 ISO/IEC JTC 1/SC 29/WG 7 N0924

**ISO/IEC JTC 1/SC 29/WG 7  
MPEG 3D Graphics and Haptics Coding   
Convenorship: AFNOR (France)**

**Document type:** Output Document

**Title:** Text of ISO/IEC 23090-30 CD Low complexity low latency LIDAR coding

**Status:** Approved

**Date of document:** 2024-04-17

**Source:** ISO/IEC JTC 1/SC 29/WG 7

**Expected action:** None

**Action due date:** None

**No. of pages:** 160 (with cover page)

**Email of Convenor:** marius.preda @ imt . fr

**Committee URL:** [https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg7](https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3)

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 7 MPEG Coding for 3D Graphics and haptics**

**ISO/IEC JTC 1/SC 29/WG 7 N0924**

**April 2024, Rennes**

|  |  |
| --- | --- |
| **Title** | **Text of ISO/IEC 23090-30 CD Low complexity low latency LIDAR coding** |
| **Status** | **Approved** |
| **Source** | **WG 7, MPEG Coding for 3D Graphics and Haptics** |
| **Editors** | **Mary-Luc Champel, Gao Shuo** |
| **Serial Number** | **24022** |

**Abstract**

The first committee draft (CDv1.0) of MPEG-I Part 30 for low latency, low complexity LiDAR coding (L3C2) is hereby established.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ed. Notes (WD1):

• Edition started from G-PCCv1 FDIS w19617\_d23e (see m55637 container for spec edition).

• Removed any part related to occupancy tree.

• Put back entropy stream from earlier version of occupancy tree and that are currently used by L3C2 for partitioning the bitstream between sensor heads’ geometry sensing and low latency attributes.

• Temporarily kept most of predictive tree decoding process, because some parts may potentially be reused with L3C2.

• Integrated high-level syntax (HLS) of parameter sets and geometry data unit header (syntax and semantics)

Ed. Notes (WD2):

• Edition continued as tracked changes over G-PCCv1 FDIS w19617\_d23e.

• Integrated specification for reconstruction of

of coarse representation decoding (entropy coding contexts and maintained variables)

Ed. Notes (WD3):

• Edition continued as tracked changes over G-PCCv1 FDIS w19617\_d23e.

• Integrated specification for decoding and reconstruction of point position

• Integrated specification for decoding and reconstruction of low latency attributes

Ed. Notes (draft of CD)

• Terms and definitions updated with new terms specific to L3C2.

• Unused terms and definitions from G-PCC are removed.

• Unused sections from G-PCC are removed.

• Addition of syntax and semantic for L3C2 geometry data unit (points geometry and low latency attributes).

• Addition of decoding process/Slice geometry for L3C2.

• Addition of section on dynamic OBUF.

• Update on parsing process for L3C2 (predictive tree contexts removed, L3C2 contexts added).

• Update on Annexes (Profiles and Levels).

**ISO/IEC CD 23090-30:2024(E)**

ISO/IEC JTC 1/SC 29/WG 7

Secretariat: JISC

**Information technology — Coded representation of immersive media — Part 30: Low latency, low complexity LiDAR coding**

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 2022

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 viii](#_Toc168435878)

[Introduction ix](#_Toc168435879)

[1 Scope 9](#_Toc168435880)

[2 Normative references 9](#_Toc168435881)

[3 Terms and definitions 9](#_Toc168435882)

[3.1 General terms 10](#_Toc168435883)

[3.2 High-level syntax and entropy coding terms 12](#_Toc168435884)

[3.3 Tree structure terms 15](#_Toc168435885)

[3.4 Geometry coding terms 15](#_Toc168435886)

[3.5 Attribute coding terms 15](#_Toc168435887)

[4 Abbreviated terms 16](#_Toc168435888)

[4.1 Acronyms 16](#_Toc168435889)

[4.2 Mnemonics 16](#_Toc168435890)

[5 Conventions 17](#_Toc168435891)

[5.1 General 17](#_Toc168435892)

[5.2 Symbolic names 17](#_Toc168435893)

[5.3 Numerical representation 17](#_Toc168435894)

[5.4 Arithmetic operators 17](#_Toc168435895)

[5.5 Logical operators 18](#_Toc168435896)

[5.6 Relational operators 18](#_Toc168435897)

[5.7 Bit-wise operators 18](#_Toc168435898)

[5.8 Assignment operators 19](#_Toc168435899)

[5.9 Range notation 19](#_Toc168435900)

[5.10 Mathematical functions 19](#_Toc168435901)

[5.10.1 General 19](#_Toc168435902)

[5.10.2 IntCos and IntSin 20](#_Toc168435903)

[5.10.3 IntSqrt 20](#_Toc168435904)

[5.10.4 IntRecipSqrt 21](#_Toc168435905)

[5.10.5 Div 22](#_Toc168435906)

[5.10.6 Morton 22](#_Toc168435907)

[5.10.7 FromMorton 22](#_Toc168435908)

[5.11 Order of operation precedence 23](#_Toc168435909)

[5.12 Named expressions 23](#_Toc168435910)

[5.12.1 General 23](#_Toc168435911)

[5.12.2 Scope 24](#_Toc168435912)

[5.12.3 Arguments of named expressions 24](#_Toc168435913)

[5.12.4 Sub-expressions 25](#_Toc168435914)

[5.12.5 Definitions with multiple statements 25](#_Toc168435915)

[5.12.6 Textual definitions 25](#_Toc168435916)

[5.13 Variables, syntax elements and tables 25](#_Toc168435917)

[6 Point cloud format and relationship to coded and output representations 26](#_Toc168435918)

[6.1 General format 26](#_Toc168435919)

[6.2 Attributes 26](#_Toc168435920)

[6.2.1 General 26](#_Toc168435921)

[6.2.2 Colour 26](#_Toc168435922)

[6.2.3 Opacity 27](#_Toc168435923)

[6.2.4 Reflectance 27](#_Toc168435924)

[6.2.5 Normal vector 27](#_Toc168435925)

[6.2.6 Material identifier 27](#_Toc168435926)

[6.2.7 Frame number/index 27](#_Toc168435927)

[6.2.8 User defined attributes 28](#_Toc168435928)

[6.3 Codec-derived attributes 28](#_Toc168435929)

[6.3.1 General 28](#_Toc168435930)

[6.3.2 Slice identifier 28](#_Toc168435931)

[6.3.3 Slice tag 28](#_Toc168435932)

[6.3.4 Canonical point order 28](#_Toc168435933)

[6.3.5 Point Morton order 28](#_Toc168435934)

[6.4 Coded point cloud format 29](#_Toc168435935)

[6.4.1 Sequence coordinate system 29](#_Toc168435936)

[6.4.2 Coding coordinate system 30](#_Toc168435937)

[6.4.3 Coded point cloud sequence 31](#_Toc168435938)

[6.4.4 Coded point cloud frame 31](#_Toc168435939)

[6.4.5 Slice of a coded point cloud frame 31](#_Toc168435940)

[6.4.6 Repetition of slices 32](#_Toc168435941)

[6.4.7 Relationship between tiles and slices 32](#_Toc168435942)

[6.4.8 Parameter sets 33](#_Toc168435943)

[6.5 Output point cloud format 34](#_Toc168435944)

[6.5.1 General 34](#_Toc168435945)

[6.5.2 Coordinate system 34](#_Toc168435946)

[6.5.3 Fixed-point conformance output 34](#_Toc168435947)

[6.5.4 Attributes 34](#_Toc168435948)

[6.5.5 Output point cloud sequence 34](#_Toc168435949)

[6.5.6 Output point cloud frame 34](#_Toc168435950)

[7 Syntax and semantics 35](#_Toc168435951)

[7.1 Method of specifying syntax in tabular form 35](#_Toc168435952)

[7.2 Specification of syntax functions and descriptors 36](#_Toc168435953)

[7.3 Syntax in tabular form 36](#_Toc168435954)

[7.3.1 General 36](#_Toc168435955)

[7.3.2 Parameter sets, ancillary data and byte alignment 36](#_Toc168435956)

[7.3.3 Geometry data unit 43](#_Toc168435957)

[7.3.4 Attribute data unit 46](#_Toc168435958)

[7.3.5 Defaulted attribute data unit syntax 48](#_Toc168435959)

[7.4 Semantics 49](#_Toc168435960)

[7.4.1 General 49](#_Toc168435961)

[7.4.2 Parameter sets, ancillary data and byte alignment 49](#_Toc168435962)

[7.4.3 Geometry data unit 60](#_Toc168435963)

[7.4.4 Attribute data unit 63](#_Toc168435964)

[7.4.5 Defaulted attribute data unit semantics 63](#_Toc168435965)

[8 Decoding process 63](#_Toc168435966)

[8.1 General decoding process 63](#_Toc168435967)

[8.2 Frame decoding processes 63](#_Toc168435968)

[8.2.1 General 63](#_Toc168435969)

[8.2.2 Frame counter 64](#_Toc168435970)

[8.3 Slice decoding processes 64](#_Toc168435971)

[8.3.1 General 64](#_Toc168435972)

[8.3.2 State variables 64](#_Toc168435973)

[8.3.3 Geometry decoding process 64](#_Toc168435974)

[8.3.4 Default attribute values 65](#_Toc168435975)

[8.3.5 Attribute decoding process 65](#_Toc168435976)

[8.3.6 At the end of a slice 65](#_Toc168435977)

[9 Slice geometry 65](#_Toc168435978)

[9.1 General 65](#_Toc168435979)

[9.2 Sensed point sequence 65](#_Toc168435980)

[9.2.1 General 65](#_Toc168435981)

[9.2.2 State variables and expressions 65](#_Toc168435982)

[9.2.3 Syntax element semantics 66](#_Toc168435983)

[9.2.4 Contextualization of elevation\_offset 69](#_Toc168435984)

[9.2.5 Reconstruction of sensed point positions 72](#_Toc168435985)

[9.2.6 Reconstructed output point positions 81](#_Toc168435986)

[9.2.7 Reconstructed output low latency attributes values 81](#_Toc168435987)

[9.2.8 Reconstructed point positions for attributes decoding 81](#_Toc168435988)

[10 Slice attributes 82](#_Toc168435989)

[10.1 General 82](#_Toc168435990)

[10.2 Point coordinates 82](#_Toc168435991)

[10.2.1 General 82](#_Toc168435992)

[10.2.2 Conversion to scaled angular coordinates 82](#_Toc168435993)

[10.3 Syntax element semantics 82](#_Toc168435994)

[10.4 Raw attribute decoding 83](#_Toc168435995)

[10.5 Attribute decoding using the region-adaptive hierarchical transform 83](#_Toc168435996)

[10.5.1 General 83](#_Toc168435997)

[10.5.2 Transform tree 84](#_Toc168435998)

[10.5.3 Coefficient order 86](#_Toc168435999)

[10.5.4 Coefficient scaling 87](#_Toc168436000)

[10.5.5 Transform domain prediction 89](#_Toc168436001)

[10.5.6 Inverse transform 93](#_Toc168436002)

[10.5.7 Reconstructed attribute values 95](#_Toc168436003)

[10.6 Attribute decoding using levels of detail 95](#_Toc168436004)

[10.6.1 General 95](#_Toc168436005)

[10.6.2 Syntax element semantics 96](#_Toc168436006)

[10.6.3 Reconstruction process 96](#_Toc168436007)

[10.6.4 State variables 96](#_Toc168436008)

[10.6.5 Levels of detail 97](#_Toc168436009)

[10.6.6 Predictor search 103](#_Toc168436010)

[10.6.7 Reconstruction of attribute values 109](#_Toc168436011)

[10.6.8 Prediction mode coding 110](#_Toc168436012)

[10.6.9 Scaling 112](#_Toc168436013)

[10.6.10 Coefficient prediction 112](#_Toc168436014)

[10.6.11 Transform coefficient weights 113](#_Toc168436015)

[10.6.12 Transform 114](#_Toc168436016)

[10.7 Attribute quantization parameters 114](#_Toc168436017)

[10.7.1 Syntax element semantics 114](#_Toc168436018)

[10.7.2 Per-point regional QP offset 115](#_Toc168436019)

[10.7.3 Attribute coefficient QP 116](#_Toc168436020)

[10.7.4 Definition of *AttrQstep* 116](#_Toc168436021)

[11 Parsing process 116](#_Toc168436022)

[11.1 General 116](#_Toc168436023)

[11.2 Data unit buffer 118](#_Toc168436024)

[11.2.1 General 118](#_Toc168436025)

[11.2.2 State 118](#_Toc168436026)

[11.2.3 Initialization at the start of parsing a data unit 118](#_Toc168436027)

[11.2.4 Initialization at the start of parsing a geometry data unit footer 119](#_Toc168436028)

[11.2.5 Definition of *DuNextBit* 119](#_Toc168436029)

[11.3 Chunked bytestream parsing 119](#_Toc168436030)

[11.3.1 General 119](#_Toc168436031)

[11.3.2 Chunk syntax 119](#_Toc168436032)

[11.3.3 Chunk semantics 120](#_Toc168436033)

[11.3.4 State 120](#_Toc168436034)

[11.3.5 Span of chunked bytestream data within a data unit 120](#_Toc168436035)

[11.3.6 The chunk buffer 120](#_Toc168436036)

[11.3.7 State update at the start of every CBS 121](#_Toc168436037)

[11.3.8 Unpacking a single chunk 121](#_Toc168436038)

[11.3.9 Definition of *ChunkNextAeBit* 121](#_Toc168436039)

[11.3.10 Definition of *ChunkNextBpBit* 121](#_Toc168436040)

[11.3.11 Boundary between spliced chunked bytestreams 122](#_Toc168436041)

[11.3.12 Location of chunked bytestream boundaries 123](#_Toc168436042)

[11.4 General inverse binarization processes 123](#_Toc168436043)

[11.4.1 Parsing unsigned fixed-length codes (FL) 123](#_Toc168436044)

[11.4.2 Parsing signed fixed-length codes (FL+S) 123](#_Toc168436045)

[11.4.3 Parsing 𝑘-th order exp-Golomb codes (EGk) 123](#_Toc168436046)

[11.4.4 Parsing concatenated truncated unary and 𝑘-th order exp-Golomb codes (TU+EGk) 124](#_Toc168436047)

[11.4.5 Parsing concatenated bounded truncated unary and 𝑘-th order exp-Golomb codes (BTU+EGk) 124](#_Toc168436048)

[11.4.6 Parsing signed concatenated truncated unary and 𝑘-th order exp-Golomb codes (TU+EGk+S) 125](#_Toc168436049)

[11.4.7 Parsing signed concatenated bounded truncated unary and 𝑘-th order exp-Golomb codes (BTU+EGk+S) 125](#_Toc168436050)

[11.4.8 Parsing signed concatenated asymmetrically bounded truncated unary and 𝑘-th order exp-Golomb codes (ABTU+EGk+S) 125](#_Toc168436051)

[11.4.9 Parsing truncated unary codes (TU) 126](#_Toc168436052)

[11.4.10 Mapping process for signed codes 126](#_Toc168436053)

[11.4.11 Parsing ASN.1 object identifiers 127](#_Toc168436054)

[11.5 CABAC parsing processes 127](#_Toc168436055)

[11.5.1 Initialization 127](#_Toc168436056)

[11.5.2 Definition of *AeReadBin* 127](#_Toc168436057)

[11.5.3 Contextual probability models 128](#_Toc168436058)

[11.5.4 Arithmetic decoding engine 131](#_Toc168436059)

[11.6 Sensor Coding State 133](#_Toc168436060)

[11.6.1 State variable 133](#_Toc168436061)

[11.6.2 Initialization at the start of a GDU 136](#_Toc168436062)

[11.6.3 Initialization with decoded coarse position of first sensed point 138](#_Toc168436063)

[11.6.4 Update with azimuth\_offset 138](#_Toc168436064)

[11.6.5 Update while decoding elevation\_offset 139](#_Toc168436065)

[11.6.6 Update with sensed point coarse position 140](#_Toc168436066)

[11.7 Parsing state memorization and restoration 140](#_Toc168436067)

[11.7.1 General 140](#_Toc168436068)

[11.7.2 Geometry data units 141](#_Toc168436069)

[11.7.3 Attribute data units 141](#_Toc168436070)

[11.7.4 Defaulted attribute data units 141](#_Toc168436071)

[12 OBUF parsing process 141](#_Toc168436072)

[12.1 General 141](#_Toc168436073)

[12.2 Creation of an OBUF instance 142](#_Toc168436074)

[12.2.1 Creation and initialization of OBUF trees 142](#_Toc168436075)

[12.2.2 Creation and initialization of an array of OBUF ACPMs 143](#_Toc168436076)

[12.3 Call and update of an OBUF instance 143](#_Toc168436077)

[12.3.1 Decode and update according to OBUF trees 143](#_Toc168436078)

[12.4 Decode of a bin based on an OBUF ACPM 145](#_Toc168436079)

[Annex A (normative) Profiles and levels 146](#_Toc168436080)

[A.1 Overview of profiles and levels 146](#_Toc168436081)

[A.2 Requirements on decoder capability 146](#_Toc168436082)

[A.3 Profiles 146](#_Toc168436083)

[A.3.1 General 146](#_Toc168436084)

[A.3.2 Simple and Main profiles 146](#_Toc168436085)

[A.4 Levels 147](#_Toc168436086)

[A.4.1 Level limits 147](#_Toc168436087)

[A.5 Permitted ranges for syntax elements 147](#_Toc168436088)

[Annex B (normative) Type-length-value encapsulated bytestream format 151](#_Toc168436089)

[B.1 General 151](#_Toc168436090)

[B.2 Syntax and semantics 151](#_Toc168436091)

[B.2.1 Syntax 151](#_Toc168436092)

[B.2.2 Semantics 151](#_Toc168436093)

[B.3 Parsing process 152](#_Toc168436094)

[Annex C (informative) Arithmetic encoding engine 153](#_Toc168436095)

[C.1 General 153](#_Toc168436096)

[C.2 State variables 153](#_Toc168436097)

[C.3 Initial state 153](#_Toc168436098)

[C.4 Encoding process for a single binary symbol 153](#_Toc168436099)

[C.5 Arithmetic encoder state renormalization process 154](#_Toc168436100)

[C.6 Arithmetic encoding engine termination process 154](#_Toc168436101)

[Bibliography 155](#_Toc168436102)

[Index of named expressions and variables 156](#_Toc168436103)

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, and other industrial processes. Point clouds captured with LiDAR (Light Detection And Ranging) sensors, sometimes coupled with traditional camera sensors, 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 intensity, colour, timestamp, and classification. Such representations require a large amount of data, which can be costly in terms of storage and transmission and are often used in real-time application which requires fast processing of the point cloud data. This document provides the method for efficiently compressing point cloud representations while ensuring low complexity and low latency constraints can be met.

**Information technology — Coded representation of immersive media — Part 30: Low latency, low complexity LiDAR coding**

# Scope

This document specifies low latency, low complexity LiDAR coding.

# 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* (3.1.8) and zero or more *attributes* (3.1.19)

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)

canonical point order  
canonical decoding order

order of *points* (3.1.1) decoded from a *slice* (3.1.21) 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* (3.1.11) 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* (3.1.12) or scaled *RPI* (3.1.13) point coordinates used to code an attribute

XYZ (axes)

X, Y and Z axes, in that order, used to represent *Cartesian coordinates* (3.1.8)

STV (axes)

S, T and V axes, in that order, that are a sequence-dependent permutation of the *XYZ axes* (3.1.11); used to represent the coded *geometry* (3.1.18)

RPI (axes)

R, P and I axes, in that order, used to represent *angular coordinates* (3.1.9)

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* (3.1.14) 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* (3.1.15) that applies for a single *slice* (3.1.21), and in which all *points* (3.1.1) in the *slice* have non-negative integer coordinates

sensing 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* (3.1.21) identified by a common slice\_tag *syntax element* value (3.2.15) whose *geometry* (3.1.18) should be contained within a *bounding box* (3.1.7) specified in a tile inventory data unit

Morton code

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

Morton order

elements ordered according to their *Morton code* (3.1.23)

sparse array

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

**3.1.26**

**coarse position**

a coarse representation in 2D plane of position of each *point* (3.1.1) in a *point cloud* (3.1.2), used to represent the coded *geometry* (3.1.18)

**3.1.27**

**lexicographic order**

dictionary order used to order a list of points (3.1.1) by firstly in the azimuthal angle and secondly in the elevation angle of these points, and these points are decoded in the order

**3.1.28**

**late points**

points whose order is late according to the lexicographicorder due to the noise of data, or transmission delay

**3.1.29**

**Neighbour**

point whose coarse position is preceding coarse position of current point according to the lexicographic order

**3.1.30**

**Sensing heads**

LiDAR capture systems may include more than one sensor (sensing head) in order to cover a larger field of view, each sensing head being able to cover a limited azimuthal range.

**3.1.31**

**Sensing order index**

The order index, per head, in which points are captured by the sensing head.

## 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* (3.1.21) 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* (3.1.21) 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

geometry entropy stream

Stream of entropy coded sequence of points information containing for each head, geometry information,

low latency attributes entropy stream

Stream of entropy coded sequence of points information containing for each head, low-latency attributes

L3C2 entropy stream

Stream of entropy coded sequence of points information containing for each head, geometry information, low-latency attributes and other attributes.

## Tree structure terms

tree

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

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)

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)

## Geometry coding terms

position

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

## Attribute coding terms

low latency attributes

specific additional attributes coded altogether with geometry to support low latency decoding (9.2.5.4.3).

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* (3.1.18)

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* (3.1.19) value is predicted

# Abbreviated terms

## Acronyms

APS Attribute parameter set

ADU Attribute data unit

CBS Chunked bytestream

CPM Contextual probability model

DU Data unit

FBDU Frame boundary marker data unit

FSAP Frame-specific attribute properties

GDU Geometry data unit

GPS Geometry parameter set

L3C2 Low Latency, Low Complexity LiDAR coding

LiDAR Light Detection And Ranging

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

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:

b: sensing beam index

h: sensor head index

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

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 |
| Clip3( min, max, 𝑥 ) |  |
| Cos( 𝑥 ) | Trigonometric cosine function |
| DivExp2Floor( 𝑥, 𝑝 ) |  |
| DivExp2Fz( 𝑥, 𝑝 ) |  |
| DivExp2Up( 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 |
| IntLog2( 𝑥 ) | Floor( Log( 𝑥 ) ÷ Log( 2 ) ) |
| Log( 𝑥 ) | Natural logarithm of the argument 𝑥 |
| Min( 𝑥, 𝑦 ) |  |
| Max( 𝑥, 𝑦 ) |  |
| RoundFz( 𝑥 ) | ( 2 × 𝑥 + Sign( 𝑥 ) ) / 2 |
| RoundUp( 𝑥 ) | ( 2 × 𝑥 + 1 ) / 2 |
| Sign( 𝑥 ) |  |
| Sin( 𝑥 ) | Trigonometric sine function |

### IntCos and IntSin

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

parameter 𝜃 is an integer value representing a 24-bit fixed point which specify an angle measured in Radians;

result for IntCos shall be equal to the value of the expression iCos[𝜃] and result for IntSin shall be equal to the value of the expression iSin[𝜃].

