**ISO/IEC 23090-9:2020(E)**

ISO/IEC JTC 1/SC 29/WG 11

Secretariat: JISC

**Information technology — MPEG-I (Coded Representation of Immersive Media) — Part 9: Geometry-based Point Cloud Compression**

DIS 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.

*To help you, this guide on writing standards was produced by the ISO/TMB and is available at <https://www.iso.org/iso/how-to-write-standards.pdf>*

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

© ISO/IEC 2020

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 5](#_Toc38236434)

[Introduction 6](#_Toc38236435)

[1 Scope 1](#_Toc38236436)

[2 Normative references 1](#_Toc38236437)

[3 Terms and definitions 1](#_Toc38236438)

[3.1 General 1](#_Toc38236439)

[3.2 Geometry coding related 3](#_Toc38236440)

[3.3 Attribute coding related 4](#_Toc38236441)

[4 Abbreviations 4](#_Toc38236442)

[5 Conventions 5](#_Toc38236443)

[5.1 General 5](#_Toc38236444)

[5.2 Numerical representation 5](#_Toc38236445)

[5.3 Arithmetic operators 5](#_Toc38236446)

[5.4 Logical operators 6](#_Toc38236447)

[5.5 Relational operators 6](#_Toc38236448)

[5.6 Bit-wise operators 6](#_Toc38236449)

[5.7 Assignment operators 6](#_Toc38236450)

[5.8 Range notation 7](#_Toc38236451)

[5.9 Mathematical functions 7](#_Toc38236452)

[5.9.1 Definition of iAtan2 7](#_Toc38236453)

[5.9.2 Definition of popCnt 8](#_Toc38236454)

[5.9.3 Definition of iLog2 8](#_Toc38236455)

[5.9.4 Definition of iSqrt 8](#_Toc38236456)

[5.9.5 Definition of inverse square root function invSqrt 9](#_Toc38236457)

[5.9.6 Definition of divExp2RoundHalfInf 11](#_Toc38236458)

[5.9.7 Definition of divExp2RoundHalfUp 11](#_Toc38236459)

[5.9.8 Conversion of a tuple to 3D Morton code (TupleToMorton) 11](#_Toc38236460)

[5.9.9 Conversion of 3D Morton codes to a tuple (MortonToTuple) 12](#_Toc38236473)

[5.9.10 Definition of QpToQstep 12](#_Toc38236474)

[5.10 Vector operations 12](#_Toc38236475)

[5.11 Order of operation precedence 13](#_Toc38236476)

[5.12 Variables, syntax elements, and tables 13](#_Toc38236477)

[5.13 Text description of logical operations 15](#_Toc38236478)

[5.14 Processes 16](#_Toc38236479)

[6 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships 16](#_Toc38236480)

[6.1 Bitstream formats 16](#_Toc38236481)

[6.2 Source, decoded, and output point cloud formats 16](#_Toc38236482)

[6.2.1 Data partitioning 17](#_Toc38236483)

[6.2.2 Frame index attribute component 17](#_Toc38236484)

[6.3 Geometry octree 17](#_Toc38236485)

[6.4 Neighbour relationships 17](#_Toc38236486)

[6.4.1 Neighbour dependent geometry octree child node scan order inverse mapping process 17](#_Toc38236487)

[6.4.2 Neighbour depending geometry occupancy map permutation process 18](#_Toc38236488)

[7 Syntax and semantics 19](#_Toc38236489)

[7.1 Method of specifying syntax in tabular form 19](#_Toc38236490)

[7.2 Specification of syntax functions and descriptors 20](#_Toc38236491)

[7.3 Syntax in tabular form 21](#_Toc38236492)

[7.3.1 General 21](#_Toc38236493)

[7.3.2 Data unit and byte alignment syntax 21](#_Toc38236494)

[7.3.3 Geometry data unit syntax 25](#_Toc38236495)

[7.3.4 Attribute data unit syntax 29](#_Toc38236496)

[7.4 Semantics 31](#_Toc38236497)

[7.4.1 General 31](#_Toc38236498)

[7.4.2 Data unit and byte alignment semantics 32](#_Toc38236499)

[7.4.3 Geometry data unit semantics 40](#_Toc38236500)

[7.4.4 Attribute data unit semantics 49](#_Toc38236501)

[8 Decoding process 51](#_Toc38236502)

[8.1 General decoding process 51](#_Toc38236503)

[8.2 Geometry decoding process 51](#_Toc38236504)

[8.2.1 General geometry decoding process 51](#_Toc38236505)

[8.2.2 Octree decoding process 51](#_Toc38236506)

[8.2.3 Geometry Trisoup decoding process 53](#_Toc38236507)

[8.2.4 Planar coding mode 62](#_Toc38236508)

[8.2.5 Angular coding mode 66](#_Toc38236509)

[8.3 Attribute decoding 68](#_Toc38236510)

[8.3.1 Region adaptive hierachical transform decoding process 68](#_Toc38236511)

[8.3.2 LoD with Lifting Transform decoding process 77](#_Toc38236512)

[8.3.3 LoD with Predicting Transform decoding process 87](#_Toc38236513)

[8.4 Slice concatenation process 88](#_Toc38236514)

[9 Parsing process 89](#_Toc38236515)

[9.1 General 89](#_Toc38236516)

[9.2 Chunked bytestream parsing process 91](#_Toc38236517)

[9.2.1 General 91](#_Toc38236518)

[9.2.2 Syntax 92](#_Toc38236519)

[9.2.3 Semantics 92](#_Toc38236520)

[9.3 Definition of readDataUnitBit 93](#_Toc38236521)

[9.4 Definition of readAeStreamBit 93](#_Toc38236522)

[9.5 Definition of readBypassStreamBit 93](#_Toc38236523)

[9.6 General inverse binarisation processes 94](#_Toc38236524)

[9.6.1 Parsing of fixed-length codes 94](#_Toc38236525)

[9.6.2 Parsing of k-th order exp-Golomb codes 94](#_Toc38236526)

[9.6.3 Parsing of truncated unary codes 94](#_Toc38236527)

[9.6.4 Mapping process for signed codes 94](#_Toc38236528)

[9.7 Bit-wise geometry octree occupancy parsing process 95](#_Toc38236529)

[9.7.1 General process 95](#_Toc38236530)

[9.7.2 Initialisation process 95](#_Toc38236531)

[9.7.3 Determination of planar masks used in the inverse binarization process 95](#_Toc38236532)

[9.7.4 Occupancy\_idx[] parsing process 96](#_Toc38236533)

[9.7.5 Inverse binarization process 96](#_Toc38236534)

[9.7.6 Definition of readOccBin() 98](#_Toc38236535)

[9.7.7 ctxMapIdx and ctxIdx derivation processes 98](#_Toc38236536)

[9.7.8 Context map update process 100](#_Toc38236537)

[9.7.9 Occupancy prediction process using neighbouring octree nodes 101](#_Toc38236538)

[9.8 Inferred Direct Coding Mode parsing process 103](#_Toc38236539)

[9.8.1 General process 103](#_Toc38236540)

[9.8.2 Determination of the angular context idcmIdxAngular 103](#_Toc38236541)

[9.8.3 Inverse binarization process 104](#_Toc38236542)

[9.9 Dictionary-based parsing 104](#_Toc38236543)

[9.9.1 General process 104](#_Toc38236544)

[9.9.2 Initializing lut0 106](#_Toc38236545)

[9.9.3 Initializing lut1 106](#_Toc38236546)

[9.9.4 Definition of decodeLut0Index() 106](#_Toc38236547)

[9.9.5 Definition of pushLut0() 107](#_Toc38236548)

[9.9.6 Definition of updateLut0() 107](#_Toc38236549)

[9.9.7 Definition of lut0ComputeMostFrequentSymbols() 107](#_Toc38236550)

[9.9.8 Definition of pushLut1() 107](#_Toc38236551)

[9.10 CABAC parsing process 108](#_Toc38236552)

[9.10.1 General 108](#_Toc38236553)

[9.10.2 Definition of readBin() 108](#_Toc38236554)

[9.10.3 Context variables 110](#_Toc38236555)

[9.10.4 Arithmetic decoding engine 111](#_Toc38236556)

[9.10.5 Arithmetic encoding engine (informative) 113](#_Toc38236557)

[9.11 Parsing state memorization process 115](#_Toc38236558)

[9.12 Parsing state restoration process 115](#_Toc38236559)

[Annex A Profiles and levels 116](#_Toc38236560)

[A.1 Overview of profiles and levels 116](#_Toc38236561)

[A.2 Requirements on decoder capability 116](#_Toc38236562)

[A.3 Profiles 116](#_Toc38236563)

[A.3.1 General 116](#_Toc38236564)

[A.3.2 Main profile 116](#_Toc38236565)

[A.4 Levels 117](#_Toc38236566)

[A.4.1 Level limits 117](#_Toc38236567)

[Annex B Type-length-value bytestream format 118](#_Toc38236568)

Foreword

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

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

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

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

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

This document was prepared by 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

ISO/IEC 23090-9 specifies Geometry-based Point Cloud Compression (G-PCC).

Advance in 3D capturing and rendering technologies is enabling new applications and services in the field of assisted and autonomous driving, maps, cultural heritage, industrial processes, immersive real-time communication, and Virtual/Augmented/Mixed reality (VR/AR/MR) content creation, transmission and communication. Point clouds have arisen as one of the main representations for such applications. A point cloud frame consists of a set of 3D points. Each point, in addition to having a 3D position may also be associated with numerous other attributes such as colour, transparency, reflectance, timestamp, surface normal, and classification. Such representations require a large amount of data, which can be costly in terms of storage and transmission. Therefore, the ISO/IEC Moving Picture Experts Group (MPEG) developed a new International Standard, which aims at efficiently compressing point cloud representations.

**Information technology — MPEG-I (Coded Representation of Immersive Media) — Part 9: Geometry-based Point Cloud Compression**

# Scope

This document specifies geometry-based point cloud compression.

# Normative references

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

ISO/IEC 8825-1 (ITU-T X.690)

ISO/IEC 9834-1 (ITU-T X.660)

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

3.1.1

point

position specified by their *Cartesian co-ordinates* (x, y, z) and associated with zero or more sets of attributes

3.1.2

point cloud frame

set of points at a particular time instance

3.1.3

point cloud

sequence of point cloud frames

3.1.4

Cartesian co-ordinates

three scalars (x, y, z) with finite precision and dynamic range that indicate the location of a point relative to a fixed reference point

3.1.5

geometry

set of Cartesian co-ordinates associated with a point cloud frame

3.1.6

attribute

scalar or vector property optionally associated with each point in a point cloud such as colour, reflectance, frame index, etc.

3.1.7

may

term that is used to refer to behaviour that is allowed, but not necessarily required.

Note 1 to entry: In some places where the optional nature of the described behaviour is intended to be emphasized, the phrase "may or may not" is used to provide emphasis.

3.1.8

must

term that is used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Specification (used exclusively in an informative context)

3.1.9

shall

term used to express mandatory requirements for conformance to this Specification.

3.1.10

should

a term used to refer to behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this Specification.

3.1.11

informative

term used to refer to content provided in this Specification that does not establish any mandatory requirements for conformance to this Specification and thus is not considered an integral part of this Specification

3.1.12

byte

sequence of 8 bits, written and read with the most significant bit on the left and the least significant bit on the right. When represented in a sequence of data bits, the most significant bit of a byte is first.

3.1.13

byte-aligned

position in a bitstream is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the bitstream, and a bit or byte or syntax element is said to be byte-aligned when the position at which it appears in a bitstream is byte-aligned.

3.1.14

unspecified

term unspecified, when used in the clauses specifying some values of a particular *syntax element*, indicates that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.

3.1.15

syntax element

element of data represented in the *bitstream*.

3.1.16

bitstream

a sequence of bits that forms the representation of coded *point cloud frames*

3.1.17

coded point cloud frame

a coded representation of a point cloud frame

3.1.18

syntax structure

zero or more syntax elements present together in the bitstream in a specified order

3.1.19

bounding box

rectangular cuboid in which the source point cloud frame is included.

3.1.20

3D tile

rectangular cuboid inside a bounding box.

3.1.21

slice

series of *syntax element* representing a part of or entire *coded point cloud frame*

## Geometry coding related

3.2.1

position

(x, y, z) co-ordinates of a point, wherethe values are normalized by the bounding box so that the values of the positions shall be equal to or greater than 0.

3.2.2

octree

8-ary tree representing the 3D geometry of the point cloud.

3.2.3

node

element of the octree representing a sub-volume of the 3D space (or volume) containing the point cloud.

3.2.4

root node

node of the octree with no parent

3.2.5

leaf node

terminating node of the octree having no children

3.2.7

level

number of hops from the root to the node.

3.2.8

occupied node

node for which one or more points belong to the associated sub-volume.

3.2.9

occupancy code

byte for a node whose bits indicate which child nodes are occupied.

3.2.10

Morton code

non-negative 3d-bit integer obtained by interleaving the bits of the non-negative d-bit integers s, t, and v.

## Attribute coding related

3.3.1

Colour

Three dimensional signal representing the characteristics of the light of the assoicated point (e.g. RGB, YUV)

Note 1 The colour is, for example, signalled by Red, Green and Blue components (RGB) or Luma and two Chroma components (YUV).

3.3.2

Reflectance

One dimensional signal representing the ratio of the intensity of the light reflection of the assosiated point

3.3.3

Frame index

One dimensional signal representing the timing information of the assosiated point as the frame order index

3.3.4

Material ID

One dimensional signal representing the material type information of the associated point

Note 1 For example, the material type could be used as an indicator for identifying an object or the characteristic of the associated point. The interpretation of the values is outside the scope of this document.

3.3.5

Transparency

One dimensional signal representing the condition of being transparent of the associated point

3.3.7

Normals

Three-dimensional signal representing the unit vector of the perpendicular direction to the surface of the associated point

Note 1 The order of the three components (i.e. the co-ordinate system) shall be identical to the one in the source point cloud frame.

# Abbreviations

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

APS Attribute Parameter Set

ASH Attribute Slice Header

GSH Geometry Slice Header

GPS Geometry Parameter Set

LSB Least Significant Bit

MSB Most Significant Bit

PCC Point Cloud Compression

RAHT Region Adaptive Hierarchical Transform

SPS Sequence Parameter Set

# Conventions

## General

NOTE – The mathematical operators used in this Specification 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.

## Numerical representation

The following numerical representation format are defined.

binary representation formatted as 0bXXX... where each digit X is 0 or 1

octal representation formatted as 0oXXX... where each digit X is 0 to 7

decimal representation formatted as XXX... where each digit X is 0 to 9

hexadecimal representation formatted as 0xXXX... where each digit X is 0 to 9 or a to f

## Arithmetic operators

The following arithmetic operators are defined as follows:

|  |  |
| --- | --- |
| + | Addition |
| − | Subtraction (as a two-argument operator) or negation (as a unary prefix operator) |
| × | Multiplication, including matrix multiplication |
| xy | 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. |
| ÷ | Used to denote division in mathematical equations where no truncation or rounding is intended. |
|  | Used to denote division in mathematical equations where no truncation or rounding is intended. |
|  | The 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

The following logical operators are defined as follows:

x && y Boolean logical "and" of x and y

x | | y Boolean logical "or" of x and y

! Boolean logical "not"

x ? y : z If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

## Relational operators

The following relational operators are defined as follows:

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

= = Equal to

!= Not equal to

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

## Bit-wise operators

The following bit-wise operators are defined as follows:

& Bit-wise "and". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

| Bit-wise "or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

^ Bit-wise "exclusive or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

x >> y Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y. Bits shifted into the most significant bits (MSBs) as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

x << y Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y. Bits shifted into the least significant bits (LSBs) as a result of the left shift have a value equal to 0.

## Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator

++ 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.

+= 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).

## Range notation

The following notation is used to specify a range of values:

x = y .. z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers and z being greater than y.

## Mathematical functions

The following mathematical functions are defined:

Abs( x ) =

Ceil( x ) the smallest integer greater than or equal to x.

Clip1Y( x ) = Clip3( 0, ( 1 << BitDepthY ) − 1, x )

Clip1C( x ) = Clip3( 0, ( 1 << BitDepthC ) − 1, x )

Clip3( x, y, z ) =

Floor( x ) the largest integer less than or equal to x.

Min( x, y ) =

Max( x, y ) =

Sign( x ) =

Sqrt( x ) =

Swap( x, y ) = ( y, x )

### Definition of iAtan2

The inputs to this process are the variables a and b.

The output of this process is the variable t.

The derivation process for is defined as follows.

If a is equal to 0 and b is equal to 0, t is set to 0.

Otherwise, if b is equal to 0, t is set to 804.

Otherwise, if a is equal to 0 and b is greater than 0, t is set to 402.

Otherwise, if a is equal to 0 and b is smaller than 0, t is set to 1206.

Otherwise, following steps apply:

c = Abs((b << 8) / a)

if (c <= 256)

idx = c / 12

else

idx = c > 40 ? 40 : c

t = atanLut[idx]

if (a < 0 && b > 0)

t += 402

else if (a < 0 && b < 0)

t += 804

else if (a > 0 && b < 0)

t += 1206

The array atanLut is defined in Table 1.

Table 1 — the value of atanLut[ i + j ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **j** | **i** | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **0** | 0 | 12 | 25 | 38 | 50 | 62 | 74 | 86 | 97 | 108 | 118 | 128 | 138 | 147 | 156 |
| **15** | 164 | 172 | 180 | 187 | 194 | 201 | 283 | 319 | 339 | 351 | 359 | 365 | 370 | 373 | 376 |
| **30** | 378 | 380 | 382 | 383 | 385 | 386 | 387 | 387 | 388 | 389 | 389 |

### Definition of high-precision iAtan2Hp

This process defines the function theta = iAtan2hp( a, b ).

The inputs to this process are the variables a and b.

The output of this process is the variable theta.

The variables dx and dy are derived as follows:

dy = Abs(b) <= Abs(a) ? Abs(b) : Abs(a)

dx = Abs(b) <= Abs(a) ? Abs(a) : Abs(b)

The angle theta is derived as follows:

if (dx == 0)

phi = 0

else {

rinv = iSqrt(dx\*dx + dy\*dy);

r = (dy \* rinv) >> 20

idx = r >> 11

lambda = r - (idx << 11)

theta = Asin[idx] + (lambda \* (Asin[idx + 1] - Asin[idx]) >> 11)

}

Where the array aSin is defined by Table 2.

