parameters.

max\_points\_per\_entry\_minus1 plus one specifies the maximum number of points of a reference frame that may be stored for a given laser ID and azimuth look up value derived using inter\_azim\_scale\_log2.

down\_sampling\_range specifies the range of downsampling a reference frame. The syntax down\_sampling\_range may be present only when max\_points\_per\_entry\_minus1 is greater than to 0. When down\_sampling\_range is not present, it shall be inferred to be -1.

occtree\_inter\_angular\_direct\_coding\_enabled specifies whether (when 1) or not (when 0) use angular information to determine the eligibility for direct coding of the occupancy tree nodes. When occtree\_inter\_angular\_direct\_coding\_enabled is not present, it shall be inferred to be 0.

##### TriSoup parameters

trisoup\_enabled specifies whether (when 1) or not (when 0) an occupancy tree (7.3.3.4) used for coding slice geometry is followed by TriSoup (7.3.3.10). When not present, it shall be inferred to 0.

trisoup\_non\_cubic\_node\_start\_edge\_present and trisoup\_non\_cubic\_node\_end\_edge\_present together specifies whether (when one or both are 1) or not (when both are 0) the location TriSoupNodeLoc[ nodeIdx ][ k ] and the edge length TriSoupNodeSize[ nodeIdx ][ k ] of the nodeIdx-th TriSoup node are modified as specified in 9.4.1.1.

#### Attribute parameter set data unit semantics

##### General parameters

The parameters specified by an APS shall apply to any DU where that APS is activated.

1. A single APS can be used by multiple coded attributes. The attributes are not required to be of the same type or to have the same number of components.

aps\_attr\_parameter\_set\_id identifies the APS for reference by other DUs.

aps\_seq\_parameter\_set\_id identifies the active SPS by its sps\_seq\_parameter\_set\_id.

attr\_coding\_type specifies the attribute coding method. Valid values are specified by Table 11. Other values are reserved for future use by ISO/IEC. Decoders conforming to this version of this document shall ignore (remove from the bitstream and discard) attribute data units coded with reserved values of attr\_coding\_type.

Table 11 — Interpretation of attr\_coding\_type

| attr\_coding\_type | Description | Decoding process |
| --- | --- | --- |
| 0 | Region Adaptive Hierarchical Transform (RAHT) | 10.5 |
| 1 | LoD with Predicting Transform | 10.6 |
| 2 | LoD with Lifting Transform | 10.6 |
| 3 | Raw attribute data | 10.3 |

attr\_primary\_qp\_minus4 plus 4 specifies the QP for the primary attribute component before the addition of per slice, per region and per-transform-level offsets.

attr\_secondary\_qp\_offset specifies an offset to be applied to the primary attribute QP to derive the QP for any secondary attribute components.

cross\_attr\_prediction\_enabled\_this\_type specifies whether (when 1) or not (when 0) the cross-attribute prediction is enabled for coding the current attribute if cross\_attr\_prediction\_enabled is 1. When cross\_attr\_prediction\_enabled\_this\_type is not present, it shall be inferred to be 0.

refAttrIdx specifies the index of attribute identified by its attrIdx that is used for decoding the current attribute. It shall range from 0 to num\_attributes - 1 when cross\_attr\_\_prediction\_enabled\_this\_type is 1. When refAttrIdx is not present, it shall be inferred to be - 1.

attr\_qp\_offsets\_present specifies whether (when 1) or not (when 0) per-slice attribute QP offsets, attr\_qp\_offset[ 𝑐 ], are present in the ADU header.

attr\_coord\_conv\_enabled specifies whether (when 1) attribute coding shall use scaled angular coordinates or (when 0) slice-relative STV point positions. It is a requirement of bitstream conformance that attr\_coord\_conv\_enabled shall be 0 when geom\_angular\_enabled is 0. When attr\_coord\_conv\_enabled is not present, it shall be inferred to be 0.

attr\_coord\_conv\_scale\_bits\_minus1[ 𝑘 ] plus 1 specifies the length in bits of the syntax element attr\_coord\_conv\_scale[ 𝑘 ].

attr\_coord\_conv\_scale[ 𝑘] specifies the scale factor used to scale points' 𝑘-th angular coordinate for attribute coding. The scale factor shall be in units of .

aps\_extension\_present specifies whether aps\_extension\_data syntax elements are present in the APS syntax structure. aps\_extension\_present shall be 0 in bitstreams conforming to this version of this document. The value of 1 for aps\_extension\_present is reserved for future use by ISO/IEC.

aps\_extension\_data may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders shall ignore all aps\_extension\_data syntax elements.

##### Region adaptive hierarchical transform parameters

raht\_prediction\_enabled specifies whether (when 1) or not (when 0) RAHT coefficients are predicted by upsampling and transforming the preceding coarser transform level.

raht\_prediction\_subtree\_min and raht\_prediction\_samples\_min specify thresholds that control the use of RAHT coefficient prediction.

**raht\_prediction\_samples\_min** specifies the minimum number of spatially adjacent samples from which RAHT coefficient prediction can be performed.

**raht\_prediction\_subtree\_min** specifies the minimum number of spatially adjacent samples that need to be present to prevent the disabling of RAHT coefficient prediction for every descendant of a RAHT node.

lossless\_coding\_enabled specifies whether (when 1) or not (when 0) lossless coding is applied by using Haar Transform. When lossless\_coding\_enabled is not present, it shall be inferred to be 0.

raht\_subnode\_prediction\_enabledspecifies whether (when 1) or not (when 0) the adjoining blocks are replaced by their children blocks when generate the upsampled prediction block. When raht\_subnode\_prediction\_enabled is not present, it shall be inferred to be 0.

raht\_intra\_layer\_code\_enabled specifies whether (when 1) or not (when 0) per-RAHT-layer AC coding mode is present in the ADU. It is a requirement of bitstream conformance that raht\_intra\_layer\_code\_enabled shall be 0 when raht\_prediction\_enabled is 0. When raht\_intra\_layer\_code\_enabled is not present, it shall be inferred to be 0.

raht\_prediction\_weights[  ] specifies the prediction weights applied to the normalized DC value of the blocks in transform domain prediction. raht\_prediction\_weights[ ] specify the prediction weights for the co-located block, blocks that adjoin by a face, blocks that adjoin by only an edge, child blocks that adjoin by a face and child blocks that adjoin by only an edge, respectively.

raht\_buffer\_extension\_flag specifies whether (when 1) or not (when 0) to use the extended reconstruction buffer for upsampling the attributes. When equal to 1, the attributes of previous layers are stored without rounding. When equal to 0, the attributes of previous layers are rounded prior to be stored to the buffer. When raht\_buffer\_extension\_flag is not present, it shall be inferred to be 0.

raht\_prediction\_search\_range specifies the range of Morton code in the preceding coarser transform level for which searched for spatially adjacent samples. When raht\_prediction\_search\_range is not present, it shall be inferred to be MaxSlicePoints – 1.

raht\_last\_comp\_pred\_enabled specifies whether (when 1) or not (when 0) the second coefficient component of a three-component attribute shall be used to predict the value of the third coefficient component when RAHT is used for attribute coding. When raht\_last\_comp\_pred\_enabled is not present, it shall be inferred to be 0.

##### Level of detail generation and transform parameters

pred\_set\_size\_minus1 plus 1 specifies the maximum size of the per-point predictor set.

pred\_inter\_lod\_search\_range specifies the range of indexes around a search centre which can be searched in an extended inter-detail-level search for nearest neighbours to include in a point's predictor set.

pred\_dist\_bias\_minus1\_xyz[ 𝑘 ] plus 1 specifies the factor used to weight the 𝑘-th XYZ component of the distance vector between two point positions used to calculate inter-point distances in the predictor search for a single refinement point. The expression PredBias[ 𝑘 ] specifies the factor for the 𝑘-th STV component.

PredBias[k] := pred\_dist\_bias\_minus1\_xyz[StvToXyz[k]] + 1

last\_comp\_pred\_enabled specifies whether (when 1) or not (when 0) the second coefficient component of a three-component attribute shall be used to predict the value of the third coefficient component. When last\_comp\_pred\_enabled is not present, it shall be inferred to be 0.

lod\_scalability\_enabled specifies whether (when 1) or not (when 0) attribute values shall be coded using constrained LoD generation and predictor searches. When equal to 1, attribute values can be reconstructed for a partially decoded occupancy tree as specified in Annex D. When lod\_scalability\_enabled is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that lod\_scalability\_enabled shall be 0 when any of the following conditions are true:

* geom\_tree\_type is 1, or
* occtree\_coded\_axis\_list\_present is 1, or
* geom\_scaling\_enabled is 1 and geom\_qp\_mul\_log2 is not 3, or
* pred\_blending\_enabled is 1.

pred\_max\_range\_minus1 plus 1 specifies, when present, the distance beyond which point predictor candidates shall be discarded during predictor set pruning for scalable attribute coding. The distance is specified in units of the per-detail-level block size.

lod\_max\_levels\_minus1 plus 1 specifies the maximum number of detail levels that can be generated by the LoD generation process. When lod\_max\_levels\_minus1 is not present, it shall be inferred to be MaxSliceDimLog2 − 1.

attr\_canonical\_order\_enabled specifies whether (when 1) or not (when 0) the order in which point attributes are coded is the canonical order that points are output by the geometry decoding processes specified in this document. When attr\_canonical\_order\_enabled is not present, it shall be inferred to be 0.

lod\_decimation\_mode specifies the decimation method used to generate detail levels. Valid values are specified by Table 12. Other values are reserved for future use by ISO/IEC. Decoders conforming to this version of this document shall ignore (remove from the bitstream and discard) attribute data units coded with reserved values of lod\_decimation\_mode.

Table 12 — Interpretation of lod\_decimation\_mode

| lod\_decimation\_mode | Description | Decoding process |
| --- | --- | --- |
| 0 | No decimation | 10.6.5.6 |
| 1 | Periodic subsampling | 10.6.5.5 |
| 2 | Block-based subsampling | 10.6.5.8 |

lod\_sampling\_period\_minus2[ lvl ] plus 2 specifies the sampling period used by LoD generation to sample points in detail level lvl to generate the next coarser detail level lvl + 1.

lod\_initial\_dist\_log2 specifies the block size at the finest detail level for use by LoD generation and predictor searches. When lod\_initial\_dist\_log2 is not present, it shall be inferred to be 0.

lod\_dist\_log2\_offset\_present specifies whether (when 1) or not (when 0) the per-slice block-size offset specified by lod\_dist\_log2\_offset shall be present in the ADU header. When lod\_dist\_log2\_offset\_present is not present, it shall be inferred to be 0.

pred\_direct\_max\_idx\_plus1 specifies the maximum number of single point predictors that can be used for direct prediction.

pred\_direct\_threshold specifies when a point shall be eligible for direct prediction. The threshold is for the maximum difference between predictor values in a point's predictor set. When the maximum difference is greater than or equal to the threshold, direct prediction is eligible. When the attribute bit depth is greater than eight bits, the threshold shall be scaled by Exp2( AttrBitDepth − 8 ).

pred\_direct\_avg\_disabled specifies whether (when 0) or not (when 1) the point predictor set average is a direct prediction mode.

pred\_intra\_lod\_search\_range specifies the range of indexes in a detail level's refinement list for which searched for nearest neighbours to include in a point's predictor set.

pred\_intra\_min\_lod specifies the finest detail level in which intra-detail-level prediction is enabled. When pred\_intra\_min\_lod is not present, it shall be inferred to be lod\_max\_levels\_minus1 + 1. It is a requirement of bitstream conformance that pred\_intra\_min\_lod shall be 0 when lod\_max\_levels\_minus1 is 0.

inter\_comp\_pred\_enabled specifies whether (when 1) or not (when 0) the first component of a multi-component attribute coefficient shall be used to predict the coefficients of any subsequent components. When inter\_comp\_pred\_enabled is not present, it shall be inferred to be 0.

pred\_blending\_enabled specifies whether (when 1) or not (when 0) the neighbour weights used for neighbourhood average prediction shall be blended according to the relative spatial positions of the associated points. When pred\_blending\_enabled is not present, it shall be inferred to be 0.

quant\_neigh\_weight[k] specifies the weights of k-nearest neighbor points used by point quantization weights derivation. When quant\_neigh\_weight[k] is not present, it shall be inferred to be 0.

max\_points\_per\_sort\_log2\_plus1 minus 1 specifies the max group size that points are ordered by group before point attributes are coded. When max\_points\_per\_sort\_log2\_plus1 is equal to 0, the max group size is equal to the points number in slice. When max\_points\_per\_sort\_log2\_plus1 is not present, it shall be inferred to be 0.

prediction\_with\_distribution\_enabled specifies whether (when 1) or not (when 0) prediction coefficients are derived based on the spatial distribution of the predictors. When prediction\_with\_distribution\_enabled is not present, it shall be inferred to be 0.

##### Raw attribute parameters

raw\_attr\_width\_present specifies whether (when 0) raw attribute values shall use the same fixed length encoding for every syntax element or (when 1) a per-syntax-element length.

##### Attribute inter prediction parameters

attr\_inter\_prediction\_enabled specifies whether (when 1) or not (when 0) inter prediction may be used to code the attributes of the point cloud. When attr\_inter\_prediction\_enabled is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that when inter\_frame\_enabled\_flag is 0 or inter\_prediction\_enabled is 0, attr\_inter\_prediction\_enabled shall be 0.

raht\_inter\_layer\_depth\_minus1 +1 specifies the number of layers of the transform tree where the inter prediction block (10.5.5.9) is used to modify the prediction block. When raht\_inter\_layer\_depth\_minus1 is not present, it shall be inferred to be 0.

raht\_send\_inter\_filters specifies whether (when 1) or not (when 0) RAHT inter-prediction (temporal) filters will be sent to the decoder. When raht\_send\_inter\_filters is not present, it shall be inferred to be 0.

raht\_inter\_skip\_layers specifies the number of layers of the transform tree for which the inter prediction block is skipped from the temporal filtering process (10.5.6.4). When raht\_inter\_skip\_layers is not present, it shall be inferred to be 0.

raht\_inter\_layer\_code\_enabled specifies whether (when 1) or not (when 0) per-RAHT-layer AC coding mode is present in the ADU. It is a requirement of bitstream conformance that raht\_inter\_layer\_code\_enabled shall be 0 when attr\_inter\_prediction\_enabled is 0. When raht\_inter\_layer\_code\_enabled is not present, it shall be inferred to be 0.

**attr\_inter\_prediction\_search\_range** specifies the range of indexes in a detail level's refinement list in the reference slice for which searched for nearest neighbours to include in a point's predictor set. When attr\_inter\_prediction\_search\_range is not present, it shall be inferred to be 0.

#### Frame-specific attribute properties data unit semantics

Frame-specific attribute properties apply to an attribute of a specific frame. The properties shall:

* override any corresponding properties signalled in the active SPS for the specified frame only;
* apply to all ADUs in the frame with AttrIdx equal to fsap\_sps\_attr\_idx.

All attribute properties with the same value of attr\_prop\_type shall be identical within a frame for any single attribute.

Each FSAP DU shall occur, at least, before the first ADU within the frame to which it applies.

1. The requirements of fsap\_frame\_ctr\_lsb prevent an FSAP DU from preceding the first GDU in the frame to which it applies.

fsap\_seq\_parameter\_set\_id identifies the active SPS by its sps\_seq\_parameter\_set\_id.

fsap\_frame\_ctr\_lsb\_bits specifies the length in bits of the syntax element fsap\_frame\_ctr\_lsb. It is a requirement of bitstream conformance that fsap\_frame\_ctr\_lsb\_bits shall be equal to frame\_ctr\_lsb\_bits of the active SPS.

fsap\_frame\_ctr\_lsb identifies the frame to which the frame-specific attribute properties apply. Identification shall use fsap\_frame\_ctr\_lsb\_bits LSBs of the notional frame counter, FrameCtr. fsap\_frame\_ctr\_lsb shall be equal to frame\_ctr\_lsb of the preceding GDU.

fsap\_sps\_attr\_idx identifies the coded attribute to which the frame-specific attribute properties shall apply. Identification shall be by the index into the active SPS attribute list.

fsap\_num\_props specifies the number of attribute properties present in the syntax structure.

#### Frame boundary marker data unit semantics

The frame boundary marker DU explicitly marks the end of a frame.

fbdu\_frame\_ctr\_lsb\_bits specifies the length in bits of the syntax element fbdu\_frame\_ctr\_lsb. It is a requirement of bitstream conformance that fbdu\_frame\_ctr\_lsb\_bits shall be equal to frame\_ctr\_lsb\_bits of the active SPS.

fbdu\_frame\_ctr\_lsb identifies the frame to which the frame boundary marker applies. Identification shall use fbdu\_frame\_ctr\_lsb\_bits LSBs of the notional frame counter FrameCtr.

#### User data data unit semantics

The user data DU contains user data identified by an ASN.1 object identifier. The user data are not specified by this document.

user\_data\_oid specifies an ASN.1 object identifier value in the international object identifier tree, as specified in Rec. ITU‑T X.660﻿ |‌ ISO/IEC 9834‑1.

user\_data\_byte is a byte containing data having syntax and semantics as specified by the registration of the object identifier.

#### Byte alignment semantics

The byte\_alignment syntax structure causes the bitstream to become byte-aligned.

alignment\_bit\_equal\_to\_zero shall be 0.

### Geometry data unit

#### Geometry data unit semantics

A GDU conveys the geometry of a slice and associated slice information such as a frame counter or a slice origin. A GDU comprises a GDU header, geometry coded using either an occupancy tree (when geom\_tree\_type is 0) or a predictive tree (when geom\_tree\_type is 1), and a GDU footer.

#### Geometry data unit header semantics

gdu\_geometry\_parameter\_set\_id specifies the active GPS by its gps\_geom\_parameter\_set\_id.

gdu\_temporal\_id specifies the temporal ID of the frame associated with the geometry data unit.

slice\_id identifies the slice for reference by other DUs.

slice\_tag identifies the slice as a member of a slice group with the same values for slice\_tag. When a tile inventory DU is present, the slice group shall be a tile identified by a tile id. Otherwise, when tile inventory DUs are not present, the interpretation of slice\_tag is application specific.

frame\_ctr\_lsb specifies the frame\_ctr\_lsb\_bits LSBs of the notional frame counter FrameCtr. Consecutive slices with different values of frame\_ctr\_lsb form parts of separate output point cloud frames. Consecutive slices with identical values of frame\_ctr\_lsb without an intervening frame boundary marker data unit form parts of the same coded point cloud frame.

slice\_entropy\_continuation equal to 1 specifies that the entropy parsing state restoration process (11.6.2.2 and 11.6.3.2) shall be applied at the start of the GDU and any ADUs in the slice. slice\_entropy\_continuation equal to 0 specifies that the parsing of the GDU and any ADUs in the slice is independent of any other slice in the frame when slice\_inter\_entropy\_continuation is 0. When slice\_entropy\_continuation is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that slice\_entropy\_continuation shall be 0 when the GDU is the first GDU in a coded point cloud frame. A decoder shall ignore (remove from the bitstream and discard) all slices in a coded point cloud frame with slice\_entropy\_continuation equal to 1 that are not preceded by a slice in the same frame with slice\_entropy\_continuation equal to 0.

prev\_slice\_id shall be equal to the GDU slice\_id of the preceding slice in bitstream order. A decoder shall ignore (remove from the bitstream and discard) slices where prev\_slice\_id is both present and not equal to slice\_id of the preceding slice in the same frame.

1. It is recommended that slice\_entropy\_continuation is 0 if slice\_tag is not equal to the slice\_tag of the GDU identified by prev\_slice\_id. For example, if slice\_tag is used to select a subset of slices, then decoding might be prevented if there are dependencies upon slices that were not selected.

slice\_geom\_origin\_bits\_minus1 plus 1 specifies the length in bits of each slice\_geom\_origin\_xyz syntax element.

slice\_geom\_origin\_xyz[ 𝑘 ] and slice\_geom\_origin\_log2\_scale specify the 𝑘-th XYZ coordinate of the slice origin in the coding coordinate system. The slice origin in STV coordinates is specified by the expression SliceOrigin[ 𝑘 ]. When slice\_geom\_origin\_log2\_scale is not present, it shall be inferred to be gps\_geom\_origin\_log2\_scale.

SliceOrigin[k] := slice\_geom\_origin\_xyz[StvToXyz[k]] << Max(slice\_geom\_origin\_log2\_scale, triSoupNodeSizeLog2)

slice\_angular\_origin\_bits\_minus1 plus 1 specifies the length in bits of each slice\_angular\_origin\_xyz syntax element.

slice\_angular\_origin\_xyz[ 𝑘 ] specifies the 𝑘-th XYZ coordinate of the angular origin relative in the slice's coordinate system. When slice\_angular\_origin\_xyz[ 𝑘 ] is not present, it shall be inferred to be 0.

The slice-relative angular origin in STV coordinates is specified by the expression AngularOrigin[ 𝑘 ].

AngularOrigin[k] := slice\_angular\_origin\_present  
 ? slice\_angular\_origin\_xyz[StvToXyz[k]]  
 : gps\_angular\_origin\_xyz[StvToXyz[k]] − SliceOrigin[k]

slice\_geom\_qp\_offset specifies the slice geometry QP as an offset to the GPS geom\_qp. When slice\_geom\_qp\_offset is not present, it shall be inferred to be 0.

slice\_inter\_prediction equal to 1 specifies that inter prediction may be used to derive the positions in the GDU. slice\_inter\_prediction equal to 0 specifies that inter prediction is not used to derive the positions in the GDU. When slice\_inter\_prediction is not present, it shall be inferred to be 0.

slice\_biprediction equal to 1 specifies that bi-prediction may be used to derive the positions in the GDU. slice\_biprediction equal to 0 specifies that bi-prediction is not used to derive the positions in the GDU. When slice\_biprediction is not present, it shall be inferred to be 0.

gm\_matrix[ i ][ j ] and gm\_trans[ i ] specify the motion compensation parameters in the form of a matrix and an offset that is to be applied when global motion is enabled. When gm\_matrix[ i ][ j ] is not present, it is inferred to be 0. When gm\_trans[ i ] is not present, it is inferred to be 0. The global motion matrix is specified by the expression GMMatrix[ i ][ j ].

GMMatrix[i][j] := (i == j ? 65536 : 0) + gm\_matrix[i][j]

gm\_thres\_top and gm\_thres\_bot specify the two thresholds used to determine the points to which motion compensation is applied.

**slice\_inter\_frame\_ref\_gmc** specifies whether (when 1) or not (when 0) the inter prediction GMC reference frame to be applied to the GDU unit associated with the GDU header as specified by 9.3.3.11. When slice\_inter\_frame\_ref\_gmc is not present, it shall be inferred to be 0.

gm\_matrix2[ i ][ j ] and gm\_trans2[ i ] specify the motion compensation parameters in the form of a matrix and an offset that is to be applied to the second reference frame when global motion is enabled. When gm\_matrix2[ i ][ j ] is not present, it is inferred to be 0. When gm\_trans2[ i ] is not present, it is inferred to be 0. The global motion matrix is specified by the expression GMMatrix2[ i ][ j ].

GMMatrix2[i][j] := (i == j ? 65536 : 0) + gm\_matrix2[i][j]

gm\_thres\_top2 and gm\_thres\_bot2 specify the two thresholds used to determine the points of the second reference frame to which motion compensation is applied.

**slice\_inter\_frame\_ref\_gmc2** specifies the second inter prediction GMC reference frame to be applied to the GDU unit associated with the GDU header as specified by 9.3.3.11. When slice\_inter\_frame\_ref\_gmc2 is not present, it shall be inferred to be 0.

motion\_partition\_type equal to 0 specifies that road and object partitioning shall be used to determine the points for which global motion compensation is to be applied. motion\_partition\_type equal to 1 specifies that cuboid partitioning shall be used to determine the points for which global motion compensation is to be applied.

The variable MotionPartitionType is set equal to -1 when motion\_partition\_type is not present and set equal to motion\_partition\_type when present.

motion\_zero\_origin equal to 1 specifies that the origin position {0, 0, 0} shall be used in global motion compensation using cuboid partitioning. motion\_zero\_origin equal to 0 specifies that the origin position seq\_origin\_xyz shall be used in global motion compensation using cuboid partitioning.

motion\_block\_size[k] specifies the partition block size that is to be used in global motion compensation using cuboid partitioning. When motion\_block\_size[k] is equal to 0, the block size in the k-th dimension is equal to the size the k-th dimension of the slice bounding box.

The number of motion partition blocks in the k-th axis is specified by the expression NumMotionBlocksPerAxis[k].

NumMotionBlocksPerAxis[k] := motion\_block\_size[k]   
 ? (PredPointCloudBboxSize[k] + motion\_block\_size[k] – 1)/ motion\_block\_size[k] : 1

The number of motion partition blocks is specified by the expression NumMotionBlocks.

NumMotionBlocks := NumMotionBlocksPerAxis[0] × NumMotionBlocksPerAxis[1] × NumMotionBlocksPerAxis[2]

slice\_inter\_entropy\_continuation equal to 1 specifies that the entropy parsing state restoration process (11.6.2.2 and 11.6.3.2) shall be applied at the start of the GDU and any ADUs in the slice. slice\_inter\_entropy\_continuation equal to 0 specifies that the parsing of the GDU and any ADUs in the slice is independent of any other slice when slice\_entropy\_continuation is 0. When slice\_inter\_entropy\_continuation is not present, it shall be inferred to be 0.

It is a requirement of bitstream conformance that slice\_inter\_entropy\_continuation shall be 0 when the GDU is the not the first GDU in a coded point cloud frame.

prev\_inter\_entropy\_frame\_ctr\_lsb shall be equal to the frame\_ctr\_lsb of the preceding frame in bitstream order. A decoder shall ignore (remove from the bitstream and discard) slices where prev\_inter\_entropy\_frame\_ctr\_lsb is both present and not equal to frame\_ctr\_lsb of the preceding frame in bitstream order. The number of bits used to code prev\_inter\_entropy\_frame\_ctr\_lsb is frame\_ctr\_lsb\_bits

prev\_inter\_entropy\_slice\_id shall be equal to the GDU slice\_id of the preceding slice of the preceding frame in bitstream order. A decoder shall ignore (remove from the bitstream and discard) slices where prev\_inter\_entropy\_slice\_id is both present and not equal to slice\_id of the preceding slice in the preceding frame in bitstream order.

1. It is recommended that slice\_inter\_entropy\_continuation is 0 if slice\_tag is not equal to the slice\_tag of the GDU identified by prev\_inter\_entropy\_slice\_id of the preceding frame. For example, if slice\_tag is used to select a subset of slices, then decoding might be prevented if there are dependencies upon slices that were not selected.

num\_subsequent\_subgroups specifies the number of the subsequent dependent data units which reference the context state of the current data unit.

#### Geometry data unit footer semantics

The start of the GDU footer shall be determined from the end of the GDU as specified by 11.2.4.

slice\_num\_points\_minus1 plus 1 specifies the number of points coded in the DU. It is a requirement of bitstream conformance that slice\_num\_points\_minus1 plus 1 shall be equal to the number of decodable points in the DU. Decoders shall not rely upon bitstream conformance to prevent overflow of implementation buffers.

### Attribute data unit

#### Attribute data unit semantics

An ADU codes attribute values for a single attribute in a slice. It comprises an ADU header and either attribute coefficients (attribute\_coeffs) when transform coding is enabled or directly coded attribute values (attribute\_raw).

#### Attribute data unit header semantics

adu\_attr\_parameter\_set\_id specifies the active APS by its aps\_attr\_parameter\_set\_id.

adu\_temporal\_id specifies the temporal ID of the frame associated with the attribute data unit.

adu\_sps\_attr\_idx identifies the coded attribute by its index into the active SPS attribute list.

At the start of every ADU, the variable AttrIdx is set to adu\_sps\_attr\_idx:

AttrIdx = adu\_sps\_attr\_idx

The attribute coded by the ADU shall have at most three components when attr\_coding\_type is not 3.

adu\_slice\_id specifies the value of the preceding GDU slice\_id.

**slice\_attr\_inter\_prediction** equal to 1 specifies that inter prediction may be used to derive the attributes in the ADU. slice\_attr\_inter\_prediction equal to 0 specifies that inter prediction is not used to derive the attributes in the ADU. When slice\_attr\_inter\_prediction is not present, it shall be inferred to be 0.

**slice\_attr\_inter\_prediction2** equal to 1 specifies that the inter prediction from the second reference frame may be used to derive the attributes in the ADU. slice\_attr\_inter\_prediction equal to 0 specifies that the inter prediction from the second reference frame is not used to derive the attributes in the ADU. When slice\_attr\_inter\_prediction2 is not present, it shall be inferred to be 0.

num\_inter\_filters specifies the number of RAHT temporal filters. When num\_inter\_filters is present, it is a requirement of bitstream conformance that shall be equal to Min(raht\_inter\_layer\_depth\_minus1+1, RahtRootLvl) – raht\_send\_inter\_filters. When num\_inter\_filters is not presented, it shall be infered to be 0.

raht\_inter\_filter\_qidx[] specifies the quantized residual of the RAHT temporal filters.

layer\_code\_depthspecifies the number of layers in the transform tree for which AC coefficients coding mode are signalled. When layer\_code\_depth is present, it is a requirement of bitstream conformance that layer\_code\_depth shall be equal to Min(raht\_inter\_layer\_depth\_minus1, RahtRootLvl - 1). When layer\_code\_depth is not present, it shall be inferred to be 0.

slice\_raht\_inter\_layer\_code\_mode[d]equal to 1 specifies that inter prediction is enabled for AC coefficients at the prediction depth *d* of the transform tree. slice\_raht\_inter\_layer\_code\_mode[*d*] equal to 0 specifies that inter prediction is disabled for AC coefficients at the prediction depth *d* of the transform tree.

slice\_raht\_intra\_layer\_code\_mode[d]equal to 1 specifies that intra prediction is enabled for AC coefficients at the prediction depth d of the transform tree. slice\_raht\_intra\_layer\_code\_mode[d] equal to 0 specifies that intra prediction is disabled for AC coefficients at the prediction depth d of the transform tree.

### Defaulted attribute data unit semantics

A defaulted attribute data unit specifies a single attribute value for all points in the slice.

defattr\_seq\_parameter\_set\_id specifies the active SPS by its sps\_seq\_parameter\_set\_id.

defattr\_reserved\_zero\_3bits shall be 0 in bitstreams conforming to this version of this document. Other values of defattr\_reserved\_zero\_3bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of defattr\_reserved\_zero\_3bits.

defattr\_sps\_attr\_idx identifies the coded attribute by its index into the active SPS attribute list.

At the start of every defaulted attribute data unit, the variable AttrIdx is set to defattr\_sps\_attr\_idx:

AttrIdx = defattr\_sps\_attr\_idx

defattr\_slice\_id specifies the value of the preceding GDU slice\_id.

defattr\_value[ 𝑐 ] specifies the value of the 𝑐-th attribute component for every point in the slice. The length in bits of defattr\_value[ 𝑐 ] is AttrBitDepth.

# Decoding process

## General decoding process

The reconstruction of a point cloud is specified such that all decoders that conform to a specified profile and level will produce numerically identical output point cloud frames for a bitstream conforming to that profile and level. Any decoding process that produces an identical output point cloud sequence to that produced by the process described herein conforms to the decoding process requirements of this document.

The frame decoding process (8.2) shall be repeatedly performed for each coded point cloud frame in the coded point cloud sequence.

## Frame decoding processes

### General

The result of this process is a reconstructed point cloud frame.

At the start of every coded point cloud frame, the output point cloud frame shall be initialized to the empty point cloud.

RecCloudPointCnt = 0

When inter prediction is enabled, PtnCurrFramePos is initialized to -1 for all entries. When the current frame is not an I-frame and the slices of the frame are coded using octree coding, the reference frame generation process in subclause 9.2.15 is invoked

The slice decoding process (8.3) shall be repeatedly performed for each slice in the coded point cloud frame.

### Frame counter

The variable FrameCtr represents the notional frame counter. For the first decoded frame, FrameCtr shall be set equal to frame\_ctr\_lsb. Otherwise, the variable FrameCtr shall be updated for each frame:

window = Exp2(frame\_ctr\_lsb\_bits) >> 1  
curLsb = FrameCtr % Exp2(frame\_ctr\_lsb\_bits)  
curMsb = FrameCtr >> frame\_ctr\_lsb\_bits  
if ((frame\_ctr\_lsb < curLsb) && (curLsb − frame\_ctr\_lsb) ≥ window)  
 curMsb++  
else if ((frame\_ctr\_lsb > curLsb) && (frame\_ctr\_lsb – curLsb) > window)  
 curMsb−−  
FrameCtr = (curMsb << frame\_ctr\_lsb\_bits) + frame\_ctr\_lsb

## Slice decoding processes

### General

A slice in a coded point cloud frame shall be decoded as follows:

1. Point positions are decoded from one GDU in the slice as specified by 8.3.3.
2. Default attribute values are set for each attribute as specified by 8.3.4.
3. Point attributes are decoded from each ADU in the slice as specified by 8.3.5.
4. The decoded point positions are offset and the output point count incremented as specified by 8.3.6.

Only one slice shall be decoded for every set of slices in a coded point cloud frame with the same value of slice\_id as specified in subclause 6.4.6.

### State variables

Slice decoding is specified in terms of the following state variables:

* The variable PointCnt, a cumulative count of decoded points.
* The array PointAng of angular coordinates for decoded points; PointAng[ ptIdx ][ 𝑘 ] is the 𝑘-th angular coordinate of the point position PointPos[ ptIdx ].

### Geometry decoding process

The GDU shall be decoded and the reconstructed positions stored in the output point cloud.

The expression PointPos[ ptIdx ][ 𝑘 ] is an alias into the output point cloud for points in the slice.

PointPos[ptIdx][k] := RecCloudPos[RecCloudPointCnt + ptIdx][k]

1. The definition of PointPos implicitly concatenates the points of consecutive slices.

When geom\_angular\_enabled is 1, the geometry decoding process populates the array PointAng with points' angular coordinates.

At the start of every slice, PointCnt is initialized to 0. It is incremented for each point decoded by the geometry decoding process.

Point positions shall be decoded and reconstructed as specified by Clause 9.

### Default attribute values

Attribute values for every point in the slice shall be set to their respective default values. This process shall be equivalent to the following steps for each attribute, attrIdx = 0 .. num\_attributes − 1:

* All components of the attribute values shall be set to Exp2( attr\_bitdepth\_minus1[ attrIdx ] ).
* If the attribute property attr\_default\_value[ attrIdx ] is present, the attribute values shall be set to attr\_default\_value[ attrIdx ][ 𝑐 ], for each component 𝑐.
* If the slice contains a defaulted attribute data unit with defattr\_sps\_attr\_idx equal to attrIdx, the attribute values shall be set to defattr\_value[ 𝑐 ] of that DU, for each component 𝑐.

### Attribute decoding process

The ADU shall be decoded and the reconstructed attribute values stored in the corresponding output point cloud attribute.

The expression PointAttr[ ptIdx ][ 𝑐 ] is an alias into the output point cloud attribute array for the points in the slice.

PointAttr[ptIdx][c] := RecCloudAttr[RecCloudPointCnt + ptIdx][AttrIdx][c]

Point attributes shall be decoded and reconstructed as specified by Clause 10.

### At the end of a slice

The variable RecCloudPointCnt is incremented by the number of points decoded.

RecCloudPointCnt += PointCnt

The slice geometry shall be translated from the slice's coordinate system to the coding coordinate system by the addition of the slice origin, SliceOrigin.

1. The attribute decoding processes specified in Clause 10 are performed prior to the coordinate system conversion.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (k = 0; k < 3; k++)  
 PointPos[ptIdx][k] += SliceOrigin[k]

### Update of inter prediction buffer

This subclause is invoked when inter prediction is enabled under predictive geometry coding and biprediction\_enabled is 0. The points in the reference frame are downsampled using the azimuth scale and added such that for each beamId and azimuth look value, at most max\_points\_per\_entry\_minus1 points are stored.

if (inter\_prediction\_enabled)  
 for(ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for(j = 0; j <= max\_points\_per\_entry\_minus1; j++) {  
 if (PtnCurrFramePos[beamId][qAzim][j][0] == -1) {  
 if (max\_points\_per\_entry\_minus1 > 0 && j > 0) {  
 lAttr = PtnCurrFrameAttr[beamId][qAzim][j-1]  
 lPtn = PtnCurrFramePos[beamId][qAzim][j-1]  
 if (lAttr[0] == PtnCurrFrameAttr[beamId][qAzim][j][0])  
 if ((Abs(lPtn[0] - PtnCurrFramePnt[beamId][qAzim][j][0]) > dnRadiusRange) ||  
 (Abs(lPtn[1] - PtnCurrFramePnt[beamId][qAzim][j][1]) > dnAzimuthRange)) {  
 for(k = 0; k < 3; k ++)  
 PtnCurrFramePos[beamId][qAzim][j][k] = PointAng[ptIdx][k]  
 PtnCurrFrameAttr[beamId][qAzim][j][0] = PointAttr[ptIdx][0]  
 break  
 }  
 }  
 for(k = 0; k < 3; k ++)  
 PtnCurrFramePos[beamId][qAzim][j][k] = PointAng[ptIdx][k]  
 PtnCurrFrameAttr[beamId][qAzim][j][0] = PointAttr[ptIdx][0]  
 break  
 }  
 }  
where  
 beamId := PointAng[ptIdx][2]  
 qAzim := DivExp2Fz(PointAng[ptIdx][1], inter\_azim\_scale\_log2)  
 dnRadiusRange := down\_sampling\_range × (attr\_coord\_conv\_scale[0] >> 8)  
 dnAzimuthRange := down\_sampling\_range

# Slice geometry

## General

Clause 9 specifies the coding of slice geometry and the reconstruction of point positions, storing the reconstructed geometry in the arrays PointPos and PointAng.

## Occupancy tree

### General

Subclause 9.2 specifies the parsing and reconstruction of point positions from a coded occupancy tree. It applies when geom\_tree\_type is 0.

An occupancy tree represents the slice geometry as a tree of occupancy tree nodes. Parsing or traversing a coded occupancy tree implicitly generates a representation of the slice geometry.

### Coded occupancy tree

#### General tree structure

Individual point positions are represented in the occupancy tree either by the position of leaf nodes, or by direct nodes that encode node-relative positions.

An occupancy tree node shall identify the presence of at least one point contained within the volume of an axis-aligned cuboid. The volume is defined in the slice's coordinate system by an inclusive lower corner and an exclusive upper corner . The volume edge lengths are non-negative integer powers of two. A node's size, nodeSize, is synonymous with the volume dimensions .

The occupancy tree shall be formed of one or more tree levels. Every tree level consists of tree nodes with non-overlapping volumes. All tree nodes within a tree level shall have identical volume dimensions.

The occupancy tree shall contain a single root node. The root node shall be the only node in the top level of the tree. The volume identified by the root node shall have a lower corner at position ( 0, 0, 0 ), coincident with the slice origin. The upper corner shall be at an integer position equal to the root node size.

With each subsequent tree level, starting from the top tree level, the node volume dimensions are halved along one or more coded axes. The coded axes in each tree level are enumerated in the GDU header, as specified by occtree\_coded\_axis.

The location of a node, nodeLoc, within a tree level is related to the spatial position of the node volume's lower corner in the slice coordinate system by:

Two tree nodes are spatially adjacent if their volumes share a face.

Unless an early termination condition applies (9.2.6.5), tree nodes with a volume greater than the unit cube shall have one or more child nodes. Depending upon the number of coded axes, these nodes shall have at most eight, four, two, or one child nodes, as illustrated for cubic nodes in Figure 5.

形状

描述已自动生成

Figure 5 — Arrangement of child nodes depending upon coded axes.

Leaf nodes, in the absence of geometry scaling (9.2.14), represent indivisible volumes with dimensions equal to the unit cube. When duplicate point coding is enabled, a leaf node may represent more than one point. In such cases, all points represented by the leaf node shall have identical positions.

#### Tree traversal order

The coded occupancy tree shall be traversed in breadth-first order. Traversal shall start from the top tree level. All nodes in a tree level shall be sequentially traversed before proceeding to the next level. Within a tree level, nodes shall be traversed in ascending Morton order of node location.

The traversal order for an example tree is illustrated in Figure 6. Each tree level progressively refines the slice geometry. Starting from the root node, a, the node traversal order is from a to y. Occupied nodes are shaded.

图示

描述已自动生成

Figure 6 — Occupancy tree traversal order. Illustrated for three tree levels.

#### Node occupancy bitmap

The structure of the occupancy tree is coded as a sequence of node occupancy bitmaps.

Each node occupancy bitmap shall enumerate the child nodes for a single node. Each set bit position identifies the relative location of a child node in the next level of the occupancy tree as specified by the expression OccLocC[ bitIdx ][ 𝑘 ] and Table 13. The count of set bits in the bitmap shall be the number of child nodes.

When a node has fewer than three coded axes, bits in the occupancy bitmap that do not enumerate a valid child node location shall be unset.

OccLocC[bitIdx][k] := Bit(bitIdx, 2 − k)

The tree level location of a child node is related to its parent's by:

Table 13 — Identification of valid relative child node location relLoc from set bits in an occupancy bitmap

| Coded Axes | | | Bit position (bitIdx) in occupancy bitmap | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | T | V | 7 (MSB) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (LSB) |
| 1 | 1 | 1 | ( 1, 1, 1 ) | ( 1, 1, 0 ) | ( 1, 0, 1 ) | ( 1, 0, 0 ) | ( 0, 1, 1 ) | ( 0, 1, 0 ) | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 1 | 1 | 0 |  | ( 1, 1, 0 ) |  | ( 1, 0, 0 ) |  | ( 0, 1, 0 ) |  | ( 0, 0, 0 ) |
| 1 | 0 | 1 |  |  | ( 1, 0, 1 ) | ( 1, 0, 0 ) |  |  | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 0 | 1 | 1 |  |  |  |  | ( 0, 1, 1 ) | ( 0, 1, 0 ) | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 1 | 0 | 0 |  |  |  | ( 1, 0, 0 ) |  |  |  | ( 0, 0, 0 ) |
| 0 | 1 | 0 |  |  |  |  |  | ( 0, 1, 0 ) |  | ( 0, 0, 0 ) |
| 0 | 0 | 1 |  |  |  |  |  |  | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 0 | 0 | 0 |  |  |  |  |  |  |  | ( 0, 0, 0 ) |

#### Terminal nodes

In the coded occupancy tree, leaf nodes are immediately encoded by their parent (terminal) node.

When geom\_dup\_point\_counts\_enabled is 1, terminal nodes shall encode the duplicate point counts for the leaf nodes they contain.

### Occupancy tree syntax element semantics

occtree\_depth\_minus1 plus 1 specifies the maximum number of tree levels present in the coded occupancy tree. When occtree\_coded\_axis\_list\_present is 0, the root node size is a cubic volume with edge lengths equal to Exp2( occtreeMaxDepthMinus1 + triSoupNodeSizeLog2 + 1 ).

The number of coded tree levels coded is equal to occtreeMaxDepthMinus1 plus 1. The value of occtreeMaxDepthMinus1 is set equal to occtree\_depth\_minus1 when trisoup\_enabled is equal to 0, otherwise when trisoup\_enabled is equal to 1 the value of occtreeMaxDepthMinus1 is set as

occtreeMaxDepthMinus1 = occtree\_depth\_minus1 - 1

In the latter case, occupancy tree coding is performed until reaching not the unit cube size but node size triSoupNodeSizeLog2 instead. The TriSoup process (9.4) is performed for each node with node size triSoupNodeSizeLog2.

When trisoup\_enabled is equal to 1, the value triSoupNodeSizeLog2 is set to  trisoup\_node\_size\_log2\_minus2 + 2 . Otherwise, when trisoup\_enabled is equal to 0, the value triSoupNodeSizeLog2 is set to 0.

occtree\_coded\_axis[ dpth ][ 𝑘 ] specifies whether (when 1) or not (when 0) a subdivision along the 𝑘-th STV axis is coded by tree nodes at depth dpth. occtree\_coded\_axis shall be used to determine the node volume dimensions in each level of the occupancy tree. When occtree\_coded\_axis[ dpth ][ 𝑘 ] is not present, it shall be inferred to be 1.

It is a requirement of bitstream conformance that:

* There shall be at least one coded axis in every tree level specified by occtree\_coded\_axis; i.e. MaxVec( occtree\_coded\_axis[ dpth ] ) == 1.
* The log2 dimensions of the root node shall be less than or equal to MaxSliceDimLog2.
* The largest log2 dimension of the root node shall be greater than  occtreeMaxDepthMinus1 + triSoupNodeSizeLog2 – 4.

occtree\_stream\_cnt\_minus1 plus 1 specifies the maximum number of entropy streams used to code the occupancy tree. When occtree\_stream\_cnt\_minus1 is greater than zero, each of the bottom occtree\_stream\_cnt\_minus1 tree levels shall be conveyed in a separate entropy stream; the parsing state shall be memorized and restored according to subclause 1.

The expression OcctreeEntropyStreamDepth is the depth of the last tree level that is coded in the first entropy stream.

OcctreeEntropyStreamDepth := occtreeMaxDepthMinus1 − occtree\_stream\_cnt\_minus1

occtree\_end\_of\_entropy\_stream is a non-coded syntax element used to specify the termination point for the arithmetic decoder at the end of an entropy stream. The syntax element has no value.

occtree\_lvl\_point\_cnt\_minus1[ dpth ] plus 1 indicates, when present, the number of points that can be partially decoded (See Annex D) from the root node to the end of the tree level at depth dpth. occtree\_lvl\_point\_cnt\_minus1[ 0 ] shall be inferred to be 0. occtree\_lvl\_point\_cnt\_minus1[ occtreeMaxDepthMinus1 ] shall be inferred to be slice\_num\_points\_minus1.

### Node dimensions per tree level

The log2 node dimensions at depth dpth are specified by the expression OccLvlNodeSizeLog2[ dpth ][ 𝑘 ]. They are derived from the list of coded axes:

* In the bottom tree level, at depth occtreeMaxDepthMinus1 + 1, the node dimensions shall be equal to the cube with the size triSoupNodeSizeLog2.
* The log2 node dimensions in any tree level shallower than the bottom tree level shall be the count of the respective coded axes, proceeding from the bottom tree level.

OccLvlNodeSizeLog2[dpth][k] := lvl ≤ occtreeMaxDepthMinus1  
 ? OccLvlNodeSizeLog2[dpth + 1][k] + occtree\_coded\_axis[dpth][k]  
 : triSoupNodeSizeLog2

### State representation

#### State variables

The occupancy tree is specified in terms of the following state variable:

* The sparse array OccNodePresent that identifies nodes present in the occupancy tree. Each element OccNodePresent[ dpth ][ ns ][ nt ][ nv ] equal to either 1 or −1 indicates the presence of a node at location ( ns, nt, nv ) and depth dpth. Elements equal to −1 identify leaf nodes that are not coded at that depth (9.2.2.4). Unset elements of OccNodePresent are inferred to be 0.

Traversal of the occupancy tree is specified in terms of the following state variables:

* The array OccNodeCnt; OccNodeCnt[ dpth ] is the cumulative count of nodes present at depth dpth.
* The array OccNodeLoc; OccNodeLoc[ dpth ][ nodeIdx ][ 𝑘 ] identifies the 𝑘-th location component of the nodeIdx-th coded node in the traversal order of the tree level at depth dpth.

#### The root node

At the start of the occupancy tree syntax structure, the arrays OccNodePresent, OccNodeLoc and OccNodeCnt are initialized to represent the root node at location ( 0, 0, 0 ); all other elements are cleared.

OccNodePresent[0][0][0][0] = 1  
OccNodeLoc[0][0][0] = OccNodeLoc[0][0][1] = OccNodeLoc[0][0][2] = 0  
OccNodeCnt[0] = 1

### Occupancy tree node coding

#### General

Subclause 9.2.6 specifies the semantics of the NodeIdx-th coded node at tree depth Dpth.

#### Syntax element semantics

occ\_single\_child equal to 1 specifies that the coded node has a single child. occ\_single\_child equal to 0 specifies that the coded node may generate multiple child nodes. When occ\_single\_child is not present, it shall be inferred to be 0.

occupancy\_idx[ 𝑘 ] specifies the 𝑘-th component of the relative child node location for the only child of the coded node.

occupancy\_bit and occupancy\_byte specify the child nodes of the coded node as neighbourhood-permuted node occupancy bitmaps. The syntax elements shall be coded as specified by 9.2.10 and 9.2.9, respectively.

occ\_dup\_point\_cnt[ 𝑖 ] plus 1 specifies the number of points represented by the 𝑖-th coded child leaf node. All points represented by a child have identical positions. When occ\_dup\_point\_cnt[ 𝑖 ] is not present in a terminal node, it is inferred to be 0.

When unique\_point\_positions\_constraint is 1, it is a requirement of bitstream conformance that occ\_dup\_point\_cnt[ 𝑖 ] shall be 0.

#### Node, parent, grandparent and child tree-level locations

The tree-level location ( Ns, Nt, Nv ) of the coded node is specified by the expression Nloc[ 𝑘 ].

Nloc[k] := OccNodeLoc[Dpth][NodeIdx][k]

Ns := Nloc[0]  
Nt := Nloc[1]  
Nv := Nloc[2]

The parent node has a location ( NsP, NtP, NvP ) in the tree level at depth Dpth − 1. It is specified by the expression NlocP[ 𝑘 ].

NlocP[k] := Dpth ? Nloc[k] >> occtree\_coded\_axis[Dpth − 1][k] : 0

NsP := NlocP[0]  
NtP := NlocP[1]  
NvP := NlocP[2]

The grandparent node has a location ( NsG, NtG, NvG ) in the tree level at depth Dpth − 2. It is specified by the expression NlocG[ 𝑘 ].

NlocG[k] := Dpth > 1 ? NlocP[k] >> occtree\_coded\_axis[Dpth − 2][k] : 0

NsG := NlocG[0]  
NtG := NlocG[1]  
NvG := NlocG[2]

The corresponding location ( NsC, NtC, NvC ) in the tree level at depth Dpth + 1 for the coded node is specified by the expression NlocC[ 𝑘 ].

NlocC[k] := Nloc[k] << AxisCoded[k]

NsC := NlocC[0]  
NtC := NlocC[1]  
NvC := NlocC[2]

#### Node size

The expressions NodeSizeLog2[ 𝑘 ] and ChildNodeSizeLog2[ 𝑘 ] specify the log2 dimensions of the coded node and its children, respectively.

NodeSizeLog2[k] := OccLvlNodeSizeLog2[Dpth][k]

ChildNodeSizeLog2[k] := OccLvlNodeSizeLog2[Dpth + 1][k]

1. When geom\_scaling\_enabled is 0, QuantizedNodeSizeLog2[ 𝑘 ] is equal to NodeSizeLog2[ 𝑘 ].

#### Whether the node is a terminal node

A node is a terminal node, as specified by the expression TerminalNode, if its children are leaf nodes, or it is a direct node.

TerminalNode := MaxVec(ChildNodeSizeLog2) == 0  
 || geom\_scaling\_enabled && MaxVec(QuantizedChildNodeSizeLog2) == 0  
 || occtree\_direct\_coding\_mode && occ\_direct\_node

#### Coded axes

A node shall only code an axis for child locations when specified by the expression AxisCoded[ 𝑘 ]. An axis is coded when:

* it is specified to be coded in the tree level by the coded axis list, and
* if geometry subtree scaling is enabled, the corresponding dimension of the quantized node size is greater than 1.

AxisCoded[k] := occtree\_coded\_axis[Dpth][k]  
 && (¬geom\_scaling\_enabled || QuantizedNodeSizeLog2[k] > 0)

The expression CodedAxisCnt is the number of axes coded by the node.

CodedAxisCnt := AxisCoded[0] + AxisCoded[1] + AxisCoded[2]

The location of child nodes along each coded axis may be constrained by planar occupancy coding (9.2.11.4). A free axis is a coded axis that is not constrained so, as specified by the expression OccFreeAxis[ 𝑘 ]. The number of free axes is specified by the expression OccFreeAxisCnt.

1. When planar occupancy coding is disabled, OccFreeAxisCnt is equal to CodedAxisCnt.

OccFreeAxis[k] := AxisCoded[k] && (¬occtree\_planar\_enabled || PlanarFreeAxis[k])

OccFreeAxisCnt := OccFreeAxis[0] + OccFreeAxis[1] + OccFreeAxis[2]

#### Limits to the number of child nodes

The number of child nodes a node can contain is constrained by the tree and node syntax.

The maximum number of child nodes is specified by the expression OccMaxChildren. Unless occ\_single\_child is 1, the limit shall be the number of child node locations that can be identified by the free axes. Otherwise, the limit shall be 1 when occ\_single\_child is 1.

OccMaxChildren := occ\_single\_child ? 1 : Exp2(OccFreeAxisCnt)

The minimum number of child nodes is specified by the expression OccMinChildren. A node shall contain at least one child node unless:

* planar occupancy coding specifies that there shall be at least two child nodes (9.2.11.3), or
* occ\_single\_child is both present and equal to 0, in which case there shall be at least two child nodes.

OccMinChildren :=  
 OccMaybeSingleChild && ¬occ\_single\_child ? 2 :  
 occtree\_planar\_enabled ? PlanarMinChildren : 1

#### Presence of occ\_single\_child

The presence of occ\_single\_child is specified by the expression OccMaybeSingleChild. It shall be present in the occupancy node syntax when all the following conditions are true:

* No nodes are present in the occupied neighbourhood pattern (9.2.7.4).
* OccupancyIsPredictable is 0 or there is at least one free axis to code child node locations.
* Planar occupancy coding does not specify that there shall be at least two child nodes (9.2.11.3).

OccMaybeSingleChild :=  
 ¬OccNeighPat && OccFreeAxisCnt > 0 && PlanarMinChildren == 1

#### Presence of occupancy\_bit and occupancy\_byte

The node occupancy bitmap shall be coded using either occupancy\_bit or occupancy\_byte when specified by the expression OccMapPresent. One of the two syntax elements shall be present when both of the following conditions are true:

* The maximum number of child nodes is greater than 1.
* The locations of child nodes are not completely prescribed by constraints on occupancy. i.e. the maximum number of child nodes is greater than the minimum number of child nodes.

OccMapPresent := OccMaxChildren > 1 && OccMinChildren != OccMaxChildren

#### Node occupancy bitmap

This subclause specifies the node occupancy bitmap by the expression OccupancyMap.

When occupancy\_bit syntax elements are present (OccMapPresent is 1), the node occupancy bitmap is specified by bitwise occupancy coding (9.2.10.2).

if (OccMapPresent && occtree\_bitwise\_coding)  
 OccupancyMap = OccBitMap

When occupancy\_byte is present (OccMapPresent is 1), the node occupancy bitmap shall be rearranged from the neighbourhood-permuted bitmap (9.2.8) coded by occupancy\_byte.

if (OccMapPresent && ¬occtree\_bitwise\_coding)  
 OccupancyMap = OccFromNpOcc(occupancy\_byte)

When constraints on occupancy require there to be a single child node, each component 𝑘 of the child location shall be specified by whichever of occ\_plane\_pos[ 𝑘 ] or occupancy\_idx[ 𝑘 ] are present, or shall be 0 if neither is present.

if (OccMaxChildren == 1 && OccMinChildren == 1) {  
 occupancyIdx[k] :=  
 OccFreeAxis[k] && occupancy\_idx[k] || ¬PlanarFreeAxis[k] && occ\_plane\_pos[k]  
  
 OccupancyMap = 1 << Morton[occupancyIdx]  
}

1. In this case, an axis cannot be both a free axis and eligible for planar occupancy coding.

When constraints on occupancy require there to be two child nodes, there shall be one child node at both permitted locations along the free axis.

if (OccMaxChildren == 2 && OccMinChildren == 2) {  
 baseIdx[k] := ¬PlanarFreeAxis[k] && occ\_plane\_pos[k]  
  
 if (OccFreeAxis[0]) OccupancyMap = 0x11 << Morton[baseIdx]  
 if (OccFreeAxis[1]) OccupancyMap = 0x05 << Morton[baseIdx]  
 if (OccFreeAxis[2]) OccupancyMap = 0x03 << Morton[baseIdx]  
}

#### Child node count

The number of child nodes is equal to the number of set bits in the node occupancy bitmap, as specified by the expression OccChildCnt.

OccChildCnt := PopCnt(OccupancyMap)

#### Insertion of non-terminal child nodes

Unless the coded node is a terminal node, its child nodes shall be inserted into the state representation of the occupancy tree and included in the traversal list of the next tree level. The node occupancy bitmap shall be scanned to enumerate the child nodes.

if (¬TerminalNode)  
 for (occBitIdx = 0; occBitIdx < 8; occBitIdx++) {  
 if (¬Bit(OccupancyMap, occBitIdx))  
 continue  
  
 cs = NsC + OccLocC[occBitIdx][0]  
 ct = NtC + OccLocC[occBitIdx][1]  
 cv = NvC + OccLocC[occBitIdx][2]  
 OccNodePresent[Dpth + 1][cs][ct][cv] = 1  
  
 childNodeIdx = OccNodeCnt[Dpth + 1]  
 OccNodeCnt[Dpth + 1]++  
  
 OccNode[Dpth + 1][childNodeIdx][0] = cs  
 OccNode[Dpth + 1][childNodeIdx][1] = ct  
 OccNode[Dpth + 1][childNodeIdx][2] = cv  
 }

#### Points represented by child leaf nodes

When the node is a non-direct terminal node, points represented by child leaf nodes shall be scaled (9.2.14.6) and appended to the output point list. The node occupancy bitmap shall be scanned to enumerate the child nodes.

1. When geometry scaling is disabled, this condition is equivalent to the child node size being equal to the unit cube.

When adjacent child occupancy contextualization is enabled, the child leaf nodes shall be inserted into the state representation of the occupancy tree for use by other nodes in the same tree level; but they shall be excluded from traversal in the next tree level. The child leaf nodes shall not be included in the occupied neighbourhood pattern for any node in the next tree level.

if (TerminalNode && ¬occ\_direct\_node)  
 for (child = 0, occBitIdx = 0; occBitIdx < 8; occBitIdx++) {  
 if (¬Bit(OccupancyMap, occBitIdx))  
 continue  
  
 cs = NsC + OccLocC[occBitIdx][0]  
 ct = NtC + OccLocC[occBitIdx][1]  
 cv = NvC + OccLocC[occBitIdx][2]  
 if (occtree\_adjacent\_child\_enabled)  
 OccNodePresent[Dpth + 1][cs][ct][cv] = −1  
  
 for (i = 0; i < occ\_dup\_point\_cnt[child] + 1; i++, PointCnt++) {  
 PointPos[PointCnt][0] = OccPosScaleK(0, cs)  
 PointPos[PointCnt][1] = OccPosScaleK(1, ct)  
 PointPos[PointCnt][2] = OccPosScaleK(2, cv)  
 }  
  
 child++  
 }

#### Definition of ChildIdx

This subclause specifies the expression ChildIdx[ pt ] that is the child index of point pt for a given node.

ChildIdx[pt] = (pt[2] & mask[2] ? 1 : 0) +   
 ((pt[1] & mask[1] ? 1 : 0) << 1) +  
 ((pt[0] & mask[0] ? 1 : 0) << 2)  
where  
 mask[k] = (NodeSizeLog2[k] != ChildNodeSizeLog2[k]) ? 1 << ChildNodeSizeLog2[k] : 0

#### Derivation of OccupancyIsPredictable

OccupancyIsPredictable specifies whether the current node satisfies conditions to enable inter prediction of occupancy of child nodes.

occupancyIsPredictable = numSiblingsMispredcted <= 5 && PredOccBitMap > 0  
where  
 numSimblingsMispredicted = PopCnt(PredOccBitMap ^ OccupancyMap)

#### Derivation of PredOccBitMap

The occupancy of the regions collocated to the child nodes of the current node is stored in the variable PredOccBitMap.

PredOccBitMap is derived as follows

PredOccBitMap1 = 0  
if(enableInterPredFromRef)  
 for(i = 0;i < 8; i++)  
 if(PredCounts1[i])  
 PredOccBitMap1 |= 1 << i  
PredOccBitMap2 = 0  
if(enableInterPredFromRef2)  
 for(i = 0;i < 8; i++)  
 if(PredCounts2[i])  
 PredOccBitMap2 |= 1 << i  
PredOccBitMap = predDir ? PredOccBitMap2 : PredOccBitMap1

#### Derivation of nStart, nEnd, nStart2 and nEnd2

When *enableInterPredFromRef* is 1, the variables nStart and nEnd for a node that is the i-th child node of its parent node specify the start and end indices of a sorted reference frame that contains points belonging to the reference frame that are collocated with the node. Let nStart\_parent denote the nStart variable of the parent node.

When the node is the root node, nStart is 0 and nEnd is the total number of points in the slice.

Otherwise, (node is not the root node), nStart and nEnd are derived as follows:

nStart = nStart\_parent  
for(j = 0; j < i; j++)  
 nStart += PredCounts[j]  
nEnd = nStart + PredCounts[i]

When *enableInterPredFromRef2* is 1, the variables nStart2 and nEnd2 for a node that is the i-th child node of its parent node specify the start and end indices of the second sorted reference frame that contains points belonging to the second reference frame that are collocated with the node. nStart2 and nEnd2 are derived from the total number of points in the slice of second reference frame or *PredCounts2*, in the same way as nStart and nEnd.

#### Derivation of PredCounts and PredCounts2

When *enableInterPredFromRef* is 1, the array PredCounts[i] denotes the start indices of the positions of the octree node in the reference frame collocated with the i-th child node.

The frame buffers RefFramePos and RefFramePos2 are set equal to PredPointCloud and SecondPredPointCloud, respectively, at the beginning of the octree slice.

First, the number of points in reference frame that are collocated with each child node, colRefPoints[i] is calculated. colRefPoints[0..7] is initialized to 0.

for(i = nStart; i < nEnd; i++) {  
 pt = RefFramePos[i]  
 colRefPoints[ChildIdx[pt]]++  
}

The array PredCounts is derived as follows:

PredCounts[0] = 0  
for(i = 1; i < 8; i++)  
 PredCounts[i] = PredCounts[i-1] + colRefPoints[i-1]

When *enableInterPredFromRef2* is 1, the array PredCounts2[i] denotes the start indices of the positions of the octree node in the second reference frame collocated with the i-th child node. PredCounts2 is derived from the number of points in second reference frame that are collocated with each child node, in the same way as PredCounts.

#### Update of RefFramePos and RefFramePos2 at the beginning of coding a node

When *enableInterPredFromRef1* is 1, the points in the reference frame buffer RefFramePos are sorted based on the child node regions.

nIdx[0] = nStart  
for(i = 1; i < 8; i++)  
 nIdx[i] = nStart + PredCounts[i]  
lastIdx = 0  
for (int i = 0; i < Radix; i++) {  
 lastIdx += PredCounts[i]  
 while (nIdx[i] != lastIdx) {  
 radix = ChildIdx[RefFramePos[nIdx[i]]]  
 Swap(RefFramePos[nIdx[i]], RefFramePos[nIdx[radix]])  
 nIdx[radix]++  
 }  
}

When *enableInterPredFromRef2* is 1, the points in the second reference frame buffer RefFramePos2 are sorted based on the child node regions.

nIdx[0] = nStart2  
for(i = 1; i < 8; i++)  
 nIdx[i] = nStart2 + PredCounts2[i]  
lastIdx = 0  
for (int i = 0; i < Radix; i++) {  
 lastIdx += PredCounts2[i]  
 while (nIdx[i] != lastIdx) {  
 radix = ChildIdx[RefFramePos2[nIdx[i]]]  
 Swap(RefFramePos2[nIdx[i]], RefFramePos2[nIdx[radix]])  
 nIdx[radix]++  
 }  
}

#### Derivation of *enableInterPredFromRef* and *enableInterPredFromRef2*

When slice\_biprediction is 1 and frame\_merge\_enabled is 0, *enableBipredDerive* is derived as follows:

enableBipredDerive = slice\_biprediction && !frame\_merge\_enabled && (nodeSizeLog2[0] >= 1) && (nodeSizeLog2[1] >= 1) && (nodeSizeLog2[2] >= 1)

The variables *enableInterPredFromRef* and *enableInterPredFromRef2* specify whether (1) or not (0) the first and second reference frames are used in inter prediction. *enableInterPredFromRef* and *enableInterPredFromRef2* are derived as:

enableInterPredFromRef = enableBipredDerive || !predDir  
enableInterPredFromRef2 = enableBipredDerive || predDir

#### Derivation of *predDir*

The variables predDir for a node that is the i-th child node of its parent node specify the inter prediction is from the first reference frame (0) or the second reference frame (1). predDir is initilized to 0 for the root node.

Let predDir\_parent denote the predDir variable of the parent node. predDir is derived as follows:

predDir = predDir\_parent  
if (enableBiPredDerive)  
 if (!PredCounts2[i])  
 predDir = 0  
 else if (!PredCounts[i])  
 predDir = 1  
 else  
 predDir = predFailureCount != predFailureCount2 ? (predFailureCount >=   
 predFailureCount2) : predDir\_parent  
 Where  
 predFailureCount = enableInterPredFromRef ? PopCnt(PredOccBitMap ^ OccupancyMap) : 0  
 predFailureCount2 = enableInterPredFromRef2 ? PopCnt(PredOccBitMap2 ^ OccupancyMap) :   
 0

### Occupied neighbourhood patterns

#### General

Coding of the node occupancy bitmap depends upon the existence and arrangement of up to six spatially adjacent tree nodes within an availability window. The occupied neighbourhood pattern for an occupancy tree node shall identify the spatial arrangement of these adjacent nodes from the 64 possible combinations. Examples of occupied neighbourhood patterns are illustrated in Figure 7.

图示

描述已自动生成

Figure 7 — Characteristic occupied neighbourhood patterns.

#### Neighbour availability

Nodes are grouped into availability windows by their spatial location within their tree level. Nodes can form part of the occupied neighbourhood pattern of adjacent nodes within the same window. Nodes shall not form part of the occupied neighbourhood pattern of any node in a different window.

The size of the availability window is specified by the expression OccAvailWinLog2[ 𝑘 ]:

* Unless occtree\_neigh\_window\_log2\_minus1 is 0, each availability window shall span 𝑛×𝑛×𝑛 node locations, 𝑛 = Exp2( occtree\_neigh\_window\_log2\_minus1 + 1 ), within the tree level. The availability windows form a contiguous grid starting from the location ( 0, 0, 0 ).
* Otherwise, the availability window for any node shall be restricted to its sibling nodes.

OccAvailWinLog2[k] := occtree\_neigh\_window\_log2\_minus1  
 + (occtree\_neigh\_window\_log2\_minus1 > 0 || Dpth > 0 && occtree\_coded\_axis[Dpth − 1][k])

1. Only the Main profile permits an element of occtree\_coded\_axis to be 0 when occtree\_neigh\_window\_log2\_minus1 is 0.

The expression OccNeighAvail[ ns ][ nt ][ nv ] specifies whether the node at location ( ns, nt, nv ) is within the same availability window as the coded node ( Ns, Nt, Nv ).

OccNeighAvail[ns][nt][nv] :=  
 (ns ^ Ns) >> OccAvailWinLog2[0] == 0  
 && (nt ^ Nt) >> OccAvailWinLog2[1] == 0  
 && (nv ^ Nv) >> OccAvailWinLog2[2] == 0

#### Presence of another coded node within the availability window

The expression OccNeigh[ ns ][ nt ][ nv ] identifies whether there exists a node with tree location ( ns, nt, nv ) and depth Dpth that is not a leaf node and is within the availability window of the coded node ( Ns, Nt, Nv ).

1. OccNodePresent[ Dpth ][ ns ][ nt ][ nv ] equal to −1 identifies a leaf node.

OccNeigh[ns][nt][nv] := OccNeighAvail[ns][nt][nv] && OccNodePresent[Dpth][ns][nt][nv] == 1

#### Occupied neighbourhood pattern

The occupied neighbourhood pattern for the coded node located at ( Ns, Nt, Nv ) in the tree level at depth Dpth is specified by the expression OccNeighPat. It is a linear combination of spatially adjacent nodes coded in the same tree level that are available and adjoin the coded node by a face. Leaf nodes shall not be included in the occupied neighbourhood pattern.

An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood pattern equal to 0.

OccNeighPat := (uN << 5) | (dN << 4) | (bN << 3) | (fN << 2) | (rN << 1) | lN  
 where  
 rN := OccNeigh[Ns + 1][Nt][Nv]  
 lN := OccNeigh[Ns − 1][Nt][Nv] && (¬occtree\_adjacent\_child\_enabled || lNadj)  
 bN := OccNeigh[Ns][Nt + 1][Nv]  
 fN := OccNeigh[Ns][Nt − 1][Nv] && (¬occtree\_adjacent\_child\_enabled || fNadj)  
 uN := OccNeigh[Ns][Nt][Nv + 1]  
 dN := OccNeigh[Ns][Nt][Nv − 1] && (¬occtree\_adjacent\_child\_enabled || dNadj)

When adjacent child contextualization is enabled (occtree\_adjacent\_child\_enabled is 1), a tree node that adjoins the left (Ns − 1), front (Nt − 1) or bottom (Nv − 1) face shall not be included in the occupied neighbourhood pattern unless it contains at least one child node that also adjoins the same face. Their inclusion is specified by the expressions lNadj, fNadj and dNadj, equivalent to the following:

lNadj = fNadj = dNadj = 0  
for (s = 0; s ≤ occtree\_coded\_axis[Dpth][0]; s++)  
 for (t = 0; t ≤ occtree\_coded\_axis[Dpth][1]; t++)  
 for (v = 0; v ≤ occtree\_coded\_axis[Dpth][2]; v++) {  
 lNadj |= OccNodePresent[Dpth + 1][NsC − 1][NtC + t][NvC + v] ≠ 0  
 fNadj |= OccNodePresent[Dpth + 1][NsC + s][NtC − 1][NvC + v] ≠ 0  
 dNadj |= OccNodePresent[Dpth + 1][NsC + s][NtC + t][NvC − 1] ≠ 0  
 }

The exclusion of a node from the occupied neighbourhood pattern due to adjacent child contextualization is illustrated in Figure 8. The child node of the left neighbour does not adjoin the left face of the coded node N resulting in its parent node being excluded from the occupied neighbourhood pattern; OccNeighPat is 2. The child node does adjoin N and its parent node is not excluded; OccNeighPat is 3.

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| N | Coded node |
|  | Child node at location 𝑚 in left neighbour |

Figure 8 — Effect of adjacent child contextualization on an occupied neighbourhood pattern. (Top) Exclusion. (Bottom) Inclusion.

#### Reduced occupied neighbourhood pattern

The occupied neighbourhood pattern shall be reduced to one of a smaller set of patterns as specified by the expression OccNeighPatR.

When occtree\_neigh\_window\_log2\_minus1 is greater than 0, the smaller set of patterns is specified by Table 14 as a mapping of spatial rotations and reflections in the arrangement of the six spatially adjacent neighbours to produce nine unique arrangements.

Otherwise, when occtree\_neigh\_window\_log2\_minus1 is 0, the smaller set of patterns is specified by Table 15 as a mapping of adjacent siblings that produces six arrangements.

OccNeighPatR := occtree\_neigh\_window\_log2\_minus1 > 0  
 ? NeighPat64to9[OccNeighPat]  
 : NeighPat64to6[OccNeighPat]

Table 14 — Reduction of occupied neighbourhood pattern 𝑗 + 𝑖 to nine patterns, NeighPat64to9[ 𝑗 + 𝑖 ]

| 𝑗 | 𝑖 | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **0** | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 3 | 1 | 3 | 3 | 4 |
| **16** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **32** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **48** | 1 | 3 | 3 | 4 | 3 | 6 | 6 | 7 | 3 | 6 | 6 | 7 | 4 | 7 | 7 | 8 |

Table 15 — Reduction of occupied neighbourhood pattern 𝑗 + 𝑖 to six patterns, NeighPat64to6[ 𝑗 + 𝑖 ]

| 𝑗 | 𝑖 | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **0** | 0 | 5 | 5 | na | 5 | 1 | 1 | na | 5 | 1 | 1 | na | na | na | na | na |
| **16** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **32** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **48** | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na |
| 1. The specification of values 5 and 7 aligns with further reductions performed in bitwise occupancy coding. | | | | | | | | | | | | | | | | |

#### Advanced occupied neighbourhood patterns

The advanced occupied neighbourhood patterns for one node located at ( *Ns*, *Nt*, *Nv* ) in the tree level at depth *Dpth* are the occupancy combinations of the decoded spatially adjacent nodes in the tree level at depth *Dpth* and neighbouring child nodes in the tree level at depth *Dpth*+1. Leaf nodes shall not be included in the advanced occupied neighbourhood patterns. For coding occupancy bit of each child node, one advanced occupied neighbourhood pattern *OccAdvNeiPati* is derived for the child node. The derived (*OccAdvNeiPat0*, *OccAdvNeiPat1*, *OccAdvNeiPat2*, *OccAdvNeiPat3*, *OccAdvNeiPat4*, *OccAdvNeiPat5*, *OccAdvNeiPat6*, *OccAdvNeiPat7*) are used to construct contextual information *CI* to be used in Dynamic OBUF according to 9.2.10.6.10.

For one child node, the expression *occL* specifies the occupancy of the 4 neighbouring child nodes neighbouring current node on the left, *occF* specifies the occupancy of the 4 neighbouring child nodes neighbouring current node on the front, and *occB* specifies the occupancy of the 4 neighbouring child nodes neighbouring current coded node on the bottom. An example of occL, occF and occB for the first child node is illustrated in Figure 9.

图片包含 游戏机, 体育

描述已自动生成

Figure 9 — occL, occF, occB for the first child node

occL, occF and occB are calculated from the occupancy of child nodes of three spatially adjacent nodes (*occLeft*, *occFront* and *occBottom*) on the left, front and bottom of current node.

occL := occLeft >> 4  
occF := ((occFront >> 2) & 3) | ((occFront >> 4) & 12)  
occB := ((occBottom >> 1) & 1) | ((occBottom >> 2) & 2) | ((occBottom >> 3) & 4) | ((occBottom >> 4) & 8

The expression *occOfLFB* specifies the occupancy of child nodes of three spatially adjacent nodes on the left, front and bottom.

occOfLFB := occLeft | occFront | occBottom

The expression edgeSets specifies the occupancy of 6 neighbouring child nodes, *fLF0*, *fLF1*, *fLB0*, *fLB1*, *fFB0* and *fFB1*. *fLF0* and *fLF1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occF*. *fLB0* and *fLB1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occB*. *fFB0* and *fFB1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occF*. An example of edgeSets the first child node is illustrated in Figure 10.

图片包含 游戏机

描述已自动生成

Figure 10 — example of edgeSets for the first child node

The expression *neighPatternRBT* specifies the occupancy of 3 spatially adjacent nodes on the right, back and top of the current node. An example of *neighPatternRBT* for the node and their order to construct *neighPatternRBT* is illustrated in Figure 11. *neighPatternRBT* is calculated by accessing occupancy OccNeighPat(9.2.7.4) of six spatially adjacent nodes of the node.

neighPatternRBT := ((neighPattern >> 3) & 4) | ((neighPattern >> 2) & 2) | (neighPattern & 1)

图片包含 矩形

描述已自动生成

Figure 11 — example of three adjacent nodes of neighPatternRBT of node N

The expression *neighPatternLFB* specifies the occupancy of 3 spatially adjacent nodes on the bottom, front and left of the current node. *neighPatternLFB* is calculated by accessing occupancy OccNeighPat(9.2.7.4) of six spatially adjacent nodes of the node.

neighPatternLFB =((neighPattern & 0b110) >> 1)| ((neighPattern & 16) >> 2)

The expression neighb20 specifies of the occupancy of 20 spatially adjacent nodes of the current node which adjoin the current node by an edge or a vertex and not a face. An example of neighb20 for the node is illustrated in Figure 12. neighb20 is calculated by accessing OccNeigh (9.2.7.3) and constructed by an order starting from node 0 to node 19.

neighb20 = 0  
for(i = 0; i < 20; i++)  
 neighb20 |= OccNeigh[Ns + LUTds[i]][Nt + LUTdt[i]][Nv + LUTdv[i]]<<i  
 where  
 LUTds[20] := {-1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1}  
 LUTdt[20] := {-1, -1, -1, 0, 0, 1, 1, 1, -1, -1, 1, 1, -1, -1, -1, 0, 0, 1, 1, 1}  
 LUTdv[20] := {-1, 0, 1, -1, 1, -1, 0, 1, -1, 1, -1, 1, -1, 0, 1, -1, 1, -1, 0, 1}

图片包含 游戏机, 钟表

描述已自动生成

Figure 12 — Example of neighbouring nodes for neighb20 of node N

##### Derivation of OccAdvNeiPat0

The expression *OccAdvNeiPat0* specifies advanced occupied neighbourhood pattern constructed to code occupancy bit of the first child node *b0*of the current node.

The expression *NN0* specifies the number of occupied child nodes in the three sets of neighbouring child nodes specified by *occL*, *occF* and *occB*.

* If *NN0* is larger than , then the occupied neighbour nodes of child node *b0* are not sparse and *OccAdvNeiPat0* comprises 19 bits. *OccAdvNeiPat0* is derived by evoking 9.2.7.6.2.
* Otherwise, the occupied neighbour nodes of child node *b0* are sparse and *OccAdvNeiPat0* comprises 16 bits. *OccAdvNeiPat0* is derived by evoking 9.2.7.6.3.

##### Derivation of OccAdvNeiPat0 with non-sparse neighbour nodes

When the occupied neighbour nodes of child node *b0* are not sparse, *OccAdvNeiPat0* is derived as follows.

* If more than one sets within the three sets specified by *occL*, *occF* and *occB* are occupied, *OccAdvNeiPat0*[0] is set as 1 and *OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] indicates which sets are occupied.

OccAdvNeiPat0[1] = occL && occF && occB ? 0 :  
 (!occF ? 0 : 1)

OccAdvNeiPat0[2] = occL && occF && occB ? 0 :  
 (!occF ? 1 : (!occL ? 0 : 1))

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 100,
    - *OccAdvNeiPat0*[3]*…OccAdvNeiPat0*[5] indicates the occupancy of the 3 neighboring child nodes sharing the same face with *b0* in the three sets specified by *occB*, *occF* and *occL* in order;
    - *OccAdvNeiPat0*[6]*OccAdvNeiPat0*[7] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the set specified by *occB*;

OccAdvNeiPat0[6] = (occB >> 2) & 1

OccAdvNeiPat0[7] = (occB >> 1) & 1

* + - *OccAdvNeiPat0*[8]*OccAdvNeiPat0*[9] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the set specified by *occF*;

OccAdvNeiPat0[8] = (occF >> 2) & 1

OccAdvNeiPat0[9] = (occF >> 1) & 1

* + - *OccAdvNeiPat0*[10]*OccAdvNeiPat0*[11] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the set specified by *occL*;

OccAdvNeiPat0[10] = (occL >> 2) & 1

OccAdvNeiPat0[11] = (occL >> 1) & 1

* + - *OccAdvNeiPat0*[12]…*OccAdvNeiPat0*[14] is set as neighPatternRBT; *OccAdvNeiPat0*[15]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[15] = neighb20[8]

OccAdvNeiPat0[16] = neighb20[3]

OccAdvNeiPat0[17] = neighb20[1]

OccAdvNeiPat0[18] = neighb20[0]

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 101,
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occB and occL*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB and occL*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occB and occL*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[1]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 110,
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occB and occF*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB and occF*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occB and occF*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[0]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

* + Otherwise (*OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 111),
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occL and occF*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occL and occF*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occL and occF*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[2]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

* Otherwise, *OccAdvNeiPat0*[0] is set as 0 and *OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] indicates which set is occupied.

OccAdvNeiPat0[1] = occL ? 0 : (occF ? 0 : 1)

OccAdvNeiPat0[2] = occL ? 0 : (occF ? 1 : 0)

* + If *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 000,
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occL*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occL*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occL*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 13.

OccAdvNeiPat0[7] = 0

OccAdvNeiPat0[8] = 0

OccAdvNeiPat0[9] = edgeSets[2]

OccAdvNeiPat0[10] = edgeSets[3]

OccAdvNeiPat0[11] =neighPatternRBT[0] || neighPatternRBT[1] || neighPatternRBT[2]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

图片包含 游戏机, 钟表

描述已自动生成

Figure 13 — Example of edge sub-set of current child node when only the set specified by occL is occupied

* + If *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 001,
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occF*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occF*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occF*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 14.

OccAdvNeiPat0[7] = edgeSets[4]

OccAdvNeiPat0[8] =edgeSets[5]

OccAdvNeiPat0[9] = neighPatternRBT[0]

OccAdvNeiPat0[10] neighPatternRBT[1]

OccAdvNeiPat0[11] = neighPatternRBT[2]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

QR 代码

描述已自动生成

Figure 14 — Example of edge sub-set of current child node when only the set specified by occF is occupied

* + Otherwise (*OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 010),
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occB*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occB*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 15.

OccAdvNeiPat0[7] = edgeSets[0]

OccAdvNeiPat0[8] =edgeSets[1]

OccAdvNeiPat0[9] = neighPatternRBT[0]

OccAdvNeiPat0[10] neighPatternRBT[1]

OccAdvNeiPat0[11] = neighPatternRBT[2]

OccAdvNeiPat0[12] = neighb20[8]

OccAdvNeiPat0[13] = neighb20[3]

OccAdvNeiPat0[14] = neighb20[1]

OccAdvNeiPat0[15] = neighb20[0]

OccAdvNeiPat0[16] = neighb20[18]

OccAdvNeiPat0[17] = neighb20[19]

OccAdvNeiPat0[18] = neighb20[11]

图片包含 游戏机

描述已自动生成

Figure 15 — Example of edge sub-set of current child node when only the set specified by occB is occupied

##### Derivation of OccAdvNeiPat0 with sparse neighbour nodes

When the occupied neighbour nodes of child node *b0* are sparse, *OccAdvNeiPat0* is derived as follows.

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occL* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 01;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occL;*
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the bottom and front of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3] = neighPatternLFB[1]

OccAdvNeiPat0[4] = neighPatternLFB[0]

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occF* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 10;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occF;*
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the bottom and left of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3] = neighPatternLFB[4]

OccAdvNeiPat0[4] = neighPatternLFB[1]

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occB* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 11;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occB*;
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the front and left of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3] = neighPatternLFB[1]

OccAdvNeiPat0[4] = neighPatternLFB[0]

* Otherwise (*NN0* is equal to 0),
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 00;
  + *OccAdvNeiPat0*[2]…*OccAdvNeiPat0*[4] indicates the occupancy of the 3 spatially adjacent nodes on the bottom, front and left of the coded node and adjoin the coded node by a face.

OccAdvNeiPat0[2] = neighPatternLFB[2]

OccAdvNeiPat0[3] = neighPatternLFB[1]

OccAdvNeiPat0[4] = neighPatternLFB[0]

* *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] is calculated as follows.

OccAdvNeiPat0[5] = neighb20[1]

OccAdvNeiPat0[6] = neighb20[3]

OccAdvNeiPat0[7] = neighb20[8]

OccAdvNeiPat0[8] = neighb20[0]

* If at least one of the 3 adjacent parent nodes on the bottom, front and left of the current node is occupied,
  + If the first child node within the set of neighbouring child nodes specified by *occOfLFB* is occupied, *OccAdvNeiPat0*[9]…*OccAdvNeiPat0*[12] is calculated as follows. *OccAdvNeiPat0*[10]…*OccAdvNeiPat0*[12] indicates the occupancy of the next 3 farther neighboring child nodes as illustrated in Figure 16.

OccAdvNeiPat0[9] = 1

OccAdvNeiPat0[10] = occBottom & 1

OccAdvNeiPat0[11] = occFront & 1

OccAdvNeiPat0[12] = occLeft & 1

图片包含 图标

描述已自动生成

Figure 16 — Example of farther neighbouring child nodes

* + Otherwise,
    - *OccAdvNeiPat0*[9] is set as 0;
    - *OccAdvNeiPat0*[10] indicates whether the 3 edge sub-sets of two farther neighboring nodes of current child node *b0* is occupied or not;

OccAdvNeiPat0[10] = !edgeSets[5] && !edgeSets[4] && !edgeSets[3] && !edgeSets[2] &&   
 !edgeSets[1] && !edgeSets[0]

* + - *OccAdvNeiPat0*[11]*OccAdvNeiPat0*[12] is calculated as follows.

OccAdvNeiPat0[11] = (occLeft & 4) || (occFront & 2) || (occBottom & 4)

OccAdvNeiPat0[12] = (occLeft & 2) || (occFront & 16) || (occBottom & 16)

* *OccAdvNeiPat0*[13]…*OccAdvNeiPat0*[15] is calculated as follows.

OccAdvNeiPat0[13] = neighb20[18]

OccAdvNeiPat0[14] = neighb20[19]

OccAdvNeiPat0[15] = neighb20[11]

* Otherwise, *OccAdvNeiPat0*[9]…*OccAdvNeiPat0*[15] is calculated as follows.

OccAdvNeiPat0[9] = !(edgeSets[5]edgeSets[4])

OccAdvNeiPat0[10] = !(edgeSets[3]edgeSets[2])

OccAdvNeiPat0[11] = !(edgeSets[1]edgeSets[0])

OccAdvNeiPat0[12] = 0

OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] =  
 neighb20[18]neighb20[19]neighb20[11]

##### Derivation of OccAdvNeiPat1

The expression *OccAdvNeiPat1* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of second child node *b1*of the current node.

The two neighbouring child nodes sets specified by *occL* and *occF* for coding the second child node *b1*is illustrated in Figure 17.

图片包含 游戏机, 建筑, 窗户

描述已自动生成

Figure 17 — Two sets of neighboring child nodes for b1

*OccAdvNeiPat1* is derived as follows.

* *OccAdvNeiPat1*[0] is set as the occupancy of the first child node *b0*.
* If the nerighouring child nodes set specified by *occF* is occupied, the occupied neighbour nodes of child node *b1* are not sparse.