The expression iCos is obtained by a π÷2 rotation of iSin.

iCos[theta] := iSin[26353589 - Abs(theta)]

The expression iSin is obtained by mapping 𝜃 to [0; π÷2) interval and linearly interpolating the function approximation tabulated on this interval and adjusting the sign.

iSin[theta] := theta >= 0 ? sinOnHalfPi : -sinOnHalfPi  
 where  
 nextSinFp := idx < 1608 ? SinFp[idx + 1] : (1 << 24)  
 sinOnHalfPi := SinFp[idx] + (lambda × (nextSinFp - SinFp[idx]) >> 14)  
 where  
 theta1 := Abs(theta)  
 theta2 := theta1 >= 52707179 ? theta1 – 52707179 : theta1  
 theta3 := theta2 >= 26353589 ? 52707179 – theta2 : theta2  
 idx := theta3 >> 14  
 lambda := theta3 – (idx << 14)

The expression SinFp[ 𝑥 ] specifies the 24-bit fixed-point approximation of the sine function tabulated for integers x representing 10-bit precision fixed-point angle expressed in Radian in the interval [0; pi÷2).

### 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 |

### 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.

This specification is intended for point clouds where at least geometry information is captured by LiDAR. As such L3C2 point clouds are ordered according to the LiDAR sensor characteristics (typically following the capture order) and the ordering is mostly kept unchanged in order to ensure low latency processing.

Typical attributes that may be present in L3C2 point clouds are attributes captured by a LiDAR sensor such as reflectance, intensity, return number, number of returns, timestamp, classification. Additional attributes may also be present when the LiDAR sensor is coupled with other types of sensors : this is for instance the case of RGB attributes that may be captured by a traditional camera associated to the LiDAR sensor.

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.

## 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

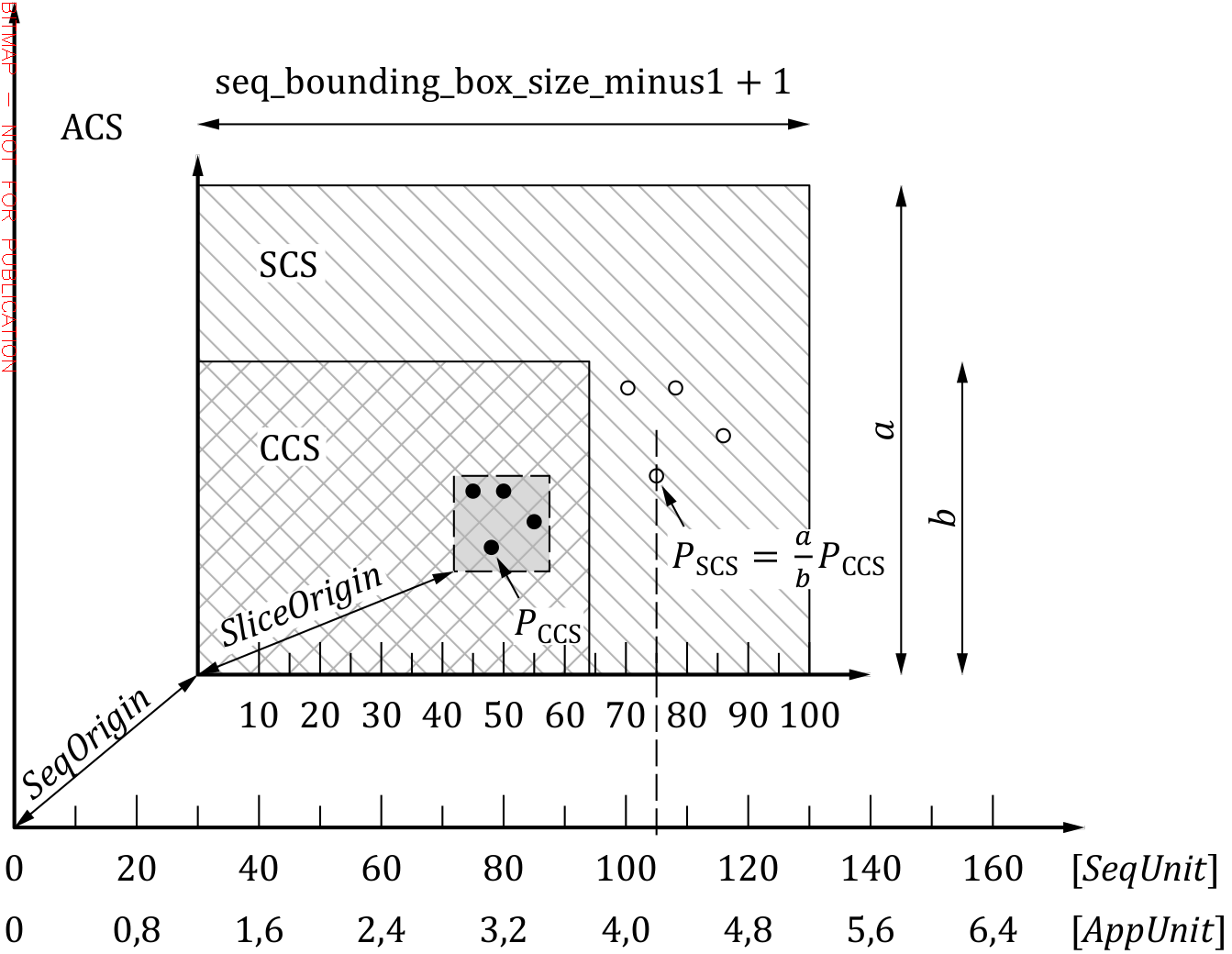
The sequence coordinate system is specified by the position of its origin in an externally defined application-specific coordinate system and by the length of its unit vectors.

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.



Key

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

Figure 1 — 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.

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

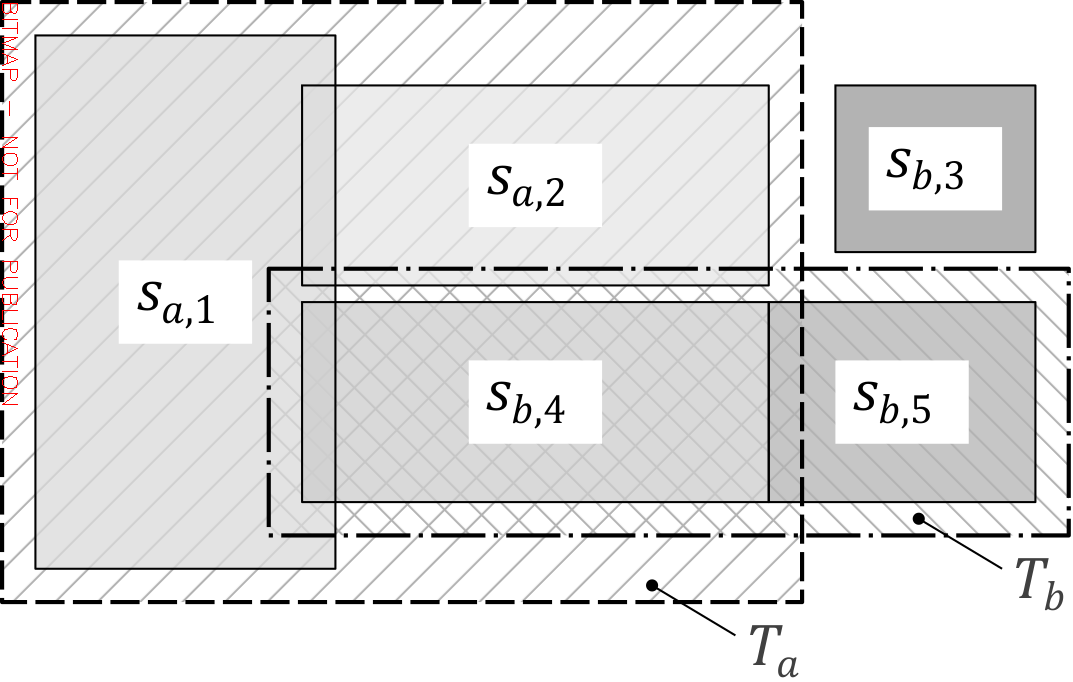
A group of slices can be identified by a common slice tag identifier (slice\_tag).

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.



Key

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

Figure 2 — Example arrangement of tiles and slices.

An example arrangement of tiles and slices within a coded point cloud frame is shown in Figure 2. 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 an L3C2 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 L3C2 bitstream generates a sequence of output point cloud frames.

### 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.

# 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 |
| main\_profile\_compliant | u(1) |
| reserved\_profile\_21bits | u(21) |
| 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\_sensing\_coverage\_enabled | u(1) |
| 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) |
| if( entropy\_continuation\_enabled ) |  |
| per\_beam\_coding\_state\_preserved\_enabled | u(1) |
| sps\_extension\_present | u(1) |
| if( sps\_extension\_present ) |  |
| 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) |
| geom\_sensing\_coverage\_enabled | u(1) |
| slice\_geom\_origin\_scale\_present | u(1) |
| if( ¬slice\_geom\_origin\_scale\_present ) |  |
| gps\_geom\_origin\_log2\_scale | ue(v) |
| geom\_dup\_points\_enabled | u(1) |
| 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) |
| } |  |
| num\_heads | ue(v) |
| for(h = 0; h < num\_heads; h++ ) { |  |
| num\_beams\_minus1[ h ] | ue(v) |
| beam\_steps\_per\_rotation\_minus1[ h ] | ue(v) |
| beam\_elevation\_init[ h ] | se(v) |
| beam\_voffset\_init[ h ] | se(v) |
| for( b = 1; b ≤ num\_beams\_minus1[ h ]; b++ ) { |  |
| beam\_elevation\_diff[ h ][ b ] | se(v) |
| beam\_voffset\_diff[ h ][ b ] | se(v) |
| } |  |
| } |  |
| geom\_use\_num\_azimuth\_steps\_per\_quant\_step | u(1) |
| geom\_use\_coarse\_neighbours | u(1) |
| geom\_use\_vertical\_prediction | u(1) |
| geom\_unordered\_points | u(1) |
| geom\_quantization\_step\_size | u(14) |
| geom\_quantization\_scaling\_log2 | u(4) |
| geom\_minimum\_azimuth\_quantization\_step\_size | u(25) |
| geom\_low\_latency\_attributes | u(1) |
| if( geom\_low\_latency\_attributes ) { |  |
| low\_latency\_attributes\_qp | u(8) |
| low\_latency\_attributes\_bit\_depth | u(5) |
| } |  |
| gps\_extension\_present | u(1) |
| if( gps\_extension\_present ) |  |
| 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 ) |  |
| 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( ) |  |
| sensed\_point\_sequences( ) |  |
| geometry\_data\_unit\_footer( ) |  |
| } |  |

#### Geometry data unit header syntax

|  |  |  |
| --- | --- | --- |
| geometry\_data\_unit\_header( ) { | Descriptor | Semantics |
| gdu\_geometry\_parameter\_set\_id | u(4) | 7.4.3.2 |
| gdu\_reserved\_zero\_3bits | 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( seq\_sensing\_coverage\_enabled && geom\_sensing\_coverage\_enabled ) { |  |  |
| sensing\_coverage\_present | u(1) | 7.4.3.2 |
| if( sensing\_coverage\_present ) { |  |  |
| sensing\_coverage\_may\_have\_missing\_points | u(1) | 7.4.3.2 |
| for(h = 0; h < num\_heads; h++ ) { |  |  |
| num\_sensing\_ranges\_minus1\_bits[  h  ] | ue(v) | 7.4.3.2 |
| num\_sensing\_ranges\_minus1[  h  ] | u(v) | 7.4.3.2 |
| sensing\_range0\_start\_bits[  h  ] | ue(v) | 7.4.3.2 |
| sensing\_range\_length\_minus1\_bits[  h  ] | ue(v) | 7.4.3.2 |
| if(num\_sensing\_ranges\_minus1[  h  ] ) |  |  |
| sensing\_range\_start\_offset\_minus1\_bits[  h  ] | ue(v) | 7.4.3.2 |
| sensing\_range0\_start[  h  ] | u(v) | 7.4.3.2 |
| sensing\_range0\_length\_minus1[  h  ] | u(v) | 7.4.3.2 |
| for(i = 0; i < num\_sensing\_ranges\_minus1[  h  ] ; i++ ) { |  |  |
| sensing\_range\_start\_offset\_minus1[  h  ][  i  ] | s(v) | 7.4.3.2 |
| sensing\_range\_length\_minus1[  h  ][  i  ] | 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( per\_sensor\_coding\_state\_preserved\_enabled ) |  |  |
| per\_sensor\_coding\_state\_preserved | u(1) | 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 |
| } |  |  |
| stream\_cnt\_minus1 | ue(v) | 7.4.3.2 |
| if( stream\_cnt\_minus1) { |  |  |
| stream\_len\_bits | u(6) | 7.4.3.2 |
| for(i = 0; i < stream\_cnt\_minus1; i++ ) |  |  |
| stream\_len[ i] | u(v) | 7.4.3.2 |
| } |  |  |
| byte\_alignment( ) |  |  |
| } |  |  |

#### Geometry data unit footer syntax

|  |  |  |
| --- | --- | --- |
| geometry\_data\_unit\_footer( ) { | Descriptor | Semantics |
| byte\_alignment( ) |  |  |
| slice\_num\_points\_minus1 | u(24) | 7.4.3.3 |
| } |  |  |

#### Sensed point sequences syntax

|  |  |  |
| --- | --- | --- |
| sensed\_point\_sequences( ) { | Descriptor | Semantics |
| PtnCnt = 0 |  |  |
| for( HeadIdx = 0; HeadIdx < num\_heads; HeadIdx++ ) { |  |  |
| num\_points[ HeadIdx ] | u(20) | 9.2.3.1 |
| for( PtnIdx = 0; PtnIdx < num\_points[ HeadIdx ]; PtnIdx++ ) { |  |  |
| sensed\_point(  ) |  |  |
| PtnCnt = PtnCnt + 1 |  |  |
| } |  |  |
| end\_of\_entropy\_stream | ae(v) | 9.2.3.1 |
| } |  |  |
| } |  |  |

#### Sensed point syntax

|  |  |  |
| --- | --- | --- |
| sensed\_point(  ) { | Descriptor | Semantics |
| if( PtnIdx == 0       &&  !(per\_sensor\_coding\_state\_preserved          && SensingHeadStartedCapturing[ HeadIdx ]) ) { |  |  |
| azimuth\_index\_start[ HeadIdx] | u(14) | 9.2.3.2 |
| elevation\_index\_start[ HeadIdx ] | u(8) | 9.2.3.2 |
| } else if( !NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx  − 1] ) |  |  |
| coarse\_position( ) |  |  |
| if(PtnIdx == 0        || !NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx  − 1] ) |  |  |
| num\_coarse\_position\_duplicates[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.2 |
| else if( geom\_dup\_point\_enabled ) |  |  |
| dup\_point[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.2 |
| if( !dup\_point ) |  |  |
| point\_position( ) |  |  |
| if( geom\_low\_latency\_attributes) |  |  |
| point\_low\_latency\_attributes( ) |  |  |
| } |  |  |

#### Coarse position syntax

|  |  |  |
| --- | --- | --- |
| coarse\_position(  ) { | Descriptor | Semantics |
| AzimuthIdx = SensingHeadLastAzimuthIdx[ HeadIdx ] |  |  |
| ElevationIdx = SensingHeadLastElevationIdx[ HeadIdx ] + 1 |  |  |
| if( geom\_unordered\_points ) |  |  |
| late\_point[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.3 |
| if( late\_point[ HeadIdx ][ PtnIdx ] ) |  |  |
| AzimuthIdx−−, ElevationIdx−− |  |  |
| next = 1 |  |  |
| while(next) { |  |  |
| elevation\_offset | ae(v) | 9.2.3.3 |
| ElevationIdx += elevation\_offset |  |  |
| if(ElevationIdx == num\_beams\_minus1[ HeadIdx ] + 1) { |  |  |
| azimuth\_offset | ae(v) | 9.2.3.3 |
| AzimuthIdx += AzimuthIdxOffset[AzimuthIdx][azimuth\_offset] |  |  |
| ElevationIdx = 0 |  |  |
| } |  |  |
| else |  |  |
| next = 0 |  |  |
| } |  |  |
| } |  |  |

#### Point position syntax

|  |  |  |
| --- | --- | --- |
| point\_position(  ) { | Descriptor | Semantics |
| pred\_idx[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| if( geom\_use\_vertical\_prediction      && pred\_idx[ HeadIdx ][ PtnIdx ] == 1     && VerticalPredEligible) |  |  |
| vertical\_radius\_pred[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| radius\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| azimuth\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| if( geom\_quantization\_step\_size == 64) { |  |  |
| x\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| y\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| } |  |  |
| z\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.4 |
| } |  |  |

#### Point low latency attributes syntax

|  |  |  |
| --- | --- | --- |
| point\_low\_latency\_attributes(  ) { | Descriptor | Semantics |
| attr\_pred[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.5 |
| if( attr\_pred[ HeadIdx ][ PtnIdx ]) |  |  |
| attr\_res[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.5 |
| else |  |  |
| attr\_val[ HeadIdx ][ PtnIdx ] | ae(v) | 9.2.3.5 |
| } |  |  |

### Attribute data unit

#### Attribute data unit syntax

|  |  |
| --- | --- |
| attribute\_data\_unit( ) { | Descriptor |
| attribute\_data\_unit\_header( ) |  |
| if( attr\_coding\_type ≠ 3 ) |  |
| 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\_reserved\_zero\_3bits | 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 ) |  |  |
| 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 |
| } |  |  |
| 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 |
| } |  |  |
| } |  |  |

### 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.

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

Reserved\_profile\_21bits shall be equal to 0 in bitstreams conforming to this version of this document. Other values for reserved\_profile\_21bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of reserved\_profile\_21bits.

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.

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.

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.

Seq\_sensing\_coverage\_enabled specifies whether (when 1) or not (when 0) the sensing coverage may be presented in a DU where that SPS is activated.

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.

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.

Per\_sensor\_coding\_state\_preserved\_enabled specifies whether (when 1) or not (when 0) the sensor coding state data may be preserved between DUs where that SPS is activated. Sensor coding state data are used to guide entropy parsing of a DU and updated during this process. When per\_sensor\_coding\_state\_preserved\_enabled is 1 the sensor coding state data preserved at the end of a previous DU may be used to initialize the sensor coding state data at the beginning of a DU. Otherwise, it is initialized with default values. When not present, per\_sensor\_coding\_state\_preserved\_enabled shall be inferred to be 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.

##### 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.

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 | L3C2 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.

##### L3C2 user defined attribute properties

L3C2 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.

Geom\_sensing\_coverage\_enabled specifies whether (when 1) or not (when 0) sensing coverage may be presented in a DU where that GPS is activated.

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\_points\_enabled specifies whether (when 1) or not (when 0) a duplicated points can be signalled in a GDU by a flag per coarse position duplicate.

1. geom\_dup\_point\_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 dup\_point syntax elements.

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

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 sensing beam in Figure 4.

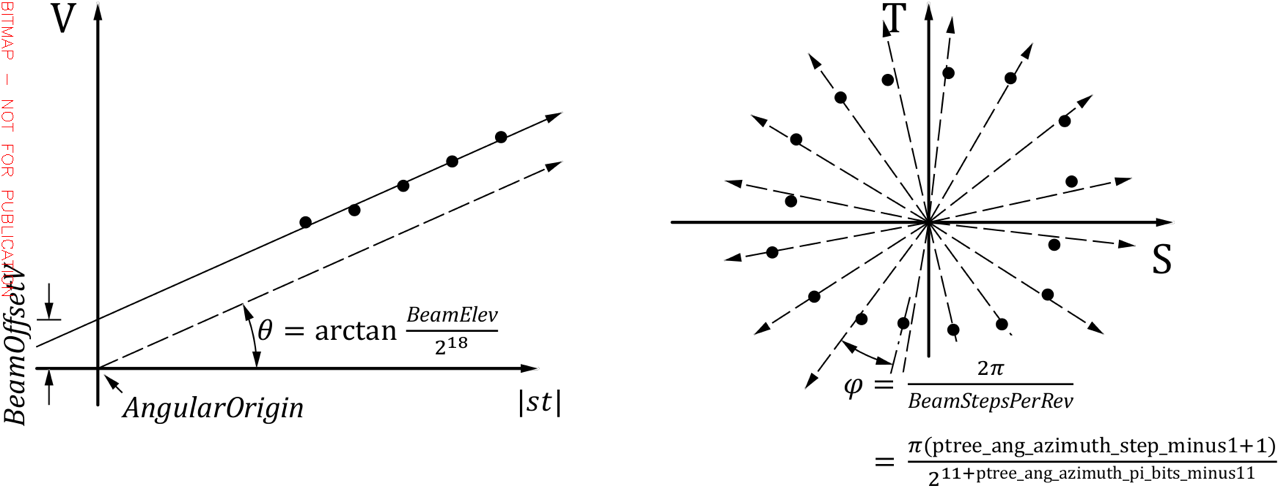
****

Figure 3 — 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\_heads specifies the number of sensing heads enumerated by the GPS.

Num\_beams\_minus1[ h ] plus 1 specifies the number of sensing beams enumerated by the GPS for the h-th sensing head.

The expression ElevationPackIndex[ h ][ b ]  provides the elevation pack index for sensing elevation angle with sensing beam index b and for sensing head with index h. The elevation pack index is an index for grouping successive elevation angle indexes together in at most 8 packs of sensing elevation angles per sensing head. The elevation pack index is derived as follows.

ElevationPackIndex[ h ][ b ] := (b \* 8) / (num\_beams\_minus1[ h ] + 1)

beam\_steps\_per\_rotation\_minus1[ h ] plus 1 specifies the number of steps made per revolution by the h-th sensing head.

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

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

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

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

geom\_use\_num\_azimuth\_steps\_per\_quant\_step specifies whether (when 1) or not (when 0) the coded geometry of a DU may rely on an estimated number of azimuthal angle steps covered by an azimuthal angle quantization step to determine contexts for entropy coding of a coarse order offset.

geom\_use\_coarse\_neighbours specifies whether (when 1) or not (when 0) the coded geometry of a DU may rely on coarse neighbours’ occupancy to determine contexts for entropy coding of a coarse order offset.

geom\_use\_vertical\_prediction specifies whether (when 1) or not (when 0) the coded geometry of a DU may rely on radial prediction made from point associated with different sensing beams.

geom\_unordered\_points specifies whether (when 1) or not (when 0) the coded geometry of a DU may contain negative coarse order offset between successive points.

geom\_quantization\_step\_size specifies the unitary quantization step size for unitary measure in the downscaled geometry space defined by geom\_quantization\_scaling\_log2. When equal to 64, the scaled geometry is lossless coded.

geom\_quantization\_scaling\_log2 specifies downscaling parameters of the geometry coordinates.

geom\_minimum\_azimuth\_quantization\_step\_size specifies minimum azimuthal angle quantization step size that shall be considered for azimuthal angle coding for any point of the point cloud (3.1). It is representing a fixed-point value with a number of fractional bits equal to 20.

geom\_low\_latency\_attributes specifies whether (when 1) or not (when 0) low latency attributes coding is used, and the coded attributes are attached to coded geometry in the same DU.

1. Only reflectance attribute may be coded using low latency attributes. If other attributes are present in the point cloud, they are coded in ADU.

When geom\_low\_latency\_attributes equals 1, no ADU shall be present for the reflectance attribute with index AttrIdx and no Attribute parameter set shall be provided for this AttrIdx.

low\_latency\_attributes\_qp specifies the quantization parameter for low latency coded attributes. When equal to 0, the low latency attributes are lossless coded.