Finally, the variable theta is updated as follows:

if (Abs(b) > Abs(a))

theta = 1647099 - theta

if (a < 0)

theta = 3294199 - theta

if (b < 0)

theta = -theta

Table 2 — the value of aSin[ i + j ]

| **j** | **i** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0** | 0 | 2048 | 4096 | 6144 | 8192 | 10240 | 12288 | 14336 |
| **8** | 16385 | 18433 | 20481 | 22530 | 24578 | 26627 | 28676 | 30724 |
| **16** | 32773 | 34822 | 36872 | 38921 | 40970 | 43020 | 45070 | 47120 |
| **24** | 49170 | 51220 | 53271 | 55322 | 57373 | 59424 | 61475 | 63527 |
| **32** | 65579 | 67631 | 69683 | 71736 | 73789 | 75842 | 77896 | 79949 |
| **40** | 82004 | 84058 | 86113 | 88168 | 90223 | 92279 | 94335 | 96392 |
| **48** | 98449 | 100506 | 102563 | 104621 | 106680 | 108739 | 110798 | 112858 |
| **56** | 114918 | 116978 | 119040 | 121101 | 123163 | 125225 | 127288 | 129352 |
| **64** | 131416 | 133480 | 135545 | 137611 | 139677 | 141743 | 143810 | 145878 |
| **72** | 147946 | 150015 | 152085 | 154155 | 156225 | 158297 | 160368 | 162441 |
| **80** | 164514 | 166588 | 168662 | 170737 | 172813 | 174890 | 176967 | 179045 |
| **88** | 181123 | 183203 | 185283 | 187363 | 189445 | 191527 | 193610 | 195694 |
| **96** | 197779 | 199864 | 201950 | 204037 | 206125 | 208214 | 210303 | 212393 |
| **104** | 214485 | 216577 | 218669 | 220763 | 222858 | 224954 | 227050 | 229148 |
| **112** | 231246 | 233345 | 235445 | 237547 | 239649 | 241752 | 243856 | 245961 |
| **120** | 248068 | 250175 | 252283 | 254392 | 256502 | 258614 | 260726 | 262840 |
| **128** | 264954 | 267070 | 269187 | 271305 | 273424 | 275544 | 277666 | 279788 |
| **136** | 281912 | 284037 | 286163 | 288290 | 290419 | 292549 | 294680 | 296812 |
| **144** | 298945 | 301080 | 303216 | 305354 | 307492 | 309632 | 311773 | 313916 |
| **152** | 316060 | 318206 | 320352 | 322500 | 324650 | 326801 | 328953 | 331107 |
| **160** | 333262 | 335419 | 337577 | 339737 | 341898 | 344061 | 346225 | 348391 |
| **168** | 350558 | 352727 | 354897 | 357069 | 359243 | 361418 | 363595 | 365773 |
| **176** | 367953 | 370135 | 372318 | 374503 | 376690 | 378879 | 381069 | 383261 |
| **184** | 385455 | 387650 | 389847 | 392046 | 394247 | 396450 | 398655 | 400861 |
| **192** | 403069 | 405279 | 407491 | 409705 | 411921 | 414139 | 416359 | 418581 |
| **200** | 420804 | 423030 | 425258 | 427488 | 429720 | 431954 | 434190 | 436428 |
| **208** | 438668 | 440910 | 443155 | 445401 | 447650 | 449901 | 452155 | 454410 |
| **216** | 456668 | 458928 | 461190 | 463455 | 465722 | 467991 | 470262 | 472536 |
| **224** | 474813 | 477091 | 479373 | 481656 | 483942 | 486231 | 488522 | 490815 |
| **232** | 493111 | 495410 | 497711 | 500015 | 502322 | 504631 | 506943 | 509257 |
| **240** | 511574 | 513894 | 516217 | 518542 | 520870 | 523201 | 525535 | 527872 |
| **248** | 530211 | 532553 | 534899 | 537247 | 539598 | 541952 | 544310 | 546670 |
| **256** | 549033 | 551399 | 553769 | 556142 | 558517 | 560896 | 563278 | 565664 |
| **264** | 568052 | 570444 | 572839 | 575238 | 577640 | 580045 | 582454 | 584866 |
| **272** | 587282 | 589701 | 592123 | 594549 | 596979 | 599412 | 601849 | 604290 |
| **280** | 606734 | 609183 | 611634 | 614090 | 616549 | 619013 | 621480 | 623951 |
| **288** | 626426 | 628905 | 631388 | 633875 | 636366 | 638862 | 641361 | 643865 |
| **296** | 646373 | 648885 | 651401 | 653922 | 656447 | 658976 | 661510 | 664049 |
| **304** | 666592 | 669139 | 671691 | 674248 | 676809 | 679375 | 681946 | 684522 |
| **312** | 687103 | 689688 | 692278 | 694874 | 697474 | 700080 | 702690 | 705306 |
| **320** | 707927 | 710553 | 713184 | 715821 | 718463 | 721111 | 723764 | 726423 |
| **328** | 729087 | 731757 | 734433 | 737115 | 739802 | 742495 | 745194 | 747899 |
| **336** | 750611 | 753328 | 756051 | 758781 | 761517 | 764259 | 767008 | 769763 |
| **344** | 772525 | 775294 | 778069 | 780850 | 783639 | 786435 | 789237 | 792047 |
| **352** | 794863 | 797687 | 800518 | 803357 | 806202 | 809056 | 811917 | 814785 |
| **360** | 817662 | 820546 | 823438 | 823438 |  |  |  |  |

### Definition of popCnt

The input to this process is the integer variable x.

The output of this process is the number of 1-valued bits present in the binary representation of x.

### Definition of iLog2

The input to this process is the variable x.

The output of this process is the variable y.

The function iLog2 is defined as follows:

y = Floor(Log(x) ÷ Log(2))

where Log( ) is the natural logarithmic function.

### Definition of iSqrt

The integer square root function, y = iSqrt( x ), is defined in terms of the inverse square root function invSqrt.

The input to this process is the variable x.

The output of this process is the variable y, derived as follows:

if (x <= (1 << 46))

y = 1 + ((x \* irsqrt(x)) >> 40);

else {

x' = (x + 65536) >> 16;

y = 1 + ((x' \* invSqrt(x')) >> 32);

}



### Definition of inverse square root function invSqrt

The input to this process is the variable pIn.

The output of this process is the variable pOut.

The variables pInScaled and nShift are a normzlised representation of pIn.

shift = −3;

pInScaled = pIn;

while (pIn & 0xffffffff00000000) {

pInScaled >>= 2;

nShift−−;

}

while (!(pInScaled & 0xc0000000)) {

pInScaled <<= 2;

nShift++;

}

A first approximation, invSqrtApprox, of the inverse square root is obtained using the arrays threeTimesR and rCubed.

idx = (pInScaled >> 25) − 32;

invSqrtApprox = threeTimesR[idx] − ((rCubed[idx] × pInScaled) >> 32);

A second apprixomation, invSqrtApprox2, is obtained as follows:

s = (invSqrtApprox × pInScaled) >> 32;

s = 0x30000000 − ((invSqrtApprox × s) >> 32);

invSqrtApprox2 = (invSqrtApprox × s) >> 32;

Finally, the output is obtained by inverse scaling the second approximation

if (nShift >= 0)

pOut = invSqrtApprox2 << nShift;

else

pOut = invSqrtApprox2 >> (−nShift);

Table 3 — the value of tableThreeR[ i + j ]

| **j** | **i** | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **0** | 3196059648 | 3145728000 | 3107979264 | 3057647616 | 3019898880 | 2969567232 |
| **6** | 2931818496 | 2894069760 | 2868903936 | 2831155200 | 2793406464 | 2768240640 |
| **12** | 2730491904 | 2705326080 | 2667577344 | 2642411520 | 2617245696 | 2592079872 |
| **18** | 2566914048 | 2541748224 | 2516582400 | 2491416576 | 2466250752 | 2441084928 |
| **24** | 2428502016 | 2403336192 | 2378170368 | 2365587456 | 2340421632 | 2327838720 |
| **30** | 2302672896 | 2290089984 | 2264924160 | 2252341248 | 2239758336 | 2214592512 |
| **36** | 2202009600 | 2189426688 | 2164260864 | 2151677952 | 2139095040 | 2126512128 |
| **42** | 2113929216 | 2101346304 | 2088763392 | 2076180480 | 2051014656 | 2038431744 |
| **48** | 2025848832 | 2013265920 | 2000683008 | 2000683008 | 1988100096 | 1962934272 |
| **54** | 1962934272 | 1950351360 | 1937768448 | 1925185536 | 1912602624 | 1900019712 |
| **60** | 1900019712 | 1887436800 | 1874853888 | 1862270976 | 1849688064 | 1849688064 |
| **66** | 1837105152 | 1824522240 | 1811939328 | 1811939328 | 1799356416 | 1786773504 |
| **72** | 1786773504 | 1774190592 | 1761607680 | 1761607680 | 1749024768 | 1736441856 |
| **78** | 1736441856 | 1723858944 | 1723858944 | 1711276032 | 1698693120 | 1698693120 |
| **84** | 1686110208 | 1686110208 | 1673527296 | 1660944384 | 1660944384 | 1648361472 |
| **90** | 1648361472 | 1635778560 | 1635778560 | 1623195648 | 1623195648 | 1610612736 |

Table 4 — the value of tableRCube[ i + j ]

| **j** | **i** | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **0** | 4195081216 | 3999986688 | 3857709056 | 3673323520 | 3538940928 | 3364924416 |
| **6** | 3238224896 | 3114735616 | 3034196992 | 2915990528 | 2800922624 | 2725880832 |
| **12** | 2615890944 | 2544223232 | 2439185408 | 2370818048 | 2303728640 | 2237913088 |
| **18** | 2173355008 | 2110061568 | 2048008192 | 1987165184 | 1927563264 | 1869150208 |
| **24** | 1840392192 | 1783783424 | 1728321536 | 1701024768 | 1647311872 | 1620883456 |
| **30** | 1568898048 | 1543306240 | 1492993024 | 1468236800 | 1443762176 | 1395656704 |
| **36** | 1372007424 | 1348605952 | 1302626304 | 1280060416 | 1257736192 | 1235650560 |
| **42** | 1213861888 | 1192294400 | 1171008512 | 1149979648 | 1108673536 | 1088379904 |
| **48** | 1068352512 | 1048567808 | 1029031936 | 1029036032 | 1009729536 | 971888640 |
| **54** | 971882496 | 953319424 | 934993920 | 916897792 | 899011584 | 881389568 |
| **60** | 881392640 | 864009216 | 846846976 | 829900800 | 813182976 | 813201408 |
| **66** | 796721152 | 780459008 | 764412928 | 764417024 | 748601344 | 732995584 |
| **72** | 733017088 | 717624320 | 702468096 | 702466048 | 687520768 | 672786432 |
| **78** | 672787456 | 658258944 | 658256896 | 643947520 | 629854208 | 629862400 |
| **84** | 615976960 | 615952384 | 602276864 | 588779520 | 588804096 | 575512576 |
| **90** | 575526912 | 562433024 | 562439168 | 549556224 | 549564416 | 536876032 |

### Definition of the integer division function Div

The inputs to this process are the variables dividend, divisor, and log2Scale.

The output of this process is the variable quotient, approximating 2log2Scale × dividend ÷ divisor, computed as follows:

excess = Max(0, ilog2(divisor) - 7)

index = (divisor + ((1 << excess) >> 1) >> excess) - 1

quotient = dividend \* invDivisor[index] >> 16 + excess - log2Scale

Table 1 — the value of invDivisor[ i + j ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **j** | **i** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **0** | 65536 | 32768 | 21845 | 16384 | 13107 | 10923 | 9362 | 8192 | 7282 | 6554 | 5958 | 5461 |
| **12** | 5041 | 4681 | 4369 | 4096 | 3855 | 3641 | 3449 | 3277 | 3121 | 2979 | 2849 | 2731 |
| **24** | 2621 | 2521 | 2427 | 2341 | 2260 | 2185 | 2114 | 2048 | 1986 | 1928 | 1872 | 1820 |
| **36** | 1771 | 1725 | 1680 | 1638 | 1598 | 1560 | 1524 | 1489 | 1456 | 1425 | 1394 | 1365 |
| **48** | 1337 | 1311 | 1285 | 1260 | 1237 | 1214 | 1192 | 1170 | 1150 | 1130 | 1111 | 1092 |
| **60** | 1074 | 1057 | 1040 | 1024 | 1008 | 993 | 978 | 964 | 950 | 936 | 923 | 910 |
| **72** | 898 | 886 | 874 | 862 | 851 | 840 | 830 | 819 | 809 | 799 | 790 | 780 |
| **84** | 771 | 762 | 753 | 745 | 736 | 728 | 720 | 712 | 705 | 697 | 690 | 683 |
| **96** | 676 | 669 | 662 | 655 | 649 | 643 | 636 | 630 | 624 | 618 | 612 | 607 |
| **108** | 601 | 596 | 590 | 585 | 580 | 575 | 570 | 565 | 560 | 555 | 551 | 546 |
| **120** | 542 | 537 | 533 | 529 | 524 | 520 | 516 | 512 | 508 | 504 | 500 | 496 |
| **132** | 493 | 489 | 485 | 482 | 478 | 475 | 471 | 468 | 465 | 462 | 458 | 455 |
| **144** | 452 | 449 | 446 | 443 | 440 | 437 | 434 | 431 | 428 | 426 | 423 | 420 |
| **156** | 417 | 415 | 412 | 410 | 407 | 405 | 402 | 400 | 397 | 395 | 392 | 390 |
| **168** | 388 | 386 | 383 | 381 | 379 | 377 | 374 | 372 | 370 | 368 | 366 | 364 |
| **180** | 362 | 360 | 358 | 356 | 354 | 352 | 350 | 349 | 347 | 345 | 343 | 341 |
| **192** | 340 | 338 | 336 | 334 | 333 | 331 | 329 | 328 | 326 | 324 | 323 | 321 |
| **204** | 320 | 318 | 317 | 315 | 314 | 312 | 311 | 309 | 308 | 306 | 305 | 303 |
| **216** | 302 | 301 | 299 | 298 | 297 | 295 | 294 | 293 | 291 | 290 | 289 | 287 |
| **228** | 286 | 285 | 284 | 282 | 281 | 280 | 279 | 278 | 277 | 275 | 274 | 273 |
| **240** | 272 | 271 | 270 | 269 | 267 | 266 | 265 | 264 | 263 | 262 | 261 | 260 |
| **252** | 259 | 258 | 257 | 256 |  |  |  |  |  |  |  |  |

### Definition of divExp2RoundHalfInf

The inputs to this process are the variables scalar and shift.

The output of this process is the variable value approximating scalar/2shift, computed as follows:

if (!shift) {

value = scalar;

} else {

s0 = 1 << (shift − 1);

value = scalar >= 0 ? (s0 + scalar) >> shift : −((s0 − scalar) >> shift)

}

### Definition of divExp2RoundHalfUp

The inputs to this process are the variables scalar and shift.

The output of this process is the variable value approximating scalar/2shift, computed as follows:

if (!shift) {

value = scalar;

} else {

s0 = 1 << (shift − 1);

value = (s0 + scalar) >> shift;

}

### Conversion of a tuple to 3D Morton code (TupleToMorton)

The input to this process is a three-tuple of variables ( s, t, v ).

The output of this process is the 3D Morton code representation, m, of the input tuple as follows:

Table 5 illustrates the construction of 3D morton codes from the bit string representation of the variables s, t, and v.

Table 5 — Construction of 3D Morton codes m from the tuple ( s, t, u )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bit string form** | | | | **Integer 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 |
| 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 |
| 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 |
| sn ... s1 s0 | tn ... t1 t0 | vn ... v1 v0 | sn tn vn  ... s1 t1 v1  s0 t0 v0 | ... |

### Conversion of 3D Morton codes to a tuple (MortonToTuple)

The input to this process is a variable m representing a 3D Morton code.

The output of this process is the three-tuple ( s, t, u) derived as follows:

### Definition of QpToQstep

The inputs to this process are:

the variable qP, representing the quantization parameter.

the variable isFirstComp

The output of this process is the variable qstep, representing a quantization step size and computed as follows:

if (isFirstComp)

qpBdOffset = 6 × (attribute\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] – 7)

else

qpBdOffset = 6 × (attribute\_secondary\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] – 7)

qP' = Clip3(4, 51 + qPBdOffset, qP);

qstep = levelScale[qP' % 6] << (qP' / 6);

Where the array levelScale is specified as levelScale[ k ] = { 161, 181, 203, 228, 256, 287 }, with k = 0 .. 5.

## Vector operations

The following mathematical functions are defined:

The function c[ i ] = CrossProduct ( a[ i ], b[ i ] ) with i = 0 .. 2 is defined as follows:

c[0] = a[1] × b[2] − a[2] × b[1]

c[1] = a[2] × b[0] − a[0] × b[2]

c[2] = a[0] × b[1] − a[1] × b[0]

The function c = InnerProduct ( a[i], b[i] ) with i = 0 .. 2 is defined as follows:

c = a[0] × b[0] + a[1] × b[1] + a[2] × b[2]

## Order of operation precedence

When order of precedence in an expression is not indicated explicitly by use of parentheses, 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 6 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

Table 6 – 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) |
| xy |
| "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 : z" |
| "x..y" |
| "x = y", "x += y", "x −= y" |

## 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 may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower case and upper-case letter and without any underscore characters. Variables starting with an upper-case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper-case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower-case letter are only used within the clause in which they are derived.

In some cases, "mnemonic" names for syntax element values or variable values are used interchangeably with their numerical values. Sometimes "mnemonic" names are used without any associated numerical values. The association of values and names is specified in the text. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an upper-case letter and may contain more upper-case letters.

NOTE – 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. These functions are specified in clause 7.2 and 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 parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in clause 5.9) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[ x ][ y ] or as syx. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[ x ].