OccAdvNeiPat1 = (b0 << 18) | (!(occF & 2) << 17) | (!occL << 16) | (occL ? AdvNeiFL :  
 AdvNeiFNL) | neighb20[11] << 2 | neighb20[16] << 1 | neighb20[19]  
 where  
 AdvNeiFL := ((occL & 2) << 15) | ((neighPatternRBT & 4) << 14) | (!(occF & 1)  
 << 13) | (!(occF & 8) << 12) | (!(occL & 1) << 11) | (!(occL & 8) << 10) |  
 (!(occF & 4) << 9) | (!(occL & 4) << 8) | ((neighPatternRBT & 1) << 7) |  
 neighb20[9] << 6 | neighb20[4] << 5 | neighb20[1] << 4 | neighb20[2] << 3  
 AdvNeiFNL := (!(neighPatternRBT & 4) << 15) | (!(occF & 1) << 14) | (!(occF &  
 8) << 13) | (!(occF & 4) << 12) | neighb20[9] << 11 | neighb20[4] << 10 |  
 neighb20[1] << 9 | neighb20[2] << 8 | (!(occBottom & 2) << 7) | (!(occFront &  
 2) << 6) | (!(occLeft & 2) << 5) | ((neighPatternRBT & 3) << 3)

* Otherwise, the occupied neighbour nodes of child node *b1* are sparse.

OccAdvNeiPat1 = (b0 << 18) | (!(occL & 2) << 17) | ((neighPatternRBT & 4) << 16) | (!(occL  
 & 1) << 15) | (!(occL & 8) << 14) | (!(occL & 4) << 13) | ((neighPatternRBT  
 & 1) << 12) | neighb20[1] << 11 | neighb20[4] << 10 | neighb20[9] << 9 | neighb20[2]  
 << 8 | (occOfLFB & 2 ? AdvBFL : AdvELF) | (!(occB) << 3) | neighb20[11] << 2 |  
 neighb20[16] << 1 | neighb20[19]  
 where  
 AdvBFL := (1 << 7) | (!(occBottom & 2) << 6) | (!(occFront & 2) << 5) |   
 ((occLeft & 2) << 4)  
 AdvELF := (!(edgeSets & 53) << 6) | (((occLeft & 8) || (occFront & 32)) << 5) |  
 (((occLeft & 1) || (occFront & 1)) << 4)

##### Derivation of OccAdvNeiPat2

The expression *OccAdvNeiPat2* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the third child node *b2*of the current node. *OccAdvNeiPat2* is derived as follows.

* *OccAdvNeiPat2*[0] is set as the occupancy of the first child node *b0*.
* If the nerighouring child nodes set specified by *occB* is occupied, the occupied neighbour nodes of child node *b2* are not sparse.

OccAdvNeiPat2 = (b0 << 18) | (!(occB & 2) << 17) | (!occL << 16) | (occL ? AdvNeiBL :  
 AdvNeiBNL) | neighb20[0] << 3 | neighb20[18] << 2 | neighb20[19] << 1 | neighb20[11]  
 where  
 AdvNeiBL := (!(occL & 4) << 15) | (!(neighPatternRBT & 2) << 14) | (!b1 << 13)  
 | (!(occB & 8) << 12) | (!(occL & 8) << 11) | (!(occL & 1) << 10) | (!(occB &  
 1) << 9) | neighb20[10] << 8 | neighb20[6] << 7 | neighb20[3] << 6 | (!(occB &  
 4) << 5) | (!(occL & 2) << 4)  
 AdvNeiBNL := (!(neighPatternRBT & 4) << 15) | (!(occF & 1) << 14) | (!(occF &  
 8) << 13) | (!(occF & 4) << 12) | neighb20[9] << 11 | neighb20[4] << 10 |  
 neighb20[1] << 9 | neighb20[2] << 8 | (!(occBottom & 2) << 7) | (!(occFront &  
 2) << 6) | (!(occLeft & 2) << 5) | ((neighPatternRBT & 3) << 3)

* Otherwise, the occupied neighbour nodes of child node *b2* are sparse.

OccAdvNeiPat2 = (b0 << 18) | (!(occL & 4) << 17) | (!(neighPatternRBT & 2) << 16) |  
 (!(b0) << 15) | (!(occL & 8) << 14) | (!(occL & 1) << 13) | (!(occL & 2) << 12) |  
 neighb20[3] << 11 | neighb20[6] << 10 | neighb20[10] << 9 | neighb20[5] << 8 |  
 (occOfLFB & 4 ? AdvNeiLBF : AdvNeiLBE)  
 where  
 AdvNeiLBF := (1 << 7) | (!(occLeft & 4) << 6) | (!(occBottom & 4) << 5) |  
 (!(occFront & 4) << 4)  
 AdvNeiLBE := (((occLeft & 1) || (occBottom & 1)) << 6) | (((occLeft & 8) ||  
 (occBottom & 64)) << 5) | (!(edgeSets & 3) << 4)

##### Derivation of OccAdvNeiPat3

The expression *OccAdvNeiPat3* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the fourth child node *b3*of the current node. The expression *NN3* specifies the number of occupied child nodes in the set of neighbouring child nodes specified by *occL* and the set of coded child nodes (*b0*, *b1*, *b2*). *OccAdvNeiPat3* is derived as follows.

* If *NN3* is larger than , then the occupied neighbour nodes of child node *b3* are not sparse.

OccAdvNeiPat3 = (!b2 << 16) | (!(b1) << 15) | (!(occL & 8) << 14) | (neighPatternRBT  
 << 11) | (!b0 << 10) | (!(occL & 4) << 9) | (!(occL & 2) << 8) | (occL & 1) << |  
 neighb20[11] << 6 | neighb20[6] << 5 | neighb20[4] << 4 | neighb20[0] << 3 |  
 neighb20[16] << 2 | neighb20[19] << 1 | neighb20[18]

* Otherwise, the occupied neighbour nodes of child node *b3* are sparse.

OccAdvNeiPat3 = (!((b2 << 2) | (b1 << 1) | b0) << 17) | (((b2 << 2) | (b1 << 1) |  
 b0) ? AdvNei3Occp : AdvNei3L) | ((neighPatternRBT & 6) << 13) | neighb20[4] << 12 |  
 neighb20[6] << 11 | neighb20[11] << 10 | neighb20[7] << 9 | (occOfLFB & 8 ?  
 AvdNei3BFL : AvdNei3LE) | (!occB << 4) | (!occF << 3) | neighb20[18] << 2 |  
 neighb20[19] << 1 | neighb20[16]  
 where  
 AdvNei3Occp := (!!((b2 << 2) | (b1 << 1) | b0) + !!((b2 << 1) | b1) + !!b2)  
 << 15  
 AdvNei3L := (!!(occL >> 1) + !!(occL >> 2) + !!(occL >> 3)) << 15  
 AvdNei3BFL := (1 << 8) | (!(occBottom & 8) << 7) | (!(occFront & 8) << 6) |  
 (!(occLeft & 8) << 5)  
 AvdNei3LE := ((occLeft & 6) << 5) | (!(edgeSets & 50) << 5)

##### Derivation of OccAdvNeiPat4

The expression *OccAdvNeiPat4* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the fifth child node *b4*of the current node.

The expression *occLeftChilds* specifies the set of the 4 child nodes *b0*, *b1*, *b2* and *b3*. *occRightAvailChilds* specifies the set of another 3 child nodes *b4*, *b5* and *b6*.

occLeftChilds = b3 << 3 | b2 << 2 | b1 << 1 | b0

occRightAvailChilds = b6 << 2 | b5 << 1 | b4

The expression *NN4* specifies the number of occupied child nodes in the sets of neighbouring child nodes specified by *occL*, *occB* and *occLeftChilds*. *OccAdvNeiPat4* is derived as follows.

* If *NN4* is larger than , then the occupied neighbour nodes of child node *b4* are not sparse.
  + The expression *NLFB* specifies the number NLFB of occupied sets among the sets specified by *occF* and *occB*, *occLeftChilds*.

NLFB = !!occLeftChilds + !!occF + !!occB

* + *OccAdvNeiPat4* is derived as follows.

OccAdvNeiPat4 = 0  
OccAdvNeiPat4 |= (NLFB == 3 ? AdvNei4NLFB3 : NLFB == 2 ? AdvNei4NLFB2 :  
 AdvNei4NLFB1)  
 where  
 AdvNei4NLFB3 := (8 << 15) | (!(occB & 4) << 17) | (!(occF & 4) << 16) |  
 ((occLeftChilds & 1) << 15) | (!(neighPatternRBT & 1) << 14) | (!(occB &  
 1) << 13) | (!(occB & 8) << 12) | (!(occF & 1) << 11) | (!(occF & 8) <<  
 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8) | (!(occB  
 & 2) << 7) | (!(occF & 2) << 6) | (!(neighPatternRBT >> 1) << 4) |  
 neighb20[15] << 3 | neighb20[13] << 2 | neighb20[8] << 2 | neighb20[12]  
 AdvNei4NLFB2 := (occLeftChilds && occB ? AdvNei4BChild : occF && occB ?  
 AdvNei4FB : AdvNei4Lchild) | neighb20[15] << 5 | neighb20[13] << 4 |  
 neighb20[8] << 3 | neighb20[12] << 2 | neighb20[16] << 1 | neighb20[18]  
 AdvNei4NLFB1 := (occLeftChilds ? AdvNei4Child : occF ? AdvNei4F : AdvNei4B)  
 | ((neighPatternRBT >> 1) << 6) | neighb20[15] << 5 | neighb20[13] << 4 |  
 neighb20[8] << 3 | neighb20[12] << 2 | neighb20[16] << 1 | neighb20[18]  
 AdvNei4BChild := (4 << 15) | (!(occB & 4) << 14) | (!(occLeftChilds & 1) <<  
 13) | (!(neighPatternRBT & 1) << 12) | (!(occB & 1) << 11) | (!(occB & 8)  
 << 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8)) |  
 (!(occB & 2) << 7) | (!(occLeftChilds & 8) << 6)  
 AdvNei4FB := (5 << 15) | (!(occB & 4) << 14) | (!(occF & 4) << 13) |  
 (!(neighPatternRBT & 1) << 12) | (!(occB & 1) << 11) | (!(occB & 8) << 10)  
 | (!(occF & 1) << 9) | (!(occF & 8) << 8) | (!(occB & 2) << 7) | (!(occF &  
 2) << 6)  
 AdvNei4Lchild := (6 << 15) | (!(occF & 4) << 14) | (!(occLeftChilds & 1) <<  
 13) | (!(neighPatternRBT & 1) << 12) | (!(occF & 1) << 11) | (!(occF & 8)  
 << 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8) |  
 (!(occF & 2) << 7) | (!(occLeftChilds & 8) << 6)  
 AdvNei4Child := (0 << 15) | ((occLeftChilds & 1) << 14) |  
 (!(neighPatternRBT & 1) << 13) | ((occLeftChilds & 6) << 12) |  
 (!(occLeftChilds & 8) << 10) | ((edgeSets & 12) << 8)

* Otherwise, the occupied neighbour nodes of child node *b4* are sparse. *OccAdvNeiPat4* is derived as follows.

OccAdvNeiPat4=0

if (NN4 <= 1){  
 if (NN4 == 1){  
 OccAdvNeiPat4 |= occLeftChilds ? ((1 << 14) | (!(occLeftChilds & 1) << 13) |  
 (!(neighPatternLFB & 4) << 12) | (!(neighPatternLFB & 2) << 11)) : occF ? ((2  
 << 14) | (!(occF & 1) << 13) | (!(neighPatternLFB & 4) << 12) |  
 (!(neighPatternLFB & 1) << 11)) : ((3 << 14) | (!(occB & 1) << 13) |  
 (!(neighPatternLFB & 2) << 12) | (!(neighPatternLFB & 1) << 11)  
 }  
 else {  
 OccAdvNeiPat4 |= (0 << 14) | (neighPatternLFB << 11)  
 }  
 OccAdvNeiPat4 |= neighb20[8] << 10 | neighb20[13] << 9 | neighb20[15] << 8  
 | neighb20[12] << 7  
 OccAdvNeiPat4 |= (neighPatternLFB ? AdvNei4LFB : AdvNei4ELFB) | neighb20[16] << 2  
 | neighb20[18] << 1 | neighb20[19] << 0

}  
 where  
 AdvNei4ELFB := (!(edgeSets & 48) << 6) | (!(edgeSets & 12) << 5) | (!(edgeSets &  
 3) << 4)  
 AdvNei4LFB := (occOfLFB & 16) ? AdvNei4LFBfar : AdvNei4LFBfarer  
 AdvNei4LFBfar := (1 << 6) | (!(occBottom & 16) << 5) | (!(occFront & 16) << 4) |  
 (!(occLeft & 16) << 3)  
 AdvNei4LFBfarer := (!edgeSets << 5) | (((occLeft & 64) || (occFront & 8) ||  
 (occBottom & 8)) << 4) | (((occLeft & 32) || (occFront & 64) || (occBottom & 32))  
 << 3)

##### Derivation of OccAdvNeiPat5

The expression *OccAdvNeiPat5* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the sixth child node *b5*of the current node. *OccAdvNeiPat5* is derived as follows.

* If the nerighouring child nodes set specified by *occF* is occupied, the occupied neighbour nodes of child node *b5* are not sparse.

OccAdvNeiPat5 = (b4 << 18) | (!(occF & 8) << 17) | (!occLeftChilds << 16) |  
 (occLeftChilds ? AdvNei5occFChilds : AdvNei5fartherBFL) | neighb20[18] << 2 |  
 neighb20[19] << 1 | neighb20[11]  
 where  
 AdvNei5occFChilds := (!(occLeftChilds & 2) << 15) | (!(neighPatternRBT & 4) <<  
 14) | (!(neighPatternRBT & 1) << 13) | (!(occF & 2) << 12) | (!(occF & 4) <<  
 11) | (!(occLeftChilds & 1) << 10) | (!(occLeftChilds & 8) << 9) | (!(occF &  
 1) << 8) | (!(occLeftChilds & 4) << 7) | neighb20[16] << 6 | neighb20[13] << 5  
 | neighb20[9] << 4 | neighb20[14]   
 AdvNei5fartherBFL := (!(neighPatternRBT & 4) << 15) | (!(neighPatternRBT & 1)   
 << 14) | (!(occF & 2) << 13) | (!(occF & 4) << 12) | (!(occF & 1) << 11) |  
 neighb20[16] << 10 | neighb20[13] << 9 | neighb20[9] << 8 | neighb20[14] << 7  
 | (!(occBottom & 32) << 6) | (!(occFront & 32) << 5) | (!(occLeft & 32) << 4)  
 | (!(neighPatternRBT & 2) << 3)

* Otherwise, the occupied neighbour nodes of child node *b5* are sparse.

OccAdvNeiPat5 = (!b4 << 18) | (!(occLeftChilds & 2) << 17) | (!(neighPatternRBT & 4)  
 << 16) | (!(neighPatternRBT & 1) << 15) | (!(occLeftChilds & 1) << 14) |  
 (!(occLeftChilds & 8) << 13) | (!(occL & 4) << 12) | neighb20[9] << 11 |neighb20[13]  
 << 10 |neighb20[16] << 9 | neighb20[14] << 8 | (occOfLFB & 32 ? AdvNei5BFL :  
 AdvNei5EFL) | (!occB << 3) | neighb20[18] << 2 | neighb20[19] << 1 | neighb20[11]

where

AdvNei5BFL := (1 << 7) | (!(occBottom & 32 ) << 6) | (!(occFront & 32) << 5) |  
 (!(occLeft & 32) << 4)  
 AdvNei5EFL := (!(edgeSets & 60) << 6) | (((occLeft & 128) || (occFront & 2)) <<  
 5) | (((occLeft & 16) || (occFront & 16)) << 4)

##### Derivation of OccAdvNeiPat6

The expression *OccAdvNeiPat6* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the seventh child node *b6*of the current node. *OccAdvNeiPat6* is derived as follows.

* If the nerighouring child nodes set specified by *occB* is occupied, the occupied neighbour nodes of child node *b6* are not sparse.

OccAdvNeiPat6 = (b4 << 18) | (!(occB & 8) << 17) | (!occLeftChilds << 16) |  
 (occLeftChilds ? AdvNei6occBChilds : AdvNei6occBLFB)  
 where  
 AdvNei6occBChilds := (!(occLeftChilds & 4) << 15) | (!(neighPatternRBT & 1) << 14) |  
 (!(neighPatternRBT & 2) << 13) | (!b5 << 12) | (!(occB & 2) << 11) |  
 (!(occLeftChilds & 1) << 10) | (!(occLeftChilds & 8) << 9) | (!(occB & 4) << 8) |  
 neighb20[18] << 7 | neighb20[15] << 6 | neighb20[10] << 5 | (!(occB & 1) << 4) |  
 (!(occLeftChilds & 2) << 3) | neighb20[17] << 2 | neighb20[0] << 1 | neighb20[11]  
 AdvNei6occBLFB := (!(neighPatternRBT & 4) << 15) | (!(neighPatternRBT & 1) << 14) |  
 (!(occF & 2) << 13) | (!(occF & 4) << 12) | (!(occF & 1) << 11) |  
 neighb20[16] << 10 | neighb20[13] << 9 | neighb20[9] << 8 | neighb20[14] << 7  
 | (!(occBottom & 32) << 6) | (!(occFront & 32) << 5) | (!(occLeft & 32) << 4)  
 | (!(neighPatternRBT & 2) << 3)

* Otherwise, the occupied neighbour nodes of child node *b6* are sparse.

OccAdvNeiPat6 = (!b4 << 18) | (!(occLeftChilds & 4) << 17) | (!(neighPatternRBT & 1) <<  
 16) | (!b5 << 15) | (!(occLeftChilds & 8) << 14) | (!(occLeftChilds & 1) << 13) |  
 (!(occLeftChilds & 2) << 12) | neighb20[17] << 11 | neighb20[18] << 10 | neighb20[15] <<  
 9 | neighb20[10] << 8 | (occOfLFB & 64 ? AdvNei6LBF : AdvNei6LBFE) | (!occF << 3) |  
 neighb20[19] << 2 | neighb20[16] << 1 | neighb20[11]

where  
 AdvNei6LBF := (1 << 7) | (!(occLeft & 64) << 6) | (!(occBottom & 64) << 5) |  
 (!(occFront & 64) << 4)  
 AdvNei6LBFE := (((occLeft & 1) || (occBottom & 1)) << 6) | (((occLeft & 8) ||  
 (occBottom & 64)) << 5) | (!(edgeSets & 3) << 4)

##### Derivation of OccAdvNeiPat7

The expression *OccAdvNeiPat7* specifies advanced occupied neighbourhood pattern constructed to code the occupancy bit of the eighth child node *b7*of the current node. The expression *NN7* specifies the number of occupied child nodes in the set of neighbouring child nodes specified by *occLeftChilds* and *occRightAvailChilds*. *OccAdvNeiPat7* is derived as follows.

* If *NN7* is larger than , then the occupied neighbour nodes of child node *b7* are not sparse.

OccAdvNeiPat7 = (!b6 << 16) | (!b5 << 15) | (!(occLeftChilds & 8) << 14) |  
 (neighPatternRBT << 11) | (!b4 << 10) | neighb20[11] << 9 | (!(occLeftChilds & 4) << 8)  
 | neighb20[16] << 7 | (!(occLeftChilds & 2) << 6) | neighb20[18] << 5 | ((occLeftChilds  
 & 1) << 4) | neighb20[19] << 3 | neighb20[0] << 2 | neighb20[17] << 1 | neighb20[10]

* Otherwise, the occupied neighbour nodes of child node *b7* are sparse.

OccAdvNeiPat7 = (!occup << 17) | (occup ? AdvNei7occRight : AdvNei7occLeft) |  
 (!(neighPatternRBT & 4) << 13) | neighb20[11] << 12 | neighb20[16] << 11 | neighb20[18]  
 << 10 | neighb20[19] << 9 | (occOfLFB & 128 ? AdvNei7LFB : AdvNei7LoccFoccB) | (!occB <<  
 4) | (!occF << 3) | neighb20[7] << 2 | neighb20[17] << 1 | neighb20[10]  
 where  
 occup = occRightAvailChilds & 7  
 AdvNei7occRight := ((!!occup + !!(occup >> 1) + !!(occup >> 2)) << 15) |  
 (!(neighPatternRBT & 2) << 14)  
 AdvNei7occLeft := (!!(occLeftChilds >> 1) + !!(occLeftChilds >> 2)  
 + !!(occLeftChilds >> 3) << 15) | (!(neighPatternRBT & 1) << 14)  
 AdvNei7LFB := (1 << 8) | (!(occLeft & 128) << 7) | (!(occFront & 128) << 6)  
 | (!(occBottom & 128) << 5)  
 AdvNei7LoccFoccB := ((occLeft & 96) << 1) | (((occF & 3) || (occB & 6)) << 5)

### Neighbourhood-permuted node occupancy bitmap

The neighbourhood-permuted node occupancy bitmap is a rearrangement of the bits forming the node occupancy bitmap. It is used in the coding of occupancy\_byte and occupancy\_bit. The permutation shall be selected according to the occupied neighbourhood pattern.

The permutations for every occupied neighbourhood pattern are specified by Table 16. Each entry is a base eight value with digits numbered from right to left. The 𝑖-th digit is the bit position in the node occupancy bitmap of the 𝑖-th bit in the neighbourhood-permuted bitmap as specified by the expression OccBitIdxFromNpBit[ 𝑖 ].

OccBitIdxFromNpBit[i] := (OccArrangement[OccNeighPat] >> i × 3) & 7

The expression OccFromNpOcc[ npocc ] is the node occupancy bitmap derived from the neighbourhood-permuted node occupancy bitmap npocc.

OccFromNpOcc[npocc] :=  
 OccFromNpOcc = 0  
 for (i = 0; i < 8; i++)  
 OccFromNpOcc |= Bit(npocc, i) << OccBitIdxFromNpBit[i]

Two example derivations of OccupancyMap from a single occupancy\_byte by OccFromNpOcc are illustrated in Figure 18. Each derivation uses a different occupied neighbourhood pattern, OccNeighPat. Bit of occupancy\_byte is permuted to bit of OccupancyMap when OccNeighPat is 17; in this case, OccBitIdxFromNpBit[ 4 ] would be 1.

1. occupancy\_bit codes the bits of the neighbourhood-permuted node occupancy bitmap in a different order. i.e. occupancy\_bit[ 𝑖 ] does not correspond to bit of occupancy\_byte.

图示, 工程绘图

描述已自动生成

Figure 18 — Example relationships between the node occupancy bitmap OccupancyMap and neighbourhood-permuted node occupancy bitmap as coded by occupancy\_byte.

Table 16 — Arrangements for neighbourhood-permuted node occupancy bitmaps by occupied neighbourhood pattern as OccArrangement[ 𝑖 + 𝑗 ]

| 𝑖 | 𝑗 | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| **0** | 765432108 | 103254768 | 765432108 | 765432108 | 541076328 | 103254768 |
| **6** | 541076328 | 327610548 | 327610548 | 327610548 | 765432108 | 541076328 |
| **12** | 327610548 | 765432108 | 103254768 | 765432108 | 260437158 | 203164758 |
| **18** | 465702138 | 571346028 | 041526378 | 012345678 | 450167238 | 236701458 |
| **24** | 627340518 | 236701458 | 674523018 | 450167238 | 735162408 | 674523018 |
| **30** | 012345678 | 674523018 | 371526048 | 312075648 | 574613028 | 460257138 |
| **36** | 150437268 | 103254768 | 541076328 | 327610548 | 736251408 | 327610548 |
| **42** | 765432108 | 541076328 | 624073518 | 765432108 | 103254768 | 765432108 |
| **48** | 371526048 | 647520318 | 021346578 | 574613028 | 263704158 | 574613028 |
| **54** | 736251408 | 135702468 | 405162738 | 150437268 | 312075648 | 753164208 |
| **60** | 736251408 | 371526048 | 517340628 | 765432108 |  | |

### Dictionary coding of occupancy\_byte

#### General

The occupancy\_byte syntax element shall be coded as symbols by one of nine instances of this dictionary codec. Coding shall proceed according to the syntax and semantics of the occupancy\_byte\_symbol syntax structure.

Each dictionary instance comprises a list of thirty-two most probable symbols (occupancy\_byte values), a list of sixteen recently coded symbols, a histogram of symbol counts and state variables used to control updates to the dictionary state.

#### Syntax of a dictionary coded symbol

|  |  |
| --- | --- |
| occupancy\_byte\_symbol( ) { | Descriptor |
| occ\_histogram\_hit | ae(v) |
| if( occ\_histogram\_hit ) |  |
| occ\_histogram\_index | ae(v) |
| else { |  |
| occ\_recent\_hit | ae(v) |
| if( occ\_recent\_hit ) |  |
| occ\_recent\_index | ae(v) |
| else |  |
| occ\_symbol\_escape | ae(v) |
| } |  |
| } |  |

#### Syntax element semantics of a dictionary coded symbol

occ\_histogram\_hit specifies whether (when 1) or not (when 0) the coded symbol is present in the list of most probable symbols.

occ\_histogram\_index specifies the index of the coded symbol in the list of most probable symbols.

occ\_recent\_hit specifies whether (when 1) or not (when 0) the coded symbol is present in the most recently coded symbol list. When occ\_recent\_hit is not present, it shall be inferred to be 0.

occ\_recent\_index specifies the index of the coded symbol in the most recently coded symbol list.

occ\_symbol\_escape specifies the value of the decoded symbol when occ\_histogram\_hit and occ\_recent\_hit are both 0.

#### State variables

The dictionary codec is specified in terms of the following state variables; the index dictIdx identifies an instance of the dictionary codec:

* A 9×256 element array DictsHistogram of symbol occurrence histograms per dictionary instance; DictsHistogram[ dictIdx ][ sym ] is the cumulative count for the symbol sym.
* A 9×32 element array DictsMostProb of thirty-two most probable symbols per dictionary instance; DictsMostProb[ dictIdx ][ 𝑖 ] is the 𝑖-th most probable symbol.
* A 9×16 element array DictsRecent of sixteen recently coded symbols per dictionary instance; DictsRecent[ dictIdx ][ 𝑖 ] is a recently coded symbol that was not coded using the most probable symbols list.
* A 9 element array DictsMostProbAge; DictsMostProbAge[ dictIdx ] is the count of symbols since the last generation of the dictionary's most probable symbol list.
* A 9 element array DictsMostProbMaxAge; DictsMostProbMaxAge[ dictIdx ] is the maximum allowed age in symbols of the dictionary's most probable symbol list.
* A 9 element array DictsNextEvictIdx; DictsNextEvictIdx[ dictIdx ] is the index of the next element to be evicted from the dictionary's DictsRecent array.

#### Initial state

The dictionary state shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed by the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), the dictionary state variables shall be initialized:

* Elements of DictsMostProb shall be initialized according to Table 17.
* Elements of DictsHistogram shall be set to 1 if they identify a symbol present in the corresponding most probable symbol sub-array. i.e. DictsHistogram[ dictIdx ][ 𝑖 ] = 1 if 𝑖 ∈ DictsMostProb[ dictIdx ]. All other elements shall be set to 0.
* Elements of DictsRecent, DictsRecent[ dictIdx ][ 𝑖 ], shall be set to 𝑖.
* Elements of DictsNextEvictIdx and DictsMostProbAge shall be set to 0.
* Elements of DictsMostProbMaxAge shall be set to 16.

if (¬slice\_entropy\_continuation && ¬slice\_inter \_entropy\_continuation) {  
 … /\* Initialize DictsMostProb using Table 17 \*/  
  
 for (dictIdx = 0; dictIdx < 9; dictIdx++) {  
 for (i = 0; i < 16; i++)  
 DictsRecent[dictIdx][i] = i  
  
 for (i = 0; i < 32; i++) {  
 symbol = DictsMostProb[dictIdx][i]  
 DictsHistogram[dictIdx][symbol] = 1  
 }  
  
 DictsNextEvictIdx[dictIdx] = 0  
 DictsMostProbAge[dictIdx] = 0  
 DictsMostProbMaxAge[dictIdx] = 16  
 }  
}

Table 17 — Initial values of DictsMostProb[ dictIdx ][ 𝑖 ]

| 𝑖 | dictIdx | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **0** | 5 | 85 | 128 | 64 | 85 | 16 | 170 | 170 | 255 |
| **1** | 17 | 255 | 32 | 128 | 170 | 64 | 10 | 255 | 223 |
| **2** | 34 | 170 | 64 | 192 | 255 | 17 | 42 | 128 | 239 |
| **3** | 68 | 64 | 16 | 204 | 119 | 80 | 8 | 160 | 251 |
| **4** | 160 | 80 | 192 | 136 | 127 | 128 | 138 | 136 | 127 |
| **5** | 136 | 252 | 80 | 68 | 254 | 68 | 15 | 168 | 247 |
| **6** | 12 | 223 | 160 | 170 | 87 | 32 | 255 | 204 | 119 |
| **7** | 80 | 84 | 48 | 200 | 223 | 85 | 2 | 240 | 253 |
| **8** | 192 | 117 | 68 | 85 | 95 | 81 | 14 | 250 | 63 |
| **9** | 21 | 68 | 8 | 196 | 117 | 84 | 136 | 192 | 191 |
| **10** | 10 | 247 | 136 | 255 | 245 | 192 | 11 | 238 | 221 |
| **11** | 48 | 221 | 176 | 4 | 213 | 48 | 175 | 162 | 254 |
| **12** | 3 | 4 | 2 | 8 | 247 | 51 | 32 | 234 | 238 |
| **13** | 170 | 192 | 240 | 80 | 93 | 4 | 238 | 223 | 95 |
| **14** | 168 | 128 | 144 | 160 | 234 | 34 | 47 | 138 | 175 |
| **15** | 162 | 174 | 17 | 240 | 69 | 240 | 191 | 254 | 240 |
| **16** | 204 | 253 | 208 | 208 | 238 | 1 | 34 | 10 | 85 |
| **17** | 85 | 204 | 224 | 76 | 21 | 136 | 239 | 186 | 187 |
| **18** | 14 | 240 | 112 | 221 | 221 | 170 | 245 | 8 | 244 |
| **19** | 81 | 69 | 19 | 140 | 191 | 255 | 174 | 251 | 250 |
| **20** | 35 | 127 | 255 | 244 | 253 | 204 | 3 | 2 | 170 |
| **21** | 69 | 213 | 85 | 72 | 187 | 196 | 63 | 127 | 245 |
| **22** | 84 | 5 | 51 | 93 | 16 | 160 | 95 | 125 | 117 |
| **23** | 176 | 119 | 170 | 168 | 251 | 12 | 223 | 247 | 34 |
| **24** | 51 | 238 | 84 | 250 | 171 | 208 | 253 | 85 | 126 |
| **25** | 65 | 175 | 162 | 32 | 17 | 69 | 168 | 171 | 51 |
| **26** | 138 | 160 | 238 | 252 | 5 | 191 | 142 | 32 | 93 |
| **27** | 200 | 87 | 204 | 187 | 174 | 119 | 246 | 197 | 243 |
| **28** | 212 | 136 | 1 | 223 | 125 | 21 | 206 | 221 | 207 |
| **29** | 11 | 16 | 76 | 238 | 239 | 95 | 13 | 87 | 234 |
| **30** | 50 | 244 | 138 | 243 | 12 | 2 | 162 | 42 | 59 |
| **31** | 15 | 23 | 187 | 84 | 241 | 206 | 250 | 239 | 236 |

#### Selection of a dictionary instance

A dictionary instance shall be selected for each coded occupancy\_byte syntax element according to the reduced occupied neighbourhood pattern, OccNeighPatR.

The following expressions are aliases used in the specification of operations on the selected dictionary instance:

OccDictHistogram[i] := DictsHistogram[OccNeighPatR][i]  
  
OccDictMostProb[i] := DictsMostProb[OccNeighPatR][i]  
  
OccDictRecent[i] := DictsRecent[OccNeighPatR][i]  
  
OccDictMostProbAge := DictsMostProbAge[OccNeighPatR]  
  
OccDictMostProbMaxAge := DictsMostProbMaxAge[OccNeighPatR]  
  
OccDictNextEvictIdx := DictsNextEvictIdx[OccNeighPatR]

#### The value for occupancy\_byte

The decoded value of the syntax element shall be:

* when occ\_histogram\_hit is 1: OccDictMostProb[ occ\_histogram\_index ];
* when occ\_recent\_hit is 1: OccDictRecent[ occ\_recent\_index ];
* otherwise (neither occ\_histogram\_hit nor occ\_recent\_hit is 1): occ\_symbol\_escape.

#### Update of dictionary state after each coded symbol

##### List of most recently coded symbols

After each coded occupancy\_byte syntax element when occ\_histogram\_hit is 0, the syntax element value shall be used to update the list of most recently coded symbols.

If the syntax element value is already present in the list, its position in the list shall be exchanged with the symbol scheduled to be evicted (at index OccDictNextEvictIdx).

for (i = 0; i < 16; i++)  
 if (OccDictRecent[i] == occupancy\_byte) {  
 OccDictRecent[i] = OccDictRecent[OccDictNextEvictIdx]  
 break  
 }

The syntax element value shall be inserted into the list, replacing the symbol at index OccDictNextEvictIdx.

OccDictRecent[OccDictNextEvictIdx] = occupancy\_byte

After updating the list of most recently coded symbols, the eviction index shall be incremented modulo 16.

OccDictNextEvictIdx = (OccDictNextEvictIdx + 1) % 16

##### Histogram of symbol counts

After each coded occupancy\_byte syntax element, the histogram of symbol occurrences shall be updated and the age of the most probable symbol list shall be incremented.

OccDictHistogram[occupancy\_byte]++  
OccDictMostProbAge++

When the histogram count of symbols equal to occupancy\_byte reaches 1 024, all counts in the histogram shall be halved and any fractional parts discarded.

if (OccDictHistogram[occupancy\_byte] == 1024)  
 for (i = 0; i < 256; i++)  
 OccDictHistogram[i] >>= 1

#### Generation of the most probable symbol list

When OccDictMostProbAge is equal to OccDictMostProbMaxAge, the most probable symbol list shall be recalculated from the histogram of symbol counts.

The most probable symbol list shall be a stable descending ordering of the OccDictHistogram array. Each element OccDictMostProb[ 𝑖 ] shall be the index of the 𝑖-th largest element in the OccDictHistogram array. The ordering shall be such that the following conditions are true:

* OccDictHistogram[ OccDictMostProb[ 𝑖 ] ] ≥ OccDictHistogram[ OccDictMostProb[ 𝑖 + 1 ] ], and
* OccDictMostProb[ 𝑖 ] < OccDictMostProb[ 𝑖 + 1 ] when OccDictHistogram[ OccDictMostProb[ 𝑖 ] ] is equal to OccDictHistogram[ OccDictMostProb[ 𝑖 + 1 ] ].

The age of the generated most probable symbol list shall be 0.

OccDictMostProbAge = 0

The maximum age of the generated most probable symbol list shall be the next value in the bounded geometric progression specified by the expression OccDictMostProbMaxAgeNext:

OccDictMostProbMaxAgeNext := Min(5 × OccDictMostProbMaxAge >> 2, 1024)  
OccDictMostProbMaxAge = OccDictMostProbMaxAge

#### Resetting the histogram of symbol counts

The histogram of symbol counts shall be reset immediately after the first recalculation of the most probable symbol list in each level of the occupancy tree. Counts for symbols that are present in the most probable symbol list shall be set to 1; all other counts shall be set to 0.

if (OccDictMostProbAge == 0)  
 if (… /\* First occurrence in tree level \*/) {  
 for (i = 0; i < 256; i++)  
 OccDictHistogram[i] = 0  
  
 for (i = 0; i < 32; i++) {  
 symbol = OccDictMostProb[i]  
 OccDictHistogram[symbol] = 1  
 }  
 }

#### Determination of CtxIdxDictHg for a bin of occ\_histogram\_index

Contextualization depends upon the reduced neighbourhood pattern, the bin index and the MSBs of the occ\_histogram\_index syntax element value.

Table 18 specifies the value ctxIdxInc for the bin. If the ctxIdxInc is 'bypass', the value of CtxIdxDictHg shall be 'bypass'. Otherwise, the value of CtxIdxDictHg shall be 5 × OccNeighPatR + ctxIdxInc.

Table 18 — Values of ctxIdxInc for bins of the syntax element occ\_histogram\_index

| MSBs of binarized occ\_histogram\_index | BinIdx | | | | |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| '000' | 0 | 1 | 2 | 3 | 4 |
| '001' | 0 | 1 | 2 | bypass | bypass |
| '01' | 0 | 1 | bypass | bypass | bypass |
| '1' | 0 | bypass | bypass | bypass | bypass |

### Bitwise occupancy coding

#### General

Subclause 9.2.10 applies when occtree\_bitwise\_coding is 1.

The neighbourhood-permuted node occupancy bitmap shall be coded as a sequence of individual occupancy\_bit syntax elements. Coding uses constraints on occupancy to infer the value of certain occupancy\_bit syntax elements.

Entropy coding of each coded bit is contextualized by a combination of the coded bit index, previously coded occupancy\_bit syntax elements, the reduced occupied neighbourhood pattern, the number of spatially adjacent child nodes in neighbouring nodes and a ternary prediction based upon the presence of neighbouring nodes.

#### Correspondence between the node occupancy bitmap and occupancy\_bit

Bits of the neighbourhood-permuted node occupancy bitmap shall be coded in the order specified by Table 19. Each occupancy\_bit[ cbIdx ] syntax element codes the bit OccBitCodingOrder[ cbIdx ].

The expression OccBitIdx[ cbIdx ] is the bit position in the node occupancy bitmap of the bit coded by occupancy\_bit[ cbIdx ]. For example, when OccNeighPat is 17, occupancy\_bit[ 6 ] corresponds to the second bit () of the node occupancy bitmap.

OccBitIdx[cbIdx] := OccBitIdxFromNpBit[OccBitCodingOrder[cbIdx]]

The expression OccBitLocC[ cbIdx ][ 𝑘 ] is the node-relative child location represented by occupancy\_bit[ cbIdx ].

OccBitLocC[cbIdx][k] := OccLocC[OccBitIdx[cbIdx]][k]

The expression OccBitMap is the node occupancy bitmap.

OccBitMap :=  
 OccBitMap = 0  
 for (cbIdx = 0; cbIdx < 8; cbIdx++)  
 OccBitMap = OccBitMap | (occupancy\_bit[cbIdx] << OccBitIdx[cbIdx])

Table 19 — Order for coding bits of the neighbourhood-permuted node occupancy bitmap as occupancy\_bit[ cbIdx ]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| cbIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| OccBitCodingOrder[ cbIdx ] | 1 | 7 | 5 | 3 | 2 | 6 | 4 | 0 |

#### Presence of occupancy\_bit

An occupancy bitmap bit shall not be coded if its value can be inferred to be set or unset. The expression OccBitPresent[ cbIdx ] specifies whether occupancy\_bit[ cbIdx ] is present.

OccBitPresent[cbIdx] := ¬(OccBitInferUnset[cbIdx] || OccBitInferSet[cbIdx])

#### Inference of an unset bit

An occupancy bitmap bit shall be inferred to be 0 when either:

* the bit represents an invalid child according to the node's coded axes (9.2.2.3), or
* the bit represents a child within an unoccupied plane signalled by planar occupancy coding.

When the expression OccBitInferUnset[ cbIdx ] is equal to 1, occupancy\_bit[ cbIdx ] shall be inferred to be 0.

OccBitInferUnset[cbIdx] :=  
 ¬AxisCoded[0] && OccBitLocC[cbIdx][0]  
 || ¬AxisCoded[1] && OccBitLocC[cbIdx][1]  
 || ¬AxisCoded[2] && OccBitLocC[cbIdx][2]  
 || ¬PlanarFreeAxis[0] && OccBitLocC[cbIdx][0] ^ occ\_plane\_pos[0]  
 || ¬PlanarFreeAxis[1] && OccBitLocC[cbIdx][1] ^ occ\_plane\_pos[1]  
 || ¬PlanarFreeAxis[2] && OccBitLocC[cbIdx][2] ^ occ\_plane\_pos[2]

#### Inference of a set bit

An occupancy bitmap bit occupancy\_bit[ cbIdx ] shall be inferred to be 1, as specified by the expression OccBitInferSet[ cbIdx ], when:

* the bit is the last present bit and all previous coded bits are 0, or
* the bit is the penultimate present bit, all previous coded bits are 0 and the node is required to have two child nodes, or
* the bit is in a plane identified as occupied by planar occupancy coding, the bit is the last bit in the plane and all previous bits in the plane are 0.

OccBitInferSet[cbIdx] :=  
 PlanarEligible[0] && PopCnt(OccKnownZero & (0x0F << 4 × OccBitLocC[cbIdx][0])) == 3  
 || PlanarEligible[1] && PopCnt(OccKnownZero & (0x33 << 2 × OccBitLocC[cbIdx][1])) == 3  
 || PlanarEligible[2] && PopCnt(OccKnownZero & (0x55 << 1 × OccBitLocC[cbIdx][2])) == 3  
 || cbIdx == 6 && PopCnt(OccKnown) == 0 && OccMinChildren == 2  
 || cbIdx == 7 && PopCnt(OccKnown) == 0

The expression OccKnownMask is a bit mask that identifies bits of the node occupancy bitmap that have a known value prior to coding occupancy\_bit[ cbIdx ].

OccKnownMask :=  
 OccKnownMask = 0  
 for (i = 0; i < cbIdx; i++)  
 OccKnownMask |= 1 << OccBitIdx[i]  
 for (i = 0; i < 8; i++)  
 OccKnownMask |= OccBitInferUnset[i] << OccBitIdx[i]

The expression OccKnown is the partially coded node occupancy bitmap comprising the bits coded prior to occupancy\_bit[ cbIdx ].

OccKnown :=  
 OccKnown = 0  
 for (i = 0; i < cbIdx; i++)  
 OccKnown |= occupancy\_bit[i] << OccBitIdx[i]

The expression OccKnownZero is a bitmap of occupancy bits that are known to be 0.

OccKnownZero := (0xFF ^ OccKnown) & OccKnownMask

#### Contextualization

##### General

Contextualization of occupancy\_bit syntax elements is a two-stage process. First, context discriminators are used to select a demi-CPM. Then, the demi-CPM is used to select the CPM that codes the syntax element.

A demi-CPM is an 8-bit unsigned integer that models the probability of a coded zero-valued occupancy\_bit syntax element.

1. The values 0, 128 and 256 represent the probability of a zero bin as impossible, equiprobable and certain, respectively. The values 0 and 256 can never be attained due to the operation of the probability models' update process.

##### State variables

Context selection is specified in terms of the following state variable:

* The array OccCtxSel; OccCtxSel[ selNeigh ][ cbIdx ][ selSib ][ selAdj ][ selPred ] is a demi-CPM, contextualized by selNeigh, cbIdx, selSib, selAdj and selPred.

##### Initial state

The demi-CPMs shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed according to the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), all elements of OccCtxSel shall be set to 127.

##### Determination of CtxIdxOccBit for the syntax element occupancy\_bit

The expression OccCtxSelVar specifies the demi-CPM for the syntax element occupancy\_bit[ CbIdx ] using:

* SelNeigh, the reduced occupied neighbourhood context discriminator (9.2.10.6.6);
* SelSib, the sibling occupancy context discriminator (9.2.10.6.7);
* SelAdj, the adjacent child neighbour context discriminator (9.2.10.6.8);
* SelPred, the neighbour-predicted occupancy context discriminator (9.2.10.6.9).

OccCtxSelVar := OccCtxSel[SelNeigh][CbIdx][SelSib][SelAdj][SelPred]

The CPM index, CtxIdxOccBit, shall be the value of the demi-CPM exclusive of the bottom three bits:

CtxIdxOccBit := OccCtxSelVar >> 3

##### Update after each coded occupancy\_bit syntax element

After each coded occupancy\_bit syntax element, its demi-CPM shall be updated. The update specified by Table 20 supplies a value for incrementing or decrementing the probability of a zero bin based upon the upper four bits of the demi-CPM's value:

if (OccBitPresent[CbIdx])  
 if (occupancy\_bit[CbIdx])  
 OccCtxSelVar += OccCtxSelUpdate[255 – OccCtxSelVar >> 4]  
 else  
 OccCtxSelVar −= OccCtxSelUpdate[OccCtxSelVar >> 4]

Table 20 — Values of OccCtxSelUpdate[ 𝑖 ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| OccCtxSelUpdate[ 𝑖 ] | 0 | 1 | 1 | 2 | 4 | 7 | 9 | 11 | 14 | 16 | 19 | 23 | 22 | 18 | 13 | 6 |

##### Reduced occupied neighbourhood context discriminator

The reduced occupied neighbourhood context discriminator shall distinguish between different reduced occupied neighbourhood patterns (OccNeighPatR) depending upon the coded bit index (CbIdx) as specified by Table 21 as the expression SelNeigh.

Table 21 — Discriminated values SelNeigh for occupancy\_bit[ CbIdx ] and OccNeighPatR

| CbIdx | OccNeighPatR | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 .. 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 4 .. 5 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 4 |
| 6 | 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

##### Sibling occupancy context discriminator

The sibling occupancy context discriminator shall distinguish between arrangements of previously coded/inferred siblings for the node coded by occupancy\_bit[ CbIdx ] as specified by the expression SelSib:

* If there are no nodes present in the occupied neighbourhood pattern, discrimination shall be by the number of present child nodes identified by the syntax elements occupancy\_bit[ 𝑖 ] with 𝑖 < CbIdx.
* If there is at least one node present in the occupied neighbourhood pattern, discrimination shall be by the combination of present child nodes identified by the syntax elements occupancy\_bit[ 𝑖 ] with 𝑖 < CbIdx.

SelSib := OccNeighPat ? occPrevBits : PopCnt(occPrevBits)

The expression occPrevBits is the concatenation of occupancy\_bit[ 𝑖 ] for 𝑖 < CbIdx.

occPrevBits :=  
 occPrevBits = 0  
 for (i = 0; i < CbIdx; i++)  
 occPrevBits |= occupancy\_bit[i] << i

##### Adjacent child neighbour context discriminator

The adjacent child neighbour context discriminator for child the node coded by occupancy\_bit[ CbIdx ] is specified by the expression SelAdj. When adjacent child neighbour contextualization is enabled (occtree\_adjacent\_child\_enabled is 1), it distinguishes between contexts by:

* the number of child nodes from available, previously coded nodes in the same tree level (9.2.7.2) that adjoin the coded child by a face; and
* whether any of the available, previously coded nodes in the same tree level that adjoin the coded child node do not have a child node that also adjoins the coded child.

An example is illustrated in Figure 19. The child node of the coded node N is adjoined by a single child node. There are two available previously coded nodes that adjoin , one of which does not contain a child node that also adjoins .

SelAdj := occtree\_adjacent\_child\_enabled  
 ? 2 × Min(2, adjCntC) + ((cbIdx ≤ 4 || adjCntC == 1) && adjUnocc)  
 : 0

The expression adjOccN[ 𝑘 ] identifies whether there is a spatially adjacent node along the 𝑘-th axis within the occupied neighbourhood availability window. Values for the expressions ds, dt and dv are specified in Table 22 for each axis 𝑘.

adjOccN[k] := ¬OccBitLocC[CbIdx][k] && OccNeigh[Ns + ds][Nt + dt][Nv + dv]

The expression adjOccC[ 𝑘 ] identifies whether there is a spatially adjacent child node along the 𝑘-th axis within the occupied neighbourhood availability window. Values for the expressions ds, dt and dv are specified in Table 22 for each axis, 𝑘.

adjOccC[k] := adjOccN[k] && OccNodePresent[Dpth + 1][cs + ds][ct + dt][cv + dv] ≠ 0  
 where  
 cs := NsC + OccBitLocC[CbIdx][0]  
 ct := NtC + OccBitLocC[CbIdx][1]  
 cv := NvC + OccBitLocC[CbIdx][2]

Table 22 — Relative neighbour locations ( ds, dt, dv ) used in the computation of adjOccN[ 𝑘 ] and adjOccC[ 𝑘 ]

| 𝑘 | ds | dt | dv |
| --- | --- | --- | --- |
| 0 | −1 | 0 | 0 |
| 1 | 0 | −1 | 0 |
| 2 | 0 | 0 | −1 |

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| N | Coded node |
|  | Contextualized child with OccBitIdx[ cbIdx ] = 0 |

Figure 19 — Example of adjacent child neighbour context discrimination.

The expressions adjCntN and adjCntC are the number of spatially adjacent nodes and child nodes, respectively, that are within the occupied neighbourhood availability window.

adjCntN := adjOccN[0] + adjOccN[1] + adjOccN[2]  
adjCntC := adjOccC[0] + adjOccC[1] + adjOccC[2]

The expression adjUnocc identifies whether there exists a spatially adjacent node within the occupied neighbourhood availability window that does not have a child node spatially adjacent to the coded child.

adjUnocc := adjCntN ≠ adjCntC

##### Neighbour-predicted-occupancy context discriminator

###### General

The neighbour-predicted-occupancy context discriminator shall, for eligible nodes (9.2.10.6.9.2), distinguish between three predictions for the presence of the child node coded by occupancy\_bit[ CbIdx ]. The discriminator is specified by the expression SelPred. The three predictions are that the node is present, not present, or that it is unpredictable.

SelPred := SelPredEligible ? OccIntraPred : 0

###### Eligibility

The discriminator shall only form a prediction for eligible nodes as specified by the expression SelPredEligible. Eligible nodes shall have both:

* three free axes and
* a maximum log2 node dimension less than occtree\_intra\_pred\_max\_nodesize\_log2.

SelPredEligible :=  
 OccFreeAxisCnt == 3 && MaxVec(NodeSizeLog2) < occtree\_intra\_pred\_max\_nodesize\_log2

###### Occupancy prediction

Occupancy prediction generates a ternary prediction for the presence of a child node identified by occupancy\_bit[ CbIdx ] of a coded node. The prediction is specified by the expression OccIntraPred. It is based upon how many of the nodes that neighbour the coded node also adjoin the volume of the identified child node by a face, edge or corner (as illustrated by Figure 20):

* A child node shall be predicted to be not present if there are two or fewer adjoining nodes.
* A child node shall be predicted to be present if there is at least a threshold number of adjoining nodes. The threshold is specified by the expression OccIntraThreshold. The threshold is four nodes unless there are more than 13 neighbouring nodes; in which case the threshold is 5 nodes.
* Otherwise, the presence is unpredictable.

1. The size of the child node volume is half the size of the neighbour nodes' in each dimension.

OccIntraPred := (OccAdjCnt ≤ 2) + 2 × (OccAdjCnt ≥ OccIntraThreshold)  
OccIntraThreshold := 4 + (OccNeighCnt ≥ 14)

图片包含 体育, 游戏机, 桌子

描述已自动生成

Figure 20 — Nodes that adjoin a child node by a face, edge or corner.

The expression OccAdj[ ds ][ dt ][ dv ] identifies whether a neighbouring node with a relative tree location ( ds, dt, dv ) to the coded node would adjoin the identified child volume.

OccAdj[ds][dt][dv] := (OccBitIdx[CbIdx] & adjMask) == adjLoc  
 where  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

The expression OccAdjCnt is the number of neighbours that adjoin the identified child volume.

OccAdjCnt := SumN26[neighAdj]  
 where  
 neighAdj[ds][dt][dv] := OccNeigh[Ns + ds][Nt + dt][Nv + dv] && OccAdj[ds][dt][dv]

The expression OccNeighCnt is the number of nodes that neighbour the coded node.

OccNeighCnt := SumN26[neighRel]  
 where  
 neighRel[ds][dt][dv] := OccNeigh[Ns + ds][Nt + dt][Nv + dv]

The expression SumN26[ expr ] sums the result of applying expr to the relative tree location of each of the 26 possible neighbouring nodes.

SumN26[expr] :=  
 SumN26 = 0  
 for (ds = −1; ds ≤ 1; ds++)  
 for (dt = −1; dt ≤ 1; dt++)  
 for (dv = −1; dv ≤ 1; dv++)  
 if (ds ≠ 0 && dt ≠ 0 && dv ≠ 0)  
 SumN26 += expr[ds][dt][dv]

##### Coding occupancy bits using OBUF

The occupancy bits of octree nodes are coded as follows.

* A first buffer is created for all OBUF instances according to subclause 12.3. The OBUF buffer size obufBufferSize is set as 20000, the buffer depth obufLeafDepth of fully deployed trees is set as 4.
* The array of OBUF ACPMs is created according to subclause 12.2.2.
* All OBUF instances are created according to subclause 12.2 by using the octree buffer created, and the OBUF instances are divided into two types, sparse OBUF instances and non-sparse OBUF instances. Each type of OBUF instances has specific sizes nBit1 and nBit2 to specify the sizes of contextual information info1 and info2 used as input when calling the OBUF instance.
* The sizes nBit1[j][i] and nBit2[j][i] for sparse OBUF instances are set according to Table 23and the sizes nBit1[j][i] and nBit2[j][i] for non-sparse OBUF instances are set according to Table 24, 𝑘, where index i represents the child node in current coded node, and index j represents intra prediction method (j = 0) or inter prediction method (j = 1) for octree coding.

Table 23 — the size values (nBit1[ j][ i ], nBit2[ j][ i ]) of sparse OBUF instance

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | i | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0** | (4,12) | (7,12) | (7,12) | (6,12) | (4,12) | (7,12) | (7,12) | (6,12) |
| **1** | (4,12) | (7,12) | (7,12) | (6,12) | (4,12) | (7,12) | (7,12) | (6,12) |

Table 24 — the size values (nBit1[ j][ i ], nBit2[ j][ i ]) of non-sparse OBUF instance

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | i | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0** | (6,13) | (6,13) | (6,13) | (6,11) | (6,13) | (6,13) | (6,13) | (6,11) |
| **1** | (6,13) | (6,13) | (6,13) | (6,11) | (6,13) | (6,13) | (6,13) | (6,11) |

* Optionally, for each child node, if the corresponding table among Table 25 to Table 32 is provided, the context array *ctxIdxMap*[][] of the created OBUF instances is initialized according to the provided table. Otherwise, for this child node, *ctxIdxMap*[][] is initialized with all the values set as 127.
* The array *ctxIdxMap*[ ][ ] corresponds to 8-bit context indices pointing (after right shift by 3) to OBUF ACPMs of the array *obufCtxArray*[ ].

Table 25 — Initial values of *ctxIdxMap*[ j][0] for child node under intra octree coding

| j | *ctxIdxMap*[ j][0] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 127 | 17 | 82 | 38 | 127 | 105 | 141 | 81 |
| 8 .. 15 | 127 | 15 | 45 | 43 | 116 | 105 | 152 | 115 |
| 16 .. 23 | 127 | 53 | 21 | 20 | 127 | 127 | 127 | 37 |
| 24 .. 31 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 32 .. 39 | 171 | 186 | 170 | 240 | 182 | 209 | 223 | 240 |
| 40 .. 47 | 44 | 101 | 101 | 74 | 65 | 66 | 134 | 199 |
| 48 .. 55 | 47 | 27 | 141 | 113 | 126 | 61 | 240 | 151 |
| 56 .. 63 | 45 | 68 | 113 | 101 | 47 | 84 | 153 | 234 |

Table 26 — Initial values of *ctxIdxMap*[ j][ 1]for child node under intra octree coding

| j | *ctxIdxMap*[ j][1] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 240 | 240 | 222 | 240 | 175 | 181 | 127 | 127 |
| 8 .. 15 | 120 | 152 | 132 | 116 | 57 | 127 | 127 | 127 |
| 16 .. 23 | 105 | 185 | 127 | 87 | 105 | 116 | 65 | 69 |
| 24 .. 31 | 66 | 105 | 58 | 43 | 44 | 49 | 18 | 15 |
| 32 .. 39 | 228 | 240 | 138 | 240 | 178 | 198 | 114 | 152 |
| 40 .. 47 | 173 | 240 | 204 | 127 | 70 | 141 | 127 | 127 |
| 48 .. 55 | 184 | 192 | 105 | 116 | 121 | 181 | 35 | 46 |
| 56 .. 63 | 58 | 87 | 114 | 73 | 51 | 15 | 101 | 40 |

Table 27 — Initial values of *ctxIdxMap*[ j][ 2]for child node under intra octree coding

| j | *ctxIdxMap*[ j][2] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 194 | 240 | 173 | 190 | 115 | 129 | 87 | 87 |
| 8 .. 15 | 168 | 161 | 116 | 92 | 127 | 127 | 26 | 96 |
| 16 .. 23 | 160 | 106 | 96 | 127 | 86 | 109 | 105 | 127 |
| 24 .. 31 | 116 | 68 | 80 | 27 | 116 | 116 | 46 | 19 |
| 32 .. 39 | 240 | 240 | 205 | 114 | 215 | 194 | 134 | 78 |
| 40 .. 47 | 225 | 182 | 191 | 141 | 122 | 127 | 58 | 127 |
| 48 .. 55 | 200 | 214 | 124 | 89 | 188 | 161 | 91 | 59 |
| 56 .. 63 | 126 | 126 | 74 | 152 | 80 | 96 | 59 | 127 |

Table 28 — Initial values of *ctxIdxMap*[ j][ 3] for child node under intra octree coding

| j | *ctxIdxMap*[ j][3] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 59 | 121 | 160 | 210 | 171 | 211 | 240 | 231 |
| 8 .. 15 | 127 | 56 | 149 | 125 | 127 | 115 | 230 | 204 |
| 16 .. 23 | 55 | 127 | 78 | 192 | 127 | 182 | 197 | 218 |
| 24 .. 31 | 35 | 39 | 15 | 72 | 96 | 87 | 151 | 139 |
| 32 .. 39 | 46 | 141 | 152 | 240 | 114 | 162 | 240 | 240 |
| 40 .. 47 | 87 | 69 | 127 | 96 | 44, | 67 | 129 | 155 |
| 48 .. 55 | 53 | 105 | 141 | 73 | 96 | 105 | 198 | 128 |
| 56 .. 63 | 15 | 35 | 96 | 57 | 127 | 96 | 127 | 96 |

Table 29 — Initial values of *ctxIdxMap*[ j][ 4] for child node under intra octree coding

| j | *ctxIdxMap*[ j][4] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 23 | 30 | 130 | 66 | 139 | 127 | 30 | 105 |
| 8 .. 15 | 113 | 127 | 87 | 127 | 127 | 127 | 127 | 127 |
| 16 .. 23 | 166 | 146 | 70 | 15 | 209 | 116 | 141 | 90 |
| 24 .. 31 | 114 | 138 | 71 | 15 | 127 | 127 | 127 | 127 |
| 32 .. 39 | 204 | 240 | 198 | 219 | 232 | 240 | 142 | 240 |
| 40 .. 47 | 151 | 139 | 87 | 127 | 209 | 190 | 43 | 141 |
| 48 .. 55 | 141 | 181 | 116 | 127 | 240 | 210 | 88 | 127 |
| 56 .. 63 | 73 | 170 | 65 | 61 | 140 | 194 | 48 | 65 |

Table 30 — Initial values of *ctxIdxMap*[ j][ 5] for child node under intra octree coding

| j | *ctxIdxMap*[ j][5] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 240 | 99 | 240 | 69 | 189 | 96 | 105 | 80 |
| 8 .. 15 | 154 | 233 | 152 | 141 | 127 | 152 | 127 | 127 |
| 16 .. 23 | 166 | 48 | 57 | 15 | 97 | 41 | 43 | 15 |
| 24 .. 31 | 127 | 116 | 127 | 127 | 127 | 85 | 127 | 127 |
| 32 .. 39 | 235 | 214 | 177 | 154 | 240 | 240 | 161 | 61 |
| 40 .. 47 | 219 | 185 | 152 | 208 | 157 | 90 | 127 | 127 |
| 48 .. 55 | 117 | 138 | 69 | 30 | 154 | 80 | 62 | 15 |
| 56 .. 63 | 141 | 121 | 127 | 127 | 127 | 41 | 127 | 105 |

Table 31 — Initial values of *ctxIdxMap*[ j][ 6] for child node under intra octree coding

| j | *ctxIdxMap*[ j][6] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 227 | 199 | 188 | 103 | 212 | 141 | 205 | 55 |
| 8 .. 15 | 240 | 240 | 210 | 141 | 178 | 70 | 127 | 127 |
| 16 .. 23 | 240 | 84 | 139 | 73 | 139 | 60 | 127 | 59 |
| 24 .. 31 | 161 | 127 | 127 | 127 | 80 | 65 | 127 | 127 |
| 32 .. 39 | 201 | 195 | 127 | 69 | 175 | 80 | 87 | 39 |
| 40 .. 47 | 115 | 240 | 127 | 175 | 116 | 168 | 127 | 127 |
| 48 .. 55 | 115 | 96 | 42 | 23 | 65 | 65 | 49 | 15 |
| 56 .. 63 | 96 | 141 | 127 | 127 | 105 | 127 | 127 | 127 |

Table 32 — Initial values of *ctxIdxMap*[ j][ 7] for child node under intra octree coding

| j | *ctxIdxMap*[ j][7] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 141 | 141 | 139 | 146 | 127 | 144 | 177 | 218 |
| 8 .. 15 | 127 | 63 | 127 | 115 | 127 | 164 | 240 | 194 |
| 16 .. 23 | 127 | 127 | 73 | 97 | 127 | 190 | 186 | 128 |
| 24 .. 31 | 73 | 16 | 15 | 88 | 116 | 127 | 80 | 161 |
| 32 .. 39 | 127 | 116 | 116 | 240 | 42 | 166 | 161 | 230 |
| 40 .. 47 | 96 | 47 | 127 | 127 | 58 | 88 | 116 | 109 |
| 48 .. 55 | 105 | 116 | 15 | 61 | 15 | 80 | 73 | 155 |
| 56 .. 63 | 15 | 15 | 15 | 45 | 36 | 73 | 57 | 121 |

* Then, the advanced occupied neighbourhood pattern *OccAdvNeiPati* (9.2.7.6) is used as the contextual information *CI* for entropy coding occupancy of child node of current coded node as follows.
  + *CI* is split into a primary part *CI1* and a second part *CI2* to derive the contextural information info1 and info2 for dynamic OBUF (12.5). *CI1* is a first representation of occupancy information of the set of neighboring nodes of current child node andcontains the first nBit1 bits of *OccAdvNeiPati*. *CI2* is a finer representation of the occupancy information and contains the remaining bits of *OccAdvNeiPati*. *CI1* is set as the first contextual information info1, and *CI2* is set as the second contextual information info2.
  + For example, to code child node of the coded node,
    - if the occupied neighbour nodes of a child node are not sparse, the contextual information is set as below,

CI1= OccAdvNeiPat0 >> 13

CI2= OccAdvNeiPat0 & 0x1FFF

* + - Otherwise, the contextual information is set as below.

CI1= OccAdvNeiPat0 >> 12

CI2= OccAdvNeiPat0 & 0x0FFF

* Then, each bit of child node occupancy of current node is decoded according to subclause 12.4 by calling the OBUF instances of octree, and usinginfo1,info2, *ctxIdxMap*[ ][ ] and *obufCtxArray*[ ]. The type (sparse or non-sparse) of called OBUF instances depends on the first bit of *OccAdvNeiPati* (9.2.7.6), which indicates if the occupied neighbour nodes of coded child node are sparse or not.

### Planar occupancy coding

#### General

Subclause 9.2.11 applies when occtree\_planar\_enabled is 1.

Planar occupancy coding decomposes the node occupancy bitmap into axis-aligned planes. Each coded axis has two perpendicular planes that child nodes can occupy as illustrated by Table 33. For each planar-eligible coded axis (9.2.11.5), planar occupancy coding specifies whether one of the two planes is unoccupied. Plane occupancy is then used by bitwise occupancy coding to constrain and infer the coding of bits in the node occupancy bitmap.

There shall be at least one child node in each occupied plane.

1. The definition of an occupancy tree node requires that at least one plane is occupied along each coded axis.

For example, if a node has three planar-eligible coded axes, there is a total of six axis-aligned planes. Along the S axis (𝑘 = 0), information about the occupied state of the two T-V planes is coded.

Table 33 — Plane, perpendicular to each planar axis, 𝑘

| 𝑘 | Planar axis | Plane axes |
| --- | --- | --- |
| 0 | S | T-V |
| 1 | T | S-V |
| 2 | V | S-T |

#### Syntax element semantics

planar\_copy\_modespecifies, when present, whether (when 1) or not (when 0) the values of occ\_plane\_pos[ 𝑘 ] indicating the occupancy of the child nodes are copied from the corresponding node in the reference frame.

multi\_planar\_flagspecifies, when present, whether the positions of child nodes in the node occupancy bitmap shall locate at the intersection of planes (equal to 1) or not (equal to 0).

occ\_single\_plane[ 𝑘 ] specifies, when present, whether (when 1) the locations of child nodes in the node occupancy bitmap shall occupy a single plane or (when 0) both planes perpendicular to the 𝑘-th axis. When equal to 1, the location of the single plane is specified by occ\_plane\_pos[ 𝑘 ]. When not present, the child nodes can be located in either or both planes perpendicular to the 𝑘-th axis. The number of occupied planes is illustrated in Table 34.

Table 34 — Interpretation of PlanarEligible[ 𝑘 ] and occ\_single\_plane[ 𝑘 ]

| PlanarEligible[ 𝑘 ] | occ\_single\_plane[ 𝑘 ] | № occupied planes | PlanarMinPlanes[ 𝑘 ] |
| --- | --- | --- | --- |
| 0 | not present | 1 or 2 | 1 |
| 1 | 0 | 2 | 2 |
| 1 | 1 | 1 | 1 |

occ\_plane\_pos[ 𝑘 ] specifies the node-relative location along the 𝑘-th axis for the occupied plane specified by occ\_single\_plane[ 𝑘 ] equal to 1.

Examples of planar occupancy are illustrated in Figure 21. Each entry shows the child node indices for a node with three coded axes. If an axis 𝑘 is eligible for planar coding and occ\_single\_plane[ 𝑘 ] is 0, the two occupied planes are marked with a dotted line. If occ\_single\_plane[ 𝑘 ] is 1, the unoccupied plane is marked by hatching and a dashed (red) line, with its child indices in grey; the occupied plane is not marked for clarity. In the case where three axes are eligible and each has occ\_single\_plane equal to 1, there is only a single child node present; its location is fully constrained by the planes.

图示, 示意图

描述已自动生成

Figure 21 — Example planar constraints on node occupancy bitmap.

#### Minimum number of child nodes

Planar occupancy coding can require that the coded node has a minimum of either one or two child nodes, as specified by the expression PlanarMinChildren.

A node shall have at least two child nodes if, for any planar-eligible axis 𝑘, occ\_single\_plane[ 𝑘 ] specifies that there shall be a minimum of two occupied planes; otherwise, the node shall have at least one child node.

PlanarMinChildren := MaxVec(PlanarMinPlanes)

#### Free axes

A free axis is a coded axis whose occupancy is not constrained to a single plane by occ\_single\_plane, as specified by the expression PlanarFreeAxis[ 𝑘 ].

PlanarFreeAxis[k] := ¬PlanarEligible[k] || PlanarMinPlanes[k] == 2

In Figure 21 a free axis is an axis with either two marked planes or no marked planes. In the case where three axes are eligible, of which two have occ\_single\_plane equal to 1, PlanarMinChildren is 2.

#### Per-axis eligibility

##### Condition

Only certain axes are eligible for planar occupancy coding. Eligibility for the 𝑘-th axis is specified by the expression PlanarEligible[ 𝑘 ]. Eligibility shall be determined after any applicable update to the eligibility state (9.2.11.5.2).

An axis is not eligible for planar coding when either planar occupancy coding is disabled, or the axis is not coded, or geo\_planar\_idcm\_angular\_disabled is equal to 1 and occ\_direct\_node is 1. Otherwise, the determination of eligibility depends upon the use of the angular coding and whether the node is eligible for angular contextualization (9.2.13.7.2) as specified in Table 35.

PlanarEligible[k] :=  
 occtree\_planar\_enabled && AxisCoded[k]  
 && (geom\_angular\_enabled ? PlanarEligibleByAng[k] : PlanarEligibleByDensity[k])

PlanarEligibleByAng[k] :=  
 AngularEligible ? k == 2 || k == AzimuthAxis  
 : 0

Table 35 — Method to determine eligibility for an axis

| Axis | 𝑘 | Angular coding disabled | Angular coding enabled | |
| --- | --- | --- | --- | --- |
| AngularEligible == 0 | AngularEligible == 1 |
| S | 0 | PlanarEligibleByDensity[ 0 ] | Not eligible | AzimuthAxisIsS |
| T | 1 | PlanarEligibleByDensity[ 1 ] | Not eligible | AzimuthAxisIsQ |
| V | 2 | PlanarEligibleByDensity[ 2 ] | Not eligible | Eligible |

Axes whose eligibility is determined by the expression PlanarEligibeByDensity[ 𝑘 ] are eligible if:

* the density of the points in the tree level at depth *dpth* is sparse enough.

PlanarEligibleByDensity[k]:= PointDensity[dpth ] < 13

The expression *PointDensity*[*dpth* ] is a factor that identifies the density of the points in the tree level at depth *dpth*:

PointDensity[dpth ]:= (slice\_num\_points\_minus1 + 1 – DirectNodePointCnt) × 10 / OccNodeCnt[ dpth ]

##### State variable and update

When octree\_planar\_neigh\_prediction\_enabled is 1, eligibility is specified in terms of the following state variables:

* The sparse array *NodeOccMap* of node occupancy bitmaps; *NodeOccMap*[ *dpth* ][ *ns* ][ *nt* ][ *nv* ] is the coded node occupancy bitmap of the node located at ( *ns*, *nt*, *nv* ) in the tree level at depth *dpth*.

At the end of each occupancy\_tree\_node syntax structure, the state shall be updated for the next tree level:

NodeOccMap[Dpth][Ns][Nt][Nv] = OccupancyMap.

#### Previous coded node for contextualization of occ\_plane\_pos

##### General

Subclause 9.2.11.6 does not apply when occtree\_planar\_buffer\_disabled is 1.

Planar contextualization of occ\_plane\_pos[ 𝑘 ] can use the following information about the previous planar-eligible coded node that is located in the same plane (9.2.11.6.2) as the coded node:

* The zone within the plane that the node resides (9.2.11.6.3).
* The values for occ\_single\_plane[ 𝑘 ] and occ\_plane\_pos[ 𝑘 ].

##### Identification of the plane

The plane normal to the 𝑘-th axis of a coded node is identified by its location along the axis modulo .

PlanarNodeAxisLoc[k] := Nloc[k] & 0x3FFF

##### Zone within a plane

The plane normal to the 𝑘-th axis of a coded node is partitioned into zones according to the norm of the node location within the plane. The expression PlanarNodeZone[ 𝑘 ] identifies the zone for the coded node.

PlanarNodeZone[k] :=  
 k == 0 ? Max(Nt & 0xF8, Nv & 0xF8) >> 3 :  
 k == 1 ? Max(Ns & 0xF8, Nv & 0xF8) >> 3 :  
 k == 2 ? Max(Ns & 0xF8, Nt & 0xF8) >> 3 : na

Figure 22 illustrates the partitioning of an S-T plane (𝑘 = 0) according to node location.

图示

描述已自动生成

Figure 22 — S-T plane divided into zones.

##### State variables

Information about previous planar-eligible coded nodes is specified in terms of the following state variables; the indexes 𝑘 and axisLoc identify the location of a plane along the 𝑘-th axis:

* The array PrevPlanarNodeZone; PrevPlanarNodeZone[ 𝑘 ][ axisLoc ] is the plane zone of the previous planar-eligible node in the identified plane.
* The array PrevOccSinglePlane; PrevOccSinglePlane[ 𝑘 ][ axisLoc ] is the value of occ\_single\_plane[ 𝑘 ] for the previous planar-eligible node in the identified plane.
* The array PrevOccPlanePos; PrevOccPlanePos[ 𝑘 ][ axisLoc ] is the value of occ\_plane\_pos[ 𝑘 ] for the previous planar-eligible node in the identified plane.

##### Initial state

At the start of every occupancy tree level, every element of PrevOccSinglePlane shall be initialized to 0.

##### State update at the end of each node

After each occupancy\_tree\_node syntax structure, the state shall be updated for each planar-eligible axis:

for (k = 0; k < 3; k++)  
 if (PlanarEligible[k]) {  
 PrevPlanarNodeZone[k][PlanarNodeAxisLoc[k]] = PlanarNodeZone[k]  
 PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]] = occ\_single\_plane[k]  
  
 if (occ\_single\_plane[k])  
 PrevOccPlanePos[k][PlanarNodeAxisLoc[k]] = occ\_plane\_pos[k]  
 }

#### Determination of CtxIdxPlanePos for occ\_plane\_pos[ 𝑘 ]

##### Case for angular-ineligible axes

Contextualization of occ\_plane\_pos[ 𝑘 ] for nodes not eligible for angular contextualization (AngularEligible is 0) is specified by the expression CtxIdxPlanePos. When octree\_planar\_neigh\_prediction\_enabled is 1, CtxIdxPlanePos is set equal to *CtxIdxNeighPlanePos* using the planar information of neighbors (9.2.11.7.2).

CtxIdxPlanePos :=octree\_planar\_neigh\_prediction\_enabled ? CtxIdxNeighPlanePos:  
 (occtree\_planar\_buffer\_disabled || ¬PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]]  
 ? adjPlaneCtxInc  
 : 12 × k + 4 × adjPlaneCtxInc + 2 × zoneCtxInc + prevPlanePosCtxInc + 3)  
.

The variable OccupancyMapP for a node specifies the coded node occupancy bitmap of its parent node.

The expression adjPlaneCtxInc discriminates by whether nodes have adjoining neighbours on a single side along the 𝑘-th axis, and if so, on which of the two sides they are present. Adjoining neighbours are:

* those along the 𝑘-th axis identified by the corresponding bits of the occupied neighbourhood pattern (adjNeighHL); and
* when the node is in the lower 𝑘-th axis plane of its parent, the sibling nodes in the corresponding upper plane (identified by OccPlaneMask[ 1 ][ 𝑘 ]). The bit masks OccPlaneMask[ planeLoc ][ 𝑘 ] that identify planes in an occupancy bitmap are specified by Table 36.

adjPlaneCtxInc := (adjNeighHL | sibPlaneH << 1) % 3  
 where  
 adjNeighHL := (OccNeighPat >> 2 × k) & 3  
 sibPlaneH := (Nloc[k] & 1) ≠ 1 && (OccupancyMapP & OccPlaneMask[1][k]) ≠ 0

1. Whenever occtree\_coded\_axis[ Dpth − 1 ][ 𝑘 ] is 0, sibPlaneH is always 0.

Table 36 — Bit masks OccPlaneMask[ planeLoc ][ 𝑘 ] that identify planes of a node occupancy bitmap

|  |  |  |  |
| --- | --- | --- | --- |
| 𝑘 | 0 | 1 | 2 |
| OccPlaneMask[ 0 ][ 𝑘 ] | 0x0F | 0x33 | 0x55 |
| OccPlaneMask[ 1 ][ 𝑘 ] | 0xF0 | 0xCC | 0xAA |

If occtree\_planar\_buffer\_disabled is 0, contextualization uses information about the previous planar-eligible node in the plane identified by the coded node location (9.2.11.6.2).

The expression zoneCtxInc discriminates by whether the coded node is within ±1 zones of the identified previous node.

zoneCtxInc := Abs(a − b) > 1  
 where  
 a := PrevPlanarNodeZone[k][PlanarNodeAxisLoc[k]]  
 b := PlanarNodeZone[k]

The expression prevPlanePosCtxInc discriminates by the occupied plane position of the identified previous node.

prevPlanePosCtxInc := PrevOccPlanePos[k][PlanarNodeAxisLoc[k]]

##### Neighbourhood planar information based prediction

The expression adjOccC[ *n*] identifies the occupancy bitmaps of the spatial adjacent nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 37.

adjOccC[n] := OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0 ? NodeOccMap[Dpth][Ns + ds][Nt + dt][Nv + dv] : 0

The expression adjOccSinglePlane[ *k*] identifies the occupied neighbourhood planar pattern along the 𝑘-th axis. It is a linear combination of spatial adjacent nodes coded in the same tree level that are available and adjoin the coded node. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 37. An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood planar pattern equal to 0.

adjOccSinglePlane[k]:= 0  
  
for (n = 0; n < 7; n++)  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] == 0)  
 adjOccSinglePlane[k] <<= 1  
 else {  
 plane0 = (adjOccC[n] & OccPlaneMask[0][k]) ≠ 0  
 plane1 = (adjOccC[n] & OccPlaneMask[1][k]) ≠ 0  
 hasSinglePlane = plane0 ^ plane1  
 adjOccSinglePlane[k] |= hasSinglePlane  
 }

The expression adjOccPlanePos[ *k*] identifies the occupied neighbourhood planar position pattern along the 𝑘-th axis. It is a linear combination of spatial adjacent nodes coded in the same tree level that are available and adjoin the coded node. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 37. An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood planar position pattern equal to 0.

adjOccPlanePos[k] :=0  
  
for (n = 0; n < 7; n++)  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] == 0)  
 adjOccPlanePos[k] << =1  
 else{  
 plane0 = (adjOccC[n] & OccPlaneMask[0][k]) ≠ 0  
 plane1 = (adjOccC[n] & OccPlaneMask[1][k]) ≠ 0  
 hasSinglePlane = plane0 ^ plane1  
 adjOccPlanePos[k] |= hasSinglePlane & OccPlaneMask[1][k]  
 adjOccPlanePos[k] <<= 1  
 }

Table 37 — Relative locations (*ds*, *dt*, *dv*) used in the computation of adjOccC[ *n*], adjOccSinglePlane[ *k*], adjOccPlanePos[ *k*] and neighAvailable

| *n* | *ds* | *dt* | *dv* |
| --- | --- | --- | --- |
| 0 | −1 | 0 | 0 |
| 1 | 0 | −1 | 0 |
| 2 | 0 | 0 | −1 |
| 3 | -1 | -1 | 0 |
| 4 | -1 | 1 | -1 |
| 5 | 0 | -1 | -1 |
| 6 | -1 | -1 | -1 |

The expression adjOccNeighPat identifies the bitmap for the spatial nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 38.

adjOccNeighPat: =0  
for (n = 0; n < 12; n++){  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0)  
 adjOccNeighPat |= 1  
 adjOccNeighPat <<= 1   
}

Table 38 — Relative locations ( ds, dt, dv ) used in the computation of adjOccNeighPat

| *n* | ds | dt | dv |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | 0 | -1 |
| 6 | 1 | -1 | 0 |
| 7 | 0 | 1 | 1 |
| 8 | 0 | 1 | -1 |
| 9 | 0 | -1 | 1 |
| 10 | -1 | 1 | 0 |
| 11 | -1 | 0 | 1 |

The expression neighAvailable identifies whether there are spatial nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 38.

neighAvailable:= false  
for (n = 0; n < 7; n++)  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0)  
 neighAvailable |= adjOccC[n] ≠ 0

The expression primaryCtx discriminates by spatial nodes’ geometry primary information. The expression minorCtx discriminates by spatial nodes’ geometry minor information. primaryCtx and minorCtx are used to construct contextual information info1 and info2 used in Dynamic OBUF according to 9.2.11.7.3.

primaryCtx: = neighAvailable ? zoneCtxInc << 7 + prevPlanePosCtxInc << 6 + ((adjOccPlanePos[k] & 0x70) >> 4) << 3+ ((adjOccSinglePlane[k]&0x70) >> 4) :  
zoneCtxInc << 6 + prevPlanePosCtxInc << 5 + adjPlaneCtxInc << 3 + ((adjOccNeighPat & 0xe00) >> 9) + (0x01 << 7)

minorCtx: = neighAvailable ? adjPlaneCtxInc << 8 + (adjOccPlanePos[k] & 0xf) << 4 + (adjOccSinglePlane[k] & 0xf) :   
(adjOccNeighPat & 0x1ff) +(0x01 << 9)

*CtxIdxNeighPlanePos* is derived from the Dynamic\_OBUF based primaryCtx and minorCtx according to 9.2.11.7.3.

##### Coding occ\_plane\_pos[ 𝑘 ] using OUBF

* A first buffer is created for all OBUF instances according to 12.3. The OBUF buffer size obufBufferSize is set as 20000, the buffer depth obufLeafDepth of fully deployed trees is set as 4.
* The array of OBUF ACPMs is created according to 12.2.2.
* All OBUF instances are created according to 12.2 by using the octree buffer created, and the OBUF instances are divided into two types, sparse OBUF instances and non-sparse OBUF instances. Each type of OBUF instances has specific sizes nBit1 and nBit2 to specify the sizes of contextual information info1 and info2 used as input when calling the OBUF instance.
* The sizes nBit1 and nBit2 for sparse OUBF instances are set 10 and 8. For each axes, *ctxIdxMap*[][] is initialized with all the values set as 127.
* The array *ctxIdxMap*[ ][ ] corresponds to 8-bit context indices pointing (after right shift by 3) to OBUF ACPMs of the array *obufCtxArray*[ ].
* Then the primaryCtxand minorCtx (9.2.11.7.2) to derive the contextual information info1 and info2 for dynamic OBUF (12.5). primaryCtxis set as the first contextual information info1, and minorCtx is set as the second contextual information info2.
* Then, occ\_plane\_pos[ 𝑘 ] is decoded according to 12.4 by calling the OBUF instances of octree, and usinginfo1,info2, *ctxIdxMap*[ ][ ] and *obufCtxArray*[ ]. The type (sparse or non-sparse) of called OBUF instances depends on the primaryCtx *and* minorCtx(9.2.11.7.2), which indicates the spatial nodes’ geometry primary information and the minor information.

##### Case for angular-eligible axes

Contextualization of occ\_plane\_pos[ 𝑘 ] for nodes eligible for angular contextualization (AngularEligible is 1) is specified by 9.2.13.7.

#### Multiple axes eligibility

Eligibility for planar coding of multiple axes is specified by the expression MultiPlanarEligible.

NumEligibleAxes indicates the number of axes that are eligible for planar coding. It is used to determine the value of MultiPlanarEligible .

NumEligibleAxes = PlanarEligible[0] + PlanarEligible[1] + PlanarEligible[2]  
MultiPlanarEligible = NumEligibleAxes > 1 ? 1 : 0

#### occ\_single\_plane inference

When multi\_planar\_flag is present or when planar\_copy\_mode is present, inferences on the value of occ\_single\_plane[ 𝑘 ] can be made.

PlanarInferred[ 𝑘 ] indicates that occ\_single\_plane[ 𝑘 ] can be inferred (equal to 1) or not (equal to 0).

When planar\_copy\_mode equals to 1 or multi\_planar\_flag equals to 1, occ\_single\_plane[ 𝑘 ] can be inferred if 𝑘-th axis is eligible for planar coding.

if(planar\_copy\_mode)  
 for(k = 0; k < 3; k++)  
 PlanarInferred[k] = 1  
if(multi\_planar\_flag)  
 for(k = 0; k < 3; k++)  
 PlanarInferred[k] = PlanarEligible[k]

When planar\_copy\_mode equals to 0 and multi\_planar\_flag equals to 0, occ\_single\_plane[ 𝑘 ] can be inferred if 𝑘-th axis is eligible for planar coding and present NumEligibleAxes – 1 occ\_single\_plane[ j ] which equals to 1 and j < 𝑘.

if(¬multi\_planar\_flag){  
 count = 0  
 for(k = 0; k <3; k++){  
 for(j = 0; j < k; j++)  
 if(PlanarEligible[j])  
 count += occ\_single\_plane[j] ? 1 :0  
 PlanarInferred[k] = (count == NumEligibleAxes – 1) && PlanarEligible[k] ? 1 :0  
 }  
}

If PlanarInferred[ 𝑘 ] equal to 1, occ\_single\_plane[ 𝑘 ] will be inferred as follows:

if(PlanarInferred[k])  
 occ\_single\_plane[k] = planar\_copy\_mode ? PlaneRef[k] : multi\_planar\_flag

#### Determination of AllowPlanarCopyMode for planar\_copy\_mode

Determination of signalling of planar\_copy\_mode is specified by the expression AllowPlanarCopyMode.

AllowPlanarCopyMode = isInter && OccupancyIsPredictable && planarEligibile  
 where   
 planarEligible = PlanarEligible[0] || PlanarEligible[1] || PlanarEligible[2]

#### occ\_plane\_pos inference

When planar\_copy\_mode is present, inferences on the value of occ\_plane\_pos[ 𝑘 ] may be made.

PlanarPosInferred[ 𝑘 ] indicates that occ\_plane\_pos [ 𝑘 ] can be inferred (equal to 1) or not (equal to 0).

When AllowPlanarCopyMode equals to 1, occ\_plane\_pos[ 𝑘 ] can be inferred if planar copy mode is indicated or if the 𝑘-th axis is last eligible axis for planar coding and planar coding of previous axes match that of the reference frame.

prevDirMatch = true  
for(k = 0; k < 3; k++) {  
 if(planar\_copy\_mode)  
 PlanarPosInferred[k] = 1  
 else if(AllowPlanarCopyMode && k == lastDirIdx && prevDirMatch && PlaneRef[k])  
 PlanarPosInferred[k] = 1  
 else  
 PlanarPosInferred[k] = 0  
 prevDirMatch = prevDirMatch & (occ\_single\_plane[k] == PlaneRef[k]) && (occ\_plane\_pos[k] == PlanePosRef[k])  
}  
 where  
 lastDirIdx = PlanarEligible[2] ? 2 : (PlanarEligible[1] ? 1 : 0)

If PlanarPosInferred[ 𝑘 ] equal to 1, occ\_plane\_pos[ 𝑘 ] will be inferred as follows:

if(PlanarPosInferred[k])  
 occ\_plane\_pos[k] = planar\_copy\_mode ? PlanePosRef[k] : (PlanePosRef[k] ? 0 : 1)

#### Determination of CtxIdxPlanarCopyMode for planar\_copy\_mode

Contextualization of planar\_copy\_mode is specified by the expression CtxIdxPlanarCopyMode.