**low\_latency\_attributes\_bit\_depth** specifies the bit depth of low latency coded attributes.

#### 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.

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 point’' 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.

##### 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. When lod\_scalability\_enabled is not present, it shall be inferred to be 0.

LoD scalability is not supported in the first version of this specification. It is a requirement of bitstream conformance that lod\_scalability\_enabled shall be 0.

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.

##### 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.

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

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.

sensing\_coverage\_present equal to 1 specifies that GDU header provides a sensing coverage for the GDU, and that GDU coding relies on that sensing coverage.

sensing\_coverage\_may\_have\_missing\_points equal to 1 specifies that some points may be missing in the GDU from within the sensing coverage provided for the GDU. And thus, not having a point for a given sensor direction does not always mean that there is no object/obstacle in that direction.

1. An application using data decoded from a GDU having a sensing\_coverage\_may\_have\_missing\_points equal to 1 must take special care with its interpretation of the absence of point for a given sensor direction. Thus it shall not assume that not having a point implies there is nothing in front of the sensor in that direction. If needed, the application may have to use robust methods to determine if there is truly nothing.

num\_sensing\_ranges\_minus1\_bits[  h  ] specifies the length in bits of num\_sensing\_ranges\_minus1[  h  ] syntax element.

sensing\_range0\_start\_bits[  h  ] specifies the length in bits of sensing\_range0\_start[  h  ] syntax element.

sensing\_range\_length\_minus1\_bits[  h  ] specifies the length in bits of sensing\_range0\_length\_minus1[  h  ] and sensing\_range\_length\_minus1[  h  ][  i  ] syntax elements.

sensing\_range\_start\_offset\_minus1\_bits[  h  ] specifies the length in bits of sensing\_range\_start\_offset\_minus1[  h  ][  i  ] syntax elements.

num\_sensing\_ranges\_minus1[  h  ], sensing\_range0\_start[  h  ], sensing\_range0\_length\_minus1[  h  ], sensing\_range\_start\_offset\_minus1[  h  ][  i  ] and sensing\_range\_length\_minus1[  h  ][  i  ] altogether specify the sensing coverage of the GDU for the sensing head with index h. The sensing coverage corresponds to the union of a number num\_sensing\_ranges\_minus1[  h  ] plus 1 of sensing ranges. Each sensing range correspond to a range of successive azimuthal angle indexes for which the sensing head was able to probe, sense the presence of points and provide the points which are encoded in the GDU.

The first and the last azimuthal angle indexes within a sensing range with index k are specified by the expressions SensingRangeFirst[  h  ][  k  ] and SensingRangeLast[  h  ][  k  ].

SensingRangeFirst[h][k] := k == 0  
 ? sensing\_range0\_start[h]  
 : SensingRangeLast[k–- 1] + sensing\_range\_start\_offset\_minus1[h][k–- 1] + 1

SensingRangeLast[h][k] := SensingRangeFirst[h][k] +  
 (k == 0  
 ? sensing\_range0\_length\_minus1[h]  
 : sensing\_range\_lenght\_minus1[h][k – 1])

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. 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.

per\_sensor\_coding\_state\_preserved equal to 1 specifies that the sensor coding state data preserved at the end of the previous GDU identified by prev\_slice\_id shall be used to initialize the sensor coding state data at the start of the GDU.

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]] << slice\_geom\_origin\_log2\_scale

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]

stream\_cnt\_minus1 plus 1 specifies the maximum number of entropy streams used to code the geometry information and possible low latency attributes for all sensing heads. When stream\_cnt\_minus1 is greater than zero, each of the stream\_cnt\_minus1 shall be conveyed in a separate entropy stream; the parsing state shall be memorized at the end of an entropy stream and restored according to 11.7 for each sensing head index, independently.

It is a requirement of the bitstream that stream\_cnt\_minus1 + 1 shall be equal to (1 + geom\_low\_latency\_attributes) × num\_heads.

stream\_len\_bits specifies the number of bits used to represent each of the stream\_len[ i] syntax elements.

stream\_len[ i] specifies the length in bytes of the  i-th entropy stream.

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

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.

### 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

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 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 attribute decoding 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.

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]

# 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.

## Sensed point sequence

### General

Subclause 9.2 specifies the reconstruction of point positions from parsed sensed point sequence.

The slice geometry can be represented by a sequence of coarse positions. Every coarse position specifies a single position projected in 2D plane in () domain for one or more points. Traversal of coarse positions is according to lexicographic order.

### State variables and expressions

Sensed point sequence decoding is specified in terms of the following state variables and expressions:

The variable HeadIdx an iterative index on the sensing heads.

The variable PtnIdx an iterative index on the sensed points decoded for as specific sensing head.

The arrays SensingAzimuthIdx[HeadIdx][PtnIdx] and SensingElevationIdx[HeadIdx][PtnIdx] are two arrays of variables representing the coarse sensing coordinates of the coarse position of a sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. These coordinates represent the projection of the point in the discrete 2D space of coarse sensing directions, specified by a sensing azimuthal angle index (SensingAzimuthIdx[HeadIdx][PtnIdx]) and a sensing elevation angle index (SensingElevationIdx[HeadIdx][PtnIdx]), where the sensing azimuthal angle index (SensingAzimuthIdx[HeadIdx][PtnIdx]) is obtained by the quantization of azimuthal angle of a sensed point based on the the azimuthal angle step size ptree\_ang\_azimuth\_step\_minus1

and where the sensing elevation angle index (SensingElevationIdx[HeadIdx][PtnIdx]) is the laser index of laser beam pointed to a sensed point.

The expression CoarsePosition[ HeadIdx ][ PtnIdx ]  is an alias to the pair of sensing coordinates SensingAzimuthIdx[HeadIdx][PtnIdx] and SensingElevationIdx[HeadIdx][PtnIdx]:

CoarsePosition[HeadIdx][PtnIdx] :=  
 (SensingAzimuthIndex[HeadIdx][PtnIdx], SensingElevationIndex[HeadIdx][PtnIdx])

The expression SensingOrderIdx[ HeadIdx ][ PtnIdx ]  specifies the sensing order index as the index of CoarsePosition[ HeadIdx ][ PtnIdx ]  in lexicographic ordered coordinates of every position in discrete 2D space of coarse sensing directions:

SensingOrderIdx[HeadIdx][PtnIdx] :=  
 SensingAzimuthIdx[HeadIdx][PtnIdx] \* (num\_beams\_minus1[ HeadIdx ] + 1)  
 + SensingElevationIdx[HeadIdx][PtnIdx]

The expression SensedPtnAng[ HeadIdx ][ PtnIdx ][k]  specifies the k-th reconstructed RPI coordinate of a sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. k=0 is for radius, k=1 is for azimuth angle (Phi) and k=2 is for elevation index. It is specified in sub-clause 9.2.5.

The expression SensedPtnCoord[ HeadIdx ][ PtnIdx ][k]  specifies the k-th reconstructed XYZ coordinate of a sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. k=0 is for X coordinate, k=1 is for Y coordinate and k=2 is for Z coordinate. It is specified in sub-clause 9.2.5.

The expression AbsoluteSensingElevationIdx[HeadIdx][ElvIdx] specifies an absolute index for the sensing elevation angle with index ElvIdx for the sensing head with index HeadIdx. It is used for traditional (excluding low-latency) attributes decoding.

absoluteElvIdx = 0  
for (h = 0; h < num\_heads; h++)  
 for (elvIdx = 0; elvIdx ≤ num\_beams\_minus1[h]; elvIdx++)  
 AbsoluteSensingElevationIdx[h][elvIdx] := absoluteElvIdx++

### Syntax element semantics

#### Sensed point sequence

num\_points[ HeadIdx] specifies the number of sensed points coded in the entropy stream for the sensor head with index HeadIdx.

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. When geom\_low\_latency\_attributes is equal to 1, end\_of\_entropy\_stream specifies the end for both geometry entropy stream and low latency attributes entropy streams. Otherwise, it only specifies the end of geometry entropy stream.

#### Sensed point

Sensed point decoding is specified in terms of the following expression:

The expression ElvPackIdx is the elevation pack index of the sensed point:

ElvPackIdx :=  
 ElevationPackIndex[ HeadIdx ][SensingElevationIdx[HeadIdx][PtnIdx]]

azimuth\_index\_start[ HeadIdx ] and elevation\_index\_start[ HeadIdx ] specify the coarse position of the first sensed point coded for the sensing head with index HeadIdx, unless the sensor coding state is dependent on previous GDU(s) (i.e. the per\_sensor\_coding\_state\_preserved  
is equal to  1) and a preceding point was available from that previous GDU(s) (i.e. the SensingHeadStartedCapturing[ HeadIdx ] is equal to 1).

When azimuth\_index\_start[ HeadIdx ] and elevation\_index\_start[ HeadIdx ] are present in the GDU,

* SensingAzimuthIdx[HeadIdx][PtnIdx] and SensingElevationIdx[HeadIdx][PtnIdx] shall be set respectively to azimuth\_index\_start[ HeadIdx ] and elevation\_index\_start[ HeadIdx ], and then
* sensor coding state data shall be initialized using the subclause for initialization with decoded coarse position of first sensed point (11.6.3).

num\_coarse\_position\_duplicates[ HeadIdx ][ PtnIdx ] specifies a number of duplicates of the coarse position which corresponds to a number of successive sensed points sharing the same coarse position as the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx.

The arrays SensingAzimuthIdx and SensingElevationIdx shall be updated accordingly to the number of duplicates of the coarse position:

for (k = 1; k ≤ num\_coarse\_position\_duplicates[ HeadIdx ][ PtnIdx ]; ++k) {  
 SensingAzimuthIdx[HeadIdx][PtnIdx + k] = SensingAzimuthIdx[HeadIdx][PtnIdx]  
 SensingElevationIdx[HeadIdx][PtnIdx + k] = SensingElevationIdx[HeadIdx][PtnIdx]  
}

dup\_point[ HeadIdx ][ PtnIdx ] specifies whether (when 1) or not (when 0) the sensed point with index PtnIdx  has a duplicated point position of the sensed point with in–ex PtnIdx − 1, and if so, the corresponding reconstructed point has same coordinates. When dup\_point is not present, it shall be inferred to be 0.

#### Coarse position

Coarse position decoding consists in decoding coarse order difference information between two successive points sorted by the lexicographic order, as is shown in Figure 3 where ∆o is the coarse order difference*.* The coarse order difference ∆o between two successive points can be divided into two components azimuth\_offset and elevation\_offset, respectively, which are described below.

Coarse position decoding is specified in terms of the following expression:

The expression ElvPackIdxLast, the elevation pack index for elevation index of the last decoded point:

ElvPackIdxLast :=  
 ElevationPackIndex[ HeadIdx ][SensingHeadLastElevationIdx[ HeadIdx ]]

At the beginning of the decoding of a coarse\_position() syntax element, the variable ElevationOffsetCtxSel is initialized to 0.

late\_point[ HeadIdx ][ PtnIdx ]  specifies whether (when 1) or not (when 0) the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx, has a sensing order index SensingOrderIdx[ HeadIdx ][ PtnIdx ]  that is lower than sensing order index of previously decoded point SensingOrderIdx[ HeadIdx–][ PtnIdx − 1 ] . If so, the point with the sensed point is considered as a late point. Whether the sensed point is a late point or not affects the decoding process to obtain SensingOrderIdx[ HeadIdx ][ PtnIdx ] . When late\_point is not present, it shall be inferred to be 0.

When late\_point is present and after it is decoded, BufferLate[ HeadIdx ][ ElvPackIdxLast ] is updated:

BufferLate[ HeadIdx ][ElvPackIdxLast] =  
 ((BufferLate[ HeadIdx ][ElvPackIdxLast] & 3) << 1) | late\_point[HeadIdx][PtnIdx]

azimuth\_offset and elevation\_offset specify offsets to be applied respectively on AzimuthIdx and ElevationIdx while decoding the coarse position specified by the expression CoarsePosition[ HeadIdx ][ PtnIdx ]  for the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. The azimuth\_offset is expressed as a number of coarse azimuth positions belonging to the sensing coverage. As is described above, the two offsets (azimuth\_offset and elevation\_offset) represents coarse order difference between a point with CoarsePosition[ HeadIdx ][ PtnIdx ]  and its previously coded point in lexicographic ordered coordinates*.* The coarse order difference between a **late\_point**[ *HeadIdx* ][ *PtnIdx* ]  and its successively previously decoded point is a negative value.

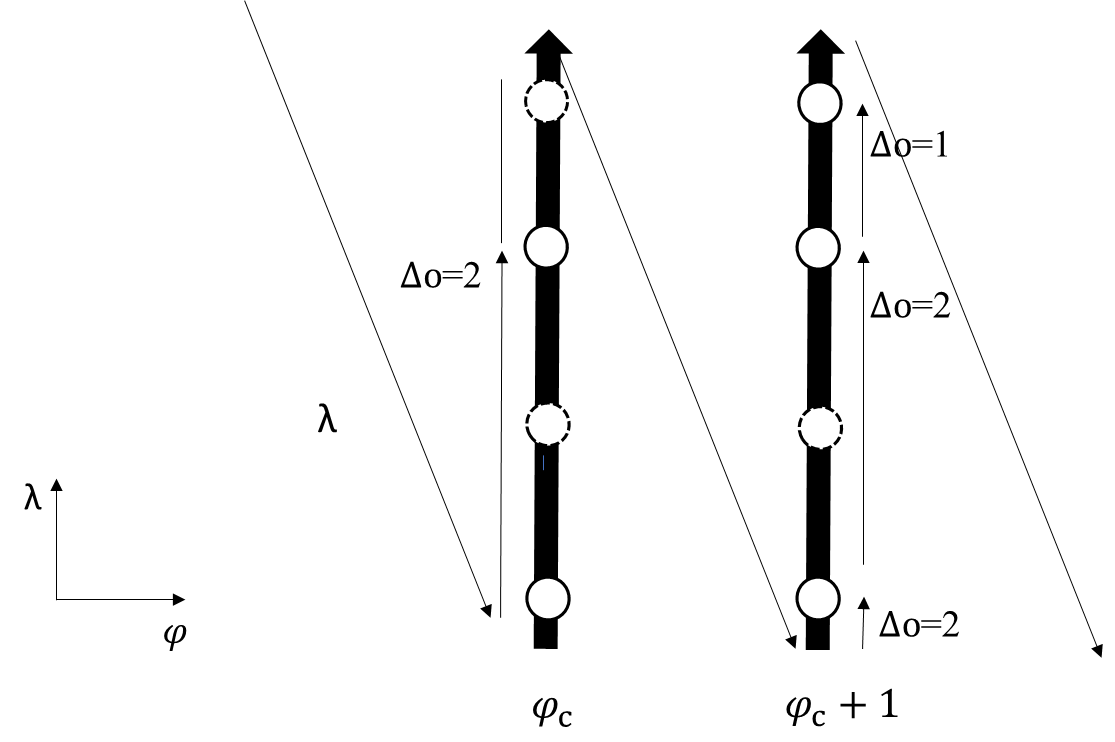


Figure 4 example of coarse position of point cloud represented by successive order difference ∆o

After any azimuth\_offset syntax element, the sensor coding state data shall be updated according to subclause 11.6.4, and the AzimuthIdx variable is updated according to the variable AzimuthIdxOffset[AzimuthIdx][azimuth\_offset].

At the end of a coarse\_position(  ) syntax element, the state variables SensingAzimuthIdx[HeadIdx][PtnIdx] and SensingElevationIdx[HeadIdx][PtnIdx] shall be set to the decoded coarse positions:

SensingAzimuthIdx[HeadIdx][PtnIdx] = AzimuthIdx

SensingElevationIdx[HeadIdx][PtnIdx] = ElevationIdx

And then, the sensor coding state data shall be updated using the subclause for updating with decoded coarse position (11.6.4).

#### Point position

pred\_idx[ HeadIdx ][ PtnIdx ]  specifies the predictor index of the predictor to be used for predicting the coordinates of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx.

vertical\_radius\_pred[ HeadIdx ][ PtnIdx ]  specifies whether (when 1) or not (when 0) the radius of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx, is being predicted from a previous sensed point with a different sensing elevation index. When vertical\_radius\_pred is not present, it shall be inferred to be 0.

radius\_res[ HeadIdx ][ PtnIdx ]  specifies the prediction residual for the radius of the RPI position of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx.

azimuth\_res[ HeadIdx ][ PtnIdx ]  specifies the prediction residual for the azimuthal angle of the RPI position of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx.

x\_res[ HeadIdx ][ PtnIdx ]  specifies the residual for the X axis of the XYZ position after RPI to XYZ position conversion of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. When x\_res is not present, it shall be inferred to be 0.

y\_res[ HeadIdx ][ PtnIdx ]  specifies the residual for the Y axis of the XYZ position after RPI to XYZ position conversion of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. When y\_res is not present, it shall be inferred to be 0.

z\_res[ HeadIdx ][ PtnIdx ]  specifies the residual for the Z axis of the XYZ position after RPI to XYZ position conversion of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx.

#### Point low latency attribute

attr\_pred[ HeadIdx ][ PtnIdx ]  specifies whether (when 1) or not (when 0) the low latency attribute value of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx, is being predicted from a previous low latency attribute value.

attr\_res[ HeadIdx ][ PtnIdx ]  specifies the prediction residual for the low latency attribute value of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. When attr\_res is not present, it shall be inferred to be 0.

attr\_val[ HeadIdx ][ PtnIdx ]  specifies the attribute value of the sensed point with index PtnIdx  in the sensed point sequence for the sensing head with index HeadIdx. When attr\_res is not present, it shall be inferred to be 0.

### Contextualization of elevation\_offset

The context selection for decoding elevation\_offset is specified in terms of the following state variables and expressions:

The expression BinIdxTu used in contextualization of elevation\_offset specifies the occupancy information of an intermediate point, which is the i-th point after the last occupied point in the order of the coarse representation, as shown in Figure 5, and the context information used to code BinIdxTu is determined by neighbours of the i-th intermediate point.

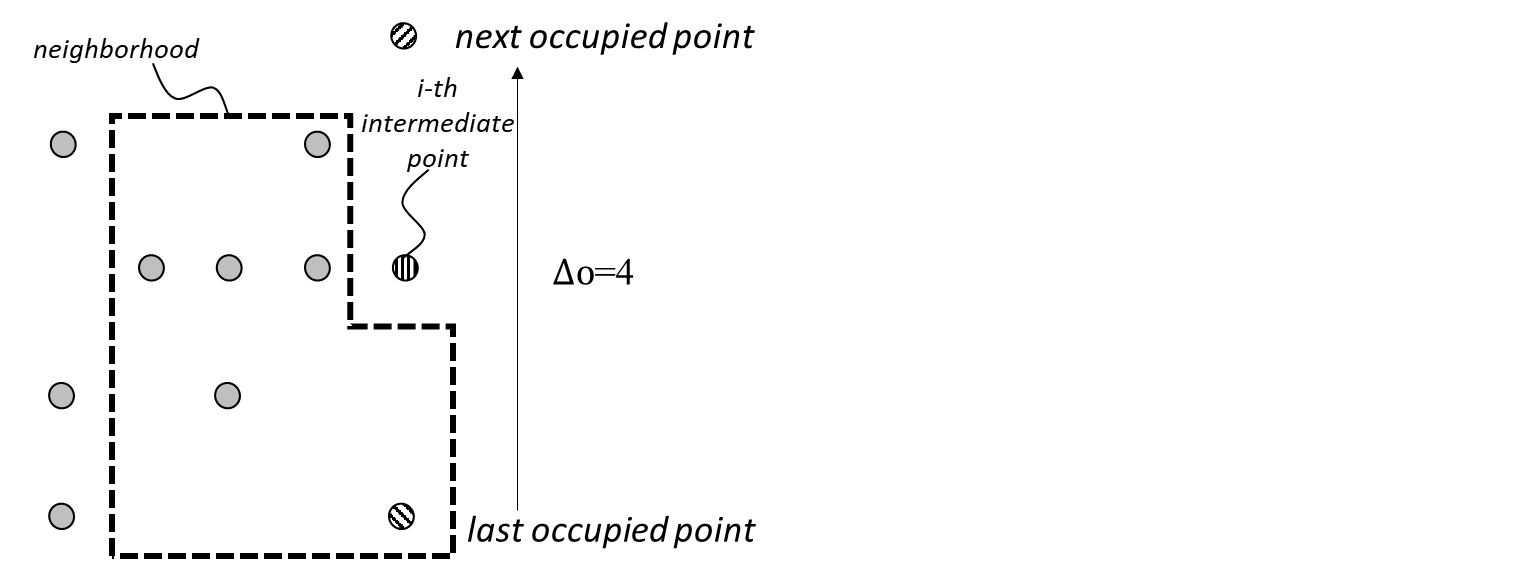


Figure 5 intermediate point and its neighbourhood

The expression OffsetElv is the elevation index ElevationIdx being offset by the decoded bin index BinIdxTu, during the decoding of an elevation\_offset syntax element:

OffsetElv :=  
 ElevationIdx + BinIdxTu

The expression OffsetElvPackIdx is the elevation pack index for the elevation index ElevationIdx being offset by the decoded bin index BinIdxTu, during the decoding of an elevation\_offset syntax element:

OffsetElvPackIdx :=  
 ElevationPackIndex[ HeadIdx ][OffsetElv]

The variable ElvOffsetCtxSel, is used for context selection during the decoding of elevation\_offset syntax elements.

The variable idxOffsetPrev is horizonal information to anticipate the value of the the decoded bin index BinIdxTu*,* and the idxOffsetPrev is determined as coarse angle difference between coarse angle AzimuthIdxandcoarse angleSensingBeamPrevLastAzimuthIdx[HeadIdx][OffsetElv]of the penultimate already coded point with same sensor indexOffsetElv.The value of idxOffsetPrev   
corresponds to the length for the horizontal arrow in Figure 6.

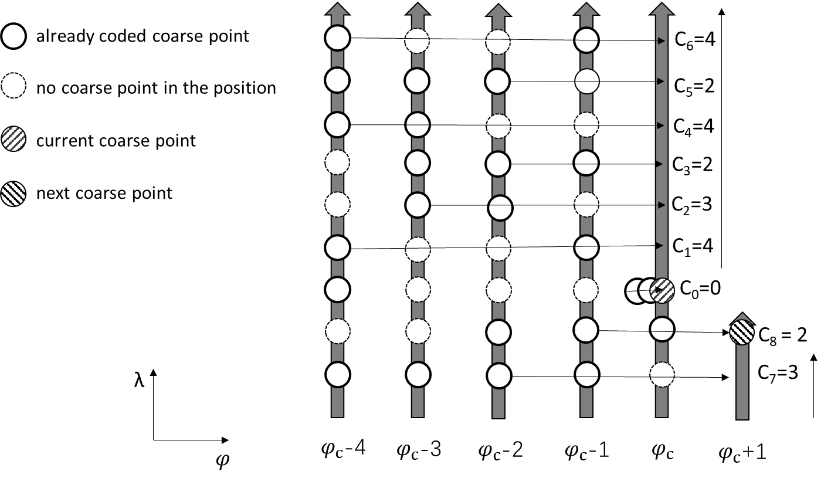


Figure 6 determination of the horizontal information idxOffsetPrev

At the beginning of a coarse\_position() syntax element, ElvOffsetCtxSel is set to 0.