A specification of values of the entries in rows and columns of an array may be denoted by { {...} {...} }, where each inner pair of brackets specifies the values of the elements within a row in increasing column order and the rows are ordered in increasing row order. Thus, setting a matrix s equal to { { 1 6 } { 4 9 } specifies that s[ 0 ][ 0 ] is set equal to 1, s[ 1 ][ 0 ] is set equal to 6, s[ 0 ][ 1 ] is set equal to 4, and s[ 1 ][ 1 ] is set equal to 9.

Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.

A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value different from zero.

## Text description of logical operations

In the text, a statement of logical operations as would be described mathematically in the following form:

if (condition 0)  
 statement 0  
else if (condition 1)  
 statement 1  
...  
else /\* informative remark on remaining condition \*/  
 statement n

may be described in the following manner:

... as follows / ... the following applies:

– If condition 0, statement 0

– Otherwise, if condition 1, statement 1

– ...

– Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".

In the text, a statement of logical operations as would be described mathematically in the following form:

if (condition 0a && condition 0b)  
 statement 0  
else if (condition 1a| | | | condition 1b)  
 statement 1  
...  
else  
 statement n

may be described in the following manner:

... as follows / ... the following applies:

– If all of the following conditions are true, statement 0:

– condition 0a

– condition 0b

– Otherwise, if one or more of the following conditions are true, statement 1:

– condition 1a

– condition 1b

– ...

– Otherwise, statement n

In the text, a statement of logical operations as would be described mathematically in the following form:

if (condition 0)  
 statement 0  
if (condition 1)  
 statement 1

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

## Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and upper-case variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification may also have a lower-case variable explicitly specified as input. Each process specification has explicitly specified an output. The output is a variable that can either be an upper-case variable or a lower-case variable.

When invoking a process, the assignment of variables is specified as follows:

– If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.

– Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

In the specification of a process, a specific coding block may be referred to by the variable name having a value equal to the address of the specific coding block.

# Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships

## Bitstream formats

This clause specifies the G-PCC bitstream. This clause is not an essential component of this document and all G-PCC components including any associated G-PCC GPSs or APSs could be encapsulated using a different format depending on application.

## Source, decoded, and output point cloud formats

This clause specifies the relationship between source and decoded point cloud that is given via the bitstream.

The point cloud source that is represented by the bitstream is a set of points in the decoding order.

The source and decoded point clouds are each comprised of one or more sample arrays:

– Geometry information – cartesian co-ordinates of the occupied point in 3-dimensional space (0 1 2, also known as XYZ).

– Single stimulus (Luma only, Reflectance).

– Colour, for example Green, Blue and Red (GBR, also known as RGB).

– Arrays representing other unspecified monochrome or multi-stimulus attribute samplings (for example, Frame index, Transparency).

The number of bits necessary for the representation of each of the samples in the co-ordinates arrays in a point cloud is in range of 8 to 32, inclusive.

The number of bits necessary for the representation of each of the samples in the attribute arrays in a point cloud is in the range of 8 to 16, inclusive. The number of bits used in the different attribute array may differ from the number of bits used in the other attribute arrays.

The order of the samples in the decoded point cloud is not specified. The order of the source point cloud and decoded point cloud may be different.

### Data partitioning

This subclause specifies how a frame is partitioned into tiles and slices.

Source point cloud data may be partitioned to multiple slices and can be encoded in a bitstream.

A slice is a set of points that can be encoded or decoded independently. A slice comprises one geometry data unit and zero or more attribute data units. Attribute data units depend upon the corresponding geometry data unit within the same slice. Within a slice, the geometry data unit must appear before any associated attribute units. The data units of a slice must be contiguous. The ordering of slices within a frame is unspecified.

A group of slices may be identified by a common tile identifier. This specification provides a tile inventory that describes a bounding box for each tile. A tile may overlap another tile in the bounding box. Each slice contains an index that identifies to which tile it belongs. Tile information is not used by the decoding process in this Specification.

### Frame index attribute component

Point cloud data consisting of multiple frames may be encoded by using frame combine coding. Arbitrary multiple frames may be combined into one input point cloud by preprocessing and each point of the input point cloud has a frame index as attribute component that indicate the frame to which the point belongs. The frame index is encoded as one of attribute component. After decoding the bitstream, each point may be split to multiple frames by using decoded frame index. When a frame index is encoded, it is recommended to set SliceQpY equal to 4 and unique\_geometry\_points\_flag equal to 0.

## Geometry octree

When the geometry octree is used, then the geometry encoding proceeds as follows. First, a cubical axis-aligned bounding box B is defined by the two extreme points and . An octree structure is then built by recursively subdividing B. At each stage, a cube is subdivided into 8 sub-cubes. An 8-bit code, named an occupancy code, is then generated by associating a 1-bit value with each sub-cube in order to indicate whether it contains points (i.e., full and has value 1) or not (i.e., empty and has value 0). Only full sub-cubes with a size greater than 1 (i.e., non-voxels) are further subdivided. Since points may be duplicated, multiple points may be mapped to the same sub-cube of size 1 (i.e., the same voxel). In order to handle such a situation, the number of points for each sub-cube of dimension 1 is also arithmetically encoded. The same arithmetic encoder is used to encode all the information put into the bitstream.

The decoding process starts by reading from the bitstream the dimensions of the bounding box B. The same octree structure is then built by subdividing B according to the occupancy codes. Each time a sub-cube of dimension 1 is reached, the number of points c for that sub-cube is arithmetically decoded and c points located at the origin of the sub-cube are generated.

## Neighbour relationships

### Neighbour dependent geometry octree child node scan order inverse mapping process

This process maps an index in one scan order to the corresponding index of another scan order.

The inputs to this process are

* an index, inIdx, in the neighbour dependent permuted child node scan order, and
* the neighbourhood occupancy pattern, neighbourPattern.

The output of this process is the corresponding index, outIdx, in the octree child node scan order.

The output index is determined as follows

outIdx = (childScanMap[neighbourPattern] >> (inIdx × 3)) & 7

Values of the array childScanMap are given by Table 7.

Table 7 — Values of childScanMap[ i + j ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **j** | | | |
| **i** | **0** | **1** | **2** | **3** |
| **0** | 0o76543210 | 0o76543210 | 0o10325476 | 0o76543210 |
| **4** | 0o54107632 | 0o54107632 | 0o10325476 | 0o32761054 |
| **8** | 0o32761054 | 0o76543210 | 0o32761054 | 0o54107632 |
| **12** | 0o32761054 | 0o10325476 | 0o76543210 | 0o76543210 |
| **16** | 0o26043715 | 0o46570213 | 0o20316475 | 0o57134602 |
| **20** | 0o04152637 | 0o45016723 | 0o01234567 | 0o23670145 |
| **24** | 0o62734051 | 0o67452301 | 0o23670145 | 0o45016723 |
| **28** | 0o73516240 | 0o01234567 | 0o67452301 | 0o67452301 |
| **32** | 0o37152604 | 0o57461302 | 0o31207564 | 0o46025713 |
| **36** | 0o15043726 | 0o54107632 | 0o10325476 | 0o32761054 |
| **40** | 0o73625140 | 0o76543210 | 0o32761054 | 0o54107632 |
| **44** | 0o62407351 | 0o10325476 | 0o76543210 | 0o76543210 |
| **48** | 0o37152604 | 0o02134657 | 0o64752031 | 0o57461302 |
| **52** | 0o26370415 | 0o73625140 | 0o57461302 | 0o13570246 |
| **56** | 0o40516273 | 0o31207564 | 0o15043726 | 0o75316420 |
| **60** | 0o73625140 | 0o51734062 | 0o37152604 | 0o76543210 |

### Neighbour depending geometry occupancy map permutation process

The inputs to this process are

* a neighbourhood occupancy pattern neighbourPattern
* a decoded occupancy map value occMap

The output of this process is a permuted occupancy map value occMapP.

The output is derived as follows

occMapP = 0

for (srcIdx = 0; srcIdx < 8; srcIdx++) {

dstIdx = (childScanMap[neighbourPattern] >> (srcIdx × 3)) & 7

occMapP = occMapP | (((occMap >> srcIdx) & 1) << dstIdx)

}

The values of the array childScanMap are given by Table 7.

# 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.

NOTE – An actual decoder should implement some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this Specification.

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 an expression used to specify conditions for the existence, type, and quantity of syntax elements, as in the following two examples \*/ |  |
| **syntax\_element** | ue(v) |
| conditioning 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 a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte\_aligned( ) is specified as follows:

– If the current position in the bitstream is on a byte boundary, i.e. the next bit in the bitstream is the first bit in a byte, the return value of byte\_aligned( ) is equal to TRUE.

– Otherwise, the return value of byte\_aligned( ) is equal to FALSE.

more\_data\_in\_byte\_stream( ), which is specified as follows:

– If more data follow in the byte stream, the return value of more\_data\_in\_byte\_stream( ) is equal to TRUE.

The following descriptors specify the parsing process of each syntax element. The parsing process for all descriptors and syntax elements is specified in clause 9.

– ae(v): adaptive arithmetic entropy-coded syntax element.

– de(v): dictionary coded syntax element.

– s(n): signed integer using n bits plus sign bit.

– se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first.

– u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.

– ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first.

– oid(v): an international object identifier as specified in Recommendation ITU-T X.660 | ISO/IEC 9834-1.

## Syntax in tabular form

### General

The syntax structures and the syntax elements within these structures are specified in this sub clause. Any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this Specification.

### Data unit and byte alignment syntax

#### Sequence parameter set syntax

|  |  |
| --- | --- |
| seq\_parameter\_set( ) { | **Descriptor** |
| **main\_profile\_compatibility\_flag** | u(1) |
| **reserved\_profile\_compatibility\_21bits** | u(21) |
| **slice\_reordering\_constraint\_flag** |  |
| **unique\_point\_positions\_constraint\_flag** | u(1) |
| **level\_idc** | u(8) |
| **sps\_seq\_parameter\_set\_id** | ue(v) |
| **sps\_bounding\_box\_present\_flag** | u(1) |
| if( sps\_bounding\_box\_present\_flag ) { |  |
| for( k = 0; k < 3; k++ ) |  |
| **sps\_bounding\_box\_offset\_xyz**[ k ] | se(v) |
| **sps\_bounding\_box\_offset\_log2\_scale** | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **sps\_bounding\_box\_size\_xyz**[ k ] | ue(v) |
| } |  |
| **sps\_source\_scale\_factor\_numerator\_minus1** | ue(v) |
| **sps\_source\_scale\_factor\_denominator\_minus1** | ue(v) |
| **sps\_num\_attribute\_sets** | ue(v) |
| for( i = 0; i < sps\_num\_attribute\_sets; i++ ) { |  |
| **attribute\_instance\_id**[ i ] | ue(v) |
| **attribute\_dimension\_minus1**[ i ] | ue(v) |
| **attribute\_bitdepth\_minus1**[ i ] | ue(v) |
| if(attribute\_dimension\_minus1[ i ] > 0 ) |  |
| **attribute\_secondary\_bitdepth\_minus1**[ i ] | ue(v) |
| **known\_attribute\_label\_flag**[ i ] | u(1) |
| if( known\_attribute\_label\_flag[ i ] ) |  |
| **known\_attribute\_label**[ i ] | ue(v) |
| else |  |
| **attribute\_label\_oid**[ i ] | oid(v) |
| **num\_attribute\_parameters** | u(8) |
| byte\_alignment( ) |  |
| for( j = 0; j < num\_attribute\_parameters; j++ ) |  |
| seq\_attribute\_parameters( i ) |  |
| } |  |
| **log2\_max\_frame\_idx** | u(5) |
| **axis\_coding\_order** | u(3) |
| **sps\_bypass\_stream\_enabled\_flag** | u(1) |
| **sps\_entropy\_continuation\_enabled\_flag** | u(1) |
| **sps\_extension\_flag** | u(1) |
| if( sps\_extension\_flag ) |  |
| while( more\_data\_in\_byte\_stream( ) ) |  |
| **sps\_extension\_data\_flag** | u(1) |
| byte\_alignment( ) |  |
| } |  |

#### Attribute parameters syntax

|  |  |
| --- | --- |
| seq\_attribute\_parameters( attrId ) { | **Descriptor** |
| **attr\_param\_type** | u(8) |
| **attr\_param\_len** | u(8) |
| if( attr\_param\_type == 0 ) { |  |
| **attr\_itu\_t\_t35\_country\_code** | u(8) |
| if( itu\_t\_t35\_country\_code == 255 ) |  |
| **attr\_itu\_t\_t35\_country\_code\_extension\_byte** | u(8) |
| seq\_attribute\_parameter\_data( AttrParamDataLen ) |  |
| } else if( attr\_param\_type == 1 ) { |  |
| **attr\_param\_oid** | oid(v) |
| seq\_attribute\_parameter\_data( AttrParamDataLen ) |  |
| } else if( attr\_param\_type == 2 ) { |  |
| **attribute\_cicp\_colour\_primaries**[ attrId ] | ue(v) |
| **attribute\_cicp\_transfer\_characteristics**[ attrId ] | ue(v) |
| **attribute\_cicp\_matrix\_coeffs**[ attrId ] | ue(v) |
| **attribute\_cicp\_video\_full\_range\_flag**[ attrId ] | u(1) |
| } else if( attr\_param\_type == 3 ) { |  |
| **attribute\_source\_offset\_log2\_plus1**[ attrId ] | ue(v) |
| **attribute\_source\_scale\_log2**[ attrId ] | ue(v) |
| } else if( attr\_param\_type == 4 ) { |  |
| for( k = 0; k <= attr\_num\_dimensions\_minus1[ attrId ]; k++ ) |  |
| **attr\_default\_value**[ k ] | u(v) |
| } else if( attr\_param\_type < 128 ) { |  |
| /\* Reserved for future use \*/ |  |
| } else |  |
| seq\_attribute\_parameter\_data( attrId, AttrParamDataLen ) |  |
| byte\_alignment() |  |
| } |  |

#### Attribute parameter data syntax

|  |  |
| --- | --- |
| seq\_attribute\_parameter\_data( attrId, numBytes ) { | **Descriptor** |
| for( i = 0; i < numBytes; i++) |  |
| **attr\_param\_byte**[ i ] | u(8) |
| } |  |

#### Tile inventory syntax

|  |  |
| --- | --- |
| tile\_inventory( ) { | **Descriptor** |
| **tile\_frame\_idx** | **tbu** |
| **tile\_seq\_parameter\_set\_id** | u(7) |
| **tile\_id\_present\_flag** | u(1) |
| **tile\_cnt** | u(16) |
| **tile\_bounding\_box\_bits** | u(8) |
| for( tileIdx = 0; tileIdx < tile\_cnt; tileIdx++ ) { |  |
| if( tile\_id\_present\_flag ) |  |
| **tile\_id** | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **tile\_bounding\_box\_offset\_xyz**[ tile\_id ][ k ] | s(v) |
| for( k = 0; k < 3; k++ ) |  |
| **tile\_bounding\_box\_size\_xyz**[ tile\_id ][ k ] | u(v) |
| } |  |
| for( k = 0; k < 3; k++ ) |  |
| **tile\_origin\_xyz**[ k ] | se(v) |
| **tile\_origin\_log2\_scale** | ue(v) |
| byte\_alignment( ) |  |
| } |  |

#### Geometry parameter set syntax

|  |  |
| --- | --- |
| geometry\_parameter\_set( ) { | **Descriptor** |
| **gps\_geom\_parameter\_set\_id** | ue(v) |
| **gps\_seq\_parameter\_set\_id** | ue(v) |
| **gps\_gsh\_box\_log2\_scale\_present\_flag** | u(1) |
| if( !gps\_gsh\_box\_log2\_scale\_present\_flag) |  |
| **gps\_gs\_box\_log2\_scale** | ue(v) |
| **geom\_octree\_flag** | u(1) |
| **unique\_geometry\_points\_flag** | u(1) |
| if( geom\_octree\_flag ) { |  |
| **geometry\_planar\_mode\_flag** | u(1) |
| if( geometry\_planar\_mode\_flag ){ |  |
| **geom\_planar\_mode\_th**[ 0 ] | ue(v) |
| **geom\_planar\_mode\_th**[ 1 ] | ue(v) |
| **geom\_planar\_mode\_th**[ 2 ] | ue(v) |
| **geom\_planar\_mode\_th\_idcm** | ue(v) |
| } |  |
| **geometry\_angular\_mode\_flag** | u(1) |
| if( geometry\_angular\_mode\_flag ){ |  |
| for( k = 0; k < 3; k++ ) |  |
| **geom\_angular\_origin\_xyz**[ k ] | se(v) |
| **number\_lasers\_minus1** | ue(v) |
| **laser\_angle**[ 0 ] | se(v) |
| **laser\_correction**[ 0 ] | ue(v) |
| **laser\_phi\_per\_turn**[ 0 ] | ue(v) |
| for( i = 1; i <= number\_lasers\_minus1; i++ ) { |  |
| **laser\_angle\_diff**[ i ] | ue(v) |
| **laser\_correction\_diff**[ i ] | se(v) |
| **laser\_phi\_per\_turn**[ i ] | ue(v) |
| } |  |
| **planar\_buffer\_disabled\_flag** | u(1) |
| } |  |
| **neighbour\_context\_restriction\_flag** | u(1) |
| **inferred\_direct\_coding\_mode\_enabled\_flag** | u(1) |
| **bitwise\_occupancy\_coding\_flag** | u(1) |
| **adjacent\_child\_contextualization\_enabled\_flag** | u(1) |
| **log2\_neighbour\_avail\_boundary** | ue(v) |
| **log2\_intra\_pred\_max\_node\_size** | ue(v) |
| **log2\_trisoup\_node\_size** | ue(v) |
| **geom\_scaling\_enabled\_flag** | u(1) |
| if( geom\_scaling\_enabled\_flag ) { |  |
| **geom\_base\_qp** | ue(v) |
| **geom\_direct\_coding\_mode\_qp\_offset** | se(v) |
| } |  |
| **geom\_tree\_coded\_axis\_list\_present\_flag** | u(1) |
| } |  |
| **gps\_extension\_flag** | u(1) |
| if( gps\_extension\_flag ) |  |
| while( more\_data\_in\_byte\_stream( ) ) |  |
| **gps\_extension\_data\_flag** | u(1) |
| byte\_alignment( ) |  |
| } |  |