CtxIdxPlanarCopyMode = 8 × (4 × (eligibilityPlanarSum – 1) + matchDirSum) + planarRefSum  
 where  
 eligibilityPlanarSum = PlanarEligible[0] + PlanarEligible[1] + PlanarEligible[2]  
 matchedDirSum = matchedDir[0] + matchedDir[1] + matchedDir[2]  
 planeRefSum = PlaneRef[0] << 2 + PlaneRef[1] << 1 + PlaneRef[0]

The variable matchedDir[] is used to determine whether the planar information in the reference frame is equal to the planar information of the closest zone in the planar buffer.

matchedDir[k] = 0  
if(PlanarEligible[k])  
 matchedDir[k] = PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]] == PlanarRef[k]   
 && PrevOccPlanePos[k][PlanarNodeAxisLoc[k]] == PlanePosRef[k]

#### Derivation of PlanarRef and PlanePosRef

The variables PlanarRef and PlanePosRef denote the planar information of the collocated node in the reference frame.

PlanarRef[0] = (PredOccBitMap & 0x0f) ≠ (PredOccBitMap & 0xf0)  
PlanarRef[1] = (PredOccBitMap & 0x33) ≠ (PredOccBitMap & 0xcc)  
PlanarRef[2] = (PredOccBitMap & 0x55) ≠ (PredOccBitMap & 0xaa)  
PlanarPosRef[0] = PlanarRef[0] && (PredOccBitMap & 0xf0) > 0  
PlanarPosRef[1] = PlanarRef[1] && (PredOccBitMap & 0xcc) > 0  
PlanarPosRef[2] = PlanarRef[2] && (PredOccBitMap & 0xaa) > 0

### Direct nodes

#### General

Subclause 9.2.12 applies when occtree\_direct\_coding\_mode is not 0.

Certain occupancy tree nodes may immediately code point positions as a direct node, instead of coding a node occupancy bitmap for subsequent traversal. A direct node can represent either two distinct point positions, or a single position that is identical for every represented point.

A direct node codes a position as a residual relative to the node position.

The number of points coded in direct nodes is counted cumulatively, *DirectNodePointCnt*. The variable is initialized to 0 at the start of every slice.

Direct coding is limited to nodes that are both eligible and not prohibited by the planar direct node rate limit. Eligibility shall be determined for each occupancy tree node based upon the degree of spatial isolation as specified in 9.2.12.3.

#### Syntax element semantics

occ\_direct\_node equal to 1 specifies that the occupancy tree node is a direct node that codes the position of at least one point. When occ\_direct\_node is not present, it shall be inferred to be 0.

direct\_point\_cnt\_eq2 equal to 1 specifies that the direct node codes two point positions. direct\_point\_cnt\_eq2 equal to 0 specifies that the occupancy tree node codes a single point position for one or more points.

direct\_dup\_point\_cnt plus 1 specifies, when present, the number of points the direct node represents when direct\_point\_cnt\_eq2 is 0. When direct\_dup\_point\_cnt is not present, it shall be inferred to be 0.

direct\_joint\_prefix[ 𝑘 ] specifies a sequence of identical MSBs in the 𝑘-th component of two coded position residuals. The MSB position of the syntax element value indicates the number of position bits coded by the syntax element and does not form part of the reconstructed point position.

direct\_joint\_diff\_bit[ 𝑘 ] specifies, when direct\_joint\_prefix[ 𝑘 ] is present, the value of a bit in the binary representation of the 𝑘-th component of the two coded position residuals. The bit is the most significant non-identical bit of the two coded residual components. Its value is that for the first point. When direct\_joint\_diff\_bit[ 𝑘 ] is not present, it shall be inferred to be 0.

direct\_rem[ dnPt ][ 𝑘 ], direct\_rem\_st\_ang[ dnPt ] and direct\_rem\_v\_ang[ dnPt ] specify the remaining position bits of the dnPt-th point. When present, direct\_rem codes the 𝑘-th component, direct\_rem\_st\_ang either the 𝑠- or 𝑡-component depending upon the node location and direct\_rem\_v\_ang codes the 𝑣-component. When not present, they shall be inferred to be 0.

direct\_v\_ang\_resid\_abs[ dnPt ] and direct\_v\_ang\_resid\_sign[ dnPt ] together specify, when present, the residual of the 𝑣-component of dnPt-th point, for a prediction made from a selected beam’s elevation and vertical offset. When not present, they shall be inferred to be 0. The residual of the 𝑣-component of dnPt-th point is specified by the expression DirectVAngResid[ dnPt ].

DirectVAngResid[dnPt] :=  
 (1 − 2 × direct\_v\_ang\_resid\_sign[dnPt]) × direct\_v\_ang\_resid\_abs[dnPt]

beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ] together specify the index of an enumerated beam relative to a per-node prediction. The residual between the enumerated beam and the per node prediction is specified by the expression BeamIdxResid[ dnPt ].

BeamIdxResid[dnPt] := (1 − 2 × beam\_idx\_resid\_sign[dnPt]) × beam\_idx\_resid\_abs[dnPt]

The beam is used in the contextualization of the syntax elements direct\_rem\_st\_ang[ dnPt ], direct\_rem\_v\_ang[ dnPt ] and in 𝑣-component of dnPt-th point prediction to be used with DirectVAngResid[ dnPt ].

#### Eligibility

##### Decision for each occupancy tree node

Only certain occupancy tree nodes are eligible to be direct nodes. They are specified by the expression DirectModeEligible. An eligible node:

* is not the root node;
* is not the root node of a fully quantized subtree (9.2.14.2.6); and
* meets one of the following mode-dependent conditions:
  + When occtree\_direct\_coding\_mode is 1: if there are no nodes in the occupied neighbourhood pattern of the parent node, the coded node has no siblings and the parent node has at most one sibling.
  + When occtree\_direct\_coding\_mode is 2: if there are no nodes in the occupied neighbourhood pattern of the parent node.
  + When occtree\_direct\_coding\_mode is 3: if the coded node has at least one sibling.

DirectModeEligible := occtree\_direct\_coding\_mode > 0  
 && Dpth > 0  
 && MaxVec(QuantizedNodeSizeLog2) > 0 && ¬OccupancyIsPredictable  
 && ( (interNonAngular && DirectMode1Eligible) ||   
 (¬interNonAngular   
 && (occtree\_direct\_coding\_mode ≠ 1 || DirectMode1Eligible)  
 && (occtree\_direct\_coding\_mode ≠ 2 || DirectMode2Eligible)  
 && (occtree\_direct\_coding\_mode ≠ 3 || DirectMode3Eligible)   
 )  
 )  
 where,  
 interNonAngular = slice\_inter\_prediction && ¬geom\_angular\_enabled

DirectMode1Eligible := OccNeighPatEq0[Dpth − 1][NsP][NtP][NvP]  
 && OccNodeChildCnt[Dpth − 1][NsP][NtP][NvP] == 1  
 && (Dpth < 2 || OccNodeChildCnt[Dpth − 2][NsG][NtG][NvG] ≤ 2)

DirectMode2Eligible := OccNeighPatEq0[Dpth − 1][NsP][NtP][NvP]

DirectMode3Eligible := OccNodeChildCnt[Dpth − 1][NsP][NtP][NvP] > 1

When occtree\_inter\_angular\_direct\_coding\_enabled is equal to 1, an eligible node to be direct nodes meets the condition when DnEligibleByAng is equal to 1. (9.2.13.8.1).

DirectModeEligible := DnEligibleByAng

##### Presence of occ\_direct\_node

The syntax element occ\_direct\_node shall only be present in occupancy tree nodes that are both eligible for direct coding and not prohibited by the rate limit for direct nodes that applies when planar occupancy coding is enabled.

The direct node rate limit mask is specified by the expression DnPresenceMask[ 𝑖 ], 𝑖 ∈ 0 .. 31.

DnPresenceMask[i] := occtree\_planar\_enabled && occtree\_direct\_coding\_mode == 1  
 ? dnRate × i % 32 + (dnRate ≥ 32)  
 : 1  
 where  
 dnRate := occtree\_direct\_node\_rate\_minus1 + 1

The expression DirectNodePresent specifies the presence of the syntax element.

DirectNodePresent := DirectModeEligible && DnPresenceMask[(Dpth + DnEligibleCnt) % 32]

##### State variables

Eligibility is specified in terms of the following state variables; the indexes dpth, ns, nt and nv identify a node with location ( ns, nt, nv ) in the tree level at depth dpth:

* The sparse array OccNeighPatEq0; OccNeighPatEq0[ dpth ][ ns ][ nt ][ nv ] identifies whether the identified node has no nodes present in its occupied neighbourhood pattern.
* The sparse array OccNodeChildCnt; OccNodeChildCnt[ dpth ][ ns ][ nt ][ nv ] is the number of child nodes of the identified node.
* The variable DnEligibleCnt, a cumulative count of eligible nodes in a tree level.

##### Initial state

At the start of every occupancy\_tree syntax structure, the OccNodeChildCnt array shall be cleared; all elements of OccNodeChildCnt are unset.

##### State update at the start of every occupancy tree level

At the start of every occupancy\_tree\_level syntax structure, the count of eligible nodes DnEligibleCnt shall be set to zero.

##### State update after each coded occupancy tree node

This subclause applies at the end of every occupancy\_tree\_node syntax structure.

The number of child nodes and the presence of any nodes in the occupied neighbourhood pattern are recorded for use in subsequent eligibility decisions.

OccNodeChildCnt[Dpth][Ns][Nt][Nv] = direct\_node ? 0 : OccChildCnt  
OccNeighPatEq0[Dpth][Ns][Nt][Nv] = OccNeighPat == 0

If the node is eligible for direct coding, irrespective of the presence of occ\_direct\_node, the count of eligible nodes shall be incremented.

if (DirectModeEligible)  
 DnEligibleCnt++

#### Points represented by direct nodes

##### General

The unscaled positions for the points coded by the direct node are specified by the expression DnPtPos[ dnPt ][ 𝑘 ]. They are the concatenation of:

* the (quantized) node position,
* any bit corresponding to an occupied plane as determined by planar occupancy coding,
* any bits from joint direct position coding, and
* any remaining bits.

When geometry subtree scaling is enabled, direct nodes code partially quantized positions relative to the quantized node position.

DnPtPos[dnPt][k] := Nloc[k] << QuantizedNodeSizeLog2[k] | DnPtPosRem[dnPt][k]  
  
DnPtPosRem[dnPt][k] := DnPlanarPos[k] | DnJointPos[dnPt][k] | DnRemPos[dnPt][k]

DnPtPosS[dnPt] := DnPtPos[dnPt][0]  
DnPtPosT[dnPt] := DnPtPos[dnPt][1]  
DnPtPosV[dnPt] := DnPtPos[dnPt][2]

##### Output

At the end of the direct node, the coded points shall be scaled (9.2.14.6) and appended to the output point list:

if (occ\_direct\_node) {  
 for (dnPt = 0; dnPt ≤ direct\_point\_cnt\_eq2; dnPt++, PointCnt++, DirectNodePointCnt++)  
 for (k = 0; k < 3; k++)  
 PointPos[PointCnt][k] = OccPosScaleK(k, DnPtPos[dnPt][k])  
  
 for (i = 0; i < direct\_dup\_point\_cnt; i++, PointCnt++, DirectNodePointCnt++)  
 for (k = 0; k < 3; k++)  
 PointPos[PointCnt][k] = PointPos[PointCnt − 1][k]  
}

##### Planar-inferred position bits

When an axis is eligible for planar occupancy coding and it has a single occupied plane, the MSB of the position residual for that axis 𝑘 is specified by DnPlanarPos[ 𝑘 ], equal to the occupied plane location.

DnPlanarPos[k] := ¬PlanarFreeAxis[k] ? occ\_plane\_pos[k] << DnBitsAfterPlanar[k] : 0

The number of bits coded by each position residual exclusive of any bit derived from planar occupancy coding is specified for each component by the expression DnBitsAfterPlanar[ 𝑘 ].

DnBitsAfterPlanar[k] := QuantizedNodeSizeLog2[k] − ¬PlanarFreeAxis[k]

##### Joint coded position bits

The position residuals shall be jointly coded for an axis when the direct node codes two positions, joint coding is enabled and it has a residual bit to code. When angular coding is enabled, components coded by direct\_rem\_st\_ang and either direct\_rem\_v\_ang or direct\_v\_ang\_resid\_abs and direct\_v\_ang\_resid\_sign shall not be jointly coded.

DnJointCoded[k] := occtree\_direct\_joint\_coding\_enabled && direct\_point\_cnt\_eq2  
 && DnBitsAfterPlanar[k] > 0  
 && (¬geom\_angular\_enabled || k == (1 ^ AzimuthAxis))

The joint-coded bits for the two positions are specified by the expression DnJointPos[ dnPt ][ 𝑘 ]. For each component 𝑘, they comprise the common MSB (9.2.12.5.2) and the first divergent bit (9.2.12.5.3), if any. Joint coding is exclusive of any planar-inferred position bit.

DnJointPos[dnPt][k] :=  
 DnJointCoded[k] ? DnJointPosCommon[k] | DnJointPosDiffBit[dnPt][k] : 0

##### Remaining bits

The number of bits coded by each position residual exclusive of any bits derived from planar occupancy coding or joint coding is specified for each component by the expression DnRemBits[ 𝑘 ].

DnRemBits[k] := DnBitsAfterPlanar[k] − DnJointCommonBits[k] − DnJointDiffBits[k]

The expression DnRemPos[ dnPt ][ 𝑘 ] specifies the position bits coded by the syntax elements direct\_rem, direct\_rem\_st\_ang and direct\_rem\_v\_ang.

DnRemPos[dnPt][k] :=  
 geom\_angular\_enabled && k == AzimuthAxis ? direct\_rem\_st\_ang[dnPt] :  
 geom\_angular\_enabled && k == 2 ? occtree\_angular\_extension\_enabled ?  
 DirectRemVAngPred[dnPt] : direct\_rem\_v\_ang[dnPt] : direct\_rem[dnPt][k]

#### Joint coding of point positions

##### Point order

The coded order of two jointly coded points shall satisfy the constraint DnPtPosConstraint equal to 1:

* The 𝑠-coordinate of the first point shall be less than or equal to that of the second point.
* If the 𝑠-coordinates are equal, the 𝑡-coordinate of the first point shall be less than or equal to that of the second point.
* If both the 𝑠- and 𝑡-coordinates are equal, the 𝑣-coordinate of the first point shall be less than or equal to that of the second point

DnPtPosConstraint :=  
 DnPtPosS[0] < DnPtPosS[1]  
 || DnPtPosT[0] < DnPtPosT[1] && dnPtPosSameS  
 || DnPtPosV[0] ≤ DnPtPosV[1] && dnPtPosSameS && dnPtPosSameT  
 where  
 dnPtPosSameS := DnPtPosS[0] == DnPtPosS[1]  
 dnPtPosSameT := DnPtPosT[0] == DnPtPosT[1]

##### Common position bits

The number of position bits coded by direct\_joint\_prefix[ 𝑘 ] is specified by DnJointPrefixBits[ 𝑘 ].

DnJointPrefixBits[k] := DnJointCoded[k] ? IntLog2(direct\_joint\_prefix[k]) : 0

The number of bits coded by the 𝑘-th component of each position residual, exclusive of any bits derived from planar occupancy coding or a joint-coded prefix, is specified by the expression DnBitsAfterJointPrefix[ 𝑘 ].

DnBitsAfterJointPrefix[k] := DnBitsAfterPlanar[k] − DnJointCommonBits[k]

The expression DnJointPosPrefix[ 𝑘 ] specifies the value of the position bits coded by direct\_joint\_prefix[ 𝑘 ].

DnJointPosPrefix[k] :=  
 (direct\_joint\_prefix[k] ^ Exp2(DnJointPrefixBits[k])) << DnBitsAfterJointPrefix[k]

##### First divergent position bit

The existence of a divergent bit in the 𝑘-th component of the two jointly coded positions is specified by the expression DnJointPosDiffBits[ 𝑘 ]. A jointly coded divergent position bit exists if direct\_joint\_prefix[ 𝑘 ] is present and does not complete the coding of the 𝑘-th position component.

DnJointDiffBits[k] := DnJointCoded[k] ? DnBitsAfterPlanar[k] > DnJointPrefixBits[k] : 0

The value of the jointly coded divergent bit is specified for the 𝑘-th component of each point by the expression DnJointPosDiffBit[ dnPt ][ 𝑘 ].

DnJointPosDiffBit[dnPt][k] := bit << DnRemBits[k]  
 where bit :=  
 DnJointDiffBitPresent[k] ? direct\_joint\_diff\_bit[k] ^ dnPt :  
 DnJointDiffBitInferred[k] ? dnPt  
 : 0 /\* DnJointDiffBits[k] is 0 \*/

##### Presence of the syntax element direct\_joint\_diff\_bit

The presence of direct\_joint\_diff\_bit[ 𝑘 ] is specified by DnJointDiffBitPresent[ 𝑘 ].

DnJointDiffBitPresent[k] := DnJointDiffBits[k] > 0 && ¬DnJointDiffBitInferred[k]

It is present when a divergent bit position exists in the 𝑘-th component of the two jointly coded positions and the bit value cannot be inferred from the ordering constraint on jointly coded positions. The inference condition is specified by the expression DnJointDiffBitInferred[ 𝑘 ].

DnJointDiffBitInferred[k] :=  
 k == 0 && DnJointDiffBits[0] == 1  
 || k == 1 && DnJointDiffBits[1] == 1 && DnJointDiffBits[0] == 0  
 || k == 2 && DnJointDiffBits[2] == 1 && DnJointDiffBits[0] + DnJointDiffBits[1] == 0

#### Parsing of direct\_joint\_prefix

The binarization of direct\_joint\_prefix[ 𝑘 ] interleaves the prefix and suffix bins of the binarized exp-Golomb code (11.4.3). Every non-zero prefix bin shall be followed by a single suffix bin.

The binarization shall have no more than DnRemBits[ 𝑘 ] suffix bins.

maxVal = Exp2(DnRemBits[k])

The syntax element is parsed as follows; the variable BinIdx is the count of coded bins used for contextualization.

value = 1  
for (BinIdx = 0; value < maxVal && AeReadBin() == 1; BinIdx++)  
 BinIdx++  
 value = (value << 1) + AeReadBin()

### Angular coding

#### General

Subclause 9.2.13 applies when geom\_angular\_enabled is 1.

Angular coding in the occupancy tree specifies the contextualization for the planar occupied plane location occ\_plane\_pos, the direct position remainders direct\_rem\_st\_ang and direct\_rem\_v\_ang, the absolute value of the direct position vertical residual direct\_v\_ang\_resid\_abs and the sign of the direct position vertical residual direct\_v\_ang\_resid\_sign and the eligibility for direct coding. The contextualization uses rays cast from the angular origin.

#### Node position relative to the angular origin

The angular-origin-relative position of the occupancy tree node position is specified by the expression NposAng[ 𝑘 ].

NposAng[k] := AngPosScaleK(k, Nloc[k] << QuantizedNodeSizeLog2[k]) − AngularOrigin[k]

NposAngS := NposAng[0]  
NposAngT := NposAng[1]  
NposAngV := NposAng[2]

#### Azimuth coded axis

Contextualization using a beam's azimuth applies to either the S or T axis. The index of the contextualized axis is specified by the expression AzimuthAxis. It is determined using the angular-origin-relative node position.

AzimuthAxis := Abs(NposAngS) > Abs(NposAngT)

AzimuthAxisIsS := AzimuthAxis == 0  
AzimuthAxisIsT := AzimuthAxis == 1

#### State variables

Contextualization of occ\_plane\_pos[ AzimuthAxis ], beam\_idx\_resid\_abs[ dnPt ], beam\_idx\_resid\_sign[ dnPt ] and direct\_rem\_st\_ang is specified in terms of the following state variables; the index beamIdx identifies an SPS enumerated beam:

* The array BeamPrevPhiValid; BeamPrevPhiValid[ beamIdx ] indicates whether a value has been recorded by BeamPrevPhi[ beamIdx ].
* The array BeamPrevPhi; BeamPrevPhi[ beamIdx ] records the azimuth of the identified beam computed from the most recently coded syntax element direct\_rem\_st\_ang.
* The array BeamPrevIdxResid; BeamPrevIdxResid[ beamIdx ] records the residual BeamIdxResid[ dnPt ] between the enumerated beam DnBeamIdx[dnPt] and the per node prediction from estimated beam index DnBeamIdxEst, as specified from the most recently coded syntax elements beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ].

#### Initial state

At the start of every occupancy tree, all elements of BeamPrevPhiValid and all elements of BeamPrevIdxResid shall have the value 0.

#### Closest beam to a point

This subclause specifies the selection of the beam that can emit the closest rays to an angular-origin-relative point ( as, at, av ) by the expression BeamIdxEst[ as ][ at ][ av ]. The selection shall use rays cast from the angular origin.

When angular extension shall not be used or when performing beam selection for planar occupancy coding (9.2.13.7.3), the expression preciseBeamSelection is set equal to 0 and the selection is made without application of vertical beam displacements.

When angular extension shall be used and not performing beam selection for planar occupancy coding (9.2.13.7.3), the expression preciseBeamSelection is set equal to 1 and the selection is made by taking into account the vertical beam displacements.

BeamIdxEst[as][at][av] := num\_beams\_minus1 ?  
 (preciseBeamSelection ? PreciseBeamIdx : BeamIdxFromGrad[aGrad]) : 0

When preciseBeamSelection is equal to 0, the selection shall be performed by comparing the gradient of a ray specified by the expression aGrad to the elevation gradients of the enumerated beams. When preciseBeamSelection is equal to 1, the selection shall be performed by comparing for each enumerated beam the gradient of a ray specified by the expression displacedPointRayGrad[i] to the elevation gradient of the corresponding enumerated beam. According to the expression preciseBeamSelection, the ray is cast from the angular origin and passes either (when equal to 0) through the point ( as, at, av ), or (when equal to 1) through the point to which vertical beam displacement is applied ( as, at, av  + BeamOffsetV[i]).

Unless performing beam selection for planar occupancy coding (9.2.13.7.3), the expressions aGrad and preciseBeamSelection and displacedPointRayGrad[i] shall be defined as:

aGrad := av × IntRecipSqrt(rs × rs + rt × rv) >> 14  
 where  
 rs := as << 8  
 rt := at << 8

preciseBeamSelection := occtree\_angular\_extension\_enabled

displacedPointRayGrad[i] := ((av << 3) + BeamOffsetV[i])  
 × IntRecipSqrt(rs × rs + rt × rt) >> 17  
 where  
 rs := as << 8  
 rt := at << 8

When performing beam selection for planar occupancy coding (9.2.13.7.3), the expression aGrad shall be defined as:

aGrad := (2 × av − 1) × IntRecipSqrt(rs × rs + rt × rt) >> 15  
 where  
 rs := (as << 8) − 128  
 rt := (at << 8) − 128

preciseBeamSelection := 0

When preciseBeamSelection is equal to 0, the beam search is specified by the expression BeamIdxFromGrad. If the gradient rayGrad is half-way between that of two beams, the beam with the lower index shall be chosen.

BeamIdxFromGrad[rayGrad] :=  
 for (i = 1; i < num\_beams\_minus1; i++)  
 if (BeamElev[i] > rayGrad)  
 break  
 if (rayGrad − BeamElev[i − 1] ≤ BeamElev[i] − rayGrad)  
 i−−  
 BeamIdxFromGrad = i

When preciseBeamSelection is equal to 1, the precise beam search is specified by the expression PreciseBeamIdx. If the smallest value of gradientDistance[i] is obtained for at least two beams, the beam with the lower index shall be chosen.

PreciseBeamIdx :=  
 PreciseBeamIdx = 0  
 for (i = 1; i < num\_beams\_minus1 + 1; i++)  
 if (gradientDistance[i] < gradientDistance[PreciseBeamIdx])  
 PreciseBeamIdx = i  
 where  
 gradientDistance[i] := abs(displacedPointRayGrad[i] - BeamElev[i])

#### Application to occupied plane location coding

##### Node dimensions and midpoint

The expression ScaledNodeSize[ 𝑘 ] specifies the scaled volume dimensions of the coded node.

ScaledNodeSize[k] := AngPosScaleK(k, Exp2(QuantizedNodeSizeLog2[k]))

ScaledNodeSizeS := ScaledNodeSize[0]  
ScaledNodeSizeT := ScaledNodeSize[1]  
ScaledNodeSizeV := ScaledNodeSize[2]

The expression ScaledHalfNodeSize[ 𝑘 ] specifies the midpoint within the scaled volume of the coded node.

ScaledHalfNodeSize[k] := AngPosScaleK(k, Exp2(QuantizedNodeSizeLog2[k]) >> 1)

ScaledHalfNodeSizeS := ScaledHalfNodeSize[0]  
ScaledHalfNodeSizeT := ScaledHalfNodeSize[1]  
ScaledHalfNodeSizeV := ScaledHalfNodeSize[2]

1. When geometry scaling is disabled, ScaledNodeSize[ 𝑘 ] and ScaledHalfNodeSize[ 𝑘 ] are equal to Exp2( NodeSizeLog2[ 𝑘 ] ) and Exp2( NodeSizeLog2[ 𝑘 ] >> 1 ) respectively.

The midpoint coordinates of the coded node relative to the angular origin are specified by the expression NposAngMid[ 𝑘 ].

NposAngMid[k] := NposAng[k] + ScaledHalfNodeSize[k]

NposAngMidS := NposAngMid[0]  
NposAngMidT := NposAngMid[1]  
NposAngMidV := NposAngMid[2]

##### Eligibility

Only certain nodes are eligible for angular contextualization of the occupied plane location and inter direct coding. They are specified by the expression AngularEligible. Eligibility prevents the use of angular contextualization when a node is sufficiently large to be intersected by rays from multiple beams.

AngularEligible := geom\_angular\_enabled  
 && (¬num\_beams\_minus1 || CathetusV > (ScaledHalfNodeSizeV << 26))

Eligibility is specified in terms of:

* the smallest difference in elevation gradient between any two beams, BeamMinDeltaGrad;
* the V-axis distance subtended (CathetusV) by a ray with elevation gradient BeamMinDeltaGrad, over the distance in the S-T plane from the angular origin to the node midpoint; and
* the 𝑣-component of the scaled node size.

An example of an eligible node is illustrated in Figure 23.

手机屏幕截图

中度可信度描述已自动生成

Figure 23 — Illustration of an eligible node.

CathetusV := BeamMinDeltaGrad × ((rs + rt) >> 1)  
 where  
 rs := Abs((NposAngMidS << 8) − 128)  
 rt := Abs((NposAngMidT << 8) − 128)

The expression BeamMinDeltaGrad is the smallest difference in elevation gradient between any two beams.

BeamMinDeltaGrad :=  
 BeamMinDeltaGrad = BeamElev[0]  
 for (i = 1; i ≤ num\_beams\_minus1; i++) {  
 delta = BeamElev[i] – BeamElev[i − 1]  
 if (delta < BeamMinDeltaGrad)  
 BeamMinDeltaGrad = delta  
 }

Only certain nodes are eligible for azimuth direct coding. They are specified by the expression AzimuthEligible.

AzimuthEligible := AngularEligible && ((HalfDeltaAziAng << 1) ≤ phiStep)  
 where  
 phiStep:= 6588397 / BeamStepsPerRev[PlanarBeamIdx]  
  
HalfDeltaAziAng := Abs(azimuthLowMId - azimuthMid)  
 where  
 azimuthLowMId := AzimuthAxis ? IntAtan2(NposAngMidT, NposAngS)  
 : IntAtan2(NposAngT, NposAngMidS)  
 azimuthMid := IntAtan2(NposAngMidT, NposAngMidS)

##### Beam selection

Every node that is eligible for angular contextualization (AngularEligible == 1) shall select a beam for that purpose. The selected beam is specified by PlanarBeamIdx. It is either the same beam as used for the parent node or the closest beam to the node midpoint (9.2.13.6).

PlanarBeamIdx :=  
 inheritBeam ? NodeBeamIdx[Dpth − 1][NsP][NtP][NvP]  
 : BeamIdxEst[NposAngMidS][NposAngMidT][NposAngMidV]

The beam selection is inherited from the parent node when both the parent node has a valid beam and the node is not too small, as specified by the expression inheritBeam.

inheritBeam :=  
 NodeBeamValid[Dpth − 1][NsP][NtP][NvP] && CathetusV > (ScaledHalfNodeSizeV << 28)

##### State variables

Beam selection is specified in terms of the following state variables; the indexes dpth, ns, nt and nv identify a node with location ( ns, nt, nv ) in the tree level at depth dpth:

* The sparse array NodeBeamValid; NodeBeamValid[ dpth ][ ns ][ nt ][ nv ] indicates whether (when 1) the identified node has a valid beam selection. Unset elements are inferred to be 0.
* The sparse array NodeBeamIdx; NodeBeamIdx[ dpth ][ ns ][ nt ][ nv ] is the selected beam, if valid, for the identified node.

##### Initial state

At the start of every occupancy tree, elements of NodeBeamValid shall be unset.

##### State update

After selecting a beam (9.2.13.7.3), the beam index and its validity shall be recorded for inheritance by child nodes.

NodeBeamIdx[Dpth][Ns][Nt][Nv] = PlanarBeamIdx  
NodeBeamValid[Dpth][Ns][Nt][Nv] = 1

##### Contextualization of occ\_plane\_pos[ AzimuthAxis ]

The syntax element occ\_plane\_pos[ AzimuthAzis ] shall be contextualized according to 9.2.13.9 with the argument phiRangeMulLog2 equal to 2 and the arguments beamIdx, phiNominal, phiLow and phiHigh as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = PlanarBeamIdx

The arguments phiLow and phiHigh are the azimuthal angles for the angular-origin-relative coordinates of the lower corner of the occupancy tree node in the S-T plane for phiLow, and of the scaled node midpoint in the S-T plane for phiHigh. The argument phiNominal is a nominal azimuthal angle use during contextualization and it is set equal to phiLow.

phiLow = IntAtan2(NposAngT, NposAngS)  
phiHigh = IntAtan2(NposAngMidT, NposAngMidS)  
phiNominal = phiLow

##### Contextualization of occ\_plane\_pos[ 2 ]

The syntax element occ\_plane\_pos[ 2 ] shall be contextualized according to 9.2.13.10 with the argument elvIntvlGradDivLog2 equal to 2 and the arguments beamIdx, elvIntvlPosS, elvIntvlPosT, elvIntvlMidV and elvIntvlLenV as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = PlanarBeamIdx

The arguments elvIntvlPosS, elvIntvlPosT and elvIntvlMidV are the angular-origin-relative coordinates of the scaled node midpoint. The argument elvIntvlLenV is the length of the V-axis interval represented by the syntax element

elvIntvlPosS = NposAngMidS  
elvIntvlPosT = NposAngMidT  
elvIntvlMidV = NposAngMidV  
elvIntvlLenV = ScaledNodeSizeV

#### Application to direct node position coding

##### Eligibility

Only certain occupancy tree nodes are eligible to be direct nodes. They are specified by the expression DirectModeEligible (9.2.12.3.1).

When occtree\_inter\_angular\_direct\_coding\_enabled is equal to 1, eligibility for direct nodes shall be performed by the expression DnEligibleByAng.

DnEligibleByAng:= occtree\_inter\_angular\_direct\_coding\_enabled ?  
  (geom\_dup\_point\_counts\_enabled ?  
 AngularEligible && AzimuthEligible: AngularEligible || AzimuthEligible): 0

Only certain occupancy tree nodes are eligible to be inter direct nodes. They are specified by the expression interPredEligible.

The reference node indicates the colocated node in the reference frameof the current node.

*interNumPredPoints* indicates the number of points within the reference node of the current node.

interPredMode is true, when the reference node has at most one point or all points position are same.

interPredEligible is true, when the follwing conditions are true:

* *inter\_prediction\_enabled* is true and
* interPredMode is true and
* interNumPredPoints larger than 0

interPredEligible:=  
 inter\_prediction\_enabled&& interPredMode && interNumPredPoints >=0

##### Beam selection

Context selection for direct\_rem\_st\_ang[ dnPt ] and direct\_rem\_v\_ang[ dnPt ] and 𝑣-component of dnPt-th point prediction to be used with DirectVAngResid[ dnPt ] shall be performed using the beam specified by DnBeamIdx[ dnPt ]. For eligible node (9.2.12.3), an estimated beam index is determined for the dnPt-th point within the current node. The estimated beam index is specified by the expression DnBeamIdx[ dnPt ].

It is a requirement of bitstream conformance that DnBeamIdx[ dnPt ] shall be in the range 0 .. num\_beams\_minus1.

DnBeamIdx[dnPt] := (interPredEligible ? interDnBeamIdxEst:DnBeamIdxEst) + BeamIdxResid[dnPt]

Every direct node shall determine an estimated beam index as specified by the expression DnBeamIdxEst. The beam is estimated from the angular-origin-relative, scaled partial point position specified by dnBeamPosAng[ 𝑘 ].

DnBeamIdxEst := BeamIdxEst[beamPosAngS][beamPosAngT][beamPosAngV]  
 where  
 beamPosAngS := dnBeamPosAng[0]  
 beamPosAngT := dnBeamPosAng[1]  
 beamPosAngV := dnBeamPosAng[2]

dnBeamPosAng[k] := NposAng[k] + AngPosScaleK(k, DnPlanarPos[k] + dnPlanarPosHalf)  
 where  
 dnPlanarPosHalf := Exp2(DnBitsAfterPlanar[k]) >> 1

When interPredEligible is equal to 1, every point shall determine an estimated beam index as specified by the expression interDnBeamIdxEst. The beam is estimated from the angular-origin-relative, scaled reference predictor point position specified by interDnBeamPosAng[ 𝑘 ].

inter  
DnBeamIdxEst := BeamIdxEst[beamPosAngS][beamPosAngT][beamPosAngV]  
 where  
 beamPosAngS := interDnBeamPosAng[0]  
 beamPosAngT := interDnBeamPosAng[1]  
 beamPosAngV := interDnBeamPosAng[2]

interDnBeamPosAng[k]:= AngPosScaleK(k, interDnPlanarPos[k])  
 where  
 interDnPlanarPos:= PredGMPointCloud[*predDnPt*]   
 *predDnPt*:= dnPt< predEnd? dnPt : predEnd -1

##### Scaled angular-origin-relative point positions and partial point positions

Contextualization of direct\_rem\_st\_ang and direct\_rem\_v\_ang depends upon approximately scaled partially coded positions. A partially coded position is specified in terms of the MSBs of the complete coded position.

Approximate scaling accumulates applications of geometry scaling to each bit in the partially coded position.

The expression DnPtPosAng[ dnPt ][ 𝑘 ] is the 𝑘-th component of the scaled, coded point position relative to the angular origin.

DnPtPosAng[dnPt][k] := AngPosScaleK(k, DnPtPos[dnPt][k]) − AngularOrigin[k]

DnPtPosAngS[dnPt] := DnPtPosAng[dnPt][0]  
DnPtPosAngT[dnPt] := DnPtPosAng[dnPt][1]

The expression DnPartialPosAng[ dnPt ][ remBits ][ 𝑘 ] is the 𝑘-th component of the approximately scaled partially coded position for the dnPt-th coded point of the direct node. It excludes the remBits LSBs of DnPtPos[ dnPt ][ 𝑘 ].

DnPartialPosAng[dnPt][remBits][k] :=  
 DnPartialPosAng = NposAng[k]  
 for (i = remBits; i < QuantizedNodeSizeLog2[k]; i++)  
 DnPartialPosAng += AngPosScaleK(k, DnPtPosRem[dnPt][k] & Exp2(i))

##### Binarization of direct\_rem\_st\_ang and direct\_rem\_v\_ang

The direct\_rem\_st\_ang and direct\_rem\_v\_ang syntax elements shall be entropy coded using fixed-length binarization. The length in bins shall be equal to the log2 quantized node size less any bit inferred by the presence of a single occupied plane.

The expression DnRemAngBitsST is the length in bins of the syntax element direct\_rem\_st\_ang.

DnRemAngBitsST := DnBitsAfterPlanar[AzimuthAxis]

The expression DnRemAngBitsV is the length in bins of the syntax element direct\_rem\_v\_ang.

DnRemAngBitsV := DnBitsAfterPlanar[2]

##### Contextualization of direct\_rem\_st\_ang

Each bin, identified by BinIdx, of direct\_rem\_st\_ang[ dnPt ] shall be contextualized according to 9.2.13.9 with the argument phiRangeMulLog2 equal to 1 and the arguments beamIdx, phiNominal, phiLow and phiHigh as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = DnBeamIdx[dnPt]

The expression PhiLowST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the lower end of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it is computed from coordinates in the S-T plane, otherwise it may be estimated.

The expression PhiHighST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the higher end of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it has no utility, otherwise it may be estimated.

The expression PhiMidST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the mid point of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it is computed from coordinates in the S-T plane, otherwise it may be estimated.

The expressions PhiPosS[ remBits ] and PhiPosT[ remBits ] are the angular-origin-relative coordinates of the lower end of the interval coded by a bin in the S-T plane, the bin index being expressed by the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ].

PhiPosS[remBits] := AzimuthAxisIsS ? DnPartialPosAngS[dnPt][remBits] : DnPtPosAngS[dnPt]  
  
PhiPosT[remBits] := AzimuthAxisIsT ? DnPartialPosAngT[dnPt][remBits] : DnPtPosAngT[dnPt]

PhiLowST[remBits] :=  
 occtree\_angular\_extension\_enabled && remBits < DnRemAngBitsST ?  
 (prevBitIs1 ? PhiMidST[remBits + 1] : PhiLowST[remBits + 1]) :  
 IntAtan2(phiIntvlLowT, phiIntvlLowS)  
 where  
 prevBitIs1 := (direct\_rem\_st\_ang[dnPt] & (1 << remBits)  
 phiIntvlLowS := PhiPosS[remBits]  
 phiIntvlLowT := PhiPosT[remBits]

PhiHighST[remBits] :=  
 occtree\_angular\_extension\_enabled && remBits < DnRemAngBitsST ?  
 (prevBitIs1 ? PhiHighST[remBits + 1] : PhiMidST[remBits + 1]) :  
 IntAtan2(phiIntvlHighT, phiIntvlHighS)  
 where  
 prevBitIs1 := (direct\_rem\_st\_ang[dnPt] & (1 << remBits)  
 binIdx := DnRemAngBitsST - remBits  
 phiIntvlLen := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> binIdx)  
 phiIntvlHighS = phiPosS[remBits] + (AzimuthAxisIsS ? phiIntvlLen : 0)  
 phiIntvlHighT = phiPosT[remBits] + (AzimuthAxisIsT ? phiIntvlLen : 0)

PhiMidST[remBits] :=  
 occtree\_angular\_extension\_enabled  
 && (Abs(PhiLowST[remBits] - PhiHighST[remBits]) < phiMidInterpThreshold ?  
 PhiLowST[remBits] + PhiHighST[remBits] >> 1 :  
 IntAtan2(phiIntvlMidT, phiIntvlMidS)  
 where  
 binIdx := DnRemAngBitsST - remBits  
 phiMidInterpThreshold := 1 << 13  
 phiHalfIntvlLen := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> 1 + binIdx)  
 phiIntvlMidS = PhiPosS[remBits] + (AzimuthAxisIsS ? phiHalfIntvlLen : 0)  
 phiIntvlMidT = PhiPosT[remBits] + (AzimuthAxisIsT ? phiHalfIntvlLen : 0)

The expression phiPartialSTBits is the number of remaining bins for direct\_rem\_st\_ang[ dnPt ], including the bin identified by BinIdx.

phiPartialSTBits := DnRemAngBitsST – BinIdx

The expression phiOneFourth and phiThreeFourth are azimuthal angles estimated for angular-origin-relative coordinates of points estimated around one fourth and around three fourth of the interval in the S-T plane.

phiOneFourth := phiIntvlLow  
 + ((offset0 – offset1) × (phiIntvlMid - phiIntvlLow) >> phiPartialSTBits)  
phiThreeFourth := phiIntvlMid  
 + ((offset0 + offset1) × (phiIntvlMid - phiIntvlLow) >> phiPartialSTBits)  
 where  
 phiIntvlLow := PhiLowST[ phiPartialSTBits ]  
 phiIntvlMid := PhiMidST[ phiPartialSTBits ]  
 offset0 := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> 1 + BinIdx) - 1  
 offset1 := phiPartialSTBits > 1 ? phiPartialSTBits > 2 ? 0 : 1 : 2

When occupancy tree angular extension is enabled, the argument phiLow is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of a point estimated around one fourth of the interval in the S-T plane, otherwise phiLow is set equal to an azimuthal angle for angular-origin-relative coordinates of the lower end of the interval in the S-T plane. When occupancy tree angular extension is enabled, the argument phiHigh is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of a point estimated around three fourth of the interval in the S-T plane, otherwise phiHigh is set equal to an azimuthal angle for angular-origin-relative coordinates of the mid point of the interval in the S-T plane. When occupancy tree angular extension is enabled, the argument phiNominal is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of the mid point of the interval in the S-T plane, otherwise phiNominal is set equal to phiLow.

phiLow = occtree\_angular\_extension\_enabled ? phiOneFourth : PhiLowST[phiPartialSTBits]  
phiHigh = occtree\_angular\_extension\_enabled ? phiThreeFourth : PhiMidST[phiPartialSTBits]  
phiNominal = occtree\_angular\_extension\_enabled ? PhiMidST[phiPartialSTBits] : phiLow

##### State update after each direct\_rem\_st\_ang syntax element

After each coded direct\_rem\_st\_ang[ dnPt ] syntax element, the azimuth of the coded point shall be recorded as the azimuth of the selected beam.

BeamPrevPhi[DnBeamIdx[dnPt]] = PhiLowST[ 0]  
BeamPrevPhiValid[DnBeamIdx[dnPt]] = 1

##### Contextualization of direct\_rem\_v\_ang

Each bin, identified by BinIdx, of direct\_rem\_v\_ang[ dnPt ] shall be contextualized according to 9.2.13.10 with the arguments beamIdx, elvIntvlPosS, elvIntvlPosT, elvIntvlLenV, elvIntvlMidV and elvIntvlGradDivLog2 as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = DnBeamIdx[dnPt]

The arguments elvIntvlPosS and elvIntvlPosT are the angular-origin-relative coordinates in the S-T plane of the approximately scaled position of the coded point. The argument elvIntvlLenV is the length of the interval represented by the syntax element before division by Exp2( elvIntvlGradDivLog2 ).

elvIntvlPosS = AzimuthAxisIsS ? DnPartialPosAngS[dnPt][0] : DnPtPosAngS[dnPt]  
elvIntvlPosT = AzimuthAxisIsT ? DnPartialPosAngT[dnPt][0] : DnPtPosAngT[dnPt]  
elvIntvlLenV = AngPosScaleV(Exp2(DnRemAngBitsV))  
elvIntvlGradDivLog2 = BinIdx

1. When geometry scaling is disabled or the scaling step size is a power of two, DtPartialPosAng[ dnPt ][ 0 ][ 𝑘 ] is equal to DnPtPosAng[ dnPt ][ 𝑘 ].

The argument elvIntvlMidV specifies the midpoint of the V-axis interval relative to the angular origin. The expression elvIntvlHalfLen is the half-interval range. The start of the V-axis interval, elvPartialPosV, is the partially coded and approximately scaled point 𝑣-coordinate.

elvPartialPosV := DnPartialPosAngV[dnPt][DnRemAngBitsV − BinIdx]  
elvIntvlHalfLen := AngPosScaleV(Exp2(DnRemAngBitsV) >> 1 + BinIdx)  
elvIntvlMidV = elvPartialPosV + elvIntvlHalfLen

##### Binarization of direct\_v\_ang\_resid\_abs and direct\_v\_ang\_resid\_sign

When angular coding extension shall be used, direct\_v\_ang\_resid\_abs syntax element is the absolute value of a residual of a prediction made from the selected beam elevation and vertical offset and shall be entropy coded using a concatenated truncated unary and 𝑘-th order exp-Golomb code (11.4.4) and direct\_v\_ang\_resid\_sign is its sign and shall be entropy coded when the absolute value of the residual is not null.

The prediction of the 𝑣-component of dnPt-th point relative to the angular origin is specified by the expression DirectVAngPred[ dnPt ]. It clips a prediction made from the selected beam to fit in V-axis interval relative to the angular origin in which 𝑣-component of dnPt-th shall be refined.

DirectVAngPred[dnPt] :=  
 Min(Max(directVAngPred0[dnPt], dnPosAng), dnPosAng + dnRemNodeSizeV - 1)  
 where  
 dnPosAngV := NposAng[2] + AngPosScaleK(2, DnPlanarPos[2])  
 dnRemNodeSizeV := 1 << DnResVAngPredBits  
 directVAngPred0[dnPt] :=  
 DivExp2Fz(radius × BeamElev[beamIdx] - (BeamOffsetV[beamIdx] << 23), 26 )  
 where  
 radius := IntSqrt(rs × rs + rt × rt)  
 where  
 rs := elvIntvlPosS << 8  
 rt := elvIntvlPosT << 8

The arguments elvIntvlPosS and elvIntvlPosT are the angular-origin-relative coordinates in the S-T plane of the approximately scaled position of the coded point.

elvIntvlPosS = AzimuthAxisIsS ? DnPartialPosAngS[dnPt][0] : DnPtPosAngS[dnPt]  
elvIntvlPosT = AzimuthAxisIsT ? DnPartialPosAngT[dnPt][0] : DnPtPosAngT[dnPt]

The expression DnResVAngPredBits is the length in bits of the expression DirectRemVAngPred[ dnPt ].

DnResVAngPredBits := DnBitsAfterPlanar[2]

The expression DirectRemVAngPred[ dnPt ] specify the remaining position bits of the 𝑣-component of the dnPt-th point when angular coding extension shall be used.

DirectRemVAngPred[dnPt] :=  
 DirectVAngPred[dnPt] + DirectVAngResid[dnPt] - NposAng[2]

It is a requirement of bitstream conformance that DirectRemVAngPred[ dnPt ] shall be in the range 0 .. Exp2(DnResVAngPredBits) – 1.

##### State update after each **beam\_idx\_resid\_abs** and **beam\_idx\_resid\_sign** syntax elements

When angular extension coding shall be used, after each coded beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ] syntax elements, the residual between the enumerated beam and the per node prediction specified by the expression BeamIdxResid[ dnPt ] shall be recorded as the most recently coded residual of the selected beam.

if(occtree\_angular\_extension\_enabled)  
 BeamPrevIdxResid[DnBeamIdxEst] = BeamIdxResid[ dnPt ]

#### Determination of CtxIdxAngPhi for a bin according to beam azimuth

Each contextualized bin represents a half-closed interval along the azimuth coding axis AzimuthAxis located along the other S-T plane axis at a distance from the angular origin. The bin values 1 and 0 identify the upper and lower halves of the interval respectively.

Contextualization discriminates by the intersection of a predicted ray with one of eight parameterized azimuth ranges.

The process is parameterized by:

* beamIdx, the index of the beam that casts the predicted ray;
* phiNominal represents a nominal azimuthal angle for the coded point;
* phiLow and phiHigh represents (possibly estimated) azimuthal angles for a point in the first and second half of the half-closed interval;
* phiRangeMulLog2, a syntax element dependent constant used to define the azimuth discrimination ranges.

The predicted ray azimuth is derived from the azimuth of a reference point. The prediction quantizes the nominal azimuth to be an integer number of azimuth steps from the reference point.

The reference azimuth for the predicted ray is that of the previous direct-node-coded point recorded by the selected beam (9.2.13.8.2); if no previous point has been recorded by the selected beam, the reference azimuth is the nominal azimuth.

phiRef := BeamPrevPhiValid[beamIdx] ? BeamPrevPhi[beamIdx] : phiNominal

The expression phiPred specifies the predicted ray azimuth.

phiPred := phiRef − beamPhiStep[beamIdx] × phiSteps  
 where  
 phiSteps := DivExp2Up((phiRef − phiNominal) × BeamPhiStepRecip[beamIdx], 30)  
 beamPhiStep := 6588397 / BeamStepsPerRev[beamIdx]  
 beamPhiStepRecip := (BeamStepsPerRev[beamIdx] << 30) / 6588397

The context selection for the coded bin is specified by the expression CtxIdxAngPhi.

CtxIdxAngPhi := (a + 2 × b + 4 × c) × 3 + d  
 where  
 dPhiL := phiLow − phiPred  
 dPhiH := phiHigh − phiPred  
 a := (dPhiL ≥ 0) == (dPhiH ≥ 0)  
 b := Abs(dPhiL) > Abs(dPhiH)  
 c := Max(Abs(dPhiL), Abs(dPhiH)) > (Min(Abs(dPhiL),Abs(dPhiH)) << phiRangeMulLog2)  
 d := occtree\_angular\_extension\_enabled ?   
 (3 × phiStep < 4 × adPhiLH) + (phiStep < 2 × adPhiLH) : 0  
 where  
 phiStep := beamPhiStep[beamIdx]  
 adPhiLH := Abs(phiLow – phiHigh)

#### Determination of CtxIdxAngTheta for a bin according to beam elevation

Each contextualized bin represents a half-closed V-axis interval located at a radial distance in the S-T plane from the angular origin. The bin values 1 and 0 identify the upper and lower halves of the interval respectively.

Contextualization discriminates by the intersection of a predicted ray cast by a beam with one of four parameterized V-axis intervals.

The process is parameterized by:

* beamIdx, the index of the beam that casts the predicted ray;
* elvIntvlPosS and elvIntvlPosT, the coordinates of the V-axis interval in the S-T plane, relative to the angular origin;
* elvIntvlLenV, the length of the V-axis interval;
* elvIntvlMidV, the midpoint of the V-axis interval relative to the angular origin;
* elvIntvlGradDivLog2, the factor used to derive the sub-interval size.

The expression rRecip specifies the reciprocal radial distance of the V-axis interval in the S-T plane from the angular origin.

rRecip := IntRecipSqrt(rs × rs + rt × rt)  
 where  
 rs := (elvIntvlPosS << 8) − 128  
 rt := (elvIntvlPosT << 8) − 128

The expression elvIntvlMidGrad specifies the gradient of a ray cast from the angular origin that intersects the midpoint of the V-axis interval.

elvIntvlMidGrad := (2 × elvIntvlMidV − 1) × rRecip / Exp2(15)

The expressions esoTopGrad and esoBotGrad specify the gradients of rays cast from the angular origin that intersect the end points of a V-axis sub-interval.

esoIntvlGrad := elvIntvlLenV × rRecip >> 18 + elvIntvlGradDivLog2  
esoTopGrad := elvIntvlMidGrad + esoIntvlGrad  
esoBotGrad := elvIntvlMidGrad − esoIntvlGrad

The expression elvBeamGrad specifies the gradient of a ray cast from the angular origin that would intersect the predicted ray at the given radial distance.

elvBeamGrad := BeamGrad[beamIdx] + BeamOffsetV[beamIdx] × rRecip / Exp2(17)

The context selection for the coded bin is specified by the expression CtxIdxAngTheta, comparing the gradients of rays cast from the angular origin to those of the threshold points.

CtxIdxAngTheta := a + 2 × b  
 where  
 a := elvBeamGrad ≥ elvIntvlMidGrad  
 b := elvBeamGrad ≥ esoTopGrad || elvBeamGrad < esoBotGrad

#### Conversion from Cartesian to angular coordinates

This subclause specifies the conversion of decoded points' positions to angular coordinates.

A decoder is not required to perform the conversion unless angular coordinates are required for attribute decoding.

The angular-origin-relative position of every point is specified by the expression posAng[ ptIdx ][ 𝑘 ].

posAng[ptIdx][k] := PointPos[ptIdx][k] − AngularOrigin[k]

Every angular-origin-relative point position is converted to angular coordinates. The radial distance is the Euclidean distance to the point in the S-T plane; the azimuth angle is the anti-clockwise angle in an S-T plane between the positive S axis and the point; the indexed elevation is the index of the beam that casts the closest rays to the point (9.2.13.6). The conversion is:

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++){  
 as = posAng[ptIdx][0]  
 at = posAng[ptIdx][1]  
 av = posAng[ptIdx][2]  
 PointAng[ptIdx][0] = IntSqrt(as × as + at × at << 16) >> 8  
 PointAng[ptIdx][1] = IntAtan2(at << 8, as << 8) + 3294199 >> 8  
 PointAng[ptIdx][2] = BeamIdxEst[as][at][av]  
}

### Subtree scaling

#### Partially quantized coordinates

Subclause 9.2.14 applies when geom\_scaling\_enabled is 1.

Certain subtrees (9.2.14.2.1) can represent point positions using partially quantized coordinates that are parameterized by a scaling volume (9.2.14.2.2) and an applicable QP (9.2.14.5).

The partially quantized representation comprises two concatenated parts:

* An upper part that, when scaled, is the position of the scaling volume's lower corner, . Scaling is by the scaling volume size rounded down to the next power of two.
* A lower part that, when scaled, is a position relative to and within the scaling volume. Scaling is by the QP derived scale factor.

An example of the representation is illustrated by Figure 24. The coded point ● is represented within the scaling volume A. The scaling volume size is 12; rounded down to the next power of two as 8 (). The QP is 12; the derived scale factor (OccQpScale) is 3. The two parts of the point's partially quantized 𝑠-coordinate are marked U and L:

* The upper part, U, scaled by , is the position of A.
* The lower part, L, scaled by OccQpScale = 3 is the relative point position; for the S axis, 2 × 3 = 6. Scaling expands L by 1 bit (OccQpScaleLog2) to be 3 bits (OccScalingVolSizeLog2) in total.
* If U were 5, then would be 5 ×  = 40, and the point's 𝑠-coordinate would be 46.

图表, 图示

描述已自动生成

Key

|  |  |
| --- | --- |
| A | Scaling volume |
| U, L | Upper and lower parts of partially quantized representation |
|  | 𝑖-th bit |

Figure 24 — Example of partially quantized coordinates. Illustrated in 2D for the S-V plane.

#### Quantized subtrees

##### Identification

Subtrees that code partially quantized coordinates are started by:

* every node at depth OccQpSubtreeDepth;
* every direct node at a shallower depth than OccQpSubtreeDepth; in this case, the subtree consists only of the subtree root node as a direct node.

The lower parts of partially quantized point coordinates are represented by the location of a leaf node in the subtree or by a direct node that codes a subtree-relative position.

1. Identification of a subtree from a direct node at a shallower depth than OccQpSubtreeDepth happens after any planar occupancy coding for that node (9.2.14.2.5).

##### Subtree scaling volume

A scaling volume is identified by every quantized subtree. Its size depends upon the tree level in which the quantized subtree starts:

* If the quantized subtree starts at a shallower depth than OccQpSubtreeDepth and planar occupancy coding is enabled, the scaling volume dimensions are equal to those of the scaled child-node volume of the subtree root node.
* Otherwise, the scaling volume dimensions are identical to those of the scaled subtree root node volume.

The expression OccScalingVolSizeLog2[ 𝑘 ] specifies the integer log2 size of the scaling volume. The scaled dimensions (9.2.14.2.3) are RoundUp( OccVolScale × Exp2( OccScalingVolSizeLog2[ 𝑘 ] ) ).

OccScalingVolSizeLog2[k] :=  
 Dpth ≥ OccQpSubtreeDepth ? OccLvlNodeSizeLog2[OccQpSubtreeDepth[k] :  
 occtree\_planar\_enabled ? ChildNodeSizeLog2[k] : NodeSizeLog2[k]

For example, a quantized subtree root node identifies a volume with dimensions 32×16×32 and an applicable QP of 10 (OccQpScale = 2.5). The quantized node size would be 16×8×16 and the scaled node size 40×20×40.

##### Scaled occupancy tree node volume dimensions

A scaled node volume is the geometric expansion of the volume associated with a node in the general occupancy tree (9.2.2.1). The expansion is by a QP derived scale factor for the subtree, OccVolScale, and is relative to the lower corner, , of the subtree scaling volume. Scaling by OccVolScale rounds half-values up.

The quantized node size QuantizedNodeSize is a power-of-two contraction by OccQpScaleLog2 of the node size for ordinary nodes in the tree level.

OccQpScaleLog2 := OccQp / 8  
  
QuantizedNodeSizeLog2[k] := Max(0, NodeSizeLog2[k] − OccQpScaleLog2)  
  
QuantizedChildNodeSizeLog2[k] := Max(0, ChildNodeSizeLog2[k] − OccQpScaleLog2)

1. The scaled node volume dimensions are equivalent to the rounded expansion of the quantized node size by the subtree scale factor OccQpScale.

##### Quantized subtree nodes

A node in a quantized subtree shall identify the presence of at least one point contained within the scaled node volume.

Leaf nodes in the quantized subtree represent indivisible volumes with dimensions equal to the unit cube scaled by the subtree scale factor. Scaled point positions outside the unscaled subtree volume shall be clipped to be within it (hatched area in Figure 25). For example, point ▪︎ in the figure is clipped to ▫︎.

Two example subtrees are illustrated over three tree levels by Figure 25. The subtree root node volumes have dashed outlines; the coded nodes in each level have a thick, solid outline. The top subtree uses QP 12, the bottom QP 0. The position of the point ● is coded, starting from depth 𝑛, by the locations of the grey child nodes. The top subtree identifies its leaf node at depth 𝑛 + 1, the bottom subtree at depth 𝑛 + 2.

图示

描述已自动生成

Figure 25 — Example decomposition of two quantized subtrees.QP 12 (top) and QP 0 (bottom).

##### Direct nodes before OccQpSubtreeDepth

This subclause applies when planar occupancy coding is enabled.

Until a node is identified as a direct node through the coding of occ\_direct\_node, it shall not be considered to start a quantized subtree.

Coding operations that are specified to occur:

* before occ\_direct\_node shall use the ordinary node volume, equivalent to QP 0;
* after occ\_direct\_node shall use the scaled node volume derived from the direct node QP (DnQp),

i.e. the QP and scaled node volume dimensions change during the coding of the node.

This case is illustrated by Figure 26. When planar occupancy coding is enabled (left), the scaled node dimensions are initially equal to NodeSize, but changes to OccScalingVolSize after occ\_direct\_node.

手机屏幕截图

描述已自动生成

Figure 26 — Example subtree scaling volume for subtrees shallower than OccQpSubtreeDepth. (Left) planar occupancy coding enabled; (Right) disabled.

##### Fully quantized subtrees

Sufficiently large subtree QPs can cause a subtree root node to have a log2 quantized node size of 0×0×0. In this case, the subtree root node is a terminal node that:

* is ineligible to be a direct node (9.2.12.3.1)
* has its points represented by a single implicit child leaf node (9.2.6.3); the child node has the same dimensions as the root node – i.e. no axes are coded

#### Syntax element semantics

occ\_subtree\_qp\_offset\_present specifies whether (when 1) or not (when 0) per-subtree QP offsets are present in the tree level. QP offsets can only be present in a single tree level. When occ\_subtree\_qp\_offset\_present is not present, it shall be inferred to be 0.

occ\_subtree\_qp\_offset\_abs[ ns ][ nt ][ nv ] and occ\_subtree\_qp\_offset\_sign[ ns ][ nt ][ nv ] together specify an offset to the slice geometry QP used to scale point positions. The offset is specified by the expression OccSubtreeQpOffset[ ns ][ nt ][ nv ]. The offset applies to all points coded in the subtree of the node located at ( ns, nt, nv ) in the tree level at depth OccQpSubtreeDepth. When occ\_subtree\_qp\_offset\_sign[ ns ][ nt ][ nv ] is not present, it shall be inferred to be 0.

OccSubtreeQpOffset[ns][nt][nv] :=  
 (1 − 2 × occ\_subtree\_qp\_offset\_sign[ns][nt][nv]) × occ\_subtree\_qp\_offset\_abs[ns][nt][nv]

#### State variables

Occupancy tree scaling is specified in terms of the following state variable:

* The variable OccQpSubtreeDepth that identifies the tree level where occ\_subtree\_qp\_offset\_present is 1.

#### The subtree QP

##### General

The subtree QP is specified by the expression OccQp. It is determined for a coded node as:

* when geometry scaling is disabled: 0;
* when the node is a direct node at a depth shallower than OccQpSubtreeDepth: specified by 9.2.14.5.2;
* when the node is at a depth equal to or deeper than OccQpSubtreeDepth: specified by 9.2.14.5.3.

1. The provisions of 9.2.14.2.5 affect evaluations of OccQp during certain nodes.

OccQp :=  
 ¬geom\_scaling\_enabled ? 0 :  
 Dpth ≥ OccQpSubtreeDepth ? OccSubtreeQp  
 occ\_direct\_node ? DnQp : 0

##### Determination for a direct node before OccQpSubtreeDepth

The subtree QP is specified by the expression DnQp. It is the slice's direct-node QP limited by the smallest log2 scaling volume dimension.

DnQp := Min(dnSliceQp, OccSubtreeQpMax)  
 where  
 dnSliceQp := geom\_qp + occtree\_direct\_node\_qp\_offset << geom\_qp\_mul\_log2

##### Determination for nodes at or after OccQpSubtreeDepth

The subtree QP is specified by the expression OccSubtreeQp.

OccSubtreeQp := sliceQp + (OccSubtreeQpOffset[ss][st][sv] << geom\_qp\_mul\_log2)  
 where  
 sliceQp := geom\_qp + slice\_geom\_qp\_offset << geom\_qp\_mul\_log2  
 ss := OccSubtreeQpLoc[0]  
 st := OccSubtreeQpLoc[1]  
 sv := OccSubtreeQpLoc[2]

The expression OccSubtreeQpLoc is the subtree root node location in the tree level at depth OccQpSubtreeDepth for the node at ( Ns, Nt, Nv ).

OccSubtreeQpLoc[k] := Nloc[k] >> NodeSizeLog2[k] − OccScalingVolSizeLog2[k]

##### Maximum QP

The maximum QP for a subtree is specified by the expression OccSubtreeQpMax.

OccSubtreeQpMax := MinVec(OccScalingVolSizeLog2) × 8

It is a requirement of bitstream conformance that when occ\_subtree\_qp\_offset\_present is 1, OccSubtreeQp shall be in the range 0 .. OccSubtreeQpMax.

#### Scaling of a position component

This subclause specifies the scaling of the 𝑘-th partially quantized coordinate by the expressions:

* OccPosScaleK( 𝑘, posk ) that clips the scaled position to be within the subtree volume, and
* AngPosScaleK( 𝑘, posk ) that does not clip the scaled position.

1. When either geom\_scaling\_enabled or OccNodeQp is 0, the expressions OccPosScaleK( 𝑘, posk ) and AngPosScaleK( 𝑘, posk ) are both equal to posk.

The expressions upperPartQ and lowerPartQ represent the upper and lower parts of the partially quantized coordinate.

upperPartQ := posk >> OccScalingVolSizeLog2[k] − OccQpScaleLog2  
lowerPartQ := posk & Exp2(OccScalingVolSizeLog2[k] − OccQpScaleLog2) − 1

The 3-fractional-bit, fixed-point scale factor used to scale the low part is specified by the expression sF.

sF := 8 + (OccNodeQp & 7) << OccNodeQp / 8

The upper and lower parts are scaled as specified by the expressions upperPart and lowerPart.

upperPart := upperPartQ << OccScalingVolSizeLog2[k]  
lowerPart := DivExp2Up(lowerPartQ × sF, 3)

The scaled upper and lower parts are combined to produce the scaled position component:

OccPosScaleK(k, posk) := upperPart | Min(lowerPart, Exp2(OccScalingVolSizeLog2[k]) − 1)  
  
AngPosScaleK(k, posk) := upperPart + lowerPart

### Reference frame generation

#### General

Subclause 9.2.15 applies when inter\_prediction\_enabled is true. When biprediction\_enabled is 0, there are only I-frames or P-frames in the sequence. When biprediction\_enabled is 1 or 2, there may be I-frames, P-frames and B-frames in the sequence.

The following frame aliases are defined: PrevReconFrame is the previous reconstructed point cloud frame, PrevReconIPFrame is the previous reconstructed I-frame or P-frame, PrevReconBFrame is the previous reconstructed B-frame and PrevPrevReconIPFrame is the reconstructed I-frame or P-frame before the previously reconstructed I-frame or P-frame.

When the previous reconstructed point cloud frame is a B-frame (or not a B-frame), prevReconFrameIsBFrame is set to 1 (or 0).

The original reference point cloud frame PredPointCloud and (when slice\_biprediction is 1) the second reference point cloud frame SecondPredPointCloud are derived as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| biprediction\_enabled | slice\_biprediction | PredPointCloud | SecondPredPointCloud |
| 0 | - | PrevReconFrame | - |
| 1 | 0 | PrevReconIPFrame | - |
| 1 | 1 | prevReconFrameIsBFrame ? PrevReconBFrame  : PrevPrevReconIPFrame | PrevReconIPFrame |
| 2 | 1 | subclause 9.2.15.3 | subclause 9.2.15.3 |

The expression PredPointCloudBboxOrigin[ k ] is an alias for the 𝑘-th XYZ coordinate of the origin of the point cloud frame PredPointCloud. PredPointCloudBboxSize[ k ] is an alias for the 𝑘-th XYZ component of the volume dimensions of the point cloud frame PredPointCloud.

The output of this process is the compensated reference point cloud frame CompensatedPointCloud and (when slice\_biprediction is 1) the second compensated reference point cloud frame SecondCompensatedPointCloud.

When global\_motion\_enabled is equal to 0, CompensatedPointCloud is set equal to PredPointCloud and (when slice\_biprediction is 1) SecondCompensatedPointCloud is set equal to SecondPredPointCloud. When global\_motion\_enabled is equal to 1, CompensatedPointCloud and (when slice\_biprediction is 1) SecondCompensatedPointCloud are derived by invoking subclause 9.2.15.2.

When slice\_biprediction is 1 and frame\_merge\_enabled is 1, the two compensated reference point cloud frames are merged into one by invoking subclause 9.2.15.4, and the merged compensated reference point cloud frame is used in inter prediction.

#### Motion compensation of reference frame

This subclause specifies the generation of a motion compensated reference frame using the global motion parameters *ThresBot*, *ThresTop*, *AppliedGMRot*, *AppliedGMTrans* and partition type. The input of this process is the original reference point cloud, which is specified by OriPointCloud. The output of this process is the motion compensated point cloud, which is specified by *GMPointCloud*.

For the first reference point cloud frame, OriPointCloud is set equal to PredPointCloud. *ThresBot* is set equal to gm\_thres\_bot[0], *ThresTop* is set equal to gm\_thres\_top[0], *AppliedGMRot* is set equal to *GMMatrix* and *AppliedGMTrans* is set equal to gm\_trans[0]. After the motion compensation, CompensatedPointCloud is set equal to GMPointCloud.

For the second reference point cloud frame, OriPointCloud is set equal to SecondPredPointCloud. *ThresBot* is set equal to gm\_thres\_bot[1], *ThresTop* is set equal to gm\_thres\_top[1], *AppliedGMRot* is set equal to *GMMatrix2* and *AppliedGMTrans* is set equal to gm\_trans2[1]. After the motion compensation, SecondCompensatedPointCloud is set equal to GMPointCloud.

Let OriPntCnt be the number of points in the point cloud frame OriPointCloud.

When MotionPartitionType is 0, GMPointCloud is derived by invoking subclause 9.2.15.2.1. When MotionPartitionType is 1, GMPointCloud is derived by invoking subclause 9.2.15.2.2.

##### Motion compensation using road/object partitioning

This subclause specifies the generation of a motion compensated reference frame using global motion parameters and road/object partitioning.

The compensated point cloud is derived based on the thresholds *ThresTop* and *ThresBot*.

for (pointIdx=0; pointIdx < OriPntCnt; pointIdx++) {  
 pt = OriPointCloud[pointIdx]  
 if(pt[2] < ThresBot || pt[2] > ThresTop)  
 … /\* Apply global compensation to point pt and add compPt to GMPointCloud[pointIdx] (9.2.15.2.3) \*/\_  
 else  
 for (i=0; i < 3; i++)  
 GMPointCloud[pointIdx][i] = pt[i] >= 0 ? pt[i] : 0  
}

##### Motion compensation using cuboid partitioning

gm\_comp\_partition\_block[idx] specifies whether (when 1) or not (when 0) global motion compensation is to be applied to the idx-th partition block under cuboid partitioning.

This subclause specifies the generation of a motion compensated reference frame using global motion parameters and cuboid partitioning.

An intermediate compensated point cloud PredGMPointCloud is derived as follows:

for (pointIdx=0; pointIdx < OriPntCnt; pointIdx++) {  
 pt = OriPointCloud[pointIdx]  
 … /\* Apply global compensation to point pt and add compPt to PredGMPointCloud[pointIdx] (9.2.15.2.3) \*/\_  
}

Let PredGMPntCnt be the number of points in the point cloud PredGMPointCloud.

The compensated point cloud is derived based on gm\_comp\_partition\_block[blkIdx], which specifies whether motion compensation is applied for a block. When the point does not belong to a valid motion block, the value of blkIdx is -1.

compCloudPtIdx = 0  
for (n=0; n < PredGMPntCnt; n++) {  
 pt = PredGMPointCloud[n]  
 … /\* Derive motion block index for point pt and set it to blkIdx (9.2.15.2.4)\*/\_  
 if(blkIdx ¬= -1 && gm\_comp\_partition\_block[blkIdx])  
 for (i=0; i < 3; i++)  
 GMPointCloud[compCloudPtIdx++][i] = pt[i]  
}  
for (n=0; n < OriPnt; n++) {  
 pt = OriPointCloud[n]  
 … /\* Derive motion block index for point pt and set it to blkIdx (9.2.15.2.4)\*/\_  
 if(blkIdx ¬= -1 && ¬gm\_comp\_partition\_block[blkIdx])  
 for (i=0; i < 3; i++)  
 GMPointCloud[compCloudPtIdx++][i] = pt[i]  
}

##### Motion compensation for a point

This subclause specifies the generation of a motion compensated point compPt from point pt using the global motion matrix *AppliedGMRot* and global translation vector *AppliedGMTrans*.

for (i=0; i < 3; i++) {  
 compPt[i] = divExp2RoundHalfInPositiveShift (  
 rot[3×i] × offsetPt[0] + rot[3×i + 1] × offsetPt[1] + rot[3×i + 2] × offsetPt[2], 16, 1 << (15))+  
 tran[i] – minPosition[i]  
 compPt[i] = compPt[i] >= 0 ? compPt[i] : 0  
}  
 where,  
 rot[m] = AppliedGMRot[m]  
 tran[n] = AppliedGMTrans[n]  
 offsetPt[i] = pt[i] + minPosition[i]  
 where,  
 minPosition[i] = motion\_zero\_origin ? 0 : sps.seq\_origin\_xyz[i]

##### Deriving motion block index of point under cuboid partitioning

The motion block index blkIdx is derived for point pt under cuboid partitioning.

for(i=0; i > 3; i++)  
 idx[i] = motion\_block\_size[i] ? (pt[i]- PredPointCloudBboxOrigin[i]) / motion\_block\_size[i] : 0  
if (idx[0] < 0 || idx[0] >= NumMotionBlocksPerAxis[0] ||   
 idx[1] < 0 || idx[1] >= NumMotionBlocksPerAxis[1] ||   
 idx[2] < 0 || idx[2] >= NumMotionBlocksPerAxis[2])   
 blkIdx = -1  
else   
 blkIdx = (idx[0] × LPUnumInAxis[1] + idx[1]) × LPUnumInAxis[2] + idx[2]

#### Reference frames generation for B-frame in Hierarchical GOF structure

This subclause specifies the generation of reference frames for B-frame in Hierarchical GOF structure. The notional frame counters of the reconstructed I-frame or P-frame before the previously reconstructed I-frame or P-frame FrameCtrPrePreIP, the previously reconstructed I-frame FrameCtrPreIP or P-frame and the current frame FrameCtr are used to derived the reference relationship among the frames.

The size of group of frames GOFsize is determined by the frame distance between the previouly reconstructed I-frames or P-frames.

GOFsize := FrameCtrPreIP – FrameCtrPrePreIP

The hierarchical reference relationship among the frames with the notional frame counter value from FrameCtrPrePreIP to FrameCtrPreIP are derived based on a hierarchical reference GOF structure determined by GOFsize.

The notional frame counter of the B-frame to be coded *FrameCtrCur* is derived from the notional frame counter values of the reference frames *FrameCtrRef* and *FrameCtrSecondRef*.

FrameCtrCur = (FrameCtrRef + FrameCtrSecondRef) >> 1

*FrameCtrRef* and *FrameCtrSecondRef* are initialized by FrameCtrPrePreIP to FrameCtrPreI for the frames with the notional frame counter value from FrameCtrPrePreIP to FrameCtrPreIP. After one B-frame is determined, FrameCtrSecondRef is updated by FrameCtrCur to derived next B-frame, and then FrameCtrRef is updated by FrameCtrCur to derived next B-frame of the next B-frame. This process is iterated until the notional frame counter value difference between the reference frames is less than or equal to 1. An example of reference relationship among B-frames (GOFsize is 8) is shown in Table 39.

Table 39 — Example of reference relationship among point cloud frames in a Hierarchical GOF

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Notional frame counter | *a* | *a* +1 | *a* +2 | *a* +3 | *a* +4 | *a* +5 | *a* +6 | *a* +7 | *a* +8 |
| **Notional frame counters of first reference frame** | × | *a* | *a* | *a*+2 | *a* | *a*+4 | *a*+4 | *a*+6 | *a* |
| **Notional frame counters ofsecond reference frame** | × | *a*+2 | *a*+4 | *a*+4 | *a*+8 | *a*+6 | *a*+8 | *a*+8 | *a* |
| **Frame type (I/B/P)** | I | B | B | B | B | B | B | B | P |

An example of decoding order of B-frames (GOFsize is 8) is shown in Table 40.

Table 40 — Example of decoding order of B-frames in a Hierarchical GOF

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Notional frame counter of frame in decoding order | *a* | *a* +8 | *a* +4 | *a* +2 | *a* +1 | *a* +3 | *a* +6 | *a* +5 | *a* +7 |
| **Frame type (I/B/P)** | I | P | B | B | B | B | B | B | B |

For each B-frame, the reference frames are generated based on the derived reference relationship which is indicated by the notional frame counter values.

#### Frame merge mode for bi-prediction

When slice\_biprediction is 1 and frame\_merge\_enabled is 1, SecondCompensatedPointCloud is merged with CompensatedPointCloud. CompPntCnt and SecondCompPntCnt specify the numbers of points in CompensatedPointCloud and SecondCompensatedPointCloud before appending respectively.

for (pointIdx = 0; pointIdx < SecondCompPntCnt; pointIdx++) {  
 pt = SecondCompensatedPointCloud[pointIdx]   
 CompensatedPointCloud[pointIdx + CompPntCnt] = pt  
}

After merging the two compensated reference point cloud frames, only CompensatedPointCloud is used in inter prediction.

## Predictive tree

### General

Subclause 9.3 specifies the reconstruction of point positions from parsed predictive trees. It applies when geom\_tree\_type is 1.

The slice geometry can be represented by a sequence of one or more predictive trees of predictive tree nodes. Every tree node specifies a single position for one or more points. Traversal of a predictive tree is in depth-first order.

### Syntax element semantics

#### Predictive tree

slice\_ptree\_qp\_period\_log2\_offset specifies an offset to the GPS-signalled period at which predictive tree QP offsets are signalled. When slice\_ptree\_qp\_period\_log2\_offset is not present, it shall be inferred to be 0.

A QP offset is signalled once every Exp2( PtnQpInterval ) nodes.

PtnQpInterval := Exp2(ptree\_qp\_period\_log2 + slice\_ptree\_qp\_period\_log2\_offset)

ptn\_resid\_abs\_log2\_bits[ 𝑘 ] specifies the number of bins used for the fixed-length binarization of ptn\_resid\_abs\_log2[ 𝑘 ].

ptn\_radius\_min specifies the minimum angular radius coordinate for nodes where ptn\_pred\_mode is 0.

ptree\_end\_of\_slice specifies whether (when 0) or not (when 1) there is a subsequent predictive tree in the DU.

#### Predictive tree node

Nodes in the predictive trees are numbered and coded according to their position in the depth-first traversal order. The semantics specified in this subclause are for the nodeIdx-th coded node.

The syntax structure parameter depth is the tree depth of the node.

ptn\_qp\_offset\_abs[ nodeIdx ] and ptn\_qp\_offset\_sign[ nodeIdx ] together specify, when present and in accordance with PtnQpOffset[ nodeIdx ], an offset to the slice geometry QP used to scale point positions. ptn\_qp\_offset\_sign specifies whether (when 0) the offset's sign is positive or (when 1) negative. When ptn\_qp\_offset\_sign is not present, it shall be inferred to be 0.

PtnQpOffset[nodeIdx] :=  
 (1 − 2 × ptn\_qp\_offset\_sign[nodeIdx]) × ptn\_qp\_offset\_abs[nodeIdx]

The QP for a node is specified by the expression PtnQp[ nodeIdx ].

PtnQp[nodeIdx] :=  
 ¬geom\_scaling\_enabled ? 0 :  
 nodeIdx % PtnQpInterval ? PtnQp[nodeIdx − 1]  
 : sliceQp + PtnQpOffset[nodeIdx] << geom\_qp\_mul\_log2  
 where  
 sliceQp := geom\_qp + slice\_geom\_qp\_offset

It is a requirement of bitstream conformance that PtnQp[ 𝑖 ] shall be in the range 0 .. 167 for each node index 𝑖.

ptn\_dup\_point\_cnt[ nodeIdx ] plus 1 specifies the number of points represented by the node. When ptn\_dup\_point\_cnt[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_child\_cnt\_xor1[ nodeIdx ] xor 1 specifies the number of child nodes.

ptn\_inter\_flag[ nodeIdx ] equal to 1 specifies that position of the node is coded with inter prediction. When ptn\_inter\_flag[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_pred\_direction[ nodeIdx ] specifies whether (when 0) the first reference frame or (when 1) the second reference frame is used for inter prediction. When ptn\_pred\_direction[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_inter\_pred\_mode[ nodeIdx ] specifies the method used to predict the node’s coded point position using inter prediction.

ptn\_pred\_mode[ nodeIdx ] specifies the method used to predict the node's coded point position.

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), ptn\_pred\_mode[ nodeIdx ] is not present and shall be inferred to be 1.

ptn\_pred\_idx[ nodeIdx ] specifies the predictor index in a prediction list for angular coordinates to be used to predict the node's coded point position when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1).

ptn\_phi\_mul\_abs\_prefix[ nodeIdx ], ptn\_phi\_mul\_abs\_minus2[ nodeIdx ], ptn\_phi\_mul\_abs\_minus9[ nodeIdx ] and ptn\_phi\_mul\_sign[ nodeIdx ] together specify, in accordance with PtnPhiMul[ nodeIdx ], an integer number of ptree\_ang\_azimuth\_step\_minus1 + 1 steps that form an azimuth prediction residual. ptn\_phi\_mul\_sign specifies whether (when 0) the step change is in the anticlockwise or (when 1) clockwise direction. Any of ptn\_phi\_mul\_sign, ptn\_phi\_mul\_abs\_minus2 or ptn\_phi\_mul\_abs\_minus9 that are not present shall be inferred to be 0.

PtnPhiMul[nodeIdx] := (1 − 2 × ptn\_phi\_mul\_sign[nodeIdx]) × absVal  
 where  
 absVal := ptn\_phi\_mul\_abs\_prefix[nodeIdx]  
 + ptn\_phi\_mul\_abs\_minus2[nodeIdx] + ptn\_phi\_mul\_abs\_minus9[nodeIdx]

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), the expression BoundPhiResid[ nodeIdx ] provides for the point being coded a maximum boundary for the absolute value of the azimuthal angle residual. This value is determined in accordance with the radius coordinate residual residAng[ 0 ] and the predicted radius coordinate PtnPred[0 ] for that point, the azimuthal angle step size ptree\_ang\_azimuth\_step\_minus1 and the azimuthal angle precision ptree\_ang\_azimuth\_pi\_bits\_minus11.

BoundPhiResid[nodeIdx] := DivExp2Fz(scaledR × phiStep, azimuthTwoPiLog2 + 1)  
 where  
 scaledR := (PtnPred[0] + residAng[0]) × 8  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 azimuthTwoPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 12

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), the expression PtnPhiStep[ nodeIdx ] provides a dynamically adjusted azimuthal angle step for the point being coded. When ptree\_ang\_azimuth\_scaling\_enabled is 0 it provides a constant azimuthal angle step. This value is determined in accordance with the unitary azimuthal angle step size obtained from ptree\_ang\_azimuth\_step\_minus1 using the expression BoundPhiResid[ nodeIdx ], the radius coordinate residual residAng[ 0 ] and the predicted radius coordinate PtnPred[0 ] for that point.

PtnPhiStep[nodeIdx] :=  
 (ptree\_ang\_azimuth\_scaling\_enabled &&  
 radius && BoundPhiResid[nodeIdx] == 0 && (azimuthPiLog2 - phiStepArcLog2) > 0) ?  
 (ptree\_ang\_azimuth\_step\_minus1 + 1) << (azimuthPiLog2 - phiStepArcLog2) :  
 (ptree\_ang\_azimuth\_step\_minus1 + 1)  
 where  
 radius := PtnPred[0] + residAng[0]  
 phiStepArcLog2 := IntLog2(radius × (ptree\_ang\_azimuth\_step\_minus1 + 1))  
 azimuthPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11

ptn\_radius\_resid\_abs[ nodeIdx ], and ptn\_radius\_resid\_sign[ nodeIdx ] together specify, in accordance with PtnRadiusResidual[ nodeIdx ], the first component of the first coordinate-prediction residual, when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. ptn\_radius\_resid\_sign[ nodeIdx] specifies whether (when 0) the residual's sign is positive or (when 1) negative. When ptn\_radius\_resid\_sign[ nodeIdx ]is not present, it shall be inferred to be 0.

PtnRadiusResidual[nodeIdx] := (1 − 2 × ptn\_radius\_resid\_sign[nodeIdx]) × absVal[k]  
 where  
 absVal[k] := ptn\_radius\_resid\_abs[nodeIdx]

ptn\_phi\_resid\_abs\_gt0[ nodeIdx ], ptn\_phi\_resid\_sign[ nodeIdx ], ptn\_phi\_resid\_abs\_gt1[ nodeIdx ] and ptn\_phi\_resid\_abs\_rem[ nodeIdx ] together specify, in accordance with PtnPhiResidual[ nodeIdx ], the 2nd component of the first coordinate-prediction residual, when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. ptn\_phi\_resid\_sign[ nodeIdx] specifies whether (when 0) the residual's sign is positive or (when 1) negative. Any of ptn\_phi\_resid\_abs\_gt0[ nodeIdx ], ptn\_phi\_resid\_sign[ nodeIdx ], ptn\_phi\_resid\_abs\_gt1[ nodeIdx ] or ptn\_phi\_resid\_abs\_rem[ nodeIdx ] that are not present shall be inferred to be 0.

PtnPhiResidual[nodeIdx] := (1 − 2 × ptn\_phi\_resid\_sign[nodeIdx]) × absVal[k]  
 where  
 absVal[k] := ptn\_phi\_resid\_abs\_gt0[nodeIdx]  
 + ptn\_phi\_resid\_abs\_gt1[nodeIdx]  
 + ptn\_phi\_resid\_abs\_rem[nodeIdx]

ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_log2[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_rem[ nodeIdx ][ 𝑘 ] and ptn\_resid\_sign[ nodeIdx ][ 𝑘 ] together specify, in accordance with PtnResidual[ nodeIdx ][ 𝑘 ], the 𝑘-th component of the first coordinate-prediction residual. ptn\_resid\_sign[ nodeIdx][ 𝑘 ] specifies whether (when 0) the residual's sign is positive or (when 1) negative. Any of ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ], ptn\_resid\_sign[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_log2[ nodeIdx ][ 𝑘 ] or ptn\_resid\_abs\_rem[ nodeIdx ][ 𝑘 ] that are not present shall be inferred to be 0.

PtnResidual[nodeIdx][k] := (1 − 2 × ptn\_resid\_sign[nodeIdx][k]) × absVal[k]  
 where  
 absVal[k] := ptn\_resid\_abs\_gt0[nodeIdx][k]  
 + (Exp2(ptn\_resid\_abs\_log2[nodeIdx][k]) >> 1)  
 + ptn\_resid\_abs\_rem[nodeIdx][k]

ptn\_sec\_resid\_abs[ nodeIdx ][ 𝑘 ] and ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] together specify, in accordance with PtnSecResidual[ nodeIdx ][ 𝑘 ], the 𝑘-th component of the second coordinate-prediction residual that applies after conversion from angular to Cartesian coordinates. ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] specifies whether (when 0) the residual's sign is positive or (when 1) negative. If ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] is not present, it shall be inferred to be 0.

PtnSecResidual[nodeIdx][k] :=  
 (1 − 2 × ptn\_sec\_resid\_sign[nodeIdx][k]) × ptn\_sec\_resid\_abs[nodeIdx][k]

### Tree traversal for reconstruction of point positions

#### State variables

The reconstruction of point positions from predictive tree nodes is specified in terms of the following state variables:

* The array PtnStack, a stack of ancestor node indexes; PtnStack[ dpth ] is the node index of the ancestor at depth dpth in the predictive tree for the current node.

The array PtnPredList, a prediction list for angular coordinates; PtnPredList[ ptn\_pred\_idx[ nodeIdx ]][k] is used for predicting point's k-th angular coordinate when geom\_angular\_enabled is 1 and when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. The prediction list is dynamically updated after angular coordinates of points are coded (9.3.3.7).

* The variable PtnDepth, indicating the size of the stack of ancestor node indexes and equivalent to the depth of the current node in its predictive tree.
* The variable PtnIdx, the node index of the current node in the canonical traversal order.
* The variable PtnCnt, a count of nodes parsed from the bitstream.
* The variable PrevInterFrameRefIdx, providing the reference frame buffer index of the last inter decoded point.
* The variable PrevPhiResidSign, providing the sign of the last decoded non-null azimuthal angle residual.
* The variable PrevPhiMul, providing the value of the preceding decoded azimuthal angle step multiplier.
* The variable PrevRadiusResidSign, providing the sign of the last decoded non-null radius residual.

#### Initial state

The state variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul, PrevRadiusResidSign and InterFlagHist shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed by the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), the state variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul, PrevRadiusResidSign and InterFlagHist shall be set to 0.