#### Determination of *CtxIdxElvOffset*

The context selection for the coded bin of elevation\_offset syntax element is specified by the expression CtxIdxElvOffset.

CtxIdxElvOffset :=  
 late\_point[ HeadIdx ][ PtnIdx ] == 1  
 ? ctxIdxLate  
 : geom\_use\_coarse\_neighbours && idxOffset > 3  
 ? 48 + ctxIdxIsolated  
 : geom\_use\_num\_azimuth\_steps\_per\_quant\_step  
 && lastStepsFromRadius > 1  
 ? 52 + ctxIdxHoles  
 : geom\_use\_coarse\_neighbours  
 ? 80 + ctxDynOBUF  
 : 96 + ctxOther  
 where  
 idxOffset := AzimuthIdx – SensingBeamLastAzimuthIdx[HeadIdx][OffsetElv]  
 idxOffsetPrev := AzimuthIdx – SensingBeamPrevLastAzimuthIdx[HeadIdx][OffsetElv]  
 lastStepsFromRadius := SensingBeamLastAzimuthStepsFromRadius[HeadIdx][OffsetElv]  
 ctxIdxLate :=  
 12 × (BufferLate[ HeadIdx ][ElvPackIdxLast] & 2)  
 + 3 × OffsetElvPackIdx  
 + ElvOffsetCtxSel  
 ctxIdxIsolated :=  
 (idxOffset > 6 ? 1 : 0)  
 + ((OffsetElv == 0 && ElvOffsetCtxSel == 0)  
 || (OffsetElv + 1 ≤ num\_beams\_minus1[ HeadIdx ]  
 && (SensingHeadNeighOccupancyBits[HeadIdx][OffsetElv + 1]))  
 ) ? 2 : 0)  
 ctxIdxHoles :=  
 4 × (max(-5, min(1, idxOffsetPrev – lastStepsFromRadius)) + 5)  
 + max(0, min(3, idxOffsetPrev - 2))  
 ctxDynOBUF :=  
 CtxIdxDynOBUF(OffsetElv, a, ElvOffsetCtxSel)  
 where  
 a :=  
 15 × max(0, min(3, idxOffsetPrev - 2))  
 + (b < 15 ? b : b – 15)  
 where  
 b :=  
 2 × SensingHeadNeighOccupancyBits[HeadIdx][elv2]  
 + (SensingHeadNeighOccupancyBits[HeadIdx][elv1] & 1)  
 + (SensingHeadNeighOccupancyBits[HeadIdx][OffsetElv] & 3)  
 where  
 elv1 := OffsetElv ≥ 1 ? OffsetElv – 1 : OffsetElv + 2  
 elv2 := OffsetElv ≥ 2 ? OffsetElv – 2 : OffsetElv + 1  
 ctxOther :=  
 21 × OffsetElvPackIdx  
 + 3 × max(0, min(6, idxOffsetPrev – 1)  
 + ElvOffsetCtxSel

The Dynamic OBUF selection function CtxIdxDynOBUF(OffsetElv, a, ElvOffsetCtxSel) is used to determine context CtxIdxElvOffset by calling OBUF instances according to 12.4. The first contextual information *info1* used as input when calling the OBUF instance in 12.4 is set by using information a and ElvOffsetCtxSel*,* and the second contextual information *info2* used as input when calling the OBUF instance in 12.4 is set to be OffsetElv*.*

info1= a<<1 + ElvOffsetCtxSel

info2= OffsetElv

#### Update of variables for *CtxIdxElvOffset*

After any coded bin of the truncated unary code for elevation\_offset syntax element has been decoded, the variable ElvOffsetCtxSel shall be updated as follows.

ElvOffsetCtxSel +=  
 (ElvOffsetCtxSel < 2 - geom\_use\_coarse\_neighbours) ? 1 : 0

#### Update of variables for dynamic OBUF tables

After any coded bin of the truncated unary code for elevation\_offset syntax element has been decoded according to 12.4.1, then the context index updates and OBUF instance updates according to 12.4.1.

### Reconstruction of sensed point positions

#### State variables

The reconstruction of sensed point positions from sensed point sequence is specified in terms of the following state variables:

The variable PtnIdx, the sensed point index of the currently decoded point in decoding order in the sequence of sensed points.

The variable PtnCnt, a count of sensed points parsed from the bitstream.

#### Functions

The reconstruction of sensed point positions is specified in terms of the following functions.

##### RadiusScaleFromAzimuth

The function q = RadiusScaleFromAzimuth( 𝜃) is a 10-bit precision fixed-point approximation of the inverse of 1 + 0.85(|cos(𝜃)| + |sin(𝜃)| - 1) used as scaling factor for radius quantization step size providing a close to uniform quantization in cartesian space. Its:

parameter 𝜃 is an integer value representing a 20-bit precision fixed-point angle in Radian which shall belong to (−2π; 2π) opened interval;

result shall be equal to the value of the expression radiusScaleFromAzimuth[ 𝜃 ].

The expression iRadiusScale [ x ] specifies the 10-bit precision fixed-point approximation of the function tabulated for integers x representing 7-bit precision fixed-point angle expressed in Radian in the interval [0; π÷4].

The result is obtained by mapping 𝜃 to [0; π÷4) interval and linearly interpolating the function approximation tabulated on this interval.

radiusScaleFromAzimuth[theta] :=  
 iRadiusScale[idx] + (iFrac × (iRadiusScale[idx + 1] - iRadiusScale[idx]) >> 13)  
 where  
 theta1 := Abs(theta)  
 theta2 := theta1 ≥ 3294199 ? theta1 – 3294199 : theta1  
 theta3 := theta2 ≥ 1647099 ? theta2 – 1647099 : theta2  
 theta4 := theta3 ≥ 823550 ? 1647099 – theta3 : theta3  
 idx := theta4 >> 13  
 iFrac := theta4 – (idx << 13)

##### AzimuthStepScaleFromAzimuth

The function q = AzimuthStepScaleFromAzimuth( 𝜃) is a 10-bit precision fixed-point approximation of 1.1(|cos(𝜃)| + |sin(𝜃)|) used as scaling factor for azimuth step quantization step size providing a close to uniform quantization in cartesian space. Its:

parameter 𝜃 is an integer value representing a 20-bit precision fixed-point angle in Radian which shall belong to (−2π; 2π) opened interval;

result shall be equal to the value of the expression azimuthStepScaleFromAzimuth[ 𝜃 ].

The expression iAzimuthScale[ x ] specifies the 10-bit precision fixed-point approximation of the function tabulated for integers x representing 7-bit precision fixed-point angle x expressed in Radian in the interval [0; π÷4].

The result is obtained by mapping 𝜃 to [0; π÷4) interval and linearly interpolating the function approximation tabulated on this interval.

azimuthStepScaleFromAzimuth[theta] :=  
 iAzimuthScale[idx] + (iFrac × (iAzimuthScale[idx + 1] - iAzimuthScale[idx]) >> 13)  
 where  
 theta1 := Abs(theta)  
 theta2 := theta1 ≥ 3294199 ? theta1 – 3294199 : theta1  
 theta3 := theta2 ≥ 1647099 ? theta2 – 1647099 : theta2  
 theta4 := theta3 ≥ 823550 ? 1647099 – theta3 : theta3  
 idx := theta4 >> 13  
 iFrac := theta4 – (idx << 13)Decoding a sequence of sensed points

For each sensing head a sequence of sensed points is decoded according to 9.2.5.3

At the start of each sequence of sensed points, the sensor coding state is initialized according to 11.6.2.

For each sensed point of the sequence, the sensed point with index PtnIdx is decoded according to subclause 9.2.4.3.

#### Decoding of a sensed point

When decoding a sensed point the coarse position of the sensed point shall be first decoded according to subclause 9.2.4.3.1, then the reconstructed coordinates of the sensed point shall be decoded and reconstructed according to subclause 9.2.4.3.2.

##### Decoding of the coarse position of a sensed point

When decoding a sensed point with index PtnIdx in a sequence of sensed points for a sensing head with index HeadIdx, the coarse position of the sensed point is specified by CoarsePosition[ HeadIdx ][ PtnIdx ] .

When PtnIdx is equal to 0 and either the sensor coding state data may not be preserved between DUs or no sensed point has already been decoded for the sensing head with index HeadIdx, the coarse position of the sensed point is specified by azimuth\_index\_start[ HeadIdx] and elevation\_index\_start[ HeadIdx ].

if (PtnIdx == 0 &&  !(per\_sensor\_coding\_state\_preserved  
 && SensingHeadStartedCapturing[ HeadIdx ]) {  
 SensingAzimuthIndex[HeadIdx][PtnIdx] = azimuth\_index\_start[ HeadIdx]  
 SensingElevationIndex[HeadIdx][PtnIdx] = elevation\_index\_start[ HeadIdx]  
}

The number of coarse position duplicates next to the sensed point with index PtnIdx is specified by the expression NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx ] . It corresponds to a number of sensed points following the sensed point with index PtnIdx which share the same coarse position and is specified as follows:

NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx] :=   
 PtnIdx > 0 && NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx – 1] > 0  
 ? NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx – 1 ] – 1  
 : num\_coarse\_position\_duplicates[ HeadIdx ][ PtnIdx ]

When PtnIdx > 0, if the number of coarse position duplicates next to the previous sensed point of the sequence NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx  − 1]  is greater than zero, the coarse position of the sensed point is specified as being a coarse position duplicate of the previous sensed point and it is the same coarse position as the sensed point with index PtnIdx  − 1.

if (PtnIdx > 0 && NumCoarsePositionDuplicatesNext[ HeadIdx ][ PtnIdx  − 1] > 0) {  
 SensingAzimuthIndex[HeadIdx][PtnIdx] = SensingAzimuthIndex[HeadIdx][PtnIdx - 1]  
 SensingElevationIndex[HeadIdx][PtnIdx] = SensingElevationIndex[HeadIdx][PtnIdx - 1]  
}

When PtnIdx > 0, and the coarse position of the sensed point is not a coarse position duplicate, it shall be decoded according to subclauses 7.3.3.6 and 9.2.3.3.

When PtnIdx is equal to 0 or when the coarse position of the sensed point is not a coarse position duplicate, the number of coarse position duplicates of the coarse position of the sensed point shall be decoded by parsing the syntax element num\_coarse\_position\_duplicates[ HeadIdx ][ PtnIdx ] .

Otherwise, when duplicated points can be signaled in a GDU by a flag per coarse position duplicate (when geom\_dup\_points\_enabled is equal to 1), the value of the flag dup\_point[ HeadIdx ][ PtnIdx ] shall be decoded.

At the end of the decoding of the coarse position of a sensed point, sensor coding state variables shall be updated as specified in subclause 11.6.6.

##### Decoding of the reconstructed coordinates of a sensed point

When dup\_point[ HeadIdx ][ PtnIdx ] is equal to 1, the reconstructed RPI coordinates and the reconstructed XYZ coordinates of the sensed point shall be set equal to the reconstructed RPI coordinates and the reconstructed XYZ coordinates of the preceding sensed point.

if (dup\_point[HeadIdx][PtnIdx]) {  
 for (k = 0; k < 3; ++k) {  
 SensedPtnAng[HeadIdx][PtnIdx][k] = SensedPtnAng[HeadIdx][PtnIdx - 1][k]  
 SensedPtnCoord[HeadIdx][PtnIdx][k] = SensedPtnCoord[HeadIdx][PtnIdx - 1][k]  
 }  
}

When dup\_point[ HeadIdx ][ PtnIdx ] is not present or is equal to 0, the reconstructed RPI coordinates and the reconstructed XYZ coordinates of the sensed point shall be decoded as follows.

First, the predictor index pred\_idx[ HeadIdx ][ PtnIdx ]  shall be decoded.

The expression ElevationIdx is an integer value representing the elevation index of the sensed point. It is specified as:

ElevationIdx := SensingElevationIndex[HeadIdx][PtnIdx]

The expression CtxPredIdxSelector is used for contexts selection when decoding the predictor index. It is specified as follows:

CtxPredIdxSelector :=  
 (SensingPointAzimuthIncFromLast[HeadIdx][PtnIdx] > 0 ? 1 : 0)  
 + (SensingBeamLastPredIdx[HeadIdx][ElevationIdx] > 0 ? 2 : 0)

The expression VerticalPredEligible is a Boolean value representing the eligibility of the cross-sensing elevation prediction; and the expression RadiusVerticalPredicor is an 8-bit decimal precision fixed-point value representing the cross-sensing elevation prediction of the radius. When VerticalPredEligible is equal to 1 and the sensed point predictor index is equal to 1, the vertical\_radius\_pred[ HeadIdx ][ PtnIdx ] syntax element shall be decoded. The two expressions are specified as bellow. The eligible radius vertical predictor is the radius of the last sensed point having closest sensor elevation to the decoded sensed point for the closest last sensed point azimuth index, if its azimuth index is lower than the azimuth index minus 2 of the previously last sensed point with the elevation of the decoded sensed point.

VerticalPredEligible := vPredEligible  
RadiusVerticalPredictor := vPred  
 where  
 low := Max(0, ElevationIdx – 5)  
 high := Min(num\_beams\_minus1[HeadIdx], ElevationIdx + 3)  
 vPredEligible = 0  
 vPred = 0  
 if (geom\_use\_vertical\_prediction && pred\_idx[HeadIdx][PtnIdx] == 1) {  
 minEligibCrit = -(SensingBeamPrevLastAzimuthIdx[HeadIdx][ElevationIdx] – 2 << 8)  
 for (elv = low; elv ≤ high; ++elv) {  
 if (elv ≠ ElevationIdx) {  
 eligibCrit := Abs(elv – ElevationIdx)  
 – (SensingBeamLastAzimuthIdx[HeadIdx][elv] << 8)  
 if (eligibCrit <= minEligibCrit) {  
 minEligibCrit = eligibCrit  
 vPredEligible = 1  
 vPrev = SensingBeamRadiusPred[HeadIdx][elv][0]  
 }  
 }  
 }  
 }

When the cross-sensing elevation prediction of the radius is eligible, the syntax element vertical\_radius\_pred[HeadIdx][PtnIdx] is decoded. If it is equal to 1 the selected predictor for radius prediction is the cross-sensing elevation prediction and it is equal to RadiusVerticalPredictor. Otherwise, or if it is not eligible, the selected predictor for radius prediction is equal to SensingBeamRadiusPred[HeadIdx][ElevationIdx][pred\_idx[HeadIdx][PtnIdx]].

Then the prediction residual for the radius of the RPI position of the sensed point radius\_res[HeadIdx][PtnIdx] is decoded using following expressions.

The expression CtxRadiusResSelector , is used for contexts selection when decoding the radius residual of the sensed point. It is specified as follows:

CtxRadiusResSelector :=  
 pred\_idx[ HeadIdx ][ PtnIdx ] > 0  
 ? ElvPackIdx  
 : 8 + vertical\_radius\_pred[ HeadIdx ][ PtnIdx ]

The expression CtxRadiusResSignSelector, is used for contexts selection when decoding the sign of the radius residual of the sensed point. It is specified as follows:

CtxRadiusResSignSelector :=  
 SensingBeamLastRadiusResSign[HeadIdx][ElevationIdx]  
 + (SensingPointAzimuthIncFromLast[HeadIdx][PtnIdx] > 0 ? 1 : 0)  
 + (SensingPointPrevAzimuthIncFromLast[ HeadIdx ][ PtnIdx ] > 0 ? 4 : 0)

The expression DeltaRadius is an integer value representing quantization step size for radius of the RPI coordinates of the sensed point. It is specified as:

DeltaRadius := (geom\_quantization\_step\_size × deltaRadiusScale + 128) >> 8  
 where  
 deltaRadiusScale := RadiusScaleFromAzimuth(AzimuthAngleFromAzimuthStepIdx)

The expression AzimuthAngleFromAzimuthStepIdx is a 20-bits decimal precision fixed point number representing the azimuth angle in Radian for the coarse azimuth angle index SensingAzimuthIndex[HeadIdx][PtnIdx] of the sensed point. It is specified as:

AzimuthAngleFromAzimuthStepIdx :=  
 SensingAzimuthIndex[HeadIdx][PtnIdx] × AzimuthAngleStep - 3294199

The expression PtnRecRadius is an 8-bit decimal precision fixed-point number representing the reconstructed radius of the RPI coordinates of the sensed point. It is specified as:

PtnRecRadius := Max(0, radiusPredictor + radius\_res[HeadIdx][PtnIdx] × DeltaRadius)  
 where  
 radiusPredictor :=  
 vertical\_radius\_pred[HeadIdx][PtnIdx]  
 ? VerticalRadiusPredictor  
 : SensingBeamRadiusPred[HeadIdx][ElevationIdx][predIdx]  
 where  
 predIdx := pred\_idx[HeadIdx][PtnIdx]

After decoding a radius\_res[HeadIdx][PtnIdx] syntax element, the variables SensingBeamLastRadiusResSign[ HeadIdx ][ ElevationIdx ] and SensingBeamLastAzimuthStepsFromRadius[ HeadIdx ][ ElevationIdx ] shall be updated.

resR0 := PtnRecRadius - SensingBeamRadiusPred[HeadIdx][ElevationIdx][predIdx]  
if (resR0)  
 SensingBeamLastRadiusResSign[ HeadIdx ][ ElevationIdx ] = resR0 ≥ 0 ? 1 : 0  
 where  
 predIdx :=  
 vertical\_radius\_pred[HeadIdx][PtnIdx]  
 ? 0  
 : pred\_idx[HeadIdx][PtnIdx]

SensingBeamLastAzimuthStepsFromRadius[ HeadIdx ][ ElevationIdx ] =  
 (geom\_use\_num\_azimuth\_steps\_per\_quant\_step  
 && PtnRecRadius  
 && (geom\_quantization\_step\_size << 22) > PtnRecRadius × AzimuthAngleStep)  
 ? Div(azimuthAngleStepQuantStep, PtnRecRadius × AzimuthAngleStep, 12)  
 : 0  
 where  
 azimuthAngleStepQuantStep :=  
 geom\_quantization\_step\_size  
 × AzimuthStepScaleFromAzimuth(AzimuthAngleFromAzimuthStepIdx)

Then the prediction residual for the azimuthal angle of the RPI position of the sensed point azimuth\_res[HeadIdx][PtnIdx] is decoded using following expressions.

The expression CtxAzimuthResSelector , is used for contexts selection when decoding the azimuth residual of the sensed point. It is specified as follows:

CtxAzimuthResSelector :=  
 vertical\_radius\_pred[ HeadIdx ][ PtnIdx ]   
 ? 2  
 : pred\_idx[ HeadIdx ][ PtnIdx ] == 0

The expression CtxAzimuthResSignSelector , is used for contexts selection when decoding the sign of the azimuth residual of the sensed point. It is specified as follows:

CtxAzimuthResSignSelector :=  
 SensingBeamLastAzimuthResSign[HeadIdx][ElevationIdx]

The expression AzimuthResBound , is used for bounding the azimuth residual value when decoding the azimuth residual of the sensed point. It is specified as follows:

AzimuthResBound :=  
 (Div(AzimuthAngleStep, DeltaAzimuth, 6) + 16) >> 5

The expression DeltaAzimuth is an integer value representing rotation angle quantization step size for coarse azimuth (i.e. it represents the angle between two successive coarse azimuth angle indexes). It is specified as:

DeltaAzimuth := dAzimuth < (geom\_minimum\_azimuth\_quantization\_step\_size << 2)  
 ? geom\_minimum\_azimuth\_quantization\_step\_size << 2  
 : dAzimuth  
 where  
 dAzimuth := Div(DeltaRadius, Max(DeltaRadius, PtnRecRadius), 22)

The expression AzimuthAngleStep , is used when decoding the sensed point. It represents the azimuth rotation angle made by one rotation step made by the sensing head, unless it is smaller than geom\_minimum\_azimuth\_quantization\_step\_size, and it is specified as follows:

AzimuthAngleStep :=  
 Max(geom\_minimum\_azimuth\_quantization\_step\_size, azimuthStep)s  
 where  
 azimuthStep := Const2Pi / (beam\_steps\_per\_rotation\_minus1[HeadIdx] + 1)

The expression Const2Pi is an integer value representing a fixed point value with 20 bits decimal precision estimating two times the Pi angle. It is specified as:

Const2Pi := 6588397

After decoding an azimuth\_res[HeadIdx][PtnIdx] syntax element, the variable SensingBeamLastAzimuthResSign[ HeadIdx ][ ElevationIdx ] shall be updated.

if (azimuth\_res[HeadIdx][PtnIdx])  
 SensingBeamLastAzimuthResSign[ HeadIdx ][ ElevationIdx ] =   
 azimuth\_res[HeadIdx][PtnIdx] ≥ 0 ? 1 : 0

The expression PtnRecAzimuthOffset is the reconstructed azimuth angle offset of the sensed point in regards to its coarse azimuth angle. It is specified as follows:

PtnRecAzimuthOffset := azimuthQ  
 where  
 azimuthQ = SensingBeamAzimuthPred[HeadIdx][ElevationIdx][predIdx]  
 + azimuth\_res[HeadIdx][PtnIdx]  
 while (azimuthQ < -AzimuthResBound)  
 azimuthQ += 2 × AzimuthResBound + 1  
 while (azimuthQ > AzimuthResBound)  
 azimuthQ -= 2 × AzimuthResBound + 1  
 where  
 predIdx :=  
 vertical\_radius\_pred[HeadIdx][PtnIdx]  
 ? 0  
 : pred\_idx[HeadIdx][PtnIdx]

The expression PtnRecAzimuth is the reconstructed azimuth angle of the sensed point. It is specified as follows:

PtnRecAzimuth := recAzimuth2  
 where  
 recAzimuth0 := (AzimuthAngleFromAzimuthStepIdx << 4)  
 + PtnRecAzimuthOffset × (DeltaAzimuth << 2)  
 recAzimuth1 := recAzimuth0 < -52707179 ? recAzimuth0 + 105414357 : recAzimuth0  
 recAzimuth2 := recAzimuth1 > 52707179 ? recAzimuth1 - 105414357 : recAzimuth1

After decoding of the radius\_res[HeadIdx][PtnIdx] and of the azimuth\_res[HeadIdx][PtnIdx] syntax elements, the reconstructed RPI coordinates of the sensed point SensedPtnAng[HeadIdx][PtnIdx] are specified as follow:

SensedPtnAng[HeadIdx][PtnIdx][0] := PtnRecRadius  
SensedPtnAng[HeadIdx][PtnIdx][1] := PtnRecAzimuth  
SensedPtnAng[HeadIdx][PtnIdx][2] := ElevationIdx

A X axis coordinate and an Y axis coordinate are reconstructed from the reconstructed radius and reconstructed azimuth as specified by PtnRecX and PtnRecY.

PtnRecX := DivExp2Fz(PtnRecRadius × IntCos(PtnRecAzimuth), 32)

PtnRecY := DivExp2Fz(PtnRecRadius × IntSin(PtnRecAzimuth), 32)

A Z axis coordinate is reconstructed from the reconstructed radius, sensing elevation angle and sensing beam vertical offset as specified by PtnRecY.

PtnRecZ := DivExp2Fz(PtnRecRadius × zElevation - zCorrectionOffset, 26 - zRecPrec)  
 where  
 zElevation := BeamElev[HeadIdx][ElevationIdx]  
 zCorrectionOffset := BeamOffsetV[HeadIdx][ElevationIdx]  
 zRecPrec := geom\_quantization\_step\_size == 64 ? 0 : 8

The X and Y coordinate residuals are specified by PtnResX and PtnResY. When geom\_quantization\_step\_size is equal to 64, they are set to residual x\_res and y\_res and shall be added to the reconstructed X and Y coordinates.