#### Attribute parameter set syntax

|  |  |
| --- | --- |
| attribute\_parameter\_set( ) { | **Descriptor** |
| **aps\_attr\_parameter\_set\_id** | ue(v) |
| **aps\_seq\_parameter\_set\_id** | ue(v) |
| **attr\_coding\_type** | ue(v) |
| **aps\_attr\_initial\_qp\_minus4** | ue(v) |
| **aps\_attr\_chroma\_qp\_offset** | se(v) |
| **aps\_slice\_qp\_offset\_present\_flag** | u(1) |
| if(attr\_coding\_type = = 0) { //RAHT |  |
| **raht\_prediction\_enabled\_flag** | u(1) |
| if (raht\_prediction\_enabled\_flag) { |  |
| **raht\_prediction\_threshold0** | ue(v) |
| **raht\_prediction\_threshold1** | ue(v) |
| } |  |
| } |  |
| else if (attr\_coding\_type <= 2) { |  |
| **lifting\_num\_pred\_nearest\_neighbours\_minus1** | ue(v) |
| **lifting\_search\_range\_minus1** | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **lifting\_neighbour\_bias\_xyz**[ k ] | ue(v) |
| if ( attr\_coding\_type = = 2 ) |  |
| **lifting\_scalability\_enabled\_flag** | u(1) |
| if ( lifting\_scalability\_enabled\_flag ) |  |
| **lifting\_max\_nn\_range\_minus1** | u(2) |
| else { |  |
| **lifting\_num\_detail\_levels\_minus1** | ue(v) |
| if ( lifting\_num\_detail\_levels\_minus1 > 0 ) { |  |
| **lifting\_lod\_regular\_sampling\_enabled\_flag** | u(1) |
| for( idx = 0; idx < num\_detail\_levels\_minus1; idx++ ) { |  |
| if ( lifting\_lod\_regular\_sampling\_enabled\_flag ) |  |
| **lifting\_sampling\_period\_minus2**[ idx ] | ue(v) |
| else |  |
| **lifting\_sampling\_distance\_squared\_scale\_minus1**[ idx ] | ue(v) |
| if ( idx != 0 ) |  |
| **lifting\_sampling\_distance\_squared\_offset**[ idx ] | ue(v) |
| } |  |
| } |  |
| else |  |
| **lifting\_morton\_sort\_skip\_enabled\_flag** | u(1) |
| } |  |
| if( attr\_coding\_type = = 1 ) { |  |
| **lifting\_max\_num\_direct\_predictors** | ue(v) |
| if( lifting\_max\_num\_direct\_predictors) |  |
| **lifting\_adaptive\_prediction\_threshold** | ue(v) |
| **lifting\_intra\_lod\_prediction\_num\_layers** | ue(v) |
| **inter\_component\_prediction\_enabled\_flag** | u(1) |
| } |  |
| } |  |
| else if( attr\_coding\_type == 3 ) |  |
| **raw\_attr\_fixed\_width\_flag** | u(1) |
| **aps\_extension\_flag** | u(1) |
| if( aps\_extension\_flag ) |  |
| while( more\_data\_in\_byte\_stream( ) ) |  |
| **aps\_extension\_data\_flag** | u(1) |
| byte\_alignment( ) |  |
| } |  |

#### Frame boundary marker syntax

|  |  |
| --- | --- |
| frame\_boundary\_marker( ) { | **Descriptor** |
| /\* this syntax structure is intentionally empty \*/ |  |
| } |  |

#### Byte alignment syntax

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

### Geometry data unit syntax

#### General geometry data unit syntax

|  |  |
| --- | --- |
| geometry\_data\_unit ( ) { | **Descriptor** |
| geometry\_data\_unit\_header( ) |  |
| if( geom\_octree\_flag ) { |  |
| geometry\_octree\_root( ) |  |
| if( log2\_trisoup\_node\_size > 0 ) |  |
| geometry\_trisoup\_data( ) |  |
| } else |  |
| geometry\_predtree\_data( ) |  |
| geometry\_data\_unit\_footer( ) |  |
| } |  |

#### Geometry data unit header syntax

|  |  |
| --- | --- |
| geometry\_data\_unit\_header( ) { | **Descriptor** |
| **gsh\_geometry\_parameter\_set\_id** | ue(v) |
| **gsh\_tile\_id** | ue(v) |
| **gsh\_slice\_id** | ue(v) |
| **frame\_idx** | u(v) |
| if( !sps\_entropy\_continuation\_enabled\_flag ) { |  |
| **gsh\_entropy\_continuation\_flag** | u(1) |
| if( gsh\_entropy\_continuation\_flag ) |  |
| **gsh\_prev\_slice\_id** | ue(v) |
| } |  |
| if( gps\_gsh\_box\_log2\_scale\_present\_flag ) |  |
| **gsh\_box\_log2\_scale** | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **gsh\_box\_origin\_xyz**[ k ] | ue(v) |
| if( geom\_octree\_flag ) { |  |
| **geom\_tree\_depth\_minus1** | ue(v) |
| if( geom\_tree\_coded\_axis\_list\_present\_flag ) |  |
| for( lvl = 0; lvl <= geom\_tree\_depth\_minus1; lvl++ ) |  |
| for( k = 0; k < 3; k++ ) |  |
| **geom\_tree\_coded\_axis\_flag**[ lvl ][ k ] | u(1) |
| **gsh\_entropy\_stream\_cnt\_minus1** | ue(v) |
| if( gsh\_entropy\_stream\_cnt\_minus1 ) { |  |
| **gsh\_entropy\_stream\_len\_bits** | u(6) |
| for( i = 0; i < gsh\_entropy\_stream\_cnt\_minus1; i++) |  |
| **gsh\_entropy\_stream\_len**[ i ] | u(v) |
| } |  |
| } |  |
| if( geom\_scaling\_enabled\_flag ) { |  |
| **geom\_slice\_qp\_offset** | se(v) |
| **geom\_octree\_qp\_offsets\_depth** | ue(v) |
| } |  |
| if( log2\_trisoup\_node\_size ) { |  |
| **trisoup\_sampling\_value\_minus1** | ue(v) |
| **num\_unique\_segments\_minus1** | ue(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Geometry data unit footer syntax

|  |  |
| --- | --- |
| geometry\_data\_unit\_footer( ) { | **Descriptor** |
| byte\_alignment( ) |  |
| **geom\_num\_points\_minus1** | u(24) |
| } |  |

#### Geometry data unit data syntax

|  |  |
| --- | --- |
| geometry\_data\_unit\_data( ) { | **Descriptor** |
| depthS = depthT = depthV = 0 |  |
| for( depth = 0; depth <= geom\_tree\_depth\_minus1; depth++ ) { |  |
| for( nodeIdx = 0; nodeIdx < NumNodesAtDepth[ depth ]; nodeIdx++ ) { |  |
| sN = NodeS[ depthS ][ nodeIdx ] |  |
| tN = NodeT[ depthT ][ nodeIdx ] |  |
| vN = NodeV[ depthV ][ nodeIdx ] |  |
| geometry\_node( depthS, depthT, depthV, nodeIdx, sN, tN, vN ) |  |
| } |  |
| depthS += geom\_tree\_coded\_axis\_flag[ depth ][ 0 ] |  |
| depthT += geom\_tree\_coded\_axis\_flag[ depth ][ 1 ] |  |
| depthV += geom\_tree\_coded\_axis\_flag[ depth ][ 2 ] |  |
| } |  |
|  |  |
|  |  |
| } |  |

#### Geometry node syntax

|  |  |
| --- | --- |
| geometry\_node( depthS, depthT, depthV, nodeIdx, sN, tN, vN ) { | **Descriptor** |
| if( depth = = GeomScalingDepth ) { |  |
| **geom\_node\_qp\_offset\_eq0\_flag** | ae(v) |
| if( ! geom\_node\_qp\_offset\_eq0\_flag) { |  |
| **geom\_node\_qp\_offset\_sign\_flag** | ae(v) |
| **geom\_node\_qp\_offset\_abs\_minus1** | ae(v) |
| } |  |
| } |  |
| if( EffectiveDepth <= geom\_tree\_depth\_minus1 ) { |  |
| single\_occupancy( nodeIdx ) |  |
| if( ! single\_occupancy\_flag && !two\_planar\_flag[nodeIdx])) |  |
| if( bitwise\_occupancy\_flag ) |  |
| **occupancy\_map** | ae(v) |
| else |  |
| **occupancy\_byte** | de(v) |
| } |  |
| if( EffectiveDepthS >= NodeSizesLog2[ 0 ][ 0 ]− 1 &&  EffectiveDepthT >= NodeSizesLog2[ 0 ][ 1 ] – 1 &&  EffectiveDepthV >= NodeSizesLog2[ 0 ][ 2 ] – 1 ) { |  |
| if( !unique\_geometry\_points\_flag ) |  |
| for( child = 0; child < GeometryNodeChildrenCnt; child++ ) { |  |
| **num\_points\_eq1\_flag**[ child ] | ae(v) |
| if( !num\_points\_eq1\_flag ) |  |
| **num\_points\_minus2**[ child ] | ae(v) |
| } |  |
| } else { |  |
| if( geometry\_planar\_mode\_flag) { |  |
| for( child = 0; child < GeometryNodeChildrenCnt; child++ ) |  |
| for( axisIdx = 0; axisIdx <= 2; axisIdx++ ) |  |
| if( eligible\_planar\_flag[ axisIdx ] ) |  |
| geometry\_planar\_mode\_data( child, axisIdx ) |  |
| } |  |
| if( DirectModeFlagPresent ) |  |
| geometry\_direct\_mode\_data( 0 ) |  |
| } |  |
| } |  |

#### Single occupancy data syntax

|  |  |
| --- | --- |
| single\_occupancy( nodeIdx ) { | **Descriptor** |
| if ( !is\_planar\_flag[ nodeIdx ][ 0 ] ||   !is\_planar\_flag[ nodeIdx ][ 1 ] ||   !is\_planar\_flag[ nodeIdx ][ 2 ] ) { |  |
| if( NeighbourPattern = = 0 ) { |  |
| if( possibly\_planar[ nodeIdx ][ 0 ] &&   possibly\_planar[ nodeIdx ][ 1 ] &&   possibly\_planar[ nodeIdx ][ 2 ] ) { |  |
| **single\_occupancy\_flag** | ae(v) |
| if( single\_occupancy\_flag ) { |  |
| if( ! is\_planar\_flag[ nodeIdx ][ 0 ] ) |  |
| **occupancy\_idx**[ 0 ] | ae(v) |
| if( ! is\_planar\_flag[ nodeIdx ][ 1 ] ) |  |
| **occupancy\_idx**[ 1 ] | ae(v) |
| if( ! is\_planar\_flag[ nodeIdx ][ 2 ] ) |  |
| **occupancy\_idx**[ 2 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

#### Planar mode data syntax

|  |  |
| --- | --- |
| geometry\_planar\_mode\_data( childIdx, axisIdx ) { | **Descriptor** |
| **is\_planar\_flag**[ childIdx ][ axisIdx ] | ae(v) |
| if( is\_planar\_flag[ childIdx ][ axisIdx ] ) |  |
| **plane\_position**[ childIdx ][ axisIdx ] | ae(v) |
| } |  |

#### Direct mode data syntax

|  |  |
| --- | --- |
| geometry\_direct\_mode\_data( child ) { | **Descriptor** |
| **direct\_mode\_flag** | ae(v) |
| if( direct\_mode\_flag ) { |  |
| **direct\_point\_cnt\_eq2\_flag** | ae(v) |
| if( !unique\_geometry\_points\_flag && !direct\_point\_cnt\_eq2\_flag ) { |  |
| **direct\_dup\_point\_cnt\_eq0\_flag** | ae(v) |
| if( !direct\_point\_cnt\_eq1\_flag ) { |  |
| **direct\_dup\_point\_cnt\_eq1\_flag** | ae(v) |
| if( direct\_dup\_point\_cnt\_eq1\_flag ) |  |
| **direct\_dup\_point\_cnt\_minus2** | ae(v) |
| } |  |
| } |  |
| for( i = 0; i <= direct\_point\_cnt\_eq2\_flag; i++ ){ |  |
| for( k = 0; k < 3; k++ ) { |  |
| if( ChildNodeSizeLog2[ k ] >= 1  && ( !is\_planar\_flag[ child ][ k ]  || !geom\_tree\_coded\_axis\_flag[ k ])) { |  |
| **point\_offset**[ i ][ k ][ 0 ] | ae(v) |
| if( geometry\_angular\_mode\_flag ) { |  |
| **laser\_residual\_eq0\_flag** | ae(v) |
| if( !laser\_residual\_eq0\_flag ) { |  |
| **laser\_residual\_sign** | ae(v) |
| **laser\_residual\_eq1\_flag** | ae(v) |
| if( !laser\_residual\_eq1\_flag ) { |  |
| **laser\_residual\_eq2\_flag** | ae(v) |
| if( !laser\_residual\_eq2\_flag ) |  |
| **laser\_residual\_minus3** | ae(v) |
| } |  |
| } |  |
| } |  |
| for( j = 1; j < EffectiveChildNodeSizeLog2[ k ]; j++ ) |  |
| **point\_offset**[ i ][ k ][ j ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Geometry trisoup data syntax

|  |  |
| --- | --- |
| geometry\_trisoup\_data( ) { | **Descriptor** |
| for( i = 0; i <= num\_unique\_segments\_minus1; i++ ) |  |
| **segment\_indicator**[ i ] | ae(v) |
| for( i = 0; i < NumTrisoupVertices; i++ ) |  |
| **vertex\_position**[ i ] | ae(v) |
| } |  |

#### Geometry predictive tree syntax

|  |  |
| --- | --- |
| geometry\_predtree\_data( ) { | **Descriptor** |
| PtnNodeIdx = 0 |  |
| do { |  |
| geometry\_predtree\_node( PtnNodeIdx ) |  |
| } while( NodeIdx <= geom\_num\_points\_minus1 ) |  |
| } |  |

#### Geometry predictive tree node syntax

|  |  |
| --- | --- |
| geometry\_predtree\_node( nodeIdx ) { | **Descriptor** |
| if( unique\_geometry\_points\_flag ) { |  |
| **ptn\_point\_cnt\_gt1\_flag** | ae(v) |
| if( ptn\_point\_cnt\_gt1\_flag ) |  |
| **ptn\_point\_cnt\_minus2** | ae(v) |
| } |  |
| **ptn\_child\_cnt**[ nodeIdx ] | ae(v) |
| **ptn\_pred\_mode**[ nodeIdx ] | ae(v) |
| for( k = 0; k < 3; k++ ) { |  |
| **ptn\_residual\_eq0\_flag**[ k ] | ae(v) |
| if( !ptn\_residual\_eq0\_flag[ k ] ) { |  |
| if( ptn\_pred\_mode > 0 ) |  |
| **ptn\_residual\_sign\_flag**[ k ] | ae(v) |
| **ptn\_residual\_abs\_log2**[ k ] | ae(v) |
| **ptn\_residual\_abs\_remaining**[ k ] | ae(v) |
| } |  |
| } |  |
| for( i = 0; i < ptn\_child\_cnt; i++) |  |
| geometry\_predtree\_node( ++PtnNodeIdx ) |  |
| } |  |

### Attribute data unit syntax

#### General attribute data unit syntax

|  |  |
| --- | --- |
| attribute\_data\_unit ( ) { | **Descriptor** |
| attribute\_data\_unit\_header( ) |  |
| if( attr\_coding\_type != 3 ) |  |
| attribute\_data\_unit\_data( ) |  |
| else |  |
| attribute\_data\_unit\_raw\_data( ) |  |
| } |  |

#### Attribute data unit header syntax

|  |  |
| --- | --- |
| attribute\_data\_unit\_header( ) { | **Descriptor** |
| **ash\_attr\_parameter\_set\_id** | ue(v) |
| **ash\_attr\_sps\_attr\_idx** | ue(v) |
| **ash\_attr\_geom\_slice\_id** | ue(v) |
| if ( aps\_slice\_qp\_offset\_present\_flag ) { |  |
| for( k = 0; k < Min( 2, AttrDim ); k++) |  |
| **ash\_attr\_qp\_offset**[ k ] | se(v) |
| } |  |
| **ash\_attr\_layer\_qp\_offset\_present\_flag** | u(1) |
| if ( ash\_attr\_layer\_qp\_offset\_present\_flag ) { |  |
| **ash\_attr\_num\_layer\_qp\_minus1** | ue(v) |
| for( i = 0; i < NumQpLayers; i++ ){ |  |
| for( k = 0; k < Min( 2, AttrDim ); k++ ) |  |
| **ash\_attr\_layer\_qp\_offset**[ i ][ k ] | se(v) |
| } |  |
| } |  |
| **ash\_attr\_region\_cnt** | ue(v) |
| for( i = 0; i < ash\_attr\_region\_cnt; i++ ) { |  |
| for( k = 0; k < 3; k++ ) |  |
| **ash\_attr\_qp\_region\_origin\_xyz**[ i ][ k ] | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **ash\_attr\_qp\_region\_size\_minus1\_xyz**[ i ][ k ] | ue(v) |
| for( k = 0; k < Min( 2, AttrDim ); k++) |  |
| **ash\_attr\_region\_qp\_offset**[ i ][ k ] | se(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute data unit data syntax

|  |  |
| --- | --- |
| attribute\_data\_unit\_data( ) { | **Descriptor** |
| AttrDim = attribute\_dimension\_minus1[ ash\_attr\_sps\_attr\_idx ] + 1 |  |
| **all\_residual\_values\_equal\_to\_zero\_run** | ae(v) |
| for( i = 0; i < PointCount; i++ ) { |  |
| if( attr\_coding\_type = = 1 &&  MaxPredDiff[ i ] >= lifting\_adaptive\_prediction\_threshold &&  MaxNumPredictors > 1 ) { |  |
| **pred\_index**[ i ] | ae(v) |
| } |  |
| if( all\_residual\_values\_equal\_to\_zero\_run > 0 ) { |  |
| for( k = 0; k < AttrDim; k++ ) |  |
| CoeffLevel[ k ][ i ] = 0 |  |
| all\_residual\_values\_equal\_to\_zero\_run −= 1 |  |
| } else { |  |
| attribute\_coding( i ) |  |
| **all\_residual\_values\_equal\_to\_zero\_run** | ae(v) |
| } |  |
| } |  |
| } |  |

#### Attribute value syntax

|  |  |
| --- | --- |
| attribute\_coding( pointIdx ) { | **Descriptor** |
| for ( k = 0; k < AttrDim; k++ ) { |  |
| **coeff\_eq0\_flag**[ k ] | ae(v) |
| if( ! coeff\_eq0\_flag[ k ] ) { |  |
| **coeff\_abs\_level\_eq1\_flag**[ k ] | ae(v) |
| if ( ! coeff\_abs\_level\_eq1\_flag[ k ] ) { |  |
| **coeff\_abs\_level\_low**[ k ] | ae(v) |
| if (coeff\_abs\_level\_low[ k ] = = 255 ) |  |
| **coeff\_abs\_level\_remaining**[ k ] | ae(v) |
| } |  |
| } |  |
| if( AttrDim == 1 || ! coeff\_eq0\_flag[ k ] ) | ae(v) |
| **coeff\_sign\_flag**[ k ] | ae(v) |
| } |  |
| } |  |

#### Raw attribute value syntax

|  |  |
| --- | --- |
| attribute\_data\_unit\_raw\_data( ) { | **Descriptor** |
| for ( idx = 0; idx < NumPoints; idx++ ) |  |
| for ( k = 0; k < AttrDim; k++ ) { |  |
| if( !raw\_attr\_fixed\_width\_flag ) |  |
| **raw\_attr\_component\_length** | u(8) |
| **raw\_attr\_value**[ idx ][ k ] | u(v) |
| } |  |
| } |  |

### Defaulted attribute data unit syntax

|  |  |
| --- | --- |
| defaulted\_attribute\_data\_unit ( ) { | **Descriptor** |
| **defattr\_seq\_parameter\_set\_id** | ue(v) |
| **defattr\_sps\_attr\_idx** | ue(v) |
| **defattr\_geom\_slice\_id** | ue(v) |
| for( k = 0; k < AttrDim; k++ ) |  |
| **defattr\_value**[ k ] | u(v) |
| } |  |

## Semantics

### General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this sub clause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the unless otherwise specified in this Specification.