#### Decoding a sequence of predictive trees

Each predictive tree is decoded according to the recursive application of 9.3.3.4, starting from its root node:

* At the start of each tree, the stack of ancestor node indexes is empty, PtnDepth = 0, and the array PtnPredList shall be initialized to 0.
* The root node of the first tree has node index PtnIdx = 0.
* The index of each subsequent tree's root node is the successor to the index of the last node in the preceding tree.

Predictive trees are decoded while PtnIdx is less than PtnCnt.

for (PtnIdx = 0; PtnIdx < PtnCnt; PtnIdx++) {  
 PtnDepth = 0  
 for (predIdx = 0; predIdx ≤ ptree\_ang\_max\_pred\_index; predIdx++)  
 for (k = 0; k < 2; k++)  
 PtnPredList[predIdx][k] = 0  
 … /\* decode predictive tree with root node index PtnIdx; see 9.3.3.3 \*/  
}

#### Recursive decoding of a subtree

This subclause specifies the reconstruction of a node with index PtnIdx and the remainder of its subtree:

* The point positions for the node are reconstructed and appended to the output point lists as specified by 9.3.4.
* When adaptive quantization step size of the predictive geometry azimuth angle is provided (when ptree\_ang\_azimuth\_scaling\_enabled is 1), the prediction list for angular coordinates shall be updated as specified by 9.3.3.7.
* When inter prediction is enabled (inter\_prediction\_enabled is 1), the inter prediction reference buffer shall be updated as specified by 9.3.3.10.
* The node's index is appended to the stack of ancestor node indexes.
* The subtrees for every child node are reconstructed in turn by the recursive application of this subclause: the index of the first child node is PtnIdx + 1; each subsequent child node index is the successor to the last node index of the preceding subtree.

PtnStack[PtnDepth++] = PtnIdx /\* push \*/

ptnChildCnt = ptn\_child\_cnt\_xor1[PtnNodeIdx] ^ 1  
for (i = 0; i < ptnChildCnt; i++) {  
 PtnIdx++  
 … /\* Recursively reconstruct the i−th child subtree (9.3.3.4) \*/  
}

After the node's subtree has been reconstructed, the node is removed from the stack of ancestor node indexes.

PtnDepth−− /\* pop \*/

#### Update of the state variable PrevPhiResidSign and PrevInterFrameRefIdx after each coded **ptn\_phi\_resid\_sign** syntax element

After each coded ptn\_phi\_resid\_sign syntax element, the state variable PrevPhiResidSign shall be set equal to the coded value of ptn\_phi\_resid\_sign.

After each coded ptn\_phi\_resid\_sign syntax element, the state variable PrevInterFrameRefIdx shall be set equal to (ptn\_inter\_pred\_mode > 1) when ptn\_inter\_flag is equal to 1.

#### Update of the state variables PrevPhiMul and PrevRadiusResidSign after each coded **ptn\_resid\_sign**[ nodeIdx ][0 ] syntax element

When ptree\_ang\_azimuth\_scaling\_enabled is 1, after each coded ptn\_resid\_sign[ nodeIdx ][0 ] syntax element (i.e. after each code radius residual sign), the state variable PrevPhiMul shall be set equal to the expression PtnPhiMul[ nodeIdx ], and the state variable PrevRadiusResidSign shall be set equal to the coded value of ptn\_resid\_sign[ nodeIdx ][0 ].

#### Dynamic update of the prediction list for angular coordinates

After the point position for a node is decoded, the prediction list for angular coordinates shall be updated according to the value of the decoded radius residual residAng[ 0 ] and the threshold ptree\_ang\_pred\_list\_radius\_resid\_threshold:

if (ptree\_ang\_azimuth\_scaling\_enabled) {  
 for (predIdx = predIdxRemoved; predIdx > 0; predIdx--)  
 for(k = 0; k < 2; k ++)  
 PtnPredList[predIdx][k] = PtnPredList[predIdx - 1][k]  
 for(k = 0; k < 2; k ++)  
 PtnPredList[0][k] = PtnAng[k]  
}  
 where  
 predIdxRemoved := flagNewObject ? ptree\_ang\_max\_pred\_index : ptnPredIdx  
 where  
 ptnPredIdx := ptn\_pred\_idx[PtnIdx]  
 flagNewObject := Abs(residAng[0]) > ptree\_ang\_pred\_list\_radius\_resid\_threshold

#### Update of the state variable InterFlagHist after each coded **ptn\_inter\_flag**[ nodeIdx ] syntax element

When slice\_inter\_prediction is 1, after each coded ptn\_inter\_flag[ nodeIdx ] syntax element, the state variable InterFlagHist shall be set equal to the expression InterFlagHist << 1 & ptn\_inter\_flag[ nodeIdx ].

#### Generate inter prediction list for angular coordinates

After the point position for a node is decoded, the inter prediction list for angular coordinates shall be updated according to the value of the decoded azimuth and beam ID. The subclause 9.3.3.9.1 is invoked with PtnRefFrame and (when slice\_inter\_frame\_ref\_gmc[0] is 1) PtnAltRefFrame as input and inter predictor list PtnInterPredList as output. When slice\_biprediction is 1, subclause 9.3.3.9.1 is invoked with PtnSecondRefFrame and (when slice\_inter\_frame\_ref\_gmc[1] is 1) PtnSecondAltRefFrame as input and inter predictor list PtnSecondInterPredList as output.

##### Update of inter prediction list

The input to this subclause are reference frames RefFrame and in some case AltRefFrame and the output is the inter predictor list InterPredList.

cnt = 0  
for (qAzim = stAzim; qAzim <= MaxQAzim && cnt < 2; qAzim++)  
 if (PtnRefFrame[beamId][qAzim][0] != -1) {  
 for (k = 0; k < 3; k++)  
 InterPredList[cnt][k] = RefFrame[beamId][qAzim][k]  
 cnt++  
 }  
while(cnt < 2)  
 InterPredList[cnt++] = na  
if (global\_motion\_enabled)  
 for (qAzim = stAzim; qAzim <= MaxQAzim && cnt < 4; qAzim++)  
 if (AltRefFrame[beamId][qAzim][0] != -1) {  
 InterPredList[cnt][0] = AltRefFrame[beamId][qAzim][0]  
 if(slice\_inter\_frame\_ref\_gmc){  
 deltaPhi = AltRefFrame[beamId][qAzim][1] – phiP  
 InterPredList[cnt][1] = phiP  
 if(deltaPhi >= (phiStep >> 1) ||  
 deltaPhi <= -(phiStep >> 1)){  
 phiMul = Div((deltaPhi) + (phiStep >> 1), phiStep, 0)  
 InterPredList[cnt][1] += phiMul × phiStep  
 }  
 }  
 else  
 InterPredList[cnt][1] = AltRefFrame[beamId][qAzim][1]  
 InterPredList[cnt][2] = AltRefFrame[beamId][qAzim][2]  
 cnt++  
 }  
 while(cnt < 4)  
 InterPredList[cnt++] = na  
 where  
 beamId := PtnAng[2]  
 stAzim := DivExp2Fz(PtnAng[1], inter\_azim\_scale\_log2)  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 phiP := PointAng[ptIdx][1]  
 where  
 ptIdx := PtnStack[PtnDepth – 1]

#### Generation of derived reference frame

Subclause 9.3.3.10 applies to generate the derived reference frame(s).

The original reference frame is specified by PtnOriRefFrame.

When biprediction\_enabled is 0, PtnOriRefFrame is set equal to PtnCurrFramePos.

When biprediction\_enabled is 1 or 2, the reconstructed frame PtnCurrFramePos is indicated by the notional frame counter to be used as the reference frame for subsequent point cloud frames. PtnOriRefFrame is determined as follows:

* when slice\_biprediction is 0, PtnOriRefFrame is set equal to the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 1 and slice\_biprediction is 1, if the previously reconstructed point cloud frame is a B-frame, PtnOriRefFrame is set equal to the previously reconstructed B-frame; otherwise, PtnOriRefFrame is set equal to the I-frame or P-frame before the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 2 and slice\_biprediction is 1, PtnOriRefFrame is set equal to the first reference frame derived by invoking subclause 9.2.15.3.

The first derived reference frame PtnRefFrame and (when global\_motion\_enabled is 1) the second derived reference frame PtnAltRefFrame are derived as follows:

* when global\_motion\_enabled is 0, PtnRefFrame is set equal to PtnOriRefFram;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc is 0, PtnAltRefFrame is set equal to PtnRefFrame and then PtnRefFrame is set equal to PtnOriRefFrame;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc is 1, the global motion compensation is applied to points in PtnOriRefFrame as follows:

if (global\_motion\_enabled)  
 for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnOriRefFrame[beamId][qAzim][0] != -1) {   
 … /\* Apply global compensation to point pt and add compensated point to PtnGlobFrame (9.3.3.12) \*/\_  
 }  
 where  
 pt[k] = PtnOriRefFrame[beamId][qAzim][k]

When resampling\_enabled is 1, the radius values of points in PtnOriRefFrame are updated using global motion compensated points as follows:

for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnOriRefFrame[beamId][qAzim][0] != -1) {  
 … /\* Update radius of point pt in PtnOriRefFrame (9.3.3.13) \*/\_  
 }  
 where  
 pt[k] = PtnOriRefFrame[beamId][qAzim][k]

PtnRefFrame is set equal to PtnOriRefFrame and PtnAltRefFrame is set equal to PtnGlobFrame.

#### Generation of derived reference frame for the second reference frame

When biprediction\_enabled is 1 or 2, and slice\_biprediction is 1, the second original reference frame PtnSecondOriRefFrame is determined as follows.

* when biprediction\_enabled is 1, PtnSecondOriRefFrame is set equal to the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 2, PtnSecondOriRefFrame is set equal to the second reference frame derived by invoking subclause 9.2.15.3.

The corresponding first derived reference frame PtnSecondRefFrame and (when global\_motion\_enabled is 1) the second derived reference frame PtnSecondAltRefFrame are derived as follows:

* when global\_motion\_enabled is 0, PtnSecondRefFrame is set equal to PtnSecondOriRefFram;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc2 is 0, PtnSecondAltRefFrame is set equal to PtnSecondRefFrame and then PtnSecondRefFrame is set equal to PtnSecondOriRefFrame;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc2 is 1, the global motion compensation is applied to points in PtnSecondOriRefFrame as follows:

if (global\_motion\_enabled)  
 for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnSecondOriRefFrame[beamId][qAzim][0] != -1) {   
 … /\* Apply global compensation to point pt and add compensated point to PtnSecondGlobFrame (9.3.3.12) \*/\_  
 }  
 where  
 pt[k] = PtnSecondOriRefFrame[beamId][qAzim][k]

When resampling\_enabled is 1, the radius values of points in PtnSecondOriRefFrame are updated using global motion compensated points as follows:

for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnSecondOriRefFrame[beamId][qAzim][0] != -1) {  
 … /\* Update radius of point pt in PtnSecondOriRefFrame (9.3.3.13) \*/\_  
 }  
 where  
 pt[k] = PtnSecondOriRefFrame[beamId][qAzim][k]

PtnSecondRefFrame is set equal to PtnSecondOriRefFrame and PtnSecondAltRefFrame is set equal to PtnGlobFrame.

#### Global motion compensation

The spherical to cartesian coordinate conversion of point is applied as follows, the slice-relative angular origin in STV coordinates of the reference frame is specified by the expression AngularOriginRef[ 𝑘 ]:

ptCart[k] := AngularOriginRef[k] + (  
 k == 0 ? DivExp2Fz(ρ × IntCos(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24):  
 k == 1 ? DivExp2Fz(ρ × IntSin(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24):  
 k == 2 ? DivExp2Fz(DivExp2Fz(BeamElev[i] × ρ, 15) − BeamOffsetV[i], 3): na)  
 where  
 ρ := pt[0] << ptree\_ang\_radius\_scale\_log2  
 φ := pt[1]  
 i := pt[2]

The global motion compensation of a point is applied based on the threshold values *ThresBot* and *ThresTop*, global motion *AppliedGMMatrix* and *AppliedGMTrans* as follows:

if(ptCart[2] < ThresBot || ptCart[2] > ThresTop)  
 for(k = 0; k < 3; k++)  
 ptCartRef[k] = DivExp2Fz(ptCartComp[k], 16) + AppliedGMTrans[k]  
 where  
 ptCartComp[k] := AppliedGMMatrix[k][0] × ptCart[0] + AppliedGMMatrix[k][1] × ptCart[1] + AppliedGMMatrix[k][2] × ptCart[2]

For the first reference frame, *ThresBot* is set equal to gm\_thres\_bot, *ThresTop* is set equal to gm\_thres\_top, *AppliedGMMatrix* is set equal to GMMatrix and *AppliedGMTrans* is set equal to gm\_trans. When slice\_biprediction is 1, *ThresBot* is set equal to gm\_thres\_bot2, *ThresTop* is set equal to gm\_thres\_top2, *AppliedGMMatrix* is set equal to GMMatrix2 and *AppliedGMTrans* is set equal to gm\_trans2.

The compensated point is then converted back into angular domain as follows:

ptSph[0] = IntSqrt(ptC[0] × ptC[0] + ptC[1] × ptC[1] << 16) >> 8  
ptSph[1] = IntAtan2(ptC[1] << 8, ptC[0] << 8)  
ptSph[1] = (((ptSph[1] + 3294199) × 5340354 + off) >> sh) - (1 << azimLog2)  
ptSph[2] = BeamIdxEst[ptC[0]][ptC[1]][ptC[2]]  
 where  
 ptC[k] = ptCartRef[k]  
 azimLog2 = ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11  
 sh = 44 – azimLog2  
 off = 1 << sh – 1

The compensated point is then updated to buffer PtnGlobFrame as follows:

if (PtnGlobFrame[beamId][qAzim][0] == -1 || PtnGlobFrame[beamId][qAzim][0] > ptSph[0])  
 for(k = 0; k < 3; k ++)  
 PtnGlobFrame[beamId][qAzim][k] = ptSph[k]

#### Update radius after compensation

The final derived reference frame is generated as follows:

if (PtnGlobFrame[beamId][qAzim][0] != -1) {  
 for(k = 0; k < 2; k++) {  
 ptA[k] = PtnGlobFrame[beamId][qAzim][k]  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 }  
 if (ptA[1] < pt[1]) {  
 for(qA = qAzim + 1; qA <= MaxQAzim; qA++)  
 if (PtnGlobFrame[beamId][qA][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 } else if (ptA[1] > pt[1])  
 for(qB = qAzim - 1; qB >= MinQAzim; qB--)  
 if(PtnGlobFrame[beamId][qB][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 }  
} else {  
 for(qA = qAzim + 1; qA <= MaxQAzim; qA++)  
 if (PtnGlobFrame[beamId][qA][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptA[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 for(qB = qAzim - 1; qB >= MinQAzim; qB--)  
 if(PtnGlobFrame[beamId][qB][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
}

When both ptA and ptB are not initialized (there is no point PtnGlobFrame[beamId] corresponding to beamId) the radius is not updated. When only one of ptA and ptB is initialized, both ptA and ptB are set equal to the initialized point as follows:

if (qA == MaxQAzim + 1 && qB != MinQAzim – 1)  
 ptA = ptB  
else if (qA != MaxQAzim + 1 && qB == MinQAzim – 1)  
 ptB = ptA

Once points ptA and ptB are obtained, the radius is updated as follows:

if (!del0 || !del1)  
 pt[0] = ptA[0]  
else  
 pt[0] = ptA[0] + (1 – 2×sgn) × (abs(nr) + (abs(dr) >> 1)) / abs(dr)  
 where  
 nr = del0 × (pt[1] – ptA[1])  
 dr = del1  
 sgn = !((nr > 0 && dr > 0) || (nr < 0 && dr < 0))  
 where  
 del0 = ptA[0] – ptB[0]  
 del1 = ptA[1] – ptB[1]

### Reconstruction of point coordinates

#### General

Subclause 9.3.4 specifies the reconstruction of the point positions for a node with index PtnIdx at depth PtnDepth of a predictive tree.

#### Reconstructed STV coordinates

The node's reconstructed STV coordinates are specified by the expression PtnPos[ 𝑘 ]. They are the sum of a prediction (predStv) and a scaled residual (residStv), with negative coordinates clipped to 0. The predicted position is:

* when angular geometry coding is disabled: derived from the reconstructed STV coordinates of ancestor nodes (9.3.4.6);
* when angular geometry coding is enabled: a conversion from the node's reconstructed angular coordinates (9.3.4.5).

PtnPos[k] := Max(0, predStv[k] + residStv[k])  
 where  
 predStv[k] := geom\_angular\_enabled ? PtnAngStv[k] : PtnPred[k]

The scaled STV coordinate residuals, specified by residStv[ 𝑘 ], are:

* 0, when second coordinate residual coding is disabled; or otherwise
* coded by the node's first residual when angular geometry coding is disabled, and by its second residual when enabled; and
* scaled by the 3-fractional-bit, fixed-point, geometry scale factor specified by the expression sf for the node's QP; scaling shall round to the nearest integer with half-values rounded up.

residStv[k] := DivExp2Up(resid[k] × sf, 3)  
 where  
 sf := 8 + (PtnQp[PtnIdx] & 7) << PtnQp[PtnIdx] / 8  
 resid[k] := geom\_angular\_enabled ? ptree\_sec\_resid\_disabled ? 0  
 : PtnSecResidual[PtnIdx][k]  
 : PtnResidual[PtnIdx][k]

#### Reconstructed RPI node coordinates

When angular geometry coding is enabled, the node's reconstructed RPI coordinates are specified by the expression PtnAng[ 𝑘 ]. They are the sum of an angular coordinate prediction (9.3.4.6) and a residual (residAng).

It is a requirement of bitstream conformance that PtnAng[ 2 ] shall be in the range 0 .. num\_beams\_minus1.

PtnAng[k] := PtnPred[k] + residAng[k]

The residual RPI coordinates specified by residAng[ 𝑘 ] are the sum of a 𝜑-component offset, phiOffset and values PtnResiAng of primary residual.

residAng[k] := (k == 1 ? phiOffset : 0) + PtnResiAng[k]  
 where  
 phiOffset := phiOffset0 + phiOffset1  
 PtnResiAng[k] :=  
 k == 0 && ptree\_ang\_azimuth\_scaling\_enabled ? PtnRadiusResidual[PtnIdx] :  
 k == 1 && ptree\_ang\_azimuth\_scaling\_enabled ? scaledPhiAng :  
 PtnResidual[PtnIdx][k]  
 where  
 phiOffset0 := PtnPhiMul[PtnIdx] × PtnPhiStep[ nodeIdx ]  
 phiOffset1 := ptree\_ang\_azimuth\_scaling\_enabled ? (  
 PtnResiAng[k] + phiOffset0 ≥ azimuthPi ? –2 × azimuthPi :  
 PtnResiAng[k] + phiOffset0 < -azimuthPi ? 2 × azimuthPi : 0) : 0  
 where  
 azimuthPi := 1 << (ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11)

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided, scaledPhiAng is the scaled value of the quantized primary residual.

scaledPhiAng := DivExp2Tz(PtnPhiResidual[PtnIdx] × invR, invRFracBits - azimuthTwoPiLog2)  
 where  
 azimuthTwoPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 12  
 (invR, invRFracBits) := IntRecip(scaledR)  
 where  
 scaledR := (PtnPred[0] + residAng[0]) > 0 ? (PtnPred[0] + residAng[0]) × 8 : 1

#### Points represented by a node

The reconstructed STV and, if applicable, RPI point coordinates are appended to the output point lists PointPos and PointAng for each point represented by the node.

for (i = 0; i ≤ ptn\_dup\_point\_cnt[PtnIdx]; i++, PointCnt++)  
 for (k = 0; k < 3; k++) {  
 PointPos[PointCnt][k] = PtnPos[k]  
 if (geom\_angular\_enabled)  
 PointAng[PointCnt][k] = PtnAng[k]  
 }

The decoded point position is output for each point represented by the current node.

for (i = 0; i ≤ ptn\_dup\_point\_cnt[curNodeIdx]; i++, PointCnt++)  
 for (k = 0; k < 3; k++) {  
 PointPos[PointCnt][k] = nodePos[k]  
 if (geom\_angular\_enabled)  
 PointAng[PointCnt][k] = nodeAng[k]  
 }

#### Predicted STV coordinates for angular coded geometry

When angular geometry coding is enabled, the node's RPI coordinates are converted to Cartesian STV coordinates, as specified by PtnAngStv[ 𝑘 ], for prediction of the coded position.

PtnAngStv[k] := AngularOrigin[k] + (  
 k == 0 ? DivExp2Fz(ρ × IntCos(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24) :  
 k == 1 ? DivExp2Fz(ρ × IntSin(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24) :  
 k == 2 ? DivExp2Fz(DivExp2Fz(BeamElev[i] × ρ, 15) − BeamOffsetV[i], 3) : na)  
 where  
 ρ := PtnAng[0] << ptree\_ang\_radius\_scale\_log2  
 φ := PtnAng[1]  
 i := PtnAng[2]

#### Prediction from ancestor nodes

Node coordinates can be predicted from up to three ancestor nodes. Depending upon whether angular geometry is enabled or disabled, the prediction is for either RPI or STV coordinates. The coordinates of the parent, grandparent and great-grandparent nodes are specified by ptnP[ 𝑘 ], ptnG[ 𝑘 ] and ptnU[ 𝑘 ], respectively.

ptnP[k] := PtnRef[1][k]  
ptnG[k] := PtnRef[2][k]  
ptnU[k] := PtnRef[3][k]

PtnRef[ancestor][k] := geom\_angular\_enabled ? PointAng[ptIdx][k] : PointPos[ptIdx][k]  
 where  
 ptIdx := PtnStack[PtnDepth − ancestor]

When the adaptive quantization step size of the predictive geometry azimuth angle residuals is not provided, the predicted coordinates, specified by PtnPred[ 𝑘 ], are:

* when the prediction mode is 0 and if angular geometry is:
  + disabled, the origin ( 0, 0, 0 );
  + enabled, a coordinate specified by predAngMode0[ 𝑘 ] whose 𝜌-component is ptn\_radius\_min, and whose 𝜑- and 𝑖-components are the same as the parent node, or zero if the predicted node is the root node of a predictive tree.
* when the prediction mode is 1, the coordinates of the parent node;
* when the prediction mode is 2, the coordinates of the parent node, translated by the vector from the second to first ancestor;
* when the prediction mode is 3, the coordinates of the parent node, translated by the vector from the third to the second ancestor.

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided, the prediction mode is always 1 and the predicted coordinates, specified by PtnPred[ 𝑘 ], are:

* the coordinates specified by predAngMode0[ 𝑘 ] if the predicted node is the root node of a predictive tree;
* derived according to the prediction list for angular coordinates, for the two first predicted coordinates PtnPred[0 ] and PtnPred[1]; and the third coordinate of the parent node for the third predicted coordinate PtnPred[2 ].

PtnPred[k] :=  
 interFlag ? (interPredDir ? PtnSecondInterPredList[interModeIdx] :   
 PtnInterPredList[interModeIdx]) :  
 predMode == 0 && geom\_angular\_enabled ? predAngMode0[k] :  
 predMode == 0 ? 0 :  
 predMode == 1 && PtnIdx == 0 ? predAngMode0[k] :  
 predMode == 1 && usePredList ? PtnPredList[ptnPredIdx][k] + PtnPredAdjust :  
 predMode == 1 ? ptnP[k] :  
 predMode == 2 ? ptnP[k] + ptnP[k] − ptnG[k] :  
 predMode == 3 ? ptnP[k] + ptnG[k] − ptnU[k] : na  
 where  
 interFlag := ptn\_inter\_flag[PtnIdx]  
 interPredDir := ptn\_pred\_direction[PtnIdx]  
 interModeIdx:= ptn\_inter\_pred\_mode[PtnIdx]  
 predMode := ptn\_pred\_mode[PtnIdx]  
 ptnPredIdx := ptn\_pred\_idx[nodeIdx]  
 usePredList := ptree\_ang\_azimuth\_scaling\_enabled && k < 2 && ptnPredIdx > 0  
 PtnPredAdjust :=  
 k == 1 && abs(deltaPhi) >= phiStep ? phiMul × deltaPhi : 0  
 where  
 deltaPhi := ptnP[1] - PtnPredList[ptnPredIdx][1]  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 phiMul := Div(deltaPhi, phiStep, 0)

predAngMode0[k] :=  
 k == 0 ? ptn\_radius\_min :  
 PtnDepth > 0 ? ptnP[k] : 0

## TriSoup

### General

Subclause 9.4 specifies the parsing and the reconstruction of point positions from the TriSoup process. It applies when geom\_tree\_type is 0 and when trisoup\_enabled is 1.

At first, the occupancy tree process described in subclause 9.2 is applied to obtain an occupancy tree at depth occtreeMaxDepthMinus1. The log2 node dimensions at depth occtreeMaxDepthMinus1 is equal to the value trisoup\_node\_size\_log2\_minus2 obtained from the geometry data unit header.

Then, the set of all occupied leaf nodes of the occupancy tree at depth occtreeMaxDepthMinus1 constitutes the set of TriSoup nodes. The slice geometry is represented by TriSoup edge vertices (9.4.3.1) located on the edges of the cuboid volumes associated with the occupied leaf nodes and, optionally, by residual values of centroid vertices determined from TriSoup edge vertices, and, optionally, by face vertices created from two centroid vertices in the cuboid volumes and in the adjacent nodes.

The point positions are reconstructed by generating (9.4.3.2) a set of TriSoup triangles from the TriSoup edge vertices, TriSoup face vertices and centroid vertices, and by voxelization (9.4.3.3) of the TriSoup triangles into points by a ray tracing process.

#### Definition and ordering of TriSoup nodes

TriSoup nodes are the occupied leaf nodes at maximum coded tree level occtreeMaxDepthMinus1, obtained from the decoding process of the occupancy tree as described in subclause 9.2. The TriSoup log2 node dimension is OccLvlNodeSizeLog2[ occtreeMaxDepthMinus1+1 ][ 𝑘 ] along the 𝑘-th component. The occupancy tree is coded such that the three TriSoup log2 node dimensions are all equal to trisoup\_node\_size\_log2\_minus2 + 2. Consequently, the volumes associated with TriSoup nodes are cubes having edge length equal to

TriSoupCubeSize := Exp2( trisoup\_node\_size\_log2\_minus2 + 2 )

The edge length TriSoupNodeSize[ nodeIdx ][ k ] of the nodeIdx-th TriSoup node is set as follows:

TriSoupNodeSize[nodeIdx][k] := TriSoupCubeSize

The TriSoup node ordering is inherited from the occupancy tree node ordering, as described in clause 9.2.2.2 applied to the leaf nodes of the occupancy tree. This order is obtained from a breadth-first traversal of the occupancy tree and, at each level, child nodes of a node are ordered following a Morton order. This results in a traversal in the ascending Morton order of node location.

The number of TriSoup nodes is numberTriSoupNodes. In subclause 9.4 nodeIdx is a node index between 0 and numberTriSoupNodes – 1 that refers to the nodeIdx-th TriSoup node.

The location TriSoupNodeLoc[ nodeIdx ][ k ] of the nodeIdx-th TriSoup node lower corner in the slice coordinate system is obtained as described in subclause 9.2.2.1 by

TriSoupNodeLoc[nodeIdx][k] = nodeLoc[k] × TriSoupCubeSize

where nodeLoc[ 𝑘 ] is obtained recursively on the occupancy tree levels as described in clause 9.2.2.3.

When one or both of trisoup\_non\_cubic\_node\_start\_edge\_present and trisoup\_non\_cubic\_node\_end\_edge\_present are equal to 1, TriSoupNodeLoc[ *nodeIdx* ][ 𝑘 ] and TriSoupNodeSize[ *nodeIdx* ][ 𝑘 ] are modified as follows:

if(trisoup\_slice\_bb\_pos\_bits > 0 && TriSoupNodeLoc[nodeIdx][k] < TriSoupSliceBbPos[k]){  
 TriSoupNodeSize[nodeIdx][k] :=   
 TriSoupNodeLoc[nodeIdx][k] + TriSoupNodeSize[nodeIdx][k] − TriSoupSliceBbPos[k]  
 TriSoupNodeLoc[nodeIdx][k] := TriSoupSliceBbPos[k]  
}

if(trisoup\_slice\_bb\_width\_bits > 0 &&   
 TriSoupSliceBbBoundary[k]) < TriSoupNodeLoc[nodeIdx][k] + TriSoupNodeSize[nodeIdx][k])  
 TriSoupNodeSize[nodeIdx][k] :=   
 TriSoupSliceBbBoundary[k] − TriSoupNodeLoc[nodeIdx][k] + 1

#### Definition and ordering of TriSoup edges

The cuboid volume associated with the nodeIdx-th TriSoup node is depicted on Figure 27. Twelve edges are obtained from the boundaries of the volume. Each edge is defined by its start position edgePosStart and its end position edgePosEnd.

图片包含 应用程序

描述已自动生成

Figure 27 — Volume and edges associated with a TriSoup node

The start and end positions of the twelve edges are obtained from Table 41 as below:

for (edgeTag = 0; edgeTag < 12; edgeTag ++)  
 for (k = 0; k < 3; k++) {  
 edgePosStart[nodeIdx][edgeTag][k] = TriSoupNodeLoc[nodeIdx][k]  
 + edgePosStartShift[edgeTag][k] × TriSoupNodeSize[nodeIdx][k]  
 edgePosEnd[nodeIdx][edgeTag][k] = edgePosStart[nodeIdx][edgeTag][k]  
 + edgePosEndShift[edgeTag][k] × TriSoupNodeSize[nodeIdx][k]  
 }

By generating the twelve edges of the numberTriSoupNodes TriSoup nodes, a collection of 12×numberTriSoupNodes edges is obtained. An edge is uniquely defined by its start and end positions. Consequently, an edge may present multiple times in the collection of 12×numberTriSoupNodes edges because adjacent TriSoup nodes can share one or more common edges. By removing duplicated edges, a set of unique numberTriSoupEdges TriSoup edges is obtained.

Table 41 — Twelve edges' positions relative to the TriSoup node location

| edge number (edgeTag) | edgePosStartShift[edgeTag][k] | | | edgePosEndShift[edgeTag][k] | | |
| --- | --- | --- | --- | --- | --- | --- |
| k =0 | k =1 | k =2 | k =0 | k =1 | k =2 |
| **0** | 0 | 0 | 0 | 1 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 1 | 0 |
| **2** | 0 | 1 | 0 | 1 | 0 | 0 |
| **3** | 1 | 0 | 0 | 0 | 1 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 1 |
| **5** | 0 | 1 | 0 | 0 | 0 | 1 |
| **6** | 0 | 1 | 1 | 0 | 0 | 1 |
| **7** | 1 | 0 | 0 | 0 | 0 | 1 |
| **8** | 0 | 0 | 1 | 1 | 0 | 0 |
| **9** | 0 | 0 | 1 | 0 | 1 | 0 |
| **10** | 0 | 1 | 1 | 1 | 0 | 0 |
| **11** | 1 | 0 | 1 | 0 | 1 | 0 |

In the following subclause 9.4 edgeIdx is an edge index between 0 and numberTriSoupEdges – 1 that refers to the edgeIdx -th TriSoup edge in the set of unique edges. uniqueEdgePosStart[ *edgeIdx* ] and uniqueEdgePosEnd[ *edgeIdx* ] specify the start and end position of *edgeIdx*-th unique edge. The set of unique TriSoup edges is ordered lexicographically, first in the three-component start position and second in the three-component end position of the edges as described in 9.4.1.3.

#### Ordering method of TriSoup edges

When there are two unique edges with index i and j, order of edges is determined as follows.

* Derive startPos\_i and startPos\_j as follows.

startPos\_i = uniqueEdgePosStart[i][0] << 42  
 + uniqueEdgePosStart[i][1] << 21  
 + uniqueEdgePosStart[i][2]  
startPos\_j = uniqueEdgePosStart[j][0] << 42   
 + uniqueEdgePosStart[j][1] << 21   
 + uniqueEdgePosStart[j][1]

* If startPos\_i is less than startPos\_j, i shall be less than j.
* If startPos\_i is greater than startPos\_j, i shall be greater than j.
* Otherwise, endPos\_i and endPos\_j are derived as follows.

endPos\_i = uniqueEdgePosEnd[i][0] << 42  
 + uniqueEdgePosEnd[i][1] << 21  
 + uniqueEdgePosEnd[i][2]  
endPos\_j = uniqueEdgePosEnd[j][0] << 42   
 + uniqueEdgePosEnd[j][1] << 21   
 + uniqueEdgePosEnd[j][1]

* + If endPos\_i is less than endPos\_j, i shall be less than j.
  + Otherwise, i shall be greater than j.

### Syntax element semantics

#### Semantics of TriSoup syntax elements present in the geometry data unit header

trisoup\_node\_size\_log2\_minus2 plus 2 specifies the size of a TriSoup node.

trisoup\_sampling\_value\_minus1 plus 1 specifies the sampling distance of each rayOrigin used by the ray tracing process to generate decoded points.

trisoup\_num\_unique\_segments\_bits\_minus1 plus 1 specifies the length in bits of trisoup\_num\_unique\_segments\_minus1 syntax element.

trisoup\_num\_unique\_segments\_minus1 plus 1 specifies the number of TriSoup edges.

trisoup\_vertex\_number\_bits specifies the number of bits for a decoded TriSoup vertex. The values VertexPrecisionLog2 and bitDropped specify the precision and the quantization scale of a decoded TriSoup vertex position.

VertexPrecisionLog2 := trisoup\_vertex\_number\_bits > 0 ? trisoup\_vertex\_number\_bits : trisoup\_node\_size\_log2\_minus2 + 2

bitDropped := max(0,  trisoup\_node\_size\_log2\_minus2 + 2  - VertexPrecisionLog2)

trisoup\_centroid\_vertex\_residual\_present specifies whether (when 1) or not (when 0) the residuals of centroid positions are present in the TriSoup syntax structure.

trisoup\_face\_vertex\_enabled equal to 1 specifies that the eligibility of a face vertex is judged for each TriSoup node as specified in subclause 9.4.2.4 and that the face vertex may be present on the face of TriSoup nodes. When trisoup\_face\_vertex\_enabled equal to 0 specifies that the eligibility of face vertex is not judged for any TriSoup node and that the face vertex is not present on the face of any TriSoup node. When trisoup\_face\_vertex\_enabled is not present, it shall be inferred to be 0.

trisoup\_halo\_enabled specifies whether (when 1) or not (when 0) halo is applied in the ray tracing process.

trisoup\_adaptive\_halo\_enabled specifies whether (when 1) or not (when 0) adaptive halo is applied in the ray tracing process. When trisoup\_adaptive\_halo\_enabled is not present, it shall be inferred to be 0.

trisoup\_vertex\_merge\_enabled specifies whether (when 1) or not (when 0) the vertex merge process that replaces multiple vertices near the corner of TriSoup nodes by a single vertex is applied.

trisoup\_slice\_bb\_pos\_bits specifies the length in bits of each trisoup\_slice\_bb\_pos\_xyz syntax element. When trisoup\_slice\_bb\_pos\_bits is not present, it shall be inferred to be 0.

trisoup\_slice\_bb\_pos\_xyz[ k] and trisoup\_slice\_bb\_pos\_log2\_scaletogether specify the lower corner k-th XYZ coordinates of the slice bounding box in the slice coordinate system. The lower corner of the slice bounding box in STV coordinates is specified by the expression TriSoupSliceBbPos[ k]. When trisoup\_slice\_bb\_pos\_bits is 0, trisoup\_slice\_bb\_pos\_xyz[k ] and trisoup\_slice\_bb\_pos\_log2\_scale shall be inferred to be 0.

TriSoupSliceBbPos[k] :=   
trisoup\_slice\_bb\_pos\_xyz[StvToXyz[k]] << trisoup\_slice\_bb\_pos\_log2\_scale

trisoup\_slice\_bb\_width\_bitsspecifies the length in bits of each trisoup\_slice\_bb\_width\_xyz syntax element. When trisoup\_slice\_bb\_width\_bits is not present, it shall be inferred to be 0.

trisoup\_slice\_bb\_width\_xyz[ k ] and trisoup\_slice\_bb\_width\_log2\_scale together specify the k-th XYZ component of the slice bounding box dimensions in the slice coordinate system. When trisoup\_slice\_bb\_width\_bits is not 0, the upper boundary of the slice bounding box in STV coordinates is specified by the expression TriSoupSliceBbBoundary[k ].

TriSoupSliceBbBoundary[k] := TriSoupSliceBbPos[k] +   
trisoup\_slice\_bb\_width\_xyz[StvToXyz[k]] << trisoup\_slice\_bb\_width\_log2\_scale

#### Semantics of TriSoup syntax elements associated with TriSoup edges

vertex\_present[edgeIdx] specifies whether (when 1) or not (when 0) a TriSoup vertex exists on edgeIdx-th TriSoup edge.

has\_vertex[nodeIdx][edgeTag] specifies whether (when 1) or not (when 0) a TriSoup vertex exists on edgeTag-th TriSoup edge of nodeIdx-th TriSoup node. has\_vertex[nodeIdx][edgeTag] is set by corresponding vertex\_present[edgeIdx] as defined in 9.4.1.1.

vertex\_position[edgeIdx][bit] specifies a bit-th value of quantized one-dimensional vertex position according to the direction of each TriSoup edge with a vertex quantizedVertexPos[edgeIdx]. When vertex\_present[edgeIdx] is equal to 0, quantizedVertexPos[edgeIdx] is set to -1. quantizedVertexPos[edgeIdx] is derived as follows.

quantizedVertexPos[edgeIdx] = 0  
if (vertex\_present[edgeIdx])  
 for (bit = 0; bit < trisoup\_vertex\_number\_bits; bit++)  
 quantizedVertexPos[edgeIdx] += vertex\_position[edgeIdx][bit] << bit  
else  
 quantizedVertexPos[edgeIdx] = -1

edgeVertex[nodeIdx][edgeTag] specifies a quantized one-dimensional vertex position according to the direction of edgeTag-th TriSoup edge of nodeIdx-th TriSoup node. edgeVertex[nodeIdx][edgeTag] is set by corresponding quantizedVertexPos[edgeIdx] as defined in 9.4.1.1.

When trisoup\_vertex\_merge\_enabled is equal to 1, the value vertexNumOfCorner[ns][nt][nv] specifies the number of vertices around a node corner at ( ns, nt, nv ) and the threshold value thVertexMerge derived as follows.

thVertexMerge := min(2048, TriSoupCubeSize << 6)  
for (edgeIdx = 0; edgeIdx < numberTriSoupEdges; edgeIdx++){  
 if (vertex\_present[edgeIdx]) {  
 ns = uniqueEdgePosStart[edgeIdx][0]  
 nt = uniqueEdgePosStart[edgeIdx][1]  
 nv = uniqueEdgePosStart[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv] = 0  
   
 ns = uniqueEdgePosEnd[edgeIdx][0]  
 nt = uniqueEdgePosEnd[edgeIdx][1]  
 nv = uniqueEdgePosEnd[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv] = 0  
 }  
}  
for (edgeIdx = 0; edgeIdx < numberTriSoupEdges; edgeIdx++){  
 if (vertex\_present[edgeIdx]) {  
 relativePos = (quantizedVertexPos[edgeIdx] << (8 + bitDropped)) + (128 << bitDropped)  
 if (relativePos < thVertexMerge) {  
 ns = uniqueEdgePosStart[edgeIdx][0]  
 nt = uniqueEdgePosStart[edgeIdx][1]  
 nv = uniqueEdgePosStart[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv]++  
 }  
 if ((TriSoupCubeSize << 8) - relativePos < thVertexMerge) {  
 ns = uniqueEdgePosEnd[edgeIdx][0]  
 nt = uniqueEdgePosEnd[edgeIdx][1]  
 nv = uniqueEdgePosEnd[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv]++  
 }  
 }  
}

#### Semantics of TriSoup syntax elements associated with TriSoup nodes

centroid\_residual\_is\_zero[nodeIdx] specifies whether (when 1) or not (when 0) a centroid residual of nodeIdx-th node is 0.

centroid\_residual\_is\_zero[nodeIdx], centroid\_residual\_magnitude[nodeIdx][bit], and centroid\_residual\_sign[nodeIdx] together specify a quantized residual of each centroid. When centroid\_residual\_is\_zero[nodeIdx] is not present, it shall be inferred to be 1. When centroid\_residual\_magnitude[nodeIdx] and centroid\_residual\_sign[nodeIdx] are not present, they both shall be inferred to be 0.

The value driftQ[nodeIdx] specifies a quantized centroid residual.

driftQ[nodeIdx] := (2 × centroid\_residual\_sign[nodeIdx] - 1) × (¬centroid\_residual\_is\_zero[nodeIdx] + magBits[nodeIdx])

has\_face\_vertex[nodeIdx][fvIdx] equal to 1 specifies that a face vertex is added on the nodeIdx*-*th TriSoup node’s face which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin. has\_face\_vertex[nodeIdx][fvIdx] equal to 0 specifies that a face vertex is not added on the nodeIdx*-*th TriSoup node’s face which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin. When has\_face\_vertex[nodeIdx][fvIdx] is not present, it shall be inferred to 0.

#### Presence of has\_face\_vertex

When trisoup\_face\_vertex\_enabled is equal to 1, the presence of has\_face\_vertex[nodeIdx][fvIdx] is specified by the expression FaceEligible[nodeIdx][fvIdx]. The syntax element has\_face\_vertex[nodeIdx][fvIdx] shall be present in the TriSoup syntax when FaceEligible[nodeIdx][fvIdx] is equal to 1.

Centroid[nodeIdx] is the result of the process specified in 9.4.3.2.5. FaceVertex[nodeIdx][fvIdx] is the result of the process specified in 9.4.3.2.6. Let adjNode[nodeIdx][fvIdx] be the node index of the adjacent TriSoup node in the direction of fvIdx-th component from the nodeIdx-th TriSoup node.

When all the conditions listed below are true, FaceEligibleCond1 is set to 1.

* centroid\_residual\_is\_zero[nodeIdx] is equal to 0, Centroid[nodeIdx][k] is greater than or equal to 0 and Centroid[nodeIdx][k] is less than or equal to TriSoupNodeSize[nodeIdx][k] for each k (k = 0, 1, 2), and
* adjNode[nodeIdx][fvIdx]-th TriSoup node is available, and
* centroid\_residual\_is\_zero[adjNode[nodeIdx][fvIdx]] is equal to 0, Centroid[adjNode[nodeIdx][fvIdx]][k] is greater than or equal to 0 and Centroid[adjNode[nodeIdx][fvIdx]][k] is less than or equal to TriSoupNodeSize[adjNode[nodeIdx][fvIdx]][k] for each k (k = 0, 1, 2), and

When all the conditions listed below are true, FaceEligible[nodeIdx][fvIdx] is set to 1.

* FaceEligibleCond1 is 1, and
* ((fvIdx < 2) ? (has\_vertex[nodeIdx][3 − fvIdx] + has\_vertex[nodeIdx][6 − fvIdx] + has\_vertex[nodeIdx][7 − fvIdx] + has\_vertex[nodeIdx][11 − fvIdx]) : (has\_vertex[nodeIdx][8] + has\_vertex[nodeIdx][9] + has\_vertex[nodeIdx][10] + has\_vertex[nodeIdx][11])) is equal to either 2 or 3, and
* Both dp0 and dp1, which are derived as follows, are greater than 0.

evCnt = numTriSoupVertices[nodeIdx]  
distMin = 0x7fffffff  
for(evIdx = 0; evIdx < ((evCnt == 3) ? 1 : evCnt); evIdx++){  
 ev0 = evIdx  
 ev1 = evIdx + 1  
 if( ev1 >= evCnt )  
 ev1 −= evCnt  
 evCoord0 = sortedVertices[nodeIdx][ev0] + 128  
 evCoord1 = sortedVertices[nodeIdx][ev1] + 128  
 if( ( TriSoupNodeSize[nodeIdx][fvIdx] ≠ evCoord0[fvIdx] ) ||   
 ( TriSoupNodeSize[nodeIdx][fvIdx] ≠ evCoord1[fvIdx] ) )  
 continue;  
 middlePoint = ( evCoord0 + evCoord1 ) / 2  
 distVec = (middlePoint – FaceVertex[nodeIdx][fvIdx]) >> 8  
 dist = distVec[0] × distVec[0] + distVec[1] × distVec[1] +   
 distVec[2] × distVec[2]  
 if( distMin > dist ) {  
 evIdxMin[nodeIdx][fvIdx][0] = ev0  
 evIdxMin[nodeIdx][fvIdx][1] = ev1  
 distMin = dist  
 }  
}  
eeVec = sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][1]] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]]  
eeVecNorm = IntSqrt(eeVec[0] × eeVec[0] + eeVec[1] × eeVec[1] +   
 eeVec[2] × eeVec[2])  
eUnitVec = eeVecNorm ? ( ( eeVec << 8 ) / eeVecNorm ) : 0  
enVec = (faceVertex[nodeIdx][fvIdx] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]]) ×   
 eUnitVec >> 8  
nfVec = faceVertex[nodeIdx][fvIdx] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]] −   
 ((enVec × eUnitVec) >> 8)  
dp0 = DriftVec[nodeIdx] × nfVec  
dp1 = DriftVec[adjNode[nodeIdx][fvIdx]] × nfVec

Otherwise, when all the conditions listed below are true, FaceEligible[nodeIdx][fvIdx] is set to 1.

* FaceEligibleCond1 is 1, and
* numTriSoupVertices[nodeIdx] is equal to 2 and numTriSoupVertices[adjNode[nodeIdx][fvIdx]] is equal to 2.

Otherwise, FaceEligible[nodeIdx][fvIdx] is set to 0.

### TriSoup decoding process

TriSoup decoding process consists on the following three processes.

* Decoding of TriSoup edge vertices located on TriSoup edges (9.4.3.1)
* Derivation of TriSoup vertices in TriSoup nodes (9.4.3.2)
* Determination of decoded points by the voxelization of TriSoup triangles (9.4.3.3)

#### Decoding of TriSoup edge vertices located on TriSoup edges

Vertex flag (segment indicator) and vertex position of each TriSoup edge are decoded as follows.

* determination of neighbouring information (9.4.3.1.1)
* determination of contexts and decoding of vertex flag and vertex position using dynamic OBUF (9.4.3.1.2)
* initialization and decoding TriSoup vertices bits using OBUF (9.4.3.1.3)

##### Determination of Neighbouring Information

Mask information with 16 bits length neighbourMask[edgeIdx] is set for all unique TriSoup Edges (edgeIdx = 0, 1,…, numberTriSoupEdges) as follows.

* Assume edgeStartPos of edgeIdx-th unique TriSoup edge is (k0, k1, k2).
* A variable TCS is set to TriSoupCubeSize.

TCS := TriSoupCubeSize

* edgeDirection of edgeIdx-th unique TriSoup Edge is derived as follows,
  + If edgeEndPos of i-th unique TriSoup Edge is (k0+TCS, k1, k2), edgeDirection is set to 0.
  + Otherwise, if edgeEndPos of edgeIdx-th unique TriSoup Edge is (k0, k1+TCS, k2), edgeDirection is set to 1.
  + Otherwise, edgeDirection is set to 2.
* Construct neighbourMask[edgeIdx] as defined in 9.4.3.1.4.

Then edgePattern[edgeIdx] is derived for all unique TriSoup edges (edgeIdx = 0, 1,…, numberTriSoupEdges). edgePattern[edgeIdx] consists of unique edge indexes of nine neighbouring edges a through edgeIdx in the Table 42. When the corresponding edge of variable j (j  = 0, 1,…, 8) exists, edgePattern[edgeIdx][j] is set to the unique edge index of the corresponding edge. Otherwise, when the corresponding edge of variable j does not exist, edgePattern[edgeIdx][j] is set to -1.

Table 42 — edgeStartPos and edgeEndPos of neighbour edges

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***j*** | **edge** | **edgeStartPos, edgeEndPos** | | |
| **edgeDirection = 0** | **edgeDirection = 1** | **edgeDirection = 2** |
| - | E | (k0, k1, k2), (k0 + TCS, k1, k2) | (k0, k1, k2),  (k0, k1 + TCS, k2) | (k0, k1, k2),  (k0, k1, k2 + TCS) |
| 0 | a | (k0 - TCS, k1, k2),  (k0, k1, k2) | (k0, k1 - TCS, k2),  (k0, k1, k2) | (k0, k1, k2 - TCS),  (k0, k1, k2) |
| 1 | b | (k0, k1, k2 - TCS),  (k0, k1, k2) | (k0, k1, k2 - TCS),  (k0, k1, k2) | (k0, k1 - TCS, k2),  (k0, k1, k2) |
| 2 | c | (k0, k1 - TCS, k2),  (k0, k1, k2) | (k0 - TCS, k1, k2),  (k0, k1, k2) | (k0 - TCS, k1, k2),  (k0, k1, k2) |
| 3 | d | (k0, k1, k2),  (k0, k1, k2 + TCS) | (k0, k1, k2),  (k0, k1, k2 + TCS) | (k0 - TCS, k1, k2 + TCS),  (k0, k1, k2 + TCS) |
| 4 | e | (k0, k1, k2),  (k0, k1 + TCS, k2) | (k0 - TCS, k1 + TCS, k2),  (k0, k1 + TCS, k2) | (k0, k1 - TCS, k2 + TCS),  (k0, k1, k2 + TCS) |
| 5 | f | (k0, k1 - TCS, k2),  (k0 + TCS, k1 - TCS, k2) | (k0 - TCS, k1, k2),  (k0 - TCS, k1 + TCS, k2), | (k0 - TCS, k1, k2),  (k0 - TCS, k1, k2 + TCS) |
| 6 | g | (k0, k1, k2 - TCS),  (k0 + TCS, k1, k2 - TCS) | (k0, k1, k2 - TCS),  (k0, k1 + TCS, k2 - TCS) | (k0, k1 - TCS, k2),  (k0, k1 - TCS, k2 + TCS) |
| 7 | h | (k0, k1 - TCS, k2),  (k0, k1 - TCS, k2 + TCS) | (k0, k1, k2 - TCS),  (k0 + TCS, k1, k2 - TCS), | (k0, k1 - TCS, k2),  (k0 + TCS, k1 - TCS, k2), |
| 8 | i | (k0, k1, k2 - TCS),  (k0, k1 + TCS, k2 - TCS) | (k0 - TCS, k1 + TCS, k2 + TCS),  (k0, k1 + TCS, k2 + TCS) | (k0 - TCS, k1 + TCS, k2 + TCS),  (k0, k1 + TCS, k2 + TCS) |

##### Determination of Contexts and Decoding of Vertex Flag and Vertex Position

The following process is applied to all unique edges.

* Derive common information for vertex flag and position as defined in 9.4.3.1.5.
* Derive context ctxMap1 and ctxMap2 for vertex flag.

ctxMap1 = min(nclosestPattern, 2) × 15 × 2 + (neighbEdge-1) × 2 + ((ctx1 == 4))  
ctxMap2 = neighbEnd << 11  
ctxMap2 |= (patternClose & (0b00000110)) << 9 - 1  
ctxMap2 |= direction << 7  
ctxMap2 |= (patternClose & (0b00011000))<< 5 - 3   
ctxMap2 |= (patternClose & (0b00000001))<< 4   
orderedPclosePar =   
 (((pattern >> 5) & 3) << 2) + (!!(pattern & 128) << 1) + !!(pattern & 256)  
ctxMap2 |= orderedPclosePar

* Decode vertex flag vertex\_present[edgeIdx] as shown in 9.4.3.1.3.
* If vertex\_present[edgeIdx] is equal to 1, the following processes are applied to decode vertex\_position[edgeIdx][bit].
* Derive context ctxMap1 and ctxMap2 for the first bit of position.

ctxFullNbounds =  
(4 × (ctx0 <= 1 ? 0 : (ctx0 >= 3 ? 2 : 1)) + (max(1, ctx1) - 1)) × 2 + (ctxE == 3)  
ctxMap1 = ctxFullNbounds × 2 + (nclosestStart > 0)  
ctxMap2 = missedCloseStart << 8  
ctxMap2 |= (patternClosest & 1) << 7  
ctxMap2 |= direction << 5  
ctxMap2 |= patternClose & (0b00011111)

* Decode the first bit of position vertex\_position[edgeIdx][0] as shown in 9.4.3.1.3 and decoded bit is set as v.
* When trisoup\_vertex\_number\_bits is greater than 1, the following processes are applied.
  + derive context for the second bit of position

ctxMap1 = ctxFullNbounds × 2 + (nclosestStart > 0)  
ctxMap1 = (ctxMap1 << 1) + v  
ctxMap2 = missedCloseStart << 8  
ctxMap2 |= (patternClose & 1) << 7  
ctxMap2 |= (patternClosest & 1) << 6  
ctxMap2 |= direction << 4  
ctxMap2 |= (patternClose & (0b00011111)) >> 1  
orderedPclosePar = (((patternClose >> 5) & 3) << 2) + (!!(patternClose & 128) << 1) + !!(patternClose & 256)  
ctxMap2 = (ctxMap2 << 4) + orderedPclosePar

* + Decode the second bit of position vertex\_position[edgeIdx][1] as shown in 9.4.3.1.3 and variable v is updated.

v = (v << 1) + vertex\_position[edgeIdx][1]

* + When trisoup\_vertex\_number\_bits is greater than 2, the following processes are applied.
    - derive context for the third bit of position

ctxBits3 = (6 × (ctx0 >> 1) + missedCloseStart) × 2 + (ctxE == 3)  
ctxBits3 = 4 × ctxBits3 + v

* + - Decode the third bit of position vertex\_position[edgeIdx][2] as shown in 9.4.3.1.3 and variable v is updated.

v = (v << 1) + vertex\_position[edgeIdx][2]

* + - When trisoup\_vertex\_number\_bits is greater than 3, decode remaining bits of vertex\_position[edgeIdx][bit] (bit=3,…,trisoup\_vertex\_number\_bits-1) by bypass decoding and variable v is updated.

##### Initialization and Decoding TriSoup Vertices Bits Using OBUF

The following processes are applied to initialize OBUF instances.

* To decode vertex position bits of TriSoup nodes, a first buffer (TriSoup buffer) is created for all OBUF instances to be later used by TriSoup coding according to 12.3, and the OBUF buffer size obufBufferSize and the buffer depth obufLeafDepth of fully deployed trees are set as follows.

obufBufferSize := 20000  
obufLeafDepth := 4

* The context arrays OBUF ACPMs are created according to 12.2.2.
* All OBUF instances used by TriSoup coding are created and initialized according to 12.2. The size nBit1 and nBit2 are set as according to Table 43.
  + When i is equal to 0, the OBUF instance TOI[ i ] is used for coding vertex presence flag for an edge.
  + When i is equal to 1, the OBUF instance TOI[ i ] is used for coding the first bit of vertex position.
  + When i is equal to 2, the OBUF instance TOI[ i ] is used for coding the second bit of vertex position.
* The array ctxIdxMap[ j][i ] is initialized according to Table 44, Table 45 and Table 46.

The following processes are applied to decode TrisSoup vertices bits.

* Each bit of vertex position of coded TriSoup nodes is decoded according to 12.4 by using two contextual information ctxMap1 and ctxMap2, which are set as the first contextual informationinfo1and the second contextual informationinfo2,and calling the OBUF instances of TriSoup coding to get the decoded bin.
* During updating of dynamic OBUF trees, the primary part remains unchanged (is never reduced) and the dynamic of secondary part may be reduced by applying a OBUF tree updating process according to the 12.4.1.2.

Table 43 — the size values (nBit1,nBit2) of TriSoup OBUF instance TOI[ i ]

|  |  |  |  |
| --- | --- | --- | --- |
|  | *i* | | |
| **0** | **1** | **2** |
| **(**nBit1**,**nBit2**)** | (7,15) | (6,11) | (7,15) |

Table 44 — Initial values of CtxIdxMap[ j][0] for coding vertex presence flag of an edge

| j | CtxIdxMap[ j][0] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 8 .. 15 | 15 | 15 | 42 | 96 | 71 | 37 | 15 | 15 |
| 16 .. 23 | 22 | 51 | 15 | 15 | 30 | 27 | 15 | 15 |
| 24 .. 31 | 64 | 15 | 48 | 15 | 224 | 171 | 127 | 24 |
| 32 .. 39 | 127 | 34 | 80 | 46 | 141 | 44 | 66 | 49 |
| 40 .. 47 | 127 | 116 | 140 | 116 | 105 | 39 | 127 | 116 |
| 48 ..55 | 114 | 46 | 172 | 109 | 60 | 73 | 181 | 161 |
| 56 ..63 | 112 | 65 | 240 | 159 | 127 | 127 | 127 | 87 |
| 64 .. 71 | 183 | 127 | 116 | 116 | 195 | 88 | 152 | 141 |
| 72 .. 79 | 228 | 141 | 127 | 80 | 127 | 127 | 160 | 92 |
| 80 .. 87 | 224 | 167 | 129 | 135 | 240 | 183 | 240 | 184 |
| 88 .. 95 | 240 | 240 | 127 | 127 | 127 | 127 | 127 | 127 |
| 96 ..103 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 104 ..111 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 112 ..119 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 120 ..127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

Table 45 — Initial values of CtxIdxMap[ j][1] for coding the first bit of vertex position

| j | CtxIdxMap[ j][1] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 116 | 127 | 118 | 15 | 104 | 56 | 97 | 15 |
| 8 .. 15 | 96 | 15 | 29 | 15 | 95 | 15 | 46 | 15 |
| 16 .. 23 | 196 | 116 | 182 | 53 | 210 | 104 | 163 | 69 |
| 24 .. 31 | 169 | 15 | 114 | 15 | 121 | 15 | 167 | 63 |
| 32 .. 39 | 240 | 127 | 184 | 92 | 240 | 163 | 197 | 77 |
| 40 .. 47 | 239 | 73 | 179 | 59 | 213 | 48 | 185 | 108 |
| 48 ..55 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 56 ..63 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

Table 46 — Initial values of CtxIdxMap[ j][2] for coding the second bit of vertex position

| j | CtxIdxMap[ j][2] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 141 | 127 | 127 | 127 | 189 | 81 | 36 | 127 |
| 8 .. 15 | 143 | 105 | 103 | 116 | 201 | 60 | 38 | 116 |
| 16 .. 23 | 116 | 127 | 15 | 127 | 153 | 59 | 15 | 116 |
| 24 .. 31 | 69 | 105 | 15 | 127 | 158 | 93 | 36 | 79 |
| 32 .. 39 | 141 | 161 | 116 | 127 | 197 | 102 | 53 | 127 |
| 40 .. 47 | 177 | 125 | 88 | 79 | 209 | 75 | 102 | 28 |
| 48 ..55 | 95 | 74 | 72 | 56 | 189 | 62 | 78 | 18 |
| 56 ..63 | 88 | 116 | 28 | 45 | 237 | 100 | 152 | 35 |
| 64 .. 71 | 141 | 240 | 127 | 127 | 208 | 133 | 101 | 141 |
| 72 .. 79 | 186 | 210 | 168 | 98 | 201 | 124 | 138 | 15 |
| 80 .. 87 | 195 | 194 | 103 | 94 | 229 | 82 | 167 | 23 |
| 88 .. 95 | 92 | 197 | 112 | 59 | 185 | 87 | 156 | 79 |
| 96 ..103 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 104 ..111 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 112 ..119 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 120 ..127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

##### Construction of neighbour mask

Construct neighbourMask[edgeIdx] for each TriSoup edge corresponding to edgeIdx as follows based on Table 47.

* Initialize neighbourMask[edgeIdx] as 0.
* When node e1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= 1

* When node e2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 1)

* When node e3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 2)

* When node e4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 3)

* When node a1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 4)

* When node a2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 5)

* When node a3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 6)

* When node a4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 7)

* When node a1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 8)

* When node a2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 9)

* When node a3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 10)

* When node a4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 11)

* When edgeDirection is equal to 1, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 13)

* When edgeDirection is equal to 2, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 14)

Table 47 — Locations of neighbour nodes

|  |  |  |  |
| --- | --- | --- | --- |
| **Node** | **TriSoupNodeLoc** | | |
| **edgeDirection = 0** | **edgeDirection = 1** | **edgeDirection = 2** |
| e1 | (k0, k1, k2) | (k0, k1, k2) | (k0, k1, k2) |
| e2 | (k0, k1 - TCS, k2) | (k0 - TCS, k1, k2) | (k0, k1 - TCS, k2) |
| e3 | (k0, k1, k2 - TCS) | (k0, k1, k2 - TCS) | (k0 - TCS, k1 - TCS, k2) |
| e4 | (k0, k1 - TCS, k2 - TCS) | (k0 - TCS, k1, k2 - TCS) | (k0 - TCS, k1, k2) |
| a1 | (k0 - TCS, k1, k2) | (k0, k1 - TCS, k2) | (k0, k1, k2 - TCS) |
| a2 | (k0 - TCS, k1 - TCS, k2) | (k0 - TCS, k1 - TCS, k2) | (k0, k1 - TCS, k2 - TCS) |
| a3 | (k0 - TCS, k1, k2 - TCS) | (k0, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 - TCS) |
| a4 | (k0 - TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1, k2 - TCS) |
| b1 | (k0 + TCS, k1, k2) | (k0, k1 + TCS, k2) | (k0, k1, k2 + TCS) |
| b2 | (k0 + TCS, k1 - TCS, k2) | (k0 - TCS, k1 + TCS, k2) | (k0, k1 - TCS, k2 + TCS) |
| b3 | (k0 + TCS, k1, k2 - TCS) | (k0, k1 + TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 + TCS) |
| b4 | (k0 + TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 + TCS, k2 - TCS) | (k0 - TCS, k1, k2 + TCS) |

##### Derive common information for vertex flag and position

Derive common information to decode vertex flag and position as follows.

* Derive ctxE, ctx0, ctx0, and direction as follows.

ctxE =   
(!!(neighbourMask[edgeIdx] & 1)) + (!!(neighbourMask[edgeIdx] & 2))  
+ (!!(neighbourMask[edgeIdx] & 4)) + (!!(neighbourMask[edgeIdx] & 8)) - 1

ctx0 =   
(!!(neighbourMask[edgeIdx] & 16)) + (!!(neighbourMask[edgeIdx] & 32))  
+ (!!(neighbourMask[edgeIdx] & 64)) + (!!(neighbourMask[edgeIdx] & 128))

ctx1 = (!!(neighbourMask[edgeIdx] & 256)) + (!!(neighbourMask[edgeIdx] & 512))  
+ (!!(neighbourMask[edgeIdx] & 1024)) + (!!(neighbourMask[edgeIdx] & 2048))

direction = neighbourMask[edgeIdx] >> 13

* Derive pattern, patternClose, patternClosest, and nclosestPattern.

for(v = 0; v < 9; v++){  
 if (segind[edgePattern[v]]){  
 pattern |= 1 << v  
 vertexPos2bits =   
 vertices[correspondanceSegment2V[idxEdge]] >> max(0, nbitsVertices - 2)  
 if (towardOrAway[v18])  
 vertexPos2bits = max2bits - vertexPos2bits // reverses for away  
 if (vertexPos2bits >= mid2bits)  
 patternClose |= 1 << v  
 if (vertexPos2bits >= max2bits)  
 patternClosest |= 1 << v  
 nclosestPattern += vertexPos2bits >= max2bits && v <= 4  
 }  
}

* Derive missedCloseStart and nclosestStart.

missedCloseStart = !(pattern & 2) + !(pattern & 4)  
nclosestStart =  
 !!(patternClosest & 1) + !!(patternClosest & 2) + !!(patternClosest & 4)

* Derive neighbEdge, neighbEnd, and neighbStart.

neighbEdge = (neighbourMask[edgeIdx] >> 0) & 15  
neighbEnd = (neighbourMask[edgeIdx] >> 4) & 15  
neighbStart = (neighbourMask[edgeIdx] >> 8) & 15  
if (direction == 2) {  
 neighbEdge = ((neighbourMask[edgeIdx] >> 0 + 0) & 1)  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 3) & 1) << 1  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 1) & 1) << 2  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 2) & 1) << 3  
  
 neighbEnd = ((neighbourMask[edgeIdx] >> 4 + 0) & 1)  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 3) & 1) << 1  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 1) & 1) << 2  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 2) & 1) << 3  
  
 neighbStart = ((neighbourMask[edgeIdx] >> 8 + 0) & 1)  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 3) & 1) << 1  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 1) & 1) << 2  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 2) & 1) << 3  
}

#### Derive TriSoup vertices corresponding each TriSoup node

TriSoup vertices for each TriSoup node shall be derived as following steps:

* Derive information related to vertex merge (9.4.3.2.1)
* Derive TriSoup vertices corresponding each TriSoup edge (9.4.3.2.2)
* Derive initial position of centroid vertices (9.4.3.2.3),
* Determine the dominant axis and sort vertices (9.4.3.2.4),
* Derive refined centroid position (9.4.3.2.5),
* Derive TriSoup face vertices corresponding each TriSoup node (9.4.3.2.6),
* Insert TriSoup face vertices to corresponding sorted TriSoup vertices (9.4.3.2.7)

##### Derive information related to vertex merge

When trisoup\_vertex\_merge\_enabled is equal to 1, expressions activeNode[nodeIdx] and activeCorner[nodeIdx][n] for each TriSoup node corresponding to nodeIdx are derived as follows.

The expression activeNode[nodeIdx] indicates whether the TriSoup node corresponding to nodeIdx need to be processed.

numVertex = 0  
for (edgeTag = 0; edgeTag < 12; edgeTag++) {  
 if (edgeVertex[nodeIdx][edgeTag] >= 0) {  
 numVertex++  
 }  
}  
if (numVertex >= 6) {  
 activeNode[nodeIdx] = 1  
} else {  
 activeNode[nodeIdx] = 0  
}

The expression activeCorner[nodeIdx][n] indicates whether the corner corresponding to n (n=0,…,7) of the TriSoup node corresponding to nodeIdx need to be dealt with.

for (n = 0; n < 8; n++) {  
 ns = TriSoupNodeLoc[nodeIdx][0] + TriSoupNodeSize[nodeIdx][0] × Bit(n, 0)  
 nt = TriSoupNodeLoc[nodeIdx][1] + TriSoupNodeSize[nodeIdx][1] × Bit(n, 1)  
 nv = TriSoupNodeLoc[nodeIdx][2] + TriSoupNodeSize[nodeIdx][2] × Bit(n, 2)  
 if (vertexNumOfCorner[ns][nt][nv] >= 4) {  
 activeCorner[nodeIdx][n] = 1  
 } else {  
 activeCorner[nodeIdx][n] = 0  
 }  
}

##### Derive TriSoup vertices corresponding each TriSoup edge

Vertices vertex[nodeIdx][vIdx][k] for each TriSoup node corresponding to nodeIdx are derived as follows.

The variable vIdx is initialized by 0.

For all edges (edgeTag=0,…,11) on a TriSoup node corresponding to nodeIdx, the edge direction and the expression edgeVertexModify[nodeIdx][edgeTag], which indicates whether the edge indexed by edgeTag on the TriSoup node corresponding to nodeIdx need to be modified are derived as follows.

* When edgeVertex[nodeIdx][edgeTag] is greater than or equal to 0, the following processes are applied.
  + Derive direction[k] (k=0, 1, 2):

for (k = 0; k < 3; k++) {  
 direction[k] = edgePosEnd[nodeIdx][edgeTag][k] - edgePosStart[nodeIdx][edgeTag][k]  
}

* + The expression edgeVertexModify[nodeIdx][edgeTag] is initialized by 0.
  + When trisoup\_vertex\_merge\_enabled is equal to 1 and activeNode[nodeIdx] is equal to 1, the expression edgeVertexModify[nodeIdx][edgeTag] is modified as follows.

relativePos = (edgeVertex[nodeIdx][edgeTag] << (8 + bitDropped))  
 + (128 << bitDropped)  
n = edgePosStartShift[edgeTag][0]  
 + (edgePosStartShift[edgeTag][1] << 1)  
 + (edgePosStartShift[edgeTag][2] << 2)  
if (relativePos < thVertexMerge && activeCorner[nodeIdx][n]) {  
 edgeVertexModify[nodeIdx][edgeTag] = 1  
}  
  
n = edgePosEndShift[edgeTag][0]  
 + (edgePosEndShift[edgeTag][1] << 1)  
 + (edgePosEndShift[edgeTag][2] << 2)  
if ((TriSoupCubeSize << 8) - relativePos < thVertexMerge  
 && activeCorner[nodeIdx][n]) {  
 edgeVertexModify[nodeIdx][edgeTag] = 1  
}

* + When edgeVertexModify[nodeIdx][edgeTag] is equal to 0, dequantized vertex positions vertex[nodeIdx][vIdx][k] is derived.

for (k = 0; k < 3; k++) {  
 vertex[nodeIdx][vIdx][k] =  
 (edgePosStart[nodeIdx][edgeTag][k] - TriSoupNodeLoc[nodeIdx][k]) << 8  
 vertex[nodeIdx][vIdx][k] -= 128  
 if (direction[k] > 0){  
 vertex[nodeIdx][vIdx][k] +=  
 (edgeVertex[nodeIdx][edgeTag] << (bitDropped + 8)) + (128 << bitDropped)  
 }  
}  
vIdx++

When trisoup\_vertex\_merge\_enabled is equal to 1 and activeNode[nodeIdx] is equal to 1, vertex[nodeIdx][vIdx][k] is added as follows.

for (n = 0; n < 8; n++){  
 if (activeCorner[nodeIdx][n]){  
 vertex[nodeIdx][vIdx][0] = ((TriSoupNodeSize[nodeIdx][0] × Bit(n, 0)) << 8) – 128  
 vertex[nodeIdx][vIdx][1] = ((TriSoupNodeSize[nodeIdx][1] × Bit(n, 1)) << 8) - 128  
 vertex[nodeIdx][vIdx][2] = ((TriSoupNodeSize[nodeIdx][2] × Bit(n, 2)) << 8) - 128  
 vIdx++  
 }  
}

Number of vertices numTriSoupVertices[nodeIdx] is set by vIdx.

numTriSoupVertices[nodeIdx] = vIdx

##### Derive initial position of centroid vertices

The initial position of centroid vertices Centroid[nodeIdx][k] for each TriSoup node nodeIdx is derived as follows as a weighted sum. Firstly, weights weights[i] associated with edge vertices of the node are initialized to zero.

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 weights[i] = 0  
}

A total weight wTotal is also initialized to zero.

wTotal = 0

Then, weights weights[i] and total weight wTotal are derived as follows.

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 w = abs(vertex[nodeIdx][i][0] - vertex[nodeIdx][i2][0])  
 w += abs(vertex[nodeIdx][i][1] - vertex[nodeIdx][i2][1])  
 w += abs(vertex[nodeIdx][i][2] - vertex[nodeIdx][i2][2])  
  
 weights[i] += w  
 weights[i2] += w  
 wTotal += 2 × w  
}

The position Centroid[nodeIdx][] of the centroid is obtained as a weighted sum of edge vertices.

Centroid[nodeIdx][0] = 0  
Centroid[nodeIdx][1] = 0  
Centroid[nodeIdx][2] = 0  
for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 Centroid[nodeIdx][0] += weights[i] × vertex[nodeIdx][i][0]  
 Centroid[nodeIdx][1] += weights[i] × vertex[nodeIdx][i][1]  
 Centroid[nodeIdx][2] += weights[i] × vertex[nodeIdx][i][2]  
}  
Centroid[nodeIdx][0] = Centroid[nodeIdx][0] / wTotal  
Centroid[nodeIdx][1] = Centroid[nodeIdx][1] / wTotal  
Centroid[nodeIdx][2] = Centroid[nodeIdx][2] / wTotal

##### Determine the dominant axis and sort TriSoup vertices

When numTriSoupVertices[nodeIdx] is greater than 3, the dominant axis and sorted TriSoup vertices are derived as follows. Otherwise, the dominant axis is set to 0.

* The sum of triangle areas for each axis k (0, 1, 2) is derived as follows.
  + Derive two axes to calculate triangle areas based on Table 48.

Table 48 — Corresponding axes S1 and S2 with k

|  |  |  |
| --- | --- | --- |
| **Axis k** | **S1** | **S2** |
| 0 | 2 | 1 |
| 1 | 2 | 0 |
| 2 | 1 | 0 |

* + The score to sort each vertex is calculated in anti-clockwise order according to the edges of the projected square as follows:

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 if(vertex[nodeIdx][i][S1] >= TriSoupNodeSize[nodeIdx][S1]){  
 Score[i] = vertex[nodeIdx][i][S2]  
 } else if (vertex[nodeIdx][i][S2] >= TriSoupNodeSize[nodeIdx][S2]){  
 Score[i] = TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] -   
 vertex[nodeIdx][i][S1]  
 } else if (vertex[nodeIdx][i][S1] <= 0) {  
 Score[i] = 2 × TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] -   
 vertex[nodeIdx][i][S2]  
 } else {  
 Score[i] = 2 × TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] +   
 vertex[nodeIdx][i][S1]  
 }  
}

* + Sort vertices by ascending order of scores.
  + Calculate the sum of triangle areas as follows

for (Axis = 0; Axis < 3; Axis++){  
 Area = 0  
 for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 for (k = 0; k < 3; k++){  
 v1[k] = sortedVertices[nodeIdx][i][k] – Centroid[nodeIdx][k]  
 v2[k] = sortedVertices[nodeIdx][i2][k] – Centroid[nodeIdx][k]  
 }  
 cP = CrossProduct(v1,v2)  
 Area += abs(cP[Axis])   
 }  
}

* The axis with the largest area is determined as the dominant axis, and sorted vertices associated with the dominant axis is retained.

##### Derive refined centroid position

When all the following conditions are true, centroid position Centroid[nodeIdx][k] for each TriSoup node shall be refined.

* trisoup\_centroid\_vertex\_residual\_present is equal to 1,
* At least one of the following conditions are true,
  + numTriSoupVertices[nodeIdx] is greater than 3,
  + All the conditions listed in 9.4.3.2.8 are true.

Otherwise, output of this process is the same as the output of 9.4.3.2.3.

Refinement process of centroid position is as follows.

* Derive a normal vector by retained dominant axis and sorted vertices derived in 9.4.3.2.4.

accumNormal = 0  
if (numTriSoupVertices[nodeIdx] ≠ 2){  
 for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTrisoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 accumNormal = CrossProduct (sortedVertices[nodeIdx][i] – Centroid[nodeIdx],  
 sortedVertices[nodeIdx][i2] – Centroid[nodeIdx])  
 }  
} else {  
 accumNormal = Centroid[nodeIdx] - TriSoupNodeSize[ adjNode[nodeIdx][fvIdx] ] -  
 Centroid[adjNode[nodeIdx][fvIdx]]  
}  
absNormal = isqrt(accumNormal[0] × accumNormal[0] +   
 accumNormal[1] × accumNormal[1] +  
 accumNormal[2] × accumNormal[2])  
for (k = 0; k < 3; k++){  
 normal[k] = (accumNormal[k] << 8) / absNormal  
}

* Derive context indexes ctxMinMax for ctxDrift0 (centroid\_residual\_is\_zero[nodeIdx]) and context indexes ctxIdxDritSign, lowS and highS for ctxDriftSign (centroid\_residual\_sign[nodeIdx]) as defined in 9.4.3.2.9.
* Derive dequantized centroid position residual driftDQ from decoded centroid position residual driftQ.

driftDQ = 0  
if (driftQ){  
 driftDQ = abs(driftQ) << bitDropped + 6  
 half = 1 << 5 + bitDropped  
 driftDQ += 2 × half / 3 – half  
 if (driftQ < 0)  
 driftDQ = -driftDQ  
}

* Derive drift vector DriftVec[nodeIdx][].

for (k = 0; k < 3; k++) {  
 DriftVec[nodeIdx][k] = (driftDQ × normal[k]) >> 6  
}

* Refine centroid position.

for (k = 0; k < 3; k++) {  
 Centroid[nodeIdx][k] += DriftVec[nodeIdx] [k]  
}

* Bound centroid position.

for (k = 0; k < 3; k++) {  
 Centroid[nodeIdx][k] = max(-128, Centroid[nodeIdx][k])  
 Centroid[nodeIdx][k] =   
 min(((TriSoupCubeSize - 1) << 8) + 127, Centroid[nodeIdx][k])  
}

##### Derive TriSoup face vertices corresponding each TriSoup node

When trisoup\_face\_vertex\_enabled is equal to 1, for each TriSoup node, TriSoup face vertex faceVertex[nodeIdx][fvIdx] is set to the intersection of the face, which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin, and the line segment between the centroid vertex of nodeIdx-th TriSoup node and that of adjNode[nodeIdx][fvIdx]-th TriSoup node for each fvIdx (fvIdx = 0, 1, 2) as follows:

c0facePos = (TriSoupNodeSize[nodeIdx][fvIdx] << 8 ) − 128  
c0 = Centroid[nodeIdx]  
c1 = Centroid[adjNode[nodeIdx][fvIdx]]  
c1[fvIdx] += TriSoupNodeSize[nodeIdx][fvIdx] << 8  
denom = c1[fvIdx] − c0[fvIdx]  
t = denom ? (((c0facePos − c0[fvIdx]) << 8) / denom) : 0  
faceVertex[nodeIdx][fvIdx] = c0 + ((t × (c1 − c0) + 128) >> 8)  
faceVertex[nodeIdx][fvIdx][fvIdx] = c0facePos

##### Insert TriSoup face vertices to corresponding sorted TriSoup vertices

For each TriSoup node, when has\_face\_vertex[nodeIdx][fvIdx] is equal to 1 and numTriSoupVertices[nodeIdx] is not equal to 2, faceVertex[nodeIdx][fvIdx] is added to sortedVertices[nodeIdx] and sortedVertices[adjNode[nodeIdx][fvIdx]] by the following steps for each fvIdx (fvIdx = 0, 1, 2):

* faceVertex[nodeIdx][fvIdx] is inserted between sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]] and sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][1]].
* Face vertex adjFaceVertex to be added to sortedVertices[adjNode[nodeIdx][fvIdx]] is derived as follows:

adjFaceVertex = faceVertex[nodeIdx][fvIdx]  
adjFaceVertex[fvIdx] = −128

* Indices of sortedVertices[adjNode[nodeIdx][fvIdx]] adjEvIdxMin0 and adjEvIdxMin1 are derived as follows:

evCnt = numTriSoupVertices[adjNode[nodeIdx][fvIdx]]  
distMin = 0x7fffffff  
for(evIdx = 0; evIdx < ((evCnt == 3) ? 1 : evCnt); evIdx++){  
 ev0 = evIdx  
 ev1 = evIdx + 1  
 if( ev1 >= evCnt )  
 ev1 −= evCnt  
 evCoord0 = sortedVertices[adjNode[nodeIdx][fvIdx]][ev0] + 128  
 evCoord1 = sortedVertices[adjNode[nodeIdx][fvIdx]][ev1] + 128  
 if( (evCoord0[fvIdx] ≠ 0) || (evCoord1[fvIdx] ≠ 0) )  
 continue  
 middlePoint = ( evCoord0 + evCoord1 ) / 2  
 distVec = (middlePoint – adjFaceVertex) >> 8  
 dist = distVec[0] × distVec[0] + distVec[1] × distVec[1] +   
 distVec[2] × distVec[2]  
 if( distMin > dist ) {  
 adjEvIdxMin0 = ev0  
 adjEvIdxMin1 = ev1  
 distMin = dist  
 }  
}

* adjFaceVertex is inserted between sortedVertices[adjNode[nodeIdx][fvIdx]][adjEvIdxMin0] and sortedVertices[adjNode[nodeIdx][fvIdx]][adjEvIdxMin1].

For each TriSoup node, when has\_face\_vertex[nodeIdx][fvIdx] is equal to 1 and numTriSoupVertices[nodeIdx] is equal to 2 (there are only 2 vertices in sortedVertices), faceVertex[nodeIdx][fvIdx] is added to sortedVertices[nodeIdx] by the following steps for each fvIdx (fvIdx = 0, 1, 2):

* If face\_vertex[nodeIdx][fvIdx] shares the same face as the first TriSoup edge vertex in sortedVertices[nodeIdx], insert it ahead of the first element.
* If face\_vertex[nodeIdx][fvIdx] shares the same face as the second TriSoup edge vertex in sortedVertices[nodeIdx], insert it after the last element.

##### Applicable conditions of non-closed surface reconstruction

The following conditions are applicable conditions of non-closed surface reconstruction for a TriSoup node corresponding to nodeIdx.

* trisoup\_face\_vertex\_enabled is equal to 1.
* numTrisoupVertices[nodeIdx] is equal to 2.
* onSameFace, which indicates that edge vertices vertex[nodeIdx][0] and vertex[nodeIdx][1] are positioned on the same node face, is true.

onSameFace = false  
for (k = 0; k < 3; k++) {  
 if (vertex[nodeIdx][0][k] == vertex[nodeIdx][1][k]) {  
 onSameFace = true  
 }  
}

* notOnSameEdge, which indicates that edge vertices vertex[nodeIdx][0] and vertex[nodeIdx][1] are not positioned on the same TriSoup edge, is true.

notOnSameEdge = true  
counter = 0  
for (k = 0; k < 3; k++) {  
 if (vertex[nodeIdx][0][k] == vertex[nodeIdx][1][k] &&  
 (vertex[nodeIdx][0][k] < 0 ||  
 vertex[nodeIdx][0][k] >= (TriSoupNodeSize[nodeIdx] << 8) – 128)) {  
 counter++  
 }  
}  
if (counter >= 2){  
 notOnSameEdge = false  
}

##### Derive context indexes for refined centroid position

Derive context indexes for refined centroid position as follows.

* Derive context indexes ctxMinMax for ctxDrift0 (centroid\_residual\_is\_zero[nodeIdx]).

minPos = sortedVertices[nodeIdx][0][dominantAxis]  
maxPos = sortedVertices[nodeIdx][0][dominantAxis]  
for (i = 1; i < numTriSoupVertices[nodeIdx]; i++){  
 if (sortedVertices[nodeIdx][i][dominantAxis] < minPos)  
 minPos = sortedVertices[nodeIdx][i][dominantAxis]  
 if (sortedVertices[nodeIdx][i][dominantAxis] > maxPos)  
 maxPos = sortedVertices[nodeIdx][i][dominantAxis]  
}  
ctxIdxDrift0 = min(8, (maxPos - minPos) >> (8 + bitDropped))

* Derive expressions highBound[nodeIdx] and lowBound[nodeIdx], context indexes ctxIdxDritSign, lowS and highS for ctxDriftSign (centroid\_residual\_sign[nodeIdx]).

halfDropped = bitDropped == 0 ? 0 : 1 << (bitDropped - 1)  
bound = (TriSoupNodeSize[nodeIdx][dominantAxis] – 1) << 8  
for (m = 1; m < TriSoupNodeSize[nodeIdx][dominantAxis]; m++){  
 temp = Centroid[nodeIdx] + m × normal  
 if (temp[0] < 0 || temp[0] > bound   
 || temp[1] < 0 || temp[1] > bound  
 || temp[2] < 0 || temp[2] > bound)  
 break  
}  
highBound[nodeIdx] = (m – 1) + halfDropped >> bitDropped  
  
for (m = 1; m < TriSoupNodeSize[nodeIdx][dominantAxis]; m++){  
 temp = Centroid[nodeIdx] - m × normal  
 if (temp[0] < 0 || temp[0] > bound   
 || temp[1] < 0 || temp[1] > bound  
 || temp[2] < 0 || temp[2] > bound)  
 break  
}  
lowBound[nodeIdx] = (m – 1) + halfDropped >> bitDropped  
  
lowBoundSurface = ((Centroid[nodeIdx][dominantAxis] – minPos) + 128 >> 8) + halfDropped >> bitDropped  
highBOundSurface = ((maxPos – Centroid[nodeIdx][dominantAxis]) + 128 >> 8) + halfDropped >> bitDropped  
  
lowS = min(7, lowBoundSurface)  
highS = min(7, highBoundSurface)  
ctxIdxDriftSign = lowBound[nodeIdx] == highBound[nodeIdx] ?  
 0 : 1 + (lowBound[nodeIdx] < highBound[nodeIdx])

#### Determination of decoded points by the voxelization of TriSoup triangles

In this process, decoded points by the voxelization of TriSoup triangles are derived and added to PointPos as follows.

* Set PointCnt is equal to 0.
* Apply the following processes for each TriSoup nodes with nodeIdx
  + Initialize numDecPointsInNode as 0.
  + Point positions corresponding to TriSoup edge vertices are derived and added to decPoints[nodeIdx][] as defined in 9.4.3.3.1.
  + When numTriSoupVertices[nodeIdx] is greater than or equal to 3, or all the conditions listed in 9.4.3.2.8 are true, following processes are applied.
    - When numTriSoupVertices[nodeIdx] is greater than 3, add centroid to decoded points as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented.
    - Construct TriSoup triangles as defined in 9.4.3.3.2.
    - Derive point positions corresponding to each triangle and add the points to decode points as defined in 9.4.3.3.3.
  + Eliminate duplicate points in decPoints[nodeIdx][]. numDecPointsInNode is updated to the number of decoded points in the node nodeIdx after elimination of duplicated points. Because list of points decPoints[nodeIdx][] are disjoint by construction, elimination of duplicated points can be performed locally for each node nodeIdx individually independently on other nodes.
  + Add decoded points to pointPos.

for (i = 0; i < numDecPointsInNode; PointCnt++, i++){  
 for (k = 0; k < 3; k++){  
 pointPos[PointCnt][k] = decPoints[nodeIdx][numDecPointsInNode][k]  
 }  
}

##### Adding edge vertices to decoded points

The following processes applied for all vIdx (vIdx = 0,…,numTriSoupVertices[nodeIdx] - 1).