PtnResX := geom\_quantization\_step\_size == 64 ? x\_res[HeadIdx][PtnIdx] : 0

PtnResY := geom\_quantization\_step\_size == 64 ? y\_res[HeadIdx][PtnIdx] : 0

The Z coordinate residual is specified by PtnResZ. It is equal to residual z\_res and shall be added to the reconstructed Z.

PtnResZ := z\_res[HeadIdx][PtnIdx]

After decoding of the residual z\_res[ HeadIdx ][ PtnIdx ]  syntax element, the variable SensingBeamLastZResSign[ HeadIdx ][ ElevationIdx ] shall be updated as follows.

if (z\_res[ HeadIdx ][ PtnIdx ])  
 SensingBeamLastZResSign[HeadIdx][ElevationIdx] = z\_res[ HeadIdx ][ PtnIdx ] > 0 ? 1 : 0

The reconstructed RPI coordinates of the sensed point are converted to Cartesian XYZ coordinates, as specified by SensedPtnAngXyz[ 𝑘 ].

SensedPtnAngXyz[k] := (  
 k == 0 ? PtnRecX + PtnResX :  
 k == 1 ? PtnRecY + PtnResY :  
 k == 2 ? geom\_quantization\_step\_size == 64 ? PtnRecZ + PtnResZ :  
 DivExp2Fz(PtnRecZ + PtnResZ × (geom\_quantization\_step\_size << 2), 8) : na)

These Cartesian XYZ coordinates shall be scaled according to the downscaling parameter of the geometry coordinates, and offset by the slice-relative angular origin, to get the reconstructed XYZ coordinate of a sensed point, as specified by:

SensedPtnCoord[HeadIdx][PtnIdx][k] :=  
 AngularOrigin[k] + (SensedPtnAngXyz[k] << geom\_quantization\_scaling\_log2)

At the end of the decoding of a sensed point, the variable SensingBeamLastPredIdx[HeadIdx][ElevationIdx] shall be set with the decoded predictor index:

SensingBeamLastPredIdx[HeadIdx][ElevationIdx] = pred\_idx[HeadIdx][PtnIdx]

At the end of the decoding of a sensed point, the arrays of predictor variables SensingBeamRadiusPred[ HeadIdx ][ ElevationIdx ] and SensingBeamAzimuthPred[ HeadIdx ][ ElevationIdx ] shall be updated as specified below.

for (predIdx = predIdxLastUpdated; predIdx > 0; predIdx--) {  
 SensingBeamRadiusPred[ HeadIdx ][ ElevationIdx ][predIdx] =  
 SensingBeamRadiusPred[ HeadIdx ][ ElevationIdx ][predIdx - 1]  
 SensingBeamAzimuthPred[ HeadIdx ][ ElevationIdx ][predIdx] =  
 SensingBeamAzimuthPred[ HeadIdx ][ ElevationIdx ][predIdx – 1]  
}  
SensingBeamRadiusPred[ HeadIdx ][ ElevationIdx ][0] = PtnRecRadius  
SensingBeamAzimuthPred[ HeadIdx ][ ElevationIdx ][0] = PtnRecAzimuthOffset  
 where  
 numPredictors = 5  
 newObjectThreshold := 4096 \* 64 / (  
 (1 << geom\_quantization\_scaling\_log2) × geom\_quantization\_step\_size)  
 predIdxLastUpdated :=  
 vertical\_radius\_pred[HeadIdx][PtnIdx]  
 || Abs(radius\_res[HeadIdx][PtnIdx]) > newObjectThreshold  
 ? numPredictors – 1  
 : pred\_idx[ HeadIdx ][ PtnIdx ]

##### Decoding of the low latency attributes

Syntax elements describing low latency attributes of all the sensed points for a sensing head with index HeadIdx are written in specific low latency attributes entropy stream. This low latency attributes entropy stream is embedded within the geometry data unit when geom\_low\_latency\_attributes is equal to 1.

For every sensed point the decoding of the low latency attributes is specified as follows.

A radius value is derived for low latency attributes decoding as specified by PtnRadiusForLLAttr.

PtnRadiusForLLAttr := SensedPtnAng[HeadIdx][PtnIdx][0] >> 8

Then, PtnRadiusForLLAttr shall be compared to the radiuses buffered in SensingBeamAttrBuffLastRadius[ HeadIdx ][ ElevationIdx ] to determine an attribute buffer index, as specified by AttrBuffIdx.

AttrBuffIdx := minRDistIdx  
 where  
 radiusEqualZeroIdx = -1  
 minRDist = 1 << 30  
 minRDistIdx = 0  
 for (buffIdx = 0; buffIdx < 5; buffIdx++) {  
 if (SensingBeamAttrBuffLastRadius[ HeadIdx ][ ElevationIdx ][ buffIdx ] == 0)  
 radiusEqualZeroIdx = buffIdx  
 if (rDist < minRDist) {  
 minRDist = rDist  
 minRDistIdx = buffIdx  
 }  
 }  
 if (radiusEqualZeroIdx ≠ -1 && PtnRadiusForLLAttr < 5 × minRDist) {  
 minRDistIdx = radiusEqualZeroIdx  
 }  
 where  
 rDist := Abs(  
 SensingBeamAttrBuffLastRadius[ HeadIdx ][ ElevationIdx ][ buffIdx ]  
 - PtnRadiusForLLAttr)

The expressions LastAttrValue, LastAttrRes, LastAttrResSign and PrevLastAttrRes shall be derived as the attribute value, the attributes residual, its sign and the previous attribute value buffered at buffer index AttrBuffIdx as specified below.

LastAttrValue := SensingBeamAttrBuffLastAttrValue[HeadIdx][ElevationIdx][AttrBuffIdx]

LastAttrRes := SensingBeamAttrBuffLastAttrRes[HeadIdx][ElevationIdx][AttrBuffIdx]

LastAttrResSign :=  
 SensingBeamAttrBuffLastAttrRes[HeadIdx][ElevationIdx][AttrBuffIdx] < 0 ? 1 : 0

PrevLastAttrRes := SensingBeamAttrBuffPrevLastAttrRes[HeadIdx][ElevationIdx][AttrBuffIdx]

The expression AttrPAE which indicates the peak absolute error for low latency attributes is specified as:

AttrPAE := attrQP == 0  
 ? 0  
 : Floor(Exp2(attrQP ÷ 6.0) ÷ 2.0 + 0.5)  
 where  
 attrQP := low\_latency\_attributes\_quantization\_step\_size

The maximum low latency attributes value MaxAttrValue is specified as:

MaxAttrValue := (1 << low\_latency\_attributes\_bit\_depth) - 1

When attr\_pred[ HeadIdx ][ PtnIdx ]  syntax element is equal to 1, the expressions AttrResBoundLow and AttrResBoundHigh shall be derived as specified below.

AttrResBoundLow := - LastAttrValue

AttrResBoundHigh := MaxAttrValue/(2 × AttrPAE + 1) - LastAttrValue

The decoded low latency attribute value is specified by the expression AttrValue as:

AttrValue := attr\_pred[ HeadIdx ][ PtnIdx ]  
 ? LastAttrValue + attr\_res[ HeadIdx ][ PtnIdx ]  
 : attr\_val[ HeadIdx ][ PtnIdx ]

Finally the reconstructed low latency attribute value is specified by the expression AttrRec[ HeadIdx ][ PtnIdx ] :

AttrRec[ HeadIdx ][ PtnIdx ]  := AttrValue × (2 × AttrPAE + 1) + AttrPAE

At the end of the decoding of the low latency attributes of a sensed point, the buffer arrays for low latency attributes decoding shall be updated as specified below.

SensingBeamAttrBuffLastRadius[ HeadIdx ][ ElevationIdx ][ AttrBuffIdx ] = PtnRadiusForLLAttr

if (attr\_pred[ HeadIdx ][ PtnIdx ]) {  
 SensingBeamAttrBuffPrevLastAttrRes[HeadIdx][ElevationIdx][AttrBuffIdx]  
 = LastAttrRes  
 SensingBeamAttrBuffLastAttrRes[HeadIdx][ElevationIdx][AttrBuffIdx]  
 = attr\_res[ HeadIdx ][ PtnIdx ]  
}

SensingBeamAttrBuffLastAttrValue[HeadIdx][ElevationIdx][AttrBuffIdx]  
 = AttrValue

The reflectance attribute of a sensed point with a low latency reflectance attribute is equal to AttrRec.

### Reconstructed output point positions

The reconstructed output point positions are specified as follow.

ptIdx = 0;  
for (h = 0; h < num\_heads; h++)  
 for (ptIdxHead = 0; ptIdxHead < num\_points[h]; ptIdxHead++, ptIdx++)  
 for (k = 0; k < 3; ++k)  
 PointPos[ptIdx][k] = SensedPtnCoord[h][ptIdxHead][k]

### Reconstructed output low latency attributes values

When geom\_low\_latency\_attributes equals 1, the reflectance component with index AttrIdx is output with its values specified as follow,

ptIdx = 0;  
for (h = 0; h < num\_heads; h++)  
 for (ptIdxHead = 0; ptIdxHead < num\_points[h]; ptIdxHead++, ptIdx++)  
 PointAttr[ptIdx][0] = AttrRec[ h ][ ptIdxHead ]

and no ADU shall be present for the reflectance attribute with index AttrIdx and no attribute parameter set shall be provided for this AttrIdx.

### Reconstructed point positions for attributes decoding

The reconstructed point positions for attributes decoding are specified as follow.

ptIdx = 0;  
for (h = 0; h < num\_heads; h++)  
 for (ptIdxHead = 0; ptIdxHead < num\_points[h]; ptIdxHead++, ptIdx++) {  
 PointAng[ptIdx][0] :=  
 (SensedPtnAng[h][ptIdxHead][0] << geom\_quantization\_scaling\_log2) >> 8  
 PointAng[ptIdx][1] :=  
 SensedPtnAng[h][ptIdxHead][1] >> 12  
 PointAng[ptIdx][2] :=  
 AbsoluteSensingElevationIdx[h][SensedPtnAng[h][ptIdxHead][2]]

# 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, AttrPos is equivalent to PointPos.

Otherwise, AttrPos[ ptIdx ][ 𝑘 ] are angular point coordinates as specified by 10.2.2.

AttrPos[ptIdx][k] := attr\_coord\_conv\_enabled  
 ? AttrPosAng[ptIdx][k]  
 : PointPos[ptIdx][k]

### Conversion to scaled angular coordinates

The conversion is specified by the expression AttrPosAng[ ptIdx ][ 𝑘 ]. The point's angular coordinates shall be offset by the minimum angular coordinates and 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])  
 minAng[k] := k == 1  
 ? -12868  
 : 0

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.

## 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. A transform domain prediction process predicts AC coefficients from the DC coefficients of certain adjoining blocks.

The reconstruction process is specified for a single attribute component Cidx ∈ 0 .. AttrDim − 1. It 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 of coded coefficients (10.5.5) and

applying the inverse transform (10.5.6)

The reconstructed attribute values are specified by 10.5.7.

### Transform tree

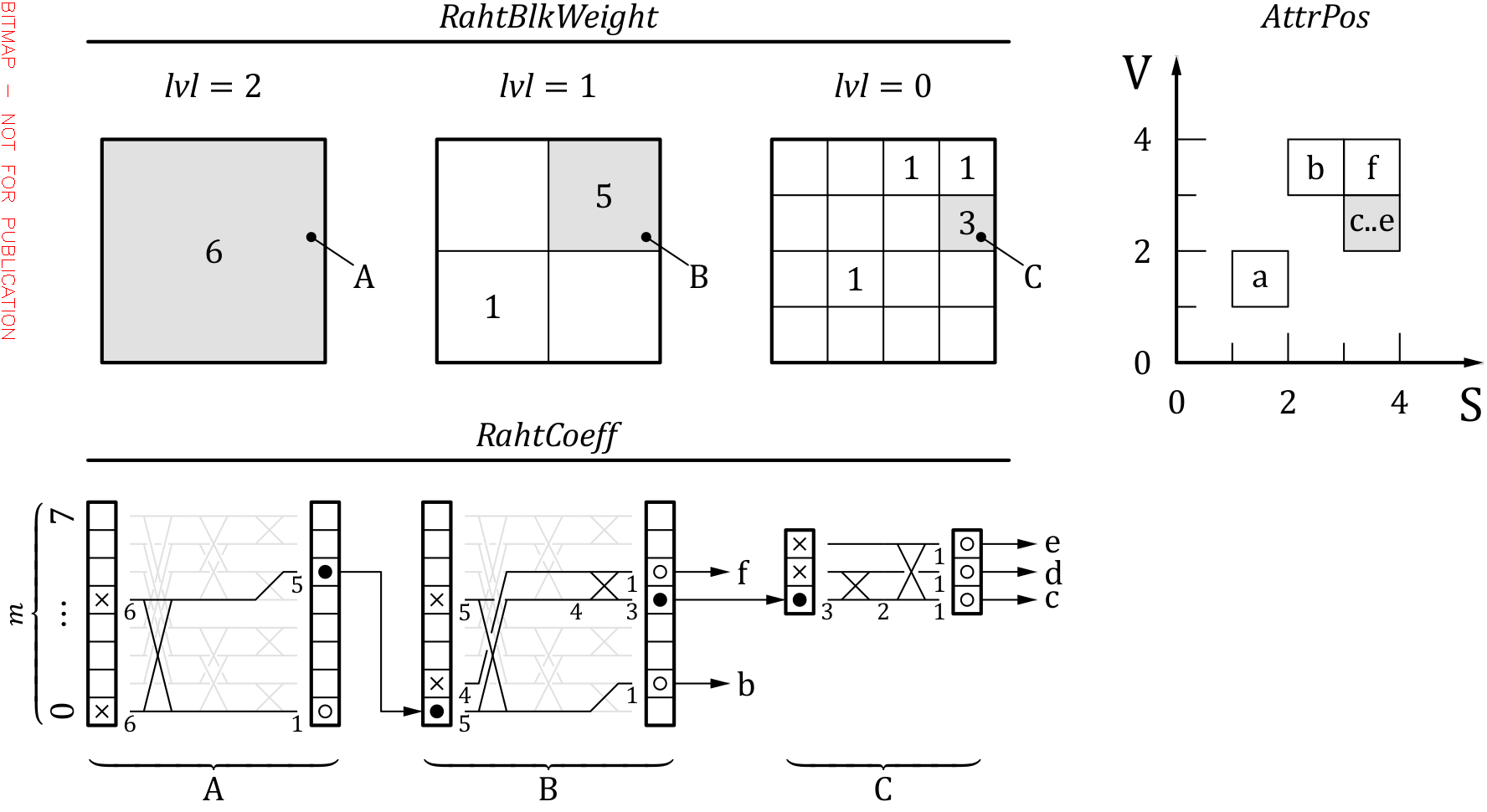
#### General

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 5. 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.



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 7 — 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.

The sum of all block weights in any transform level is equal to the number of coded points (PointCnt).

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 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 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 transform stage and 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 5, 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.

### 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) {  
 … /\* See 10.5.3.2 \*/  
 } else {  
 … /\* 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 13 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 13 — 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 := RahtCoeffQp[lvl][bs][bt][bv][m]

#### 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.

RahtCoeffQp[lvl][bs][bt][bv][m] := AttrQp[dpth][rgnOffset][Cidx > 0]  
 where  
 dpth := RahtRootLvl − lvl  
 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:

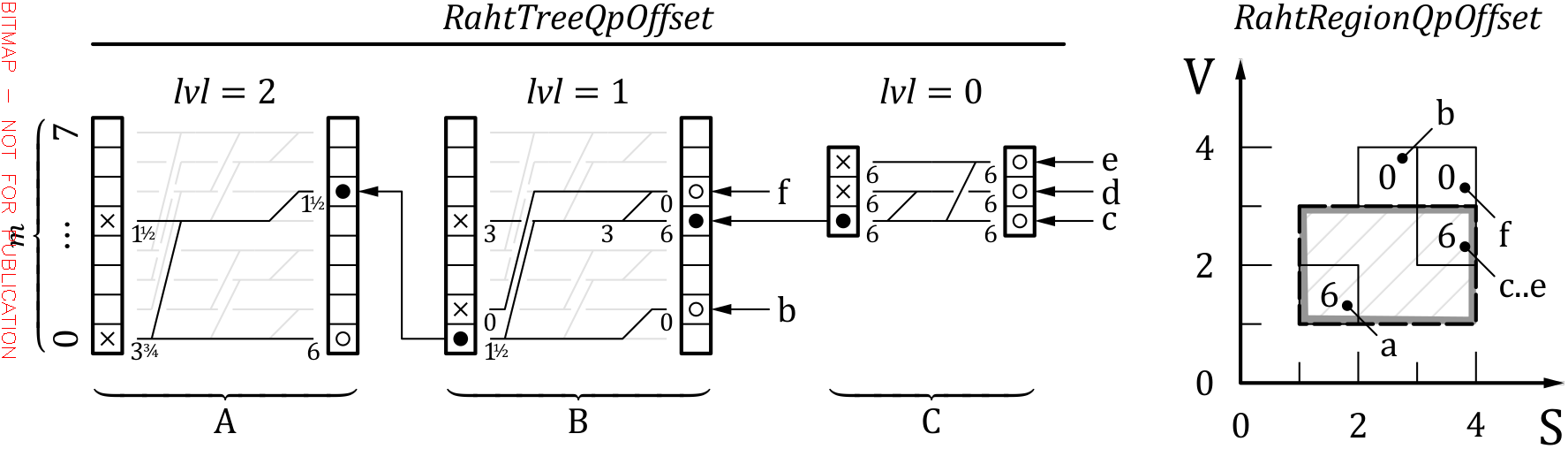
The weight for the corresponding DC coefficient is 𝑎 + 𝑏 divided by 2 if wa and wb are non-zero, or 1 otherwise.

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 6 for the transform tree of Figure 5. 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.



Key

|  |  |
| --- | --- |
| b to f | Points (See Figure 5) |
| A to C | Transform blocks (See Figure 5) |
| × | 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 8 — Example region-dependent QP offset tree

### Transform domain prediction

#### General

Subclause 10.5.5 applies when raht\_prediction\_enabled\_flag is 1. It specifies the transform domain prediction for a block located at ( Bs, Bt, Bv ) in tree level Lvl. It shall be performed for every eligible block (10.5.5.2) in the tree level, in any order, by:

generating a prediction block (10.5.5.4);

weighting the values of the prediction block (10.5.5.5) and applying the forward transform (10.5.5.6);

adding the resulting AC transform coefficients to the coefficient residuals in the coefficient tree.

The prediction block and its transform is specified in terms of the eight-element array RahtPredBlk; RahtPredBlk[ 𝑚 ] is the 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]  
  
 … /\* in−place, forward transform of RahtPredBlk (10.5.5.6) \*/  
  
 for (m = 1; m < 8; m++)  
 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;

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 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  
 && RahtNeighCnt[lvl][bs][bt][bv] ≥ raht\_prediction\_samples\_min  
 && RahtNeighCntMinAncestor[lvl][bs][bt][bv] ≥ raht\_prediction\_subtree\_min

#### 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 ) in tree level lvl.

RahtNeighCnt[lvl][bs][bt][bv] := SumN19[neighWeightGt0]  
 where  
 neighWeightGt0[ds][dt][dv] := RahtBlkWeight[lvl][bs + ds][bt + dt][bv + dv] > 0

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 (10.5.6.2) normalized by their weight as specified by RahtDcNorm; 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] :=  
 DivExp2Fz((coeff >> wShift) × (IntRecipSqrt(w) >> 25 − wShift), 30)  
 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]

##### Upsampled prediction block

The samples of the 2x2x2 prediction block are the weighted averages of the adjoining blocks' normalized DC values. 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 ] 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 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 14.

1. Only samples that correspond to child blocks with non-zero block weights need to be calculated.

RahtPred[m] := SumN19[wNeigh] × RahtPredRecipW[SumN19[w]]  
 where  
 w[ds][dt][dv] := RahtPredExcluded[ds][dt][dv] ? 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 14 — 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 |

#### Weighted prediction block

The forward transform of an upsampled prediction block shall use weighted sample values. Each sample shall be weighted by its 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] := DivExp2Fz(RahtPred[m] × w, 15)  
 where  
 w := IntSqrt(RahtCoeffWeightM[Lvl][0][Bs][Bt][Bv][m] << 30)

#### 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) {  
 yl = DivExp2Fz(aCoeff × a, 15) + DivExp2Fz(bCoeff × b, 15)  
 yh = DivExp2Fz(bCoeff × a, 15) − DivExp2Fz(aCoeff × b, 15)  
}

The transform result replaces the input coefficients:

aCoeff = yl  
bCoeff = yh

### Inverse transform

#### General

Subclause 10.5.6 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.

#### 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. 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] = DivExp2Fz(RahtDcCoeff[Lvl][Bs][Bt][Bv], 13) << 13

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] := RahtCoeff[lvl + 1][bs / 2][bt / 2][bv / 2][mP]  
 where  
 mP := Morton[bs & 1][bt & 1][bv & 1]

#### 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) {  
 ya = wa ≠ 0 ? aCoeff : 0  
 yb = wb ≠ 0 ? aCoeff : 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) {  
 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- and intra-detail-level 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.

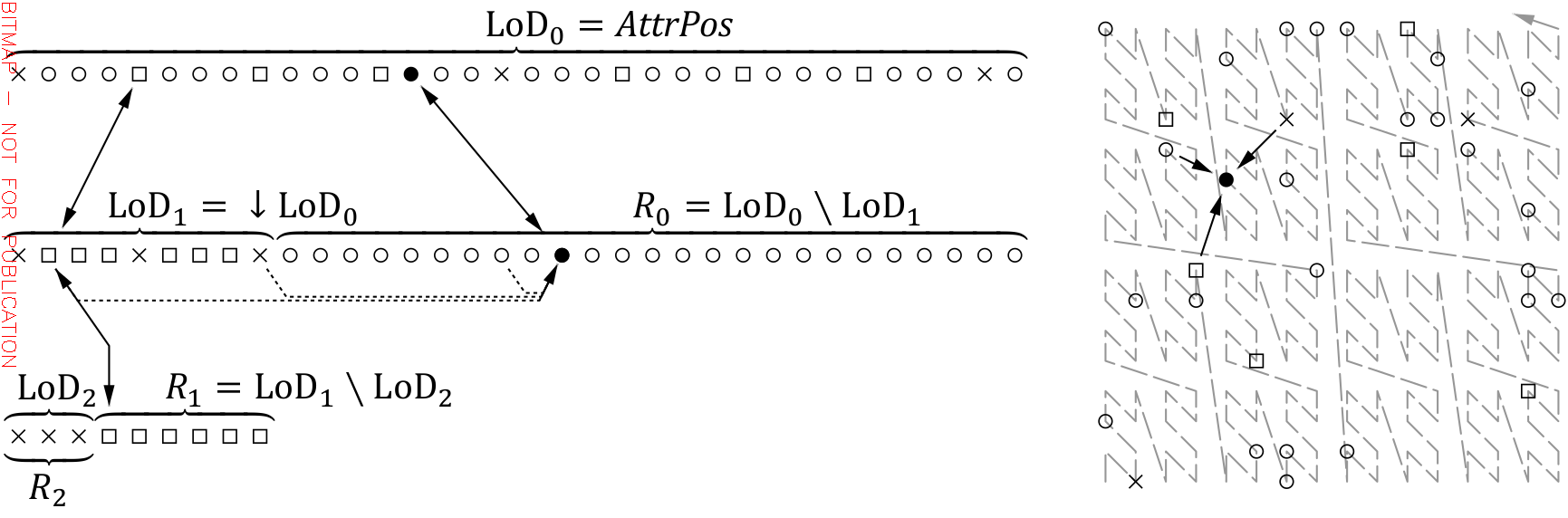


Figure 9 — Example of points in three detail levels and their spatial arrangement.