### Data unit and byte alignment semantics

#### Sequence parameter set semantics

**main\_profile\_compatibility\_flag** equal to 1 specifies that the bitstream conforms to the Main profile. main\_profile\_compatibility\_flag equal to 0 specifies that the bitstream conforms to a profile other than the Main profile.

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

**slice\_reordering\_constraint\_flag** indicates that the bitstream is sensitive to the reordering and removal of data units.

**unique\_point\_positions\_constraint\_flag** equal to 1 indicates that in each point cloud frame that refers to the current SPS, all output points have unique positions. unique\_point\_positions\_constraint\_flag equal to 0 indicates that in any point cloud frame that refers to the current SPS, two and more output points may have the same position.

Note – For example, even if all points are unique in each slices, the point from different slices in a frame may overlap. In that case, unique\_point\_positions\_constraint\_flag should be set to 0.

**level\_idc** indicates a 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** provides an identifier for the SPS for reference by other syntax elements. In the value of sps\_seq\_parameter\_set\_id shall be 0 in bitstreams conforming to this version of this Specification. The value other than 0 for sps\_seq\_parameter\_set\_id is reserved for future use by ISO/IEC.

**sps\_bounding\_box\_present\_flag** equal to 1 indicates that a bounding box is present in the sequence parameter set. sps\_bounding\_box\_present\_flag equal to 0 indicates that the size of the bounding box is undefined.

**sps\_bounding\_box\_offset\_xyz**[ k ] indicates the k-th component of the quantized ( x, y, z ) co-ordinate offset of the source bounding box in Cartesian co-ordinates. When not present, the values of sps\_bounding\_box\_offset\_xyz[ k ] are each inferred to be 0.

**sps\_bounding\_box\_offset\_log2\_scale** indicates the scaling factor to scale the quantized x, y, and z source bounding box offsets. When not present, the value of sps\_bounding\_box\_offset\_log2\_scale is inferred to be 0.

**sps\_bounding\_box\_size\_xyz**[ k ] indicates the k-th component of the width, height, and depth, respectively, of the source bounding box in Cartesian co-ordinates.

**sps\_source\_scale\_factor\_numerator\_minus1** plus 1 indicates the scale factor numerator of the source point cloud.

**sps\_source\_scale\_factor\_denominator\_minus1** plus 1 indicates the scale factor denominator of the source point cloud.

**sps\_num\_attribute\_sets** indicates the number of coded attributes in the bitstream. The value of sps\_num\_attribute\_sets shall be in the range of 0 to 63.

It is a requirement of bitstream conformance that all slices have attribute data units that corresponds to all attribute component enumerated in SPS.

**attribute\_dimension\_minus1**[ i ]plus 1 specifies the number of components of the i-th attribute.

**attribute\_instance\_id**[ i ]specifies the instance id for the i-th attribute.

NOTE – The value of the attribute\_instance\_id may be used to differentiate attributes with identical attribute labels. For example, it is useful for the point cloud having multiple color from the different view point.

**attribute\_bitdepth\_minus1**[i]plus 1 specifies the bitdepth for first component of the i-th attribute signal(s).

**attribute\_secondary\_bitdepth\_minus1**[i]plus 1 specifies the bitdepth for secondary component of the i-th attribute signal(s).

**known\_attribute\_label\_flag**[ i ], **known\_attribute\_label**[ i ], and **attribute\_label\_oid**[ i ] together identify the type of data conveyed in the i-th attribute. known\_attribute\_label\_flag[ i ] indicates whether the attribute is identified by the value of known\_attibute\_label[ i ] or by the object identifier attribute\_label\_oid[ i ].

Attribute types identified by known\_attribute\_label are specified in Table 8. Values of known\_attribute\_label not specified are reserved for future use by this Specification.

Attribute types identified by attribute\_label\_oid are not specified in this Specification. attribute\_label\_oid is an international object identifier as specified in Recommendation ITU-T X.660 | ISO/IEC 9834-1.

Table 8 — Identification of attribute type by known\_attribute\_label

|  |  |
| --- | --- |
| **known\_attribute\_label** | **Attribute type** |
| 0 | Colour |
| 1 | Reflectance |
| 2 | Frame index |
| 3 | Material ID |
| 4 | Transparency |
| 5 | Normals |

**log2\_max\_frame\_idx** plus 1 specifies the number of bits used to signal the frame\_idx syntax variable.

**axis\_coding\_order** specifies the correspondence between the X, Y, and Z output axis labels and the three position components of all points in the reconstructed point cloud.

The array XyzToStv defines the mapping of the k-th component of an ( x, y, z ) co-ordinate to an index of the coded geometry axis order ( s, t, v ). Values of XyzToStv[ k ], k = 0 .. 2, are defined according to axis\_coding\_order in Table 9.

The output axis labels X, Y, and Z are each assigned to the axis index given by XyzToStv[ k ], for k = 0 .. 2, according to Table 10.

Table 9 — Definition of XyzToStv[ k ] according to the value of axis\_coding\_order

|  |  |  |  |
| --- | --- | --- | --- |
| axis\_coding\_order | XyzToStv[ k ] | | |
| 0 | 1 | 2 |
| 0 | 2 | 1 | 0 |
| 1 | 0 | 1 | 2 |
| 2 | 0 | 2 | 1 |
| 3 | 2 | 0 | 1 |
| 4 | 2 | 1 | 0 |
| 5 | 1 | 2 | 0 |
| 6 | 1 | 0 | 2 |
| 7 | 0 | 1 | 2 |

Table 10 — Mapping of output X, Y, and Z axis labels to indicies axis of RecPic[ pointIdx ][ axis ]

|  |  |
| --- | --- |
| Label | axis |
| X | XyzToStv[ 0 ] |
| Y | XyzToStv[ 1 ] |
| Z | XyzToStv[ 2 ] |

**sps\_bypass\_stream\_enabled\_flag** equal to 1 specifies that the bypass coding mode may be used on reading the bitstream. sps\_bypass\_stream\_enabled\_flag equal to 0 specifies that the bypass coding mode is not used on reading the bitstream.

**sps\_entropy\_continuation\_enabled\_flag** equal to 1 indicates that a slice's initial entropy context state may depend upon the final entropy context state of the preceeding slice. sps\_entropy\_continuation\_enabled\_flag equal to 0 specifies that the initial entropy context state of each slice is independent. It is a requirement of bitstream conformance that sps\_entropy\_continuation\_enabled\_flag is sequal to 0 when slice\_reordering\_constaint\_flag is equal to 0.

**sps\_extension \_flag** equal to 0 specifies that no sps\_extension\_data\_flag syntax elements are present in the SPS syntax structure. sps\_extension \_flag shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for sps\_extension \_flag is reserved for future use by ISO/IEC. Decoders shall ignore all sps\_extension\_data\_flag syntax elements that follow the value 1 for sps\_extension\_flag in an SPS syntax structure.

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

#### Attribute parameters semantics

**attr\_param\_type** identifies the type of attribute parameter data according to Table 12.

Table 12— Identification of attribute parameter type by attr\_param\_type

|  |  |
| --- | --- |
| **attr\_param\_type** | **Description** |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 | Attribute source scale and offset |
| 4 |  |
| 5 .. 128 |  |
| 129 .. 255 |  |

**attr\_param\_len** indicates the length in bytes of the attribute parameter.

**attr\_param\_itu\_t\_t35\_country\_code** ... TBD ...

**attr\_param\_itu\_t\_t35\_country\_code\_extension\_flag** ... TBD ...

**attr\_param\_oid** ... TBD ...

**attribute\_source\_offset\_log2\_plus1**[ i ] indicates the offset factor to convert the decoded attribute value to source attribute value. When not present, the offset value is inferred to be 0. The offset value is equal to ( 1 << attribute\_source\_offset\_log2\_plus1[ i ] ) >> 1.

**attribute\_source\_scale\_log2**[ i ] indicates the scaling factor to convert the decoded attribute value to source attribute value. When not present, the scaling value is inferred to be 0. The scaling value is equal to 1 << attribute\_source\_scale\_log2[ i ].

**attr\_default\_value**[ k ] specifies the k-th component of the default value of the attrId-th attribute. The length n is equal to attr\_bitdepth\_primary when k = 0, and attr\_bitdepth\_secondary otherwise.

#### Attribute parameter data semantics

**attr\_param\_byte**[ i ] ... TBD ...

#### Tile inventory semantics

**tile\_frame\_idx** contains an identifying number that may be used to identify the purpose of the tile inventory.

**tile\_seq\_parameter\_set\_id** specifies the value of sps\_seq\_parameter\_set\_id for the active SPS. The value of gps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**tile\_id\_present\_flag** equal to 1 specifies that tiles are identified according to the value of the tile\_id syntax element. tile\_id\_present\_flag equal to 0 specifies that tiles are identified according to their position in the tile inventory.

**tile\_cnt** specifies the number of tile bounding boxes present in the tile inventory.

**tile\_bounding\_box\_bits** specifies the bitdepth to represent the bounding box information for the tile inventory.

**tile\_id** identifies a particular tile within the tile\_inventory. When not present, the value of tile\_id is inferred to be the index of the tile within the tile inventory as given by the loop variable tileIdx. It is a requirement of bitstream conformance that all values of tile\_id are unique within a tile inventory.

**tile\_bounding\_box\_offset\_xyz**[ tileId ][ k ] and **tile\_bounding\_box\_size\_xyz**[ tileId ][ k ] specify a bounding box encompasing slices identified by gsh\_tile\_id equal to tileId.

tile\_bounding\_box\_offset\_xyz[ tileId ][ k ] is the k-th component of the ( x, y, z ) origin co-ordinate of the tile bounding box relative to TileOrigin[ k ].

tile\_bounding\_box\_size\_xyz[ tileId ][ k ] is the k-th component of the tile bounding box width, height, and depth, respectively.

**tile\_origin\_xyz**[ k ] specifies the k-th component of the tile origin in Cartesian co-ordinates. The value of tile\_origin\_xyz[ k ] should be equal to sps\_bounding\_box\_offset[ k ].

**tile\_origin\_log2\_scale** specifies a scaling factor to scale components of tile\_origin\_xyz. The value of tile\_origin\_log2\_scale should be equal to sps\_bounding\_box\_offset\_log2\_scale.

The array TileOrigin, with elements TileOrigin[ k ] for k = 0 .. 2, is derived as follows:

TileOrigin[ k ] = tile\_origin\_xyz[ k ] << tile\_origin\_log2\_scale

#### Geometry parameter set semantics

**gps\_geom\_parameter\_set\_id** provides an identifier for the GPS for reference by other syntax elements. The value of gps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**gps\_seq\_parameter\_set\_id** specifies the value of sps\_seq\_parameter\_set\_id for the active SPS. The value of gps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**gps\_gsh\_box\_log2\_scale\_present\_flag** equal to 1 specifies gsh\_box\_log2\_scale is signalled in each geometry slice header that references the current GPS. gps\_gsh\_box\_log2\_scale\_present\_flag equal to 0 specifies gsh\_box\_log2\_scale is not signalled in each geometry slice header and common scale for all slices is signalled in gps\_gsh\_box\_log2\_scale of current GPS.

**gps\_gs\_box\_log2\_scale** indicates a scale factor to be applied to the slice origin of all slices that reference the current GPS.

**unique\_geometry\_points\_flag** equal to 1 indicates that in all slices that refer to the current GPS, all output points have unique positions within a slice. unique\_geometry\_points\_flag equal to 0 indicates that in all slices that refer to the current GPS, two or more of the output points may have same positions within a slice.

**geometry\_planar\_mode\_flag** equal to 1 indicates that the planar coding mode is activated. geometry\_planar\_mode\_flag equal to 0 indicates that the planar coding mode is not activated.

**geom\_planar\_mode\_th\_idcm** specifies the value of the threshold of activation for the direct coding mode. geom\_planar\_mode\_th\_idcm is an integer in the range 0 to 127 inclusive. When not present, geom\_planar\_mode\_th\_idcm is inferred to be 127.

**geom\_planar\_mode\_th**[i], for i in the range 0 .. 2,specifies the value of the threshold of activation for planar coding mode along the i-th most probable direction for the planar coding mode to be efficient. geom\_planar\_mode\_th[i] is an integer in the range 0 .. 127.

**geometry\_angular\_mode\_flag** equal to 1 indicates that the angular coding mode is activated. geometry\_angular\_mode\_flag equal to 0 indicates that the angular coding mode is not activated.

**geom\_angular\_origin\_xyz**[ k ] specifies the k-th component of the ( x, y, z ) co-ordinate of the origin used in the processing of the angular coding mode.When not present, geom\_angular\_origin\_x, geom\_angular\_origin\_y, and geom\_angular\_origin\_z are inferred to be 0.

The array geomAngularOrigin, with values geomAngularOrigin[ k ], for k = 0 .. 2, represents the values of geom\_angular\_origin\_xyz permuted into the coded geometry axis order as follows:

geomAngularOrigin[XyzToStv[k]] = geom\_angular\_origin\_xyz[k], for k = 0..2

**number\_lasers** specifies the number of lasers used for the angular coding mode. When not present, number\_lasers is inferred to be 0.

**laser\_angle**[ i ] and **laser\_angle\_diff**[ i ], for i in the range 0 .. number\_lasers\_minus1, specify the tangent of the elevation angle of the i-th laser relative to the horizontal plane defined by the first and second coded axes.When not present, laser\_angle[ i ] is inferred to be 0.

**laser\_correction**[ i ] and **laser\_correction\_diff**[ i ], for i in the range 0 .. number\_lasers\_minus1, specifies the correction, along the second internal axis, of the i-th laser position relative to the geomAngularOrigin[ 2 ]. When not present, laser\_correction[ i ] is inferred to be 0.

The arrays LaserAngle and LaserCorrection, with elements laserAngle[ i ] and LaserCorrection[ i ], for i in the range of 0 .. number\_lasers\_minus1, are derived as follows:

LaserAngle[0] = laser\_angle[0]

LaserCorrection[0] = laser\_correction[0]

for (i = 1; i <= number\_lasers\_minus1; i++) {

LaserAngle[i] = LaserAngle[i - 1] + laser\_angle\_diff[i]

LaserCorrection[i] = LaserCorrection[i - 1] + laser\_correction\_diff[i]

}

**laser\_phi\_per\_turn**[ i ], specifies the number of samples produced by the i-th laser of a rotating sensing sytem located at the origin used in the processing of the angular coding mode. When not present, laser\_phi\_per\_turn[ i ] is inferred to be 1.

The arrays DeltaPhi and InvDeltaPhi, with elements DeltaPhi[ i ] and InvDeltaPhi[ i ], for i = 0 .. number\_lasers\_minus1, are derived as follows:

for (i = 0; i <= number\_lasers\_minus1; i++) {

DeltaPhi[i] = 6588397 / laser\_phi\_per\_turn[i]

InvDeltaPhi[i] = (laser\_phi\_per\_turn[i] <<30) / 6588397

}

**planar\_buffer\_disabled\_flag** equal to 1 indicates that tracking the closest nodes using a buffer is not used in process of coding the planar mode flag and the plane position in the planar mode. planar\_buffer\_disabled\_flag equal to 0 indicates that tracking the closest nodes using a buffer is used. When not present, planar\_buffer\_disabled\_flag is inferred to be 0.

**neighbour\_context\_restriction\_flag** equal to 0 indicates that geometry node occupancy of the current node is coded with the contexts determined from neighbouring nodes which is located inside the parent node of the current node. neighbour\_context\_restriction\_flag equal to 0 indicates that geometry node occupancy of the current node is coded with the contexts determined from neighbouring nodes which is located inside or outside the parent node of the current node.

**inferred\_direct\_coding\_mode\_enabled\_flag** equal to 1 indicates that direct\_mode\_flag may be present in the geometry node syntax. inferred\_direct\_coding\_mode\_enabled\_flagequal to 0 indicates that direct\_mode\_flag is not present in the geometry node syntax.