* Point position d\_point[k] is derived:

for (k = 0; k < 3; k++) {  
 d\_point[k] = vertex[nodeIdx][vIdx][k] + 128 >> 8  
}

* When all the following conditions are true, d\_point is added as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented.
* d\_point belongs to the node nodeIdx. The condition for the d\_point belongs to the node nodeIdx is, for all k (k=0, 1, 2), d\_point[k] is greater than or equal to TriSoupNodeLoc[nodeIdx][k] and lower than or equal to TriSoupNodeLoc[nodeIdx] [k] + TriSoupCubeSize – 1.
* bitDropped is greater than 0 or samplingValue is greater than 1.

if (pointWithinNode(d\_point) && (bitDropped > 0 || samplingValue > 1)){  
 for (k = 0; k < 3; k++) {  
 decPoints[nodeIdx][numDecPointsInNode][k] = d\_point[k]  
 }  
 numDecPointsInNode++  
}  
where  
 pointWithinNode(point) :=  
 point[0] >= nodeLoc[0]  
 && point[0] <= nodeLoc[0] + TrisoupCubeSize – 1  
 && point[1] >= nodeLoc[1]  
 && point[1] <= nodeLoc[1] + TrisoupCubeSize – 1  
 && point[2] >= nodeLoc[2]  
 && point[2] <= nodeLoc[2] + TrisoupCubeSize – 1  
 where  
 nodeLoc = TriSoupNodeLoc[nodeIdx]

##### Construct TriSoup Triangles

Construct TriSoup Triangles for each TriSoup node corresponding to nodeIdx as follows.

* The number of sorted vertices numSortedVertices is set as the number of vertices in sortedVertices[nodeIdx].
* Set numTriangles as follows.
  + numTriangles = 1 (if numSortedVertices is equal to 3),
  + numTriangles = numSortedVertices (otherwise).
* triangles[i][j][k] are derived as follows.

for (i = 0; i < numTriangles; i++){  
 for (k = 0; k < 3; k++){  
 triangles[i][0][k] = sortedVertices[nodeIdx][i][k]  
 }  
 i2 = i + 1  
 if (numTriSoupVertices[nodeIdx]==2 && i2 >= numTriSoupVertices[nodeIdx])  
 break  
 else if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 for (k = 0; k < 3; k++){  
 triangles[i][1][k] = sortedVertices[nodeIdx][i2][k]  
 triangles[i][2][k] = numTriangles == 1 ?   
 sortedVertices[nodeIdx][i + 2][k] : Centroid[nodeIdx][k]  
 }  
}

##### Derive point positions corresponding to each triangle and add to decoded points

The following processes operated for each triangle with index i (=0,…,numTriangles).

1. In this process, *edge1*[], *edge2*[], *rayVector*[], *v0*[], preC[4], *h* and *a* shall be computed into 64 bits signed integer registers if letting the possibly of computing them into 64 bits signed integer registers.

* Derive edge vectors of a triangle.

edge1 = triangles[i][1] - triangles[i][0]  
edge2 = triangles[i][2] - triangles[i][0]

* Determine the two directions for ray tracing.

minVal = 1 << 28  
directionExcluded = 0  
h = CrossProduct(edge1, edge2) >> 8  
for (k = 0; k < 3; k++){  
 rayVector = {0, 0, 0}  
 rayVector[k] = 256  
 a = (rayVector × h) >> 8  
 if (abs(a) < minVal){  
 minVal = abs(a)  
 directionExcluded = k  
}

* minRange and maxRange are derived as follows.

for (k = 0; k < 3; k++){  
 minPos = min(triangles[i][0][k], min(triangles[i][1][k], triangles[i][2][k]))  
 minRange[k] = max(0, minPos + 128 >> 8)  
 maxPos = max(triangles[i][0][k], max(triangles[i][1][k], triangles[i][2][k]))  
 maxRange[k] = min(TriSoupNodeSize[nodeIdx][k], maxPos + 128 >> 8)  
}

* precompute preC[4] as follows.

v0 = triangles[i][0]  
preC[0] = edge1[1] × edge2[2] - edge1[2] × edge2[1]  
preC[1] = edge1[2] × edge2[0] - edge1[0] × edge2[2]  
preC[2] = edge1[0] × edge2[1] - edge1[1] × edge2[0]  
tmp = CrossProduct(v0, edge1) >> 8  
preC[3] = tmp[0] × edge2[0] + tmp[1] × edge2[1] + tmp[2] × edge2[2]

* ray tracing is applied to each direction except for the direction corresponding to directionExcluded. decPoints[nodeIdx][] and numDecPointsInNode are updated as described in 9.4.3.3.4.

##### Ray tracing

In this process, decoded points are generated from a TriSoup triangle by ray tracing as follows.

1. In the processes defined in 9.4.3.3.4, 9.4.3.3.5, 9.4.3.3.6, and 9.4.3.3.7, rayOrigin[], c[4], a1, a2, a3, deltaT1, deltaT2, V0[2], V1[2], V2[2], S and P[2]shall be computed into 64 bits signed integer registers if letting the possibly of computing them into 64 bits signed integer registers.

* Initialize variables rayVector[3], h, a, startposG1, startposG2, endposG1, endposG2, and rayOrigin[3] as defined in 9.4.3.3.5.
* haloTriangle, haloThickness and haloTriangle2D are set as defined in 9.4.3.3.6.
* isVisible[129][129] is initialized as follows.

for (i = startposG1; i <= endposG1; i += samplingValue)  
 for (j = startposG2; j <= endposG2; j += samplingValue)  
 isVisible[i][j] = false

* A variable g1 is initialized as startposG1.
* When g1 is less than or equal to endposG1, the following processes applied.
  + rayOrigin is updated.

rayOrigin[g1pos[direction]] = g1 << 8

* + A variable g2 is initialized as startposG2.
  + When g2 is less than or equal to endposG2, the following processes applied.
    - rayOrigin is updated.

rayOrigin[g2pos[direction]] = g2 << 8

* + - P[2] is initialized.

P = {g1 << 8, g2 << 8}

* + - isVisible[g1][ g2] is updated.

isVisible[g1][g2] = isPointInTriangle(P, V0, V1, V2, haloTriangle2D) ? true : false  
where  
 isPointInTriangle(P, V0, V1, V2, haloTriangle2D) :=  
 ((P[0] - V0[0]) × (V1[1] - V0[1]) - (P[1] - V0[1]) × (V1[0] - V0[0])  
 >= haloTriangle2D)  
 && ((P[0] - V1[0]) × (V2[1] - V1[1]) - (P[1] - V1[1]) × (V2[0] - V1[0])  
 >= haloTriangle2D)  
 && ((P[0] - V2[0]) × (V0[1] - V2[1]) - (P[1] - V2[1]) × (V0[0] - V2[0])  
 >= haloTriangle2D)

* + - intersection[3] is initialized.

intersection[0] = rayOrigin[0]  
intersection[1] = rayOrigin[1]  
intersection[2] = rayOrigin[2]

* + - intersection is updated.

if ((g2 - samplingValue) >= startposG2 && isVisible[g1][g2 - samplingValue])  
 tBuffer[g1][g2] = tBuffer[g1][g2 - samplingValue] + deltaT2  
else if ((g1 - samplingValue) >= startposG1 && isVisible[g1 - samplingValue][g2])  
 tBuffer[g1][g2] = tBuffer[g1 - samplingValue][g2] + deltaT1  
else  
 tBuffer[g1][g2] = ((P[0] × a1) >> kTriSoupFpBits)+ ((P[1] × a2) >>   
 kTriSoupFpBits) + a3  
intersection[direction] = intersection[direction] + tBuffer[g1][g2]

* + - When isVisible[g1][ g2] is true, decPoints[nodeIdx][] and numDecPointsInNode are updated as defined in 9.4.3.3.7.
    - g2 is updated to search the next point.

g2 += samplingValue

* + g1 is updated to search the next point.

g1 += samplingValue

##### Initialize ray tracing parameters

Parameters related to the ray tracing process are initialized as follows.

* Initialize rayVector[3].

rayVector[0] = 0  
rayVector[1] = 0  
rayVector[2] = 0  
rayVector[direction] = 1 << 8

* Calculate a cross product of rayVector and edge2 as h.

h = CrossProduct(rayVector, edge2) >> 8

* Calculate an inner product of edge1 and h as a.

a = InnerProduct(edge1, h) >> 8

* If a is greater than 256, the following processes are applied. Otherwise, the ray tracing process is completed for the input triangle and the direction.
* Starting positions startposG1 and startposG2, and ending positions endposG1 and endposG2 are derived.

g1pos[3] = { 1, 0, 0 }  
g2pos[3] = { 2, 2, 1 }  
startposG1 = minRange[g1pos[direction]]  
startposG2 = minRange[g2pos[direction]]  
endposG1 = maxRange[g1pos[direction]]  
endposG2 = maxRange[g2pos[direction]]

* rayOrigin[3] is initialized.

rayOrigin[0] = minRange[direction][0] << 8  
rayOrigin[1] = minRange[direction][1] << 8  
rayOrigin[2] = minRange[direction][2] << 8

##### Set halo parameters

Parameters related to halo are set as follows.

* haloTriangle is set.
  + If trisoup\_halo\_enabled is equal to 0, haloTriangle is set as 0.
  + Otherwise, if samplingValue is strictly greater than 1, the value of haloTriangle is obtained depending on trisoup\_adaptive\_halo\_enabled and samplingValue by

haloTriangle = trisoup\_adaptive\_halo\_enabled? 50 × samplingValue : 50  
haloTriangle = haloTriangle > 100 ? 100 : haloTriangle

* + Otherwise, if samplingValue is equal to 1, the value of haloTriangle is obtained depending on bitDropped by

haloBit = (((1 << bitDropped) - 1) << 8) / TriSoupCubeSize  
haloBit = (haloBit × 24) / 32  
haloTriangle = haloBit > 40 ? 40 : haloBit

* haloThickness is set depending on samplingValue.

haloThickness = samplingValue > 1 ? 16 : 32

* c[4], a1, a2, a3, deltaT1 and deltaT2 are calculated as follows.

c[0] = preC[0] / a  
c[1] = preC[1] / a  
c[2] = preC[2] / a  
c[3] = preC[3] / a

a1 = c[g1pos[direction]]  
a2 = c[g2pos[direction]]  
a3 =((c[direction] × rayOrigin[direction]) >> 8) - c[3]

deltaT1 = a1 × samplingValue  
deltaT2 = a2 × samplingValue

* V0[2], V1[2], V2[2] are set.

V0 = {triangles[i][0][g1pos[direction]], triangles[i][0][g2pos[direction]]}  
V1 = {triangles[i][1][g1pos[direction]], triangles[i][1][g2pos[direction]]}  
V2 = {triangles[i][2][g1pos[direction]], triangles[i][2][g2pos[direction]]}

* A variable S is calculated as follows.

S = (V0[0] - V1[0]) × (V2[1] - V1[1]) - (V0[1] - V1[1]) × (V2[0] - V1[0])

* Determine whether to swap V0 and V2 according to the value of S as follows.

if (S < 0)  
 swap(V0,V2)

* 2D halo parameter haloTriangle2D is calculated as follows.

haloTriangle2D = -(((abs(S) + 128) >> 8) × haloTriangle)

##### Point position reconstruction

Point positions based on triangles are reconstructed and added to the decoded point cloud as follows.

* Voxel position foundvoxel[3] is calculated as follows.

for (k = 0; k < 3; k++)  
 foundvoxel[k] = (TriSoupNodeLoc[nodeIdx][k] + intersection[k] + 128) >> 8

* Voxel position foundvoxelUp[3] is calculated as follows.

for (k = 0; k < 3; k++)  
 intersectionUp[k] = intersection[k]  
intersectionUp[direction] += haloThickness  
for (k = 0; k < 3; k++){  
 foundvoxelUp[k] =   
 (TriSoupNodeLoc[nodeIdx][k] + intersectionUp[k] + 128) >> 8  
}

* Voxel position foundvoxelDown[3] is calculated as follows.

for (k = 0; k < 3; k++)  
 intersectionDown[k] = intersection[k]  
intersectionDown[direction] -= haloThickness  
for (k = 0; k < 3; k++){  
 foundvoxelDown[k] =   
 (TriSoupNodeLoc[nodeIdx][k] + intersectionDown[k] + 128) >> 8  
}

* foundvoxel, foundvoxelUp and foundvoxelDown are added as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented respectively, under the condition they belong the TriSoup node nodeIdx. The condition for a point voxel belongs to the node nodeIdx is, for all k (k=0, 1, 2), voxel[k] is greater than or equal to TriSoupNodeLoc[nodeIdx] [k] and lower than or equal to TriSoupNodeLoc[nodeIdx][k] + TriSoupCubeSize – 1.

if(pointWithinNode(foundvoxel)){  
 for (k = 0; k < 3; k++)  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxel[k]  
 numDecPointsInNode++  
}  
if(pointWithinNode(foundvoxelUp)){  
 for (k = 0; k < 3; k++)  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxelUp[k]  
 numDecPointsInNode++  
}  
if(pointWithinNode(foundvoxelDown)){  
 for (k = 0; k < 3; k++)  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxelDown[k]  
 numDecPointsInNode++  
}  
where  
 pointWithintNode(point) :=  
 point[0] >= nodeLoc[0]  
 && point[0] <= nodeLoc[0] + TrisoupCubeSize – 1  
 && point[1] >= nodeLoc[1]  
 && point[1] <= nodeLoc[1] + TrisoupCubeSize – 1  
 && point[2] >= nodeLoc[2]  
 && point[2] <= nodeLoc[2] + TrisoupCubeSize – 1  
 where  
 nodeLoc = TriSoupNodeLoc[nodeIdx]

# Slice attributes

## General

Clause 10 specifies the reconstruction of a single slice attribute for the coded slice geometry. The reconstructed attribute values are stored in the array PointAttr.

## Point coordinates

### General

Attribute coding can use either the slice's reconstructed STV point positions or the points' scaled angular coordinates.

The expression AttrPos[ ptIdx ][ 𝑘 ] specifies the coordinates of each point for attribute coding:

* When attr\_coord\_conv\_enabled is 0 and attr\_inter\_prediction\_enabled is 0, AttrPos is equivalent to PointPos, which is the slice geometry in the slice’s coordinate system.
* When attr\_coord\_conv\_enabled is 0 and attr\_inter\_prediction\_enabled is 1, AttrPos is equivalent to the slice geometry translated to the coding coordinate system by the addition of the slice origin, SliceOrigin.
* Otherwise, AttrPos[ ptIdx ][ 𝑘 ] are angular point coordinates as specified by 10.2.2.

AttrPos[ptIdx][k] := attr\_coord\_conv\_enabled  
 ? AttrPosAng[ptIdx][k]  
 : (attr\_inter\_prediction\_enabled   
 ? PointPos[ptIdx][k] + SliceOrigin[k]  
 : PointPos[ptIdx][k])

### Conversion to scaled angular coordinates

The conversion is specified by the expression AttrPosAng[ ptIdx ][ 𝑘 ].

When geom\_tree\_type is equal to 1 and  slice\_attr\_inter\_prediction is equal to 1, the point’s angular coordinates shall be offset by the minimum value between the minimum angular coordinates of the current slice and previously applied offset value. The minimum angular coordinates of the current slice are specified by the expression *MinCurAng*[*k*]. The previously applied offset value is specified by the expression *MinRefAng*[*k*]. The minimum value between *MinRefAng* and *MinCurAng* is specified by the expression *MinAng*[*k*].

MinCurAng[k] := geom\_tree\_type == 1 && k == 1  
 ? −Exp2(ptree\_ang\_azimuth\_pi\_bits\_minus11 + 10) : 0

MinAng[k] := geom\_tree\_type == 1 && slice\_attr\_inter\_prediction == 1  
 ? Min(MinCurAng[k], MinRefAng[k]) : MinCurAng[k]

Otherwise, the point's angular coordinates shall be offset by the minimum angular coordinates.

The offset coordinates shall be scaled. Any negative coordinate after conversion shall be clipped to 0.

AttrPosAng[ptIdx][k] := DivExp2Up(relPos × attr\_coord\_conv\_scale[k], 8)  
 where  
 relPos := Max(0, PointAng[ptIdx][k] – MinAng[k])

It is a requirement of bitstream conformance that attr\_coord\_conv\_scale shall not cause any converted coordinate, AttrPosAng[ ptIdx ][ 𝑘 ], to be greater than Exp2( MaxSliceDimLog2 ) − 1.

When geom\_tree\_type is equal to 1, after the coordinate’s conversion of a slice, *MinRefAng* shall be set equal to *MinAng.*

## Syntax element semantics

#### Attribute data unit coefficients

The array AttrCoeff, with elements AttrCoeff[ coeffIdx ][ 𝑐 ], contains transform coefficient values. Elements of the array shall be initialized to zero.

zero\_run\_length\_prefix, zero\_run\_length\_minus3\_div2, zero\_run\_length\_minus3\_mod2 and zero\_run\_length\_minus11 together specify, in accordance with the expression ZeroRunLength, the number of consecutive transform coefficient tuples with all components equal to zero. Any of zero\_run\_length\_minus3\_div2, zero\_run\_length\_minus3\_mod2 and zero\_run\_length\_minus11 that are not present shall be inferred to be 0.

ZeroRunLength := zero\_run\_length\_prefix  
 + 2 × zero\_run\_length\_minus3\_div2 + zero\_run\_length\_minus3\_mod2  
 + zero\_run\_length\_minus11

#### Attribute coefficient tuples

Attribute coefficient values are signalled for a coeffIdx-th coefficient tuple when at least one component is not equal to zero.

coeff\_abs[ 𝑐 ] and coeff\_sign[ 𝑐 ] together specify the 𝑐-th transform coefficient component AttrCoeff[ coeffIdx ][ 𝑐 ]. coeff\_sign[ 𝑐 ] specifies whether (when 0) the coefficient's sign is positive or (when 1) negative. If coeff\_sign[ 𝑐 ] is not present, it shall be inferred to be 0.

The coefficients of the coeffIdx-th tuple are specified by the derivation of AttrCoeff:

for (c = 0; c < AttrDim; c++){  
 absVal = coeff\_abs[c]  
 if (c == AttrDim − 1)  
 if (AttrDim == 1  
 || AttrDim == 2 && coeff\_abs[0] == 0  
 || AttrDim == 3 && coeff\_abs[0] == 0 && coeff\_abs[1] == 0)  
 absVal++  
 AttrCoeff[coeffIdx][(c + 1) % AttrDim] = (1 − 2 × coeff\_sign[c]) × absVal  
}

1. When a point is eligible for direct prediction, the LSBs of coeff\_abs encode the direct predictor mode.

#### Raw attribute values

raw\_attr\_component\_length, when present, specifies the length in bytes of each syntax element raw\_attr\_value.

raw\_attr\_value[ ptIdx ][ 𝑐 ] specifies the attribute value for the 𝑐-th component of the ptIdx-th point in canonical decoding order. The length in bits of each syntax element is specified by the expression RawAttrValueBits.

RawAttrValueBits := raw\_attr\_width\_present  
 ? 8 × raw\_attr\_component\_length  
 : AttrBitDepth

## Raw attribute decoding

This subclause applies when attr\_coding\_type is 3.

Attribute values shall be set equal to the corresponding raw\_attr\_value syntax elements.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[ptIdx][c] = raw\_attr\_value[ptIdx][c]

## Attribute decoding using the region-adaptive hierarchical transform

### General

The region-adaptive hierarchical transform specified by 10.5 is a recursive two-point transform. It applies when attr\_coding\_type is 0.

The transform constructs a spatial tree of 3D transform blocks using the slice geometry (10.5.2). Basis vectors are calculated for each application of the transform, weighted in proportion to the significance of each coefficient.

When lossless\_coding\_enabled is 1, a transform domain prediction process predicts AC coefficients from the DC coefficients of certain adjoining blocks. When slice\_attr\_inter\_prediction is 1, a spatial tree of reference 3D transform blocks is constructed using the reference slice geometry (10.5.2.6). The transform domain inter prediction process predicts the DC coefficient of the transform block of the root node in the spatial tree from the DC coefficient of certain reference block. For layers other than the root, the transform domain inter prediction process modifies the prediction of AC coefficients from the AC coefficients of certain reference block.

When lossless\_coding\_enabled is 0, a sample domain prediction process predicts DC coefficients of the children nodes from the DC coefficients of certain adjoining blocks. When slice\_attr\_inter\_prediction is 1, a spatial tree of reference 3D transform blocks is constructed using the reference slice geometry (10.5.2.6). The sample domain inter prediction process predicts the DC coefficient of the root node in the spatial tree from the DC coefficient of certain reference block. For layers other than the root, the sample domain inter prediction predicts the DC coefficients of the children nodes.

The reconstruction process is specified for a single attribute component Cidx ∈ 0 .. AttrDim − 1. It is skipped in transform level Lvl when:

* Lvl is greater than 0 and the number of blocks in this level is equal to the number of blocks in the previous transform level Lvl + 1, or
* Lvl is equal to 0 and the number of blocks in this transform level is equal to the number of coded points.

The reconstruction process starts by:

* mapping coded coefficients to the transform tree (10.5.3) and
* scaling the coded coefficients (10.5.4).

Then in turn for each level, starting from the root of the transform tree (Lvl = RahtRootLvl) and proceeding down the tree until completing level 0, :

* performing transform domain prediction (10.5.5) when lossless\_coding\_enabled is 1 or performing sample domain prediction (10.5.6) when lossless\_coding\_enabled is 0,
* applying the inverse transform (10.5.7),
* adding sample domain prediction (10.5.7) when lossless\_coding\_enabled is 0

The reconstructed attribute values are specified by 10.5.8.

### Transform tree

#### General

The tree of transform bocks is defined recursively:

* In tree level 0, each block groups together points with identical attribute coordinates.
* Each subsequent tree level 𝑙 shrinks the preceding level 𝑙 − 1 by a factor of two in each dimension; each 2×2×2 block groups together up to eight blocks from the preceding level.

An example tree is illustrated in Figure 28. The points a to f are grouped into blocks according to their attribute coordinates: C groups together c, d and e. The weight of each block is the number of points spanned by the block (10.5.2.3): C has a weight of 3; B with a weight of 5 groups together C, b and f; A with a weight of 6 groups together B and a.

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| a to f | Points |
| A to C | Transform blocks |
| × | Transform coefficients at input to inverse transform for labelled block |
| ○ | Inverse transformed coefficient |
| ● | Inherited DC coefficient (See RahtDcCoeff) |
| 1 to 6 | Coefficient weights (See RahtCoeffWeightM) |

Figure 28 — Example RAHT tree, block weights and transform structure

#### State variables

The RAHT tree is specified in terms of the following state variables:

* The sparse array RahtCoeff of transform block coefficients; RahtCoeff[ lvl ][ bs ][ bt ][ bv ][ 𝑖 ] is the 𝑖-th coefficient for the block located at ( bs, bt, bv ) in transform level lvl. Unset elements shall be inferred to be 0.
* The array RahtBlkLoc of transform block locations; RahtBlkLoc[ lvl ][ nIdx ][ 𝑘 ] is the location of the nIdx-th coded block in transform level lvl.
* The array RahtBlkCnt of node counts per tree level; RahtBlkCnt[ lvl ] is the number of blocks in transform level lvl.
* The variable RahtLvlCnt, the number of transform levels.

#### Transform block weight

The weight of the DC transform coefficient for a block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkWeight[ lvl ][ bs ][ bt ][ bv ]. It is equal to the number of points that the coefficient applies to.

* 1. The sum of all block weights in any transform level is equal to the number of coded points (PointCnt).
  2. A block's weight is equal to the sum of its child block weights.

RahtBlkWeight[lvl][bs][bt][bv] :=  
 RahtBlkWeight = 0  
 for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 RahtBlkWeight += isPointInSubtree[ptIdx]  
 where  
 isPointInSubtree[ptIdx] :=  
 bs == AttrPos[ptIdx][0] >> lvl  
 && bt == AttrPos[ptIdx][1] >> lvl  
 && bv == AttrPos[ptIdx][2] >> lvl

#### Number of transform levels and per-level block order

The root node of the transform tree is the lowest block in the tree with a DC coefficient that spans the entire geometry; i.e. it has a weight equal to the number of coded points, PointCnt.

The tree level containing the root node is RahtRootLvl:

RahtRootLvl := RahtLvlCnt − 1

Within a transform level, blocks are ordered for coefficient coding by ascending Morton-coded block location, as specified by the derivation of RahtBlkLoc. Empty blocks are ignored.

for (RahtLvlCnt = 0; !done; RahtLvlCnt++)  
 for (mIdx = 0, nIdx = 0, wSum = 0; wSum < PointCnt; mIdx++) {  
 (bs, bt, bv) = FromMorton(mIdx)  
 wSum += RahtBlkWeight[RahtLvlCnt][bs][bt][bv]  
 if (RahtBlkWeight[RahtLvlCnt][bs][bt][bv] == 0)  
 continue  
 RahtBlkCnt[RahtLvlCnt]++  
 RahtBlkLoc[RahtLvlCnt][nIdx][0] = bs  
 RahtBlkLoc[RahtLvlCnt][nIdx][1] = bt  
 RahtBlkLoc[RahtLvlCnt][nIdx][2] = bv  
 nIdx++  
 done = RahtBlkWeight[RahtLvlCnt][bs][bt][bv] == PointCnt  
 }

#### 2×2×2 transform block coefficient weights

Transform coefficient weights are specified for each directional stage of the two-point transform and inverse transform for 2×2×2 transform blocks by the expression RahtCoeffWeightM[ lvl ][ stage ][ bs ][ bt ][ bv ][ 𝑚 ]; the parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl, lvl > 0;
* 𝑚 specifies the transform coefficient index in forward transform stage or inverse transform stage stage.

RahtCoeffWeightM[lvl][stage][bs][bt][bv][m] := RahtCoeffWeight[lvl][stage][s][t][v]  
 where  
 s := 2 × bs + FromMorton[m][0]  
 t := 2 × bt + FromMorton[m][1]  
 v := 2 × bv + FromMorton[m][2]

Within a block, coefficient weights are determined iteratively starting from its child block weights (stage 0) to the transform block coefficient weights of stage 3. At each stage, for each pair of inverse-transformed values 𝑎 and 𝑏, the weight for the DC (wL) and AC (wH) coefficient is the sum of the weights for a and b. If the weight for either 𝑎 or 𝑏 is 0, the AC coefficient weight is 0.

The expression RahtCoeffWeight[ lvl ][ stage ][ 𝑠 ][ 𝑡 ][ 𝑣 ] specifies the derivation of a weight in transform stage stage for the coefficient corresponding to the block located at ( 𝑠, 𝑡, 𝑣 ) in tree level lvl − 1.

RahtCoeffWeight[lvl][stage][s][t][v] :=  
 stage == 0 ? RahtBlkWeight[lvl − 1][s][t][v] :  
 stage == 1 ? v % 2 == 0 ? wL[0][0][1] : wH[0][0][−1] :  
 stage == 2 ? t % 2 == 0 ? wL[0][1][0] : wH[0][−1][0] :  
 stage == 3 ? s % 2 == 0 ? wL[1][0][0] : wH[−1][0][0] : na  
 where  
 wL[ds][dt][dv] := wSum[ds][dt][dv]  
 wH[ds][dt][dv] := wSum[ds][dt][dv] × wHnz[ds][dt][dv]  
 wSum[ds][dt][dv] := wP[s][t][v] + wP[s + ds][t + dt][v + dv]  
 wHnz[ds][dt][dv] := wP[s][t][v] × wP[s + ds][t + dt][v + dv] > 0  
 wP[s][t][v] := RahtCoeffWeight[lvl][stage − 1][s][t][v]

1. RahtBlkWeight[ lvl ][ 𝑠 ][ 𝑡 ][ 𝑣 ] ≡ RahtCoeffWeight[ lvl ][ 3 ][ 2 × 𝑠 ][ 2 × 𝑡 ][ 2 × 𝑣 ]; lvl > 0.

In the example of Figure 28, block B has stage 0 coefficient weights of 1, 3 and 1; stage 1 and 2 weights of 1, 4 and 1; and stage 3 weights of 5, 4 and 5. RahtCoeffWeight[ 1 ][ 1 ][ 3 ][ 0 ][ 2 ] would be 4.

#### Reference transform tree

The tree of reference transform blocks is defined recursively, applying the same process as the tree of transform blocks and using the reference slice geometry RefAttrPos:

* In tree level 0, each block groups together points with identical attribute coordinates.
* Each subsequent tree level 𝑙 shrinks the preceding level 𝑙 − 1 by a factor of two in each dimension; each 2×2×2 block groups together up to eight blocks from the preceding level.

The weight of the DC transform coefficient for a reference block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkWeightRef[ lvl ][ bs ][ bt ][ bv ]. It is equal to the number of points that the coefficient applies to, which is derived from the RefAttrPos in a similar way as RahtBlkWeight in 10.5.2.3.

The sum of attributes of points within each reference block is recorded when defining the tree of reference transform blocks. The sum of attributes of points within a reference block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ], which is derived from *RefPointAttr*.

Transform coefficient weights are specified for each directional stage of the two-point transform for reference 2×2×2 transform blocks by the expression RahtCoeffWeightMRef[ lvl ][ stage ][ bs ][ bt ][ bv ][ 𝑚 ], which is derived from RahtBlkWeightRef in a similar way as RahtCoeffWeightM in 10.5.2.5; the parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl, lvl > 0;
* 𝑚 specifies the transform coefficient index in forward transform stage stage.

### Coefficient order

#### General

Subclause 10.5.3 specifies the correspondence between coded transform coefficients and the transform tree.

Starting from the root of the transform tree and proceeding in breadth-first order, coefficients are coded for each transform block; all transform blocks within one tree level are coded before those of the next level. Within a tree level, blocks shall be traversed in ascending Morton order of block location.

The order of coefficients within a transform block is specified by 10.5.3.2 for 2×2×2 blocks (tree levels greater than 0) and 10.5.3.3 for blocks of co-located points (tree level 0).

The mapping from the coded order to the transform tree is specified in terms of the following variables:

* Lvl, the index of the mapped transform level.
* CoeffIdx, the index into the decoded coefficient array AttrCoeff for the next mapped coefficient.

CoeffIdx = 0  
for (Lvl = RahtLvlCnt; Lvl ≥ 0; Lvl−−) {  
 if (Lvl > 0)  
 if (RahtBlkCnt[Lvl] != RahtBlkCnt[Lvl-1]) {  
 … /\* See 10.5.3.2 \*/  
 }   
 else   
 if (RahtBlkCnt[Lvl] != PointCnt) {  
 … /\* See 10.5.3.3 \*/  
 }  
}

#### Mapping for a tree level of 2×2×2 transform blocks

This subclause applies to tree levels greater than 0.

For each 2×2×2 transform block, up to 7 AC coefficients are mapped from the bitstream to coefficient indexes within the block. In the case of the root transform block, the DC coefficient is additionally mapped.

Table 49 specifies the order in which transform block coefficients are coded; RahtCoeffOrder[ 𝑖 ] is the block index of the 𝑖-th coded coefficient.

Only coefficients with a non-zero transform coefficient weight are coded.

Table 49 — 2×2×2 RAHT coefficient coding order

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| RahtCoeffOrder[ 𝑖 ] | 0 | 4 | 2 | 1 | 6 | 5 | 3 | 7 |

for (nIdx = 0; nIdx < RahtBlkCnt[Lvl]; nIdx++) {  
 bs = RahtBlkLoc[Lvl][nIdx][0]  
 bt = RahtBlkLoc[Lvl][nIdx][1]  
 bv = RahtBlkLoc[Lvl][nIdx][2]  
 for (i = 0; i < 8; i++) {  
 /\* skip the DC coefficient that will be inherited \*/  
 if (i == 0 && Lvl < RahtRootLvl)  
 continue  
 if (RahtCoeffWeightM[Lvl][3][bs][bt][bv][RahtCoeffOrder[i]] > 0)  
 RahtCoeff[Lvl][bs][bt][bv][RahtCoeffOrder[i]] = AttrCoeff[CoeffIdx++][Cidx]  
 }  
}

#### Mapping for co-located points

This subclause applies to the final tree level (Lvl == 0), after all other tree levels have been mapped.

Each transform block with a node weight 𝑤 greater than 1 codes 𝑤 − 1 AC coefficients with the same attribute coordinates.

for (nIdx = 0; nIdx < RahtBlkCnt[0]; nIdx++) {  
 ns = RahtBlkLoc[0][nIdx][0]  
 nt = RahtBlkLoc[0][nIdx][1]  
 nv = RahtBlkLoc[0][nIdx][2]  
 for (i = 1; i < RahtBlkWeight[0][ns][nt][nv]; i++)  
 RahtCoeff[0][ns][nt][nv][i] = AttrCoeff[CoeffIdx++][Cidx]  
}

### Coefficient scaling

#### General

Subclause 10.5.4 specifies the scaling of coded coefficients for a block located at ( Bs, Bt, Bv ) in tree level Lvl. It shall be applied to every block in every tree level in any order.

If a regional QP offset is present (i.e. attr\_region\_cnt > 0), a tree (10.5.4.4) is specified that blends QP offsets along region boundaries according to the structure of the RAHT tree.

#### For a transform block

Within a transform block, coded coefficients shall be scaled according to a per-coefficient QP. The DC coefficient is not scaled except when coded in the root node of the transform tree.

mCnt := Lvl > 0 ? 8 : RahtBlkWeight[0][Bs][Bt][Bv]  
for (m = 0; m < mCnt; m++) {  
 /\* skip the DC coefficient that will be inherited \*/  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] = RahtCoeffScaled[Lvl][Bs][Bt][Bv][m]  
}

The scaling of the 𝑚-th coded coefficient of a transform block located at ( bs, bt, bv ) in tree level lvl is specified by the expression RahtCoeffScaled[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ]: it is scaled by the fixed-point step size AttrQstep[ qp ] (10.7.4) and represented as a 15 fractional-bit, fixed-point coefficient value.

RahtCoeffScaled[lvl][bs][bt][bv][m] := coeff × AttrQstep[qp] << 7  
 where  
 coeff := RahtCoeff[lvl][bs][bt][bv][m]  
 qp := lossless\_coding\_enabled == 0 ? RahtCoeffQp[lvl][bs][bt][bv][m] : 4

#### Per coefficient QP

The expression RahtCoeffQp[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ] specifies the QP for the 𝑚-th coefficient of the transform block located at ( bs, bt, bv ) in tree level lvl for the Cidx-th attribute component:

* rgnOffset[ qc ] is the per-coefficient offset from the region-dependent QP offset tree.
* dpth is the depth of the transform block in the RAHT tree.
* ACqpoffset is the per-coefficient AC coefficient QP offset.

RahtCoeffQp[lvl][bs][bt][bv][m] := AttrQp[dpth][rgnOffset][Cidx > 0] + ACqpoffset  
 where  
 dpth := RahtRootLvl − lvl  
 isACoffset = dpth <= attr\_AC\_qp\_layer\_cnt\_minus1 && m > 0  
 ACqpoffsetP = isACoffset ? attr\_AC\_qp\_offset[ dpth ][0][m – 1] : 0  
 ACqpoffsetS = isACoffset ? attr\_AC\_qp\_offset[ dpth ][1][m – 1] : 0  
 Acqpoffset := Cidx > 0 ? ACqpoffsetS : ACqpoffsetP  
 rgnOffset[qc] := RahtTreeQpOffsetM[lvl][bs][bt][bv][m][qc]

#### Region-dependent QP offset tree

The integer, averaged region-dependent QP offset for each coefficient of a 2×2×2 transform block is specified by the expression RahtTreeQpOffsetM[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ]. The parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl;
* 𝑚 specifies the transform coefficient index from the final forward transform stage.

RahtTreeQpOffsetM[lvl][bs][bt][bv][m] :=  
 lvl == 0 ? RahtTreeQpOffset[ 0][3][bs][bt][bv][qc] >> 4  
 : RahtTreeQpOffset[lvl][3][ms][mt][mv][qc] >> 4  
 where  
 ms := 2 × bs + FromMorton[m][0]  
 mt := 2 × bt + FromMorton[m][1]  
 mv := 2 × bv + FromMorton[m][2]

The fixed-point, region-dependent QP offset tree is structurally identical to the transform tree. It is specified recursively for a QP component qc by RahtTreeQpOffset[ lvl ][ 𝑠 ][ 𝑡 ][ 𝑣 ][ qc ]:

* For tree level 0, the offset is the regional QP offset for a point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ).
* For each subsequent tree level 𝑙, a 2×2×2 block of QP offsets is averaged for each transform stage in turn. Each QP offset in a block at stage 0 is that of the DC transform block coefficient for a child block in the preceding level 𝑙 − 1.

Within a block, each subsequent transform stage averages, along the transformed axis, adjacent pairs of QP offsets from the preceding stage that have a non-zero transform coefficient weight. For a pair of QPs 𝑎 and 𝑏 with respective weights wa and wb:

* The weight for the corresponding DC coefficient is 𝑎 + 𝑏 divided by 2 if wa and wb are non-zero, or 1 otherwise.
* The weight for the corresponding AC coefficient is 𝑏 if wa and wb are non-zero, or 0 otherwise.

Averages shall be calculated using four fractional bits.

RahtTreeQpOffset[lvl][stage][s][t][v][qc] :=  
 lvl == 0 ? RahtRegionQpOffset[qs][qt][qv][qc] << 4 :  
 stage == 0 ? RahtTreeQpOffset[lvl − 1][3][2 × s][2 × t][2 × v][qc] :  
 stage == 1 ? v % 2 == 0 ? qpL[0][0][1] : qpH[0][0][−1] :  
 stage == 2 ? t % 2 == 0 ? qpL[0][1][0] : qpH[0][−1][0] :  
 stage == 3 ? s % 2 == 0 ? qpL[1][0][0] : qpH[−1][0][0] : na  
 where  
 qpL[ds][dt][dv] := qpSum[ds][dt][dv] >> qpHnz[ds][dt][dv]  
 qpH[ds][dt][dv] := qpC[s][t][v] × qpHnz[ds][dt][dv]  
 qpSum[ds][dt][dv] := qpC[s][t][v] + qpC[s + ds][t + dt][v + dv]  
 qpHnz[ds][dt][dv] := wC[s][t][v] × wC[s + ds][t + dt][v + dv] > 0  
 qpC[s][t][v] := RahtTreeQpOffset[lvl][stage − 1][s][t][v][qc]  
 wC[s][t][v] := RahtCoeffWeight[lvl][stage − 1][s][t][v]

An example tree is illustrated in Figure 29 for the transform tree of Figure 28. The hatched area has a regional QP offset of +6; co-located points c, d and e have an offset of +6; points b and f, 0. In block A, the stage 3 QPs are used to scale the transform coefficients. For the coefficient at 𝑚 = 0, 3¾, the QP is the mean of the stage 2 QPs 1½ and 6; for the coefficient at 𝑚 = 4, it is 1½. Scaling of coefficient uses the integer part of the fractional QP.

图示, 工程绘图

描述已自动生成

Key

|  |  |
| --- | --- |
| b to f | Points (See Figure 28) |
| A to C | Transform blocks (See Figure 28) |
| × | Transform coefficients at input to inverse transform for labelled block |
| ○ | Inverse transformed coefficient |
| ● | Inherited DC coefficient (See RahtDcCoeff) |
| 0 to 6 | QP values (See RahtTreeQpOffset) |

Figure 29 — Example region-dependent QP offset tree

### Transform domain prediction

#### General

Subclauses 10.5.5.2 - 10.5.5.6 apply when raht\_prediction\_enabled\_flag is 1 and lossless\_coding\_enabled is 1, subclauses 10.5.5.7 - 10.5.5.9 apply when slice\_attr\_inter\_prediction is 1 and lossless\_coding\_enabled is 1. These specify the transform domain prediction for a block located at ( Bs, Bt, Bv ) in tree level Lvl. These shall be performed for every eligible block for intra prediction (10.5.5.2) or inter prediction (10.5.5.7) in the tree level, in any order, by:

* generating a prediction block (10.5.5.3) and/or;
* generating a inter prediction block and resampling the inter prediction block (10.5.5.8);
* applying the forward transform for the intra prediction block (10.5.5.5) and/or;
* applying the forward transform for the inter prediction block (10.5.5.9);
* modifying the prediction block from the inter prediction block;
* adding the resulting AC transform coefficients to the coefficient residuals in the coefficient tree.

The prediction block and its transform are specified in terms of the eight-element array RahtPredBlk; RahtPredBlk[ 𝑚 ] is the prediction block value for the Morton-coded location 𝑚. The inter prediction block and its transform are specified in terms of the eight-element array RahtInterPredBlk; RahtInterPredBlk[ 𝑚 ] is the inter prediction block value for the Morton-coded location 𝑚.

if (RahtPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtPredBlk[m] = (RahtPredW[m] >> 15} << 15  
  
 … /\* in−place, forward transform of RahtPredBlk (10.5.5.5) \*/  
  
}   
if (RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtInterPredBlk[m] = RahtInterPredW[m]  
  
 … /\* in−place, forward transform of RahtInterPredBlk （10.5.5.9） \*/  
  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtPredBlk[m] = RahtInterPredBlk[m]  
 }  
}  
if (RahtPredEligible[Lvl][Bs][Bt][Bv] || RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] += RahtPredBlk[m]  
 }  
}

#### Eligibility

When enabled, transform domain prediction shall be performed for 2×2×2 transform blocks unless the block:

* is the root of the transform tree; or
* has only one non-empty child block; or
* is adjoined (10.5.5.3) by fewer than raht\_prediction\_samples\_min non-empty blocks; or
* has an ancestor, except the root block, that is adjoined by fewer than raht\_prediction\_subtree\_min non-empty blocks.

The expression RahtBlkChildCnt[ lvl ][ bs ][ bt ][ bv ] is the number of non-empty child blocks of the block located at ( bs, bt, bv ) in tree level lvl .

RahtBlkChildCnt[lvl][bs][bt][bv] :=  
 RahtBlkChildCnt = 0  
 for(m = 0; m ≤ 8; m++){  
 if(RahtCoeffWeightM[lvl][0][bs][bt][bv][m] > 0)  
 RahtBlkChildCnt ++   
 }

The expression RahtPredEligible[ lvl ][ bs ][ bt ][ bv ] specifies whether the transform block located at ( bs, bt, bv ) in tree level lvl is eligible.

RahtPredEligible[lvl][bs][bt][bv] := raht\_prediction\_enabled\_flag  
 && lvl > 0  
 && lvl < RahtRootLvl  
 && !DisabledBySendingMode  
 && RahtBlkChildCnt[lvl][bs][bt][bv] > 1  
 && RahtNeighCnt[lvl][bs][bt][bv] ≥ raht\_prediction\_samples\_min  
 && RahtNeighCntMinAncestor[lvl][bs][bt][bv] ≥ raht\_prediction\_subtree\_min  
 where  
 DisabledBySendingMode := (raht\_intra\_layer\_code\_enabled && depth < layer\_code\_depth &&  
 depth ≥ 0) ? !slice\_raht\_intra\_layer\_code\_mode[depth] : 0  
 depth := RahtRootLevel – lvl – 1

#### Generation of prediction block from adjoining blocks

A prediction block is generated from up to 19 transform blocks that contain a DC coefficient: the co-located block and those that adjoin the predicted block by a face or an edge.

The expression RahtNeighCnt[ lvl ][ bs ][ bt ][ bv ] is the number of non-empty blocks that can be used to predict the block located at ( bs, bt, bv ), which is defined by (RahtBlkLoc[ lvl ][ nIdx ][ 0 ], RahtBlkLoc[ lvl ][ nIdx ][ 1 ], RahtBlkLoc[ lvl ][ nIdx ][ 2]), is the location of the nIdx-th coded block in tree level lvl. When the block located at ( bs, bt, bv ) in tree level lvl has only one non-empty child block*, the* number of non-empty blocks that can be used to predict that block is set as 19*.*

RahtNeighCnt[lvl][bs][bt][bv] :=   
 RahtBlkChildCnt[lvl][bs][bt][bv] == 1 ? 19 : SumN19[neighWeightGt0]  
 where  
 neighWeightGt0[ds][dt][dv] := RahtBlkWeight[lvl][bs + ds][bt + dt][bv + dv] > 0 &&  
 ¬isNeighOutsideSearchRange  
 isNeighOutsideSearchRange :=  
 Morton(bs + ds, bt + dt, bv + dv) < Morton(bs, bt, bv) ?  
 Morton(bs + ds, bt + dt, bv + dv) < Morton(LE(0), LE(1), LE(2)) :  
 Morton(bs + ds, bt + dt, bv + dv) > Morton(BE(0), BE(1), BE(2))  
 sr := raht\_prediction\_search\_range  
 LE(k) := RahtBlkLoc[lvl][Max(nIdx - sr, 0)][k]  
 BE(k) := RahtBlkLoc[lvl][Min(nIdx + sr, RahtBlkCnt[lvl] - 1)][k]

The expression RahtNeighCntMinAncestor[ lvl ][ bs ][ bt ][ bv ] is lowest value of RahtNeighCnt for any ancestor of the block located at ( bs, bt, bv ) in tree level lvl. In determining eligibility, the root node shall be considered to have 19 adjoining blocks.

RahtNeighCntMinAncestor[lvl][bs][bt][bv] :=  
 lvl >= RahtRootLvl − 1 ? 19 : Min(neighCntP, minAncestorCnt)  
 where  
 neighCntP := RahtNeighCnt[lvl + 1][bs / 2][bt / 2][bv / 2]  
 minAncestorCnt := RahtNeighCntMinAncestor[lvl + 1][bs / 2][bt / 2][bv / 2]

The expression SumN19[ expr ] sums the result of applying expr to the relative tree location of each of the 19 possible adjacent blocks.

SumN19[expr] :=  
 SumN19 = 0  
 for (ds = −1; ds ≤ 1; ds++)  
 for (dt = −1; dt ≤ 1; dt++)  
 for (dv = −1; dv ≤ 1; dv++)  
 if (Abs(ds) + Abs(dt) + Abs(dv) < 3)  
 SumN19 += expr[ds][dt][dv]

#### Upsampling

##### Normalized DC values

The samples used to generate an upsampled prediction block are transform-block DC coefficients (10.5.7.2) normalized by their weight as specified by RahtDcNorm, that is a 15 fractional-bit, fixed-point value; RahtDcNorm[ lvl ][ bs ][ bt ][ bv ] is the sample value for the block located at ( bs, bt, bv ) in tree level lvl.

RahtDcNorm[lvl][bs][bt][bv] := raht\_buffer\_extension\_flag == 0 ?  
 (lossless\_coding\_enabled == 0 ? DivExp2Fz((coeff >> wShift) × (IntRecipSqrt(w) >> 25 −  
 wShift), 30) : DivExp2Fz(coeff, 15)) :  
 (lossless\_coding\_enabled == 0 ? DivExp2Fz((coeff >> wShift) × (IntRecipSqrt(w) >> 25 −  
 wShift), 15) : DivExp2Fz(coeff, 0))  
 where  
 w := RahtBlkWeight[lvl][bs][bt][bv]  
 coeff := RahtDcCoeff[lvl][bs][bt][bv]  
 wShift := w > 1024 ? IntLog2(w − 1) >> 1 : 0

##### Exclusion of adjoining blocks

Adjoining blocks shall be excluded from the upsampling process if either their weight is zero or the normalized DC value for their primary attribute component is:

* less than or equal to 0,2 times that of co-located block; or
* greater than or equal to 2,5 times that of the co-located block.

The expression RahtPredExcluded[ ds ][ dt ][ dv ] specifies whether the block with relative location ( ds, dt, dv ) is excluded from contributing to the upsampled prediction of the block ( Bs, Bt, Bv ).

RahtPredExcluded[ds][dt][dv] :=  
 Cidx == 0 ? empty || sample ≤ limitMin || sample ≥ limitMax  
 : … /\* Value of RahtPredExcluded[ds][dt][dv] for Cidx == 0 \*/  
 where  
 empty := RahtBlkWeight[Lvl][Bs][Bt][Bv] == 0  
 sample := 10 × RahtSample[Lvl][Bs + ds][Bt + dt][Bv + dv]  
 limitMin := 2 × RahtSample[Lvl][Bs][Bt][Bv]  
 limitMax := 25 × RahtSample[Lvl][Bs][Bt][Bv]

##### Eligibility of replacing adjoining blocks with child blocks

Adjoining block shall be replaced by its one child block to calculate the sample value of the prediction block for the Morton-coded sample location *m* in the block ( Bs, Bt, Bv ) when the eligibility conditions are meet. The child block to replace the adjoining block is the child block with relative location ( dsc, dtc, dvc ) to the 𝑚-th child block in tree level lvl - 1. The replacement shall be performed when the following eligibility conditions are true:

* raht\_subnode\_prediction\_enabled is 1;
* The Morton code of the Adjoining block is less than the Morton code of the block;
* The Adjoining block adjoin the block by a face and it has a non-empty child block that adjoin the 𝑚-th child block of the block by a face; or the Adjoining block adjoin the block by an edge and it has a non-empty child block that adjoin the 𝑚-th child block of the block by an edge.

The expression *IsReplaced*[ ds ][ dt ][dv][ 𝑚 ] specifies whether the adjoining block with relative location ( ds, dt, dv ) is replaced by its child block to calculate the sample of the prediction block for the 𝑚-th child block of the block ( Bs, Bt, Bv ).

IsReplaced[ds][dt][dv][m] :=  
 isAdj && isEarly && (RahtBlkWeight[lvl-1][bsc + dsc][btc + dtc][bvc + dvc] > 0)  
 where  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)  
 isAdj := (m & adjMask) == adjLoc  
 isEarly := Morton(Bs + ds, Bt + dt, Bv + dv) < Morton(Bs, Bt, Bv)  
 bsc := 2 × Bs + FromMorton[m][0]  
 btc := 2 × Bt + FromMorton[m][1]  
 bvc := 2 × Bv + FromMorton[m][2]  
 dsc := ds  
 dtc := dt  
 dvc := dv

##### Upsampled prediction block

The samples of the 2x2x2 prediction block are the weighted averages of the adjoining blocks' normalized DC values when raht\_subnode\_prediction\_enabled is 0. For each sample, the weight for the adjoining block depends upon the relative positions of the block and the sample location.

The expression RahtPred[ 𝑚 ], that is a 15 fractional-bit, fixed-point value, specifies the value for the Morton-coded sample location m; where:

* SumN19[ 𝑤 ] is sum of the weight of each adjoining block;
* SumN19[ wNeigh ] is the 15 fractional-bit, fixed-point the sum of the weighted normalized DC values for each adjoining block;
* the 15 fractional-bit, fixed-point reciprocal of the sum of weights, RahtPredRecipW[ 𝑥 ] is specified by Table 50.

1. Only samples that correspond to child blocks with non-zero block weights need to be calculated.

RahtPred[m] := (raht\_buffer\_extension\_flag == 0 ? SumN19[wNeigh] << 15 : SumN19[wNeigh])  
 × RahtPredRecipW[SumN19[w]]  
 where  
 w[ds][dt][dv] := RahtPredExcluded[ds][dt][dv] && Abs(Morton(bs + ds, bt + dt, bv +  
 dv)- Morton(bs, bt, bv)) <=raht\_prediction\_search\_range ? 0 :  
 RahtPredWeight[ds][dt][dv][m]  
 wNeigh[ds][dt][dv] := w[ds][dt][dv] × RahtDcNorm[Lvl][Bs + ds][Bs + dt][Bs + dv]

The expression RahtPredWeight[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be 4 for the co-located block, 2 for blocks that adjoin by a face and 1 for blocks that adjoin by only an edge.

RahtPredWeight[ds][dt][dv][m] := (m & adjMask) == adjLoc ? weight : 0  
 where  
 weight := 4 >> (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

Table 50 — Values of RahtPredRecipW[ 𝑥 ]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑥 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| RahtPredRecipW[ 𝑥 ] | 8 192 | 6 554 | 5 461 | 4 681 | 4 096 | 3 641 | 3 277 | 2 979 | 2 731 | 2 521 |

##### Replacement of adjoining blocks with child blocks

Replacement of adjoining blocks with their child blocks shall be performed when raht\_subnode\_prediction\_enabled is 1. The child blocks’ normalized DC values and weights are to replace the adjoining blocks’ normalized DC values and weights when the eligibility conditions are meet.

RahtPred[m] := (raht\_buffer\_extension\_flag == 0 ? SumN19[wNeigh] << 15 : SumN19[wNeigh])  
 × RahtPredRecipWC[SumN19[w]-1]  
 where  
 s := 2 × Bs + FromMorton[m][0]  
 t := 2 × Bt + FromMorton[m][1]  
 v := 2 × Bv + FromMorton[m][2]  
 RahtPredRealWeight[ds][dt][dv][m] :=   
 IsReplaced [ds][dt][dv][m] ? RahtPredWeightC[ds][dt][dv][m]  
 : RahtPredWeightB[ds][dt][dv][m]  
 w[ds][dt][dv] := RahtPredExcluded[ds][dt][dv] ? 0 : RahtPredRealWeight[ds][dt][dv][m]  
 RahtDcN := IsReplaced [ds][dt][dv][m] ? RahtDcNorm[lvl-1][s + ds][t + dt][v + dv]   
 : RahtDcNorm[Lvl][Bs + ds][Bt + dt][Bv + dv]  
 wNeigh[ds][dt][dv] := w[ds][dt][dv] × RahtDcN

The expression *RahtPredWeightB*[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the not replaced block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be specified by raht\_prediction\_weights[0] for the co-located block, raht\_prediction\_weights[1] for blocks that adjoin by a face and raht\_prediction\_weights[2] for blocks that adjoin by only an edge.

RahtPredWeightB[ds][dt][dv][m] := (m & adjMask) == adjLoc ? weight : 0  
 where  
 idx := (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 weight := raht\_prediction\_weights[idx]  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

The expression *RahtPredWeightC*[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the child block of the replaced block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be specified by raht\_prediction\_weights[3] for child blocks that adjoin by a face and raht\_prediction\_weights[4] for child blocks that adjoin by only an edge.

RahtPredWeightC[ds][dt][dv][m] := IsReplaced[ds][dt][dv][m] ? weight : 0  
 where  
 idx := (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 weight := raht\_prediction\_weights[2+idx]

The 15 fractional-bit, fixed-point reciprocal of the sum of weights, RahtPredRecipWC[ 𝑥 ] is specified by Table 51.

Table 51 — Values of RahtPredRecipWC[ 𝑥 ]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑥 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| RahtPredRecipWC[ 𝑥 ] | 32768 | 16384 | 10923 | 8192 | 6554 | 5461 | 4681 | 4096 | 3641 | 3277 |
| 𝑥 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| RahtPredRecipWC[ 𝑥 ] | 2979 | 2731 | 2521 | 2341 | 2185 | 2048 | 1928 | 1820 | 1725 | 1638 |
| 𝑥 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| RahtPredRecipWC[ 𝑥 ] | 1560 | 1489 | 1425 | 1365 | 1311 | 1260 | 1214 | 1170 | 1130 | 1092 |
| 𝑥 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| RahtPredRecipWC[ 𝑥 ] | 1057 | 1024 | 993 | 964 | 936 | 910 | 886 | 862 | 840 | 819 |
| 𝑥 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| RahtPredRecipWC[ 𝑥 ] | 799 | 780 | 762 | 745 | 728 | 712 | 697 | 683 | 669 | 655 |
| 𝑥 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| RahtPredRecipWC[ 𝑥 ] | 643 | 630 | 618 | 607 | 596 | 585 | 575 | 565 | 555 | 546 |
| 𝑥 | 60 | 61 | 62 | 63 |  |  |  |  |  |  |
| RahtPredRecipWC[ 𝑥 ] | 537 | 529 | 520 | 512 |  |  |  |  |  |  |

#### Forward transform for a 2×2×2 block prediction block

The forward transform for a 2×2×2 prediction block comprises transforming pairs of coefficients along each axis.

First, along the V axis:

rahtFwd1D[2][0][1]  
rahtFwd1D[2][2][3]  
rahtFwd1D[2][4][5]  
rahtFwd1D[2][6][7]

Second, along the T axis:

rahtFwd1D[1][0][2]  
rahtFwd1D[1][1][3]  
rahtFwd1D[1][4][6]  
rahtFwd1D[1][5][7]

Third, along the S axis:

rahtFwd1D[0][0][4]  
rahtFwd1D[0][1][5]  
rahtFwd1D[0][2][6]  
rahtFwd1D[0][3][7]

The expression rahtFwd1D[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place, forward, two-point transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis.

rahtFwd1D[k][aIdx][bIdx] := RahtFwd(aCoeff, bCoeff, wa, wb)  
 where  
 aCoeff := RahtPredBlk[aIdx]  
 bCoeff := RahtPredBlk[bIdx]  
 wa := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wb := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

#### Forward two-point transform

This subclause specifies the in-place, forward, two-point transform RahtFwd( aCoeff, bCoeff, wa, wb ). Its parameters are:

* the expressions aCoeff and bCoeff that identify the coefficients to be transformed in-place;
* the weights wa and wb that are the coefficient weights for aCoeff and bCoeff, respectively.

1. The specification of the forward two-point transform applies only to prediction blocks for the reconstruction of point attributes.

The transform basis vectors 15-fractional-bit, fixed-point coefficients 𝑎 and 𝑏:

a := IntSqrt(wa << 30) × IntRecipSqrt(wa + wb) >> 40  
b := IntSqrt(wb << 30) × IntRecipSqrt(wa + wb) >> 40

If both wa or wb are 0, the transform result is:

if (wa == 0 && wb == 0)  
 yl = yh = 0

If either wa or wb is 0, the transform result is:

if (wa == 0 || wb == 0) {  
 yl = wa ≠ 0 ? aCoeff : bCoeff  
 yh = 0  
}

1. If either wa or wb is zero, the respective opposing coefficient 𝑏 or 𝑎 is not necessarily .

Otherwise (both wa and wb are greater than 0), the transform result is:

if (wa ≠ 0 && wb ≠ 0) {  
 yh = bCoeff − aCoeff  
 yl = aCoeff + ((yh >> 16) << 15)  
}

The transform result replaces the input coefficients:

aCoeff = yl  
bCoeff = yh

#### Eligibility for inter prediction

When inter prediction is enabled, inter prediction shall be performed for 2×2×2 transform blocks unless:

* inter prediction is disabled for current level; or
* the block has no reference block which share the same location in the reference transform tree.

The expression RahtInterPredEligible[ lvl ][ bs ][ bt ][ bv ] specifies whether the transform block located at ( bs, bt, bv ) in tree level lvl is eligible for inter prediction.

RahtInterPredEligible[lvl][bs][bt][bv] := slice\_attr\_inter\_prediction  
 && lvl > 0  
 && lvl > (RahtRootLvl – raht\_inter\_layer\_depth\_minus1 - 1)  
 && !DisabledBySendingMode  
 && RahtBlkWeightRef[lvl][bs][bt][bv] > 0  
 where  
 DisabledBySendingMode := (raht\_inter\_layer\_code\_enabled && depth < layer\_code\_depth &&  
 depth ≥ 0) ? !slice\_raht\_inter\_layer\_code\_mode[depth] : 0  
 depth := RahtRootLevel – lvl – 1

#### Generation of inter prediction block

The samples used to generate an inter prediction block are the sum of point attributes within each child block of the reference block as specified by RahtInterPred; RahtInterPred[ lvl  - 1][ bsc ][ btc ][ bvc ] is the sample value for the child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

RahtInterPred[lvl][bs][bt][bv] := RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ]

Each sample in the prediction block is specified by RahtInterPredW[ m ] for the m-th child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

* If one child block is with zero weight, the sample used to generate the inter prediction block shall be set to 0;
* Else, if the corresponding child block of the reference block is with zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the reference block;
* Else, the corresponding child block of the reference block is with non-zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the corresponding child block of the reference block.

RahtInterPredW[ m ] := RahtBlkWeight[lvl - 1][bsc][btc][bvc] ?  
 (RahtBlkWeightRef[lvl - 1][bsc][btc][bvc] ? RahtInterPred[lvl – 1][bsc][btc][bvc] :  
 RahtInterPred[lvl][bs][bt][bv]) : 0

#### Forward transform for a 2×2×2 block inter prediction block

The forward transform for a reference 2×2×2 prediction block comprises transforming pairs of coefficients along each axis.

First, along the V axis:

rahtFwd1D[2][0][1]  
rahtFwd1D[2][2][3]  
rahtFwd1D[2][4][5]  
rahtFwd1D[2][6][7]

Second, along the T axis:

rahtFwd1D[1][0][2]  
rahtFwd1D[1][1][3]  
rahtFwd1D[1][4][6]  
rahtFwd1D[1][5][7]

Third, along the S axis:

rahtFwd1D[0][0][4]  
rahtFwd1D[0][1][5]  
rahtFwd1D[0][2][6]  
rahtFwd1D[0][3][7]

The expression rahtFwd1Dref[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place, forward, two-point transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis.

rahtFwd1Dref[k][aIdx][bIdx] := RahtFwd(aCoeffRef, bCoeffRef, waRef, wbRef)  
 where  
 aCoeffRef := RahtInterPredBlk[aIdx]  
 bCoeffRef := RahtInterPredBlk[bIdx]  
 waRef := RahtCoeffWeightMRef[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wbRef := RahtCoeffWeightMRef[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

### Sample domain prediction and last component prediction

#### General

Subclauses 10.5.5.2 - 10.5.5.4, subclause 10.5.6.2 and apply when raht\_prediction\_enabled\_flag is 1 and lossless\_coding\_enabled is 0, subclause 10.5.5.7 and subclauses 10.5.6.3 - 10.5.6.4 apply when slice\_attr\_inter\_prediction is 1 and lossless\_coding\_enabled is 0, subclause 10.5.6.5 applies when last\_comp\_pred\_enabled is 1 and lossless\_coding\_enabled is 0. These specify the sample domain prediction and last component prediction in transform domain for a block located at ( Bs, Bt, Bv ) in tree level Lvl. These shall be performed for every eligible block for intra prediction (10.5.5.2) or inter prediction (10.5.5.7) in the tree level, in any order, by:

* generating a prediction block (10.5.5.3) and/or;
* generating a inter prediction block (10.5.6.3);
* weighting the values of the prediction block (10.5.6.2);
* resampling and weighting the inter prediction block (10.5.6.3);
* if the DC values of the current block and the reference block are similar, applying temporal filtering (10.5.6.4) for the inter prediction block and modifying the prediction block from the inter prediction block;
* applying the last component prediction in transform domain(10.5.6.5);

The prediction block is specified in terms of the eight-element array RahtPredBlk; RahtPredBlk[ 𝑚 ] is the prediction block value for the Morton-coded location 𝑚. The inter prediction block is specified in terms of the eight-element array RahtInterPredBlk; RahtInterPredBlk[ 𝑚 ] is the inter prediction block value for the Morton-coded location 𝑚.

if (RahtPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtPredBlk[m] = RahtPredW[m]  
}   
if (RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtInterPredBlk[m] = RahtInterPredW[m]  
 if (RahtDcCoeffRef[lvl][Bs][Bt][Bv] < RahtDcCoeff[lvl][Bs][Bt][Bv] × 0.5 || RahtDcCoeffRef[lvl][Bs][Bt][Bv] > RahtDcCoeff[lvl][Bs][Bt][Bv] × 2)  
 continue (not inter-eligible)  
  
 … /\* in-place, temporal filtering of RahtInterPredBlk （10.5.6.4） \*/  
  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtPredBlk[m] = RahtInterPredBlk[m]  
 }  
}  
  
/\* in-place, last component prediction for RahtCoeff（10.5.6.5） \*/

#### Weighted prediction block

The upsampled prediction block shall be weighted with each sample weighted by the square root of the corresponding transform coefficient weight. The weighted sample value is specified by RahtPredW[ 𝑚 ] for the block located at ( Bs, Bt, Bv ) in tree level Lvl.

RahtPredW[m] := lossless\_coding\_enabled == 0 ? DivExp2Fz(RahtPred[m] × w, 15) :  
 (RahtPred[m] >> 15) << 15  
 where  
 w := IntSqrt(RahtCoeffWeightM[Lvl][0][Bs][Bt][Bv][m] << 30)

#### Resampling and Weighted inter prediction block

The samples used to generate an inter prediction block are the sum of point attributes within each child block of the reference block as specified by RahtInterPred; RahtInterPred[ lvl  - 1][ bsc ][ btc ][ bvc ] is the sample value for the child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

RahtInterPred[lvl][bs][bt][bv] := RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ]

The weighted sample value is specified by RahtInterPredW[ m ] for the m-th child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

* If one child block is with zero weight, the sample used to generate the inter prediction block shall be set to 0;
* Else, if the corresponding child block of the reference block is with zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the reference block and be weighted by the transform coefficient weight of the reference block;
* Else, the corresponding child block of the reference block is with non-zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the corresponding child block of the reference block and be weighted by its corresponding transform coefficient weight.

RahtInterPredW[ m ] := RahtBlkWeight[lvl - 1][bsc][btc][bvc] ?  
 (RahtBlkWeightRef[lvl - 1][bsc][btc][bvc] ?  
 DivExp2Fz((RahtInterPred[lvl – 1][bsc][btc][bvc] >> wShift ) × rsqrtW, 15) :  
 DivExp2Fz((RahtInterPred[lvl][bs][bt][bv] >> wShiftP ) × rsqrtWP, 15)) : 0  
 where  
 w := RahtBlkWeightRef[lvl - 1][bsc][btc][bvc]  
 wShift := w > 1024 ? IntLog2(w - 1) >> 1 : 0  
 rsqrtW := IntRecipSqrt(w) >> (25 – wShift)  
 w := RahtBlkWeightRef[lvl][bs][bt][bv]  
 wShiftP := wP > 1024 ? IntLog2(wP - 1) >> 1 : 0  
 rsqrtWP := IntRecipSqrt(wP) >> (25 – wShiftP)

#### RAHT temporal filtering

When the tree level lvl  of the current block is smaller than or equal to RahtRootLvl – raht\_inter\_skip\_layers, the temporal filtering is enabled for the transformed inter prediction block. The transformed inter-predicted block is filtered by the derived filter as specified by RahtInterFilter.

RahtInterPredBlk[m] = lvl <= (RahtRootLvl – raht\_skip\_layers) ? (RahtInterFilter ×  
 RahtInterPredBlk[m]) >> 7 : RahtInterPredBlk[m]

RahtInterFilter is determined for each tree level. If raht\_send\_filters is equal to 0, RahtInterFilter is determined by a set of stored filters as specified by FixedRahtFilters, based on the tree level lvl. If raht\_send\_filters is equal to 1, raht\_inter\_filter\_qidx[dpth - raht\_skip\_layers] is inverse quantized and subtracted from 128 to derive RahtInterFilter.

if(raht\_send\_filters)  
 RahtInterFilter = 128 – (raht\_inter\_filter\_qidx[dpth - raht\_skip\_layers] ×  
 AttrQstep[qp]>> 7)  
else  
 RahtInterFilter = FixedRahtFilters[LvlIdx]  
 where  
 FixedRahtFileters := [128, 128, 128, 127, 125, 121, 115]  
 LvlIdx := dpth > 6 ? 6 : dpth  
 dpth := RahtRootLevel – lvl

#### Last component prediction for RAHT

When attr\_coding\_type is 0 and raht\_last\_comp\_pred\_enabled is 1, the third attribute coefficient component is, if present, a residual to a prediction by the second scaled coefficient component. The expression *LcpScale*[lvl][nIdx]specifies the scale factor applied at the nIdx -th coded block in transform level lvl to the second coefficient components to predict third coefficient components. The scale factor *LcpScale*[lvl][nIdx]is calculated using the second and third components of the coefficients of its preceding 128 blocks.

if (AttrDim == 3) {  
 bs = RahtBlkLoc[Lvl][nIdx][0]  
 bt = RahtBlkLoc[Lvl][nIdx][1]  
 bv = RahtBlkLoc[Lvl][nIdx][2]  
 LcpScale[lvl][nIdx] = 0;  
 for(n = 0; n < Min(nIdx,128); n++){  
 ns = RahtBlkLoc[Lvl][nIdx - n][0]  
 nt = RahtBlkLoc[Lvl][nIdx - n][1]  
 nv = RahtBlkLoc[Lvl][nIdx - n][2]  
 for(m = 0; m < 8; m++){  
 if(m == 0 && Lvl < RahtRootLvl)  
 continue；  
 sumk1k1 += RahtCoeffLcp[lvl][ns][nt][nv][m][1] ×  
 RahtCoeffLcp[lvl][ns][nt][nv][m][1];  
 sumk1k2 += RahtCoeffLcp[lvl][ns][nt][nv][m][1] ×  
 RahtCoeffLcp[lvl][ns][nt][nv][m][2];  
 }  
 }  
 if (sumk1k2 && sumk1k1) {  
 LcpScale[lvl][nIdx] = Clip3(-16, 16, Div(sumk1k2, sumk1k2,4))  
 }  
 RahtCoeffLcp[lvl][bs][bt][bv][m][2] += DivExp2Floor(LcpScale[lvl][nIdx] ×  
 RahtCoeffLcp[lvl][bs][bt][bv][m][1],4)  
}

when AttrDim is 3, RahtCoeffLcp[ lvl ][ bs ][ bt ][ bv ][ m][ Cidx] specifies the coefficient residuals for the Cidx component of RahtCoeff[ lvl ][ bs ][ bt ][ bv ][ m], where Cidx ∈ 0 .. AttrDim – 1.

### Inverse transform

#### General

Subclause 10.5.7 specifies the inverse transform for a block located at ( Bs, Bt, Bv ) in tree level Lvl. It shall be applied to every block in the tree level in any order.

When sample domain prediction is applied to the transform block, the prediction is added to the inverse transformed block.

if (RahtPredEligible[Lvl][Bs][Bt][Bv] || RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] += RahtPredBlk[m]  
 }  
}

#### DC transform coefficient inheritance

Each block other than the root node of the transform tree shall inherit its DC coefficient from the corresponding inverse-transformed coefficient in its parent block.

When sample domain prediction is not applied to the transform block, the DC coefficient is obtained as follows:

If raht\_buffer\_extension\_flag is equal to 0, the inherited coefficient shall be rounded to retain two fractional bits, with half values rounded away from zero.

if (Lvl < RahtRootLvl)  
 RahtCoeff[Lvl][Bs][Bt][Bv][0] = raht\_buffer\_extension\_flag == 0 ?  
 DivExp2Fz(RahtDcCoeff[Lvl][Bs][Bt][Bv], 13) << 13 : RahtDcCoeff[Lvl][Bs][Bt][Bv]

For a block located at ( bs, bt, bv ) in tree level lvl, the corresponding coefficient in the parent block is specified by RahtDcCoeff[ lvl ][ bs ][ bt ][ bv ].

RahtDcCoeff[lvl][bs][bt][bv] := lvl < RahtRootLvl ?  
 RahtCoeff[lvl + 1][bs / 2][bt / 2][bv / 2][mP] : 0  
 where  
 mP := Morton[bs & 1][bt & 1][bv & 1]

When sample domain prediction is applied to the transform block, the DC coefficient is inferred as the difference between the inherited coefficient and the DC coefficient of the prediction block.

if (Lvl < RahtRootLvl)  
 RahtCoeff[Lvl][Bs][Bt][Bv][0] = RahtDcCoeff[Lvl][Bs][Bt][Bv] –  
 RahtPredDcCoeff[Lvl][Bs][Bt][Bv]

For a block located at ( bs, bt, bv ) in tree level lvl, the corresponding DC coefficient of the prediction block is specified by RahtPredDcCoeff[ lvl ][ bs ][ bt ][ bv ].

RahtPredDcCoeff[ lvl ][ bs ][ bt ][ bv ] = 0  
rsqrtWeightSum = IntRecipSqrt(RahtBlkWeight[Lvl][bs][bt][bv])  
for(m = 0; m < 8; m++){  
 bcs = bs << 1 + m & 1  
 bct = bt << 1 + m & 2  
 bcv = bv << 1 + m & 4  
 sqrtWeight = IntSqrt(RahtBlkWeight[Lvl - 1][bcs][bct][bcv] << 30)  
 normSqrtWeight = RahtBlkWeight[Lvl - 1][bcs][bct][bcv] == 1 ? rsqrtWeightSum >> 25 :  
 rsqrtWeightSum × sqrtWeight >> 40  
 predDC = normSqrtWeight \* RahtInterPredBlk[m] >> 15  
 RahtPredDcCoeff[ lvl ][ bs ][ bt ][ bv ] += RahtBlkWeight[Lvl - 1][bcs][bct][bcv] ? predDC : 0  
}

#### For a 2×2×2 transform block

The inverse transform for a 2×2×2 block located at ( Bs, Bt, Bv ) in tree level Lvl comprises transforming pairs of coefficients along each axis.

First, along the S axis:

rahtInv1D[0][0][4]  
rahtInv1D[0][1][5]  
rahtInv1D[0][2][6]  
rahtInv1D[0][3][7]

Second, along the T axis:

rahtInv1D[1][0][2]  
rahtInv1D[1][1][3]  
rahtInv1D[1][4][6]  
rahtInv1D[1][5][7]

Third, along the V axis:

rahtInv1D[2][0][1]  
rahtInv1D[2][2][3]  
rahtInv1D[2][4][5]  
rahtInv1D[2][6][7]

The expression rahtInv1D[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place inverse transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis of the block.

rahtInv1D[k][aIdx][bIdx] := RahtInv(aCoeff, bCoeff, wa, wb)  
 where  
 aCoeff := RahtCoeff[Lvl][Bs][Bt][Bv][aIdx]  
 bCoeff := RahtCoeff[Lvl][Bs][Bt][Bv][bIdx]  
 wa := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wb := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

#### For co-located points

The inverse transform for a block of co-located points located at ( Bs, Bt, Bv ) in tree level 0 comprises iteratively transforming the block's DC coefficient paired with each successive block coefficient.

for (i = 1; i < RahtBlkWeight[0][Bs][Bt][Bv]; i++)  
 rahtInvDup[i]

The expression rahtInvDup[ 𝑖 ] specifies the invocation of the in-place inverse transform for the 𝑖-th coefficient.

rahtInvDup[i] := RahtInv(aCoeff, bCoeff, wa, 1)  
 where  
 aCoeff := RahtCoeff[0][Bs][Bt][Bv][0]  
 bCoeff := RahtCoeff[0][Bs][Bt][Bv][i]  
 wa := RahtBlkWeight[0][Bs][Bt][Bv] − i

#### Inverse two-point transform

This subclause specifies the in-place, inverse, two-point transform RahtInv( aCoeff, bCoeff, wa, wb ). Its parameters are:

* the expressions aCoeff and bCoeff that respectively identify low- and high-frequency transform coefficients;
* the weights wa and wb that are the respective coefficient weights for aCoeff and bCoeff.

The transform basis vectors use 15-fractional-bit, fixed-point coefficients 𝑎 and 𝑏:

a := IntSqrt(wa << 30) × IntRecipSqrt(wa + wb) >> 40  
b := IntSqrt(wb << 30) × IntRecipSqrt(wa + wb) >> 40

If either wa or wb is 0, the transform result is:

if (wa == 0 || wb == 0) {  
 if (lossless\_coding\_enabled == 0) {  
 ya = wa ≠ 0 ? aCoeff : 0  
 yb = wb ≠ 0 ? aCoeff : 0  
 }  
 else {  
 ya = aCoeff – ((bCoeff >> 16) << 15)  
 yb = bCoeff + ya  
 }  
}

1. If either wa or wb is zero, the respective opposing coefficient 𝑏 or 𝑎 is not necessarily .

Otherwise (both wa and wb are greater than 0), the transform result is:

if (wa ≠ 0 && wb ≠ 0) {  
 ya = DivExp2Fz(aCoeff × a, 15) − DivExp2Fz(bCoeff × b, 15)  
 yb = DivExp2Fz(bCoeff × a, 15) + DivExp2Fz(aCoeff × b, 15)  
}

### Reconstructed attribute values

Reconstructed attribute values are specified by the expression RahtRecon[ 𝑠 ][ 𝑡 ][ 𝑣 ][ 𝑖 ] for an 𝑖-th co-located point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ). They are:

* extracted from inverse transformed blocks in the bottom two tree levels; unique points from tree level 1; duplicate points from tree level 0; then
* rounded to discard the 15 fractional bits of the fixed-point representation, with halve values rounded away from zero; then
* clipped to be within the attribute value range [ 0, AttrMaxVal ].

RahtRecon[s][t][v][i] := Clip3(0, AttrMaxVal, DivExp2Fz(value, 15))  
 where  
 value := RahtBlkWeight[0][s][t][v] == 1  
 ? RahtDcCoeff[0][s][t][v]  
 : RahtCoeff[0][s][t][v][i]

The mapping of the slice geometry to reconstructed attribute values shall map points with identical attribute coordinates to successive elements 𝑖 of RahtRecon[ 𝑠 ][ 𝑡 ][ 𝑣 ][ 𝑖 ] in canonical point order. i.e. the 𝑖-th element shall be the 𝑖-th instance of the attribute coordinates ( 𝑠, 𝑡, 𝑣 ) from the start of AttrPos.

The following is specified in terms of the sparse array dupPtIdx; dupPtIdx[ 𝑠 ][ 𝑡 ][ 𝑣 ] is the cumulative count of points with attribute coordinates ( 𝑠, 𝑡, 𝑣 ). Unset elements of dupPtIdx shall be inferred to be 0.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++) {  
 s = AttrPos[ptIdx][0]  
 t = AttrPos[ptIdx][1]  
 v = AttrPos[ptIdx][2]  
 i = dupPtIdx[s][t][v]  
 dupPtIdx[s][t][v]++  
 PointAttr[ptIdx][Cidx] = RahtRecon[s][t][v][i]  
}

## Attribute decoding using levels of detail

### General

The attribute decoding processes specified by 10.6 are distance-based prediction schemes that use a hierarchical level-of-detail representation of the slice geometry. They apply when attr\_coding\_type is either 1 or 2.

Detail levels are defined by an iterative subsampling process (10.6.5). The finest detail level comprises all points in the slice geometry. With each iteration, a coarser detail level is generated from the previous coarsest detail level.

Every detail level comprises a list of points present in the detail level, and is associated with a list of refinement points. A refinement point is a point that is present in a detail level and not present in any coarser detail level; the refinement points for detail level lvl, when combined with the coarser detail level lvl + 1, form detail level lvl.

For each refinement point, a set of neighbouring points is determined (10.6.6) using inter-detail-level, intra-detail-level and inter-frame searches. The neighbouring points form a predictor set that is used to predict attribute/transform coefficient values.

Attribute reconstruction (10.6.7) proceeds from the coarsest to the finest detail level. Transform coefficients are coded in the same order.

A coded transform coefficient is associated with each refinement point. The transform (10.6.12) comprises two operations: an update step that modifies the predicting points and a prediction step that adds the transform coefficient to a predicted attribute/coefficient value.

图片包含 图示

描述已自动生成

Figure 30 — Example of points in three detail levels and their spatial arrangement.

An example level-of-detail hierarchy is illustrated in Figure 30. is the finest detail level and corresponds to all points in the slice. The generation of subsequent detail levels is performed using periodic subsampling with samplingPeriod equal to 4. The number of detail levels is limited to 3. The points in that are not assigned to are in refinement list . The attribute value for the marked point ● in for is predicted from a set of spatially neighbouring points found using an inter-detail-level search of and an intra-detail-level search of points earlier in . Transform coefficient values are associated with each refinement point and coded from to . The refinement list comprises all points in .

### Syntax element semantics

lod\_dist\_log2\_offset specifies an offset to the APS-specified finest detail level block size lod\_initial\_dist\_log2 for LoD generation and predictor searches. When lod\_dist\_log2\_offset is not present, it shall be inferred to be 0.

### Reconstruction process

The reconstruction of point attribute values comprises:

* deriving a set of detail levels from the slice geometry (10.6.5);
* searching for point predictors (10.6.6);
* determining transform coefficient weights (10.6.11) or point quantization weights (10.6.13); and
* reconstructing attribute values from coded coefficients (10.6.7).

The reconstructed values are stored in the array PointAttr.

### State variables

Levels of detail are specified in terms of the following state variables; the index lvl identifies a detail level:

* The variable LodCnt, a count of detail levels generated from the slice geometry.
* The array LodPtCnt, the size of each detail level; LodPtCnt[ lvl ] is the number of points in the identified detail level.
* The array LodPtIdx, identifying points in each detail level by their index in the canonical decoding order; LodPtIdx[ lvl ][ 𝑖 ] is the AttrPos index of the 𝑖-th point in the identified detail level.
* The array LodRfmtPtCnt, the size of each detail level's refinement list; LodRfmtPtCnt[ lvl ] is the number of points in the refinement list for the identified detail level.
* The array LodRfmtPtIdx, identifying points in each refinement list by their index in the canonical decoding order; LodRfmtPtIdx[ lvl ][ 𝑖 ] is the AttrPos index of the 𝑖-th refinement point for the identified detail level.

Point predictors are specified in terms of the following state variables; the index ptIdx identifies a point by its index into AttrPos:

* The array PredCnt; PredCnt[ ptIdx ] is the size of the predictor set for the identified point.
* The array PredPtIdx, identifies point predictors by their index in the canonical decoding order; PredPtIdx[ ptIdx ][ ni ] is the AttrPos index or RefAttrPos index of the ni-th point in the predictor set for the identified point.
* The array PredPtRef of the flags to specify whether the point predictors are searched from the reference slice. When PredPtRef[ ptIdx ][ ni ] is equal to 1, the ni-th point in the predictor set is specified to be searched from the reference slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ]; When PredPtRef[ ptIdx ][ ni ] is equal to 0, the ni-th point in the predictor set is specified to be searched from the current slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ].
* The array PredWeight of point predictor weights; PredWeight[ ptIdx ][ ni ] is the prediction weight for the predictor identified by PredPtIdx[ ptIdx ][ ni ].
* The array CoeffWeight of transform coefficient weights; CoeffWeight[ ptIdx ] is the normalization weight for the transform coefficients associated with the identified point.
* The array QuantWeight of point quantization weights; QuantWeight[ ptIdx ] is the quantization weight for the transform coefficients associated with the identified point.

### Levels of detail

#### General generation process

The effect of this process is to represent the LoD structure in the state variables LodCnt, LodPtCnt, LodPtIdx, LodRfmtPtCnt and LodRfmtPtIdx.

The finest detail level shall contain the entire slice geometry (10.6.5.2). It is identified by the detail level index 0.

Detail levels shall be iteratively subsampled (10.6.5.4), starting from the finest detail level, until either a single point remains or lod\_max\_levels\_minus1 subsampled detail levels have been produced. The variable Lvl identifies the detail level to be subsampled.

Lvl = 0  
for (; Lvl < lod\_max\_levels\_minus1; Lvl++) {  
 if (LodPtCnt[Lvl] == 1)  
 break  
 … /\* subsample LodPtIdx[Lvl] \*/  
}  
LodCnt = Lvl + 1

The coarsest detail level is identified by the detail level index LodCnt − 1. All points in the coarsest detail level shall be assigned to the coarsest level's refinement list (10.6.5.3).

#### The finest detail level

The AttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the slice geometry.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 LodPtIdx[0][ptIdx] = ptIdx  
LodPtCnt[0] = PointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The variable maxPtsPerSort identifies the max group size when sorting by group.

maxPtsPerSort = !attr\_canonical\_order\_enabled && !max\_points\_per\_sort\_log2\_plus1 ?  
 LodPtCnt[0] : 1 << (max\_points\_per\_sort\_log2\_plus1 - 1)

The sorted order shall be identical for the decoding of all attributes in a single slice with identical attribute coordinate arrays (AttrPos).

1. Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < LodPtCnt[0]; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, LodPtCnt[0])  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = LodPtIdx[0][i]  
 jPtIdx = LodPtIdx[0][j]  
 iMorton = Morton(AttrPos[iPtIdx][0], AttrPos[iPtIdx][1], AttrPos[iPtIdx][2])  
 jMorton = Morton(AttrPos[jPtIdx][0], AttrPos[jPtIdx][1], AttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(LodPtIdx[0][i], LodPtIdx[0][j])  
 }  
}

#### The coarsest detail level

After generation of the LoD hierarchy, all points in the coarsest detail level shall be assigned to its refinement list.

for (i = 0; i < LodPtCnt[LodCnt − 1]; i++)  
 LodRfmtPtIdx[LodCnt − 1][i] = LodPtIdx[LodCnt − 1][i]

#### Generation of a single detail level

The coarser detail level Lvl + 1 shall be produced by subsampling the points of detail level Lvl.

The following definitions are used in the specification of the subsampling processes:

* The expression InLodPtCnt is an alias for LodPtCnt[ Lvl ], the number of points in the input detail level.
* The expression InLodPtIdx[ 𝑖 ] is an alias for LodPtIdx[ Lvl ][ 𝑖 ], the point indexes of the input detail level.
* The expression OutLodPtCnt is an alias for LodPtCnt[ Lvl + 1 ], the number of points in the output detail level.
* The expression OutLodPtIdx[ 𝑖 ] is an alias for LodPtIdx[ Lvl + 1 ][ 𝑖 ], the point indexes of the output detail level.
* The expression OutRfmtPtIdx[ 𝑖 ] is an alias for LodRfmtPtIdx[ Lvl ][ 𝑖 ], the point indexes of the refinement list for detail level Lvl.

Subsampling partitions points in the input detail level into an output detail level and the refinement list for the input detail level. The partitioning process shall preserve the relative ordering of points in the input detail level.

Subsampling shall proceed according to:

* block-based subsampling (10.6.5.8) if lod\_scalability\_enabled is 1, or lod\_decimation\_mode is 2;
* periodic subsampling (10.6.5.5) if lod\_decimation\_mode is 1; or
* distance-based subsampling (10.6.5.6) otherwise.

#### Periodic subsampling

When aps\_extension\_present is 1, geom\_tree\_type is 1, and attr\_canonical\_order\_enabled is 0, the variable CanonicalLodSubsampling is set to 1. Otherwise, CanonicalLodSubsampling is set to 0.

Periodic subsampling generates a subsampled output detail level by sampling every one-in-sampling-period points in the input detail level.