An example level-of-detail hierarchy is illustrated in Figure 7. 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); 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 of the ni-th point in the predictor set for the identified point.

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.

### 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

Unless attr\_canonical\_order\_enabled is 1, the point indexes of the finest detail level shall be sorted 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 (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 (i = 0; i < LodPtCnt[0]; i++)  
 for (j = i + 1; j < LodPtCnt[0]; 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 the 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

Periodic subsampling generates a subsampled output detail level by sampling every one-in-sampling-period points in the input detail level.

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]  
}

#### 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)

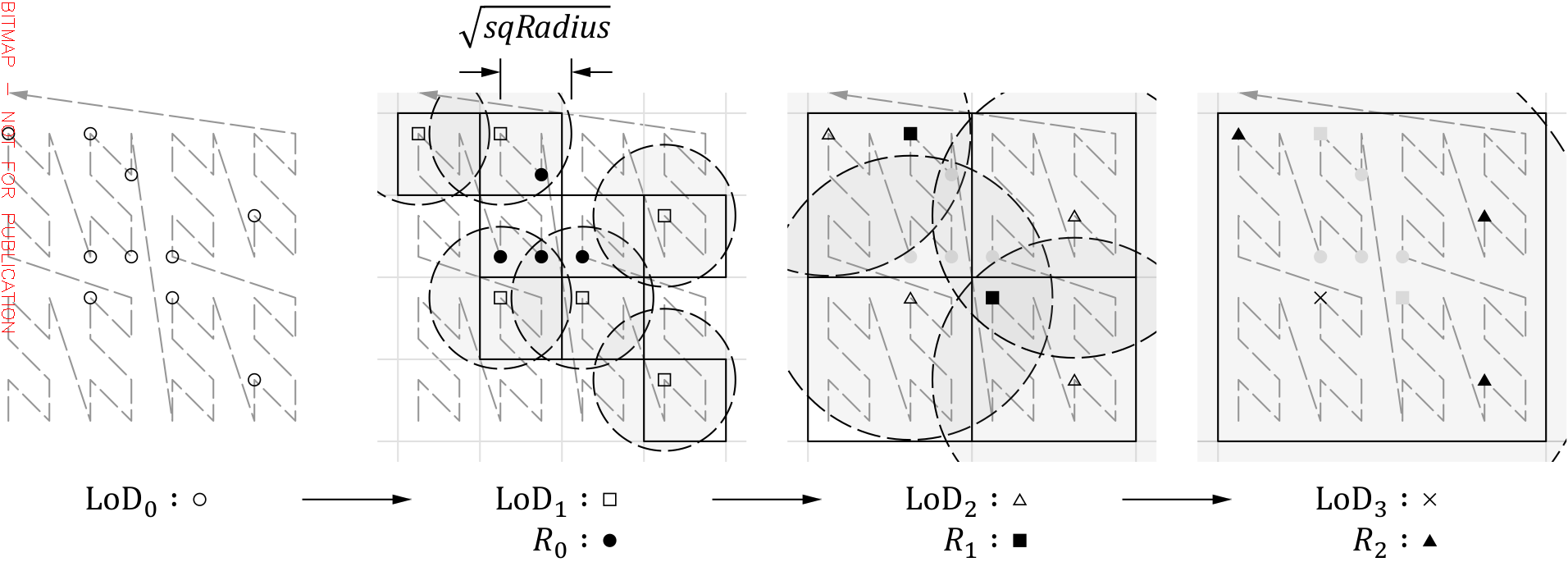


Figure 10 — Example decisions using distance-based subsampling.

Example per-point decisions are illustrated in Figure 8. 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 15 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 15 — 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

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.

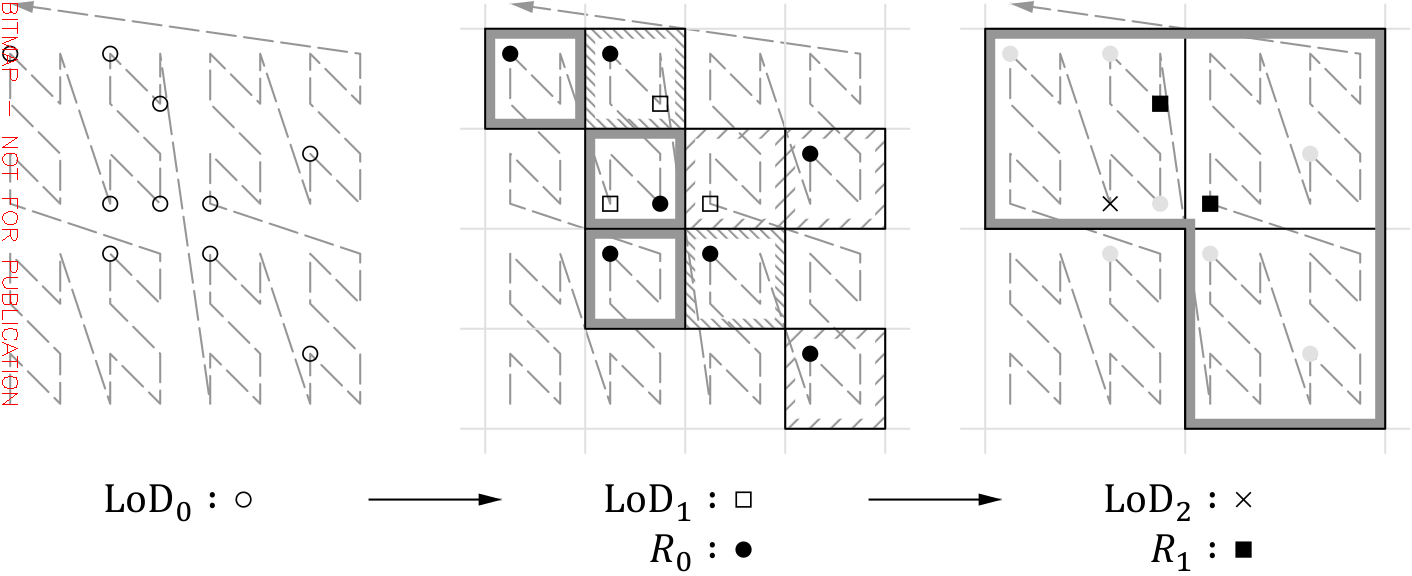


Figure 11 — Example decisions using block group subsampling with minGrpPts = 3.

Example per-point decisions are illustrated in Figure 9. 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

### 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 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.

After completing the searches, weights shall be calculated for each predictor (10.6.6.9), during which the predictor set is pruned and re-sorted. When pred\_blending\_enabled is 1, predictor weights shall be blended (10.6.6.10).

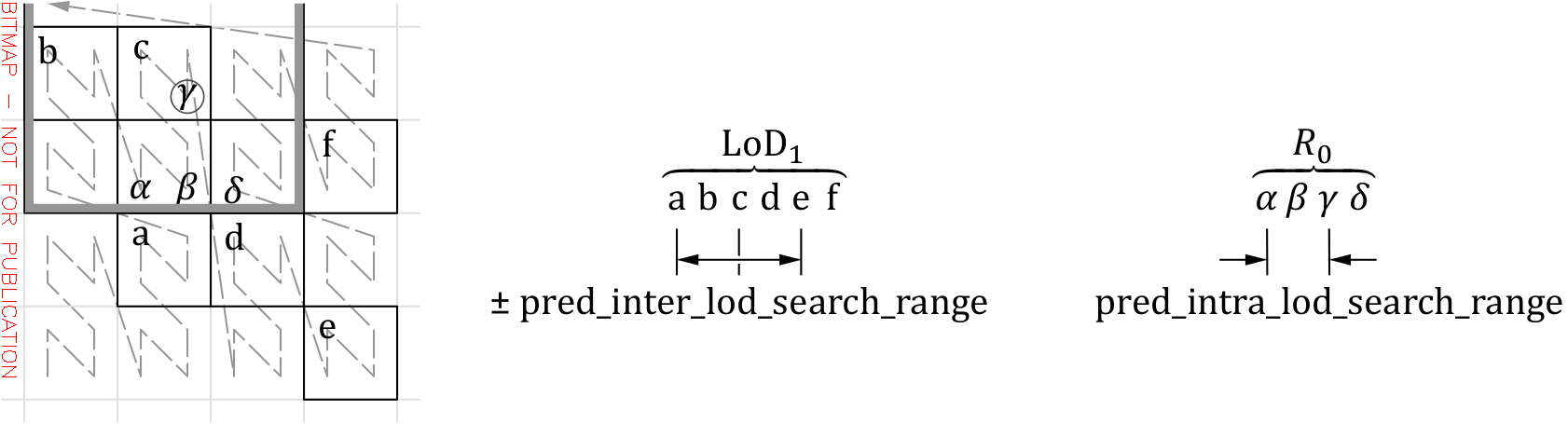


Figure 12 — Example of searches performed for a single refinement point.

An example predictor search is illustrated in Figure 10. 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 ) 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 of the candidate point.

A candidate shall only be inserted into the a 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

Otherwise (the candidate is not already present), the spatial distance between the candidate and the refinement point shall be used to decide the inclusion in the predictor set. The distance shall be calculated as the biased norm weighted by PredBias.

dist = BiasedNorm1(PtIdx, candPtIdx)

The point shall be inserted into the predictor set, with elements ordered according to the biased distance to the refinement 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]))  
 break  
for (j = PredCnt[PtIdx]; j > i; j−−)  
 PredPtIdx[PtIdx][j] = PredPtIdx[PtIdx][j − 1]  
PredPtIdx[PtIdx][i] = candPtIdx

The size of the predictor set shall be limited to three elements by discarding the furthest predictor if necessary.

PredCnt[PtIdx] = Min(3, PredCnt[PtIdx] + 1)

#### Distance computation using the biased L1 norm (BiasedNorm1)

This subclause defines the function BiasedNorm1( ptIdxA, ptIdxB ) that is the weighted Manhattan distance between two points.

The parameters ptIdxA and ptIdxB are two AttrPos indexes.

The result of this function is specified by the expression BiasedNorm1. The expression pos[ ptIdx ][ 𝑘 ] specifies 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) := dist[0] + dist[1] + dist[2]  
 where  
 dist[k] := Abs(pos[ptIdxA][k] − pos[ptIdxB][k]) × PredBias[k]  
 pos[ptIdx][k] := lod\_scalability\_enabled  
 ? (AttrPos[ptIdx][k] >> Lvl) << Lvl  
 : AttrPos[ptIdx][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 16. 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)  
 }  
}

1. For each search block, the indices 𝑖 for which inSblk is true are consecutive.

Table 16 — 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 extended search shall proceed over each index offset 𝑖 of the following, in order: 0, +1, −1, +2, −2, +3 .. pred\_inter\_lod\_search\_range, and −( 3 .. pred\_inter\_lod\_search\_range ).

A predictor candidate shall be evaluated for each valid search index centre + 𝑖 that is within the range specified by pred\_inter\_lod\_search\_range and does not exceed the bounds of the reference detail level.

if (Abs(i) ≤ pred\_inter\_lod\_search\_range)  
 if (centre + i ≥ 0 && centre + i < LodPtCnt[RefLvl])  
 InsertPredictor(LodPtIdx[RefLvl][centre + i])

#### 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.

A predictor candidate shall be evaluated for each valid search index offset −𝑖 from the refinement point, for 𝑖 = 1 .. pred\_inter\_lod\_search\_range, that does not exceed the bounds of the refinement list.

for (i = 1; i ≤ Min(RfmtIdx, pred\_intra\_lod\_search\_range); i++)  
 InsertPredictor(LodRfmtPtIdx[Lvl][RfmtIdx − i])

#### 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])

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])

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]) > threshold) {  
 PredCnt[PtIdx] = ni  
 break  
 }  
}

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])  
distB := Norm2(PredPtIdx[PtIdx][0], PredPtIdx[PtIdx][2])  
distC := Norm2(PredPtIdx[PtIdx][1], PredPtIdx[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 ) that is the weighted squared distance between two points.

The parameters ptIdxA and ptIdxB are two AttrPos indexes.

The result of this function is specified by the expression BiasedNorm2. The expression pos[ ptIdx ][ 𝑘 ] represents 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) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(pos[ptIdxA][k] − pos[ptIdxB][k]) × PredBias[k]  
 dist2[k] := dist[k] × dist[k]  
 pos[ptIdx][k] := lod\_scalability\_enabled  
 ? (AttrPos[ptIdx][k] >> Lvl) << Lvl  
 : AttrPos[ptIdx][k]

#### Distance computation using the unbiased L2 norm (Norm2)

This subclause defines the function Norm2( ptIdxA, ptIdxB ) that is the squared distance between two points.

The arguments ptIdxA and ptIdxB are two AttrPos indexes.

The result of this function is specified by the expression Norm2. The expression pos[ ptIdx ][ 𝑘 ] represents 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) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(pos[ptIdxA][k] − pos[ptIdxB][k])  
 dist2[k] := dist[k] × dist[k]  
 pos[ptIdx][k] := lod\_scalability\_enabled  
 ? (AttrPos[ptIdx][k] >> Lvl) << Lvl  
 : AttrPos[ptIdx][k]

### 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 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 Default.

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++) {  
 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++) {  
 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] += 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] × PointAttr[PredPtIdx[PtIdx][ni]][c]  
 PointAttr[PtIdx][c] += DivExp2Fz(sum, 8)  
 }

## 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.

### 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 17.

Table 17 — Values of AttrLevelScale[ 𝑖 ]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 |
| AttrLevelScale[ 𝑖 ] | 161 | 181 | 203 | 228 | 256 | 287 |

# Parsing process

## General

Syntax elements are parsed according to the processes corresponding to the syntax element’s descriptor and name as specified in Tables 18 to 20.

Table 18 — Descriptor parsing processes

| Descriptor | Parsing process | Arguments | Channel read method (readBin) |
| --- | --- | --- | --- |
| u(𝑛) | 11.4.1 | maxBins = 𝑛 | DuNextBit (11.2.5) |
| u(v) | 11.4.1 | See Table 19 | 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 19 | 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 20 | See Table 20 | AeReadBin (11.5.2) |
| de(v) | 1.1 |  | na |

Table 19 — 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 |

Table 20 — Syntax element specific parsing processes (ae(v))

| Syntax element | Parsing process | Arguments |
| --- | --- | --- |
| num\_coarse\_position\_duplicates[ ][ ] | 11.4.4 (TU+EGk) | maxOffset = 5, 𝑘 = 1 |
| dup\_point[ ][ ] | 11.4.1 (FL) | numBins = 1 |
| late\_point[ ][ ] | 11.4.1 (FL) | numBins = 1 |
| elevation\_offset | 11.4.8 (TU) | maxVal = num\_beams\_minus1[ HeadIdx ] + 1    − ElevationIdx |
| azimuth\_offset | 11.4.8 (TU) | maxVal = infinity |
| pred\_idx[  ][  ] | 11.4.8 (TU) | maxVal = 4 |
| vertical\_radius\_pred[  ][  ] | 11.4.1 (FL) | numBins = 1 |
| radius\_res[  ][  ] | 11.4.6 (TU+EGK+S) | maxOffset = 3, 𝑘 = 1 + vertical\_radius\_pred[ HeadIdx ][ PtnIdx ] |
| azimuth\_res[  ][  ] | 11.4.7 (BTU+EGK+S) | maxOffset = 3, bound = AzimuthResBound, 𝑘 = 1 |
| x\_res[  ][  ] | 11.4.6 (TU+EGK+S) | maxOffset = 3, 𝑘 = 1 |
| y\_res[  ][  ] | 11.4.6 (TU+EGK+S) | maxOffset = 3, 𝑘 = 1 |
| z\_res[  ][  ] | 11.4.6 (TU+EGK+S) | maxOffset = 3, 𝑘 = 1 |
| attr\_pred[ ][ ] | 11.4.1 (FL) | numBins = 1 |
| attr\_res[  ][  ] | 11.4.8 (ABTU+EGK+S) | maxOffset = 3, boundLow = AttrResBoundLow, boundHigh = AttrResBoundHigh, 𝑘 = 0 |
| attr\_val[ ][ ] | 11.4.4 (TU+EGk) | maxOffset = 3, 𝑘 = 0 |
| coeff\_abs | 11.4.4 (TU+EGk) | maxOffset = 2, 𝑘 = 1 |
| coeff\_sign | 11.4.1 (FL) | numBins = 1 |
| end\_of\_entropy\_stream | 11.4.1 (FL) | numBins = 0 |
| 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  
  
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 11. It starts with two ChunkLen length chunks. From the CBS, two subtreams, AeBits and BpBits are extracted.

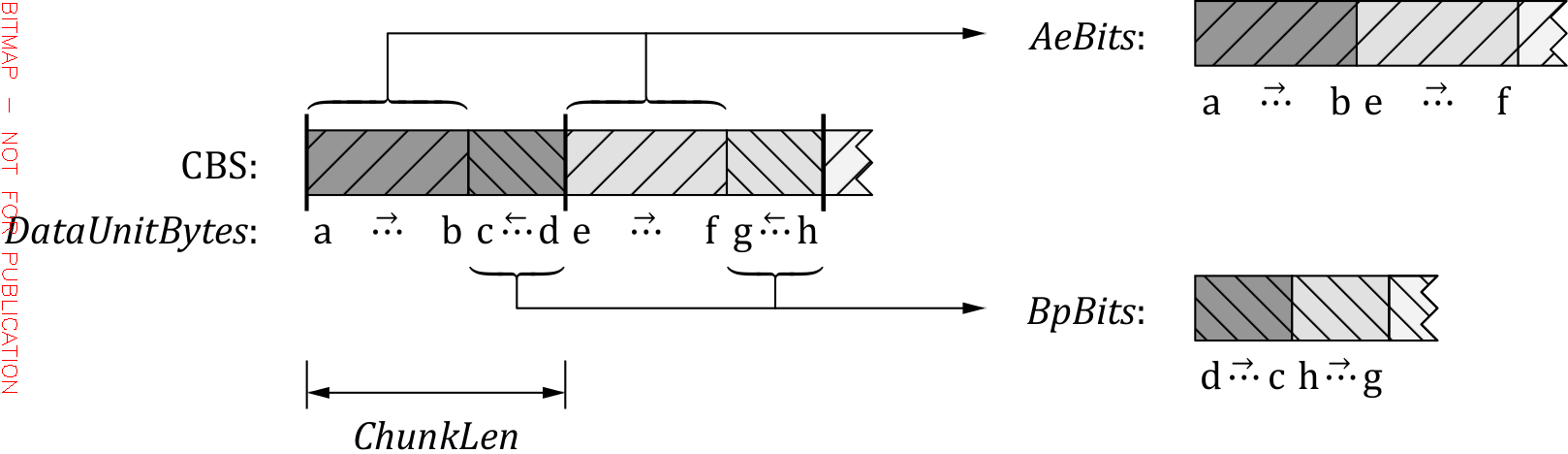


Figure 13 — Multiplexed data streams in a chunked bytestream.

When stream\_cnt\_minus1 is greater than 0, each entropy stream 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 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.

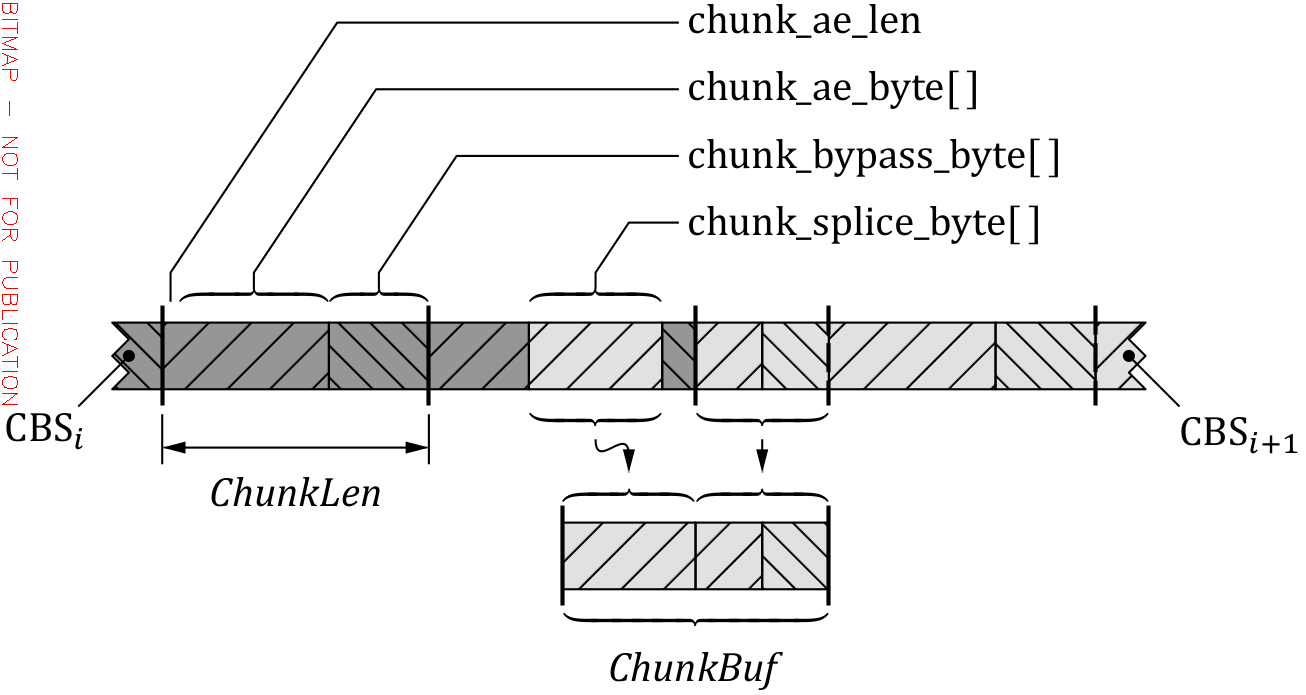


Figure 14 — Extraction of the first chunk from a spliced CBS.

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 12. 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 stream\_cnt\_minus1 is greater than 0.

An additional CBS shall commence at the start of each entropy stream.

## General inverse binarization processes

### Parsing unsigned fixed-length codes (FL)

Parsing is parameterized by:

numBins, the number of bins that represent the syntax element;

readBin, the channel read method expression.

The result is the unsigned syntax element value PartVal, parsed and constructed as:

PartVal = 0  
for (BinIdx = 0; BinIdx < numBins; BinIdx++)  
 PartVal = (PartVal << 1) + readBin

### Parsing signed fixed-length codes (FL+S)

Parsing is parameterized by:

numBins, the number of bins that represent the absolute syntax element value;

readBin, the channel read method expression.