**bitwise\_occupancy\_coding\_flag** equal to 1 indicates that geometry node occupancy is encoded using bit-wise contextualisation of the syntax element ocupancy\_map. bitwise\_occupancy\_coding\_flag equal to 0 indicates that geometry node occupancy is encoded using the dictionary encoded syntax element occypancy\_byte.

**adjacent\_child\_contextualization\_enabled\_flag** equal to 1 indicates that the adjacent children of neighbouring octree nodes are used for bit-wise occupancy contextualization. adjacent\_child\_contextualization\_enabled\_flag equal to 0 indicates that the children of neighbouring octree nodes are is not used for the occupancy contextualization.

**log2\_neighbour\_avail\_boundary** specifies the variable NeighbAvailabilityMask as follows.

When neighbour\_context\_restriction\_flag is equal to 1, NeighbAvailabilityMask is set equal to 1. Otherwise, neighbour\_context\_restriction\_flag equal to 0, NeighbAvailabilityMask is set equal to 1 << log2\_neighbour\_avail\_boundary.

**log2\_intra\_pred\_max\_node\_size** specifies the octree node size eligible for occupancy intra prediction.

**log2\_trisoup\_node\_size** specifies the variable TrisoupNodeSize as the size of the triangle nodes as follows.

TrisoupNodeSize = 1 << log2\_trisoup\_node\_size

When log2\_trisoup\_node\_size is equal to 0, the geometry bitstream includes only the octree coding syntax. When not present, log2\_trisoup\_node\_size is not present is inferred to be 0.

When log2\_trisoup\_node\_size is greater than 0, it is a requirement of bitstream conformance that:

* inferred\_direct\_coding\_mode\_enabled\_flag must be equal to 0, and
* unique\_geometry\_points\_flag must be equal to 1.

**geom\_scaling\_enabled\_flag** equal to 1 specifies that a scaling process for geometry positions is invoked during the geometry slice decoding process. geom\_scaling\_enabled\_flag equal to 0 specifies that geometry positions do not require scaling. When not present, geom\_scaling\_enabled\_flag is inferred to be 0.

**geom\_base\_qp** specifies the base value of the geometry position quantization parameter. When not present, geom\_base\_qp is inferred to be 0.

**geom\_direct\_coding\_mode\_qp\_offset** specifies an offset relative to the sliceQp for use in scaling point positions coded by the direct coding mode.

**geom\_tree\_coded\_axis\_list\_present\_flag** equal to 1 specifies that each geometry data unit contains geom\_tree\_coded\_axis\_flag syntax elements used to derive the geometry root node size. geom\_tree\_coded\_axis\_list\_present\_flag equal to 0 specifies that geom\_tree\_coded\_axis\_flag syntax elements are not present in the geometry data unit syntax. and that the coded geometry tree represents a cubic volume.

**gps\_extension \_flag** equal to 0 specifies that no gps\_extension\_data\_flag syntax elements are present in the GPS syntax structure. gps\_extension\_flag shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for gps\_extension\_flag is reserved for future use by ISO/IEC. Decoders shall ignore all gps\_extension\_data\_flag syntax elements that follow the value 1 for gps\_extension\_flag in a GPS syntax structure.

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

#### Attribute parameter set semantics

**aps\_attr\_parameter\_set\_id** provides an identifier for the APS for reference by other syntax elements. The value of aps\_attr\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**aps\_seq\_parameter\_set\_id** specifies the value of sps\_seq\_parameter\_set\_id for the active SPS. The value of aps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**attr\_coding\_type** indicates that the coding type for the attribute in Table 11 for the given value of attr\_coding\_type. The value of attr\_coding\_type shall be equal to 0, 1, or 2 in bitstreams conforming to this version of this Specification. Other values of attr\_coding\_type are reserved for future use by ISO/IEC. Decoders conforming to this version of this Specification shall ignore 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) | 8.3.1 |
| 1 | LoD with Predicting Transform | 8.3.3 |
| 2 | LoD with Lifting Transform | 8.3.2 |
| 3 | Raw attribute data | TBD |

**aps\_attr\_initial\_qp\_minus4** plus 4 specifies the initial value of the variable SliceQp for each slice referring to the APS. The value of aps\_attr\_initial\_qp\_minus4 shall be in the range of 0 to 47, inclusive.

**aps\_attr\_chroma\_qp\_offset** specifies an offset to be applied to the initial quantization parameter signalled by the syntax aps\_attr\_initial\_qp\_minus4.

**aps\_slice\_qp\_offset\_present\_flag** equal to 1 specifies that per-component QP offsets indicated by ash\_attr\_qp\_offset are present in the attribute data unit header. aps\_slice\_qp\_offset\_present\_flag equal to 0 specifies that no such offsets are present.

**raht\_prediction\_enabled\_flag** equal to 1 specifies the transform weight prediction from the neighbour points is enabled in the RAHT decoding process. raht\_prediction\_enabled\_flag equal to 0 specifies the transform weight prediction from the neighbour points is enabled in the RAHT decoding process.

**raht\_prediction\_threshold0** specifies the thredhold to terminate the transform weight prediction from neighbour points. The value of raht\_prediction\_threshold0 shall be in the range of 0 to 19.

**raht\_prediction\_threshold1** specifies the thredhold to skip the transform weight prediction from neighbour points. The value of raht\_prediction\_threshold1 shall be in the range of 0 to 19.

**lifting\_num\_pred\_nearest\_neighbours\_minus1** plus 1 specifies the maximum number of nearest neighbours to be used for prediction. The value of lifting\_num\_pred\_nearest\_neighbours shall be in the range of 1 to xx.

The value of NumPredNearestNeighbours is set equal to lifting\_num\_pred\_nearest\_neighbours

**lifting\_search\_range\_minus1** plus 1 specifies the search range used to determine nearest neighbours to be used for prediction and to build distance-based levels of detail.

The variable LiftingSearchRange is derived as follows:

LiftingSearchRange = lifting\_search\_range\_minus1 + 1

**lifting\_neighbour\_bias\_xyz**[ k ] specifies the factor used to weight the k-th component of the ( x, y, z ) point positions in the calculation of the euclidean distance between two points as part of the nearest neighbour derivation process.

The array liftingNeighbourBiasStv, with values liftingNeighbourBiasStv[ k ], k = 0 .. 2, represents the values of lifting\_neighobur\_bias\_xyz permuted into the coded geometry axis order as follows:

liftingNeighbourBiasStv[XyzToStv[k]] = lifting\_neighbour\_bias\_xyz[k]

**lifting\_scalability\_enabled\_flag** equal to 1 specifies that the attribute decoding process allows the pruned octree decode result for the input geometry points. lifting\_scalability\_enabled\_flag equal to 0 specifies that that the attribute decoding process requires the complete octree decode result for the input geometry points. When not present, the value of lifting\_scalability\_enabled\_flag is inferred to be 0. When the value of log2\_trisoup\_node\_size is greater than 0, the value of lifting\_scalability\_enabled\_flag shall be 0. When geom\_tree\_coded\_axis\_list\_present\_flag is equal to 1, lifting\_scalability\_enabled\_flag shall be equal to 0.

**lifting\_max\_nn\_range\_minus1** plus 1specifies that maximum nearest neighbour range used to limit the distance of the point registered as neighbour. The value of lifting\_max\_nn\_range is the number of octree nodes around the point.

**lifting\_num\_detail\_levels\_minus1** specifies the number of levels of detail for the attribute coding. The value of lifting\_num\_detail\_levels\_minus1 shall be in the range of 0 to xx.

The variable LevelDetailCount specifying the number of level of detail is derived as follows:

LevelDetailCount = lifting\_num\_detail\_levels\_minus1 + 1

**lifting\_lod\_regular\_sampling\_enabled\_flag** equal to 1 specifies levels of detail are built by using a regular sampling strategy. lifting\_lod\_regular\_sampling\_enabled\_flag equal to 0 specifies that a distance-based sampling strategy is used instead.

**lifting\_sampling\_period\_minus2**[ idx ]plus 2 specifies the sampling period for the level of detail idx. The value of lifting\_sampling\_period\_minus2[ ] shall be in the range of 0 to xx.

**lifting\_sampling\_distance\_squared\_scale\_minus1**[ idx ]plus 1specifies the scaling factor for the derivation of the square of the sampling distance for the level of detail idx. The value of lifting\_sampling\_distance\_squared\_scale\_minus1[ idx ] shall be in the range of 0 to xx. When lifting\_sampling\_distance\_squared\_scale\_minus1[ idx ] is not present in the bitstream, it is inferred to be 0.

**lifting\_sampling\_distance\_squared\_offset**[ idx ]specifies the offset for the derivation of the square of the sampling distance for the level of detail idx. The value of lifting\_sampling\_distance\_squared\_offset[ idx ] shall be in the range of 0 to xx. When lifting\_sampling\_distance\_squared\_offset[ idx ] is not present in the bitstream, it is inferred to be 0.

The variable LiftingSamplingDistanceSquared[ idx ] for idx = 0 .. num\_detail\_level\_minus1 − 1, specifying the sampling distance for the level of detail idx, are derived as follows:

LiftingSamplingDistanceSquared[0] = lifting\_sampling\_distance\_squared\_scale\_minus1[0] + 1

for (idx = 1; idx < num\_detail\_level\_minus1; idx++) {

LiftingSamplingDistanceSquared[idx] =   
 (lifting\_sampling\_distance\_squared\_scale\_minus1[idx] + 1)  
 × LiftingSamplingDistanceSquared[idx − 1]  
 + lifting\_sampling\_distance\_squared\_offset[idx]

}

**lifting\_morton\_sort\_skip\_enabled\_flag** equal to 1 specifies that the sorting process based on Morton code is skipped. lifting\_morton\_sort\_skip\_enabled\_flag equal to 0 specifies that the sorting process based on Morton code is applied.

\_minus1 + 1, inclusive

**lifting\_adaptive\_prediction\_threshold** specifies the threshold to enable adaptive prediction. The value of lifting\_adaptive\_prediction\_threshold[ ] shall be in the range of 0 to xx. When not present, the value of lifting\_adaptive\_prediction\_threshold is inferred to be 0.

The variable AdaptivePredictionThreshold specifying the threshold to switch to adaptive predictor selection mode is set equal to lifting\_adaptive\_prediction\_threshold.

**lifting\_intra\_lod\_prediction\_num\_layers** specifies number of LoD layer where decoded points in the same LoD layer could be referred to generate prediction value of target point. lifting\_intra\_lod\_prediction\_num\_layers equal to LevelDetailCount indicates that target point could refer decoded points in the same LoD layer for all LoD layers. lifting\_intra\_lod\_prediction\_num\_layers equal to 0 indicates that target point could not refer decoded points in the same LoD layer for any LoD layers. lifting\_intra\_lod\_prediction\_num\_layers shall be in the range of 0 to LevelDetailCount.

The variable IntraLodPredNumLayers specifying the number of LoD layer where intra lod prediction is enabled is set equal to lifting\_intra\_lod\_prediction\_num\_layers.

**inter\_component\_prediction\_enabled\_flag** equal to 1 specifies that the primary component of a multi component attribute is used to predict the reconstructed value of non-primary components. inter\_component\_prediction\_enabled\_flag equal to 0 specifies that all attribute components are reconstructed independently.

**raw\_attr\_fixed\_with\_flag** equal to 1 indicates that the number of bits used in the coding of raw attribute values is equal to the attribute component bitdepth. raw\_attr\_fixed\_width\_flag equal to 0 indicates that raw attribute values are coded using a variable length syntax.

**aps\_extension \_flag** equal to 0 specifies that no aps\_extension\_data\_flag syntax elements are present in the APS syntax structure. aps\_extension\_flag shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for aps\_extension\_flag is reserved for future use by ISO/IEC. Decoders shall ignore all aps\_extension\_data\_flag syntax elements that follow the value 1 for aps\_extension\_flag in an APS syntax structure.

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

#### Frame boundary marker semantics

The frame boundary marker explicitly marks the end of the current frame.

#### Byte alignment semantics

**alignment\_bit\_equal\_to\_one** shall be equal to 1.

**alignment\_bit\_equal\_to\_zero** shall be equal to 0.

### Geometry data unit semantics

#### General geometry data unit semantics

The variable GeometryNodeOccupancyCnt[ depth ][ sN ][ tN ][ vN ] represents the number of child nodes present in the geometry octree node at position (sN, tN, vN) at the given depth of the octree. Undefined values of GeometryNodeOccupancyCnt are treated as 0.



The variables NodeS[ depthS ][ idx ], NodeT[ depthT ][ idx ], and NodeV[ depthV ][ idx ] represent the s, t, and v co-ordinates of the idx-th node in decoding order at the given depth. The variable NumNodesAtDepth[ depth ] represents the number of nodes to be decoded at the given depth. The variables depthS, depthT and depthV specify respectively the depth in s, t and v dimensions.

The variables NodeS, NodeT, NodeV, NumNodesAtDepth, and GeometryNodeOccupancyCnt are initialized as follows:

NodeS[0][0] = NodeT[0][0] = NodeV[0][0] = 0

NumNodesAtDepth[0] = 1

GeometryNodeOccupancyCnt[−1][0][0][0] = 8

#### Geometry data unit header semantics

**gsh\_geometry\_parameter\_set\_id** specifies the value of the gps\_geom\_parameter\_set\_id of the active GPS. The value of gsh\_geometry\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**gsh\_tile\_id** specifies the value of the tile id that is referred to by the GSH . The value of gsh\_tile\_id shall be in the range of 0 to xx, inclusive.

**gsh\_slice\_id** identifies the slice header for reference by other syntax elements. The value of gsh\_slice\_id shall be in the range of 0 to xx, inclusive.

It is a requirement of bitstream conformance that when any two geometry data units that are associated with the same frame and that have the same value of gsh\_slice\_id are present, the contents of the two geometry data units shall be the same. Decoders shall ignore (remove from the bitstream and discard) any data unit of a frame with a particular value of gsh\_slice\_id if the decoder has already decoded another data unit of the same frame with the same value of gsh\_slice\_id. [Ed. refine and move to general decoder process]

**frame\_idx** specifies the log2\_max\_frame\_idx + 1 least significant bits of a notional frame number counter. Consecutive slices with differing values of frame\_idx form parts of different output point cloud frames. Consecutive slices with identical values of frame\_idx without an intervening frame boundary marker data unit form parts of the same output point cloud frame.

**gsh\_entropy\_continuation\_flag** equal to 1 indicates that the contexts used in entropy coding of the current geometry data unit are continued from the previous geometry data unit. gsh\_entropy\_continue\_flag equal to 0 indicates that the contexts used in entropy coding of the current geometry data unit are initialized. XXX constrains

**gsh\_prev\_slice\_id** indicates the slice ID of the geometry data unit whose contexts are continued for the entropy coding of the current geometry data unit. It is a requirement of bitsream conformance that gsh\_prev\_slice\_id is equal to the value of gsh\_slice\_id of the preceeding geometry slice in bitstream order.

**gsh\_box\_log2\_scale** specifies the scaling factor of the slice bounding box origin. When not present, gsh\_box\_log2\_scale is inferred to be equal to gps\_gs\_box\_log2\_scale.

**gsh\_box\_origin\_xyz**[ k ]specifies the k-th component of the quantized ( x, y, z ) co-ordinate of the slice bounding box origin.

The array SliceOriginStv, with values SliceOriginStv[ k ], k = 0 .. 2, represents the scaled values of gsh\_box\_origin\_xyz permuted into the coded geometry axis order as follows:

SliceOriginStv[XyzToStv[k]] = gsh\_box\_origin\_xyz[k] << gsh\_box\_log2\_scale

**geom\_tree\_depth\_minus1** plus 1 specifies the number of geometry tree levels present in the data unit. When geom\_tree\_coded\_axis\_list\_present\_flag is equal to 0, the root geometry node size is a cubic volume with edge lengths equal to 1 << ( geom\_tree\_depth\_minus1 + 1 ).

**geom\_tree\_coded\_axis\_flag**[ lvl ][ k ] specifies whether the k-th axis is coded at a given depth, lvl, of the geometry tree. The values of geom\_tree\_coded\_axis\_flag are used to determine the size of the root node. When not present, the value of geom\_tree\_coded\_axis\_flag[ lvl ][ k ] is inferred to be 1.

[Ed. The order of the levels in the software is reversed. SW should be fixed so that we can directly tie the mask to the tree current tree level, or we re-define lvl to be from the bottom and decrement in loop]

The array NodeSizesLog2, with values NodeSizesLog2[ lvl ][ k ] for lvl = 0 .. geom\_tree\_depth\_minus1 + 2 and k = 0 .. 2 indicating the size of the k-th dimension of a geometry node at tree depth lvl, is derived as follows:

for (k = 0; k < 3; k++) {

NodeSizesLog2[geom\_tree\_depth\_minus1 + 1][k] = log2\_trisoup\_node\_size

NodeSizesLog2[geom\_tree\_depth\_minus1 + 2][k] = log2\_trisoup\_node\_size

}

for (lvl = geom\_tree\_depth\_minus1; lvl >= 0; lvl--)

for (k = 0; k < 3; k++)

NodeSizesLog2[lvl][k] = NodeSizesLog2[lvl + 1][k] + geom\_tree\_coded\_axis\_flag[lvl][k]

[Ed. consider re-adding RootNodeSize if that makes expressions involving NodeSizesLog2[0] more comprehensible]

It is a requirement of bitstream conformance that the values of NodeSizesLog2[ 0 ][ k ] for k = 0 .. 2 shall be less than or equal to MaxRootNodeDimLog2.

**gsh\_entropy\_stream\_cnt\_minus1** plus 1 indicates the number of entropy streams used to convey the geometry tree data. When gsh\_entropy\_stream\_cnt\_minus1 is greater than zero, each of the last gsh\_entropy\_stream\_cnt\_minus1 tree levels is contained in a separate entropy stream.

The variable GeomEntropyStreamDepth is derived as follows:

GeomEntropyStreamDepth = geom\_tree\_depth\_minus1 − gsh\_entropy\_stream\_cnt\_minus1

**gsh\_entropy\_stream\_len\_bits** specifies the number of bits used to represent each of the gsh\_entropy\_stream\_len[ i ] syntax elements.

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

**geom\_slice\_qp\_offset** specifies an offset to the base geometry quantisation parameter geom\_base\_qp. When not present, geom\_slice\_qp\_offset is inferred to be 0.