If CanonicalLodSubsampling is equal to 0, periodic subsampling generates a subsampled output detail level by:

The sampling period for the current detail level is subsamplingPeriod.

samplingPeriod := 2 + lod\_sampling\_period\_minus2[Lvl]

Input points shall be assigned to either the output detail level or the refinement list according to their index in the input detail level modulo the sampling period:

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 if (i % samplingPeriod)  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 else  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
}

If CanonicalLodSubsampling is equal to 1, canonical periodic subsampling generates a subsampled output detail level by:

The sampling period for the current detail level is subsamplingPeriod.

samplingPeriod := (Lv1==0 ? 1 : canonical\_samplingPeriod[Lvl-1]) \* (2 +  
 lod\_sampling\_period\_minus2[Lvl])

Input points shall be assigned to either the output detail level or the refinement list according to their index in the input detail level modulo the sampling period:

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 if (InLodPtIdx[i] % samplingPeriod)  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 else  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
}  
canonical\_samplingPeriod[Lvl] = samplingPeriod

#### Distance-based subsampling

Distance-based subsampling generates a subsampled output detail level by:

* spatially partitioning the input detail level into a lattice of sized cubic blocks; and
* assigning at most one point from each block to the subsampled detail level.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The subsampling process is specified in terms of the following state variables; the indexes bs, bt and bv identify the block location ( bs, bt, bv ):

* The sparse array MapSub; MapSub[ bs ][ bt ][ bv ] equal to 1 indicates that the identified block contains a single point previously assigned to the subsampled detail level. Unset elements of MapSub are inferred to be 0.
* The sparse array MapPtIdx, identifies points assigned to the subsampled detail level. When MapSub[ bs ][ bt ][ bv ] is 1, MapPtIdx[ bs ][ bt ][ bv ] is the AttrPos index of the point assigned to the subsampled detail level.

The points in the input detail level shall be processed sequentially. For each input point:

* The variable PtIdx is the AttrPos index of the point.
* The block location ( Bs, Bt, Bv ) is determined from the point's attribute coordinates.
* Depending upon the result of a per-point test (10.6.5.7), the point shall be assigned to either the output detail level or the refinement list. The result of the test is the variable IsSubsampledPoint.

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 PtIdx = InLodPtIdx[i]  
 Bs = AttrPos[PtIdx][0] >> BlkSizeLog2  
 Bt = AttrPos[PtIdx][1] >> BlkSizeLog2  
 Bv = AttrPos[PtIdx][2] >> BlkSizeLog2  
  
 … /\* IsSubsampledPoint = result of per−point test (10.6.5.7) \*/  
  
 if (MapSub[Bs][Bt][Bv] || ¬IsSubsampledPoint)  
 OutRfmtPtIdx[outRfmtPtCnt++] = PtIdx  
 else {  
 OutLodPtIdx[OutPtCnt++] = PtIdx  
 MapSub[Bs][Bt][Bv] = 1   
 MapPtIdx[Bs][Bt][Bv] = PtIdx  
 }  
}

#### Per-point decision for distance-based subsampling

The derivation of IsSubsampledPoint specifies whether the point shall be assigned to the output detail level.

A point shall be assigned to the output detail level unless the squared distance between it and any previously assigned point from a set of adjacent blocks within an availability window is less than or equal to sqRadius. Each availability window shall be a 128×128×128 block volume identified by ( Bs >> 7, Bt >> 7, Bv >> 7 ).

sqRadius = 3 << 2 × (BlkSizeLog2 − 1)

图示, 工程绘图

描述已自动生成

Figure 31 — Example decisions using distance-based subsampling.

Example per-point decisions are illustrated in Figure 31. Subsampling generates three detail levels. The points assigned to by subsampling (when Lvl = 0) are not within the shaded radius of any other point in . The block size used to subsample is BlkSizeLog2 = 1. All points are within a single availability window.

The array neighPtIdx is a neighCnt-element list of AttrPos indexes of points present in the adjacent blocks of the output detail level that are within the availability window. Table 52 specifies the relative locations of the adjacent blocks.

neighCnt = 0  
for (i = 0; i < 19; i++) {  
 ns = Bs + adjBlkOffset[i][0]  
 nt = Bt + adjBlkOffset[i][1]  
 nv = Bv + adjBlkOffset[i][2]  
 unavailable = (ns ^ Bs) >> 7 || (nt ^ Bt) >> 7 || (nv ^ Bv) >> 7  
 if (unavailable)  
 continue  
 if (MapSub[ns][nt][nv])  
 neighPtIdx[neighCnt++] = MapPtIdx[ns][nt][nv]  
}

Table 52— Adjacent block coordinates, adjBlkOffset[ 𝑖 ][ 𝑘 ], relative to ( Bs, Bt, Bv )

| 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| **0** | −1 | 0 | 0 | **5** | −1 | 1 | 0 | **10** | −1 | 0 | −1 | **15** | 1 | −1 | −1 |
| **1** | 0 | −1 | 0 | **6** | 0 | 1 | −1 | **11** | −1 | −1 | 0 | **16** | −1 | 1 | −1 |
| **2** | 0 | 0 | −1 | **7** | 1 | 0 | −1 | **12** | −1 | 1 | 1 | **17** | −1 | −1 | 1 |
| **3** | 0 | −1 | 1 | **8** | 1 | −1 | 0 | **13** | 1 | −1 | 1 | **18** | −1 | −1 | −1 |
| **4** | −1 | 0 | 1 | **9** | 0 | −1 | −1 | **14** | 1 | 1 | −1 |  | | | |

The point's attribute coordinates shall be compared to those of each point identified by the neighPtIdx array to determine the value of IsSubsampledPoint.

IsSubsampledPoint = 1  
for (i = 0; i < neighCnt; i++) {  
 sqDist = 0  
 for (k = 0; k < 3; k++) {  
 d = AttrPos[neighPtIdx[i]][k] − AttrPos[PtIdx][k]  
 sqDist += d × d  
 }  
 if (sqDist ≤ sqRadius)  
 IsSubsampledPoint = 0  
}

#### Block-based subsampling

Block-based subsampling generates a subsampled output detail level by:

* spatially partitioning the input detail level into a lattice of sized cubic blocks;
* grouping together blocks, in Morton order, according to the number of points they contain; and
* assigning one point from each block group to the subsampled detail level.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

1. Under certain conditions, blocks correspond to nodes of the occupancy tree. For instance, when lod\_scalability\_enabled is 1.

A list of block groups shall be generated by traversing the input detail level in canonical order. Consecutive blocks shall be grouped together until the group spans at least minGrpPts points.

minGrpPts := lod\_scalability\_enabled ? 0 : 2 + lod\_sampling\_period\_minus2[Lod]

The array grpBdry, with elements grpBdry[ grpIdx ], identifies block group boundaries as indexes into the input detail level array InLodPtIdx.

for (i = 1, grpStart = 0; i < InLodPtCnt; i++) {  
 ptIdx = InLodPtIdx[i]  
 ptIdxPrev = InLodPtIdx[i − 1]  
 bdryS = (AttrPos[ptIdx][0] ^ AttrPos[ptIdxPrev][0]) >> BlkSizeLog2  
 bdryT = (AttrPos[ptIdx][1] ^ AttrPos[ptIdxPrev][1]) >> BlkSizeLog2  
 bdryV = (AttrPos[ptIdx][2] ^ AttrPos[ptIdxPrev][2]) >> BlkSizeLog2  
 if (bdryS | bdryT | bdryV)  
 if (i − grpStart ≥ minGrpPts)  
 grpBdry[grpCnt++] = grpStart = i  
}  
grpBdry[grpCnt++] = InLodPtCnt

For each group of blocks, a test (10.6.5.9) shall be performed to determine the index of the point to be assigned to the output detail level. All other points shall be assigned to the refinement list. The variables GrpStart and GrpEnd identify the start and end of a block group. The result of the test is the variable IdxOfSubsampledPoint.

OutLodPtCntSize = outRfmtPtCnt = 0  
for (GrpStart = grpIdx = 0; grpIdx < grpCnt; GrpStart = grpBdry[grpIdx++]) {  
 GrpEnd = grpBdry[grpIdx]  
  
 … /\* IdxOfSubsampledPoint = result of per−point test (10.6.5.9) \*/  
  
 for (i = GrpStart; i < GrpEnd; i++) {  
 if (IdxOfSubsampledPoint == i)  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
 else  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 }  
}

#### Per block-group decision for block-based subsampling

The derivation of IdxOfSubsampledPoint specifies the input detail level index of the point in the block group that shall be assigned to the output detail level.

The distance to the block group centroid shall be used to select the point assigned to the output detail level. The block group centroid and point distances shall be calculated using attribute coordinates quantized by Exp2( BlkSizeLog2 − 1 ). The distance metric shall be the Manhattan distance.

图示, 工程绘图

描述已自动生成

Figure 32 — Example decisions using block group subsampling with minGrpPts = 3.

Example per-point decisions are illustrated in Figure 32. Subsampling generates two detail levels. To subsample , the points are grouped into block groups containing a minimum of three points. In this example, the block size for (Lvl = 0) is BlkSizeLog2 = 1. The first block group (solid shading) comprises four points from three blocks each with one, one and two points, respectively. The first point that is the closest to the centroid of the points in the block group is assigned to .

The block group centroid shall be the sum of all quantized attribute coordinates, centroidSum, divided by the number of points in the block group, numPtsInGrp.

numPtsInGrp := GrpEnd − GrpStart  
for (k = 0; k < 3; k++)  
 centroidSum[k] = 0  
for (i = 0; i < numPtsInGrp; i++) {  
 ptIdx = InLodPtIdx[GrpStart + i]  
 for (k = 0; k < 3; k++)  
 centroidSum[k] += AttrPos[ptIdx][k] >> BlkSizeLog2 − 1  
}

The array ptDist maps the index of each point in the block group to the distance between it and the centroid.

for (i = 0; i < numPtsInGrp; i++) {  
 ptIdx = InLodPtIdx[GrpStart + i]  
 ptDist[i] = 0  
 for (k = 0; k < 3; k++) {  
 posk = AttrPos[ptIdx][k] >> BlkSizeLog2 − 1  
 ptDist[i] += Abs(posk × numPtsInGrp − centroidSum[k])  
 }  
}

The point closest to the block group centroid shall be assigned to the output detail level. In the case that the block group contains multiple closest points, the selected point is the closest point with:

* when lod\_scalability\_enabled is 1 and Lvl is odd: the greatest InLodPtIdx index;
* when lod\_scalability\_enabled is 0 or Lvl is even: the lowest InLodPtIdx index.

last := lod\_scalability\_enabled ? Lvl & 1 : 1  
minIdx = 0  
for (i = 1; i < numPtsInGrp; i++)  
 if (last ? dist[i] ≤ dist[minIdx] : dist[i] < dist[minIdx])  
 minIdx = i  
  
IdxOfSubsampledPoint = GrpStart + minIdx

#### The finest detail level of the reference slice

When slice\_attr\_inter\_prediction is equal to 1, the finest detail level of the reference slice is specified in terms of the following state variables:

* The variable refLodPtCnt, the size of the finest detail level.
* The array RefLodPtIdx, identifying points in the finest detail level by their index in the canonical decoding order; RefLodPtIdx [ 𝑖 ] is the RefAttrPos index of the 𝑖-th point in the finest detail level.

The RefAttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the reference slice geometry.

for (ptIdx = 0; ptIdx < refPointCnt; ptIdx++)  
 LodPtIdx[ptIdx] = ptIdx  
refLodPtCnt = refPointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The sorted order shall be identical for the decoding of all attributes in a single slice with identical attribute coordinate arrays (RefAttrPos).

1. Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < RefLodPtCnt; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, RefLodPtCnt)  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = RefLodPtIdx[i]  
 jPtIdx = RefLodPtIdx[j]  
 iMorton = Morton(RefAttrPos[iPtIdx][0], RefAttrPos[iPtIdx][1],  
 RefAttrPos[iPtIdx][2])  
 jMorton = Morton(RefAttrPos[jPtIdx][0], RefAttrPos[jPtIdx][1],  
 RefAttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(RefLodPtIdx[i], RefLodPtIdx[j])  
 }  
}

### Predictor search

#### General process

The points used to predict the refinement points of each detail level shall be determined by a search (10.6.6.3).

The effect of this process is to represent the point predictors in the state variables PredCnt, PredPtIdx, PredPtRef and PredWeight.

When attr\_coding\_type is 2, no searches shall be performed for the refinement points of the coarsest detail level.

maxLvl = LodCnt − (attr\_coding\_type == 2)  
for (Lvl = 0; Lvl < maxLvl; Lvl++)  
 for (RfmtIdx = 0; RfmtIdx < LodRfmtPtCnt[Lvl]; RfmtIdx++) {  
 … /\* find predictors (10.6.6.2) of the current point \*/  
 }

#### Minimum reference detail level for inter-level predictor searches

The variable MinInterRefLvl identifies the finest detail level that shall be used as a reference for inter-detail level prediction. When lod\_scalability\_enabled is 1, it shall be the finest detail level with fewer refinement points than the total number of refinement points associated with all finer detail levels.

MinInterRefLvl = 1  
if (lod\_scalability\_enabled) {  
 for (lvl = 1; lvl < LodCnt − 1; lvl++) {  
 if (LodRfmtPtCnt[lvl] < slice\_num\_points\_minus1 − LodPtCnt[lvl])  
 break  
 MinInterRefLvl++  
 }  
}

#### Predictor search for a single refinement point

For a refinement point with index RfmtIdx in detail level Lvl, a search shall be performed to find the closest neighbouring points from a set of candidate neighbours.

The search process is specified in terms of the following variables:

* The variable PtIdx, the AttrPos index of the refinement point.
* The variable RefLvl, the reference detail level used for inter-level predictor searches.

PtIdx = LodRfmtPtIdx[RfmtIdx]  
RefLvl = Max(Lvl + 1, MinInterRefLvl)

An inter-detail-level search shall be performed prior to any intra-level search. Except for the coarsest detail level, the following inter-level searches shall be performed:

* An initial search (10.6.6.6).
* If fewer than three predictors are found (PredCnt[ PtIdx ] < 3), an extended search (10.6.6.7).

When Lvl is greater than or equal to pred\_intra\_min\_lod, an intra-detail-level search (10.6.6.8) shall be performed.

When slice\_attr\_inter\_prediction is equal to 1, an initial inter-frame search (10.6.6.9) and an extended inter-frame search (10.6.6.10) shall be performed.

After completing the searches, weights shall be calculated for each predictor (10.6.6.11), during which the predictor set is pruned and re-sorted. When pred\_blending\_enabled is 1, predictor weights shall be blended (10.6.6.12).

图示

中度可信度描述已自动生成

Figure 33 — Example of searches performed for a single refinement point.

An example predictor search when slice\_attr\_inter\_prediction is equal to 0 is illustrated in Figure 33. Searches are performed for the refinement point 𝛾 in of ; other points in are denoted 𝛼, 𝛽 and 𝛿. Points in the next coarsest detail level, , are marked a to f. After the initial inter-level search, the predictor set is { c, b }. Then, since fewer than three predictors were found, an extended inter-level search is performed over ±pred\_inter\_lod\_search\_range points in the point list. This search adds predictor d to the predictor set { c, b, d }. Finally, an intra- level search is performed over pred\_intra\_lod\_search\_range previous points in . The final predictor set for is { c, 𝛽, 𝛼 }.

#### Inclusion of a candidate point in the predictor set (InsertPredictor)

This subclause defines the function InsertPredictor( candPtIdx  , *candRef*) that conditionally inserts a candidate point into the predictor set of the current refinement point. Each candidate shall be tested against the refinement point's predictor set to determine if and where it is to be inserted.

The parameter candPtIdx is the AttrPos index or the *RefAttrPos* index of the candidate point. The parameter candRef specifies whether the candidate point is searched from the reference slice.

A candidate shall only be inserted into the point's neighbour set once. If candPresent is 1, the candidate is not inserted into the predictor set.

candPresent = 0  
for (i = 0; i < PredCnt[PtIdx]; i++)  
 candPresent |= PredPtIdx[PtIdx][i] == candPtIdx  
 && (slice\_attr\_inter\_prediction  
 ? PredPtRef[PtIdx][i] == candRef  
 : 1)

Otherwise (the candidate is not already present), the following shall be used to decide the inclusion in the predictor set.

* the spatial distance between the candidate and the refinement point.
* the relative spatial location of the candidate to the current point.

The distance shall be calculated as the biased norm weighted by PredBias.

dist = BiasedNorm1(PtIdx, candPtIdx, 0, candRef)

The point shall be inserted into the predictor set, with elements ordered according to the biased distance to the refinement point. When the size of predictor set is less than 6 and prediction\_with\_distribution\_enabled is 1, the point is inserted only when the distance between the point and the current point is equal to the distance between the third predictor in the predictor set and the current point. Points at the same distance shall be ordered by insertion order, with earlier members being ordered before later members.

for (i = 0; i < PredCnt[PtIdx]; i++)  
 if (dist < BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i], 0, PredPtRef[PtIdx][i]))  
 break  
 if(i < 3 || ((dist == BiasedNorm1(PtIdx, PredPtIdx[PtIdx][2])) && PredCnt[PtIdx] < 6)  
 for (j = PredCnt[PtIdx]; j > i; j−−){  
 PredPtIdx[PtIdx][j] = PredPtIdx[PtIdx][j − 1]  
 if (slice\_attr\_inter\_prediction)  
 PredPtRef[PtIdx][j] = PredPtRef[PtIdx][j - 1]  
 }  
 PredPtIdx[PtIdx][i] = candPtIdx  
 PredPtRef[PtIdx][i] = slice\_attr\_inter\_prediction ? candRef : 0

The size of the predictor set shall be limited to three elements (when prediction\_with\_distribution\_enabled is 0) or six elements (when prediction\_with\_distribution\_enabled is 1) by discarding the furthest predictor if necessary.

PredCnt[PtIdx] = Min(prediction\_with\_distribution\_enabled ? 6 : 3, PredCnt[PtIdx] + 1)

#### Distance computation using the biased L1 norm (BiasedNorm1)

This subclause defines the function BiasedNorm1( ptIdxA, ptIdxB, ptA, ptB) that is the weighted Manhattan distance between two points.

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression BiasedNorm1. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance: when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level.

BiasedNorm1(ptIdxA, ptIdxB, pcA, pcB) := dist[0] + dist[1] + dist[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k]) × PredBias[k]  
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

#### Initial inter-level predictor search

The initial inter-level search shall be performed by spatially partitioning the reference detail level into a lattice of sized cubic blocks. Only blocks adjacent to the block containing the refinement point that are within an availability window shall be searched.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The block location ( bs, bt, bv ) identifies the block containing the refinement point.

bs := AttrPos[PtIdx][0] >> BlkSizeLog2  
bt := AttrPos[PtIdx][1] >> BlkSizeLog2  
bv := AttrPos[PtIdx][2] >> BlkSizeLog2

The availability window shall be a 128×128×128 block volume identified by ( bs >> 7, bt >> 7, bv >> 7 ).

The search shall proceed over the search blocks in the order specified by Table 53. Within each search block, points shall be searched in ascending order of index within the reference detail level.

for (si = 0; si < 27; si++) {  
 ss = bs + searchBlkOffsets[si][0]  
 st = bt + searchBlkOffsets[si][1]  
 sv = bv + searchBlkOffsets[si][2]  
 unavailable = (ss ^ bs) >> 7 || (st ^ bt) >> 7 || (sv ^ bv) >> 7  
 if (unavailable)  
 continue  
 for (i = 0; i < LodPtCnt[RefLvl]; i++) {  
 candPtIdx = LodPtIdx[RefLvl][i]  
 cs = AttrPos[candPtIdx][0]  
 ct = AttrPos[candPtIdx][1]  
 cv = AttrPos[candPtIdx][2]  
 inSblk = cs ≥ (ss << BlkSizeLog2) && cs < (ss + 1 << BlkSizeLog2)  
 inSblk &= ct ≥ (st << BlkSizeLog2) && ct < (st + 1 << BlkSizeLog2)  
 inSblk &= cv ≥ (sv << BlkSizeLog2) && cv < (sv + 1 << BlkSizeLog2)  
 if (inSblk)  
 InsertPredictor(candPtIdx, 0)  
 }  
}

1. For each search block, the indices 𝑖 for which inSblk is true are consecutive.

Table 53 — Search block coordinates, searchBlkOffsets[ 𝑖 ][ 𝑘 ], relative to ( bs, bt, bv )

| 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| **0** | 0 | 0 | 0 | **7** | 0 | 1 | 1 | **14** | 1 | 0 | −1 | **21** | 1 | −1 | 1 |
| **1** | −1 | 0 | 0 | **8** | 1 | 0 | 1 | **15** | 1 | −1 | 0 | **22** | 1 | 1 | −1 |
| **2** | 0 | −1 | 0 | **9** | 1 | 1 | 0 | **16** | 0 | −1 | −1 | **23** | 1 | −1 | −1 |
| **3** | 0 | 0 | −1 | **10** | 0 | −1 | 1 | **17** | −1 | 0 | −1 | **24** | −1 | 1 | −1 |
| **4** | 1 | 0 | 0 | **11** | −1 | 0 | 1 | **18** | −1 | −1 | 0 | **25** | −1 | −1 | 1 |
| **5** | 0 | 1 | 0 | **12** | −1 | 1 | 0 | **19** | 1 | 1 | 1 | **26** | −1 | −1 | −1 |
| **6** | 0 | 0 | 1 | **13** | 0 | 1 | −1 | **20** | −1 | 1 | 1 |  | | | |

#### Extended inter-level search

The extended inter-level search evaluates predictor candidates over a span of indexes in the reference detail level.

The span shall be centred around the index centre in the reference detail level. It shall be the index of:

* if at least one predictor has been found for the current point: the first predictor; or

if (PredCnt[PtIdx])  
 for (centre = 0; centre < LodPtCnt[RefLvl] − 1; centre++)  
 if (LodPtIdx[RefLvl][centre] == PredPtIdx[PtIdx][0])  
 break

* otherwise (no predictors have been found): the first point with Morton-coded attribute coordinates greater than those of the current point.

if (PredCnt[PtIdx] == 0) {  
 mortonCurPt = Morton[AttrPos[PtIdx]]  
 for (centre = 0; centre < LodPtCnt[RefLvl] − 1; centre++) {  
 mortonCentre = Morton[AttrPos[LodPtIdx[RefLvl][centre]]]  
 if (mortonCurPt < mortonCentre)  
 break  
 }  
}

The search range for extended inter-level search is specified by the variable *interLodSearchRange*.

if (slice\_attr\_inter\_prediction)  
 interLodSearchRange = attr\_inter\_prediction\_search\_range  
else  
 interLodSearchRange = pred\_inter\_lod\_search\_range

The extended search shall proceed over each index offset 𝑖 of the following, in order: 0, +1, −1, +2, −2, +3 .. *interLodSearchRange*, and −( 3 .. *interLodSearchRange* ).

A predictor candidate shall be evaluated for each valid search index centre + 𝑖 that is within the range specified by *interLodSearchRange* and does not exceed the bounds of the reference detail level.

if (Abs(i) ≤ interLodSearchRange)  
 if (centre + i ≥ 0 && centre + i < LodPtCnt[RefLvl])  
 InsertPredictor(LodPtIdx[RefLvl][centre + i], 0)

#### Intra-level search

The intra-level search evaluates predictor candidates over a span of indexes in the refinement list of the current detail level. Intra-level predictor candidates shall precede the refinement point in the refinement list.

The search range for intra-level search is specified by the variable *intraLodSearchRange*.

if (slice\_attr\_inter\_prediction)  
 intraLodSearchRange = attr\_inter\_prediction\_search\_range  
else  
 intraLodSearchRange = pred\_intra\_lod\_search\_range

A predictor candidate shall be evaluated for each valid search index offset − 𝑖 from the refinement point, for 𝑖 = 1 .. *intraLodSearchRange*, that does not exceed the bounds of the refinement list.

for (i = 1; i ≤ Min(RfmtIdx, intraLodSearchRange); i++)  
 InsertPredictor(LodRfmtPtIdx[Lvl][RfmtIdx − i], 0)

#### Initial inter-frame predictor search

The initial inter-frame search shall be performed by spatially portioning the reference slice into a lattice of sized cubic blocks. Only blocks adjacent to the block containing the refinement point that are within an availability window shall be searched.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The block location ( *bs*, *bt*, *bv* ) identifies the block containing the refinement point.

bs := AttrPos[PtIdx][0] >> BlkSizeLog2  
bt := AttrPos[PtIdx][1] >> BlkSizeLog2  
bv := AttrPos[PtIdx][2] >> BlkSizeLog2

The availability window shall be an 8×8×8 block volume identified by ( *bs* >> 3, *bt* >> 3, *bv* >> 3 ).

The search shall proceed over the search blocks in the order specified by Table 53. Within each search block, points shall be searched in ascending order of index within the reference slice.

for (si = 0; si < 27; si++) {  
 ss = bs + searchBlkOffsets[si][0]  
 st = bt + searchBlkOffsets[si][1]  
 sv = bv + searchBlkOffsets[si][2]  
 unavailable = (ss ^ bs) >> 3 || (st ^ bt) >> 3 || (sv ^ bv) >> 3  
 if (unavailable)  
 continue  
 for (i = 0; i < RefLodPtCnt; i++) {  
 candPtIdx = RefLodPtIdx[i]  
 cs = RefAttrPos [candPtIdx][0]  
 ct = RefAttrPos [candPtIdx][1]  
 cv = RefAttrPos [candPtIdx][2]  
 inSblk = cs ≥ (ss << BlkSizeLog2) && cs < (ss + 1 << BlkSizeLog2)  
 inSblk &= ct ≥ (st << BlkSizeLog2) && ct < (st + 1 << BlkSizeLog2)  
 inSblk &= cv ≥ (sv << BlkSizeLog2) && cv < (sv + 1 << BlkSizeLog2)  
 if (inSblk)  
 InsertPredictor(candPtIdx, 1)  
 }  
}

1. For each search block, the indices 𝑖 for which inSblk is true are consecutive.

#### Extended inter-frame search

The inter-frame search evaluates predictor candidates over a span of indexes in the finest detail level of the reference slice.

The span shall be centred around the index centreRef in the finest detail level of the reference slice. It shall be the index of the first point with Morton-coded attribute coordinates greater than or equal to those of the current point.

mortonCurPt = Morton[AttrPos[PtIdx]]  
for (centreRef = 0; centreRef < RefLodPtCnt − 1; centreRef++) {  
 mortonCentre = Morton[RefAttrPos[RefLodPtIdx[centreRef]]]  
 if (mortonCurPt <= mortonCentre)  
 break  
}

The search range for intra-level search is specified by the variable *interFrameSearchRange*.

interFrameSearchRange := (slice\_biprediction && slice\_attr\_inter\_prediction &&  
 slice\_attr\_inter\_prediction2) ? attr\_inter\_prediction\_search\_range >> 1 :  
 attr\_inter\_prediction\_search\_range

The inter-frame search shall proceed over each index offset 𝑖 of the following, in order: 0, +1, −1, +2, −2, +3 .. *interFrameSearchRange*, and −( 3 .. *interFrameSearchRange* ).

A predictor candidate shall be evaluated for each valid search index centreRef + 𝑖 that is within the range specified by *interFrameSearchRange* and does not exceed the bounds of the finest detail level of the reference slice.

if (Abs(i) ≤ interFrameSearchRange)  
 if (centreRef + i ≥ 0 && centreRef + i < RefLodPtCnt)  
 InsertPredictor(RefLodPtIdx[centreRef + i], 1)

#### Predictor set pruning and generation of prediction weights

After the predictor search for a refinement point is complete, its predictor set shall be pruned, weights computed for each qualifying predictor and the predictors ordered according to weight.

The size of the predictor set shall be limited to pred\_set\_size\_minus1 + 1 elements by discarding the furthest predictors if necessary.

PredCnt[PtIdx] = Min(pred\_set\_size\_minus1 + 1, PredCnt[PtIdx])

When cross\_attr\_prediction\_enabled\_this\_type is 0, predictor weights shall be calculated using the biased squared distance between each predictor and the current point.

for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 dist[ni] = BiasedNorm2(PtIdx, PredPtIdx[PtIdx][ni], 0, PredPtRef[PtIdx][ni])  
 + PredPtRef[PtIdx][ni] ? 1 : 0

If the first predictor is spatially coincident with the current point, all other predictors shall be discarded.

if (dist[0] == 0)  
 PredCnt[PtIdx] = 1

When lod\_scalability\_enabled is 1, predictors with an unbiased squared distance greater than a threshold shall be discarded.

if (lod\_scalability\_enabled) {  
 threshold = 3 × (pred\_max\_range\_minus1 + 1) << 2 × Lvl  
 for (ni = 1; ni < PredCnt[PtIdx]; ni++)  
 if (Norm2(ptIdx, PredPtIdx[PtIdx][ni], 0, 0) > threshold) {  
 PredCnt[PtIdx] = ni  
 break  
 }  
}

When cross\_attr\_prediction\_enabled\_this\_type is 1, predictor weights shall be calculated using the overall distance between each predictor and the current point

for (ni = 0; ni < PredCnt[PtIdx]; ni++  
 dist[ni] = overAllDist[ni]

The overall distance is a weighted combination of the geometry distance and the attribute distance and shall be stored in array overAllDist[ni], ni = 0 .. PredCnt[ PtIdx ] − 1. The geometry distance is defined as the spatial distance which shall be calculated as the biased norm weighted by PredBias. The attribute distance is defined as the sum of the absolute difference in attribute value for each component.

overAllDist[ni] = geomDis + attrWeight \* attrDis  
 where  
 geomDis = BiasedNorm1(PtIdx, PredPtIdx[PtIdx][ni], 0, 0)  
 attrDis = 0  
 for(i = 0; i < attr\_components\_minus1[refAttrIdx] + 1; i++)  
 attrDis += Abs(RecCloudAttr[ ptIdx ][ refAttrIdx][i] -  
 RecCloudAttr[PredPtIdx[PtIdx][ni]][ refAttrIdx][i])

The attribute weight *attrWeight* which is defined by the weight of attribute distance shall be determined as:

attrWeight = lambda \* maxGeom / maxAttr  
 where  
 lambda = attr\_label[refAttrIdx] ? (4011 – 67 \* (attr\_primary\_qp\_minus4 + 4)) >> 15 :  
 (1939 – 33 \* (attr\_primary\_qp\_minus4 + 4)) >> 12  
 for (k = 0; k < 3; j++)  
 maxGeom += RefCloudAttrBboxSize[k]  
 maxAttr = (attr\_components\_minus1[refAttrIdx] + 1) \*  
 1 << (attr\_bitdepth\_minus1[refAttrIdx] + 1)

where *maxGeom* represents the summation of the length, width and height of the slice bounding box and *maxAttr* represents the possible maximum value of the encoded attribute that is used for decoding the current attribute.

If the first predictor is same as the current point, all other predictors shall be discarded.

if (dist[0] == 0)  
 PredCnt[PtIdx] = 1

When cross\_attr\_prediction\_enabled\_this\_type is 1, the predictors shall be reordered in ascending order according to their overall distance to the current point; when cross\_attr\_prediction\_enabled\_this\_type is 0, the predictors shall be reordered according to their biased squared distance to the current point:

* An array order shall have elements such that dist[ order[ 𝑖 ] ], for 𝑖 = 0 .. PredCnt[ PtIdx ] − 1, is an ascending stable sorting of the array dist.
* The members of the predictor set and the dist array shall be permuted according to the elements of the array order.

The predictor distances shall be normalized by the smallest distance to produce initial weights.

n = Max(0, IntLog2(dist[0] − 8))  
for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 weight[ni] = DivExp2Up(dist[ni], n)

Any predictors with a weight 256 times greater than or equal to the smallest weight shall be discarded.

if (PredCnt[PtIdx] == 3 && weight[2] ≥ 256 × weight[0])  
 PredCnt[PtIdx] = 2

if (PredCnt[PtIdx] == 2 && weight[1] ≥ 256 × weight[0])  
 PredCnt[PtIdx] = 1

The final weights shall be derived as:

if (PredCnt[PtIdx] == 1)  
 PredWeight[PtIdx][0] = 256

if (PredCnt[PtIdx] == 2) {  
 PredWeight[PtIdx][1] = Div(weight[0], weight[0] + weight[1], 8)  
 PredWeight[PtIdx][0] = 256 − PredWeight[PtIdx][1]  
}

if (PredCnt[PtIdx] == 3) {  
 d1d2 = weight[1] × weight[2]  
 d0d2 = weight[0] × weight[2]  
 d0d1 = weight[0] × weight[1]  
 sum = d1d2 + d0d2 + d0d1  
 PredWeight[PtIdx][2] = Div(d0d1, sum, 8)  
 PredWeight[PtIdx][1] = Div(d0d2, sum, 8)  
 PredWeight[PtIdx][0] = 256 − PredWeight[PtIdx][1] − PredWeight[PtIdx][2]  
}

#### Blending of predictor weights

When a point has three predictors in its predictor set and pred\_blending\_enabled is 1, the predictor weights shall be blended according to the distance between the predicting points.

The squared distance between each of the three predictors shall be determined:

distA := Norm2(PredPtIdx[PtIdx][0], PredPtIdx[PtIdx][1], PredPtRef[PtIdx][0],  
 PredPtRef[PtIdx][1])  
distB := Norm2(PredPtIdx[PtIdx][0], PredPtIdx[PtIdx][2], PredPtRef[PtIdx][0],  
 PredPtRef[PtIdx][2])  
distC := Norm2(PredPtIdx[PtIdx][1], PredPtIdx[PtIdx][2], PredPtRef[PtIdx][1],  
 PredPtRef[PtIdx][2])

Blending weights shall be selected according to distance:

b1 := distA ≤ distB ? 1 : 5  
b2 := distA ≤ distC ? 5 : 1  
b3 := distB ≤ distC ? 1 : 5

The predictor weights shall be blended and updated:

if (PredCnt[PtIdx] == 3 && pred\_blending\_enabled) {  
 w0 = PredWeight[PtIdx][0]  
 w1 = PredWeight[PtIdx][1]  
 w2 = PredWeight[PtIdx][2]  
 w0p = w0 × 10 + w1 × (6 − b2) + w2 × b3 >> 4  
 w1p = w0 × b1 + w2 × (6 − b3) + w1 × 10 >> 4  
 PredWeight[PtIdx][0] = w0p  
 PredWeight[PtIdx][1] = w1p  
 PredWeight[PtIdx][2] = 256 − w0p − w1p  
}

#### Distance computation using the biased L2 norm (BiasedNorm2)

This subclause defines the function BiasedNorm2( ptIdxA, ptIdxB  , *pcA*, *pcB*) that is the weighted squared distance between two points.

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression BiasedNorm2. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance: when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level.

BiasedNorm2(ptIdxA, ptIdxB, pcA, pcB) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k]) × PredBias[k]  
 dist2[k] := dist[k] × dist[k]  
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

#### Distance computation using the unbiased L2 norm (Norm2)

This subclause defines the function Norm2( ptIdxA, ptIdxB , *pcA*, *pcB* ) that is the squared distance between two points.

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression Norm2. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance. when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level.

Norm2(ptIdxA, ptIdxB, pcA, pcB) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k])  
 dist2[k] := dist[k] × dist[k]  
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

#### Reduction of predictor set

Subclause 10.6.6.15 applies when prediction\_with\_distribution\_enabledis 1.

When the size of the predictor set is larger than 3, the third point in the predictor set may be replaced by one of the following points in the predictor set according to their relative spatial locations.

The array PtDirection represents the relative direction of a point in the predictor set with respect to the current point.

The space around the current point is separated into eight octants indexed from 0 to 7 along the three axes using the current point as the origin. The relative direction of a point in the predictor set with respect to the current point is defined as the index of the octant in which the point is located.

for (j = 0; j < PredCnt[PtIdx];j++)  
 for (k = 0; k < 3; ++k) {  
 dist = ((AttrPos[PtIdx][k] - AttrPos[PredPtIdx[PtIdx][j]][k])  
 PtDirection[j] |= (dist >= 0) << k  
 }

Strict opposite octants and loose opposite octants are defined as strictOpposite and looseOpposite, respectively, where strictOpposite[ 𝑖 ][ 𝑘 ] (or looseOpposite[ 𝑖 ][ 𝑘 ]) denote whether the k-th octant is a strict (or loose) opposite octant of the i-th octant (Table 54 and Table 55).

Table 54— Strictly opposite octants, strictOpposite[ 𝑖 ][ 𝑘 ]

| 𝑖 | 𝑘 | | | | | | | | 𝑖 | 𝑘 | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **4** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | **5** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **6** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **7** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 55— Loose opposite octants, looseOpposite[ 𝑖 ][ 𝑘 ]

| 𝑖 | 𝑘 | | | | | | | | 𝑖 | 𝑘 | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **0** | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | **4** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| **1** | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | **5** | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **2** | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | **6** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| **3** | 1 | 0 | 0 | 0 |  | 1 | 1 | 0 | **7** | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

The first three points in the predictor set are referred to as initial predictor points.

The variables equal01, equal02 and equal12 indicate whether the pairs of initial predictor points are in the same octant.

equal01 = PtDirection[0] == PtDirection[1]  
equal02 = PtDirection[0] == PtDirection[2]  
equal12 = PtDirection[1] == PtDirection[2]

The predictor set needs no replacement if any two points of the initial predictor points are in strict opposite octants of each other.

Otherwise, the third point in the predictor set may be replaced by a candidate predictor point. A distance threshold dist1 is derived below that is used in the determination of replacement of the third point.

dist1 = (BiasedNorm1(PtIdx, PredPtIdx[PtIdx][2]) × (pred\_replace\_weight\_minus32 + 32))  
 >> 5

The points in the predictor set that have index value 3 or more, and whose distance to the current point is less than dist1 are referred as candidate predictor points of the predictor set.If a candidate predictor point is at the strict opposite direction of the first or the second initial predictor point, that point is used to replace the third point in the predictor set.

for (i = 3; i < PredCnt[PtIdx]; i++){  
 if (dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i])   
 && (strictOpposite[i][0] || strictOpposite[i][1])){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
}

If the third point in the predictor set is not replaced by the above process, and if the all the initial predictor points are in the same octant p, the third point in the predictor set is be replaced by a candidate point that is in loose opposite direction of octant p as below.

if(equal01 && (equal02 || equal12))  
 for (i = 3; i < PredCnt[PtIdx]; i++){  
 if (dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i]) && looseOpposite[i][0]){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
 }

If the third point of the predictor set is not replaced by the above process, the replacement process continues. If the first and second initial predictors are not in the same octant and not loose opposite octants, and if the third initial predictor is in the same octant as the first or the second predictor, a candidate predictor point that is not in the same octant as the first or the second initial predictor, that candidate predictor point is used to replace the third point in the predictor set.

if(¬equal01 && (equal02 || equal12) && ¬looseOpposite[0][1])  
 for (i = 3; i < PredCnt[PtIdx]; i++){  
 if(dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i])   
 && PtDirection[i] ≠ PtDirection[0]  
 && PtDirection[i] ≠ PtDirection[1]){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
 }

If the third point in the predictor set is not replaced by the above process, the replacement process continues. If the first and second initial predictors are in the same octant and the third initial predictor is not in loose opposite octant as the first predictor, a candidate predictor point that is at the loose opposition direction to the first initial predictor is used to replace the third point in the predictor set.

if(equal01 && ¬looseOpposite[0][2])  
 for (i = 3; i < PredCnt[PtIdx] ; i++){  
 if(dist2 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i] && looseOpposite[i][0]){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
 }

After the above processes, the size of the predictor set shall be reduced to three elements by discarding the furthest predictor.

PredCnt[PtIdx] = Min(3, PredCnt[PtIdx] + 1)

### Reconstruction of attribute values

#### General process

Each detail level shall be processed in turn, proceeding from the coarsest to the finest level, according to attr\_coding\_type (10.6.7.3, 10.6.7.4). The variable Lvl is the index of the current detail level.

for (Lvl = LodCnt − 1; Lvl ≥ 0; Lvl−−)  
 … /\* process a detail level \*/

#### Coefficient processing order within a detail level

Within a detail level, processing proceeds in coded coefficient order. The variable PtIdx is the AttrPos index of the current coefficient. The variable CoeffIdx is the AttrCoeff array index of the current coefficient.

for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[Lvl]; rfmtIdx++) {  
 PtIdx = LodRfmtPtIdx[Lvl][rfmtIdx]  
 CoeffIdx = LodPtCnt[Lvl] − rfmtIdx  
 … /\* process current coefficient \*/  
}

#### Processing a detail level (attr\_coding\_type = 1)

When attr\_coding\_type is 1, the following operations shall be performed in turn for each coefficient in the coefficient processing order of the current detail level:

* Prediction mode information is decoded from an encoded coefficient tuple (10.6.8.1). The result is the variable PredMode.
* The unencoded coefficient tuple is scaled (10.6.9.1) to produce transform coefficients.
* Transform coefficient components are divided by 256 with half-values rounded up.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = DivExp2Up(PointAttr[PtIdx][c], 8)

* Transform coefficient are scaled by the point quantization weights (10.6.13.3).
* Transform coefficient components are predicted using inter-component prediction (10.6.10.2) to form prediction residuals.
* The attribute value is predicted and combined with the prediction residual (10.6.12).
* The reconstructed attribute value is clipped.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = Clip3(0, AttrMaxVal, PointAttr[PtIdx][c])

#### Processing a detail level (attr\_coding\_type = 2)

When attr\_coding\_type is 2, the following operations shall be performed in turn, each over all the coefficients in the detail level:

* Coefficient tuples are scaled (10.6.9.1) to produce transform coefficients.
* Transform coefficient components are predicted using last-component prediction (10.6.10.3).
* Transform coefficients are weighted by transform coefficient weights (10.6.11.4).

If Lvl is less than LodCnt − 1, the transform shall be applied (10.6.12):

* Attribute values predicted from the coarser detail level, Lvl + 1, are modified by the transform update operator (10.6.12.1).
* Attribute values corresponding to coefficients in the current detail level are predicted and combined with the scaled transform coefficient to produce the detail level output (10.6.12.1).

When Lvl is 0, the reconstructed attributes values shall be divided by 256 with half-values rounded away from zero and clipped to the maximum attribute value:

if (Lvl == 0)  
 for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[ptIdx][c] = Clip3(0, AttrMaxVal, DivExp2Fz(PointAttr[ptIdx][c], 8))

### Prediction mode coding

#### General

Subclause 10.6.8 specifies the conditional coding of the prediction mode PredMode in coefficient tuples. It applies when pred\_direct\_max\_idx\_plus1 is greater than zero; when pred\_direct\_max\_idx\_plus1 is 0, PredMode shall be 0.

A per-transform-coefficient test (10.6.8.2) shall be performed to determine whether the coefficient tuple encodes a prediction mode. The result of the test is the variable PredModePresent.

If PredModePresent is:

* false, PredMode shall be 0;
* true, the coded prediction mode (PredModeCoded) shall be decoded according to the number of attribute components (10.6.8.3, 10.6.8.4) and the maximum codable prediction mode PredModeMax. As a side-effect of decoding the prediction mode, the coefficient tuple (in AttrCoeff) is updated.

PredModeMax := pred\_direct\_max\_idx\_plus1 + ¬pred\_direct\_avg\_disabled

The prediction mode shall be derived from the coded prediction mode:

PredMode = PredModePresent ? PredModeCoded + pred\_direct\_avg\_disabled : 0

#### Presence of an encoded direct prediction mode

The derivation of PredModePresent specifies the presence of an encoded direct prediction mode:

* A direct prediction mode shall not be coded when disabled, or for refinement points with fewer than two predictors.

if (PredCnt[PtIdx] < 2 || pred\_direct\_max\_idx\_plus1 == 0)  
 PredModePresent = 0

* Otherwise, a prediction mode shall be coded for a refinement point if, for any component, the absolute difference in attribute value between any of its predictors exceeds a bit-depth adjusted threshold.

for (ni = 0; ni < PredCnt[PtIdx]; ni++) {  
 ptIdx = PredPtIdx[PtIdx][ni]  
 for (c = 0; c < AttrDim; c++) {  
 minVal[c] = ni ? Min(minVal[c], PointAttr[ptIdx][c]) : PointAttr[ptIdx][c]  
 maxVal[c] = ni ? Max(maxVal[c], PointAttr[ptIdx][c]) : PointAttr[ptIdx][c]  
 }  
}

maxDiff = 0  
for (c = 0; c < AttrDim; c++)  
 maxDiff = Max(maxDiff, maxVal[c] − minVal[c])

threshold = pred\_direct\_threshold << Max(0, AttrBitDepth − 8)  
PredModePresent = maxDiff ≥ threshold

#### Decoding process for single component attributes

For single component attributes (AttrDim == 1), the prediction mode PredModeCoded is encoded by the LSBs of the coefficient magnitude:

PredModeCoded = 0  
absCoeff = Abs(AttrCoeff[CoeffIdx][0])

if (PredModeMax == 4){  
 PredModeCoded = absCoeff & 3  
 absCoeff >>= 2  
}

if (PredModeMax == 3) {  
 PredModeCoded = absCoeff & 1  
 absCoeff >>= 1  
 if (PredModeCoded){  
 PredModeCoded += absCoeff & 1  
 absCoeff >>= 1  
 }  
}

if (PredModeMax == 2){  
 PredModeCoded = absCoeff & 1  
 absCoeff >>= 1  
}

After decoding the prediction mode, the coefficients shall be updated.

AttrCoeff[CoeffIdx][0] = Sign(AttrCoeff[CoeffIdx][0]) × absCoeff

#### Decoding process for multi-component attributes

For multi-component attributes (AttrDim > 1), the prediction mode PredModeCoded is encoded by the LSB of the last two component's coefficient magnitude:

PredModeCoded = 0  
absCoeffA = Abs(AttrCoeff[CoeffIdx][AttrDim − 2])  
absCoeffB = Abs(AttrCoeff[CoeffIdx][AttrDim − 1])

if (PredModeMax == 4) {  
 PredModeCoded = ((absCoeffA & 1) << 1) + (absCoeffB & 1)  
 absCoeffA >>= 1  
 absCoeffB >>= 1  
}

if (PredModeMax == 3) {  
 PredModeCoded = absCoeffA & 1  
 absCoeffA >>= 1  
 if (PredModeCoded) {  
 PredModeCoded += absCoeffB & 1  
 absCoeffB >>= 1  
 }

if (PredModeMax == 2) {  
 PredModeCoded = absCoeffA & 1  
 absCoeffA >>= 1  
}

After decoding the prediction mode, the coefficients shall be updated.

sgnCoeffA = Sign(AttrCoeff[CoeffIdx][AttrDim − 2])  
sgnCoeffB = Sign(AttrCoeff[CoeffIdx][AttrDim − 1])

AttrCoeff[CoeffIdx][AttrDim − 2] = sgnCoeffA × absCoeffA  
AttrCoeff[CoeffIdx][AttrDim − 1] = sgnCoeffB × absCoeffB

### Scaling

#### Derivation of per-point QP

The QP for a point depends upon the detail level and its attribute coordinates as specified by the expression LodCoeffQp[ qc ] for a QP component qc:

* rgnOffset[ qc ] is the per-coefficient offset from the region-dependent QP offset tree.
* dpth is the depth of detail level in the LoD hierarchy.

LodCoeffQp[qc] := AttrQp[dpth][rgnOffset][Cidx > 0]  
 where  
 s := AttrPos[PtIdx][0]  
 t := AttrPos[PtIdx][1]  
 v := AttrPos[PtIdx][2]  
 dpth := LodCnt − 1 − Lvl  
 rgnOffset[qc] := AttrRegionQpOffset[s][t][v][qc]

#### Scaling by quantization step size

The coefficient tuple shall be scaled by the quantization step size (10.7.4) for the primary and secondary attribute components.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = AttrCoeff[CoeffIdx][c] × AttrQstep[LodCoeffQp[c > 0]]

### Coefficient prediction

#### Syntax element semantics

last\_comp\_pred\_coeff\_diff[ dpth ] specifies in accordance with LastCompPredCoeff[ dpth ] the two-fractional-bit, fixed-point scale factor applied at depth dpth of the LoD hierarchy to second coefficient components to predict third coefficient components. The syntax element codes the scale factor relative to LastCompPredCoeffPrev[ dpth ].

LastCompPredCoeff[dpth] := last\_comp\_pred\_enabled  
 ? LastCompPredCoeffPrev[dpth] + last\_comp\_pred\_coeff\_diff[dpth]  
 : 0

LastCompPredCoeffPrev[dpth] := dpth == 0 ? 4 : LastCompPredCoeff[dpth − 1]

It is a requirement of bitstream conformance that LastCompPredCoeff[ dpth ] shall be in the range −128 .. 127 for dpth ∈ 0 .. lod\_max\_levels\_minus1.

inter\_comp\_pred\_coeff\_diff[ dpth ][ 𝑐 ] specifies in accordance with InterCompPredCoeff[ dpth ][ 𝑐 ] the two-fractional-bit, fixed-point scale factor applied at depth dpth of the LoD hierarchy to first coefficient components to predict 𝑐-th coefficient components. The syntax element codes the scale factor relative to InterCompPredCoeffPrev[ dpth ][ 𝑐 ].

InterCompPredCoeff[dpth][c] := inter\_comp\_pred\_enabled  
 ? predCoeff + inter\_comp\_pred\_coeff\_diff[dpth][c]  
 : 0

InterCompPredCoeffPrev[dpth][c] := dpth == 0 ? 4 : InterCompPredCoeff[dpth − 1][c]

It is a requirement of bitstream conformance that InterCompPredCoeff[ dpth ][ 𝑐 ] shall be in the range −128 .. 127 for dpth ∈ 0 .. lod\_max\_levels\_minus1.

#### Inter-component prediction

When attr\_coding\_type is 1 and inter\_comp\_pred\_enabled is 1, secondary attribute coefficient components are residuals to a prediction by the first scaled coefficient component. The predicted value shall round the two fractional bits from the scale factor, with half-values rounded up.

for (c = 1; c < AttrDim; c++) {  
 icpCoeff = InterCompPredCoeff[LodCnt − 1 − Lvl][c]  
 PointAttr[PtIdx][c] += DivExp2Up(icpCoeff × PointAttr[PtIdx][0], 2)  
}

#### Last component prediction

When attr\_coding\_type is 2 and last\_comp\_pred\_enabled is 1, the third attribute coefficient component is, if present, a residual to a prediction by the second scaled coefficient component. The predicted value shall round the two fractional bits from the scale factor towards negative infinity.

if (AttrDim == 3) {  
 lcpCoeff = LastCompPredCoeff[LodCnt − 1 − Lvl]  
 PointAttr[PtIdx][2] += DivExp2Floor(lcpCoeff × PointAttr[PtIdx][1], 2)  
}

### Transform coefficient weights

#### General

Coefficient weights represent the relative significance of a coefficient. Coefficients with larger weights have a greater influence on the decoded attribute values.

The array CoeffWeight is initialized by setting all elements to 256.

for (i = 0; i < PointCnt; i++)  
 CoeffWeight[i] = 256

The derivation of coefficient weights depends upon whether LoD scalability is enabled.

#### Non-scalable case

When lod\_scalability\_enabled is 0, coefficient weights are calculated accumulatively, proceeding from the finest to the coarsest detail level.

The accumulated coefficient weight of each refinement point in a detail level shall be distributed to the points in its predictor set. The distribution is proportional to the respective predictor weights:

for (lvl = 0; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = CoeffWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 predW = PredWeight[ptIdx][ni]  
 CoeffWeight[PredPtIdx[ptIdx][ni]] += DivExp2Up(coeffW × predW, 8)  
 }  
 }  
 }

#### Scalable case

When lod\_scalability\_enabled is 1, a single weight shall be assigned to all refinement points within a detail level:

for (lvl = 1; lvl < LodCnt − 1; lvl++) {  
 weight = (slice\_num\_points\_minus1 + 1) / LodPtCnt[lvl]  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++)  
 CoeffWeight[LodRfmtPtIdx[lvl][rfmtIdx]] = weight × 256  
}

#### Application to coefficient scaling

Transform coefficients shall be scaled by the integer reciprocal square root of their coefficient weight and divided by with half-values rounded away from zero.

weight = IntRecipSqrt(CoeffWeight[PtIdx])  
for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = DivExp2Fz(PointAttr[PtIdx][c] × weight, 36)

### Transform

#### Update operation

When attr\_coding\_type is 2, the transform update operator shall redistribute coefficient values to predicting points in the coarser detail level.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 updateN[ptIdx] = updateD[ptIdx] = 0

for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[Lvl]; rfmtIdx++) {  
 rfmtPtIdx = LodRfmtPtIdx[Lvl][rfmtIdx]  
 coeffW = CoeffWeight[rfmtPtIdx]  
 for (ni = 0; ni < PredCnt[rfmtPtIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 nPtIdx = PredPtIdx[rfmtPtIdx][ni]  
 nWeight = DivExp2Up(PredWeight[rfmtPtIdx][ni] × coeffW, 8)  
 updateD[nPtIdx] += nWeight  
 for (c = 0; c < AttrDim; c++)  
 updateN[nPtIdx][c] += nWeight × PointAttr[rfmtPtIdx][c]  
 }  
 }  
}

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 if (updateD[ptIdx])  
 PointAttr[ptIdx] −= Div(updateN[ptIdx], updateD[ptIdx], 0)

#### Direct prediction

When attr\_coding\_type is 1 and PredMode for a refinement point is greater than zero, its value shall be predicted to be the same as the point with predictor set index PredMode − 1. If the indicated predictor is invalid, prediction shall not be performed.

It is a requirement of bitstream conformance that PredMode shall be less than or equal to PredCnt[ CoeffIdx ].

if (PredMode && PredMode ≤ PredCnt[PtIdx])  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] += PredPtRef[PtIdx][PredMode - 1] ?   
 RefPointAttr[PredPtIdx[PtIdx][PredMode − 1]][c] :   
 PointAttr[PredPtIdx[PtIdx][PredMode − 1]][c]

#### Average prediction

When attr\_coding\_type is 2 or PredMode for a refinement point is 0, the weighted average of the predictor set shall predict the value of the refinement point:

if (PredMode == 0)  
 for (c = 0; c < AttrDim; c++) {  
 sum = 0  
 for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 sum += PredWeight[PtIdx][ni] × PredPtRef[PtIdx][ni] ?   
 RefPointAttr[PredPtIdx[PtIdx][ni]][c] :   
 PointAttr[PredPtIdx[PtIdx][ni]][c]  
 PointAttr[PtIdx][c] += DivExp2Fz(sum, 8)  
 }

### Point quantization weights

#### General

Point quantization weights represent the predictive relationships between points. Points with larger quantization weights have a greater influence on the prediction process.

The array QuantWeight is initialized by setting all elements to 256.

for (i = 0; i < PointCnt; i++)  
 QuantWeight[i] = 256

#### Quantization weights derivation

Quantization weights are calculated accumulatively, proceeding from the finest to the coarsest detail level. The quantization weight of each point is updated by the points in its predictor set. The weights of the points in the predictor set to update the quantization weights are specified by quant\_neigh\_weight:

for (lvl = 0; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = QuantWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 QuantWeight[PredPtIdx[ptIdx][ni]] +=  
 DivExp2Inf(coeffW × quant\_neigh\_weight[ni], 8)  
 }  
 }  
 }

#### Application to coefficient scaling

Transform coefficients shall be scaled by quantization step size (10.7.4) or quantization weights.

weight = Min(QuantWeight[PtIdx], AttrQstep[qp]) >> 8  
for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] /= weight

## Attribute quantization parameters

### Syntax element semantics

attr\_qp\_offset[ qc ] specifies per-slice offsets used to derive QPs for the primary (qc = 0) and any secondary (qc = 1) attribute components. When attr\_qp\_offset[ qc ] is not present, it shall be inferred to be 0.

attr\_qp\_layers\_present specifies whether (when 1) or not (when 0) per-transform-layer QP offsets are present in the ADU.

attr\_qp\_layer\_cnt\_minus1 plus 1 specifies, when present, the number of levels in the LoD hierarchy or RAHT tree for which QP offsets are signalled.

attr\_qp\_layer\_offset[ dpth ][ qc ] specifies QP offsets used for the primary (qc = 0) and any secondary (qc = 1) attribute components. Each offset applies to transform coefficients at depth dpth of the LoD hierarchy or RAHT tree. If the LoD hierarchy or RAHT tree has a greater number of levels than attr\_qp\_layer\_cnt\_minus1 + 1, attr\_qp\_layer\_offset[ attr\_qp\_layer\_cnt\_minus1 ][ qc ] also specifies the QP offsets for transform coefficients at a depth greater than attr\_qp\_layer\_cnt\_minus1.

The expression AttrQpLayerOffset[ dpth ][ qc ] specifies the per layer QP offsets at depth dpth of the LoD hierarchy or RAHT tree.

AttrQpLayerOffset[dpth][qc] := attr\_qp\_layers\_present > 0  
 ? attr\_qp\_layer\_offset[Min(attr\_qp\_layer\_cnt\_minus1, dpth)][qc]  
 : 0

attr\_qp\_region\_cnt specifies the number of spatial regions within the slice that have a region QP offset signalled.

1. In profiles specified in this version of this document, all but the first region are ignored.

attr\_qp\_region\_bits\_minus1 plus 1 specifies the length in bits of each syntax element attr\_qp\_region\_origin\_xyz, attr\_qp\_region\_size\_minus1\_xyz, attr\_qp\_region\_origin\_rpi and attr\_qp\_region\_size\_minus1\_rpi.

attr\_qp\_region\_origin\_xyz[ 𝑖 ][ 𝑘 ] and attr\_qp\_region\_size\_minus1\_xyz[ 𝑖 ][ 𝑘 ] specify, when present, the 𝑖-th spatial region in the slice where attr\_qp\_region\_offset[ 𝑖 ][ qc ] applies. The region is a bounding box in the slice coordinate system with lower corner XYZ coordinates attr\_qp\_region\_origin\_xyz[ 𝑖 ][ 𝑘 ] and dimensions attr\_qp\_region\_size\_minus1\_xyz[ 𝑖 ][ 𝑘 ] + 1.

attr\_qp\_region\_origin\_rpi[ 𝑖 ][ 𝑘 ] and attr\_qp\_region\_size\_minus1\_rpi[ 𝑖 ][ 𝑘 ] specify, when present, the 𝑖-th spatial region in the slice where attr\_qp\_region\_offset[ 𝑖 ][ qc ] applies. The region is a bounding box in the scaled angular coordinate system (10.2.2) used for attribute coding with lower corner RPI coordinates attr\_qp\_region\_origin\_rpi[ 𝑖 ][ 𝑘 ] and dimensions attr\_qp\_region\_size\_minus1\_rpi[ 𝑖 ][ 𝑘 ] + 1.

The expressions AttrRegionQpOrigin[ 𝑖 ][ 𝑘 ] and AttrRegionQpSize[ 𝑖 ][ 𝑘 ] specify the 𝑘-th component of the bounding box origin and size for the 𝑖-th QP region in attribute coordinates.

AttrRegionQpOrigin[i][k] = attr\_coord\_conv\_enabled  
 ? attr\_qp\_region\_origin\_rpi[i][k]  
 : attr\_qp\_region\_origin\_xyz[i][StvToXyz[k]]

AttrRegionQpSize[i][k] = attr\_coord\_conv\_enabled  
 ? attr\_qp\_region\_size\_minus1\_rpi[i][k] + 1  
 : attr\_qp\_region\_size\_minus1\_xyz[i][StvToXyz[k]] + 1

A constraint on the bounds of a region is specified by expression AttrRegionSizeConstraint[ 𝑖 ][ 𝑘 ]. It is a requirement of bitstream conformance that AttrRegionSizeConstraint[ 𝑖 ][ 𝑘 ] shall be true for every component 𝑘 of each region 𝑖.

AttrRegionSizeConstraint[i][k] :=  
 AttrRegionQpOrigin[i][k] + AttrRegionQpSize[i][k] < Exp2(MaxSliceDimLog2)

attr\_qp\_region\_offset[ 𝑖 ][ qc ] specifies offsets used to derive the QPs for the primary (qc = 0) and any secondary (qc = 1) attribute components of points positioned within the region defined by AttrRegionQpOrigin[ 𝑖 ] and AttrRegionQpSize[ 𝑖 ]. When attr\_qp\_region\_offset[ 𝑖 ][ qc ] is not present, it shall be inferred to be 0.

attr\_AC\_qp\_offset\_present specifies whether (when 1) or not (when 0) per-RAHT-layer AC transform coefficient QP offsets are present in the ADU. When attr\_AC\_qp\_offset\_present is not present, it shall be inferred to be 0.

attr\_AC\_qp\_layer\_cnt\_minus1 plus 1 specifies, when present, the number of levels in the RAHT tree for which AC QP offsets are present in the ADU. When attr\_AC\_qp\_layer\_cnt\_minus1 is not present, it shall be inferred to be -1.

attr\_AC\_qp\_offset[ dpth ][ qc][ACcompidx] specifies the per AC coefficient (ACcompidx = 0,1,..6) QP offsets for the primary (qc = 0) and any secondary (qc = 1) attribute components at depth dpth of the RAHT tree.

### Per-point regional QP offset

The region-dependent QP offset for a point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ) is specified by the expression AttrRegionQpOffset[ 𝑠 ][ 𝑡 ][ 𝑣 ][ qc ].

AttrRegionQpOffset[s][t][v][qc] :=  
 isPointInRegion[0][s][t][v] ? attr\_qp\_region\_offset[0][qc] : 0

The expression isPointInRegion[ rgnIdx ][ 𝑠 ][ 𝑡 ][ 𝑣 ] specifies whether the coordinates ( 𝑠, 𝑡, 𝑣 ) are within the rgnIdx-th region.

isPointInRegion[rgnIdx][s][t][v] :=  
 rIdx < attr\_qp\_region\_cnt  
 && s ≥ AttrRegionQpOrigin[rIdx][0] && s < regionEnd[0]  
 && t ≥ AttrRegionQpOrigin[rIdx][1] && t < regionEnd[1]  
 && v ≥ AttrRegionQpOrigin[rIdx][2] && v < regionEnd[2]  
 where  
 regionEnd[k] := AttrRegionQpOrigin[rIdx][k] + AttrRegionQpSize[rIdx][k]

### Attribute coefficient QP

An attribute coefficient QP is specified by the expression AttrQp[ dpth ][ rgnOffset ][ qc ], parameterized by:

* dpth, the depth of a level in the LoD hierarchy or RAHT tree;
* rgnOffset, an expression that when applied to an argument qc is a region-dependent QP offset; and
* qc, indicating a primary or secondary QP component.

The expressions qpP and qpS are the QPs for the primary and secondary attribute components. Attribute QPs shall be clipped to the bit-depth dependent range [ 4, qpMax ].

AttrQp[dpth][rgnOffset][qc] := qc == 0 ? qpP : qpS  
 where  
 qpMax := 51 + 6 × (AttrBitDepth − 8)  
 qpP := Clip3(4, qpMax, attr\_primary\_qp\_minus4 + 4 + qpOffset[qc])  
 qpS := Clip3(4, qpMax, qpP + attr\_secondary\_qp\_offset + qpOffset[qc])  
 qpOffset[qc] := attr\_qp\_offset[qc] + AttrQpLayerOffset[dpth][qc] + rgnOffset[qc]

### Definition of AttrQstep

This subclause specifies the expression AttrQstep[ qp ] that is the attribute scale factor for the attribute quantization parameter qp.

AttrQstep[qp] := AttrLevelScale[qp % 6] << (qp / 6)

Values of AttrLevelScale are specified by Table 56.

Table 56 — Values of AttrLevelScale[ 𝑖 ]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 |
| AttrLevelScale[ 𝑖 ] | 161 | 181 | 203 | 228 | 256 | 287 |

## Reference slice generation for attribute

### General

Subclause 10.8 applies when attr\_inter\_prediction\_enabled is equal to 1.

After the decoding of a frame, subclause 10.8.3 applies to generate the reference frame for the attribute coding of the next point cloud frame to be decoded.

After the generation of point coordinates of the current slice as specified by subclause 10.2, subclause 10.8.4 applies to generate the reference slice.

The output of this process are the reference slice attribute and the reference slice geometry for attribute coding. The expression *RefPointAttr*[*ptIdx*][*c*] specifies the attribute values of each point in the reference slice. The expression *RefAttrPos*[*ptIdx*][*k*] specifies the coordinates of each point for attribute coding in the reference slice.

### Reconstructed frame generation

The coordinates for attribute coding of the previously reconstructed point cloud frame are stored in the array *CloudAttrPos*[*ptIdx*][*k*]. *AttrPos*[*ptIdx*][*k*] is an alias into the geometry coordinates array for attribute coding for the points in the slice.

AttrPos[ptIdx][k] :=   
CloudAttrPos[RecCloudPointCnt + ptIdx][k]

The coordinates of points for attribute coding shall be generated as specified by subclause 10.2.

### Reference frame generation

The geometry coordinates for attribute coding of the reference frame are stored in the array *RefCloudAttrPos*[*ptIdx*][*k*]. The attributes of the reference frame are stored in the array *RefCloudAttr*[*ptIdx*][*k*]. The number of points in the reference frame is specified by the variable *refCloudPointCnt*.

* When biprediction\_enabled is 0, subclause 10.8.3.1 applies to generate the reference frame for uni-prediction.
* Otherwise, subclause 10.8.3.2 applies to generate the reference frame for bi-prediction.

The expression RefCloudAttrBboxOrigin[ k ] is an alias for the 𝑘-th XYZ coordinate of the origin of the attribute reference frame *RefCloudAttrPos*. RefCloudAttrBboxSize[ k ] is an alias for the 𝑘-th XYZ component of the volume dimensions of the attribute reference frame *RefCloudAttrPos*.

#### Reference frame for uni-prediction

When geom\_tree\_type is 0, global\_motion\_enabled is equal to 0 or attr\_coord\_conv\_enabled is equal to 1, *RefCloudAttrPos* is set equal to *CloudAttrPos*, and *RefCloudAttr* is set equal to *RecCloudAttr*. When geom\_tree\_type is 0, global\_motion\_enabled is equal to 1 and attr\_coord\_conv\_enabled is equal to 0, *RefCloudAttrPos* is set equal to *GMPointCloud* (*GMPointCloud* is derived by invoking subclause 9.2.15.2) and *RefCloudAttr* is set equal to *RecCloudAttr*.

When geom\_tree\_type is 1 and attr\_coord\_conv\_enabled is equal to 1, *RefCloudAttrPos* and *RecCloudAttr* are derived as follows:

ptIdx = 0  
for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 for(j = 0; j <= maxPointsPerEntryMinus1; j++)  
 if (PtnCurrFramePos[beamId][qAzim][j][0] ¬= -1) {  
 for(k = 0; k < 3; k ++)  
 RefCloudAttrPos[ptIdx][k] = PtnCurrFramePos[beamId][qAzim][j][k]  
 RefCloudAttr[ptIdx][0] = PtnCurrFramePos[beamId][qAzim][j][0]  
 ptIdx++  
 }  
refCloudPointCnt = ptIdx

#### Reference frame for bi-prediction

When biprediction\_enabled is 1 or 2, the geometry coordinates for attribute coding *CloudAttrPos* and the reconstructed attributes *RecCloudAttr* are indicated by the notional frame counter to be used as the reference frame for subsequent point cloud frames.

The geometry coordinates for attribute coding of the first original reference frame are stored in the array *RefCloudAttrPosFirst*[*ptIdx*][*k*]. The attributes of the first original reference original frame are stored in the array *RefCloudAttrFirst*[*ptIdx*][*k*]. The number of points in the first original reference frame is specifies by the variable *refCloudPointCntFirst*. The geometry coordinates for attribute coding of the second original reference frame are stored in the array *RefCloudAttrPosSecond*[*ptIdx*][*k*]. The attributes of the second original reference frame are stored in the array *RefCloudAttrSecond*[*ptIdx*][*k*]. The number of points in the second original reference frame is specifies by the variable *refCloudPointCntSecond*.

The two original reference frames are determined as follows:

* when slice\_biprediction is 0, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 1 and slice\_biprediction is 1, *RefCloudAttrPosSecond* and *RefCloudAttrSecond* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame.
  + If the previously reconstructed point cloud frame is a B-frame, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed B-frame;
  + otherwise, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame.
* when biprediction\_enabled is 2 and slice\_biprediction is 1, *RefCloudAttrPosFirst*, *RefCloudAttrFirst*, *RefCloudAttrPosSecond* and *RefCloudAttrSecond* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the reference frames determined by invoking subclause 9.2.15.3.

*RefCloudAttrPos* and *RefCloudAttr* are determined as follows:

* when slice\_biprediction is 1, slice\_attr\_inter\_prediction2 is 1 and slice\_attr\_inter\_prediction is 0, *RefCloudAttrPos* is set equal to *RefCloudAttrPosSecond*, and *RefCloudAttr* is set equal to *RefCloudAttrSecond*;
* when slice\_biprediction is 1, slice\_attr\_inter\_prediction2 is 1 and slice\_attr\_inter\_prediction is 1, the reference frame is set equal to the fusion of the two original reference frames:

refCloudPointCnt = refCloudPointCntFirst + refCloudPointCntSecond  
for (i = 0; i < refCloudPointCntFirst; i++){  
 for (c = 0; c < AttrDim; c++)  
 RefCloudAttr[i][c] = RefCloudAttrFirst[i][c]  
 for (k = 0; k < 3; k++)  
 RefCloudAttrPos[i][k] = RefCloudAttrPosFirst[i][k]  
}  
for (i = 0; i < refCloudPointCntSecond; i++){  
 for (c = 0; c < AttrDim; c++)  
 RefCloudAttr[i + refCloudPointCntFirst][c] = RefCloudAttrSecond[i][c]  
 for (k = 0; k < 3; k++)  
 RefCloudAttrPos[i + refCloudPointCntFirst][k] = RefCloudAttrPosSecond[i][k]  
}

* otherwise, *RefCloudAttrPos* is set equal to *RefCloudAttrPosFirst*, and *RefCloudAttr* is set equal to *RefCloudAttrFirst*.

### Reference slice generation

The attribute values and coordinates of the reference slice are derived based on the bounding box of the coordinates of the current slice. The number of points in the reference slice is specified by the variable *refPointCnt*.

refPointCnt = 0  
for (i = 0; i < refCloudPointCnt; i++)  
 if (ptInBox[i]){  
 for (c = 0; c < AttrDim; c++)  
 RefPointAttr[refPointCnt][c] = RefCloudAttr[i][c]  
 for (k = 0; k < 3; k++)  
 RefAttrPos[refPointCnt][k] = RefCloudAttrPos[i][k]  
 refPointCnt++  
 }  
where,  
 ptInBox[i] = pos[0] < bMax[0] && pos[0] > bMin[0] && pos[1] < bMax[1]  
 && pos[1] > bMin[1] && pos[2] < bMax[2] && pos[2] > bMin[2]  
 where,  
 pos = RefCloudAttrPos[i]  
 bMax = RefCloudAttrBboxOrigin + RefCloudAttrBboxSize  
 bMin = RefCloudAttrBboxOrigin

# Parsing process

## General

Syntax elements are parsed according to the processes corresponding to the syntax element’s descriptor and name as specified in Table 57 to Table 59.

Table 57 — Descriptor parsing processes

| Descriptor | Parsing process | Arguments | Channel read method (readBit) |
| --- | --- | --- | --- |
| u(𝑛) | 11.4.1 | maxBins = 𝑛 | DuNextBit (11.2.5) |
| u(v) | 11.4.1 | See Table 58 | DuNextBit (11.2.5) |
| ue(v) | 11.4.3 | 𝑘 = 0 | DuNextBit (11.2.5) |
| s(𝑛) | 11.4.2 | maxBins = 𝑛 | DuNextBit (11.2.5) |
| s(v) | 11.4.2 | See Table 58 | DuNextBit (11.2.5) |
| se(v) | 11.4.3, 11.4.6 | 𝑘 = 0 | DuNextBit (11.2.5) |
| oid(v) | 11.4.7 |  | DuNextBit (11.2.5) |
| ae(v) | See Table 59 | See Table 59 | AeReadBin (11.5.2) |
| de(v) | 9.2.9 |  | na |

Table 58 — Syntax element specific parsing processes (non-ae(v))

| Syntax element | Parsing process | Arguments |
| --- | --- | --- |
| attr\_coord\_conv\_scale[ ] | 11.4.1 (FL) | numBins = attr\_coord\_conv\_scale\_bits\_minus1[ 𝑘 ] + 1 |
| attr\_default\_value[ attrIdx ][ ] | 11.4.1 (FL) | numBins = attr\_bitdepth\_minus1[ attrIdx ] + 1 |
| attr\_offset[ ] | 11.4.2 (FL+S) | numBins = attr\_offset\_bits |
| attr\_qp\_region\_origin\_rpi[ ] | 11.4.1 (FL) | numBins = attr\_qp\_region\_bits\_minus1 + 1 |
| attr\_qp\_region\_origin\_xyz[ ] | 11.4.1 (FL) | numBins = attr\_qp\_region\_bits\_minus1 + 1 |
| attr\_qp\_region\_size\_minus1\_rpi[ ] | 11.4.1 (FL) | numBins = attr\_qp\_region\_bits\_minus1 + 1 |
| attr\_qp\_region\_size\_minus1\_xyz[ ] | 11.4.1 (FL) | nmBins = attr\_qp\_region\_bits\_minus1 + 1 |
| attr\_scale\_minus1[ ] | 11.4.1 (FL) | numBins = attr\_scale\_bits |
| defattr\_value[ ] | 11.4.1 (FL) | numBins = AttrBitDepth |
| fbdu\_frame\_ctr\_lsb | 11.4.1 (FL) | numBins = fbdu\_frame\_ctr\_lsb\_bits |
| frame\_ctr\_lsb | 11.4.1 (FL) | numBins = frame\_ctr\_lsb\_bits |
| fsap\_frame\_ctr\_lsb | 11.4.1 (FL) | numBins = fsap\_frame\_ctr\_lsb\_bits |
| gps\_angular\_origin\_xyz[ ] | 11.4.1 (FL) | numBins = gps\_angular\_origin\_bits\_minus1 + 1 |
| raw\_attr\_value[ ][ ] | 11.4.1 (FL) | numBins = RawAttrValueBits |
| seq\_bbox\_size\_minus1\_xyz[ ] | 11.4.1 (FL) | numBins = seq\_bbox\_size\_bits |
| seq\_coded\_scale\_mantissa | 11.4.1 (FL) | numBins = seq\_coded\_scale\_mantissa\_bits |
| seq\_origin\_xyz[ ] | 11.4.2 (FL+S) | numBins = seq\_origin\_bits |
| slice\_angular\_origin\_xyz[ ] | 11.4.2 (FL+S) | numBins = slice\_angular\_origin\_bits\_minus1 + 1 |
| slice\_geom\_origin\_xyz[ ] | 11.4.1 (FL) | numBins = slice\_geom\_origin\_bits\_minus1 + 1 |
| slice\_tag | 11.4.1 (FL) | numBins = slice\_tag\_bits |
| ti\_frame\_ctr\_lsb | 11.4.1 (FL) | numBins = ti\_frame\_ctr\_lsb\_bits |
| ti\_origin\_xyz[ ] | 11.4.2 (FL+S) | numBins = ti\_origin\_bits\_minus1 + 1 |
| tile\_id | 11.4.1 (FL+S) | numBins = tile\_id\_bits |
| tile\_origin\_xyz[ ][ ] | 11.4.2 (FL) | numBins = tile\_origin\_bits\_minus1 + 1 |
| tile\_size\_minus1\_xyz[ ][ ] | 11.4.1 (FL) | numBins = tile\_size\_bits\_minus1 + 1 |
| subgroup\_bbox\_origin\_xyz[ ] | 11.4.2 (FL) | numBins = subgroup\_bbox\_origin\_\_bits\_minus1 + 1 |
| subgroup\_bbox\_size\_xyz[ ] | 11.4.1 (FL) | numBins = subgroup\_bbox\_size\_\_bits\_minus1 + 1 |

Table 59 — Syntax element specific parsing processes (ae(v))

| Syntax element | Parsing process | Arguments |
| --- | --- | --- |
| beam\_idx\_resid\_abs[ ] | 11.4.4 (TU+EGk) | maxOffset = 3, 𝑘 = 1 |
| beam\_idx\_resid\_sign[ ] | 11.4.1 (FL) | numBins = 1 |
| coeff\_abs | 11.4.4 (TU+EGk) | maxOffset = 2, 𝑘 = 1 |
| coeff\_sign | 11.4.1 (FL) | numBins = 1 |
| direct\_dup\_point\_cnt | 11.4.4 (TU+EGk) | maxOffset = 2, 𝑘 = 0 |
| direct\_joint\_diff\_bit[ ] | 11.4.1 (FL) | numBins = 1 |
| direct\_joint\_prefix[ ] | 9.2.12.6 |  |
| direct\_point\_cnt\_eq2 | 11.4.1 (FL) | numBins = 1 |
| direct\_rem\_st\_ang[ ] | 11.4.1 (FL) | numBins = DnRemAngBitsST (9.2.13.8.3) |
| direct\_rem\_v\_ang[ ] | 11.4.1 (FL) | numBins = DnRemAngBitsV (9.2.13.8.3) |
| direct\_rem[ 𝑘 ] | 11.4.1 (FL) | numBins = DnRemBits[ 𝑘 ] (9.2.12.4.5) |
| direct\_v\_ang\_resid\_abs[ ] | 11.4.4 (TU+EGk) | maxOffset = 3, 𝑘 = 1 |
| direct\_v\_ang\_resid\_sign[ ] | 11.4.1 (FL) | numBins = 1 |
| occ\_direct\_node | 11.4.1 (FL) | numBins = 1 |
| occ\_dup\_point\_cnt[ ] | 11.4.4 (TU+EGk) | maxOffset = 1, 𝑘 = 0 |
| occ\_histogram\_hit | 11.4.1 (FL) | numBins = 1 |
| occ\_histogram\_index | 11.4.1 (FL) | numBins = 5 |
| planar\_copy\_mode | 11.4.1 (FL) | numBins = 1 |
| occ\_plane\_pos[ ] | 11.4.1 (FL) | numBins = 1 |
| occ\_recent\_hit | 11.4.1 (FL) | numBins = 1 |
| occ\_recent\_index | 11.4.1 (FL) | numBins = 4 |
| occ\_single\_child | 11.4.1 (FL) | numBins = 1 |
| occ\_single\_plane[ ] | 11.4.1 (FL) | numBins = 1 |
| occ\_subtree\_qp\_offset\_abs | 11.4.4 (TU+EGk) | maxOffset = 1, 𝑘 = 0 |
| occ\_subtree\_qp\_offset\_present | 11.4.1 (FL) | numBins = 1 |
| occ\_subtree\_qp\_offset\_sign | 11.4.1 (FL) | numBins = 1 |
| occ\_symbol\_escape | 11.4.1 (FL) | numBins = 8 |
| occtree\_end\_of\_entropy\_stream | 11.4.1 (FL) | numBins = 1 |
| occupancy\_bit[ ] | 11.4.1 (FL) | numBins = 1 |
| occupancy\_byte | 9.2.9 |  |
| occupancy\_idx[ ] | 11.4.1 (FL) | numBins = 1 |
| gm\_comp\_partition\_block[] | 11.4.1 (FL) | numBins = 1 |
| ptn\_child\_cnt\_xor1[ ] | 11.4.5 (TU) | maxVal = 3 |
| ptn\_dup\_point\_cnt | 11.4.4 (TU+EGk) | maxOffset = 1, 𝑘 = 0 |
| ptn\_phi\_mul\_abs\_minus2 | 11.4.1 (FL) | numBins = 3 |
| ptn\_phi\_mul\_abs\_minus9 | 11.4.2 (EGk) | 𝑘 = 0 |
| ptn\_phi\_mul\_abs\_prefix | 11.4.5 (TU) | maxVal = 2 |
| ptn\_phi\_mul\_sign | 11.4.1 (FL) | numBins = 1 |
| ptn\_radius\_resid\_abs | 11.4.1 (TU+EGk) | maxOffset = 3, 𝑘 = 2 |
| ptn\_radius\_resid\_sign | 11.4.1 (FL) | numBins = 1 |
| ptn\_phi\_resid\_abs\_gt0 | 11.4.1 (FL) | numBins = 1 |
| ptn\_phi\_resid\_sign | 11.4.1 (FL) | numBins = 1 |
| ptn\_phi\_resid\_abs\_gt1 | 11.4.1 (FL) | numBins = 1 |
| ptn\_phi\_resid\_abs\_rem | 11.4.2 (EGk) | 𝑘 = 1 |
| ptn\_inter\_flag | 11.4.1 (FL) | numBins = 1 |
| ptn\_pred\_direction | 11.4.1 (FL) | numBins = 1 |
| ptn\_inter\_pred\_mode | 11.4.1 (FL) | numBins = global\_motion\_enabled ? 2: 1 |
| ptn\_pred\_mode[ ] | 11.4.1 (FL) | numBins = 2 |
| ptn\_pred\_idx[ ] | 11.4.1 (TU) | maxVal = ptree\_ang\_max\_pred\_index |
| ptn\_qp\_offset\_abs | 11.4.4 (TU+EGk) | maxOffset = 1, 𝑘 = 0 |
| ptn\_qp\_offset\_sign | 11.4.1 (FL) | numBins = 1 |
| ptn\_resid\_abs\_gt0[ ] | 11.4.1 (FL) | numBins = 1 |
| ptn\_resid\_abs\_log2[ 𝑘 ] | 11.4.1 (FL) | numBins = ptn\_resid\_abs\_log2\_bits[ 𝑘 ] |
| ptn\_resid\_abs\_rem[ 𝑘 ] | 11.4.1 (FL) | numBins = ptn\_resid\_abs\_log2[ 𝑘 ] − 1 |
| ptn\_resid\_sign[ ] | 11.4.1 (FL) | numBins = 1 |
| ptn\_sec\_resid\_abs[ ] | 11.4.4 (TU+EGk) | maxOffset = 2, 𝑘 = 0 |
| ptn\_sec\_resid\_sign[ ] | 11.4.1 (FL) | numBins = 1 |
| ptree\_end\_of\_slice | 11.4.1 (FL) | numBins = 1 |
| zero\_run\_length\_prefix | 11.4.5 (TU) | maxVal = 3 |
| zero\_run\_length\_minus3\_div2 | 11.4.5 (TU) | maxVal = 4 |
| zero\_run\_length\_minus3\_mod2 | 11.4.1 (FL) | numBins = 1 |
| zero\_run\_length\_minus11 | 11.4.2 (EGk) | 𝑘 = 2 |

## Data unit buffer

### General

The parsing of syntax elements is specified as operations on a DU buffer. The DU buffer represents the coded DU as a sequence of unencapsulated bytes as provided by an encapsulation format such as that specified by Annex B or by another application-specific means.

### State

The DU buffer is specified in terms of the following state variables:

* The array DataUnitBytes, representing the DU buffer; DataUnitBytes[ 𝑖 ] is the 𝑖-th byte of the data unit.
* The variable DataUnitLength, equal to the length of the DU in bytes.
* The variable DataUnitReadIdx, equal to the byte index and bit position of the next bit to be read from the DU buffer.

### Initialization at the start of parsing a data unit

At the start of every DU, parsing shall commence at the first bit of the DU buffer.

DataUnitReadIdx = 0

### Initialization at the start of parsing a geometry data unit footer

The parsing of a geometry\_data\_unit\_footer syntax structure shall commence at an offset from the end of the DU buffer. The length of the GDU footer is specified by the expression DuFooterLen. The expression DuIsGdu is equal to 1 when the DU is a GDU.

GduFooterLen := 3 × (1 + occtree\_point\_cnt\_list\_present × occtreeMaxDepthMinus1)  
  
DuFooterLen := DuIsGdu ? GduFooterLen : 0  
DataUnitReadIdx = 8 × (DataUnitLength − DuFooterLen)

### Definition of DuNextBit

This subclause specifies the reading of a single bit from the DU buffer by the expression DuNextBit. Each evaluation of DuNextBit returns the next unread bit from the buffer.

duStreamByte[bitIdx] := DataUnitBytes[bitIdx >> 8]  
duStreamBit[bitIdx] := Bit(duStreamByte[bitIdx], 7 − (bitIdx & 7))  
  
DuNextBit := duStreamBit[DataUnitReadIdx++]

## Chunked bytestream parsing

### General

This subclause applies to GDUs and ADUs that contain syntax elements with ae(v) descriptors when bypass\_stream\_enabled is 1.

1. An ADU with attr\_coding\_type equal to 3 does not contain any ae(v) syntax elements.

The CBS representation conveys two multiplexed data streams as a sequence of chunks: a stream of arithmetic-coded bytes (AeBits) and a stream of bits that bypass the arithmetic decoding engine (BpBits). Every chunk is a block of 256 bytes, with the exception of the final chunk which may be shorter.

An example CBS is illustrated in Figure 34. It starts with two ChunkLen length chunks. From the CBS, two subtreams, AeBits and BpBits are extracted.

手机屏幕截图

描述已自动生成

Figure 34 — Multiplexed data streams in a chunked bytestream.

When occtree\_stream\_cnt\_minus1 is greater than 0, each of an occupancy tree's entropy streams shall be conveyed in a separate CBS (11.3.12). Consecutive CBSs shall be spliced together (11.3.11) such that the last chunk of each CBS is merged with the first chunk of the next. Splicing pads the last chunk of a CBS to 256 bytes.

### Chunk syntax

|  |  |
| --- | --- |
| ae\_chunk( ) { | **Descriptor** |
| chunk\_ae\_len | u(8) |
| for( 𝑖 = 0; 𝑖 < chunk\_ae\_len; 𝑖++ ) |  |
| chunk\_ae\_byte[ 𝑖 ] | u(8) |
| for( 𝑖 = 0; 𝑖 < ChunkPadLen; 𝑖++ ) |  |
| chunk\_splice\_byte[ 𝑖 ] |  |
| if( chunk\_ae\_len < ChunkLen − 1 − ChunkPadLen ) { |  |
| chunk\_bypass\_5bits | u(5) |
| chunk\_bypass\_flushed\_bits | u(3) |
| } |  |
| for( 𝑖 = 0; 𝑖 < ChunkLen − 2 − chunk\_ae\_len − ChunkPadLen; 𝑖++) |  |
| chunk\_bypass\_byte[ 𝑖 ] | u(8) |
| } |  |

### Chunk semantics

chunk\_ae\_len specifies the number of chunk\_ae\_byte syntax elements present in the chunk. It is a requirement of bitstream conformance that chunk\_ae\_len shall be less than ChunkLen.

chunk\_ae\_byte[ 𝑖 ] specifies the 𝑖-th arithmetic-coded byte conveyed by the chunk.

chunk\_splice\_byte[ 𝑖 ] specifies a padding byte used to pad the last chunk of a CBS. The padding bytes shall consist of bytes moved from the start of the next CBS.

chunk\_bypass\_byte[ 𝑖 ], chunk\_bypass\_5bits and chunk\_bypass\_flushed\_bits together specify the bypass-coded bits conveyed by the chunk. Within a chunk, the bits are in reverse order, as specified by the unpacking process (11.3.8).