The unsigned syntax element magnitude is parsed:

PartVal = 0  
for (BinIdx = 0; BinIdx < numBins; BinIdx++)  
 PartVal = (PartVal << 1) + readBin

The result is the signed syntax element value val, parsed and constructed as:

sign = readBitval = sign ? −PartVal : PartVal

### Parsing 𝑘-th order exp-Golomb codes (EGk)

Parsing is parameterized by:

𝑘, the order of the exp-Golomb code;

readBin, the channel read method expression.

First, a unary encoded prefix is parsed as:

prefix = 0  
for (BinIdxPfx = 0; readBin ≠ 0; BinIdxPfx++)  
 prefix++

Then, a suffix comprising 𝑘 + prefix bins is parsed:

suffix = 0  
for (BinIdxSfx = 0; BinIdxSfx < k + prefix; BinIdxSfx++)  
 suffix = (suffix << 1) + readBin

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;

readBin, the channel read method expression.

First, a truncated unary encoded offset is parsed:

offset = 0  
for (BinIdxTu = 0; offset < maxOffset && readBin == 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; readBin ≠ 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) + readBin

The result is the unsigned syntax element value val, constructed as:

val = offset + Exp2(prefix + k) + suffix − Exp2(k)

### Parsing concatenated bounded truncated unary and 𝑘-th order exp-Golomb codes (BTU+EGk)

Parsing is parameterized by:

maxOffset, the limit for the bounded truncated unary offset encoding;

bound, the bound for the bounded truncated unary offset encoding;

𝑘, the order of the exp-Golomb code;

readBin, the channel read method expression.

First, a bounded truncated unary encoded offset is parsed:

offset = 0  
for (BinIdxBTu = 0; offset < bound && offset < maxOffset && readBin == 1; BinIdxBTu++)  
 offset++

Second, if the value of offset is lower than bound and equal to maxOffset, a unary encoded prefix is parsed:

prefix = 0  
if (offset < bound && offset == maxOffset)  
 for (BinIdxPfx = 0; readBin ≠ 0; BinIdxPfx++)  
 prefix++

Then, if the value of offset is lower than bound and equal to maxOffset, a suffix comprising 𝑘 + prefix bins is parsed:

suffix = 0  
if (offset < bound && offset == maxOffset)  
 for (BinIdxSfx = 0; BinIdxSfx < k + prefix; BinIdxSfx++)  
 suffix = (suffix << 1) + readBin

The result is the unsigned syntax element value val, constructed as:

val = offset + Exp2(prefix + k) + suffix – Exp2(k)

### Parsing signed concatenated truncated unary and 𝑘-th order exp-Golomb codes (TU+EGk+S)

Parsing is parameterized by:

maxOffset, the limit for the truncated unary offset encoding;

𝑘, the order of the exp-Golomb code;

readBin, the channel read method expression.

First, the absolute value of the result absVal, a concatenated truncated unary and 𝑘-th order exp-Golomb code (TU+EGk), is parsed using the parsing process 11.4.4 parameterized by the same parameters.

If it is not equal to zero the sign syntax element is then parsed. The value of the result is the signed syntax element value val, constructed as:

val = absVal  
if (absVal > 0) {  
 sign = readBin  
 if (sign)  
 val = -absVal  
}

### Parsing signed concatenated bounded truncated unary and 𝑘-th order exp-Golomb codes (BTU+EGk+S)

Parsing is parameterized by:

maxOffset, the limit for the bounded truncated unary offset encoding;

bound, the bound for the bounded truncated unary offset encoding;

𝑘, the order of the exp-Golomb code;

readBin, the channel read method expression.

First, the absolute value of the result absVal, a concatenated bounded truncated unary and 𝑘-th order exp-Golomb code (BTU+EGk), is parsed using the parsing process 11.4.5 parameterized by the same parameters.

If it is not equal to zero the sign syntax element is then parsed. The value of the result is the signed syntax element value val, constructed as:

val = absVal  
if (absVal > 0) {  
 sign = readBin  
 if (sign)  
 val = -absVal  
}

### Parsing signed concatenated asymmetrically bounded truncated unary and 𝑘-th order exp-Golomb codes (ABTU+EGk+S)

Parsing is parameterized by:

maxOffset, the limit for the absolute value of the asymmetrically bounded truncated unary offset encoding;

boundLow, the lowest bound for the asymmetrically bounded truncated unary offset encoding;

boundHigh, the highest bound for the asymmetrically bounded truncated unary offset encoding;

𝑘, the order of the exp-Golomb code;

readBin, the channel read method expression.

First, an absolute value of the asymmetrically bounded truncated unary encoded offset is parsed as offset syntax element together with its sign indicated by the sign syntax element when offset is not equal to zero, and a bound is derived for the absolute value from boundLow and boundHigh:

offset = 0  
BinIdxABTu = 0  
if (readBin == 1) {  
 BinIdxABTu++  
 offset++  
}

sign = 0  
if (offset) {  
 if (boundLow < 0 && boundHigh > 0)  
 sign = readBin  
 else if (boundHigh ≤ 0)  
 sign = 1  
}

bound = sign ? -boundLow : boundHigh

for (BinIdxABTu = 1; offset < bound && offset < maxOffset && readBin == 1; BinIdxABTu++)  
 offset++

Second, if the value of offset is lower than bound and equal to maxOffset, a unary encoded prefix is parsed:

prefix = 0  
if (offset < bound && offset == maxOffset)  
 for (BinIdxPfx = 0; readBin ≠ 0; BinIdxPfx++)  
 prefix++

Then, if the value of offset is lower than bound and equal to maxOffset, a suffix comprising 𝑘 + prefix bins is parsed:

suffix = 0  
if (offset < bound && offset == maxOffset)  
 for (BinIdxSfx = 0; BinIdxSfx < k + prefix; BinIdxSfx++)  
 suffix = (suffix << 1) + readBin

The absolute value of the result is the unsigned syntax element value absVal, constructed as:

absVal = offset + Exp2(prefix + k) + suffix – Exp2(k)

The value of the result is the signed syntax element value val, constructed as:

val = sing ? -absVal : absVal

### Parsing truncated unary codes (TU)

Parsing is parameterized by:

maxVal, the limit for the encoding;

readBin, 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 && readBin == 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 21.

Table 21 — 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 any L3C2 entropy stream.

### 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, initialization shall be performed by the parsing state restoration process (11.6).

Otherwise (slice\_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 22:

if (binVal)  
 Ctx −= CtxUpdateDelta[Ctx >> 8]  
else  
 Ctx += CtxUpdateDelta[255 − (Ctx >> 8)]

Table 22 — 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 23 (GDU) and Table 24 (ADU). Entries qualified by Offset, Prefix, Suffix, or Sign 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 23 — Values of CtxTbl and CtxIdx for binarized ae(v) coded GDU syntax elements

| Syntax element | CtxTbl | CtxIdx | | Count |
| --- | --- | --- | --- | --- |
| num\_coarse\_position\_duplicates[ ][] | 1 | Offset | For BinIdxTu < 3:    8 × BinIdxTu + ElvPackIdx  For BinIdxTu ≥ 3:    24 + BinIdxTu − 3 | 26 |
| **Prefix** | 26 + Min( 7, BinIdxPfx ) | 8 |
| **Suffix** | bypass | 0 |
| dup\_point[ ][ ] | 2 | ElvPackIdx | | 8 |
| late\_point[ ][ ] | 3 | 8 × ElvPackIdxLast   + BufferLate[ HeadIdx ][ElvPackIdxLast] | | 64 |
| elevation\_offset | 4 | CtxIdxElvOffset (9.2.4.1) | | 264 |
| azimuth\_offset | 5 | 2 × late\_point[ HeadIdx ][ PtnIdx ]   + Min( 1, BinIdxTu) | | 4 |
| pred\_idx[  ][  ] | 6 | For BinIdx == 0:    CtxPredIdxSelector  For BinIdx > 0:    3 + BinIdx | | 7 |
| vertical\_radius\_pred[  ][  ] | 7 | ElvPackIdx | | 8 |
| radius\_res[  ][  ] | 8 | **Offset** | 10 × BinIdxTu   + CtxRadiusResSelector | 30 |
| **Prefix** | 30 + 8 × CtxRadiusResSelector   + Min( 7, BinIdxPfx ) | 80 |
| **Suffix** | Bypass | 0 |
| **Sign** | 110 + 8 × CtxRadiusResSelector   + CtxRadiusResSignSelector | 80 |
| azimuth\_res[  ][  ] | 9 | **Offset** | 3 × BinIdxBTu   + CtxAzimuthResSelector | 9 |
| **Prefix** | 9   + (AzimuthResBound – 3 > 6     ? 1 : 0)   + 2 × Min( 7, BinIdxPfx ) | 16 |
| **Suffix** | Bypass | 0 |
| **Sign** | 25 + 2 × CtxRadiusResSelector   + CtxAzimuthResSignSelector | 6 |
| x\_res[  ][  ] | 10 | **Offset** | BinIdxTu | 3/11 |
| **Prefix** | 11 | 1 |
| **Suffix** | Bypass | 0 |
| **Sign** | 12 | 1/17 |
| y\_res[  ][  ] | 10 | **Offset** | BinIdxTu | 3/11 |
| **Prefix** | 11 | 1 |
| **Suffix** | Bypass | 0 |
| **Sign** | 12 | 1/17 |
| z\_res[  ][  ] | 10 | **Offset** | For BinIdxTu == 0:    3 + ElvPackIdx For BinIdxTu > 0:    BinIdxTu | 10/11 |
| **Prefix** | 11 | 1 |
| **Suffix** | Bypass | 0 |
| **Sign** | 13   +2 × ElvPackIdx   + SensingBeamLastZResSign     [HeadIdx][ElevationIdx] | 16/17 |
| attr\_pred[  ][  ] | 11 | 0 | | 1 |
| attr\_res[  ][  ] | 12 | **Offset** | For BinIdxABTu == 0:    ElvPackIdx + 8 × (     1 + 2 × PrevLastAttrRes     + 4 ×       Min(3,Abs(LastAttrRes)) For BinIdxABTu > 0:    128 + 64 × (BinIdxABTu – 1)   + ElvPackIdx + 8 × (     1 + 2 ×       Min(3, Abs(LastAttrRes)) | 232/256 |
| **Prefix** | 256 + ElvPackIdx + 8 × Min( 7, BinIdxPfx ) | 64 |
| **Suffix** | Bypass | 0 |
| **Sign** | 320 + (LastAttrResSign) | 2 |
| attr\_val[ ][] | 12 | **Offset** | For BinIdxTu == 0:    ElvPackIdx  For BinIdxTu > 0:    128 + 64 × (BinIdxTu – 1)    + ElvPackIdx | 24/256 |
| **Prefix** | 256 + ElvPackIdx + 8 × Min( 7, BinIdxPfx ) | 64 |
| **Suffix** | bypass | 0 |
| end\_of\_entropy\_stream | na | terminate | | 0 |
| The syntax elements x\_res, y\_res and z\_res use the same context table. Count column gives number of contexts actually used among all shared contexts.. The syntax elements attr\_res and attr\_val use the same context table. | | | | |

Table 24 — Values of CtxTbl and CtxIdx for binarized ae(v) coded ADU syntax elements

| Syntax element | CtxTbl | CtxIdx | | Count |
| --- | --- | --- | --- | --- |
| coeff\_abs[ 0 ] | 38 | **Offset** | BinIdxTu | 2 |
| **Prefix** | 4 + Min( 4, BinIdx Pfx) | 3 |
| **Suffix** | 9 + Min( 2, BinIdx Sfx) | 3 |
| coeff\_abs[ 1 ] | 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 ] | 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\_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 |
| 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 at the beginning of any entropy stream; 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

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  
}

#### 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 geometry or low latency attributes 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

## Sensor Coding State

### State variable

Sensor coding state data are used and maintained during the decoding process of a GDU. These data are specified in terms of the following state variables:

The array SensingHeadStartedCapturing of Boolean values; SensingHeadStartedCapturing[ HeadIdx ] is equal to 1 if at least one sensed point has already been decoded for the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1); otherwise it is equal to 0.

The arrays SensingHeadLastAzimuthIdx, and SensingHeadLastElevationIdx, two arrays of integer indices; SensingHeadLastAzimuthIdx[ HeadIdx ] and SensingHeadLastElevationIdx[ HeadIdx ] represent the coarse position of the last sensed point decoded for the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no sensed point has already been decoded for a specific head, they represent a default coarse position, as specified in subclause 11.6.2.

The array SensingHeadNeighOccupancyBits of neighbours’ occupancy; SensingHeadNeighOccupancyBits[ HeadIdx ][ElevationIdx ] is a 3-bit word, where each bit represents the occupancy of the 3 preceding coarse position with elevation index equal to ElevationIdx and the sensing head with index HeadIdx. A bit equal to 1 means the neighbour coarse position is occupied. A bit equal to 0 may indicate that the neighbour coarse position is non occupied, or that it may be occupied but occupancy has been decoded earlier in a non-lexicographic order (at least one late\_point has been decoded), or that the occupancy of the neighbour coarse position has not been decoded. When the number of sensing elevations num\_sensing\_ranges\_minus1[  HeadIdx] + 1 is lower than 3, SensingHeadNeighOccupancyBits[ HeadIdx ][ElevationIdx ] shall be defined for any elevation index ElevationIdx lower than or equal to 2.

The array SensingHeadLastSensingRangeIdx of last sensing range indices; SensingHeadLastAzimuthIdx[ HeadIdx ] is the sensing range index of the last sensed point decoded for the sensing head with index HeadIdx, in the current slice. If no sensed point has already been decoded for a specific head, it represents the default value for last sensing range index, as specified in subclause 11.6.2.

The array BufferLate of 3-bit words for each one of the 8 packs of sensing elevation angles; BufferLate[ HeadIdx ][packIdx] is a 3-bit word representing the value of 3 late\_point[ HeadIdx ][ ptIdx ] syntax elements previously decoded in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). The value of ptIdx for these 3 syntax elements is equal to the points indexes of the 3 last sensed points decoded with a coarse position CoarsePosition[ HeadIdx ][ ptIdx ]  for which the coarse position has a sensing elevation angle index SensingElevationIdx[HeadIdx][ ptIdx] which belongs to the pack of sensing elevation angles with index packIdx; i.e. ElevationPackIndex[ HeadIdx ][ SensingElevationIdx[HeadIdx][ ptIdx] ]  is equal to packIdx. If no or less than 3 sensed points has already been decoded for a specific pack of sensing elevation angles, the 3-bit word is padded with zeros, as specified in subclause 11.6.2.

The array SensingBeamLastAzimuthIdx, an array of integer indices; SensingBeamLastAzimuthIdx[ HeadIdx ][ ElevationIdx ] represents the coarse azimuth position of the last sensed point decoded for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no sensed point has already been decoded for a specific beam, it represents a default coarse azimuth position, as specified in subclause 11.6.2.

The array SensingBeamPrevLastAzimuthIdx, an array of integer indices; SensingBeamPrevLastAzimuthIdx[ HeadIdx ][ ElevationIdx ] represents the coarse azimuth position of the last previous sensed point to the last sensed point decoded for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no previous to the last sensed point or no sensed point has already been decoded for a specific beam, it represents a default coarse azimuth position, as specified in subclause 11.6.2.

The array SensingBeamLastAzimuthStepsFromRadius, an array of integer indices; SensingBeamLastAzimuthStepsFromRadius [ HeadIdx ][ ElevationIdx ] represents a number of coarse azimuth indexes covered by quantization in cartesian domain around the position of the last sensed point decoded for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). This number of azimuth indexes is estimated from the radius of the point, as specified un subclause 9.2.5.3.2. If no sensed point has already been decoded for a specific beam, it represents a default coarse azimuth position, as specified in subclause 11.6.2.

The array SensingBeamLastRadiusResSign, an array of Boolean values; SensingBeamLastRadiusResSign[ HeadIdx ][ ElevationIdx ] represents the sign of the residual of a prediction of last decoded radius for the last decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). A value of 1 indicates the residual of the prediction was positive or equal to zero and a value of 0 indicates it was negative. If no sensed point has already been decoded for a specific beam, it is set to zero, as specified in subclause 11.6.2.

The array SensingBeamLastAzimuthResSign, an array of Boolean values; SensingBeamLastAzimuthResSign[ HeadIdx ][ ElevationIdx ] represents the sign of the decoded residual for last decoded azimuthal angle for the last decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). A value of 1 indicates the residual was positive or equal to zero and a value of 0 indicates it was negative. If no sensed point has already been decoded for a specific beam, it is set to zero, as specified in subclause 11.6.2.

The array SensingBeamLastZResSign, an array of Boolean values; SensingBeamLastZResSign[ HeadIdx ][ ElevationIdx ] represents the sign of the decoded residual for last decoded z coordinate for the last decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). A value of 1 indicates the residual was positive and a value of 0 indicates it was negative or equal to zero. If no sensed point has already been decoded for a specific beam, it is set to zero, as specified in subclause 11.6.2.

The array SensingBeamLastPredIdx, an array of integer values; SensingBeamLastPredIdx[ HeadIdx ][ ElevationIdx ] represents the index of the predictor used for predicting the coordinates of the last decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx, in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no sensed point has already been decoded for a specific beam, it is set to zero, as specified in subclause 11.6.2.

The arrays SensingBeamRadiusPred and SensingBeamAzimuthPred, two arrays of integer values; SensingBeamRadiusPred[ HeadIdx ][ ElevationIdx ][ PredIdx ] and SensingBeamAzimuthPred[ HeadIdx ][ ElevationIdx ][ PredIdx ] represent a radius prediction and an azimuthal angle prediction for the predictor with index PredIdx available for predicting the coordinates of the decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx. These two arrays are buffering some radius and azimuthal angle coordinate of sensed points for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx previously decoded in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no sensed point has already been decoded for a specific beam, the arrays are initialized to default values, as specified in subclause 11.6.2.

The arrays SensingBeamAttrBuffLastAttrValue, SensingBeamAttrBuffLastAttrRes, SensingBeamAttrBuffPrevLastAttrRes and SensingBeamAttrBuffLastRadius, four arrays of integer values; SensingBeamAttrBuffLastAttrValue[ HeadIdx ][ ElevationIdx ][ AttrBuffIdx ], SensingBeamAttrBuffLastAttrRes[ HeadIdx ][ ElevationIdx ][ AttrBuffIdx ], SensingBeamAttrBuffPrevLastAttrRes[ HeadIdx ][ ElevationIdx ][ AttrBuffIdx ], and SensingBeamAttrBuffLastRadius[ HeadIdx ][ ElevationIdx ][ AttrBuffIdx ] are buffer arrays for low latency attributes decoding which represent buffered values for a previously decoded attribute value, a last decoded attribute residual, a previous to the last decoded attribute residual, and an associated radius value for the buffer index AttrBuffIdx in the buffer available for decoding the low latency attributes value of a decoded sensed point for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx. These four arrays are buffering values for the sensing beam with elevation index ElevationIdx of the sensing head with index HeadIdx previously decoded in the current slice, or in a preceding slice provided by a previous GDU if the sensor coding state data is dependent on that previous GDU (per\_sensor\_coding\_state\_preserved is 1). If no sensed point has already been decoded for a specific beam, the arrays are initialized to default values, as specified in subclause 11.6.2.

The array SensingPointAzimuthIncFromLast[HeadIdx][PtnIdx] an array of integer value used during context selection for decoding pred\_idx and radius\_residual syntax elements, and representing an azimuthal index increment between the last decoded point and the previous last decoded point for the same sensing beam as the one of the point being decoded with index PtnIdx for the sensing head with index HeadIdx.

The array SensingPointPrevAzimuthIncFromLast[HeadIdx][PtnIdx] an array of integer value used during context selection for decoding radius\_residual syntax element, and representing an azimuthal index increment between the point being decoded with index PtnIdx and the last decoded point for the same sensing beam and for the sensing head with index HeadIdx.

### Initialization at the start of a GDU

At the start of any GDU, the sensor coding state data shall be initialized.

Any element of SensingHeadLastSensingRangeIdx shall be initialized to −1.

For (h = 0; h < num\_heads; h++)  
 SensingHeadLastSensingRangeIdx[h] = −1

In addition, subclause 11.6.2.1 applies when per\_sensor\_coding\_state\_preserved is 0, otherwise subclause 11.6.2.2 applies.

#### Initialization to default Sensor Coding State

When sensor coding state is not preserved from a previous GDU, the state variables shall be initialized as follows.

All the elements of the arrays SensingHeadStartedCapturing, SensingHeadLastAzimuthIdx, SensingHeadLastElevationIdx, BufferLate, SensingBeamLastAzimuthIdx, SensingBeamPrevLastAzimuthIdx, SensingBeamLastAzimuthStepsFromRadius, SensingBeamLastRadiusResSign, SensingBeamLastAzimuthResSign, SensingBeamLastZResSign, SensingBeamLastPredIdx, SensingBeamRadiusPred, SensingBeamAzimuthPred, SensingBeamAttrBuffPrevLastAttrRes, and SensingBeamAttrBuffLastRadius are initialized to 0. All the elements of the array SensingBeamAttrBuffLastAttrValue are initialized to 128.

for (h = 0; h < num\_heads; h++) {  
 SensingHeadStartedCapturing[h] = 0  
 SensingHeadLastAzimuthIdx[h] = 0  
 SensingHeadLastElevationIdx[h] = 0  
 for (packIdx = 0; packIdx < 8; packIdx++)  
 BufferLate[h][packIdx] = 0  
 for (elvIdx = 0; elvIdx ≤ num\_beams\_minus1[ h ]; elvIdx++) {  
 SensingBeamLastRadiusResSign[ h ][ elvIdx ] = 0  
 SensingBeamLastAzimuthResSign[ h ][ elvIdx ] = 0  
 SensingBeamLastZResSign[ h ][ elvIdx ] = 0  
 SensingBeamLastPredIdx[ h ][ elvIdx ] = 0  
 for (predIdx = 0; predIdx < 5; ++predIdx) {  
 SensingBeamRadiusPred[ h ][ elvIdx ][predIdx] = 0  
 SensingBeamAzimuthPred[ h ][ elvIdx ][predIdx] = 0  
 }  
 for (attrPredIdx = 0; attrPredIdx < 5; ++attrPredIdx) {  
 SensingBeamAttrBuffLastAttrValue[ h ][ elvIdx ][attrPredIdx] = 128  
 SensingBeamAttrBuffPrevLastAttrRes[ h ][ elvIdx ][attrPredIdx] = 0  
 SensingBeamAttrBuffLastRadius[ h ][ elvIdx ][attrPredIdx] = 0  
 }  
 }  
}

#### Initialization from previously preserved Sensor Coding State

When sensor coding state is preserved from a previous GDU, the state variables shall be initialized as follows.

All the elements of the arrays SensingHeadStartedCapturing, SensingHeadLastAzimuthIdx, SensingHeadLastElevationIdx, BufferLate, SensingBeamLastAzimuthIdx, SensingBeamPrevLastAzimuthIdx, SensingBeamLastAzimuthStepsFromRadius, SensingBeamLastRadiusResSign, SensingBeamLastAzimuthResSign, SensingBeamLastZResSign, SensingBeamLastPredIdx, SensingBeamRadiusPred, SensingBeamAzimuthPred, SensingBeamAttrBuffPrevLastAttrRes, SensingBeamAttrBuffLastRadius and SensingBeamAttrBuffLastAttrValue shall be restored from previously stored values.