**geom\_octree\_qp\_offsets\_depth** specifies, when present, the depth of the geometry octree when geom\_node\_qp\_offset\_eq0\_flag is present in the geometry node syntax.

The array ScalingNodeSizeLog2 with values ScalingNodeSizeLog2[ cIdx ] represents the size of the cIdx-th scaled position component.

The variable GeomScalingDepth, indicating the geometry octree depth at which the value of ScalingNodeSizeLog2 is determined, is set as follows:

GeomScalingDepth = geom\_scaling\_enabled\_flag ? geom\_octree\_qp\_offsets\_depth : 0

**trisoup\_sampling\_value\_minus1** plus 1 specifies the step size for the point sampling on the triangle surface in the trisoup decoding process specified in 8.2.3.3

**num\_unique\_segments** specifies the number of segment indicators present in the data unit.

#### Geometry data unit footer semantics

**geom\_num\_points\_minus1** plus 1 specifies the number of points coded in the data unit. It is a requirement of bitstream conformance that geom\_num\_points\_minus1 + 1 is equal to the number of decoded points in the slice.

Decoders must not rely upon bitstream conformance to prevent buffer overflow.

#### Geometry slice data semantics

#### Geometry node semantics

A geometry node is a node of the geometry octree. An internal geometry node may be split into a maximum of eight child nodes after decoding the occupancy map for the current node. A leaf node represents one or more points.

The position of the geometry node at a given depth is given by the unscaled co-ordinate of its lower left corner as ( sN, tN, vN ).

The variables sNp, tNp, and vNp indicating the position of the current node's parent node at depth − 1 are derived as follows:

sNp = sN >> 1

tNp = tN >> 1

vNp = vN >> 1

The arrays NodeSizeLog2 and ChildNodeSizeLog2 are derived as follows:

for (k = 0; k < 3; k++) {

NodeSizeLog2[k] = NodeSizesLog2[depth][k]

ChildNodeSizeLog2[k] = NodeSizesLog2[depth + 1][k]

}

The variable NodeMaxDimLog2 is derived as follows:

NodeMaxDimLog2 = Max(NodeSizeLog2[0], NodeSizeLog2[1], NodeSizeLog2[2])

When depth is less than or equal to GeomScalingDepth and nodeIdx is equal to 0, the array ScalingNodeSizeLog2 and variable minScalingNodeDimLog2 are derived as follows:

for (k = 0; k < 3; k++)

ScalingNodeSizeLog2[k] = NodeSizeLog2[k]

minScalingNodeDimLog2 = Min(NodeSizeLog2[0], NodeSizeLog2[1], NodeSizeLog2[2])

The variable NeighbourPattern is derived as follows:

* For each node, the variables rN, lN, fN, bN, uN, and dN are derived as follows:

rN = GeometryNodeOccupancyCnt[depth][sN + 1][tN][vN] != 0

lN = GeometryNodeOccupancyCnt[depth][sN − 1][tN][vN] != 0

bN = GeometryNodeOccupancyCnt[depth][sN][tN + 1][vN] != 0

fN = GeometryNodeOccupancyCnt[depth][sN][tN − 1][vN] != 0

uN = GeometryNodeOccupancyCnt[depth][sN][tN][vN + 1] != 0

dN = GeometryNodeOccupancyCnt[depth][sN][tN][vN − 1] != 0

* If NeighbAvailabilityMask is not equal to 0, the following applies.

lN = ((sN + 1) & NeighbAvailabilityMask == 1 ? 0 : lN

rN = ((sN + 1) & NeighbAvailabilityMask == 0 ? 0 : rN

fN = ((tN + 1) & NeighbAvailabilityMask == 1 ? 0 : fN

bN = ((tN + 1) & NeighbAvailabilityMask == 0 ? 0 : bN

dN = ((vN + 1) & NeighbAvailabilityMask == 1 ? 0 : dN

uN = ((vN + 1) & NeighbAvailabilityMask == 0 ? 0 : uN

* If adjacent\_child\_contextualization\_enabled\_flag is equal to 1, the following applies.

lNadj = fNadj = dNadj = 0

for (sNc = sN × 2; sNc < sN × 2 + 2; sNc++){

for (tNc = tN × 2; tNc < tN × 2 + 2; tNc++){

for (vNc = vN × 2; vNc < vN × 2 + 2; vNc++) {

lNadj |= GeometryNodeOccupancyCnt[depth + 1][sN × 2 − 1][tNc][vNc]

fNadj |= GeometryNodeOccupancyCnt[depth + 1][sNc][tN × 2 − 1][vNc]

dNadj |= GeometryNodeOccupancyCnt[depth + 1][sNc][tNc][vN × 2 − 1]

}

}

}

lN &= lNadj

fN &= fNadj

dN &= dNadj

* Finally, the variable NeighbourPattern is set as follows:

NeighbourPattern = rN | (lN << 1) | (fN << 2) | (bN << 3) | (dN << 4) | (uN << 5)

**geom\_node\_qp\_offset\_eq0\_flag** equal to 1 specifies that the current node's quantization parameter is offset from the slice quantization parameter. geom\_node\_qp\_offset\_eq0\_flag equal to 0 specifies that the current node quantization parameter inherits the quantization parameter of the parent node.

**geom\_node\_qp\_offset\_sign\_flag** specifies, when present, the sign of nodeQpOffset as follows:

* If geom\_node\_qp\_offset\_sign\_flag is equal to 0, the corresponding nodeQpOffset has a negative value.
* Otherwise, geom\_node\_qp\_offset\_sign\_flag is equal to 1, the corresponding nodeQpOffset has a positive value.

**geom\_node\_qp\_offset\_abs\_minus1** plus 1 specifies, when present, the absolute difference between the current node's quantization parameter, nodeQp, and the slice quantisation parameter.

The variable nodeQpOffset is derived as follows:

if (geom\_node\_qp\_offset\_eq0\_flag)  
 nodeQpOffset = 0  
else  
 nodeQpOffset = (2 × geom\_node\_qp\_offset\_sign\_flag − 1) × (geom\_node\_qp\_offset\_abs\_minus1 + 1)

The variable NodeQp is derived as follows:

* When depth is equal to GeomScalingDepth:

NodeQp = geom\_base\_qp + geom\_slice\_qp\_offset + nodeQpOffset

* When depth is greater than GeomScalingDepth:

NodeQp = NodeQpMap[depth][nodeIdx]

Otherwise, when depth is less than GeomScalingDepth:NodeQp = Min(minScalingNodeDimLog2 × 4, geom\_base\_qp + geom\_direct\_coding\_mode\_qp\_offset)

It is a requirement of bitstream conformance that NodeQp is less than or equal to minScalingNodeDimLog2 × 4 + 3.

The variables EffectiveChildNodeSizeLog2, EffectiveDepth, EffectiveDepthS, EffectiveDepthT, and EffectiveDepthV are derived as follows:

for (k = 0; k < 3; k++)

EffectiveChildNodeSizeLog2[k] = ChildNodeSizeLog2[k] − (NodeQp) / 4

EffectiveDepth = depth + (NodeQp) / 4

EffectiveDepthS = depthS + (NodeQp) / 4

EffectiveDepthT = depthT + (NodeQp) / 4

EffectiveDepthV = depthV + (NodeQp) / 4

**occupancy\_map** is a bitmap that identifies the occupied child nodes of the current node. When present, the variable OccupancyMap is set equal to occupancy\_map.

**occupancy\_byte** specifies a bitmap that identifies the occupied child nodes of the current node. When present, the variable OccupancyMap is set equal to the output of the geometry occupancy map permutation process as specified in 6.4.2 when invoked with NeighbourPattern and occupancy\_map as inputs.

When EffectiveDepth is greater than or equal to geom\_tree\_depth\_minus1 + 1, OccupancyMap is set equal to 1.

The array GeometryNodeChildren[ i ] identifies the index of the i-th occupied child node of the current node. The variable GeometryNodeChildrenCnt identifies the number of child nodes in the array GeometryNodeChildren[ ].

The child node state information is derived from OccupancyMap as follows:

childCnt = 0

for (childIdx = 0; childIdx < 8; childIdx++) {

if (!(OccupancyMap & (1 << childIdx)))

continue

GeometryNodeChildren[childCnt++] = childIdx

}

GeometryNodeChildrenCnt = childCnt

GeometryNodeOccupancyCnt[depth][sN][tN][vN] = childCnt

The variable DirectModeFlagPresent is derived as follows:

* When all of the following conditions are true, DirectModeFlagPresent is set equal to 1:
  + - inferred\_direct\_coding\_mode\_enabled\_flag is equal to 1
    - proba\_planar[0] \* proba\_planar[1] \* proba\_planar[2] is less than or equal to  
       127 \* 127 \* geom\_planar\_mode\_th\_IDCM
    - NodeMaxDimLog2 is greater than 1
    - GeometryNodeOccupancyCnt[ depth − 1 ][ sNp ][ tNp ][ vNp ] is less than or equal to 2
    - GeometryNodeOccupancyCnt[ depth ][ sN ][ tN ][ vN ] is equal to 1
    - NeighbourPattern is equal to 0
* Otherwise, DirectModeFlagPresent is set equal to 0.

The determination of the probabilities proba\_planar[] is performed as described in 8.2.4.6.

**num\_points\_eq1\_flag**[ child ] equal to 1 indicates that the current child node contains a single point. num\_points\_eq1\_flag equal to 0 indicates that the current child node contains at least two points. When not present, the value of num\_points\_eq1\_flag is inferred equal to 1.

When unique\_point\_positions\_constraint\_flag is equal to 1, it is a requirement of bitstream conformance that num\_points\_eq1\_flag[ child ], when present, is equal to 1.

**num\_points\_minus2**[ child ] plus 2indicates the number of points represented by the current child node.

The array GeometryNodeDupPoints[ child ] identifies the number of duplicate points in each child of the current leaf node. When num\_points\_eq1\_flag is equal to 0, GeometryNodeDupPoints[ child ] is set equal to 1 + num\_points\_minus2[ child ]. Otherwise, GeometryNodeDupPoints[ child ] is set equal to 0.

**eligible\_planar\_flag**[ axisIdx ] equal to 1 indicates that the child nodes of the current node are eligible for the planar coding mode in the direction perpendicular to the axisIdx-th axis. eligible\_planar\_flag[ axisIdx ] equal 0 indicates that the child nodes of the current node are not eligible for the planar coding mode in the direction perpendicular to the axisIdx-th axis. When not present, the value of eligible\_planar\_flag[ axisIdx ] is inferred to be 0. The value of eligible\_planar\_flag[ axisIdx ] is determined as specified in 8.2.4.1.

#### Single occupancy data semantics

**single\_occupancy\_flag** equal to 1 indicates that the current node contains a single child node. single\_occupancy\_flag equal to 0 indicates the current node may contain multiple child nodes.

**occupancy\_idx**[ i ] with i = 0 .. 2 identifies index of the single occupied child of the current node in the geometry octree child node traversal order. When present or inferred, the variable OccupancyMap is determined from occupancy\_idx[ i ] with i = 0 .. 2 as described in 9.7.4.

#### Planar mode data semantics

**is\_planar\_flag**[ childIdx ][ axisIdx ] equal to 1 indicates that the current child node is planar in the direction perpendicular to the axisIdx-th axis. is\_planar\_flag[ child ][ axisIdx ] equal to 0 indicates that the current child node is not planar in the direction perpendicular to the axisIdx-th axis. When not present, the value of is\_planar\_flag[ childIdx ][ axisIdx ] is inferred to be 0.

The variable two\_planar\_flag indicates if a node is planar in at least two directions and is determined as follows

two\_planar\_flag[nodeIdx] =  
 (is\_planar\_flag[nodeIdx][0] && is\_planar\_flag[nodeIdx][1])  
 || (is\_planar\_flag[nodeIdx][0] && is\_planar\_flag[nodeIdx][2])  
 || (is\_planar\_flag[nodeIdx][1] && is\_planar\_flag[nodeIdx][2])

**plane\_position**[ childIdx ][ axisIdx ] equal 0 indicates that the position of the plane for the planar mode is the lower position relative to increasing i-th co-ordinates. plane\_position[ childIdx ][ axisIdx ] equal 1 indicates that the position of the plane for the planar mode is the higher position relative to increasing axisIdx-th co-ordinates.

#### Direct mode data semantics

**direct\_mode\_flag** equal to 1 indicates that the single child node of the current node is a leaf node and contains one or more delta point co-ordinates. direct\_mode\_flag equal to 0 indicates that the single child node of the current node is an internal octree node. When not present, the value of direct\_mode\_flag is inferred to be 0.

[Ed. This has been broken by moving it into a separate syntax structure]

When direct\_mode\_flag is equal to 0, the following applies:

nodeIdx = NumNodesAtDepth[depth + 1]

for (child = 0; child < GeometryNodeChildrenCnt; child++) {

childIdx = GeometryNodeChildren[child]

s = NodeS[depth + 1][nodeIdx] = 2 × sN + (childIdx & 4 == 4)

t = NodeT[depth + 1][nodeIdx] = 2 × tN + (childIdx & 2 == 2)

v = NodeV[depth + 1][nodeIdx] = 2 × vN + (childIdx & 1 == 1)

NodeQpMap[depth + 1][nodeIdx] = NodeQp

GeometryNodeOccupancyCnt[depth + 1][s][t][v] = 1

nodeIdx++

}

NumNodesAtDepth[depth + 1] = nodeIdx

**direct\_point\_cnt\_eq2\_flag** equal to 1 specifies that the current node contains two point\_offset values representing the residuals of two coded points. direct\_point\_cnt\_eq2\_flag equal to 0 specifies that the current node contains a single point\_offset value representing the residuals of a single point position duplicated zero or more times.

**direct\_dup\_point\_cnt\_eq0\_flag**, **direct\_dup\_point\_cnt\_eq1\_flag**, and **direct\_dup\_point\_cnt\_minus2** together specify the number of times a single point\_offset value is repeated in order to represent multiple points with the same position in the reconstructed point cloud. When direct\_dup\_point\_cnt\_eq0\_flag is not present, its value is inferred to be 1.

[Ed. This would be simplified by replacing all three elements with direct\_dup\_point\_cnt and describe as TR+EGk]

The variable DirectDupPointCnt representing the number of times a point is repeated is derived as follows:

DirectDupPointCnt = direct\_dup\_point\_cnt\_eq0\_flag ? 0

: direct\_dup\_point\_cnt\_eq1\_flag ? 1

: direct\_dup\_point\_cnt\_minus2 + 2

**point\_offset**[ i ][ k ][ j ]indicate the j-th bit of the k-th dimension of the current child node's i-th point's respective s, t, and v co-ordinates relative to the origin of the child node identified by the index GeometryNodeChildren[ 0 ].

When point\_offset[ i ][ k ][ 0 ] is not present, the value of point\_offset[ i ][ 0 ] is inferred by the plane position plane\_position[ child ][ k ][ 0 ].

The array PointOffset, with elements PointOffset[ i ][ k ] for i = 0 .. NumDirectPoints – 1 and k = 0 .. 2, is derived for each value of i as follows:

for (k = 0; k < 3; k++) {

PointOffset[i][k] = 0

for (j = 0; j < EffectiveChildNodeSizeLog2[k]; j++)

PointOffset[i][k] += point\_offset[i][k][j] << j;

}

**laser\_residual\_eq0\_flag**, **laser\_residual\_sign**, **laser\_residual\_eq1\_flag**, **laser\_residual\_eq2\_flag**, and **laser\_residual\_minus3** together specify the residual laser index value associated with a child node that uses the inferred direct coding mode when geometry\_angular\_mode\_flag is equal to 1.

#### Geometry trisoup data semantics

**segment\_indicator**[ i ]indicates for a unique edge whether the edge intersects the surface and hence contains a vertex (1) or not (0).

The variable NumTrisoupVertices, which specifies the number of vertices, is derived as follows:

NumTrisoupVertices = 0;

for (i = 0; i <= num\_unique\_segments\_minus1; i++)

NumTrisoupVertices += segment\_indicator[i]

**vertex\_position**[ i ]indicates the position of the vertex along the edge. The value of vertex\_position[ i ] shall be in the range of 0 to ( 1 << log2\_trisoup\_node\_size ) – 1, inclusive.

#### Geometry predictive tree semantics

The predictive geometry XXX

The variable PtnNodeIdx XXX

The array PtnPointCount with elements PtnPointCount[ nodeIdx ] XXX

The array PtnPresidual with elements PtnResidual[ nodeIdx ][ k ] XXX

#### Geometry predictive tree node semantics

**ptn\_point\_cnt\_gt1\_flag** and **ptn\_point\_cnt\_minus2** together specify the number of points represented by the current predictive tree node. When not present, the values of ptn\_point\_cnt\_gt1\_flag and ptn\_point\_cnt\_minus2 are both inferred to be 0. The number of points represented by the current predictive tree node is derived as follows:

PtnPointCount[nodeIdx] = 1 + ptn\_point\_cnt\_gt1\_flag + ptn\_point\_cnt\_minus2

**ptn\_child\_cnt**[ nodeIdx ] is the number of direct child nodes of the current predictive tree node present in the geometry predictive tree.

**ptn\_pred\_mode**[ nodeIdx ] is a mode used to predict the position associatied with the current node.

**ptn\_residual\_eq0\_flag**[ k ], **ptn\_residual\_sign\_flag**[ k ], **ptn\_residual\_abs\_log2**[ k ], and **ptn\_residual\_abs\_remaining**[ k ] together specify the prediction residual of the k-th geometry position component. ptn\_residual\_eq0\_flag specifies whether the residual is equal to zero. ptn\_residual\_sign\_flag equal to 1 indicates that the sign of the residual is positive. ptn\_residual\_sign\_flag equal to 0 indicates that the sign of the residual is negative. When ptn\_residual\_sign\_flag is not present, it is inferred to be 1. When

The prediction residual associated with the current tree node is derived as follows:

for (k = 0; k < 3; k++)

PtnResidual[nodeIdx][k] =

(2 × ptn\_residual\_sign\_flag – 1)

× (!ptn\_residual\_eq0\_flag[k]

+ ((1 << ptn\_residual\_abs\_log2[k]) >> 1)

+ ptn\_residual\_abs\_remaining[k])

### Attribute data unit semantics

#### General attribute data unit semantics

#### Attribute data unit header semantics

**ash\_attr\_parameter\_set\_id** specifies the value of the aps\_attr\_parameter\_set\_id of the active APS. The value of ash\_attr\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**ash\_attr\_sps\_attr\_idx** specifies the order of attribute set in the active SPS. The value of ash\_attr\_sps\_attr\_idx shall be in the range of 0 to sps\_num\_attribute\_sets in the active SPS.