### State

The CBSs are specified in terms of the following state variables:

* The 256 byte array ChunkBuf, a buffer used to merge and unpack chunks.
* The array AeBits of unpacked arithmetic-coded bits; each element is a single bit. The variable AeBitsLen is the length of the array.
* The array BpBits of unpacked bypass-coded bits; each element is a single bit. The variable BpBitsLen is the length of the array.
* The variables AeBitsReadIdx and BpBitsReadIdx, indexes of the next element to be read from the AeBits and BpBits arrays, respectively.

### Span of chunked bytestream data within a data unit

Each applicable DU comprises a header, the CBS data and a footer (if present).

The CBS data starts at the byte aligned position at or prior to the first ae(v) coded syntax element.

When a DU footer is present, the CBS data ends immediately prior to the first non-ae(v) coded syntax element of the fixed-length footer. Otherwise, the end shall coincide with the end of the DU buffer.

The number of bytes remaining in the CBS data is specified by the expression ChunkDuRem.

ChunkDuRem := DataUnitLength − (DataUnitReadIdx >> 3) − DuFooterLen

### The chunk buffer

Immediately prior to parsing a chunk, the chunk buffer is populated with the bytes of the next chunk from the CBS data span.

Unless 11.3.11 applies, the chunk buffer is populated by the next ChunkLen unparsed bytes from the DU buffer. Every chunk shall be either 256 bytes in length, or as long as the remaining bytes in the CBS data span, whichever is shorter.

ChunkLen = Min(256, ChunkDuRem)  
for (i = 0; i < ChunkLen; i++) {  
 ChunkBuf[i] = DataUnitBytes[DataUnitReadIdx >> 3]  
 DataUnitReadIdx += 8  
}

### State update at the start of every CBS

No unpacked data shall be preserved across CBSs. Immediately prior to unpacking the first chunk of a CBS, the unpacked arithmetic- and bypass-coded bit buffers and their respective read positions shall be cleared.

AeBitsLen = BpBitsLen = 0  
AeBitsReadIdx = BpBitsReadIdx = 0

### Unpacking a single chunk

Unpacking a single chunk comprises parsing the contents of the chunk buffer and appending the per-stream data to the unpacked streams. Parsing shall be performed according to the syntax and semantics of the ae\_chunk syntax structure with ChunkPadLen assumed to be 0.

Any arithmetic-coded data are appended to the unpacked array AeBits.

for (i = 0; i < chunk\_ae\_len; i++)  
 for (b = 7; b ≥ 0; b−−)  
 AeBits[AeBitsLen++] = Bit(chunk\_ae\_byte[i], b)

Any bypass data are appended to the unpacked array BpBits. Bypass data are appended in reverse order of the chunk data. The last chunk\_bypass\_flushed\_bits are excluded.

numChunkBypassBytes := Max(0, ChunkLen − 2 − chunk\_ae\_len)

for (j = numChunkBypassBytes − 1; j ≥ 0; j−−)  
 for (b = 7; b ≥ 0; b−−)  
 BpBits[BpBitsLen++] = Bit(chunk\_bypass\_byte[j], b)

for (b = 4; b ≥ 0; b−−)  
 BpBits[BpBitsLen++] = Bit(chunk\_bypass\_5bits, b)  
BpBitsLen −= chunk\_bypass\_flushed\_bits

### Definition of ChunkNextAeBit

This subclause specifies the reading of a single bit from the arithmetic-coded bitstream by the expression ChunkNextAeBit. Each evaluation of the expression returns the next unread bit from the stream.

Prior to reading a bit from the stream, if there are no unread bits left in the stream buffer, subsequent chunks shall be unpacked as specified by 11.3.8 until an unread bit is available.

ChunkNextAeBit :=  
 while (AeBitsReadIdx ≥ AeBitsLen) {  
 … /\* unpack chunk as specified by 11.3.6 \*/  
 }  
 ChunkNextAeBit = AeBits[AeBitsReadIdx++]

### Definition of ChunkNextBpBit

This subclause specifies the reading of a single bit from the bypass-coded bitstream by the expression ChunkNextBpBit. Each evaluation of the expression returns the next unread bit from the stream.

Prior to reading a bit from the stream, if there are no unread bits left in the stream buffer, subsequent chunks shall be unpacked as specified by 11.3.8 until an unread bit is available.

ChunkNextBpBit :=  
 while (BpBitsReadIdx ≥ BpBitsLen) {  
 … /\* unpack chunk as specified by 11.3.6 \*/  
 }  
 ChunkNextBpBit = BpBits[BpBitsReadIdx++]

### Boundary between spliced chunked bytestreams

This subclause applies when occtree\_stream\_cnt\_minus1 is greater than 0.

Multiple CBSs shall be spliced to form a contiguous span of CBS data. Splicing pads the last chunk of each CBS to maintain a fixed-length chunk size within that CBS. To pad a the padding data shall consist of bytes moved from the start of the next CBS, . If the length of is less than 256 bytes, then it shall first be spliced with , if existent, prior to the splicing of with .

* 1. The definition of splicing is recursive. For example, if the splicing of two CBSs A and B is denoted by A :: B, then splicing four CBSs, A to D, is performed as A :: ( B :: ( C :: D ) ).
  2. The padding process permits the start of the bypass data in the last chunk of any CBS to be located after the parsing of that CBS has commenced.

图示

描述已自动生成

Figure 35 — Extraction of the first chunk from a spliced CBS.

At the boundary between two CBSs, the padding data from the preceding CBS, shall form the initial part of the first chunk of the next CBS, , as illustrated by chunk\_splice\_byte[  ] in Figure 35. The rest of the first chunk shall follow the last chunk of the preceding CBS.

The length of the padding data shall be derived from the unconsumed length of the bypass-coded bitstream specified by ChunkPadLen. The length of padding data includes the bits used to code chunk\_bypass\_flushed\_bits and the number of bits discarded.

ChunkPadLen = (BpBitsLen − BpBitsReadIdx + chunk\_bypass\_flushed\_bits + 3) / 8

* 1. The initial parsing of the chunk (11.3.8) assumes that there are no padding bytes present. For the last chunk in a CBS, this assumption might be wrong and the parsed values of chunk\_bypass\_5bits and chunk\_bypass\_flushed\_bits are meaningless.

To recover the first chunk of the next CBS, the last chunk is re-parsed according to the syntax and semantics of the ae\_chunk syntax structure with the determined value of ChunkPadLen. Any padding data is moved to the start of the chunk buffer, and the remainder of the chunk buffer populated by the next unparsed bytes from the CBS data span.

for (i = 0; i < ChunkPadLen; i++)  
 ChunkBuf[i] = chunk\_splice\_byte[i]

ChunkPartBLen := Min(256 − ChunkPadLen, ChunkDuRem)

ChunkLen = ChunkPadLen + ChunkPartBLen  
for (i = ChunkPadLen; i < ChunkLen; i++) {  
 ChunkBuf[i] = DataUnitBytes[DataUnitReadIdx >> 3]  
 DataUnitReadidx += 8  
}

After populating the chunk buffer, the parsing state shall be updated for the start of the next CBS (11.3.7) and the first chunk shall be unpacked (11.3.8).

### Location of chunked bytestream boundaries

This subclause applies when occtree\_stream\_cnt\_minus1 is greater than 0.

An additional CBS shall commence at the start of each occupancy\_tree\_level( dpth ) syntax structure where dpth is greater than OcctreeEntropyStreamDepth.

## General inverse binarization processes

### Parsing unsigned fixed-length codes (FL)

Parsing is parameterized by:

* numBits, the number of bits that represent the syntax element;
* readBit, the channel read method expression.

The result is the unsigned syntax element value PartValue, parsed and constructed as:

PartVal = 0  
for (BinIdx = 0; BinIdx < numBits; BinIdx++)  
 PartVal = (PartVal << 1) + readBit

### Parsing signed fixed-length codes (FL+S)

Parsing is parameterized by:

* numBits, the number of bits that represent the absolute syntax element value;
* readBit, the channel read method expression.

The unsigned syntax element magnitude is parsed:

PartVal = 0  
for (BinIdx = 0; BinIdx < numBits; BinIdx++)  
 PartVal = (PartVal << 1) + readBit

The result is the signed syntax element value val, parsed and constructed as:

sign = readBit  
val = sign ? −PartVal : PartVal

### Parsing 𝑘-th order exp-Golomb codes (EGk)

Parsing is parameterized by:

* 𝑘, the order of the exp-Golomb code;
* readBit, the channel read method expression.

First, a unary encoded prefix is parsed as:

prefix = 0  
for (BinIdxPfx = 0; readBit ≠ 0; BinIdxPfx++)  
 prefix++

Then, a suffix comprising 𝑘 + prefix bins is parsed:

suffix = 0  
for (BinIdxSfx = 0; BinIdxSfx < k + prefix; BinIdxSfx++)  
 suffix = (suffix << 1) + readBit

The result is the unsigned syntax element value val, constructed as:

val = Exp2(prefix + k) + suffix − Exp2(k)

### 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;
* 𝑘, the order of the exp-Golomb code;
* readBit, the channel read method expression.

First, a truncated unary encoded offset is parsed:

offset = 0  
for (BinIdxTu = 0; offset < maxOffset && readBit == 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; readBit ≠ 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) + readBit

The result is the unsigned syntax element value val, constructed as:

val = offset + Exp2(prefix + k) + suffix − Exp2(k)

### Parsing truncated unary codes (TU)

Parsing is parameterized by:

* maxVal, the limit for the encoding;
* readBit, the channel read method expression.

The result is the unsigned syntax element value PartVal, parsed and constructed as:

PartVal = 0  
for (BinIdxTu = 0; PartVal < maxVal && readBit == 1; BinIdxTu++)  
 PartVal++

### Mapping process for signed codes

The signed value of a syntax element parsed according to the descriptor se(v) shall be converted from its unsigned, parsed value. If the parsed value val is:

* even, the signed syntax element value is − ( val >> 1 );
* odd, the signed syntax element value is val + 1 >> 1.

Examples of the conversion are shown in Table 60.

Table 60 — Conversion of unsigned values for signed syntax elements

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Unsigned value | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Signed value | 0 | 1 | −1 | 2 | −2 | 3 | −3 |

### Parsing ASN.1 object identifiers

#### Object identifier syntax

|  |  |
| --- | --- |
| oid( ) { | Descriptor |
| oid\_forbidden\_zero\_bit | u(1) |
| oid\_length | u(7) |
| for( 𝑖 = 0; 𝑖 < oid\_length; 𝑖++ ) |  |
| oid\_contents\_octets[ 𝑖 ] | u(8) |
| } |  |

#### Object identifier semantics

The coded representation of an ASN.1 object identifier shall follow the ASN.1 distinguished encoding rules specified in Rec. ITU‑T X.690﻿ |‌ ISO/IEC 8825‑1.

oid\_forbidden\_zero\_bit shall be 0.

oid\_length specifies the number of octets present in oid\_contents\_octets.

oid\_contents\_octets[ 𝑖 ] is the 𝑖-th contents octet of an object identifier value encoding as specified in Rec. ITU‑T X.690﻿ |‌ ISO/IEC 8825‑1.

## CABAC parsing processes

### Initialization

The arithmetic decoding engine and CPMs shall be initialized according to 11.5.4.3 and 11.5.3.2 at the start of the following syntax structures:

* occupancy\_tree (7.3.3.4);
* occupancy\_tree\_level( dpth ) (7.3.3.5) when dpth is greater than OcctreeEntropyStreamDepth;
* predictive\_tree (7.3.3.8);
* attribute\_coeffs (7.3.4.3).

### Definition of AeReadBin

This subclause specifies the reading of a single arithmetic-coded bin as the expression AeReadBin. Each evaluation reads a single bin, parameterized by the name of the coded syntax element.

A CPM identified by the expression Ctx shall be selected according to 11.5.3.4.

If the value of Ctx is neither equal to 'bypass' nor 'terminate':

* The value of the decoded bin shall be determined in accordance with 11.5.4.5 for a single arithmetic-coded bin with Ctx as the argument prob0.
* The selected CPM shall then be updated in accordance with 11.5.3.3 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:

* When bypass\_stream\_enabled is 0, in accordance with 11.5.4.6 for an arithmetic-coded bypass bin.
* When bypass\_stream\_enabled is 1, by evaluating the expression ChunkNextBpBit (11.3.10).

If the value of Ctx is 'terminate':

* The arithmetic decoder shall be flushed in accordance with 11.5.4.8.

### Contextual probability models

#### General

A CPM is a 16-bit unsigned integer value that models the probability of a zero bin.

1. 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.

#### Initialization

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed by the parsing state restoration process (11.6).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), all CPMs shall be initialized to .

#### 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 and the expression Ctx identifies the CPM used to arithmetically code it (11.5.3.4).

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 61:

if (binVal)  
 Ctx −= CtxUpdateDelta[Ctx >> 8]  
else  
 Ctx += CtxUpdateDelta[255 − (Ctx >> 8)]

Table 61 — 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 |  | | | | | | | |

#### 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 in Table 62 (GDU) and Table 63 (ADU). Entries qualified by Offset, Prefix or Suffix individually apply when selecting a CPM for a bin of that part of the binarized syntax element.

Ctx := CtxIdx ≠ 'bypass' && CtxIdx ≠ 'terminate' ? Contexts[CtxTbl][CtxIdx] : CtxIdx

Table 62 — Values of CtxTbl and CtxIdx for binarized ae(v) coded GDU syntax elements

| Syntax element | CtxTbl | CtxIdx | | Count |
| --- | --- | --- | --- | --- |
| beam\_idx\_resid\_abs[ ] | 1 | **Offset** | 3 × (BeamPrevIdxResid[   DnBeamIdxEst] != 0)  + BinIdxTu | 6 |
| **Prefix** | 6 | 1 |
| **Suffix** | bypass | 0 |
| beam\_idx\_resid\_sign[ ] | 2 | 2 × (BeamPrevIdxResid[DnBeamIdxEst] < 0)  + (BeamPrevIdxResid[DnBeamIdxEst] > 0) | | 3 |
| direct\_dup\_point\_cnt | 3 | **Offset** | BinIdxTu | 2 |
| **Prefix** | 2 | 1 |
| **Suffix** | bypass | 0 |
| direct\_joint\_diff\_bit[ ] | na | bypass | | 0 |
| direct\_joint\_prefix[ 𝑘 ] | 4 | BinIdx & 1  ? 'bypass'  : 5 × 𝑘 + Min( 4, BinIdx / 2 ) | | 15 |
| direct\_point\_cnt\_eq2 | 5 | 0 | | 1 |
| direct\_rem\_st\_ang[ ] | 6 | CtxIdxAngPhi (9.2.13.8.4) | | 24 |
| direct\_rem\_v\_ang[ ] | 7 | CtxIdxAngTheta (9.2.13.8.6) | | 4 |
| direct\_rem[ ] | na | bypass | | 0 |
| direct\_v\_ang\_resid\_abs[ ] | 41 | Offset | BinIdxTu | 3 |
| **Prefix** | 3 | 1 |
| **Suffix** | bypass | 0 |
| direct\_v\_ang\_resid\_sign[ ] | 42 | 0 | | 1 |
| occ\_direct\_node | 8 | 0 | | 1 |
| occ\_dup\_point\_cnt | 3 | **Offset** | 0 | 1 |
| **Prefix** | 2 | 1 |
| **Suffix** | bypass | 0 |
| occ\_histogram\_hit | 9 | NeighPatR | | 9 |
| occ\_histogram\_index | 10 | CtxIdxDictHg (9.2.9.11) | | 45 |
| planar\_copy\_mode | TBA | CtxIdxPlanarCopyMode (9.2.11.12) | | 128 |
| occ\_plane\_pos[ ],   when ¬AngularEligible | 11 | CtxIdxPlanePos (9.2.11.7) | | 51 |
| occ\_plane\_pos[ 𝑘 ],  when AngularEligble && 𝑘 < 2 | 12 | CtxIdxAngPhi (9.2.13.7.7) | | 8 |
| occ\_plane\_pos[ 𝑘 ],  when AngularEligble && 𝑘 == 2 | 13 | CtxIdxAngTheta (9.2.13.7.8) | | 4 |
| occ\_recent\_hit | 14 | NeighPatR | | 9 |
| occ\_recent\_index | na | bypass | | 0 |
| occ\_single\_child | 15 | 0 | | 1 |
| occ\_single\_plane[ 𝑘 ] | 16 | 𝑘 + PlaneRef[𝑘] ? (3 \* PlanePosRef[𝑘] + 1 : 0) | | 9 |
| occ\_subtree\_qp\_offset\_abs | 17 | **Offset** | 0 | 1 |
| **Prefix** | 1 | 1 |
| **Suffix** | bypass | 0 |
| occ\_subtree\_qp\_offset\_present | na | bypass | | 0 |
| occ\_subtree\_qp\_offset\_sign | 18 | 0 | | 1 |
| occ\_symbol\_escape | 19 | NeighPatR | | 9 |
| occtree\_end\_of\_entropy\_stream | na | terminate | | 0 |
| occupancy\_bit[ ] | 20 | CtxIdxOccBit (9.2.10.6) | | 32 |
| occupancy\_idx[ ] | na | bypass | | 0 |
| gm\_comp\_partition\_block[] | 48 | 0 | | 1 |
| ptn\_child\_cnt\_xor1[ ] | 21 | BinIdxTu | | 3 |
| ptn\_dup\_point\_cnt | 22 | **Offset** | 0 | 1 |
| **Prefix** | 1 | 1 |
| **Suffix** | bypass | 0 |
| ptn\_phi\_mul\_abs\_minus2 | 23 | 16 × ptn\_inter\_flag[ nodeIdx ]   + 8 x (ptn\_inter\_flag[ nodeIdx ]        ? ptn\_inter\_pred\_mode[ nodeIdx ] > 1       : ptn\_pred\_idx[ nodeIdx ] == 0)   + Exp2( BinIdx ) + PartVal − 1 | | 32 |
| ptn\_phi\_mul\_abs\_minus9 | 24 | **Prefix** | 2 ×ptn\_inter\_flag[ nodeIdx ]   + (ptn\_inter\_flag[ nodeIdx ]   ? ptn\_inter\_pred\_mode[ nodeIdx ] > 1  : ptn\_pred\_idx[ nodeIdx ] == 0) | 4 |
| **Suffix** | bypass | 0 |
| ptn\_phi\_mul\_abs\_prefix | 25 | 4 × ptn\_inter\_flag[ nodeIdx ]  + 2 x (ptn\_inter\_flag[ nodeIdx ]        ? ptn\_inter\_pred\_mode[ nodeIdx ] > 1       : ptn\_pred\_idx[ nodeIdx ] == 0)  + BinIdxTu | | 8 |
| ptn\_phi\_mul\_sign | 26 | 2 ×ptn\_inter\_flag[ nodeIdx ]  + (ptn\_inter\_flag[ nodeIdx ]     ? ptn\_inter\_pred\_mode[ nodeIdx ] > 1    : ptn\_pred\_idx[ nodeIdx ] == 0) | | 4 |
| ptn\_pred\_mode[ ] | 27 | Exp2( BinIdx ) + PartVal − 1 | | 3 |
| ptn\_qp\_offset\_abs | 28 | **Offset** | 0 | 1 |
| **Prefix** | 1 | 1 |
| **Suffix** | bypass | 0 |
| ptn\_qp\_offset\_sign | 29 | 0 | | 1 |
| ptn\_resid\_abs\_gt0[ 𝑘 ] | 30 | 𝑘 +3 × ptn\_inter\_flag[ nodeIdx ] | | 3 |
| ptn\_resid\_abs\_log2[ 𝑘 ],  when 𝑘 == 0 || geom\_angular\_enabled | 31 + 𝑘 | Exp2( BinIdx ) + PartVal – 1  +186 × ptn\_inter\_flag[ nodeIdx ] | | 62 |
| ptn\_resid\_abs\_log2[ 𝑘 ],  when 𝑘 > 0 && ¬geom\_angular\_enabled | 31 + 𝑘 | Min( 4, ptn\_resid\_abs\_log2[ 0 ] + 1 >> 1 )  × 31  + Exp2( BinIdx ) + PartVal − 1 | | 155 |
| ptn\_resid\_abs\_rem[ ] | na | bypass | | 0 |
| ptn\_resid\_sign[ 𝑘 ] | 34 | 𝑘 +3 × ptn\_inter\_flag[ nodeIdx ] | | 6 |
| ptn\_sec\_resid\_abs[ 𝑘 ] | 35 | **Offset** | 𝑘 × BinIdxTu | 6 |
| **Prefix** | 6 + 9 × 𝑘 + Min( 4, BinIdxPfx ) | 15 |
| **Suffix** | 11 + 9 × 𝑘 + Min( 3, BinIdx Sfx) | 12 |
| ptn\_sec\_resid\_sign[ 𝑘 ] | 36 | 𝑘 | | 3 |
| ptree\_end\_of\_slice | 37 | 0 | | 1 |
| ptn\_radius\_resid\_abs | 38 | **Offset** | ( ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 :   ptn\_pred\_idx[ nodeIdx ] == 0)  + 2 × (ptn\_inter\_flag[ nodeIdx ] ? Abs(PtnPhiMul[ nodeIdx ]) > 2 : Abs(PtnPhiMul[ nodeIdx ]) > thQphi)  + 4 × BinIdxTu  +12 × ptn\_inter\_flag[ nodeIdx ] | 24 |
| **Prefix** | 10 × (   ( ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 : ptn\_pred\_idx[ nodeIdx ] ≠ 0) +  2 × (ptn\_inter\_flag[ nodeIdx ] ? Abs(PtnPhiMul[ nodeIdx ]) > 2 : Abs(PtnPhiMul[ nodeIdx ] ) > thQphi) ) + Min( 9, BinIdxPfx ) + 40 × ptn\_inter\_flag[ nodeIdx ] | 80 |
| **Suffix** | 10 × (   ( ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 : ptn\_pred\_idx[ nodeIdx ] ≠ 0) +  2 × (ptn\_inter\_flag[ nodeIdx ] ? Abs(PtnPhiMul[ nodeIdx ]) > 2 : Abs(PtnPhiMul[ nodeIdx ]) > thQphi) ) + Min( 9, BinIdxSfx ) +40 × ptn\_inter\_flag[ nodeIdx ] | 80 |
| ptn\_radius\_resid\_sign | 39 | PrevRadiusResidSign    + 2 × (PtnPhiMul[ nodeIdx ] ≠ 0)    + 4 × (PrevPhiMul ≠ 0)    + 8 × (ptn\_inter\_flag[ nodeIdx ] ?           ptn\_inter\_pred\_mode[ nodeIdx ] > 1 :           ptn\_pred\_idx[ nodeIdx ]  == 0)    +16 × (ptn\_inter\_flag[ nodeIdx ] ? 2 : InterFlagHist & 0x1) | | 48 |
| ptn\_phi\_resid\_abs\_gt0 | 41 | (ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 :   ptn\_pred\_idx[ nodeIdx ] == 0) + 2 × ptn\_inter\_flag[ nodeIdx ] | | 4 |
| ptn\_phi\_resid\_sign | 42 | (ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 :  ( ptn\_pred\_idx[ nodeIdx ] == 0) ) × 5  + (ptn\_inter\_flag[ nodeIdx ] ? 4 :     (InterFlagHist & 0x1 ?      PrevInterFrameRefIdx + 2 : PrevPhiResidSign) ) | | 10 |
| ptn\_phi\_resid\_abs\_gt1 | 43 | ( ptn\_inter\_flag[ nodeIdx ] ?   ptn\_inter\_pred\_mode[ nodeIdx ] > 1 : ptn\_pred\_idx[ nodeIdx ] == 0) + 2 × ptn\_inter\_flag[ nodeIdx ] | | 4 |
| ptn\_phi\_resid\_abs\_rem | 44 | **Prefix** | Min( 3, BinIdxPfx ) + 4 × (ptn\_inter\_flag[ nodeIdx ] ? (ptn\_inter\_pred\_mode[ nodeIdx ] > 1 ) + 1 : 0) | 24 |
| **Suffix** | Min( 3, BinIdx Sfx)  + 4 × (ptn\_inter\_flag[ nodeIdx ] ? (ptn\_inter\_pred\_mode[ nodeIdx ] > 1 ) + 1 : 0) | 24 |
| ptn\_pred\_idx | 45 | BinIdxTu | | 7 |
| ptn\_inter\_flag | 46 | InterFlagHist & 1F | | 32 |
| ptn\_pred\_direction | 49 | 0 | | 1 |
| ptn\_pred\_inter\_mode | 47 | Exp2( BinIdx ) + PartVal − 1 | | 3 |
| 1. The syntax elements occ\_dup\_point\_cnt and direct\_dup\_point\_cnt use the same context table despite using different binarizations. | | | | |

Table 63 — Values of CtxTbl and CtxIdx for binarized ae(v) coded ADU syntax elements

| Syntax element | CtxTbl | CtxIdx | | Count |
| --- | --- | --- | --- | --- |
| coeff\_abs[ 0 ]  when aps\_extension\_present == 0 | 38 | **Offset** | BinIdxTu | 2 |
| **Prefix** | 4 + Min( 4, BinIdx Pfx) | 3 |
| **Suffix** | 9 + Min( 2, BinIdx Sfx) | 3 |
| coeff\_abs[ 1 ]  when aps\_extension\_present == 0 | 38 | **Offset** | For BinIdxTu == 0:    2 + ( coeff\_abs[ 0 ] ≠ 0 )  For BinIdxTu == 1:    4 + ( coeff\_abs[ 0 ] ≤ 1 ) | 4 |
| **Prefix** | 4 + Min( 4, BinIdx Pfx ) | 3 |
| **Suffix** | 9 + Min( 2, BinIdxSfx ) | 3 |
| coeff\_abs[ 2 ]  when aps\_extension\_present == 0 | 39 | **Offset** | For BinIdxTu == 0:    ( coeff\_abs[ 0 ] ≠ 0 )    + 2 × ( coeff\_abs[ 1 ] ≠ 0 )  For BinIdxTu == 1:    4 + ( coeff\_abs[ 0 ] ≤ 1 )    + 2 × ( coeff\_abs[ 1 ] ≤ 1 ) | 8 |
| **Prefix** | 6 + Min( 4, BinIdx Pfx  ) | 3 |
| **Suffix** | 11 + Min( 2, BinIdx Sfx) | 3 |
| coeff\_abs[ 0 ],  when aps\_extension\_present == 1 | 38 | **Offset** | BinIdxTu | 2 |
| **Prefix** | 4 + Min( 13, BinIdx Pfx) | 12 |
| **Suffix** | 18 + Min( 11, BinIdx Sfx) | 12 |
| coeff\_abs[ 1 ],  when aps\_extension\_present == 1 | 38 | **Offset** | For BinIdxTu == 0:    2 + ( coeff\_abs[ 0 ] ≠ 0 )  For BinIdxTu == 1:    4 + ( coeff\_abs[ 0 ] ≤ 1 ) | 4 |
| **Prefix** | 4 + Min( 13, BinIdx Pfx ) | 12 |
| **Suffix** | 18 + Min( 11, BinIdxSfx ) | 12 |
| coeff\_abs[ 2 ],  when aps\_extension\_present == 1 | 39 | **Offset** | For BinIdxTu == 0:    ( coeff\_abs[ 0 ] ≠ 0 )    + 2 × ( coeff\_abs[ 1 ] ≠ 0 )  For BinIdxTu == 1:    4 + ( coeff\_abs[ 0 ] ≤ 1 )    + 2 × ( coeff\_abs[ 1 ] ≤ 1 ) | 8 |
| **Prefix** | 6 + Min( 13, BinIdx Pfx  ) | 12 |
| **Suffix** | 19 + Min( 11, BinIdx Sfx) | 12 |
| coeff\_sign[ ] | na | bypass | | 0 |
| zero\_run\_length\_prefix | 40 | BinIdxTu | | 3 |
| zero\_run\_length\_minus3\_div2 | 40 | 3 | | 1 |
| zero\_run\_length\_minus3\_mod2 | na | bypass | | 0 |
| zero\_run\_length\_minus11 | 40 | **Prefix** | 4 | 1 |
| **Suffix** | bypass | 0 |
| slice\_raht\_inter\_layer\_code\_mode[ lvl ] | na | bypass | | 0 |
| slice\_raht\_intra\_layer\_code\_mode[ lvl] | na | bypass | | 0 |
| 1. The prefix and suffix bins of the syntax elements coeff\_abs[ 0 ] and coeff\_abs[ 1 ] use the same values of CtxIdx and CtxTbl. | | | | |

### Arithmetic decoding engine

#### General

The arithmetic decoding engine is a context-adaptive, binary arithmetic decoder, performing binary renormalization and producing binary outputs.

* 1. An arithmetic encoding engine that complements this decoding engine is described in Annex C.
  2. The arithmetic decoding engine is related to that of SMPTE VC-2.

#### 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.

#### 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 += NextAeStreamBit  
}

#### Arithmetic-coded bitstream

The next bit to be consumed as input to the arithmetic decoder is specified by the expression NextAeStreamBit.

NextAeStreamBit := bypass\_stream\_enabled ? ChunkNextAeBit : DuNextBit

#### 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)  
 IvlRange = rangeTimesProb  
else {  
 IvlLow += rangeTimesProb  
 IvlRange −= rangeTimesProb  
}

#### Decoding a single binary bypass symbol

If bypass\_bin\_coding\_prob\_update\_disabled is 0, 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)  
 IvlRange = rangeTimesProb  
else {  
 IvlLow += rangeTimesProb  
 IvlRange −= rangeTimesProb  
}

if bypass\_bin\_coding\_prob\_update\_disabled is 1, The decoded binary value binVal is determined and the state variables IvlLow and IvlCode are updated:

IvlCode <<= 1  
IvlLow <<= 1  
binVal = IvlRange ≤ IvlCode − IvlLow  
if (binVal)  
 IvlLow -= IvlRange  
}

#### 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) | NextAeStreamBit) & 0xFFFF

If IvlRange remains less than or equal to , the process shall be repeated until it is not.

#### Arithmetic decoder flushing process

The arithmetic decoder shall be flushed at the end of each occupancy tree entropy stream.

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) {  
 NextAeStreamBit  
 IvlRange <<= 1  
}

/\* byte−align \*/  
while (ReadAeStreamIdx % 8)  
 NextAeStreamBit

## Parsing state memorization and restoration

### General

Subclause 11.6 applies when either entropy\_continuation\_enabled is 1, slice\_inter\_entropy\_continuation is 1 or occtree\_stream\_cnt\_minus1 is greater than 0.

At certain moments, the entropy parsing state is recorded and later, used as the initial state for parsing other DUs or occupancy tree entropy streams.

The entropy parsing state shall comprise:

* for a GDU, the CABAC CPMs (11.5.3), the demi-CPMs for bitwise occupancy coding (9.2.10.6), the dictionary codec state for bytewise occupancy coding (9.2.9.4), the planar occupancy coding state (9.2.11.5.2) and the state of the variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul and PrevRadiusResidSign (9.3.3.1);
* for an ADU, the CABAC CPMs only (11.5.3).

The entropy parsing state shall be recorded and restored independently according to DU type (ADU versus GDU) and for each different value of ADU AttrIdx. For example, a coded point cloud sequence with num\_attributes equal to 2 would require storage for three sets of entropy parsing state.

At the start of any GDU with slice\_entropy\_continuation equal to 0 and slice\_inter\_entropy\_continuation equal to 0, all previously recorded GDU and ADU entropy parsing state shall be discarded.

### Geometry data units

#### Memorization

The GDU entropy parsing state shall be recorded at:

* the end of every geometry\_data\_unit syntax structure (7.3.3.1); and
* the end of every occupancy\_tree\_level( dpth ) syntax structure (7.3.3.5) where dpth is equal to OcctreeEntropyStreamDepth − 1.

Memorization shall record the elements and values of the GDU entropy parsing state for restoration by the restoration process (11.6.2.2).

#### Restoration

The GDU entropy parsing state shall be restored at:

* the start of a geometry\_data\_unit syntax structure (7.3.3.1) when slice\_entropy\_continuation is 1; and
* the start of a geometry\_data\_unit syntax structure (7.3.3.1) when slice\_entropy\_continuation is 0, slice\_inter\_entropy\_continuation is 1; and
* the start of every occupancy\_tree\_level( dpth ) syntax structure (7.3.3.5) where dpth is greater than OcctreeEntropyStreamDepth.

Restoration shall restore the elements and values of the GDU entropy parsing state to those previously recorded by the memorization process (11.6.2.1). At the start of a geometry\_data\_unit syntax structure, restoration shall exclude the planar occupancy coding state.

### Attribute data units

#### Memorization

The ADU entropy parsing state shall be recorded at the end of every attribute\_data\_unit syntax structure (7.3.4.1).

Memorization shall record the elements and values of the ADU entropy parsing state for restoration by the restoration process (11.6.3.2). The state shall be recorded separately for each value of AttrIdx.

#### Restoration

The ADU entropy parsing state shall be restored at the start of each attribute\_data\_unit syntax structure (7.3.4.1) when either slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1. The restoration shall be from the state recorded by the memorization process (11.6.3.1) with the same value of AttrIdx.

### Defaulted attribute data units

The recorded ADU entropy parsing state for the attribute identified by AttrIdx shall be initialized at the start of each defaulted attribute data unit when slice\_entropy\_continuation is 0 or slice\_inter\_entropy\_continuation is 0. The initialization shall be according to 11.5.3.2 as if the data unit contained a syntax element with the descriptor ae(v).

1. While a defaulted attribute data unit does not use arithmetic coding, it is necessary to record the initialized ADU entropy parsing state when slice\_entropy\_continuation is 0 or slice\_inter\_entropy\_continuation is 0 so that ADUs in any following slices where slice\_entropy\_continuation is 1 or slice\_inter\_ entropy\_continuation is 1 do not have an indeterminate ADU parsing state.

# OBUF parsing process

## General

The acronym OBUF stands for “Optimal Binary coder with Update on the Fly”. An OBUF instance decodes information from the bitstream to obtain a bit bin.

An OBUF instance is called with two contextual information info1 and info2 as input and follows the steps of

* obtaining, based on internal statistics, an index ctxIdx pointing to an element of a table of OBUF adaptive context probability models (ACPMs) as defined in 12.2.2,
* decoding an arithmetic-coded bin bin by using the CABAC decoder with the pointed OBUF ACPM (12.5),
* and updating internal statistics.

Internal statistics are modeled by a memory channel that evolves (is updated) after the decoding of each bin. The memory channel is stored into three elements:

* an OBUF tree constituted of three arrays *kShift*[ ][ ], *nVisit*[ ][ ] and *ctxIdxMap*[ ][ ],
* a buffer *obufBuffer*[ ][ ] of OBUF tree leaves,
* and an array *obufCtxArray*[ ] of OBUF ACPMs.

## Creation of an OBUF instance

An OBUF instance is a set made of the three above elements, namely an OBUF tree, a buffer of OBUF tree leaves and an array of OBUF ACPMs. The OBUF tree and the array of OBUF ACPMs are uniquely attached to the OBUF instance. However, one buffer of OBUF tree leaves can be shared among several OBUF instances.

Creating an OBUF instance follows the steps of

* associating a buffer of OBUF tree leaves, that has been previously created according to clause 12.3, with the instance,
* creating and initializing OBUF trees according to clause 12.2.1,
* and creating and initializing an array of OBUF ACPMs according to clause 12.2.2.

A buffer of OBUF tree leaves is created before creating an OBUF instance because a buffer can be shared by multiple OBUF instances.

The OBUF instance is created based on the given of

* an OBUF buffer *obufBuffer*[ ][ ] of OBUF tree leaves of depth obufLeafDepth,
* a first size nBit1 in bits corresponding to the size of first contextual information info1 used as input when calling the OBUF instance,
* a second size nBit2 in bits corresponding to the size of second contextual information info2 used as input when calling the OBUF instance,
* and optionally an initialization array initObufArray.

Creation and initialization of the OBUF trees is performed according to clause 12.2.1 based on nBit1, nBit2, obufLeafDepth and initObufArray. Creation and initialization of the array of OBUF ACPMs is performed according to clause 12.2.2.

### Creation and initialization of OBUF trees

The sizes s1 and s2 of the two contextual information as well as the size obufTreeSize2 of the s1 OBUF trees are determined by

s1 = 1 << nBit1  
s2 = 1 << nBit2  
obufTreeSize2 = 1 << nBit2 – obufLeafDepth

Three double-entry 8-bit arrays *kShift*[ ][ ], *nVisit*[ ][ ] and *ctxIdxMap*[ ][ ] are created with size s1 along the first entry and size obufTreeSize2 along the second entry for a total size of s1 × obufTreeSize2 each.

The array *kShift*[ ][ ] is initialized with all s1 × obufTreeSize2 values set to s2. The array *nVisit* [ ][ 0] is initialized with all of the s1 values set to 0. The array *ctxIdxMap* [ ][ 0] is initialized with all of the s1 values set to 127.

The array *kShift*[ ][ ] corresponds to the number of erased bits of a secondary information info2 when the instance is called. The initialising to s2 indicates that all bits of any second information info2 are erased at initial state of the OBUF instance.

The array *nVisit*[ ][ ] corresponds to the number of visits of nodes of the OBUF trees. The initialising to 0 indicates that root nodes of the OBUF trees have not been visited yet at initial state of the OBUF instance.

The array *ctxIdxMap*[ ][ ] corresponds to 8-bit context indices pointing (after right shift by 3) to OBUF ACPMs of the array *obufCtxArray*[ ]. The initialising to 127 indicates pointing to the OBUF ACPM with associated probability 0.5 at initial state of the OBUF instance.

When the optional initialization array initObufArray of size s1 is provided, the array *ctxIdxMap*[ ][ ] is further initialized by

for (j = 0; j < s1; j++)  
 ctxIdxMap[j][0] = initObufArray[j]

This further initialising provides an initial statistical model for the root nodes of the s1 OBUF trees.

### Creation and initialization of an array of OBUF ACPMs

An array of OBUF ACPMS is an extension of an array of ACPMs. It is made of

* a 16-bit array *obufCtxArray*[ ] of size 32 of ACPMs as defined in 11.5.3,
* a 16-bit array obufCtxProbaBounds[ ] of size 33 of probability bounds associated with the 32 models of the array of ACPMs such that the probability *obufCtxArray*[k ] remains between the two bounds *obufCtxProbaBounds*[k ] and *obufCtxProbaBounds*[k+1 ] before decoding a bin.

The array *obufCtxArray*[ ] is initialized by the table ObufInitProba [ ] defined by Table 64, and the array *obufCtxProbaBounds*[ ] is initialized by the table ObufInitProbaBounds [ ] defined by Table 65.

Table 64 — Values of ObufInitProba[ k ]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| k | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| value | 65461 | 65160 | 64551 | 63637 | 62426 | 60929 | 59163 | 57141 |
| k | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| value | 54884 | 52413 | 49753 | 46929 | 43969 | 40899 | 37750 | 34553 |
| k | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** |
| value | 31338 | 28135 | 24977 | 21893 | 18914 | 16067 | 13382 | 10883 |
| k | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| value | 8596 | 6542 | 4740 | 3210 | 1967 | 1023 | 388 | 75 |

Table 65 — Values of ObufInitProbaBound[ k ]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| k | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  |
| value | 65535 | 65388 | 64933 | 64169 | 63105 | 61747 | 60112 | 58214 |  |
| k | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |  |
| value | 56069 | 53699 | 51128 | 48379 | 45480 | 42458 | 39340 | 36160 |  |
| k | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** |  |
| value | 32946 | 29730 | 26541 | 23413 | 20374 | 17454 | 14681 | 12083 |  |
| k | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** |
| value | 9684 | 7509 | 5575 | 3905 | 2515 | 1419 | 627 | 150 | 0 |

## Creation of a buffer of OBUF tree leaves

A buffer of OBUF tree leaves is a rolling buffer whose elements are fully deployed trees that can be attached to a leaf node of an OBUF tree to prolong the latter.

A buffer of OBUF tree leaves is created based on the given of

* a buffer size obufBufferSize,
* and a depth obufLeafDepth of fully deployed trees constituting each element of the buffer.

The number deployedTreeSize of leaf nodes of fully deployed trees is

deployedTreeSize = 1 << obufLeafDepth

The buffer is composed of

* one 8-bit array *obufBuffer*[ ][ ] of size obufBufferSize along its first component and of size deployedTreeSize along its second component. The k-th OBUF tree leaf in the buffer is the array *obufBuffer*[k ][ ] of size deployedTreeSize,
* a buffer element index nextUsableIdx indicating the position of the next OBUF tree leaf *obufBuffer*[usableIdx] [ ] that is usable for attaching to a leaf node of an OBUF tree,
* and a rolling flag obufBufferRolled that indicates if the buffer element index has rolled back to the start of the buffer at least once.

An array *obufBuffer*[k ][ ] is made of 8-bit context indices pointing (after right shift by 3) to OBUF ACPMs of the array *obufCtxArray*[ ] associated to an OBUF instance. An array *obufBuffer*[k][] is used to prolong the array *ctxIdxMap*[ ][ ] of the OBUF instance once attached to a leaf node of the OBUF tree of the OBUF instance.

The buffer element index nextUsableIdx and the rolling flag obufBufferRolled are initialized to 0. However, the array *obufBuffer*[ ][ ] does not require initialization as each OBUF tree leaf *obufBuffer*[k][ ] will be initialized when attached to a leaf node of an OBUF tree.

## Call and update of an OBUF instance

An OBUF instance is called with input

* a first contextual information info1 of size nBit1 bits,
* and a second contextual information info2 of size nBit2 bits.

The output is a decoded bin bin.

A reduced second contextual information info2Red and the number nErasedBit of bits to be erased from the second contextual information are obtained by

info2Red = info2 >> obufLeafDepth  
nErasedBit = kShift[ info1][ info2Red]

Depending on the value of nErasedBit, the decoding and update process is performed according to either the OBUF trees or the buffer of OBUF tree leaves. In case nErasedBit >= obufLeafDepth , the process continues to clause 12.4.1; otherwise, the process continues to clause 12.4.2.

### Decode and update according to OBUF trees

#### Decode of a bin and context index update

The second contextual information info2Erased with erased bits, according to the number of erased nErasedBit bits, is computed by

nErasedBitTree = nErasedBit – obufLeafDepth  
info2Erased = (info2Red >> nErasedBitTree) << nErasedBitTree

An 8-bit context index ctxIdx is obtained from the array *ctxIdxMap*[ ][ ].

ctxIdx = ctxIdxMap[info1 ][ info2Erased]

The decoded bin bin is obtained by applying clause 12.5.

Depending on the value of the decoded bin bin, the obtained context index is updated in the array *ctxIdxMap*[ ][ ] by using ObufCtxIdxDelta[ ] as defined in Table 66.

if (bin)   
 ctxIdxMap[info1 ][ info2Erased] += ObufCtxIdxDelta[(255 - ctxIdx) >> 4]   
else  
 ctxIdxMap[info1 ][ info2Erased] -= ObufCtxIdxDelta[ctxIdx >> 4]

Table 66 — Values of ObufCtxIdxDelta[ k ]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| k | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| value | 0 | 1 | 1 | 2 | 4 | 7 | 9 | 11 |
| k | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| value | 14 | 16 | 19 | 23 | 22 | 22 | 20 | 15 |

The number of visits nVisit is incremented by one unit

nVisit[info1 ][ info2Erased]++

and a threshold thVisit on the number of visits is obtained by

thVisit = 3 + abs((ctxIdx-127) >> 4)

If the number of visits nVisit is strictly lower than the threshold thVisit then the call to the OBUF instance is finished. Otherwise, an updating process is performed, and the process continues to clause 12.4.1.2 if nErasedBit > 0 or continues to clause 12.4.1.3 otherwise.

#### OBUF tree update

The node of OBUF trees referenced by info1 and info2Erased is split into two new nodes with no visit for each.

nVisit[ info1][info2Erased] = 0 /\* first new node \*/  
nVisit[ info1][info2Erased + (1 << nErasedBit -1)] = 0 /\* second new node \*/

The context index *ctxIdxMap*[ info1][info2Erased] of the first of the two new nodes is automatically inherited from the node. The context index of the second of the two new nodes is obtained by copying the context index of the node.

ctxIdxMap[ info1][info2Erased + (1 << nErasedBit -1)] = ctxIdxMap[ info1][info2Erased]

The number of erased bits is decreased by one unit for all entries of *kShift*[info1 ][i ] where i corresponds to the two new nodes.

for (i = 0; i < (1<< nErasedBit); i++)   
 kShift[info1][info2Erased + i]--

This terminates the call of the OBUF instance.

#### Attaching of a buffer element

A buffer element of the buffer of OBUF leave is attached to the node of the OBUF tree referenced by info1 and info2Erased. Attaching depends on the availability of a non-used buffer element. This availability is obtained from the value of the rolling flag obufBufferRolled.

When the rolling flag obufBufferRolled is equal to 0, the buffer element *obufBuffer*[nextUsableIdx][ ] is initialized by the context index of the node

for (i = 0; i < deployedTreeSize; i++)   
 obufBuffer[nextUsableIdx][ i] = ctxIdx

and the pointer nextUsableIdx pointing to the attached element of the buffer of OBUF tree leaves is stored as pointer hidden information in the two arrays *ctxIdxMap*[ ][] and *nVisit*[ ][] by

nVisit[ info1][info2Erased] = nextUsableIdx & 255  
ctxIdxMap[ info1][info2Erased] = nextUsableIdx >> 8

The pointer nextUsableIdx is incremented.

nextUsableIdx++

Otherwise, when the rolling flag obufBufferRolled is equal to 1, there is no non-used buffer element left. A search for the best element to attach within a search window between nextUsableIdx and nextUsableIdx + 20 is performed.

distMin = 256  
idxMin = nextUsableIdx  
mask = (1 << obufLeafDepth) – 1

for (b = nextUsableIdx; b < nextUsableIdx + 20; b++) {  
 d = abs(ctxIdx - obufBuffer[b][info2 & mask])   
 if (d < dmin) {  
 distMin = d  
 idxMin = b  
 }

}

The pointer idxMin points to the best element of the buffer of OBUF tree leaves to be attached and this pointer is stored as pointer hidden information in the two arrays *ctxIdxMap*[ ][] and *nVisit*[ ][] by

nVisit[ info1][info2Erased] = idxMin & 255  
ctxIdxMap[ info1][info2Erased] = idxMin >> 8

The pointer nextUsableIdx is then modified by pointing to the element next to the attached element.

nextUsableIdx = idxMin + 1

After attaching a buffer element, independently on the value of obufBufferRolled, if the index nextUsableIdx is greater than or equal to the buffer size obufBufferSize, the buffer is rolled: the index is reset to 0 and the rolling flag obufBufferRolled is set to 1.

The number of erased bits of the node is decremented.

kShift[info1][info2Erased]--

This terminates the call of the OBUF instance.

### Decode and update according to an element of the buffer of OBUF tree leaves

A 16-bit pointer leafIdx to an element of the buffer of OBUF tree leaves is obtained from the pointer hidden information (as created in clause 12.4.1.3) in *ctxIdxMap*[ ][] and *nVisit*[ ][].

leafIdx = (ctxIdxMap[ info1][info2Erased] << 8) + nVisit[ info1][info2Erased]

An 8-bit context index ctxIdx is obtained from the pointed OBUF tree leaf *obufBuffer*[leafIdx][ ].

mask = (1 << obufLeafDepth) - 1  
ctxIdx = obufBuffer[leafIdx][info2 & mask ]

The decoded bin bin is obtained by applying clause 12.5.

Depending on the value of the decoded bin bin, the obtained context index is updated in the array *obufBuffer*[ ][ ] by using ObufCtxIdxDelta[ ] as defined in Table 66.

if (bin)   
 obufBuffer[leafIdx][info2 & mask ] += ObufCtxIdxDelta[(255 - ctxIdx) >> 4]   
else  
 obufBuffer[leafIdx][info2 & mask ] -= ObufCtxIdxDelta[ctxIdx >> 4]

This terminates the call of the OBUF instance.

## Decode of a bin based on an OBUF ACPM

A selected context SelCtx is obtained from the 8-bit context index ctxIdx and the array *obufCtxArray*[ ] of OBUF ACPMs by

idx = ctxIdx >> 3  
SelCtx = obufCtxArray[idx ]

A sanity check and correction of the 16-bit probability prob0 associated with the selected context SelCtx are performed (12.5.1) before decoding a bin (12.5.2) by using CABAC with the corrected selected context SelCtx as input.

### enProbability correction based on probability bounds

Lower and upper bounds of probability for the selected context are obtained by

lowProba = obufCtxProbaBounds[idx+1]   
upProba = obufCtxProbaBounds[idx]

In case the probability prob0 of the selected context SelCtx is not within the bounds, this probability is corrected toward the bounds and the bounds are adjusted by using the table CtxUpdateDelta[  ] defined by Table 61.

if (prob0 > upProba) {  
 prob0 = upProba  
 upProba += CtxUpdateDelta[255 - (upProba >> 8)] >> 2  
 if (idx > 0 && upProba > obufCtxProbaBounds [idx - 1]) {  
 upProba = obufCtxProbaBounds [idx - 1]   
 }  
}

if (prob0 < lowProba) {  
 prob0 = lowProba  
 lowProba -= CtxUpdateDelta[lowProba >> 8] >> 2  
 if (idx < 31 && lowProba > obufCtxProbaBounds [idx + 2]) {  
 lowProba = obufCtxProbaBounds [idx + 2]   
 }  
}

### Decoding of a bin using CABAC

A bin bin of information is decoded according to the clause 11.5.4.5 by using the corrected probability prob0 of the selected context SelCtx. Then, the probability evolves according to clause 11.5.3.3.

1. (normative)  
   Profiles and levels

This Annex will describe the profiles and levels.

* 1. Permitted ranges for syntax elements

Table A.1 to Table A.14 specify constraints on coded syntax element values in bitstreams conforming to this version of this document. Other constraints specified in this document may further constrain their permitted ranges.

Unless otherwise specified in this document, a decoder conforming to this version of this document may reject bitstreams containing syntax elements outside the permitted ranges.

Table A.1 — Permitted ranges for sequence parameter set syntax elements

| Syntax element | Range |
| --- | --- |
| reserved\_profile\_18bits | 0 |
| sps\_seq\_parameter\_set\_id | 0 |
| seq\_origin\_bits | 0 .. 31 |
| seq\_origin\_log2\_scale | 0 .. 31 |
| seq\_bbox\_size\_bits | 0 .. MaxSeqBboxDimLog2 |
| seq\_unit\_numerator\_minus1 | 0 .. 31 |
| seq\_unit\_denominator\_minus1 | 0 .. 31 |
| seq\_coded\_scale\_exponent | 0 .. 31 |
| seq\_coded\_scale\_mantissa\_bits | 0 .. 31 |
| num\_attributes | 0 .. 63 |
| attr\_components\_minus1 | 0 .. 3 |
| attr\_instance\_id | 0 .. 63 |
| attr\_bitdepth\_minus1 | 0 .. 63 |
| attr\_label | 0 .. 6 |
| attr\_property\_cnt | 0 .. 16 |
| sps\_extension\_present | 0 .. 1 |

Table A.2 — Permitted ranges for attribute parameter syntax elements

| Syntax element | Range |
| --- | --- |
| attr\_cicp\_colour\_primaries | 0 .. 255 |
| attr\_cicp\_transfer\_characteristics | 0 .. 255 |
| attr\_cicp\_matrix\_coeffs | 0 .. 255 |
| attr\_offset\_bits | 0 .. 64 |
| attr\_scale\_bits | 0 .. 16 |
| attr\_frac\_bits | 0 .. 31 |

Table A.3 — Permitted ranges for tile inventory syntax elements

| Syntax element | Range |
| --- | --- |
| ti\_seq\_parameter\_set\_id | 0 |
| tile\_origin\_bits\_minus1 | 0 .. 30 |
| tile\_size\_bits\_minus1 | 0 .. 30 |
| ti\_origin\_bits\_minus1 | 0 .. 30 |
| ti\_origin\_log2\_scale | 0 .. 31 |

Table A.4 — Permitted ranges for geometry parameter set syntax elements

| Syntax element | Range |
| --- | --- |
| gps\_seq\_parameter\_set\_id | 0 |
| gps\_geom\_origin\_log2\_scale | 0 .. 31 |
| occtree\_intra\_pred\_max\_nodesize\_log2 | 0 .. MaxSliceDimLog2 |
| occtree\_planar\_threshold | 0 .. 127 |
| gps\_angular\_origin\_bits\_minus1 | 0 .. 31 |
| ptree\_ang\_azimuth\_pi\_bits\_minus11 | 0 .. 9 |
| ptree\_ang\_azimuth\_step\_minus1 | 0 .. |
| ptree\_ang\_radius\_scale\_log2 | 0 .. 31 |
| num\_beams\_minus1 | 0 .. 254 |
| beam\_elevation\_init | ± |
| beam\_voffset\_init | ± |
| beam\_steps\_per\_rotation\_init\_minus1 | 0 .. 6 588 396 |
| beam\_elevation\_diff[ 𝑖 ] | ± |
| beam\_voffset\_diff[ 𝑖 ] | ± |
| beam\_steps\_per\_rotation\_diff[ 𝑖 ] | ±6 588 396 |
| geom\_qp | 0 .. 167 |
| ptree\_qp\_period\_log2 | 0 .. 21 |
| occtree\_direct\_node\_qp\_offset | ±167 |
| gps\_extension\_present | 0 .. 1 |
| biprediction\_enabled | 0 .. 2 |
| ptree\_ang\_azimuth\_scaling\_enabled | 0 .. 1 |
| ptree\_ang\_max\_pred\_index | 0 .. 7 |
| ptree\_ang\_pred\_list\_radius\_resid\_threshold | xxx .. xxx |
| ptree\_ang\_radius\_resid\_context\_qphi\_threshold | 0 .. (ptree\_ang\_azimuth\_pi\_bits\_minus11+12) / (ptree\_ang\_azimuth\_step\_minus1+1) |

Table A.5 — Permitted ranges for attribute parameter set syntax elements

| Syntax element | Range |
| --- | --- |
| aps\_seq\_parameter\_set\_id | 0 |
| attr\_coding\_type | 0 .. 3 |
| attr\_primary\_qp\_minus4 | 0 .. 95 |
| attr\_secondary\_qp\_offset | ±95 |
| raht\_prediction\_subtree\_min | 0 .. 19 |
| raht\_prediction\_samples\_min | 0 .. 19 |
| pred\_set\_size\_minus1 | 0 .. 2 |
| pred\_inter\_lod\_search\_range | 0 .. MaxSlicePoints − 1 |
| pred\_dist\_bias\_minus1\_xyz | 0 .. |
| pred\_max\_range\_minus1 | 0 .. |
| lod\_max\_levels\_minus1 | 0 .. MaxSliceDimLog2 – 1 |
| lod\_decimation\_mode | 0 .. 2 |
| lod\_sampling\_period\_minus2 | 0 .. MaxSlicePoints − 2 |
| lod\_initial\_dist\_log2 | 0 .. MaxSliceDimLog2 |
| pred\_direct\_max\_idx\_plus1 | 0 .. pred\_set\_size\_minus1 + 1 |
| pred\_intra\_min\_lod | 0 .. lod\_max\_levels\_minus1 + 1 |
| pred\_intra\_lod\_search\_range | 0 .. MaxSlicePoints − 1 |
| aps\_extension\_present | 0 .. 1 |
| attr\_inter\_prediction\_search\_range | 0 .. MaxSlicePoints − 1 |
| refAttrIdx | 0 .. num\_attributes − 1 |
| raht\_inter\_layer\_depth\_minus1 | 0 .. MaxSliceDimLog2 − 1 |
| raht\_inter\_skip\_layers | 0 .. raht\_inter\_layer\_depth\_minus1 + 1 |
| raht\_prediction\_weights[ ] | 0 .. 63 a |
| max\_points\_per\_sort\_log2\_plus1 | 0 .. IntLog2(MaxSlicePoints) + 2 |
| raht\_prediction\_search\_range | 0 .. MaxSlicePoints − 1 |
| a   The following condition should be satisfied: raht\_prediction\_weights[0 ] + 3 × Max(raht\_prediction\_weights[1 ], raht\_prediction\_weights[3 ]) + 3 × Max(raht\_prediction\_weights[2 ], raht\_prediction\_weights[4 ]) ≤ 63 | |

Table A.6 — Permitted ranges for frame-specific attribute properties syntax elements

| Syntax element | Range |
| --- | --- |
| fsap\_seq\_parameter\_set\_id | 0 |
| fsap\_sps\_attr\_idx | 0 .. num\_attributes – 1 |
| fsap\_num\_props | 0 .. 15 |

Table A.7 — Permitted ranges for geometry data unit syntax elements

| Syntax element | Range |
| --- | --- |
| gdu\_temporal\_id | 0 .. 7 |
| slice\_id | 0 ..  − 1 |
| prev\_slice\_id | 0 ..  − 1 |
| slice\_geom\_origin\_log2\_scale | 0 .. 31 |
| slice\_geom\_origin\_bits\_minus1 | 0 .. MaxSeqBboxDimLog2 − 1 |
| slice\_angular\_origin\_bits\_minus1 | 0 .. MaxSeqBboxDimLog2 − 1 |
| occtree\_depth\_minus1 | 0 .. MaxSliceDimLog2 + 3 |
| occtree\_stream\_cnt\_minus1 | 0 .. occtree\_depth\_minus1 |
| slice\_geom\_qp\_offset | ±167 |
| slice\_ptree\_qp\_period\_log2\_offset | ±21 |
| ptn\_radius\_min | 0 .. |
| occ\_subtree\_qp\_offset\_abs[ ][ ][ ] | 0 .. 167 |
| occ\_dup\_point\_cnt[ ] | 0 .. MaxSlicePoints − 1 |
| direct\_dup\_point\_cnt | 0 .. MaxSlicePoints − 1 |
| beam\_idx\_resid\_abs[ ] | 0 .. 254 |
| ptn\_qp\_offset\_abs[ ] | 0 .. 167 |
| ptn\_dup\_point\_cnt[ ] | 0 .. MaxSlicePoints − 1 |
| ptn\_child\_cnt\_xor1[ ] | 0 .. 3 |
| ptn\_inter\_pred\_mode[ ] | 0 .. global\_motion\_enabled ? 3 : 1 |
| ptn\_pred\_mode[ ] | 0 .. Min( 3, dpth ) ab |
| ptn\_phi\_mul\_abs\_minus9[ ][ ] | 0 .. |
| ptn\_sec\_resid\_abs[ ][ ] | 0 .. |
| a   where dpth is per predictive\_tree\_node( dpth, nodeIdx )  b   ptn\_pred\_mode[ ] may be 1 when dpth is 0 if ptree\_ang\_azimuth\_scaling\_enabled is 1 | |

Table A.8 — Permitted ranges for attribute data unit syntax elements

| Syntax element | Range |
| --- | --- |
| adu\_temporal\_id | 0 .. 7 |
| adu\_sps\_attr\_idx | 0 .. num\_attributes – 1 |
| adu\_slice\_id | 0 ..  − 1 |
| last\_comp\_pred\_coeff\_diff | ±255 |
| inter\_comp\_pred\_coeff\_diff | ±255 |
| lod\_dist\_log2\_offset | ±21 |
| attr\_qp\_offset | ±95 |
| attr\_qp\_layer\_cnt\_minus1 | 0 .. MaxSliceDimLog2 − 1 |
| attr\_qp\_layer\_offset[ 𝑖 ][ 𝑐 ] | ±95 |
| attr\_qp\_region\_cnt | 0 .. 1 |
| attr\_qp\_region\_bits\_minus1 | 0 .. MaxSliceDimLog2 − 1 |
| attr\_qp\_region\_offset[ 𝑖 ][ 𝑐 ] | ±95 |
| zero\_run\_length\_minus11 | 0 .. MaxSlicePoints − 11 |
| coeff\_abs | 0 ..  − 1 |
| attr\_AC\_qp\_layer\_cnt\_minus1 | 0 ..MaxSliceDimLog2− 1 |
| attr\_AC\_qp\_offset[ 𝑖 ][ 𝑐 ][ a] | ±95 |
| num\_inter\_filters | Min(raht\_inter\_layer\_depth\_minus1+1, RahtRootLvl) – raht\_send\_inter\_filters |
| raht\_inter\_filter\_qidx | ± |

Table A.9 — Permitted ranges for defaulted attribute data unit syntax elements

| Syntax element | Range |
| --- | --- |
| defattr\_seq\_parameter\_set\_id | 0 |
| defattr\_reserved\_zero\_3bits | 0 |
| defattr\_sps\_attr\_idx | 0 .. num\_attributes − 1 |
| defattr\_slice\_id | 0 ..  − 1 |
| defattr\_value | 0 .. AttrMaxVal |

Table A.10 — Permitted ranges for FGS parameter syntax elements

| Syntax element | Range |
| --- | --- |
| subgroup\_bbox\_origin\_bits\_minus1 | 0 ..  − 1 |
| subgroup\_bbox\_size\_bits\_minus1 | 0 ..  − 1 |

Table A.11 — Permitted ranges for FGS layer-group structure inventory syntax elements

| Syntax element | Range |
| --- | --- |
| lgsi\_seq\_parameter\_set\_id | 0 |
| lgsi\_slice\_id[] | 0 ..  − 1 |
| lgsi\_subgroup\_bbox\_origin\_bits\_minus1 | 0 ..  − 1 |
| lgsi\_subgroup\_bbox\_size\_bits\_minus1 | 0 ..  − 1 |
| lgsi\_origin\_bits\_minus1 | 0 .. 30 |
| lgsi\_origin\_xyz[] | 0 .. 30 |
| lgsi\_origin\_log2\_scale | 0 .. 31 |

Table A.12 — Permitted ranges for FGS dependent geometry data unit header syntax elements

| Syntax element | Range |
| --- | --- |
| dgdu\_slice\_id | 0 ..  − 1 |

Table A.13 — Permitted ranges for FGS attribute data unit parameter syntax elements

| Syntax element | Range |
| --- | --- |
| subgroup\_weight\_adj\_coeff\_a | 0 ..  − 1 |
| subgroup\_weight\_adj\_coeff\_b | 0 ..  − 1 |

Table A.14 — Permitted ranges for FGS dependent attribute data unit header syntax elements

| Syntax element | Range |
| --- | --- |
| dadu\_sps\_attr\_idx | 0 .. num\_attributes – 1 |
| dadu\_slice\_id | 0 ..  − 1 |
| last\_comp\_pred\_coeff\_diff[  ] | ±255 |
| inter\_comp\_pred\_coeff\_diff[  ][  ] | ±255 |
| attr\_qp\_offset[  ] | ±95 |
| attr\_qp\_layer\_cnt\_minus1 | 0 .. MaxSliceDimLog2 − 1 |
| attr\_qp\_layer\_offset[  ][  ] | ±95 |

1. (normative)  
   Type-length-value encapsulated bytestream format
   1. General

This annex specifies the syntax and semantics of a bytestream format for use by applications that deliver DUs as an ordered stream of bytes without any requirement for further encapsulation in a file format.

The bytestream format comprises a sequence of type-length-value encapsulation structures that each represent a single coded DU syntax structure.

* 1. Syntax and semantics
     1. Syntax

|  |  |
| --- | --- |
| tlv\_encapsulation( ) { | Descriptor |
| tlv\_type | u(8) |
| tlv\_num\_payload\_bytes | u(32) |
| for( 𝑖 = 0; 𝑖 < tlv\_num\_payload\_bytes; 𝑖++ ) |  |
| tlv\_payload\_byte[ 𝑖 ] | u(8) |
| } |  |

* + 1. Semantics

The order of tlv\_encapsulation structures shall follow the decoding order for the encapsulated syntax structures.

tlv\_type identifies the syntax structure represented by tlv\_payload\_byte[ ] as specified by Table B.1.

Table B.1 — Mapping of tlv\_type and associated data unit to syntax tables

| tlv\_type | Syntax table | Description |
| --- | --- | --- |
| 0 | 7.3.2.1 | Sequence parameter set data unit |
| 1 | 7.3.2.5 | Geometry parameter set data unit |
| 2 | 7.3.3.1 | Geometry data unit |
| 3 | 7.3.2.6 | Attribute parameter set data unit |
| 4 | 7.3.4.1 | Attribute data unit |
| 5 | 7.3.2.4 | Tile inventory data unit |
| 6 | 7.3.2.8 | Frame boundary marker data unit |
| 7 | 7.3.5 | Defaulted attribute data unit |
| 8 | 7.3.2.7 | Frame-specific attribute properties data unit |
| 9 | 7.3.2.9 | User data data unit |
| 10 | 7.3.3.1 | Geometry data unit unused for reference |
| 11 | 7.3.4.1 | Attribute data unit unused for reference |

tlv\_num\_payload\_bytes specifies the length in bytes of the syntax element array tlv\_payload\_byte[ ].

tlv\_payload\_byte[ 𝑖 ] is the 𝑖-th byte of payload data.

It is a requirement of bitstream conformance that when a TLV structure containing coded slice geometry has tlv\_type equal to 10, the TLV structure containing the coded slice attributes of the coded slice geometry shall have a tlv\_type equal to 11.

It is a requirement of bitstream conformance that all the geometry TLV data units for all slices within a frame must have the same TLV type, all the attribute TLV data units for all slices within a frame must have the same TLV type.

* 1. Parsing process

The decoder repeatedly parses tlv\_encapsulation structures until the end of the bytestream is encountered (as determined by unspecified means) and the last tlv\_encapsulation structure in the bytestream has been decoded.

After parsing each tlv\_encapsulation structure:

* The array DataUnitBytes is set equal to tlv\_payload\_byte[ ].
* The variable DataUnitLength is set to tlv\_num\_payload\_bytes.
* The parsing process for the syntax structure corresponding to tlv\_type as specified in Table B.1 is performed.

1. (informative)  
   Arithmetic encoding engine

This annex does not form an integral part of this document.

* 1. General

This annex describes an arithmetic encoding engine that complements the arithmetic decoding engine specified by 11.5.4. The encoding engine is essentially symmetric with the decoding engine, i.e. its complementary processes are performed in the same order. Table C.1 illustrates the correspondence between decoding and encoding processes.

Table C.1 — Correspondence between decoder and encoder arithmetic coding processes

| Process | Decoder | Encoder |
| --- | --- | --- |
| Initialization | 11.5.4.3 | C.3 |
| Symbol coding | 11.5.4.4 | C.4 |
| Renormalization | 11.5.4.7 | C.5 |
| Termination | 11.5.4.8 | C.6 |

* 1. State variables

The arithmetic encoding engine is described in terms of the following state variables:

* IvlLow, indicating the bottom of the 16-bit encoding interval.
* IvlRange, indicating the size of the 16-bit encoding interval.
* IvlCarry, a count of unresolved straddle conditions during renormalization.
  1. Initial state

The arithmetic encoding state is initialized before encoding the first binary symbol for an entropy stream:

IvlLow = 0  
IvlRange = 0xFFFF  
IvlCarry = 0

With 16-bit accuracy, 0xFFFF corresponds to an interval width value of (almost) 1.

* 1. Encoding process for a single binary symbol

Encoding is parameterized by the binary symbol binVal and its associated contextual probability prob0 of it being zero-valued.

The binary symbol is encoded by updating the encoding interval bounds [ IvlLow, IvlLow + IvlRange ] according to the symbol value and the contextual probability:

rangeTimesProb = (IvlRange × prob0) >> 16  
if (¬binVal)  
 IvlRange = rangeTimesProb  
else {  
 IvlLow += rangeTimesProb  
 IvlRange −= rangeTimesProb  
}

After encoding the symbol, the interval is renormalized and any available entropy stream bits output according to C.5.

* 1. Arithmetic encoder state renormalization process

Renormalization causes IvlLow and IvlRange to be modified exactly as for the decoder. It is performed when IvlRange is less than or equal to .

If, during renormalization, IvlLow and IvlLow + IvlRange straddle , a carry is recorded.

Bits are output to the entropy stream when IvlLow and IvlLow + IvlRange do not straddle . The output bits include any accumulated carries.

if (IvlRange ≤ 0x4000) {  
 if ((IvlLow + IvlRange − 1) ^ IvlLow ≥ 0x8000) {  
 IvlLow ^= 0x4000  
 IvlCarry++  
 } else {  
 writeBit(Bit(IvlLow, 15))  
 for (; IvlCarry > 0; IvlCarry−−)  
 writeBit(¬Bit(IvlLow, 15))  
 }  
 IvlRange <<= 1  
 IvlLow <<= 1  
 IvlLow &= 0xFFFF  
}

If IvlRange remains less than or equal to , the process is repeated until it is not.

* 1. Arithmetic encoding engine termination process

After encoding all binary symbols, there might be insufficient bits written to the entropy stream for a decoder to determine the final encoded symbols; partly because further renormalization is required – for example, MSBs might agree but the range is still larger than – and partly because there may be unresolved carries.

The following four-stage process adequately flushes the encoder by outputting remaining resolved MSBs, resolving remaining straddle conditions, flushing carry bits and finally byte aligning the output with padding bits.

while ((IvlLow + IvlRange − 1) ^ IvlLow < 0x8000) {  
 writeBit(Bit(IvlLow, 15))  
 for (; IvlCarry > 0; IvlCarry−−)  
 writeBit(¬Bit(IvlLow, 15))  
 IvlRange <<= 1  
 IvlLow <<= 1  
 IvlLow &= 0xFFFF  
}

while ((IvlLow & 0x4000) && ((IvlLow + IvlRange − 1) & 0x4000)) {  
 carry++  
 IvlLow ^= 0x4000  
 IvlLow &= 0x7FFF  
 IvlLow <<= 1  
 IvlRange <<= 1  
}

writeBit(Bit(IvlLow, 15))  
for (; IvlCarry > 0; IvlCarry−−)  
 writeBit(¬Bit(IvlLow, 15))

byte\_align()

1. (normative)  
   Partial decoding and spatial scalability
   1. General

A decoder may decode and reconstruct slice geometry coded by the occupancy tree at a lower precision than specified by 9.2. When used in conjunction with level of detail attribute scalability (lod\_scalability\_enabled is 1), attributes may be decoded and reconstructed for the lower precision geometry.

This annex specifies the lower precision slice geometry a decoder shall output for a partially decoded occupancy tree that excludes nodes smaller than a minimum node size (D.2). This annex shall only apply when the minimum node size is greater than the unit cube.

If LoD attribute scalability is enabled, attribute values shall be determined by partially decoding the slice attribute data in accordance with D.3. Otherwise, attribute values shall not be output by a decoder.

* 1. Partial slice geometry
     1. General

A decoder shall generate a lower-precision slice geometry that is equivalent to that specified by subclause D.2 at the end of the geometry decoding specified by Clause 9.

1. While the lower-precision slice geometry is specified as a post-process, it is equivalent to halting the decoding of an occupancy tree at the start of the tree level where NodeSizeLog2, is equal to MinNodeSizeLog2, quantizing the positions of any points coded by direct nodes and eliminating any coincident points.

The lower-precision slice geometry is specified in terms of the following variables:

* The variable MinNodeSizeLog2, an application-specific minimum occupancy tree node size that specifies the precision of output points.
* The array PartialPtIdx that maps points in the lower-precision slice geometry to point indexes in PointPos; PartialPtIdx[ idx ] is a point's index into the array PointPos.
* The variable PartialPtCnt, a cumulative count of points in the lower-precision slice geometry.
  + 1. Selection of partial point positions

To select the points that form the lower-precision slice geometry:

* Spatially partition the full slice geometry into a lattice sized cubic blocks.
* Select one point from each occupied block, recording its PointPos index.

In the following, the sparse array blkPtCnt identifies the blocks of the partitioned geometry; blkPtCnt[ ps ][ pt ][ pv ] greater than 0 indicates that the quantized point position ( ps, pt, pv ) is already present in the output partial slice geometry. Unset elements of blkPtCnt are inferred to be 0.

PartialPtCnt = 0  
for (ptIdx = 0; ptIdx < PointCnt; ptIdx++) {  
 ps = PointPos[ptIdx][0] >> MinNodeSizeLog2  
 pt = PointPos[ptIdx][1] >> MinNodeSizeLog2  
 pv = PointPos[ptIdx][2] >> MinNodeSizeLog2  
  
 blkPtCnt[ps][pt][pv]++  
 if (blkPtCnt[ps][pt][pv] > 1)  
 continue  
  
 PartialPtIdx[PartialPtCnt++] = ptIdx  
}

* + 1. Partial point positions

The points selected for the lower-precision slice geometry shall be quantized according to the minimum node size:

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (k = 0; k < 3; k++)  
 PointPos[ptIdx][k] = (PointPos[ptIdx][k] >> MinNodeSizeLog2) << MinNodeSizeLog2

If MinNodeSizeLog2 is greater than 1, points shall be centred within their corresponding block:

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (k = 0; k < 3; k++)  
 PointPos[ptIdx][k] |= (MinNodeSizeLog2 > 1) << MinNodeSizeLog2 − 1

* + 1. Output points

Partial decoding shall be equivalent to only outputting the selected points; i.e. PartialPtIdx[ 𝑖 ], 𝑖 ∈ 0 .. PartialPtCnt − 1.

* 1. Partial attribute decoding
     1. General

Slice attributes with lod\_scalability\_enabled equal to 1 shall be reconstructed in accordance with Clause 10; but:

* using the lower-precision slice geometry (D.2.3);
* constructing the finest detail level in accordance with D.3.2 instead of 10.6.5.2.

1. This is equivalent to not reconstructing LoDs with Lvl < MinNodeSizeLog2.
   * 1. The finest detail level

The finest detail level shall comprise the lower-precision slice geometry specified by D.2:

for (i = 0; i < PartialPtCnt; i++)  
 LodPtIdx[0][i] = PartialPtIdx[i]  
LodPtCnt[0] = PartialPointCnt

The point indexes of the finest detail level shall be sorted in ascending order of their respective Morton-coded attribute coordinates.

1. (normative)  
   Fine granularity slices
   1. General

This annex specifies the fine granularity slices decoder. This annex shall only apply when fgs\_layer\_group\_enabled equals to 1.

* 1. Coded point cloud format
     1. Fine granularity slices

A slice can comprise FGSs, where each FGS is mapped one-to-one to geometry or attribute of a subgroup in a layer-group.

The slice of FGSs is identified by a common slice identifier (slice\_id).

An FGS is identified by a pair consisting of a layer-group index (layer\_group\_idx) and a subgroup index (subgroup\_idx). When FGSs have the same pair of indexes, they shall be either the geometry or attribute of the subgroup indicated by the layer-group index and the subgroup index.

Every FGS shall include a GDU or DGDU that codes the partial slice geometry, or ADUs or DADUs that code the partial slice attributes.

The first FGS in a slice shall be FGS of GDU. This FGS may be followed by FGSs of DGDU that depends on the previously decoded GDU and DGDU.

The first FGS in a slice attribute shall be FGS of ADU. This FGS may be followed by FGSs of DADU that depends on the previously decoded ADU and DADU. FGSs of ADUs and DADUs shall be occur after FGSs of GDU and DGDU.

* + 1. Layer-group structure

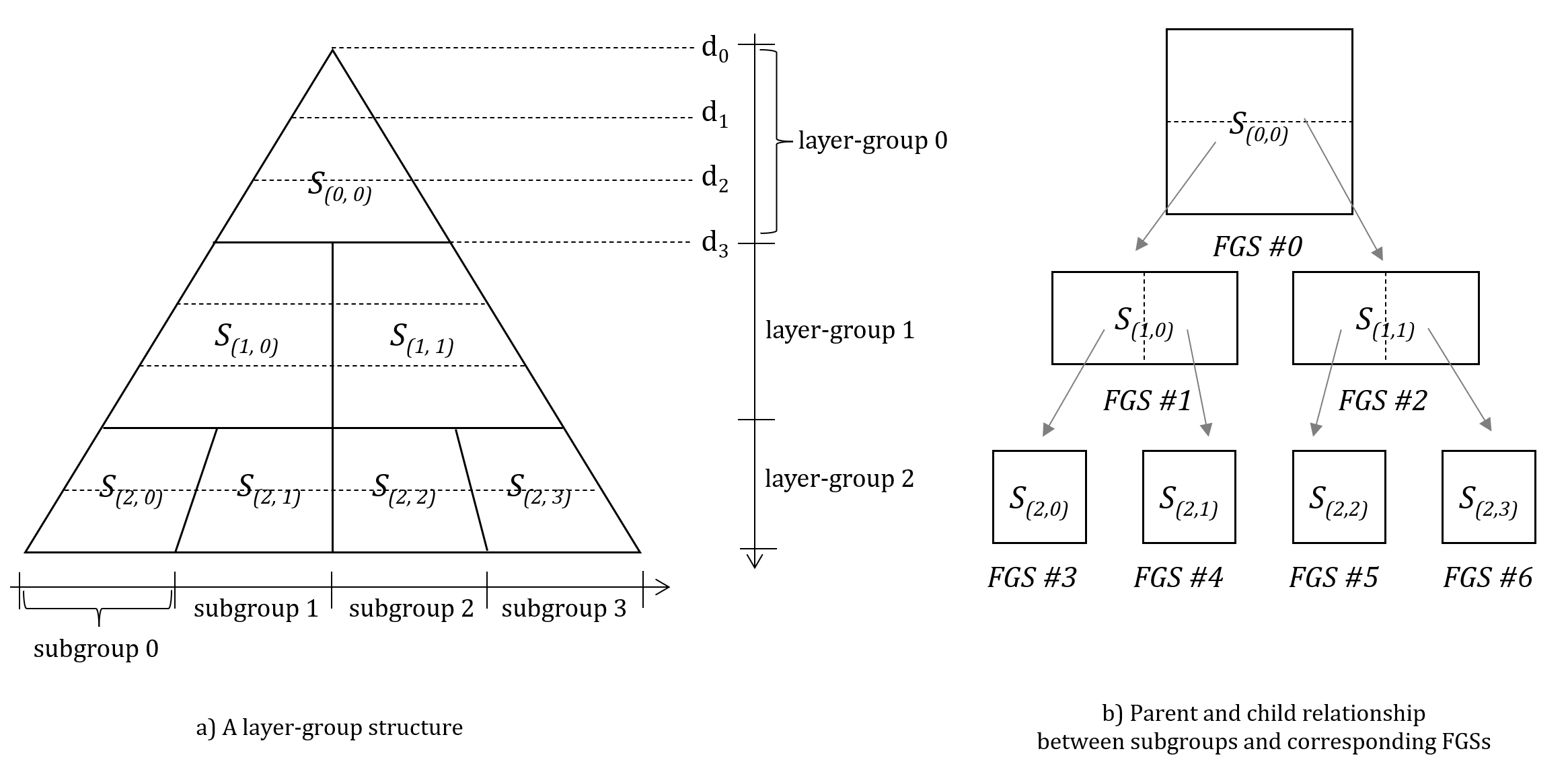
In the layer-group structure, nodes in the occupancy tree are grouped into a layer-group and a subgroup.

A layer-group is a group of consecutive tree levels, where each tree levels shall belong to only one layer-group. The minimum depth of a subgroup shall be the maximum depth of its parent subgroup plus 1, or 0 for the root layer-group. The maximum depth of a subgroup shall be the minimum depth of its child subgroup minus 1, or the greatest depth of the occupancy tree for the last layer-group. A layer-group is identified by a layer-group index (layer\_group\_idx).

A subgroup is a spatial subset of a layer-group, where a node in a tree level shall belong to only one subgroup in a layer-group. The range of the position of the nodes in a subgroup shall be described by a bounding box, which must not overlap with the bounding boxes of other subgroups in the same layer-group. The set of the nodes in all subgroups in a layer-group shall be identical to the set of the nodes in the layer-group. There is only one subgroup for the root layer-group. A subgroup in a layer-group is identified by a subgroup index (subgroup\_idx).

In Figure E.1 a), a layer-group structure of an occupancy tree with maximum depth of 8 is depicted. In this example, three layer-groups are defined and each layer-group comprises tree levels from depth 0 to 3, 4 to 6, and 7 and 8, respectively. Except for the root layer-group, layer-groups may comprise subgroups. A subgroup is indicated by the pair consisting of the layer-group index and the subgroup index. For example, the root layer-group is indicated by (0, 0).

In Figure E.1 b), the spatial region of subgroups from Figure E.1 a) are depicted by a rectangular bounding box in a xy-plane. When the bounding box of a subgroup in a layer-group is a superset of the bounding box of one or more subgroups in the next layer-group, the subgroups in adjacent layer-groups are in a parent and child relationship. In this example, subgroup (0,0) is the parent of subgroups (1,0) and (1,1). Similarly, subgroups (2,0) and (2,1) are children of subgroup (1,0). Each subgroup, indicated by a pair of layer-group index and subgroup index, is in different FGSs.



Key

|  |  |
| --- | --- |
|  | Subgroup 𝑛, associated with layer-group |
|  | Occupancy tree depth |

Figure E.1 —A layer-group structure of an occupancy tree

* 1. Syntax and semantics for fine granularity slices
     1. Syntax in tabular form
        1. General

The fine granularity slice syntax structures and the syntax elements within these structures are specified in E.3.1. Any values that are not specified in the tables shall not be present in the bitstream unless otherwise specified in this document.