When sensing\_coverage\_present is not 0 and when a sensing head with index h has started capturing points in a previous GDU, the last sensing range index for that head SensingHeadLastSensingRangeIdx[ h ] shall be updated according to the coarse position of the last sensed point decoded in previous GDU(s) as follows.

for (h = 0; h < num\_heads; h++)  
 if (sensing\_coverage\_present ≠ 0 && SensingHeadStartedCapturing[h] ≠ 0) {  
 SensingHeadLastSensingRangeIdx[ h ] = 0  
 for (b = 0; b ≤ num\_sensing\_ranges\_minus1[  h  ]; b++)  
 if (SensingHeadLastAzimuthIdx[h] ≥ SensingRangeFirst[ h ][ b]  
 && SensingHeadLastAzimuthIdx[h] ≤ SensingRangeLast[ h ][ b ]) {  
 SensingHeadLastSensingRangeIdx[ h ] = b  
 break  
 }  
 }  
}

### Initialization with decoded coarse position of first sensed point

After a coarse position (azimuth\_index\_start[ HeadIdx ] and elevation\_index\_start[ HeadIdx ]) of a first sensed point for a sensing head with index HeadIdx is decoded, the sensor coding state variable SensingHeadStartedCapturing[ HeadIdx ] shall be set to 1, and the sub-arrays of the sensor codding state SensingBeamLastAzimuthIdx[ HeadIdx ], SensingBeamPrevLastAzimuthIdx[ HeadIdx ], SensingHeadNeighOccupancyBits[ HeadIdx ], SensingBeamLastAzimuthStepsFromRadius [ HeadIdx ] shall be initialised as follows:

SensingHeadStartedCapturing[ HeadIdx ] = 1

for (b = 0; b ≤ num\_sensing\_ranges\_minus1[  HeadIdx]; b++) {  
 SensingBeamLastAzimuthIdx[ HeadIdx  ][ b ] = azimuth\_index\_start[ HeadIdx ] - 1  
 SensingBeamPrevLastAzimuthIdx[HeadIdxh ][ b ] = azimuth\_index\_start[ HeadIdx ] - 2  
 SensingHeadNeighOccupancyBits[HeadIdx][b] = 0  
 SensingBeamLastAzimuthStepsFromRadius[ HeadIdx ][ b] = 0  
}  
for (b = num\_sensing\_ranges\_minus1[  HeadIdx]; b ≤ 2; b++) {  
 SensingHeadNeighOccupancyBits[HeadIdx][b] = 0  
}  
SensingHeadNeighOccupancyBits[HeadIdx][elevation\_index\_start[ HeadIdx ]] = 1

The sensing range index of the last sensed point decoded for the sensing head with index HeadIdx shall be initialized with the index of the first sensing range containing the first sensed point:

SensingHeadLastSensingRangeIdx[ HeadIdx ] = -1  
for (b = 0; b ≤ num\_sensing\_ranges\_minus1[  h  ]; b++)  
 if (azimuth\_index\_start[ HeadIdx ] ≥ SensingRangeFirst[ HeadIdx][ b]  
 && azimuth\_index\_start[ HeadIdx ] ≤ SensingRangeLast[HeadIdx ][ b ]) {  
 SensingHeadLastSensingRangeIdx[HeadIdx ] = b  
 break  
 }

The subclause 11.6.5 shall then be applied to set and update other state variables.

### Update with azimuth\_offset

After any azimuth\_offset syntax element, the variable SensingHeadLastSensingRangeIdx[HeadIdx] of sensor coding state data shall be updated, and the variable AzimuthIdxOffset[AzimuthIdx][azimuth\_offset] representing the effective azimuth offset, expressed in number of coarse azimuth angle steps shall be set as follows:

AzimuthIdxOffset[AzimuthIdx][azimuth\_offset] = idxOffset  
SensingHeadLastSensingRangeIdx[HeadIdx] = lastRangeIdx  
 where  
 idxOffset = 0  
 lastRangeIdx = SensingHeadLastSensingRangeIdx[HeadIdx]  
 azimuthIdx = AzimuthIdx  
 for (k = 0; k < azimuth\_offset; k++) {  
 azimuthIdx += late\_point[HeadIdx][PtnIdx] ? -1 : 1  
 idxOffset += 1  
 if (late\_point[HeadIdx][PtnIdx]) {  
 UpdateAfterNegativeOffset()  
 } else {  
 UpdateAfterPositiveOffset()  
 }  
 }  
 azimuthIdx++  
 UpdateAfterPositiveOffset()  
 where  
 UpdateAfterNegativeOffset() := {  
 if (azimuthIdx < SensingRangeFirst[HeadIdx][lastRangeIdx]) {  
 lastRangeIdx--  
 if (lastRangeIdx < 0) {  
 lastRangeIdx = num\_sensing\_ranges\_minus1[  HeadIdx  ]  
 idxOffset += beam\_steps\_per\_rotation\_minus1[HeadIdx] + 1  
 }  
 idxOffset += azimuthIdx - SensingRangeLast[HeadIdx][lastRangeIdx]  
 azimuthIdx = SensingRangeLast[HeadIdx][lastRangeIdx]  
 }  
 }  
 UpdateAfterPositiveOffset() := {  
 if (azimuthIdx > SensingRangeLast[HeadIdx][lastRangeIdx]) {  
 lastRangeIdx++  
 if (lastRangeIdx > num\_sensing\_ranges\_minus1[  HeadIdx  ]) {  
 lastRangeIdx = 0  
 idxOffset += beam\_steps\_per\_rotation\_minus1[HeadIdx] + 1  
 }  
 idxOffset += SensingRangeFirst[HeadIdx][lastRangeIdx] - azimuthIdx  
 azimuthIdx = SensingRangeFirst[HeadIdx][lastRangeIdx]  
 }  
 }

Then, the array of neighbours’ occupancy shall be updated if needed.

if (geom\_use\_coarse\_neighbours && idxOffset)  
 for (b = 0; b ≤ num\_beams\_minus1[HeadIdx]; b++)  
 SensingHeadNeighOccupancyBits[HeadIdx][b] =  
 (!late\_point[HeadIdx][PtnIdx] && idxOffset < 3)  
 ? SensingHeadNeighOccupancyBits[HeadIdx][b] << idxOffset  
 : 0  
 where  
 idxOffset := AzimuthIdxOffset[AzimuthIdx][azimuth\_offset]

### Update while decoding elevation\_offset

When the coded geometry of a DU relies on coarse neighbours’ occupancy (geom\_use\_coarse\_neighbours is equal to 1), the neighbours’ occupancy information shall be updated after the decoding of any coded bin (with index BinIdxTu) of an elevation\_offset syntax element:

if (geom\_use\_coarse\_neighbours)  
 SensingHeadNeighOccupancyBits[ HeadIdx ][offsetElv ] <<= 1  
 where  
 offsetElv := ElevationIdx + BinIdxTu

When the coded geometry of a DU relies on coarse neighbours’ occupancy (geom\_use\_coarse\_neighbours is equal to 1), the neighbours’ occupancy information shall in addition be updated at the end of the decoding an elevation\_offset syntax element:

if (geom\_use\_coarse\_neighbours)  
 SensingHeadNeighOccupancyBits[ HeadIdx ][offsetElv ] |= 1  
 where  
 offsetElv := ElevationIdx + elevation\_offset

### Update with sensed point coarse position

After any decoded dup\_point[ HeadIdx ][ PtnIdx ] which is equal to 1 or before decoding any point\_position() , the sensor coding state data shall be updated based on the coarse position of the sensed point with index PtnIdx and for the head with index HeadIdx.

SensingHeadLastAzimuthIdx[ HeadIdx ] and SensingHeadLastElevationIdx[ HeadIdx ] shall be set respectively to the decoded sensing azimuthal angle index SensingAzimuthIdx[HeadIdx][PtnIdx] and to the decoded sensing elevation angle index SensingElevationIdx[HeadIdx][PtnIdx]:

SensingHeadLastAzimuthIdx[HeadIdx] = SensingAzimuthIdx[HeadIdx][PtnIdx]

SensingHeadLastElevationIdx[HeadIdx] = SensingElevationIdx[HeadIdx][PtnIdx]

The variables SensingPointPrevAzimuthIncFromLast[HeadIdx][PtnIdx], SensingPointAzimuthIncFromLast[HeadIdx][PtnIdx], SensingBeamPrevLastAzimuthIdx[ HeadIdx ][ ElevationIdx ], and SensingBeamLastAzimuthIdx[ HeadIdx ][ ElevationIdx ], shall also be set accordingly to the decoded coarse position:

SensingPointPrevAzimuthIncFromLast[HeadIdx][PtnIdx] =  
 SensingBeamLastAzimuthIdx[ HeadIdx ][ elevationIdx ]  
 – SensingBeamPrevLastAzimuthIdx[ HeadIdx ][ elevationIdx ]  
SensingPointAzimuthIncFromLast[HeadIdx][PtnIdx] =  
 azimuthIdx - SensingBeamLastAzimuthIdx[ HeadIdx ][ elevationIdx ]  
SensingBeamPrevLastAzimuthIdx[ HeadIdx ][ elevationIdx ] =  
 SensingBeamLastAzimuthIdx[ HeadIdx ][ elevationIdx ]  
SensingBeamLastAzimuthIdx[ HeadIdx ][ elevationIdx ] = azimuthIdx  
 where  
 azimuthIdx := SensingAzimuthIdx[HeadIdx][PtnIdx]  
 elevationIdx := SensingElevationIdx[HeadIdx][PtnIdx]

## Parsing state memorization and restoration

### General

Subclause 11.6 applies when entropy\_continuation\_enabled is 1.

At certain moments, the entropy parsing state is recorded and later, used as the initial state for parsing other DUs or entropy streams.

The entropy parsing state shall comprise:

for a GDU, the CABAC CPMs (11.5.3) the OBUF ACPMs(12.2.2) ;

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) for each different value of GDU HeadIdx and for each different value of ADU AttrIdx. For example, a coded point cloud sequence with num\_attributes equal to 2 and with num\_heads equal to 3 would require storage for five sets of entropy parsing state. When geom\_low\_latency\_attributes equals to 1, the entropy parsing state for a particular value of GDU HeadIdx is recorded for both geometry entropy stream and low latency attributes entropy stream.

At the start of any GDU with slice\_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 every end\_of\_entropy\_stream syntax element (9.2.3.1), for the corresponding value of HeadIdx.

Memorization shall record the elements and values of the GDU entropy parsing state for restoration by the restoration process (11.6.2.2). The state shall be recorded separately for each value of HeadIdx.

#### 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.

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) with the same value of HeadIdx.

### 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 slice\_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. The initialization shall be according to 11.5.3.2 as if the data unit contained a syntax element with the descriptor ae(v).

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 so that ADUs in any following slices where slice\_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 (12.4),
* 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 two elements:

* an OBUF tree constituted of three arrays down [ ], nVisit[ ] and ctxIdxMap[ ][ ],
* and an array obufCtxArray[ ] of OBUF ACPMs.

## Creation of an OBUF instance

An OBUF instance is a set made of the two above elements, namely an OBUF tree, and an array of OBUF ACPMs. The OBUF tree and the array of OBUF ACPMs are uniquely attached to the OBUF instance.

Creating an OBUF instance follows the steps of

* 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.

The OBUF instance is created based on the given of

* a first size *s1* corresponding to the size of first contextual information *info1* used as input when calling the OBUF instance,
* a second size *s2* 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 *s1*, *s2*, *minKDown* 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

OBUF instances are created and initialized before coding all coarse positions.

The sizes *s1* of the first contextual information is determined by

s1 = 2\*64

The maximum index of laser pack *kMaxBitsLaserPackIndex* is set as 5, and the size *obufTreeSize2* of the *s1* OBUF trees are determined by

obufTreeSize2 = 1<< kMaxBitsLaserPackIndex

The size *numBitsLaserIndex* is defined as the bit number used for representing total number of laser beams in a Lidar device.

The size *s2* of the secondary contextual information is determined by

s2 = std::max(numBitsLaserIndex, kMaxBitsLaserPackIndex)

The value *kDown0* is used to initialize array down [], and it is determined by

kDown0= std::max(numBitsLaserIndex, kMaxBitsLaserPackIndex)

The size *minKDown* is used to update OBUF instance in 12.4, and it is determined by

minKDown = std::max(0, numBitsLaserIndex - kMaxBitsLaserPackIndex)

The value *maxScaledSeen* is used to determine threshold for the number of visits of nodes to update OBUF instance, and *maxScaledSeen* is determined by

maxScaledSeen =2\*(1<< kMaxBitsLaserPackIndex)

Two double-entry 8-bit arrays down [ ], nVisit [ ]are created with size *s1*.

One double-entry 8-bit array ctxIdxMap[ ][ ] is created with size *s1* along the first entry and size *obufTreeSize2* along the second entry for a total size of *s1* × *obufTreeSize2*.

The array down [0 ] is initialized with all *s1* values set to *kDown0*. The array nVisit [0 ] is initialized with all of the *s1* values set to 0. The array ctxIdxMap [ 0][ 0] is initialized with all of the *s1* values set to 127.

The array down [ ] corresponds to the number of erased bits of a secondary information *info2* when the instance is called. The initialising to *kDown0* 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 4) 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 16 of ACPMs as defined in 11.5.3,

The array obufCtxArray[ ] is initialized to be 0.5 for each element.

## Call and update of an OBUF instance

An OBUF instance is called with input a first contextual information *info1*,and a second contextual information *info2*. The output is a decoded bin *bin*.

The number *nErasedBit* of bits to be erased from the second contextual information is obtained by

nErasedBit = down[ info1]

Depending on the value of *nErasedBit*, the decoding and update process is performed according to number of visits of nodes of the OBUF trees. In case *nErasedBit* > *minKDown*, the process continues to clause 12.3.1; otherwise the process terminates the call of the OBUF instance.

### 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

info2Erased = info2 >> nErasedBit

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 12.4.

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 1.

ctxIdxMap[info1 ][ info2Erased] += ObufCtxIdxDelta[ctxIdx >> 4][bin]

**Table 1— Values of *ObufCtxIdxDelta* [ k ][m]**

| k | m |  |
| --- | --- | --- |
| 0 | 1 |
| **0** | 0 | 6 |
| **1** | -1 | 13 |
| **2** | -1 | 18 |
| **3** | -2 | 22 |
| **4** | -4 | 23 |
| **5** | -7 | 19 |
| **6** | -9 | 16 |
| **7** | -11 | 14 |
| **8** | -14 | 11 |
| **9** | -16 | 9 |
| **10** | -19 | 7 |
| **11** | -23 | 4 |
| **12** | -22 | 2 |
| **13** | -18 | 1 |
| **14** | -13 | 1 |
| **15** | -6 | 0 |

The number of visits *nVisit* is incremented by one unit

nVisit[info1 ]++

and a threshold *thVisit* on the number of visits is obtained by using *kDown0* and *maxScaledSeen*

thVisit = (maxScaledSeen >> kDown0) + 1

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 12.3.1.2 if *nErasedBit* > *minKDown* or terminates the call of the OBUF instance otherwise.

#### OBUF tree update

The node of OBUF trees referenced by *info1* is updated with no visit.

nVisit[ info1] = 0

The node of OBUF trees referenced by *info1* is split into new nodes for context index ctxIdxMap[ *info1*][].

A value *numUsed* is used to update the context index ctxIdxMap, and the *numUsed* is determined by

numUsed =(1<<numBitsLaserIndex)>> down[info ]

The context index ctxIdxMap[ *info1*][] of the new nodes is obtained by an iteration based on the determined value *numUsed*.

for (int i = numUsed-1; i >= 0; --i) {

ctxIdxMap[(j << 1) + k][(i << 1) + 1] = ctxIdxMap[(j << 1) + k][i]

ctxIdxMap[(j << 1) + k][(i << 1) + 0] = ctxIdxMap[(j << 1) + k][i]

}

The number of erased bits is decreased by one unit for down[*info1* ].

down[info1] --

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 >> 4  
SelCtx = obufCtxArray[idx ]

A *bin* of information is decoded according to the clause 11.5.4.5 by using the selected context *SelCtx*. Then, the probability evolves according to clause 11.5.3.3.

1. (normative)  
   Profiles and levels
   1. Overview of profiles and levels

Profiles and levels specify restrictions on bitstreams and hence limits on the capabilities needed to decode the bitstreams. Profiles and levels may also be used to indicate interoperability points between individual implementations.

* 1. This document does not include individually selectable options at the decoder, as this would increase interoperability difficulties.

Each profile specifies a subset of algorithmic features and limits that shall be supported by all decoders conforming to that profile.

* 1. Encoders are not required to use any particular subset of features supported by a profile.

Each level specifies a set of limits on the values that may be coded by L3C2 syntax elements. Level definitions apply to all profiles. For any given profile, a level generally corresponds to a particular decoder processing load and memory capability.

* 1. Requirements on decoder capability

The capabilities of decoders conforming to this document are specified in terms of the ability to decode bitstreams conforming to the constraints of profiles and levels specified in this annex.

When expressing the capabilities of a decoder for a specific profile, the level supported for that profile should also be expressed.

* 1. Profiles
     1. General

All constraints for SPSs, GPSs and APSs that are specified are constraints for the parameter sets that are active during bitstream decoding.

* + 1. Simple and Main profiles

Bitstreams conforming to the Simple or Main profiles shall satisfy:

the constraints specified in Table A.1;

the level constraints specified in A.4.

Conformance of a bitstream to a particular profile shall be indicated by:

simple\_profile\_compliant equal to 1 for bitstreams conforming to the Simple profile;

main\_profile\_compliant equal to 1 for bitstreams conforming to the Main profile.

Decoders conforming to a particular profile at a specific level shall be capable of decoding all bitstreams that:

indicate conformance with the profile, and

indicate conformance with a level that is lower than or equal to the decoder's level.

Table A.1 — Allowed values of syntax elements according to profile

| Parameter set | Syntax element | Profile | |
| --- | --- | --- | --- |
|  |  | Simple | Main |
| SPS | main\_profile\_compliant | 0 | 1 |
| bypass\_stream\_enabled | 1 | 0 or 1 |
| GPS | attr\_coding\_type | 0 .. 3 | 0 .. 3 |
| lod\_max\_levels\_minus1 | ≤ 20 | ≤ 20 |
| APS | lod\_decimation\_mode | 1 or 2 | 0 .. 2 |
| lod\_scalability\_enabled | 0 | 0 or 1 |
| raht\_prediction\_enabled | 0 | 0 or 1 |
| attr\_coord\_conv\_enabled | 0 | 0 or 1 |

* 1. Levels
     1. Level limits

The level to which a bitstream conforms is indicated by a value of level\_idc 20 times the level number specified in Table A.2. All other values of level\_idc are reserved for future use by ISO/IEC.

When comparing level capabilities, a particular level 𝑎 shall be considered to be a lower level than another level 𝑏 when the value of level\_idc for 𝑎 is less than that for 𝑏.

Table A.2 specifies limits for each level:

MaxSlicePoints specifies the maximum number of points that can be coded by a slice.

MaxSliceDimLog2 specifies the maximum number of bits that represent a point coordinate in a slice.

MaxSeqBboxDimLog2 specifies the maximum number of bits that represent a point coordinate in the sequence coordinate system.

Table A.2 — Level limits

| Level | MaxSlicePoints | MaxSliceDimLog2 | MaxSeqBboxDimLog2 |
| --- | --- | --- | --- |
| 4 |  | 21 | 21 |
| 6 |  | 21 | 32 |

* 1. Permitted ranges for syntax elements

Tables A.3 to A.11 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.3 — Permitted ranges for sequence parameter set syntax elements

| Syntax element | Range |
| --- | --- |
| reserved\_profile\_21bits | 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 |

Table A.4 — 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.5 — 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.6 — Permitted ranges for geometry parameter set syntax elements

| Syntax element | Range |
| --- | --- |
| gps\_seq\_parameter\_set\_id | 0 |
| gps\_geom\_origin\_log2\_scale | 0 .. 31 |
| gps\_angular\_origin\_bits\_minus1 | 0 .. 31 |
| num\_beams\_minus1 | 0 .. 254 |
| beam\_elevation\_init | ± |
| beam\_voffset\_init | ± |
| beam\_elevation\_diff[ 𝑖 ] | ± |
| beam\_voffset\_diff[ 𝑖 ] | ± |
| gps\_extension\_present | 0 |

Table A.7 — 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 |

Table A.8 — 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.9 — Permitted ranges for geometry data unit syntax elements

| Syntax element | Range |
| --- | --- |
| gdu\_reserved\_zero\_3bits | 0 |
| 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 |
| stream\_cnt\_minus1 | (1 + geom\_low\_latency\_attributes) × num\_heads − 1 |

Table A.10 — Permitted ranges for Point low latency attributes syntax elements

| Syntax element | Range |
| --- | --- |
| attr\_pred | 0..1 |
| attr\_res | 0 .. |
| attr\_val | 0 .. |

Table A.11 — Permitted ranges for attribute data unit syntax elements

| Syntax element | Range |
| --- | --- |
| adu\_reserved\_zero\_3bits | 0 |
| 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 |

Table A.12 — 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 |

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 |

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.

* 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()

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

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

BeamElev, 7.4.2.5.2

BeamOffsetV, 7.4.2.5.2

BiasedNorm1, 10.6.6.5

BiasedNorm2, 10.6.6.11

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

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

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

CtxTbl, 11.5.3.4

CtxUpdateDelta, 11.5.3.3

DataUnitBytes, 11.2.2

DataUnitLength, 11.2.2

DataUnitReadIdx, 11.2.2

DuFooterLen, 11.2.4

DuIsGdu, 11.2.4

DuNextBit, 11.2.5

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

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

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

MapSub, 10.6.5.6

MaxSeqBboxDimLog2, A.4.1

MaxSliceDimLog2, A.4.1

MaxSlicePoints, A.4.1

MinInterRefLvl, 10.6.6.2

MinNodeSizeLog2, D.2.1

Morton, 5.10.7

NextAeStreamBit, 11.5.4.4

Norm2, 10.6.6.12

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

PartVal, 11.4.1

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

PredPtIdx, 10.6.4

PredWeight, 10.6.4

PtnAng, 9.2.4.3

PtnAngStv, 9.2.4.5

PtnCnt, 9.2.3.1

PtnDepth, 9.2.3.1

PtnIdx, 9.2.3.1

PtnPos, 9.2.4.2

PtnPred, 9.2.4.6

PtnRef, 9.2.4.6

PtnStack, 9.2.3.1

RahtBlkCnt, 10.5.2.2

RahtBlkLoc, 10.5.2.2

RahtBlkWeight, 10.5.2.3

RahtCoeff, 10.5.2.2

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

RahtDcCoeff, 10.5.6.2

RahtDcNorm, 10.5.5.4.1

RahtFwd, 10.5.5.7

RahtInv, 10.5.6.5

RahtLvlCnt, 10.5.2.2

RahtNeighCnt, 10.5.5.3

RahtNeighCntMinAncestor, 10.5.5.3

RahtPred, 10.5.5.4.3

RahtPredBlk, 10.5.5.1

RahtPredEligible, 10.5.5.2

RahtPredExcluded, 10.5.5.4.2

RahtPredRecipW, 10.5.5.4.3

RahtPredW, 10.5.5.5

RahtPredWeight, 10.5.5.4.3

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

SensingRangeFirst, 7.4.3.2

SensingRangeLast, 7.4.3.2

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

SumN19, 10.5.5.3

TileInventoryOrigin, 7.4.2.4

ZeroRunLength, 10.3.1.1