* + - 1. FGS parameter sets
         1. FGS attribute parameter syntax

|  |  |
| --- | --- |
| fgs\_attr\_parameter( ) { | Descriptor |
| fgs\_attr\_ref\_id\_present | u(1) |
| } |  |

* + - * 1. FGS layer-group structure inventory syntax

|  |  |  |
| --- | --- | --- |
| fgs\_layer\_group\_structure\_inventory( ) { | Descriptor | Semantics |
| lgsi\_seq\_parameter\_set\_id | u(4) | E.3.2.2.3 |
| lgsi\_frame\_ctr\_lsb\_bits | u(5) | E.3.2.2.3 |
| lgsi\_frame\_ctr\_lsb | u(v) | E.3.2.2.3 |
| lgsi\_num\_slice\_minus1 | u(16) | E.3.2.2.3 |
| if(lgsi\_num\_slices\_minus1  0) { |  |  |
| for(sId = 0; sId ≤ lgsi\_num\_slices\_minus1; sId++) { |  |  |
| lgsi\_slice\_id[sId] | ue(v) | E.3.2.2.3 |
| lgsi\_num\_layer\_groups\_minus1[sId] | u(8) | E.3.2.2.3 |
| lgsi\_subgroup\_bbox\_origin\_bits\_minus1 | ue(v) | E.3.2.2.3 |
| lgsi\_subgroup\_bbox\_size\_bits\_minus1 | ue(v) | E.3.2.2.3 |
| for(gId = 0; gId ≤ lgsi\_num\_layer\_groups\_minus1[sId]; gId++) { |  |  |
| lgsi\_layer\_group\_id[sId][gId] | u(8) | E.3.2.2.3 |
| lgsi\_num\_layers\_minus1[sId][gId] | u(8) | E.3.2.2.3 |
| lgsi\_num\_subgroups\_minus1[sId][gId] | u(16) | E.3.2.2.3 |
| for(sgId = 0; sgId ≤ lgsi\_num\_subgroups\_minus1[sId][gId]; sgId++) { |  |  |
| lgsi\_subgroup\_id[sId][gId][sgId] | u(16) | E.3.2.2.3 |
| lgsi\_parent\_subgroup\_id[sId][gId][sgId] | u(16) | E.3.2.2.3 |
| for(k = 0; k < 3; k++) |  |  |
| lgsi\_subgroup\_bbox\_origin\_xyz[sId][gId][sgId][k] | u(v) | E.3.2.2.3 |
| for(k = 0; k < 3; k++) |  |  |
| lgsi\_subgroup\_bbox\_size\_xyz[sId][gId][sgId][k] | u(v) | E.3.2.2.3 |
| } |  |  |
| } |  |  |
| } |  |  |
| } |  |  |
| lgsi\_origin\_bits\_minus1 | ue(v) | E.3.2.2.3 |
| for(k = 0; k < 3; k++) |  |  |
| lgsi\_origin\_xyz[k] | se(v) | E.3.2.2.3 |
| lgsi\_origin\_log2\_scale | ue(v) | E.3.2.2.3 |
| byte\_alignment( ) |  |  |
| } |  |  |

* + - 1. FGS geometry data unit
         1. FGS geometry data unit parameter syntax

|  |  |  |
| --- | --- | --- |
| fgs\_geometry\_data\_unit\_parameter( ) { | Descriptor | Semantics |
| num\_layer\_groups\_minus1 | u(8) | E.3.2.3.2 |
| for( i = 0; i ≤ num\_layer\_groups\_minus1; i++ ) { |  |  |
| layerGroupIdx =  i |  |  |
| num\_layers\_minus1[ i ] | u(8) | E.3.2.3.2 |
| subgroup\_enabled[ i ] | u(1) | E.3.2.3.2 |
| } |  |  |
| fgs\_subgroup\_enabled | u(1) | E.3.2.3.2 |
| if(fgs\_subgroup\_enabled ) { |  |  |
| subgroup\_bbox\_origin\_bits\_minus1 | ue(v) | E.3.2.3.2 |
| subgroup\_bbox\_size\_bits\_minus1 | ue(v) | E.3.2.3.2 |
| } |  |  |
| for( k = 0; k ≤ 3; k++ ) |  |  |
| root\_subgroup\_bbox\_size\_log2[ k] | u(8) | E.3.2.3.2 |
| for( i = 1; i ≤ num\_layer\_groups\_minus1; i++ ) |  |  |
| num\_subsequent\_subgroups[ i ] | u(8) | E.3.2.3.2 |
| if( occtree\_planar\_enabled && ¬geom\_angular\_enabled) { |  |  |
| for( i = 0; i ≤ num\_layers\_minus1[ 0 ]; i++ ) |  |  |
| subgroup\_planar\_eligibility\_by\_density[ i ] | u(1) | E.3.2.3.2 |
| } |  |  |
| } |  |  |

* + - * 1. FGS occupancy tree syntax

|  |  |  |
| --- | --- | --- |
| fgs\_occupancy\_tree( startDepth, endDepth) { | Descriptor | Semantics |
| OccQpSubtreeDepth = occtree\_depth\_minus1 + 1 |  | 9.2.14.4 |
| for( Dpth = startDepth; Dpth ≤ endDepth; Dpth++ ) { |  |  |
| occupancy\_tree\_level( Dpth ) |  |  |
| if( Dpth + 1 > OcctreeEntropyStreamDepth ) |  | 9.2.3 |
| occtree\_end\_of\_entropy\_stream | ae(v) | 9.2.3 |
| } |  |  |
| } |  |  |

* + - * 1. FGS dependent geometry data unit syntax

|  |  |
| --- | --- |
| dependent\_geometry\_data\_unit( ) { | Descriptor |
| dependent\_geometry\_data\_unit\_header( ) |  |
| fgs\_occupancy\_tree(startDepth, endDepth) |  |
| geometry\_data\_unit\_footer(*occtreeMaxDepthMinus1*  ) |  |
| } |  |

* + - * 1. FGS dependent geometry data unit header syntax

|  |  |  |
| --- | --- | --- |
| dependent\_geometry\_data\_unit\_header( ) { | Descriptor | Semantics |
| dgdu\_geometry\_parameter\_set\_id | u(4) | E.3.2.3.4 |
| dgdu\_slice\_id | ue(v) | E.3.2.3.4 |
| layer\_group\_id | u(8) | E.3.2.3.4 |
| if( subgroup\_enabled[ layer\_group\_id] ) { |  |  |
| subgroup\_id | u(8) | E.3.2.3.4 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| subgroup\_bbox\_origin\_xyz[ 𝑘 ] | u(v) | E.3.2.3.4 |
| for( 𝑘 = 0; 𝑘 < 3; 𝑘++ ) |  |  |
| subgroup\_bbox\_size\_xyz[ 𝑘 ] | u(v) | E.3.2.3.4 |
| } |  |  |
| ref\_layer\_group\_id | u(8) | E.3.2.3.4 |
| if( subgroup\_enabled[ layer\_group\_id] ) |  |  |
| ref\_subgroup\_id | u(8) | E.3.2.3.4 |
| subgroup\_context\_reference\_indication\_enabled | u(1) | E.3.2.3.4 |
| if( subgroup\_context\_reference\_indication\_enabled) |  |  |
| for( i = 1; i ≤ num\_layer\_groups\_minus1; i++ ) |  |  |
| num\_subsequent\_subgroups[ i ] | u(8) | E.3.2.3.2 |
| if( occtree\_planar\_enabled && ¬geom\_angular\_enabled) { |  |  |
| for( i = 0; i ≤ num\_layers\_minus1[ layer\_group\_id ]; i++ ) |  |  |
| subgroup\_planar\_eligibility\_by\_density[ i ] | u(1) | E.3.2.3.4 |
| } |  |  |
| byte\_alignment( ) |  |  |
| } |  |  |

* + - 1. FGS attribute data unit
         1. FGS attribute data unit parameter syntax

|  |  |  |
| --- | --- | --- |
| fgs\_attribute\_data\_unit\_parameter( ) { | Descriptor | Semantics |
| for( i = 1; i ≤ num\_layer\_groups\_minus1; i++ ) |  |  |
| num\_subsequent\_subgroups[ i ] | u(8) | E.3.2.3.2 |
| subgroup\_weight\_adjustment\_enabled | u(1) | E.3.2.4.2 |
| if( subgroup\_weight\_adjustment\_enabled) { |  |  |
| subgroup\_weight\_adj\_coeff\_a | se(v) | E.3.2.4.2 |
| subgroup\_weight\_adj\_coeff\_b | se(v) | E.3.2.4.2 |
| } |  |  |
| } |  |  |

* + - * 1. FGS dependent attribute data unit syntax

|  |  |
| --- | --- |
| dependent\_attribute\_data\_unit( ) { | Descriptor |
| dependent\_attribute\_data\_unit\_header( ) |  |
| if(  attr\_coding\_type == 1 || attr\_coding\_type == 2 ) |  |
| attribute\_coeffs( ) |  |
| byte\_alignment( ) |  |
| } |  |

* + - * 1. FGS dependent attribute data unit header syntax

|  |  |  |
| --- | --- | --- |
| dependent\_attribute\_data\_unit\_header( ) { | Descriptor | Semantics |
| dadu\_attr\_parameter\_set\_id | u(4) | E.3.2.4.4 |
| dadu\_sps\_attr\_idx | ue(v) | E.3.2.4.4 |
| dadu\_slice\_id | ue(v) | E.3.2.4.4 |
| dadu\_layer\_group\_id | u(8) | E.3.2.4.4 |
| if(subgroup\_enabled[dadu\_layer\_group\_id]) |  |  |
| dadu\_subgroup\_id | u(8) | E.3.2.4.4 |
| if(fgs\_ attr\_ref\_id\_present ) |  |  |
| attr\_ref\_layer\_group\_id | u(8) | E.3.2.4.4 |
| if(subgroup\_enabled[dadu\_layer\_group\_id]) |  |  |
| if(fgs\_ attr\_ref\_id\_present) { |  |  |
| attr\_ref\_subgroup\_id | u(8) | E.3.2.4.4 |
| attr\_subgroup\_context\_reference\_indication\_enabled | u(1) | E.3.2.4.4 |
| if(attr\_ subgroup\_context\_reference\_indication\_enabled) |  |  |
| for( i = 1; i ≤ num\_layer\_groups\_minus1; i++ ) |  |  |
| num\_subsequent\_subgroups[ i ] | u(8) | E.3.2.3.2 |
| } |  |  |
| if( last\_comp\_pred\_enabled && AttrDim == 3) |  |  |
| for( dpth = 0; dpth ≤ num\_layers\_minus1[dadu\_layer\_group\_id]; dpth++ ) |  |  |
| last\_comp\_pred\_coeff\_diff[ dpth ] | se(v) | 10.6.10.1 |
| if( inter\_comp\_pred\_enabled ) |  |  |
| for( dpth = 0; dpth ≤ num\_layers\_minus1[dadu\_layer\_group\_id]; dpth++ ) |  |  |
| for( 𝑐 = 1; 𝑐 < AttrDim; 𝑐++) |  |  |
| inter\_comp\_pred\_coeff\_diff[ dpth ][ 𝑐 ] | se(v) | 10.6.10.1 |
| if( attr\_qp\_offsets\_present ) |  |  |
| for( qc = 0; qc < Min( 2, AttrDim ); qc++) |  |  |
| attr\_qp\_offset[ qc ] | se(v) | 10.7.1 |
| attr\_qp\_layers\_present | u(1) | 10.7.1 |
| if( attr\_qp\_layers\_present ) { |  |  |
| attr\_qp\_layer\_cnt\_minus1 | ue(v) | 10.7.1 |
| for( dpth = 0; dpth ≤ attr\_qp\_layer\_cnt\_minus1; dpth++ ) |  |  |
| for( qc = 0; qc < Min( 2, AttrDim ); qc++) |  |  |
| attr\_qp\_layer\_offset[ dpth ][ qc ] | se(v) | 10.7.1 |
| } |  |  |
| subgroup\_weight\_adjustment\_enabled | u(1) | E.3.2.4.4 |
| if( subgroup\_weight\_adjustment\_enabled) { |  |  |
| subgroup\_weight\_adj\_coeff\_a | se(v) | E.3.2.4.4 |
| subgroup\_weight\_adj\_coeff\_b | se(v) | E.3.2.4.4 |
| } |  |  |
| byte\_alignment( ) |  |  |
| } |  |  |

* + 1. Semantics
       1. General

The semantics associated with the fine granularity slice syntax structures and with the syntax elements within these structures are specified either in E.3.2 or in the subclause identified by the semantics column of the syntax table.

When the semantics of a syntax element are specified in tabular form, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

General constraints on syntax element values are specified in Annex A.

* + - 1. FGS parameter sets
         1. FGS attribute parameter semantics

fgs\_attr\_ref\_id\_present specifies whether (when 1) or not (when 0) the context references of the attribute FGS are present in the DADU header. When fgs\_attr\_ref\_id\_present equals to 1 indicates the context reference of the attribute FGSs will be indicated by the attr\_ref\_layer\_group\_id and attr\_ref\_subgroup\_id, and the context state usage in the followed attribute slices is indicated by the attr\_subgroup\_context\_reference\_indication\_enabled in the DADU header. When fgs\_attr\_ref\_id\_present is equal to 0, the context reference and the context state usage shall be inherited from the geometry FGS whose layer\_group\_id and subgroup\_id are identical to dadu\_layer\_group\_id and dadu\_subgroup\_id of the current dependent attribute FGS, respectively.

* + - * 1. FGS layer-group structure inventory semantics

A layer-group structure inventory, when present, contains metadata that defines the spatial region and parent-child relationship of each enumerated layer-groups and subgroups. Each layer-group and subgroup is identified by either an implicit or explicit layer-group id and subgroup id.

A layer-group structure inventory shall apply from the next coded point cloud frame that follows the layer-group structure inventory. It shall remain valid until it is replaced by another layer-group structure inventory.

A layer-group structure inventory shall occur before the first GDU of the coded point cloud frame from which it applies. It shall not occur before the last DU of any coded point cloud frame that precedes that from which it applies in data unit order.

lgsi\_seq\_parameter\_set\_id identifies that active SPS by its sps\_seq\_parameter\_set\_id.

lgsi\_frame\_ctr\_lsb\_bitsspecifies the length in bits of the syntax element lgsi\_frame\_ctr\_lsb. It is a requirement of bitstream conformance that lgsi\_frame\_ctr\_lsb\_bits shall be equal to frame\_ctr\_lsb\_bits of the active SPS.

lgsi\_frame\_ctr\_lsb specifies the lgsi\_frame\_ctr\_lsb\_bits least significant bits of FrameCtr from which the group structure inventory is valid. A layer-group structure inventory remains valid until it is replaced by another layer-group structure inventory.

lgsi\_num\_slices\_minus1plus 1 specifies the number of slices present in the layer-group structure inventory.

lgsi\_slice\_id[sId] specifies the slice id of the sId-th slice within the layer-group structure inventory.

lgsi\_num\_layer\_groups\_minus1[sId] plus 1 specifies the number of layer-group in the sId-th slice.

lgsi\_subgroup\_bbox\_origin\_bits\_minus1[sId] plus 1 is the length in bits of the syntax elements lgsi\_subgroup\_bbox\_origin\_xyz in the sId-th slice.

lgsi\_subgroup\_bbox\_size\_bits\_minus1[sId] plus 1 is the length in bits of the syntax elements lgsi\_subgroup\_bbox\_size\_xyz in the sId-th slice.

lgsi\_layer\_group\_id[sId][gId]specifies the indicator of a layer-group. The range of lgsi\_layer\_group\_id shall be in the range of 0 to lgsi\_num\_layer\_groups\_minus1. It is a requirement of bitstream conformance that all values of lgsi\_layer\_group\_id of gId-th layer-group in the sId-th slice are unique within a layer-group structure inventory.

lgsi\_num\_layers\_minus1[sId][gId]plus 1 specifies the number of layers in the layer-group of gId-th layer-group in the sId-th slice.

lgsi\_num\_subgroups\_minus1[sId][gId]plus 1 specifies the number of subgroups in the gId-th layer-group in the sId-th slice.

lgsi\_subgroup\_id[sId][gId][sgId]specifies the indicator of a subgroup. The range of lgsi\_subgroup\_id shall be in 0 to lgsi\_num\_subgroups\_minus1[sId][gId].

lgsi\_parent\_subgroup\_id[sId][gId][sgId] specifies the indicator of a parent subgroup of the sgId-th subgroup in the gId-th layer-group in the sId-th slice. The range of lgsi\_parent\_subgroup\_id shall be in 0 to lgsi\_num\_subgroups\_minus1[sId][gId-1].

lgsi\_subgroup\_bbox\_origin\_xyz[sId][gId][sgId][ 𝑘 ] specifies the 𝑘-th XYZ coordinate of the bounding box of the sgId-th subgroup in the gId-th layer-group in the sId-th slice.

lgsi\_subgroup\_bbox\_size\_xyz[sId][gId][sgId][ 𝑘 ] specifies the 𝑘-th XYZ dimension of the bounding box of the sgId-th subgroup in the gId-th layer-group in the sId-th slice.

lgsi\_origin\_bits\_minus1 plus 1 specifies the length in bits of the lgsi\_origin\_xyzsyntax elements.

lgsi\_origin\_xyz[ 𝑘 ] indicate the XYZ origin of the sequence. The value of lgsi\_origin\_xyz[k] should be equal to seq\_origin\_xyz[ 𝑘 ].

lgsi\_origin\_log2\_scale indicates a scaling factor to scale components of lgsi\_origin\_xyz. The value of lgsi\_origin\_log2\_scale should be equal to seq\_origin\_log2\_scale.

* + - 1. FGS geometry data unit
         1. FGS geometry data unit semantics

When fgs\_layer\_group\_enabled is equal to 1, a GDU conveys the partial slice geometry of a root layer-group and associated slice information. A GDU of fine granularity slices comprises a GDU header, geometry coded using an occupancy tree, and a GDU footer. layer\_group\_id for a GDU of fine granularity slices shall be inferred to be 0.

* + - * 1. FGS geometry data unit parameter semantics

num\_layer\_groups\_minus1 plus 1 specifies the number of layer-groups where the layer-group represents a group of consecutive tree levels within the occupancy tree. num\_layer\_groups\_minus1 shall be in the range of 0 to the number of coding tree layers.

num\_layers\_minus1[ i ] plus 1 specifies the number of tree levels in the i-th layer-group where i represents layer-group index of the i -th layer-group. The total number of layer-groups shall be derived by adding all (num\_layers\_minus1[ i ] + 1) for i equal to 0 to num\_layer\_groups\_minus1[ i ].

subgroup\_enabled[ i ] equals to 1 specifies that the i-th layer-group comprises two or more subgroups where i represents layer-group index of the i -th layer-group. subgroup\_enabled[ i ] equals to 0 specifies that the i-th layer-group comprises a subgroup.

When subgroup\_enabled[ i ] equals to 1, the aggregation of the nodes in each subgroup in the i-th layer-group shall be identical to the set of nodes in the layer-group.

When subgroup\_enabled[ i ] is equal to 1, subgroup\_enabled[ j ] shall equal to 1 when j is greater than i.

fgs\_subgroup\_enabled equals to 1 specifies that any layer-group in FGS comprises two or more subgroups. fgs\_subgroup\_enabled equals to 0 specifies that all layer-groups comprise a subgroup.

fgs\_subgroup\_enabled = 0  
for (  
i := 0; i <= num\_layer\_groups\_minus1; i++)  
 fgs\_subgroup\_enabled |= subgroup\_enabled[i]

subgroup\_bbox\_origin\_bits\_minus1 plus 1 specifies the length in bits of the syntax elements subgroup\_ bbox\_origin.

subgroup\_bbox\_size\_bits\_minus1 plus 1 is the length in bits of the syntax elements subgroup\_bbox\_size.

root\_subgroup\_bbox\_size\_log2[ k ] specifies the size of the bounding box of the root subgroup of the coded occupancy tree. MaxVec(root\_subgroup\_bbox\_size\_log2) shall be the number of coded tree layers from root to the leaf layer.

When fgs\_layer\_group\_enabled is equal to 1, the value of occtreeMaxDepthMinus1 is set as

occtreeMaxDepthMinus1 = MaxVec(root\_subgroup\_bbox\_size\_log2)- 1

The number of missing layers of a subgroup shall be the difference between occtreeMaxDepthMinus1 plus 1 and number of levels of decoded occupancy tree. The number of missing layers of a subgroup shall be used to derive the sampling direction in the subgroup LoD generation (E.6.3.3.1) or to compensate the geometry position of a node in the intermediate layers (E.8.2.3).

num\_subsequent\_subgroups[ i ] specifies the number of the subsequent dependent data units belong to the i-th layer-group which reference the context state of the current data unit. When not present, the value of num\_subsequent\_subgroups[ i ] is inferred to be zero.

subgroup\_planar\_eligibility\_by\_density[ i ] equals to 1 indicates that planar eligibility is enabled for the (i + startDepth)-th depth of the current subgroup. subgroup\_planar\_eligibility\_by\_density[ i ] equals to 0 indicates that the planar eligibility is disabled for the (i + startDepth)-th depth of the current subgroup. When not present, subgroup\_planar\_eligibility\_by\_density[ i ] is inferred to 0.

* + - * 1. FGS dependent geometry data unit semantics

A DGDU conveys partial slice geometry and associated fine granularity slice information such as a pair of layer-group index and subgroup index, subgroup bounding box, and context state reference index. A DGDU comprises a DGDU header, geometry coded using an occupancy tree, and a GDU footer.

When fgs\_layer\_group\_enabled equals to 1 and layer\_group\_id greater than 0, slice\_num\_points\_minus1 plus 1 shall equal to the number of nodes at the maximum depth of tree levels in the DGDU.

* + - * 1. FGS dependent geometry data unit header semantics

dgdu\_geometry\_parameter\_set\_id specifies the active GPS indicated by gps\_geom\_parameter\_set\_id. The value of dgdu\_geometry\_parameter\_set\_id shall be identical to the value of gdu\_geometry\_parameter\_set\_id in the same slice.

dgdu\_slice\_id specifies the slice to which the current dependent geometry data unit belongs.

layer\_group\_id specifies the indicator of a layer-group in the layer-group structure related to the slice. The range of layer\_group\_id shall be in the range 0 .. num\_layer\_groups\_minus1. When not present, layer\_group\_id shall be inferred to be 0.

subgroup\_id specifies the indicator of the subgroup in the layer-group referred by layer\_group\_id. The range of subgroup\_id shall be in the range 0 .. num\_subgroups\_minus1[layer\_group\_id] where the subgroup\_id represent a partial region in a layer-group specified by layer\_group\_id. When not present, subgroup\_id shall be inferred to be 0.

subgroup\_bbox\_origin\_xyz[ 𝑘 ] specifies the minimum 𝑘-th XYZ component position of the subgroup bounding box of the subgroup indicated by the pair of layer-group index layer\_group\_id and the subgroup index subgroup\_id.

subgroup\_bbox\_size\_xyz[ 𝑘 ] specifies the 𝑘-th XYZ size component of the subgroup bounding box of the subgroup indicated by the pair of layer-group index layer\_group\_id and the subgroup index subgroup\_id.

ref\_layer\_group\_id specifies the indicator of the layer-group identifier of the context reference of the current dependent data unit. The range of the ref\_layer\_group\_id shall be in the range 0 .. num\_layer\_groups\_minus1.

ref\_subgroup\_id specifies the indicator of the subgroup identifier of the context reference of the current dependent data unit. When not present, ref\_subgroup\_id is inferred to be 0.

The reference context state is identified by the pair of layer-group index ref\_layer\_group\_id and subgroup index ref\_subgroup\_id.

subgroup\_context\_reference\_indication\_enabled equals to 1 indicates that the context state of the current data unit will be used to initialize one or more subsequent data units. subgroup\_context\_reference\_indication\_enabled equals to 0 indicates that the context state of the current data unit will not be used to initialize the subsequent data units. When not present, subgroup\_context\_reference\_indication\_enabled is inferred to be 1.

subgroup\_planar\_eligibility\_by\_density[ i ] equals to 1 indicates that the planar eligibility is enabled for the i-th depth of the current subgroup. subgroup\_planar\_eligibility\_by\_density equals to 0 indicates that the planar eligibility is disabled for the i-th depth of the current subgroup. When not present, subgroup\_planar\_eligibility\_by\_density[ i ] is inferred to 0.

* + - 1. FGS attribute data unit
         1. FGS attribute data unit semantics

An ADU codes attribute values for a single attribute in a slice or an FGS. ADU is described in Subclause 7.4.4.1.

* + - * 1. FGS attribute data unit parameter semantics

subgroup\_weight\_adjustment\_enabledequals to 1 indicates that the subgroup weight adjustment coefficients subgroup\_weight\_adj\_coeff\_a and subgroup\_weight\_adj\_coeff\_b are present for the current subgroup corresponding to the current FGS. subgroup\_weight\_adjustment\_enabled equals to 0 indicates that the subgroup weight adjustment coefficients are not present and subgroup\_weight\_adj\_coeff\_a and subgroup\_weight\_adj\_coeff\_b are inferred to be 0.

subgroup\_weight\_adj\_coeff\_a and subgroup\_weight\_adj\_coeff\_b indicate the coefficient of subgroup weight adjustment.

* + - * 1. FGS dependent attribute data unit semantics

A DADU codes attribute values for a single attribute in an FGS. It comprises a DADU header and either attribute coefficients (attribute\_coeffs) when transform coding is equal to 1 or 2.

* + - * 1. FGS dependent attribute data unit header semantics

dadu\_attr\_parameter\_set\_id specifies the active APS by its aps\_attr\_parameter\_set\_id.

dadu\_sps\_attr\_idx identifies the coded attribute by its index into the active SPS attribute list.

At the start of every DADU, the variable AttrIdx is set to dadu\_sps\_attr\_idx:

AttrIdx = dadu\_sps\_attr\_idx

The attribute coded by the DADU shall have at most three components.

dadu\_slice\_id specifies the value of the preceding GDU slice\_id.

dadu\_layer\_group\_idspecifies the indicator of a layer-group in the layer-group structure related to the slice. The range of dadu\_layer\_group\_id shall be in the range 0 .. num\_layer\_groups\_minus1. When not present, dadu\_layer\_group\_id shall be inferred to be 0.

dadu\_subgroup\_id specifies the indicator of the subgroup in the layer-group referred by dadu\_layer\_group\_id. The range of dadu\_subgroup\_id shall be in the range 0 .. num\_subgroups\_minus1[dadu\_layer\_group\_id] where the dadu\_subgroup\_id represent a partial region in a layer-group specified by dadu\_layer\_group\_id. When not present, dadu\_subgroup\_id shall be inferred to be 0.

attr\_ref\_layer\_group\_id specifies the indicator of the layer-group identifier of the context reference of the current DADU. The range of the attr\_ref\_layer\_group\_id shall be in the range of 0 to the dadu\_layer\_group\_id of the current DADU. When not present, attr\_ref\_layer\_group\_id is inferred to be ref\_layer\_group\_id of the geometry FGS whose layer\_group\_id and subgroup\_id in DGDU are identical to dadu\_layer\_group\_id and dadu\_subgroup\_id of the current attribute FGS.

attr\_ref\_subgroup\_id specifies the indicator of the reference subgroup of the layer-group indicated by attr\_ref\_layer\_group\_id. The range of the attr\_ref\_subgroup\_id shall be in the range of 0 to num\_subgroup\_id\_minus1 of the layer-group indicated by attr\_ref\_layer\_group\_id. When not present, attr\_ref\_subgroup\_id is inferred to be ref\_subgroup\_id of the geometry FGS whose layer\_group\_id and subgroup\_id in DGDU are identical to dadu\_layer\_group\_id and dadu\_subgroup\_id of the current attribute FGS.

attr\_subgroup\_context\_reference\_indication\_enabled equals to 1 indicates that the context state of the current DADU will be used to initialize one or more subsequent data units. attr\_subgroup\_context\_reference\_indication\_enabled equals to 0 indicates that the context state of the current attribute data unit will not be used to initialize the subsequent data units. When not present, attr\_subgroup\_context\_reference\_indication\_enabled is inferred to be 1.

* 1. Decoding process
     1. General

The decoding process is described in Clause 8. This annex specifies the difference in the decoding process for the fine granularity slice.

* + 1. Fine granularity slice decoding process
       1. General

When fgs\_layer\_group\_enabled equals to 1, an FGS in a coded point cloud frame shall be decoded as follows:

1. Point positions are decoded from one GDU and zero or more DGDUs as specified by E.4.2.3.
2. Default attribute values are set for each attribute as specified by 8.3.4.
3. Point attributes are decoded from one ADU and zero or more DADUs as specified by E.4.2.4. The ADU and DADUs shall be decoded after decoding of the GDU and DGDUs those are indicated by the identical pair of subgroup index layer\_group\_id and subgroup\_id.
4. The decoded point positions are offset and the output point count incremented as specified by 8.3.6.
   * + 1. State variables

FGS decoding is specified in terms of the following state variables:

* The variable startDepth, a start depth of tree level in a data unit
* The variable *endDepth*, an end depth of tree level in a data unit
  + - 1. FGS geometry decoding process

The GDU shall be decoded before all of the DGDUs in a slice. DGDUs of child subgroups shall be decoded after DGDU of the parent subgroup.

When decoding a GDU, startDepth and  endDepth are set to 0 and (num\_layers\_minus1[ 0] + 1), respectively.

When decoding a DGDU, startDepth is set to accumulated values of (num\_layers\_minus1[ 𝑘 ] + 1) where 𝑘 is in the range of 0 .. layer\_group\_id – 1.  endDepth is set to accumulated value of (num\_layers\_minus1[ 𝑘 ] + 1) where 𝑘 is in the range of 0 .. layer\_group\_id.

The expression SubgroupNodePos[ layerGroupIdx ][ subgroupIdx ][ nodeIdx ][ 𝑘 ] is an alias into the nodes at endDepth-th depth of a subgroup identified by layerGroupIdx and subgroupIdx, where layerGroupIdx is equal to layer\_group\_id and subgroupIdx is equal to subgroup\_id.

SubgroupNodePos[layerGroupIdx ][ subgroupIdx ][nodeIdx][k] := OccNodeLoc[ endDepth ][nodeIdx][k]

The expression SubgroupNodeCnt[ layerGroupIdx ][ subgroupIdx ] is an alias into the number of nodes at endDepth-th depth of a subgroup identified by layerGroupIdx and subgroupIdx.

SubgroupNodeCnt[layerGroupIdx ][ subgroupIdx ] := OccNodeCnt[endDepth]

The expression SubgroupBBoxMin[ layerGroupIdx ][ subgroupIdx ] and SubgroupBBoxMax [ layerGroupIdx ][ subgroupIdx ] are alias into the minimum and maximum point position of the bounding box of a subgroup identified by layerGroupIdx and subgroupIdx, respectively.

SubgroupBBoxMin[layerGroupIdx][subgroupIdx][k] := subgroup\_bbox\_origin\_xyz[k]  
SubgroupBBoxMax[layerGroupIdx][subgroupIdx][k] := subgroup\_bbox\_origin\_xyz[k] + subgroup\_bbox\_size[k]

The sparse array SubgroupOccNeighPatEq0[ CurrLayerGroupIdx ][ CurrSubgroupIdx ][ ns ][ nt ][ nv ] identifies whether the identified node of the parent subgroup has no nodes present in its occupied neighbourhood pattern.

The sparse array SubgroupOccNodeChildCnt[ CurrLayerGroupIdx ][ CurrSubgroupIdx ][ *k* ][ ns ][ nt ][ nv ] identifies the number of child node of the parent subgroup.

The expression SubgroupDirectNodePointCnt[ layerGroupIdx ][ subgroupIdx ] is an alias into the number of the points of the direct nodes at endDepth-th depth of a subgroup identified by layerGroupIdx and subgroupIdx.

When layer\_group\_id is equal to num\_layer\_groups\_minus1, the position of nodes in the subgroup are copied to the output point cloud.

if (layer\_group\_id == num\_layer\_groups\_minus1) {  
 for (i = 0; i < SubgroupNodeCnt[layerGroupIdx ][ subgroupIdx ]; i++, PointCnt++)  
 for (k = 0; k < 3; k++)  
 PointPos[PointCnt][k] = SubgroupNodePos[layerGroupIdx ][ subgroupIdx ][i][k]  
}

Node positions shall be decoded and reconstructed as specified by E.5.

* + - 1. FGS attribute decoding process

The ADU shall be decoded before all of the DADUs in a slice. DADUs of child subgroups shall be decoded after DADU of the parent subgroup.

The ADU and DADU shall be decoded and the reconstructed attribute values stored in the corresponding output point cloud attribute or the leaf nodes of the occupancy tree in the subgroup.

When decoding a ADU, startDepth is set to 0.

When decoding a DADU, startDepth is set to accumulated values of (num\_layers\_minus1[ 𝑘 ] + 1) where 𝑘 is in the range of 0 .. layer\_group\_id – 1.

DirectNodePointCntis set to the number of coded points in direct nodes in each subgroup.

DirectNodePointCnt := SubgroupDirectNodePointCnt[layerGroupIdx][subgroupIdx]

The expression SubgroupNodeAttr[ layerGroupIdx ][ subgroupIdx ][ ptIdx ][ c ] is an alias into the output point cloud attribute array for the points in a subgroup identified by layerGroupIdx and subgroupIdx, where layerGroupIdx is equal to layer\_group\_id and subgroupIdx is equal to subgroup\_id.

When layer\_group\_id is equal to num\_layer\_groups\_minus1, the expression PointAttr[ ptIdx ][ 𝑐 ] is an alias into the output point cloud attribute array for the points in the FGS.

PointAttr[ptIdx][c] := RecCloudAttr[RecCloudPointCnt + ptIdx][AttrIdx][c]

Otherwise, the expression PointAttr[ ptIdx ][ 𝑐 ] is an alias into the decoded points or the leaf nodes of the occupancy tree in the subgroup.

If ptIdx < DirectNodePointCnt:

PointAttr[ptIdx][c] := RecCloudAttr[RecCloudPointCnt + ptIdx][AttrIdx][c]

Otherwise :

PointAttr[ptIdx][c] := SubgroupNodeAttr[layerGroupIdx][subgroupIdx][ptIdx- DirectNodePointCnt][c]

Point attributes shall be decoded and reconstructed as specified by E.6.

* 1. Fine granularity slice geometry
     1. General

Slice geometry shall be parsed and reconstructed point position from a coded occupancy tree in accordance with Clause 9. This annex specifies the difference of the fine granularity slice geometry decoder for parsing and reconstruction of point position.

When fgs\_layer\_group\_enabled equals to 1, the reconstructed geometry of each subgroup is stored in the arrays SubgroupNodePos (E.4.2.3).

* + 1. Partial tree of occupancy tree
       1. General

When geom\_tree\_type is 0, slice geometry shall be parsed and reconstructed point position from a coded occupancy tree in accordance with Subclause 9.2.

When fgs\_layer\_group\_enabled equals to 1, a partial occupancy tree represents the fine granularity slice geometry as a partial tree of occupancy tree nodes in terms of spatial region and depth. Fine granularity slice geometry shall be parsed or reconstructed point position from a coded partial occupancy tree as specified by E.5.2.

* + - 1. Coded occupancy tree
         1. General tree structure

An occupancy tree node shall identify the presence of at least one point contained within the volume of an axis-aligned cuboid.

When fgs\_layer\_group\_enabled equals to 1, an occupancy tree node shall identify the presence of at least one point contained within the volume of a cube. The volume is defined in the slice's coordinate system by an inclusive lower corner and an exclusive upper corner . The volume edge lengths are non-negative integer powers of two. A node's size, nodeSize, is synonymous with the volume dimensions .

* + - * 1. Tree traverse order

The coded occupancy tree shall be traversed in breadth-first order. Traversal shall start from the top tree level. All nodes in a tree level shall be sequentially traversed before proceeding to the next level. Within a tree level, nodes shall be traversed in ascending Morton order of node location.

When fgs\_layer\_group\_enabled equals to 1, the coded partial occupancy tree shall be traversed in breadth-first order in a subgroup. Traversal shall start from the minimum depth of the subgroup. All nodes in a tree level of the subgroup shall be sequentially traversed before proceeding to the next level. Within a tree level of the subgroup, nodes shall be traversed in ascending Morton order of node location.

* + - 1. State representation
         1. State variables

Traversal of the occupancy tree is specified in terms of the following state variables:

* The array OccNodeCnt; OccNodeCnt[ dpth ] is the cumulative count of nodes present at depth dpth. When fgs\_layer\_group\_enabled equals to 1, OccNodeCnt[ dpth ] represents the cumulative count of nodes present at depth dpth of a subgroup.
* The array OccNodeLoc; OccNodeLoc[ dpth ][ nodeIdx ][ 𝑘 ] identifies the 𝑘-th location component of the nodeIdx-th coded node in the traversal order of the tree level at depth dpth. When fgs\_layer\_group\_enabled equals to 1, OccNodeLoc[ dpth ][ nodeIdx ][ 𝑘 ] identifies the 𝑘-th location component of the nodeIdx-th coded node in the traversal order of the tree level at depth dpth of a subgroup.
  + - 1. Initialization

At the start of the occupancy tree syntax structure in GDU and DGDU, all elements of OccNodePresent, OccNodeLoc and OccNodeCnt are cleared.

When layer\_group\_id equals to 0, the arrays OccNodePresent, OccNodeLoc and OccNodeCnt are initialized by 9.2.5.2.

When layer\_group\_id is greater than 0, the arrays OccNodePresent, OccNodeLoc and OccNodeCnt are initialized by E.5.2.4.1 and E.5.2.4.2 to decode the occupancy tree continuous to the parent subgroup.

* + - * 1. Parent subgroup detection

The layer-group index PrtLayerGroupIdx and the subgroup index PrtSubgroupIdx of a parent subgroup are inferred by using the parent-child subgroup relationship of SubgroupBBoxMin and SubgroupBBoxMax as follows.

CurrLayerGroupIdx := layer\_group\_id  
PrtLayerGroupIdx   
:= CurrLayerGroupIdx – 1  
CurrSubgroupIdx := subgroup\_id  
  
for (i=0; i<numSubgroups[PrtLayerGroupIdx]; i++)   
 if (pL[0] ≤ cL[0]) && (pL[1] ≤ cL[1]) && (pL[2] ≤ cL[2]) &&  
 (pR[0] > cL[0]) && (pR[1] > cL[1]) && (pR[2] > cL[2]) {  
 PrtSubgroupIdx = i  
 break  
 }  
 where  
 pL := SubgroupBBoxMin[PrtLayerGroupIdx][i]  
 pR := SubgroupBBoxMax[PrtLayerGroupIdx][i]  
 cL := SubgroupBBoxMin[CurrLayerGroupIdx][CurrSubgroupIdx]

* + - * 1. Inheritance of parent subgroup output nodes

Given parent subgroup index and the bounding box information of decoded subgroups, OccNodeLoc[ startDepth ] is set as a subset of the parent subgroup output nodes SubgroupNodePos by selecting nodes in a region within the bounding box of the current subgroup depicted by SubgroupBBoxMin[ CurrLayerGroupIdx ][ CurrSubgroupIdx ] and SubgroupBBoxMax[ CurrLayerGroupIdx ][ CurrSubgroupIdx ].

count = 0  
numMissingLayers = occtree\_depth\_minus1 + 1 – endDepth  
  
for (i = 0; i < SubgroupNodeCnt[PrtLayerGroupIdx][PrtSubgroupIdx]; i++) {  
 for (k = 0; k < 3; k++)  
 nodePos[k] = SubgroupNodePos[ layerGroupIdx ][ subgroupIdx ][i][k] << numMissingLayers  
 if ((nodePos[0] ≥ cL[0]) && (nodePos[1] ≥ cL[1]) && (nodePos[2] ≥ cL[2]) &&  
 (nodePos[0] < cR[0]) && (nodePos[1] < cR[1]) && (nodePos[2] < cR[2]) ) {  
 for(k = 0; k < 3; k++)  
 OccNodeLoc[ startDepth ][count][k] =   
 SubgroupNodePos[ PrtLayerGroupIdx ][PrtSubgroupIdx ][i][k]   
 count++  
 }  
}  
 where  
 cL := SubgroupBBoxMin[CurrLayerGroupIdx][CurrSubgroupIdx]  
 cR := SubgroupBBoxMax[CurrLayerGroupIdx][CurrSubgroupIdx]

OccNodeCnt[ startDepth ] is set as the number of nodes in OccNodeLoc[ startDepth ].

OccNodeCnt[ startDepth ] = count

OccNodePresent[ startDepth ][ ns ][ nt ][ nv ] is set to 1 at a tree location ( ns, nt, nv ) equal to OccNodeLoc[ startDepth ][ nodeIdx ].

for (nodeIdx = 0; nodeIdx < OccNodeCnt[ startDepth ]; nodeIdx ++) {  
 ns = OccNodeLoc[startDepth][nodeIdx][0]  
 nt = OccNodeLoc[startDepth][nodeIdx][1]  
 nv = OccNodeLoc[startDepth][nodeIdx][2]  
 OccNodePresent[startDepth][ns][nt][nv] = 1  
}

* + - 1. Dictionary coding of occupancy\_byte
         1. Initial state

The dictionary state shall be initialized at the start of every GDU.

At the start of every DGDU, the dictionary state shall be initialized by the parsing state restoration process (11.5.3.2).

* + - 1. Bitwise occupancy coding
         1. Contextualization

Initial state

The demi-CPMs shall be initialized at the start of every GDU.

At the start of every DGDU, the demi-CPMs shall be initialized according to the parsing state restoration process (E.7.3.1.2).

* + - 1. Planar occupancy coding
         1. Per-axis eligibility

Condition

The expression *PointDensity*[*dpth* ] is a factor that identifies the density of the points in the tree level at depth *dpth*:

PointDensity[dpth ] := (slice\_num\_points\_minus1 + 1 – DirectNodePointCnt) × 10 / OccNodeCnt[ dpth ]

When fgs\_layer\_group\_enabled equals to 1, *PointDensity*[*dpth* ] is specified by subgroup\_planar\_eligibility\_by\_density in the GDUH or DGDUH.

PointDensity[dpth] = subgroup\_planar\_eligibility\_by\_density[dpth -  startDepth ]

* + - 1. Direct nodes
         1. General

The number of points coded in direct nodes is counted cumulatively, *DirectNodePointCnt*. The variable is initialized to 0 at the start of fine granularity slice.

* + - * 1. Eligibility

Initial state

At the start of every occupancy\_tree syntax structure, the OccNodeChildCnt array shall be cleared; all elements of OccNodeChildCnt are unset.

When fgs\_layer\_group\_enabled equals to 1, at the start of every occupancy\_tree syntax structure, the initial states of OccNeighPatEq0 and OccNodeChildCntare set as follows.

if (layer\_group\_enabled && dpth == startDepth) {  
 OccNeighPatEq0[ Dpth-1 ]  
 = SubgroupOccNeighPatEq0[ PrtLayerGroupIdx ][ PrtSubgroupIdx ]  
 OccNodeChildCnt[ Dpth-1 ]   
 = SubgroupOccNodeChildCnt[ PrtLayerGroupIdx ][ PrtSubgroupIdx ][0]  
 OccNodeChildCnt[ Dpth-2 ]  
 = SubgroupOccNodeChildCnt[ PrtLayerGroupIdx ][ PrtSubgroupIdx ][1]  
}

State update after each coded occupancy tree node

This subclause applies at the end of every occupancy\_tree\_node syntax structure.

The number of child nodes and the presence of any nodes in the occupied neighbourhood pattern are recorded for use in subsequent eligibility decisions.

OccNodeChildCnt[Dpth][Ns][Nt][Nv] = direct\_node ? 0 : OccChildCnt  
OccNeighPatEq0[Dpth][Ns][Nt][Nv] = OccNeighPat == 0

If the node is eligible for direct coding, irrespective of the presence of occ\_direct\_node, the count of eligible nodes shall be incremented.

if (DirectModeEligible)  
 DnEligibleCnt++

When fgs\_layer\_group\_enabled equals to 1 and dpth is equal to endDepth, the SubgroupOccNeighPatEq0and SubgroupOccNodeChildCnt are set as follows.

if (layer\_group\_enabled && dpth == endDepth){  
 SubgroupOccNeighPatEq0[ LayerGroupIdx ][ SubgroupIdx ][ Ns ][ Nt ][ Nv ]  
 = OccNeighPatEq0[ Dpth ][ Ns ][ Nt ][ Nv ]  
 SubgroupOccNodeChildCnt[ LayerGroupIdx ][SubgroupIdx ][0][ Ns ][ Nt ][ Nv ]  
 = OccNodeChildCnt[ Dpth ][ Ns ][ Nt ][ Nv ]  
 SubgroupOccNodeChildCnt[ LayerGroupIdx ][ SubgroupIdx ][1][ NsP ][ NtP ][ NvP ]  
 = OccNodeChildCnt[ Dpth-1 ][ NsP ][ NtP ][ NvP ]  
}

* 1. Fine granularity slice attributes
     1. General

Slice attributes shall be reconstructed attributes from a coded slice geometry in accordance with Clause 10. E.6 specifies the reconstruction of an FGS attribute for the coded FGS geometry. This annex specifies the difference of the fine granularity slice attribute decoder.

When fgs\_layer\_group\_enabled equals to 1, the reconstructed attributes of each subgroup is stored in the arrays SubgroupNodeAttr (E.4.2.4).

* + 1. Point coordinates
       1. General

When fgs\_layer\_group\_enabled is 1, attr\_coord\_conv\_enabled is 0 and attr\_inter\_prediction\_enabled is 0. Attribute coding can use the FGS geometry's reconstructed STV point positions.

The expression AttrPos[ ptIdx ][ 𝑘 ] specifies the coordinates of each point for attribute coding. AttrPos is equivalent to SubgroupNodePos, which is the FGS geometry in the slice’s coordinate system.

AttrPos[ptIdx][k] := SubgroupNodePos[ptIdx][k]

* + 1. Attribute decoding using levels of detail
       1. General

When attr\_coding\_type is either 1 or 2, slice attributes shall be reconstructed the attributes from a coded geometry in accordance with Subclause 10.6. The different or additional processes for the fine granularity slice’s attribute decoder are specified in E.6.3.

The attribute decoding processes specified by Subclause 10.6 and E.6.3 are distance-based prediction schemes that use a hierarchical level-of-detail representation of the slice geometry.

Detail levels are defined by an iterative subsampling process (10.6.5). The finest detail level comprises all points in the slice geometry or the FGS geometry. With each iteration, a coarser detail level is generated from the previous coarsest detail level.

Every detail level comprises a list of points present in the detail level, and is associated with a list of refinement points. A refinement point is a point that is present in a detail level and not present in any coarser detail level; the refinement points for detail level lvl, when combined with the coarser detail level lvl + 1, form detail level lvl.

For each refinement point, a set of neighbouring points is determined (10.6.6) using inter-detail-level, intra-detail-level and inter-frame and inter-layer-group searches. The neighbouring points form a predictor set that is used to predict attribute/transform coefficient values.

The index variable lvl identifies a detail level. The index lvl starts from 0. When fgs\_layer\_group\_enabled is equal to 1, the lvl value is initialized to LodMinLevel, which is set to log2 quantized node size of the leaf nodes of the FGS geometry.

* + - 1. State variables

Point predictors are specified in terms of the following state variables; the index ptIdx identifies a point by its index into AttrPos:

* The array PredPtIdx, identifies point predictors by their index in the canonical decoding order; PredPtIdx[ ptIdx ][ ni ] is the AttrPos index of the ni-th point in the predictor set for the identified point. When fgs\_layer\_group\_enabled equals to 1, PredPtIdx[ ptIdx ][ ni ] is the RefAttrPos index of the ni-th point in the predictor set for the identified point.
* The array PredPtRef of the flags to specify whether the point predictors are searched from the reference slice. When PredPtRef[ ptIdx ][ ni ] is equal to 1, the ni-th point in the predictor set is specified to be searched from the reference slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ]; When PredPtRef[ ptIdx ][ ni ] is equal to 0, the ni-th point in the predictor set is specified to be searched from the current slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ]. When fgs\_layer\_group\_enabled equals to 1 and PredPtRef[ ptIdx ][ ni ] is equal to 1, the ni-th point in the predictor set is specified to be searched from the parent FGS for the predictor identified by PredPtIdx[ ptIdx ][ ni ]; When fgs\_layer\_group\_enabled equals to 1 and PredPtRef[ ptIdx ][ ni ] is equal to 0, the ni-th point in the predictor set is specified to be searched from the current FGS for the predictor identified by PredPtIdx[ ptIdx ][ ni ].

When fgs\_layer\_group\_enabled is equal to 1 and layer\_group\_id is greater than 0, the finest detail level of the parent FGS is specified in the array ParentLodPtIdx, identifying points in the finest detail level by their index in the canonical decoding order; ParentLodPtIdx [ 𝑖 ] is the RefAttrPos index of the 𝑖-th point in the finest detail level.

The variable LodMinLevel identifies a minimum level of detail levels generated from the FGS geometry.

* + - 1. Levels of detail

When fgs\_layer\_group\_enabled is 1, general LoD generation process is specified by E.6.3.3.1.

* + - * 1. General generation process

When fgs\_layer\_group\_enabled is 1, the minimum level of LoD is specified in the variable LodMinLevel. The maximum level of the LoD is set to overlap with the minimum level of the parent FGS's LOD.

The finest detail level is identified by the detail level index LodMinLevel. LodMinLevel is set to log2 quantized node size of the leaf nodes of the FGS geometry.

Detail levels shall be iteratively subsampled (E.6.3.3.4), starting from the finest detail level, until either a single point remains or LodMaxLevel subsampled detail levels have been produced. The variable Lvl identifies the detail level to be subsampled. When layer\_group\_id is 0, LodMaxLevel is set to log2 quantized node size of the root nodes of the FGS geometry. Otherwise, LodMaxLevel is set to plus 1 for log2 quantized node size of the root nodes of the FGS geometry.

LodMinLevel = occtreeMaxDepthMinus1 – (startDepth-1)  
LodMaxLevel = LodMinLevel+num\_layers\_minus1[layer\_group\_id]+1   
if(layer\_group\_id>0)  
 LodMaxLevel++  
  
Lvl = LodMinLevel   
for (; Lvl < LodMaxLevel; Lvl++) {  
 if (LodPtCnt[Lvl] == 1)  
 break  
 … /\* subsample LodPtIdx[Lvl] \*/  
}  
LodCnt = Lvl + 1

The coarsest detail level is identified by the detail level index LodCnt − 1. All points in the coarsest detail level shall be assigned to the coarsest level's refinement list (10.6.5.3).

* + - * 1. The finest detail level

The AttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the FGS geometry.

subgroupPointCnt = SubgroupNodeCnt[layerGroupIdx][subgroupIdx] + SubgroupDirectNodePointCnt[layerGroupIdx][subgroupIdx]  
for (ptIdx = 0; ptIdx < subgroupPointCnt; ptIdx++)  
 LodPtIdx[LodMinLevel][ptIdx] = ptIdx  
LodPtCnt[LodMinLevel] = subgroupPointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The variable maxPtsPerSort identifies the max group size when sorting by group.

maxPtsPerSort = !attr\_canonical\_order\_enabled && !max\_points\_per\_sort\_log2\_plus1 ? LodPtCnt[LodMinLevel] : 1 << (max\_points\_per\_sort\_log2\_plus1 - 1)

The sorted order shall be identical for the decoding of all attributes in an FGS with identical attribute coordinate arrays (AttrPos).

1. Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < LodPtCnt[0]; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, LodPtCnt[LodMinLevel])  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = LodPtIdx[LodMinLevel][i]  
 jPtIdx = LodPtIdx[LodMinLevel][j]  
 iMorton = Morton(AttrPos[iPtIdx][0], AttrPos[iPtIdx][1], AttrPos[iPtIdx][2])  
 jMorton = Morton(AttrPos[jPtIdx][0], AttrPos[jPtIdx][1], AttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(LodPtIdx[LodMinLevel][i], LodPtIdx[LodMinLevel][j])  
 }  
}

* + - * 1. The finest detail level of the parent FGS

The expression *RefAttrPos* [*ptIdx*][*k*] specifies the coordinates of each point for attribute coding in the parent FGS. The RefAttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the parent FGS geometry. The variable parentPointCnt, the size of the number of points in the parent FGS.

for (ptIdx = 0; ptIdx < parentPointCnt; ptIdx++)  
 LodPtIdx[ptIdx] = ptIdx  
parentLodPtCnt = parentPointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The sorted order shall be identical for the decoding of all attributes in a single FGS with identical attribute coordinate arrays (RefAttrPos).

1. Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < parentPointCnt; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, parentPointCnt)  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = ParentLodPtIdx[i]  
 jPtIdx = ParentLodPtIdx[j]  
 iMorton = Morton(RefAttrPos[iPtIdx][0], RefAttrPos[iPtIdx][1],   
 RefAttrPos[iPtIdx][2])  
 jMorton = Morton(RefAttrPos[jPtIdx][0], RefAttrPos[jPtIdx][1],   
 RefAttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(ParentLodPtIdx[i], ParentLodPtIdx[j])  
 }  
}

* + - * 1. Generation of a single detail level

The coarser detail level Lvl + 1 shall be produced by subsampling the points of detail level Lvl.

When fgs\_layer\_group\_enabled is equal to 1, block-based subsampling (10.6.5.8) shall proceed.

* + - 1. Predictor search
         1. General process

The index variable lvl value is initialized to LodMinLevel.

maxLvl = LodCnt − (attr\_coding\_type == 2)  
for (Lvl = LodMinLevel; Lvl < maxLvl; Lvl++)  
 for (RfmtIdx = 0; RfmtIdx < LodRfmtPtCnt[Lvl]; RfmtIdx++) {  
 … /\* find predictors (10.6.6.2) of the current point \*/  
 }

* + - * 1. Minimum reference detail level for inter-level predictor searches

MinInterRefLvl = LodMinLevel+1  
if (lod\_scalability\_enabled) {  
 for (lvl = LodMinLevel+1; lvl < LodCnt − 1; lvl++) {  
 if (LodRfmtPtCnt[lvl] < slice\_num\_points\_minus1 − LodPtCnt[lvl])  
 break  
 MinInterRefLvl++  
 }  
}

* + - * 1. Predictor search for a single refinement point

When fgs\_layer\_group\_enabled is equal to 1 and layer\_group\_id is greater than or equal to 1, only an inter-layer-group search (E.6.3.4.4) shall be performed for the coarsest detail level.

* + - * 1. Inter-layer-group predictor search

The inter-layer-group search shall be performed by finding the parent point which a Morton code matches.

candPtIdx = ParentLodPtIdx[RfmtIdx]  
PredCnt[PtIdx] = 1  
PredPtIdx[PtIdx][i] = candPtIdx  
PredPtRef[PtIdx][i] = 1

* + - 1. Reconstruction of attribute values
         1. General process

Each detail level shall be processed in turn, proceeding from the coarsest to the finest level, according to attr\_coding\_type (10.6.7.3, 10.6.7.4). The variable Lvl is the index of the current detail level.

for (Lvl = LodCnt − 1; Lvl ≥ LodMinLevel; Lvl−−)  
 … /\* process a detail level \*/

* + - * 1. Processing of a detail level (attr\_coding\_type = 2)

When Lvl is LodMinLevel, the reconstructed attributes values shall be divided by 256 with half-values rounded away from zero and clipped to the maximum attribute value:

if (Lvl == LodMinLevel) {  
 subgroupPointCnt = SubgroupNodeCnt[layerGroupIdx][subgroupIdx] +   
 SubgroupDirectNodePointCnt[layerGroupIdx][subgroupIdx]  
 for (ptIdx = 0; ptIdx < subgroupPointCnt; ptIdx++)  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[ptIdx][c] = Clip3(0, AttrMaxVal, DivExp2Fz(PointAttr[ptIdx][c], 8))  
}

* + - 1. Transform coefficient weights
         1. Fine granularity slice case

When fgs\_layer\_group\_enabled is equal to 1, coefficient weights are calculated accumulatively, proceeding from the finest to the coarsest detail level of an FGS.

The accumulated coefficient weight of each refinement point in a detail level shall be distributed to the points in its predictor set. The distribution is proportional to the respective predictor weights:

for (lvl = LodMinLevel; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = CoeffWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 predW = PredWeight[ptIdx][ni]  
 CoeffWeight[PredPtIdx[ptIdx][ni]] += DivExp2Up(coeffW × predW, 8)  
 }  
 }  
 }

* + - 1. Transform coefficient weights
         1. Quantization weights derivation

The index variable lvl value is initialized to LodMinLevel.

for (lvl = LodMinLevel; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = QuantWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 QuantWeight[PredPtIdx[ptIdx][ni]] +=  
 DivExp2Inf(coeffW × quant\_neigh\_weight[ni], 8)  
 }  
 }  
 }

When fgs\_layer\_group\_enabled equals to 1 and subgroup\_weight\_adjustment\_enabled equals to 1, quantization weights of a subgroup is adjusted to alleviate the missing information out of the layer-group boundary. The array NumRefNodes is the number of nodes which refer a node as a neighbor. The array NumRefNodes is initialized by setting all elements to 0

maxRefNodes = 0  
numPointsInSubgroup = SubgroupNodeCnt[dadu\_layer\_group\_id ][ dadu\_subgroup\_id ] + SubgroupDirectNodePointCnt[dadu\_layer\_group\_id ][ dadu\_subgroup\_id ]  
for (i = 0; i < numPointsInSubgroup; i++) {  
 NumRefNodes[i] = 0  
}  
for (lvl = LodMinLevel; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]   
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]) {  
 NumRefNodes[PredPtIdx[ptIdx][ni]]++  
 if (maxRefNodes < NumRefNodes[PredPtIdx[ptIdx][ni]])  
 maxRefNodes = NumRefNodes[PredPtIdx[ptIdx][ni]]  
 }  
 }  
 }  
if(maxRefNodes > 0) {  
 for(i = 0; i < numPointsInSubgroup; i++){  
 QuantWeight[i] += (NumRefNodes[i]\* subgroup\_weight\_adj\_coeff\_a)/maxRefNodes + subgroup\_weight\_adj\_coeff\_b  
 }  
}

* + 1. Parent FGS generation for attributes
       1. General

When fgs\_layer\_group\_enabled is equal to 1 and layer\_group\_id is greater than 1, E.6.4.2 shall be applied.

After the point coordinates generation of the current FGS as specified by E.6.2, the parent FGS generation is proceeded.

The output of this process are the parent FGS attribute and the parent FGS geometry for attribute coding. The expression *RefPointAttr* [*ptIdx*][*c*] specifies the attribute values of each point in the parent FGS. The expression *RefAttrPos* [*ptIdx*][*k*] specifies the coordinates of each point for attribute coding in the parent FGS.

* + - 1. Parent FGS generation

The attribute values and attribute coordinates of the parent FGS are derived based on the bounding box of the coordinates of the current FGS. The number of points in the parent FGS is specifies by the variable *parentPointCnt*.

parentPointCnt = 0  
nodeSizeLog2OfParent = occtreeMaxDepthMinus1 - startDepth  
parentNodeCnt := SubgroupNodeCnt[layerGroupIdx-1][prtSubgroupIdx]   
parentNodePos[i][k] := SubgroupNodePos[layerGroupIdx-1][prtSubgroupIdx][i][k]  
parentNodeAttr[i][c] := SubgroupNodeAttr[layerGroupIdx-1][ prtSubgroupIdx][i][AttrIdx][c]  
  
for (i = 0; i < parentNodeCnt; i++){  
 parentNodePosShift = parentNodePos[i] << nodeSizeLog2OfParent  
 if ((p[0] < pR[0]) && (p[1] < pR[1]) && (p[2] < pR[2]) && (p[0] >= pL[0]) && (p[1] >= pL[1]) && (p[2] >= pL[2])) {  
 for (c = 0; c < AttrDim; c++)  
 RefPointAttr [parentPointCnt][c] = parentNodeAttr[i][c]  
 for (k = 0; k < 3; k++)  
 RefAttrPos [parentPointCnt][k] = parentNodePos[i][k]  
 parentPointCnt++  
 }  
}  
 where  
 p := parentNodePosShift  
 pR := SubgroupBBoxMax[layerGroupIdx-1][prtSubgroupIdx]  
 pL := SubgroupBBoxMin [layerGroupIdx-1][prtSubgroupIdx]

* 1. Parsing process
     1. General

The parsing process of syntax elements is described in Clause 11. This annex specifies additional processes for the fine granularity slice.

* + 1. Data unit buffer
       1. Initialization at the start of parsing a geometry data unit footer

The parsing of a geometry\_data\_unit\_footer syntax structure shall commence at an offset from the end of the DU buffer. The length of the GDU footer is specified by the expression DuFooterLen. The expression DuIsGdu is equal to 1 when the DU is a GDU or DGDU.

GduFooterLen := 3 × (1 + occtree\_point\_cnt\_list\_present × occtreeMaxDepthMinus1)  
  
DuFooterLen := DuIsGdu ? GduFooterLen : 0  
DataUnitReadIdx = 8 × (DataUnitLength − DuFooterLen)

* + 1. CABAC parsing processes
       1. Contextual probability models
          1. General

A CPM is a 16-bit unsigned integer value that models the probability of a zero bin.

The array Contexts, with elements Contexts[ ctxTbl ][ ctxIdx ], represents individual adaptive CPMs used by the CABAC parsing process.

When fgs\_layer\_group\_enabled equals to 1, the array SubgroupContexts and SubgroupContextsForAttributes are used.

The array SubgroupContexts[ layerGroupIdx ][ subgroupIdx ] represents the array of Contexts of a subgroup indicated by the layer-group index layerGroupIdx and the subgroup index subgroupIdx.

The arraySubgroupContextsForAttributes[ AttrIdx ][ layerGroupIdx ] [ subgroupIdx ] represents the array of Contexts of a subgroup attributes indicated by the attribute index AttrIdx, and the layer-group index layerGroupIdx and the subgroup index subgroupIdx.

* + - * 1. Initialization

When fgs\_layer\_group\_enabled equals to 1, initialization shall be performed by the parsing state restoration process (E.7.4).

* + 1. Parsing state memorization and restoration
       1. General

E.7.3 applies when fgs\_layer\_group\_enabled is 1.

At certain moments, the entropy parsing state is recorded and later, used as the initial state for parsing other DUs or occupancy tree entropy streams.

The entropy parsing state shall comprise:

* for a GDU or a DGDU, the CABAC CPMs (E.7.3.1), the demi-CPMs for bitwise occupancy coding (9.2.10.6), the dictionary codec state for bytewise occupancy coding (9.2.9.4), the planar occupancy coding state (9.2.11.5.2) and the state of the variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul and PrevRadiusResidSign (9.3.3.1);
* for an ADU or DADU, the CABAC CPMs only (E.7.3.1).

The entropy parsing state shall be recorded and restored independently according to DU type (ADU and DADU versus GDU and DGDU) and for each different value of ADU AttrIdx. For example, a coded point cloud sequence with num\_attributes equal to 2 would require storage for three sets of entropy parsing state.

At the start of any GDU with slice\_entropy\_continuation equal to 0 or inter\_entropy\_continuation\_enabled equal to 0, all previously recorded GDU, ADU, DGDU and DADU entropy parsing state shall be discarded.

* + - 1. Geometry data units
         1. Memorization

The GDU or DGDU entropy parsing state shall be recorded at:

* the end of every dependent\_geometry\_data\_unit syntax structure (E.3.1.3.4) when subgroup\_context\_reference\_indication\_enabled equals to 1 and layer\_group\_id is less than num\_layer\_groups\_minus1.

When fgs\_layer\_group\_enabled equals to 1, memorization shall record the elements and values of the GDU or DGDU entropy parsing state for restoration by the restoration process (E.7.4.2.2). When subgroup\_context\_reference\_indication\_enabled equals to 1 and layer\_group\_id is less than num\_layer\_groups\_minus1, the state shall be recorded to the subgroup array context SubgroupContexts[ layerGroupIdx ][ subgroupIdx ] of layer-group index layerGroupIdx equals to layer\_group\_id and subgroup index subgroupIdx equals to subgroup\_id.

SubgroupContexts[ layer\_group\_id ][ subgroup\_id ] = Contexts

1. A decoder may release SubgroupContexts[ layerGroupIdx ][ subgroupIdx ] during decoding process of fine granularity slices. For the context buffer release, SubgroupContextsCounter[ layerGroupIdx ][ subgroupIdx ] may defined as an integer counter of SubgroupContexts[ layerGroupIdx ][ subgroupIdx ]. SubgroupContextsCounter is initialized by the number of subsequent subgroups referencing to the context state of the current subgroup. When GDU or DGDU entropy parsing state identified by the subgroup context reference identifiers ref\_layer\_group\_id and ref\_subgroup\_id is restored, SubgroupContextsCounter[ layerGroupIdx ][ subgroupIdx ] is decreased by 1. When the integer counter SubgroupContextsCounter[ layerGroupIdx ][ subgroupIdx ] is equal to 0, the corresponding context state SubgroupContexts[ layerGroupIdx ][ subgroupIdx ] may be released from the context buffer.
   * + - 1. Restoration

The GDU entropy parsing state shall be restored at:

* the start of a geometry\_data\_unit syntax structure (7.3.3.1) when slice\_entropy\_continuation is 1; and
* the start of a geometry\_data\_unit syntax structure (7.3.3.1) when slice\_entropy\_continuation is 0, slice\_inter\_prediction is 1 and inter\_entropy\_continuation\_enabled is 1; and
* the start of every occupancy\_tree\_level( dpth ) syntax structure (7.3.3.5) where dpth is greater than OcctreeEntropyStreamDepth.

The GDU or DGDU entropy parsing state shall be restored at:

* the start of a dependent\_geometry\_data\_unit syntax structure (E.3.1.3.4).

When fgs\_layer\_group\_enabled equals to 1, restoration shall restore the elements and values of the GDU entropy parsing state to those previously recorded by the memorization process (E.7.4.2.1). At the start of a geometry\_data\_unit syntax structure, restoration shall exclude the planar occupancy coding state.

Restoration shall restore the elements and values of the DGDU entropy parsing state to those previously recorded by the memorization process (E.7.4.2.1) of the layer-group index ref\_layer\_group\_id and subgroup index ref\_subgroup\_id.

* + - 1. Attribute data units
         1. Memorization

The ADU entropy parsing state shall be recorded at :

* the end of every attribute\_data\_unit syntax structure (7.3.4.1).

When fgs\_layer\_group\_enabled is 1, the DADU entropy parsing state shall be recorded at :

* the end of every dependent\_attribute\_data\_unit syntax structure (E.3.1.4.2) when attr\_subgroup\_context\_reference\_indication\_enabled equals to 1 and dadu\_layer\_group\_id is less than num\_layer\_groups\_minus1.

Memorization shall record the elements and values of the ADU entropy parsing state for restoration by the restoration process (11.6.3.2). The state shall be recorded separately for each value of AttrIdx. When fgs\_layer\_group\_enabled is 1, memorization shall record the elements and values of the DADU entropy parsing state for restoration by the restoration process (E.7.4.3.2). The state shall be recorded separately for each value of AttrIdx, dadu\_layer\_group\_id and dadu\_subgroup\_id.

When attr\_subgroup\_context\_reference\_indication\_enabled equals to 1 and dadu\_layer\_group\_id is less than num\_layer\_groups\_minus1, the state shall be recorded to the subgroup array context for attribute SubgroupContextsForAttributes[ AttrIdx ][ layerGroupIdx ][ subgroupIdx ] of layer-group index layerGroupIdx equals to dadu\_layer\_group\_id and subgroup index subgroupIdx equals to dadu\_subgroup\_id.

SubgroupContextsForAttributes[AttrIdx][ dadu\_layer\_group\_id ][ dadu\_subgroup\_id ] = Contexts

* + - * 1. Restoration

The ADU entropy parsing state shall be restored at :

* the start of each attribute\_data\_unit syntax structure (7.3.4.1) when either slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1. The restoration shall be from the state recorded by the memorization process (11.6.3.1) with the same value of AttrIdx.

When fgs\_layer\_group\_enabled is 1, the DADU entropy parsing state shall be restored at :

* the start of each dependent\_attribute\_data\_unit syntax structure (E.3.1.4.2). The restoration shall be from the state recorded by the memorization process (E.7.4.3.1) which is indicated by AttrIdx, ref\_layer\_group\_id and ref\_subgroup\_id.

When fgs\_layer\_group\_enabled equals to 1, restoration shall restore the elements and values of the ADU or DADU entropy parsing state to those previously recorded by the memorization process (E.7.4.3.1) of the layer-group index attr\_ref\_layer\_group\_id, subgroup index attr\_ref\_subgroup\_id, and the same value of AttrIdx.

Contexts = SubgroupContextsForAttributes[AttrIdx][ attr\_ref\_layer\_group\_id ][attr\_ref\_ subgroup\_id ]

* 1. Partial decoding
     1. General

A decoder may decode and reconstruct part of fine granularity slices. A slice which is segmented by fine granularity slices supports partial decoding in terms of density (E.8.2) and/or spatial region (E.8.3). When partial decoding is used, the unnecessary data units to produce the partial output shall be filtered before decoding occupancy tree or attribute coefficients.

* + 1. Partial density decoding
       1. General

A decoder shall generate a lower density slice point cloud.

The lower density FGS point cloud is specified in terms of the following variables:

* The variable SkippedLayerGroup, an application-specific number of skipped layer-groups for partial decoding in the direction of the density. The value of SkippedLayerGroup shall be in the range 0 .. num\_layer\_groups\_minus1.
* The variable MinNodeSizeLog2, a minimum occupancy tree node size that is specified by the SkippedLayerGroup.
* The arrays SubgroupNodePos[ layerGroupIdx ][ subgroupIdx ][ ptIdx ][ 𝑘 ], the subgroup output nodes of the layer-group index layerGroupIdx and the subgroup index subgroupIdx.
* The arrays SubgroupNodeCnt[ layerGroupIdx ][ subgroupIdx ], the number of nodes in the subgroup output nodes of the layer-group index layerGroupIdx and the subgroup index subgroupIdx
  + - 1. Selection of FGS

If SkippedLayerGroup is greater than 0, the layer-groups whose index is in the range 0 .. OutLayerGroup is selected to be decoded. The maximum value of the layer-group index of partial decoding OutLayerGroup is specified by the total number of layer-groups minus SkippedLayerGroup.

OutLayerGroup := num\_layer\_groups\_minus1 – SkippedLayerGroup  
  
if (layer\_group\_id == 0)  
 … /\*decode GDU or ADU\*/  
else if (layer\_group\_id ≤ OutLayerGroup)  
 … /\*decode DGDU or DADU\*/  
else  
 … /\*skip DGDU or DADU\*/

Consequently, the depth of the geometry occupancy tree of the partial decoding PartialDepth is inferred by the sum of number of layers in each layer-group whose index is in the range 0 .. OutLayerGroup.

PartialDepth = 0  
for (i=0; I ≤ OutLayerGroup; i++)  
 PartialDepth += num\_layers\_minus1[i] + 1

* + - 1. Geometry position compensation

The maximum depth of the geometry occupancy tree when decoding all layer-groups is inferred by the sum of number of layers in each layer-group whose index is in the range 0 .. num\_layer\_groups\_minus1.

TotalDepth = 0  
for (i=0; i< num\_layer\_groups\_minus1; i++)  
 TotalDepth += num\_layers\_minus1[i] + 1

The MinNodeSizeLog2 is inferred by the difference between occtreeMaxDepthMinus1 and PartialDepth.

MinNodeSizeLog2 = occtreeMaxDepthMinus1 + 1 - PartialDepth

If MinNodeSizeLog2 is greater than 1, points shall be centred within their corresponding block:

for (ptIdx = 0; ptIdx < SubgroupNodeCnt[ layerGroupIdx ][ subgroupIdx ]; ptIdx++)  
 for (k = 0; k < 3; k++)  
 SubgroupNode[ layerGroupIdx ][ subgroupIdx ][ ptIdx ][ 𝑘 ] |= (MinNodeSizeLog2 > 1) << (MinNodeSizeLog2 – 1)

* + 1. Partial region decoding
       1. General

A decoder shall generate the point cloud of a partial region of a slice.

The partial region FGS point cloud is specified in terms of the following variables:

* The arrays RoiBBoxMin and RoiBBoxMax, an application-specific array which specifies the region of interest as the minimum and the maximum position of the bounding box.
* The arrays SubgroupNodePos[ layerGroupIdx ][ subgroupIdx ], the subgroup output nodes of the layer-group index layerGroupIdx and the subgroup index subgroupIdx.
* The arrays SubgroupNodeCnt[ layerGroupIdx ][ subgroupIdx ], the number of nodes in the subgroup output nodes of the layer-group index layerGroupIdx and the subgroup index subgroupIdx
  + - 1. Selection of FGS

If RoiBBoxMin and RoiBBoxMax are present, the subgroups whose subgroup bounding box is overlapped with the bounding box of region of interest is selected to be decoded.

if (layerGroupIdx == 0)  
 … /\*decode GDU or ADU\*/  
else if ((RoiBBoxMin[0] < SubgroupBBoxMax[layerGroupIdx][subgroupIdx][0] &&   
 RoiBBoxMin[1] < SubgroupBBoxMax[layerGroupIdx][subgroupIdx][1] &&  
 RoiBBoxMin[2] < SubgroupBBoxMax[layerGroupIdx][subgroupIdx][2]) &&  
 (RoiBBoxMax[0] > SubgroupBBoxMin[layerGroupIdx][subgroupIdx][0] &&  
 RoiBBoxMax[1] > SubgroupBBoxMin[layerGroupIdx][subgroupIdx][1] &&  
 RoiBBoxMax[2] > SubgroupBBoxMin[layerGroupIdx][subgroupIdx][2]))  
 … /\*decode DGDU or DADU\*/  
else  
 … /\*skip DGDU or DADU\*/

Bibliography

1. Society of Motion Picture and Television Engineers ST 2042-1 (2017), *VC-2 Video Compression*

Index of named expressions and variables

AeBits, 11.3.4

AeBitsReadIdx, 11.3.4

AeReadBin, 11.5.2

AngPosScaleK, 9.2.14.6

AngularEligible, 9.2.13.7.2

AngularOrigin, 7.4.3.2

AttrBitDepth, 7.4.2.1.3

AttrCoeff, 10.3.1.1

AttrDim, 7.4.2.1.3

AttrIdx, 7.4.4.2, 7.4.5

AttrLevelScale, 10.7.4

AttrMaxVal, 7.4.2.1.3

AttrPos, 10.2.1

AttrPosAng, 10.2.2

AttrQp, 10.7.3

AttrQpLayerOffset, 10.7.1

AttrQstep, 10.7.4

AttrRegionQpOffset, 10.7.2

AttrRegionQpOrigin, 10.7.1

AttrRegionQpSize, 10.7.1

AttrRegionSizeConstraint, 10.7.1

AxisCoded, 9.2.6.6

AzimuthAxis, 9.2.13.3

AzimuthAxisIsS/T, 9.2.13.3

BeamElev, 7.4.2.5.2

BeamIdxEst, 9.2.13.6

BeamIdxFromGrad, 9.2.13.6

BeamMinDeltaGrad, 9.2.13.7.2

BeamOffsetV, 7.4.2.5.2

BeamStepsPerRev, 7.4.2.5.2

BiasedNorm1, 10.6.6.5

BiasedNorm2, 10.6.6.12

BinIdxPfx, 11.4.3, 11.4.4

BinIdxSfx, 11.4.3, 11.4.4

BinIdxTu, 11.4.4, 11.4.5

BlkSizeLog2, 10.6.5.6, 10.6.5.8, 10.6.6.6

BpBits, 11.3.4

BpBitsReadIdx, 11.3.4

CanonicalLodSubsampling, 10.6.5.5

CathetusV, 9.2.13.7.2

ChildNodeSizeLog2, 9.2.6.4

ChunkBuf, 11.3.4

ChunkDuRem, 11.3.5

ChunkLen, 11.3.6

ChunkNextAeBit, 11.3.9

ChunkNextBpBit, 11.3.10

ChunkPadLen, 11.3.8, 11.3.11

ChunkPartBLen, 11.3.11

CloudAttrPos, 10.8.2

CodedAxisCnt, 9.2.6.6

CoeffIdx, 10.5.3.1

CoeffWeight, 10.6.4, 10.6.11.1

Contexts, 11.5.3.1

Ctx, 11.5.3.4

CtxIdx, 11.5.3.4

CtxIdxAngPhi, 9.2.13.9

CtxIdxAngTheta, 9.2.13.10

CtxIdxDictHg, 9.2.9.11

CtxIdxOccBit, 9.2.10.6.4

CtxIdxPlanePos, 9.2.11.7.1

CtxTbl, 11.5.3.4

CtxUpdateDelta, 11.5.3.3

DataUnitBytes, 11.2.2

DataUnitLength, 11.2.2

DataUnitReadIdx, 11.2.2

DictsHistogram, 9.2.9.4

DictsMostProb, 9.2.9.4

DictsMostProbAge, 9.2.9.4

DictsMostProbMaxAge, 9.2.9.4

DictsNextEvictIdx, 9.2.9.4

DictsRecent, 9.2.9.4

DirectMode1Eligible, 9.2.12.3.1

DirectMode2Eligible, 9.2.12.3.1

DirectMode3Eligible, 9.2.12.3.1

DirectModeEligible, 9.2.12.3.1

DirectNodePresent, 9.2.12.3.2

DnBeamIdx, 9.2.13.8.1

DnBeamIdxEst, 9.2.13.8.1

DnBitsAfterJointPrefix, 9.2.12.5.2

DnBitsAfterPlanar, 9.2.12.4.3

DnEligibleCnt, 9.2.12.3.3

DnJointCoded, 9.2.12.4.4

DnJointDiffBitInferred, 9.2.12.5.4

DnJointDiffBitPresent, 9.2.12.5.4

DnJointDiffBits, 9.2.12.5.3

DnJointPos, 9.2.12.4.4

DnJointPosDiffBit, 9.2.12.5.3

DnJointPosPrefix, 9.2.12.5.2

DnJointPrefixBits, 9.2.12.5.2

DnPartialPosAng, 9.2.13.8.2

DnPlanarPos, 9.2.12.4.3

DnPresenceMask, 9.2.12.3.2

DnPtPos, 9.2.12.4.1

DnPtPosAng, 9.2.13.8.2

DnPtPosAngS, 9.2.13.8.2

DnPtPosConstraint, 9.2.12.5.1

DnPtPosRem, 9.2.12.4.1

DnPtPosS/T/V, 9.2.12.4.1

DnQp, 9.2.14.5.2

DnRemAngBitsST, 9.2.13.8.3

DnRemAngBitsV, 9.2.13.8.3

DnRemBits, 9.2.12.4.5

DnRemPos, 9.2.12.4.5

Dpth, 9.2.6.1

DuFooterLen, 11.2.4

DuIsGdu, 11.2.4

DuNextBit, 11.2.5

ExpMovAvg, 9.2.11.5.3

FixedRahtFilters, 10.5.5.11

FrameCtr, 8.2.2

GduFooterLen, 11.2.4

IdxOfSubsampledPoint, 10.6.5.9

InLodPtCnt, 10.6.5.4

InLodPtIdx, 10.6.5.4

InterCompPredCoeff, 10.6.10.1

InterCompPredCoeffPrev, 10.6.10.1

IsSubsampledPoint, 10.6.5.7

IsReplaced, 10.5.5.4.3

IvlCarry, C.2

IvlCode, 11.5.4.2

IvlLow, 11.5.4.2, C.2

IvlRange, 11.5.4.2, C.2

LastCompPredCoeff, 10.6.10.1

LastCompPredCoeffPrev, 10.6.10.1

LcpScale, 10.5.5.12

Length, 7.2

LodCnt, 10.6.4

LodCoeffQp, 10.6.9.1

LodPtCnt, 10.6.4

LodPtIdx, 10.6.4

LodRfmtPtCnt, 10.6.4

LodRfmtPtIdx, 10.6.4

Lvl, 10.5.3.1, 10.6.5.1

MapPtIdx, 10.6.5.6

MaxPtsPerSort, 10.6.5.2

MapSub, 10.6.5.6

MaxSeqBboxDimLog2, A.4.1

MaxSliceDimLog2, A.4.1

MaxSlicePoints, A.4.1

MinAng, 10.2.2

MinCurAng, 10.2.2

MinInterRefLvl, 10.6.6.2

MinNodeSizeLog2, D.2.1

MinRefAng, 10.2.2

Morton, 5.10.7

NextAeStreamBit, 11.5.4.4

Nloc, 9.2.6.3

NlocC, 9.2.6.3

NlocG, 9.2.6.3

NlocP, 9.2.6.3

NodeBeamIdx, 9.2.13.7.4

NodeBeamValid, 9.2.13.7.4

NodeIdx, 9.2.6.1

NodeOccMap, 9.2.11.5.2

NodeSizeLog2, 9.2.6.4

Norm2, 10.6.6.13

NposAng, 9.2.13.2

NposAngMid, 9.2.13.7.1

NposAngMidS/T/V, 9.2.13.7.1

NposAngS/T/V, 9.2.13.2

Ns/Nt/Nv, 9.2.6.3

NsC/NtC/NvC, 9.2.6.3

NsG/NtG/NvG, 9.2.6.3

NsP/NtP/NvP, 9.2.6.3

NumRefNodes, E.6.3.7.1

ObufCtxIdxDelta, 12.4.1.1

ObufInitProba, 12.2.2

ObufInitProbaBounds, 12.2.2

OccAdj, 9.2.10.6.9.3

OccAdjCnt, 9.2.10.6.9.3

OccAvailWinLog2, 9.2.7.2

OccBitCodingOrder, 9.2.10.2

OccBitIdx, 9.2.10.2

OccBitIdxFromNpBit, 9.2.8

OccBitInferSet, 9.2.10.5

OccBitInferUnset, 9.2.10.4

OccBitLocC, 9.2.10.2

OccBitMap, 9.2.10.2

OccBitPresent, 9.2.10.3

OccChildCnt, 9.2.6.11

OccCtxSel, 9.2.10.6.2

OccCtxSelUpdate, 9.2.10.6.5

OccCtxSelVar, 9.2.10.6.4

OccDictHistogram, 9.2.9.6

OccDictMostProb, 9.2.9.6

OccDictMostProbAge, 9.2.9.6

OccDictMostProbMaxAge, 9.2.9.6

OccDictMostProbMaxAgeNext, 9.2.9.9

OccDictNextEvictIdx, 9.2.9.6

OccDictRecent, 9.2.9.6

OccFreeAxis, 9.2.6.6

OccFreeAxisCnt, 9.2.6.6

OccFromNpOcc, 9.2.8

OccIntraPred, 9.2.10.6.9.3

OccKnown, 9.2.10.5

OccKnownMask, 9.2.10.5

OccKnownZero, 9.2.10.5

OccLocC, 9.2.2.3

OccLvlNodeSizeLog2, 9.2.4

OccMapPresent, 9.2.6.9

OccMaxChildren, 9.2.6.7

OccMaybeSingleChild, 9.2.6.8

OccMinChildren, 9.2.6.7

OccNeigh, 9.2.7.3

OccNeighAvail, 9.2.7.2

OccNeighCnt, 9.2.10.6.9.3

OccNeighPat, 9.2.7.4

OccNeighPatEq0, 9.2.12.3.3

OccNeighPatR, 9.2.7.5, 9.2.10.6.6

OccNodeChildCnt, 9.2.12.3.3

OccNodeCnt, 9.2.5.1

OccNodeLoc, 9.2.5.1

OccNodePresent, 9.2.5.1

OccPlaneMask, 9.2.11.5.5

OccPosScaleK, 9.2.14.6

OccQp, 9.2.14.5.1

OccQpScaleLog2, 9.2.14.2.3

OccQpSubtreeDepth, 9.2.14.4

OccScalingVolSizeLog2, 9.2.14.2.2

OccSubtreeQp, 9.2.14.5.3

OccSubtreeQpLoc, 9.2.14.5.3

OccSubtreeQpMax, 9.2.14.5.4

OccSubtreeQpOffset, 9.2.14.3

OcctreeEntropyStreamDepth, 9.2.3

OccupancyDensity, 9.2.11.5.2

OccupancyMapP, 9.2.11.5.5

OccVolScale, 9.2.14.2.3

OutLodPtCnt, 10.6.5.4, 10.6.5.6

OutLodPtIdx, 10.6.5.4

OutRfmtPtIdx, 10.6.5.4

PartialPtCnt, D.2.1

PartialPtIdx, D.2.1

PartVal, 11.4.5

PartValue, 11.4.1

PlanarBeamIdx, 9.2.13.7.3

PlanarEligible, 9.2.11.5.1

PlanarEligibleByAng, 9.2.11.5.1

PlanarEligibleByRate, 9.2.11.5.1

PlanarFreeAxis, 9.2.11.4

PlanarMinChildren, 9.2.11.3

PlanarMinPlanes, 9.2.11.2

PlanarNodeAxisLoc, 9.2.11.6.2

PlanarNodeZone, 9.2.11.6.3

PlanarRate, 9.2.11.5.2

PlanarRateThreshold, 9.2.11.5.1

PlaneThresholdIdx, 9.2.11.5.1

PointAng, 8.3.2

PointAttr, 8.3.5

PointCnt, 8.3.2

PointPos, 8.3.3

PredBias, 7.4.2.6.3

PredCnt, 10.6.4

PredMode, 10.6.8.1

PredModeCoded, 10.6.8.3, 10.6.8.4

PredModeMax, 10.6.8.1

PredModePresent, 10.6.8.2

PredPointCloudBboxOrigin, 9.2.15.1

PredPointCloudBboxSize, 9.2.15.1

PredPtIdx, 10.6.4

PredPtRef, 10.6.4

PredWeight, 10.6.4

PrevOccPlanePos, 9.2.11.6.4

PrevOccSinglePlane, 9.2.11.6.4

PrevPlanarNodeZone, 9.2.11.6.4

PtnAng, 9.3.4.3

PtnAngStv, 9.3.4.5

PtnCnt, 9.3.3.1

PtnDepth, 9.3.3.1

PtnIdx, 9.3.3.1

PtnPhiMul, 9.3.2.2

PtnPos, 9.3.4.2

PtnPred, 9.3.4.6

PtnQp, 9.3.2.2

PtnQpInterval, 9.3.2.1

PtnQpOffset, 9.3.2.2

PtnRef, 9.3.4.6

PtnResidual, 9.3.2.2

PtnSecResidual, 9.3.2.2

PtnStack, 9.3.3.1

QuantizedChildNodeSizeLog2, 9.2.14.2.3

QuantizedNodeSizeLog2, 9.2.14.2.3

QuantWeight, 10.6.4, 10.6.13.1

RahtBlkCnt, 10.5.2.2

RahtBlkChildCnt, 10.5.5.2

RahtBlkLoc, 10.5.2.2

RahtBlkSumAttrRef, 10.5.2.6

RahtBlkWeight, 10.5.2.3

RahtBlkWeightRef, 10.5.2.6

RahtCoeff, 10.5.2.2

RahtCoeffLcp, 10.5.5.12

RahtCoeffOrder, 10.5.3.2

RahtCoeffQp, 10.5.4.3

RahtCoeffScaled, 10.5.4.2

RahtCoeffWeight, 10.5.2.5

RahtCoeffWeightM, 10.5.2.5

RahtCoeffWeightMRef, 10.5.2.6

RahtDcCoeff, 10.5.6.2

RahtPredDcCoeff, 10.5.7.2

RahtDcNorm, 10.5.5.4.1

RahtFwd, 10.5.5.7

RahtInterFilter, 10.5.5.11

RahtInterPred, 10.5.5.9

RahtInterPredBlk, 10.5.5.1

RahtInterPredEligible, 10.5.5.8

RahtInterPredW, 10.5.5.9

RahtInv, 10.5.6.5

RahtLvlCnt, 10.5.2.2

RahtNeighCnt, 10.5.5.3

RahtNeighCntMinAncestor, 10.5.5.3

RahtPred, 10.5.5.4.4

RahtPredBlk, 10.5.5.1

RahtPredEligible, 10.5.5.2

RahtPredExcluded, 10.5.5.4.2

RahtPredRecipW, 10.5.5.4.4

RahtPredRecipWC, 10.5.5.4.5

RahtPredW, 10.5.5.5

RahtPredWeight, 10.5.5.4.4

RahtPredWeightB, 10.5.5.4.5

RahtPredWeightC, 10.5.5.4.5

RahtRecon, 10.5.7

RahtRootLvl, 10.5.2.4

RahtTreeQpOffset, 10.5.4.4

RahtTreeQpOffsetM, 10.5.4.4

RawAttrValueBits, 10.3.1.3

RecCloudAttr, 6.5.6

RecCloudPointCnt, 6.5.6

RecCloudPos, 6.5.6

RefAttrPos, 10.8.1

RefCloudAttrPos, 10.8.3

RefCloudAttrPosFirst, 10.8.3.2

RefCloudAttrPosSecond, 10.8.3.2

RefCloudAttr, 10.8.3

RefCloudAttrBboxOrigin, 10.8.3

RefCloudAttrFirst, 10.8.3.2

RefCloudAttrSecond, 10.8.3.2

RefCloudAttrBboxSize, 10.8.3

RefLodPtIdx, 10.6.5.10

RefPointAttr, 10.8.1

ScaledHalfNodeSize, 9.2.13.7.1

ScaledHalfNodeSizeS/T/V, 9.2.13.7.1

ScaledNodeSize, 9.2.13.7.1

ScaledNodeSizeS/T/V, 9.2.13.7.1

SelAdj, 9.2.10.6.8

SeCtx, 12.5

SelNeigh, 9.2.10.6.6

SelPred, 9.2.10.6.9.1

SelPredEligible, 9.2.10.6.9.2

SelSib, 9.2.10.6.7

SeqCodedScale, 7.4.2.1.2

SeqOrigin, 7.4.2.1.2

SeqUnit, 7.4.2.1.2

SliceOrigin, 7.4.3.2

StvToXyz, 7.4.2.1.2

SubgroupBBoxMin, E.4.2.3

SubgroupContexts, E.7.3.1.1

SubgroupContextsCounter, E.7.4.2.1

SubgroupContextsForAttributes, E.7.3.1.1

SubgroupDirectNodePointCnt, E.4.2.3

SubgroupNodeAttr, E.4.2.4

SubgroupNodeCnt, E.4.2.3

SubgroupNodePos, E.4.2.3

SubgroupOccNeighPatEq0, E.4.2.3

SubgroupOccNodeChildCnt, E.4.2.3

SumN19, 10.5.5.3

SumN26, 9.2.10.6.9.3

TerminalNode, 9.2.6.5

TileInventoryOrigin, 7.4.2.4

ZeroRunLength, 10.3.1.1