**ash\_attr\_geom\_slice\_id** specifies the value of the gsh\_slice\_id of the active Geometry Slice Header.

**ash\_attr\_layer\_qp\_offset\_present\_flag** equal to 1 specifies that per-layer QP offsets are present in the current data unit. ash\_attr\_layer\_qp\_offset\_present\_flag equal to 0 specifies that such offsets are not present.

**\_offset**[ k ] an offset used to derive the initial slice QP for the k-th attribute component. used to derive not present ash\_attr\_qp\_offset[ k ]

Offset

\_minus4\_offset[0]

Offset \_offset[1]

**ash\_attr\_num\_layer\_qp\_minus1** plus 1 specifies the number of layers for which QP offsets are signalled. When ash\_attr\_num\_layer\_qp is not signalled, the value of ash\_attr\_num\_layer\_qp is inferred to be 0. The value of NumQpLayers is derived as follows:

NumQpLayers = num\_layer\_qp\_minus1 + 1

**ash\_attr\_layer\_qp\_offset**[ i ][ k ]specifies an offset used to derive the QP of the k-th attribute component for use in the i-th attribute layer scaling process. When not present, the value of ash\_attr\_layer\_qp\_offset[ i ][ k ] is inferred to be 0.

The variables SliceQpY[ i ] and SliceQpOffsetC[ i ] with i = 0 .. NumQpLayers − 1 are derived as follows:

for (i = 0; i < NumQpLayers; i++) {

SliceQpY[i] = InitialSliceQpY + ash\_attr\_layer\_qp\_offset[i][0]

SliceQpOffsetC[i] = InitialSliceQpOffsetC + ash\_attr\_layer\_qp\_offset[i][1]

}

**ash\_attr\_region\_cnt** specifies the number of spatial regions within the current slice that have a region QP offset signalled. The value of ash\_attr\_region\_cnt shall be less than or equal to 1 for bitstreams conforming to this version of this Specification.

**ash\_attr\_qp\_region\_origin\_xyz**[ i ][ k ] and **ash\_attr\_qp\_region\_size\_minus1\_xyz**[ i ][ k ] specify, when present, the i-th spatial region within the current slice where ash\_attr\_region\_qp\_offset[ i ][ k ] is applied.

ash\_attr\_qp\_region\_origin\_xyz[ i ][ k ] is the k-th component of the ( x, y, z ) origin co-ordinate of the i-th region relative to the slice origin.

ash\_attr\_qp\_region\_size\_minus1\_xyz[ i ][ k ] plus 1 is the k-th component of i-th the region width, height, and depth, respectively.

The arrays AttrRegionQpOriginStv and AttrRegionQpSizeStv, with values AttrRegionQpOriginStv[ i ][ k ] and AttrRegionQpSizeStv[ i ][ k ], for n = 0 .. ash\_attr\_region\_cnt − 1 and k = 0 .. 2, represent the region origin and size, respectively permuted into the coded geometry axis order as follows:

AttrRegionQpOriginStv[i][XyzToStv[k]] = ash\_attr\_qp\_region\_origin\_xyz[i][k]

AttrRegionQpSizeStv[i][XyzToStv[k]] = ash\_attr\_qp\_region\_size\_minus1\_xyz[i][k] + 1

It is a requirement of bitstream conformance that the following condition is true for k = 0 .. 2:

* AttrRegionQpOriginStv[ i ][ k ] + AttrRegionQpSizeStv[ i ][ k ] <= NodeSizesLog2[ 0 ][ k ]

**ash\_attr\_region\_qp\_offset**[ i ][ k ]specifies the QP offset to be applied to the k-th attribute component within the i-th region defined by AttrRegionQpOriginStv[ i ] and AttrRegionQpSizeStv[ i ]. When not present, ash\_attr\_region\_qp\_offset[ i ][ k ] is inferred to be ash\_attr\_region\_qp\_offset[ i ][ k − 1 ].

#### Attribute data unit data semantics

**all\_residual\_values\_equal\_to\_zero\_run** specifies the number of occurrence of the pattern which indicates that each residual\_values of all dimension are equal to zero.

**pred\_index**[ i ] specifies the predictor index to decode the i-th point value of the attribute. The value of pred\_index[ i ] shall be range of 0 to MaxNumPredictors.

The variable MaxPredDiff[ i ] is calculated as follows:

Let be the set of the *k*-nearest neighbours of the current point *i* and let be their decoded/reconstructed attribute values. The number of nearest neighbours, *k*, shall be range of 1 to lifting\_num\_pred\_nearest\_neighbours. The decoded/reconstructed attribute value of neighbours are derived according to the Predictive Lifting decoding process (8.3.3).

minValue = maxValue =

for (j = 0; j < k; j++) {

minValue = Min(minValue, )

maxValue = Max(maxValue, )

}

MaxPredDiff[i] = maxValue − minValue;

#### Attribute value semantics

Attribute coefficient values are signalled for each pointIdx-th point when at least one attribute component coefficient level is not equal to zero.

**coeff\_eq0\_flag**[ k ] specifies whether the k-th component of the absolute coefficient level is non-zero as follows:

* If coeff\_eq0\_flag[ k ] is equal to 1, the corresponding coefficient level is set equal to AttrDim == 1.
* Otherewise, coeff\_eq0\_flag is equal to 0, the corresponding coefficient level has a non-zero value.

**coeff\_abs\_level\_eq1\_flag**[ k ] equal to 1 indicates that residual\_values[k][i] equal to 1. residual\_values\_equql\_to\_one equal to 0 indicates that residual\_values[k][i] is larger than 2. [Ed. This isn't quite accurate, plus rename residual\_values]

**coeff\_abs\_level\_low**[ k ]describes the k-th dimension and the i-th point value of the attribute.

**coeff\_abs\_level\_remaining**[ k ] describes the k-th dimension and the i-th point remaining value of the attribute. When not present, the value of remaining\_value[k][i] is inferred to be 0.

**coeff\_sign\_flag**[ k ] specifies the sign of the k-th attribute component coefficient level. Positive coefficient levels are represented by coeff\_sign\_flag[ k ] equal to 0. Negative coefficient levels are represented by coeff\_sign\_flag[ k ] equal to 1. When coeff\_sign\_flag[ k ] is not present, it is inferred to be 0.

The coefficient level of the pointIdx-th point's k components are derived as follows:

absVal = (AttrDim == 1)  
 + !coeff\_eq0\_flag[k] + !coeff\_abs\_level\_eq1\_flag[k]  
 + coeff\_abs\_level\_low[k] + coeff\_abs\_level\_remaining[k]

CoeffLevel[k][pointIdx] = coeff\_sign\_flag[k] ? -absVal : absVal

#### Raw attribute value semantics

**raw\_attr\_component\_length**, when present, specifies the number of bytes used to represent raw\_attr\_value[ idx ][ k ].

**raw\_attr\_value**[ idx ][ k ] specifies the attribute value of the k-th component of the idx-th point in canonical decoding order. The number of bits, n, used to represent raw\_attr\_value[ idx ][ k ] is as follows:

* When raw\_attr\_fixed\_width\_flag is equal to 1, n = 8 × raw\_attr\_component\_length.
* When raw\_attr\_fixed\_width\_flag is equal to 0, and k is equal to 0, n = attBitdepthPrimary.
* When raw\_attr\_fixed\_width\_flag is equal to 0, and k is greater than 0, n = attrBitdepthSecondary.

### Attribute data unit semantics

**defattr\_seq\_parameter\_set\_id** specifies the value of the sps\_seq\_parameter\_set\_id of the active SPS. The value of defattr\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**defattr\_sps\_attr\_idx** specifies the order of attribute set in the active SPS. The value of defattr\_sps\_attr\_idx shall be in the range of 0 to sps\_num\_attribute\_sets in the active SPS.

**defattr\_geom\_slice\_id** specifies the value of the gsh\_slice\_id of the active Geometry Slice Header.

**defattr\_value**[ k ] is the n-bit value of the k-th component of the current attribute for all points in the current slice. The length n is equal to attr\_bitdepth\_primary when k is equal to 0, and attr\_bitdepth\_secondary otherwise.

# Decoding process

## General decoding process

The input to this process is a sequence of typed data unit buffers.

The output of this process is a series of decoded point cloud frames.

The decoding process is specified such that all decoders that conform to a specified profile and level will produce numerically identical decoded point cloud frames when invoking the decoding process associated with that profile for a bitstream conforming to that profile and level. Any decoding process that produces identical decoded point cloud frame to those produced by the process described herein conforms to the decoding process requirements of this Specification.

The decoding processes specified in the remainder of this clause apply to each coded picture, referred to as the current picture and denoted by the variable CurrPic.

The decoding process for the current picture takes as inputs the syntax elements and upper-case variables from 7.

The decoding process operates as follows for each slice of the current picture:

1. Point positions are decoded using the geometry data unit of the current slice as specified in 8.2.
2. Point attributes are decoded for each attribute data unit in the current slice as specified in 8.3.
3. The decoded points are offset and appended to the output point cloud frame as specified in 8.4.

## Geometry decoding process

### General geometry decoding process

The output of this process is the array PointPos of reconstructed point positions with elements PointPos[ i ][ axis ] for i = 0 .. geom\_num\_points\_minus1 inclusive, and axis ranging from 0 to 2 inclusive.

The variable PointCount is initialized to 0 when the decoding process for the current slice is invoked.

The geometry bitstream comprises a description of an octree. The decoding process for the octree is specified in clause 8.2.2.

The geometry bitstream may also comprise a description of the Trisoup. The decoding process for the Trisoup bitstream is specified in clause 8.2.3.

### Octree decoding process

#### General

#### Octree node decoding process

The inputs to this process are:

* an octree node location (depth, nodeIdx) specifying the position of the current geometry octree node
* a spatial location (sN, tN, vN) specifying the position of the current geometry octree node in the current slice.

The outputs of this process are the modified array PointPos and the updated variable PointCount.

If both EffectiveDepth is less than geom\_tree\_depth\_minus1, and direct\_mode\_flag is equal to 0, no points are output by this process. Otherwise, if either EffectiveDepth is greater than or equal to geom\_tree\_depth\_minus1, or direct\_mode\_flag is equal to 1, the remainder of this process generates one or more point positions.

The function geomScale( val, cIdx ) is defined as the invocation of the scaling process for a single octree node position component 8.2.2.3 with the position val, the component cIdx, and the variable qP set equal to NodeQp as inputs.

The spatial location of points in each occupied child is determined according to the number of duplicate points in each child and the use of direct coded positions as follows:

for (child = 0; child < GeometryNodeChildrenCnt; child++) {

childIdx = GeometryNodeChildren[child];

s = 2 × sN + (childIdx & 4) == 1;

t = 2 × tN + (childIdx & 2) == 1;

v = 2 × vN + (childIdx & 1) == 1;

for (i = 0; i < GeometryNodeDupPoints[child] + 1; i++, PointCount++) {

PointPos[PointCount][0] = geomScale(s, 0);

PointPos[PointCount][1] = geomScale(t, 1);

PointPos[PointCount][2] = geomScale(v, 2);

}

if (direct\_mode\_flag) {

for (i = 0; i <= direct\_point\_cnt\_eq2\_flag; i++, PointCount++) {

PointPos[PointCount][0] = geomScale((s << EffectiveChildNodeSizeLog2[0]) + PointOffset[i][0], 0);

PointPos[PointCount][1] = geomScale((t << EffectiveChildNodeSizeLog2[1]) + PointOffset[i][1], 1);

PointPos[PointCount][2] = geomScale((v << EffectiveChildNodeSizeLog2[2]) + PointOffset[i][2], 2);

}

for (i = 0; i < DirectDupPointCnt; i++, PointCount++) {

PointPos[PointCount][0] = geomScale((s << EffectiveChildNodeSizeLog2[0]) + PointOffset[0][0], 0);

PointPos[PointCount][1] = geomScale((t << EffectiveChildNodeSizeLog2[1]) + PointOffset[0][1], 1);

PointPos[PointCount][2] = geomScale((v << EffectiveChildNodeSizeLog2[2]) + PointOffset[0][2], 2);

}

}

}

#### Scaling process for a single octree node position component

The inputs to this process are:

* a variable val representing an unscaled position component value,
* a variable cIdx specifying the position component index,
* a variable qP specifying the quantization parameter.

The output of this process is the scaled position component value pos.

(NOTE?) When geom\_scaling\_enabled\_flag is equal to 0, the output of this process ie equal to the input value pos.

The variable scalingExpansionLog2 is set equal to qP / 4.

The variables highPart and lowPart representing concatenated parts of the unscaled position component value are derived as follows:

highPart = val >> (ScalingNodeSizeLog2[cIdx] − scalingExpansionLog2)

lowPart = val & ((1 << (ScalingNodeSizeLog2[cIdx] − scalingExpansionLog2)) − 1)

The scale factor sF is derived as follows:

sF = 4 + (qP & 3) << qP / 4

The output variable pos is derived as follows:

highPartS = highPart << ScalingNodeSizeLog2[cIdx]

lowPartS = (lowPart × sF + 2) >> 2

pos = highPartS | Min(lowPartS, (1 << ScalingNodeSizeLog2[cIdx]) − 1)

### Geometry Trisoup decoding process

This process is invoked after 8.2.2 when TrisoupNodeSize is greater than 0.

This process modifies the following:

the variable PointCount as the number of the decoded geometry points,

This process invokes the processes from 8.2.3.1 to 8.2.3.4 in sequential order.

#### Derivation process for the segment index

Outputs of the process are:

an array segStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2, for the start position of a segment

an array segEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2, for the end position of a segment

an array segVertex[ i ] with i = 0 .. NodeNum – 1 for the vertex position intersecting the segment

A variable NodeNum for the number of the trisoup node is set to PointCount × 12 – 1.

This process invokes the sub processes from 8.2.3.1.1 to 8.2.3.1.3 in sequential order.

##### Derivation process for sorted segment index

Outputs of this process are:

the array segStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2

the array segEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2

an array sortedSegIdx[ i ] with i = 0 .. NodeNum − 1 for the sorted segment index.

segStPos[ i ][ axis ] and segEdPos[ i ][ axis ] with  i = 0 .. PointCount − 1 , axis = 0..2 are derived as follows.

for (k = 0; k < 12; k++) {

segStPos[i × 12+k][axis] =  
 PointPos[i][axis] + segStOffsetTable[k][axis] × TrisoupNodeSize

segEdPos[i × 12+k][axis] =  
 PointPos[i][axis] + segEdOffsetTable[k][axis] × TrisoupNodeSize

}

The tables segStOffsetTable[ k ][ axis ] and segEdOffsetTable[ k ][ axis ] are defined in Table 13 and Table 14, respectively.

Table 13 — segStOffsetTable[ k ][ axis ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **axis** | **k** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **0** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| **1** | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Table 14 — segEdOffsetTable[ k ][ axis ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **axis** | **k** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **0** | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| **1** | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| **2** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

An array stPos1D[ i ] with i = 0 .. NodeNum − 1  is derived as follows.

stPos1D[i] = (segStPos[i][0] << 42) + (segStPos[i][1] << 21) + segStPos[i][2]

The array sortedSegIdx[ i ] is sorted based on the value of stPos1D[ i ] for  i = 0 .. NodeNum − 1.

sort(sortedSegIdx[i], stPos1D[i])

where sort( a[ ], b[ ] ) is a process to reorder the content of the 1D array a[ ] depending on the value of 1D array b[ ] in the ascending order.

##### Derivation process for unique segment index

Input to this process are:

the array segStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2,

the array segEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2,

the array sortedSegIdx[ i ] with i = 0 .. NodeNum − 1 .

Outputs of this process are:

a variable numUniqSeg for the number of unique segments,

an array uniqSegIdx[ i ] with  i = 0 .. NodeNum − 1  for the unique segment index,

an array uniqSegStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2  for the start position of an unique segment,

an array uniqSegEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2  for the end position of an unique segment,

A variable uIdx is initialized to 1, and numUniqSeg is initialized to 0.

uniqSegStPos[ 0 ][axis] and uniqSegEdPos[ 0 ][ axis ] with axis = 0 .. 2  are initialized as follows:

uniqSegStPos[0][axis] = segStPos[sortedSegIdx[0]][axis]

uniqSegEdPos[0][axis] = segEdPos[sortedSegIdx[0]][axis]

uniqSegIdx[ 0 ] is initialized to 0.

For the variable i = 1 .. NodeNum , the following applies:

If segStPos[ i ][ axis ] is not equal to uniqSegStPos[ uIdx ][ axis ] with axis = 0 .. 2  or segEdPos[ i ][ axis ] is not equal to uniqSegEdPos[ uIdx ][ axis ] with axis = 0 .. 2 , the following applies:

uniqSegStPos[ uIdx ][ axis ] and uniqSegEdPos[ uIdx ][ axis ] with axis = 0 .. 2  are derived as follows:

uniqSegStPos[uIdx][axis] = segStPos[sortedSegtIdx[i]][axis]

uniqSegEdPos[uIdx][axis] = segEdPos[sortedSegtIdx[i]][axis]

uIdx is set equal to (uIdx + 1).

uniqSegIdx[ ] is updated as follows:

uniqSegIdx[sortedSegtIdx[i]] = uIdx − 1

Finally, numUniqSeg is derived as follows,

numUniqSeg = uIdx

##### Derivation process for unique segment vertex

Inputs to the process are:

the variable numUniqSeg,

the array uniqSegIdx[ i ] with  i = 0 .. NodeNum − 1 ,

the array sortedSegIdx[ i ] with i = 0 .. NodeNum − 1 .

Output of the process is

the array segVertex[ i ] with i = 0 .. NodeNum − 1

A variable vertexCount is initialized equal to 0.

An array uniqSegVertex[ i ] with i = 0 .. numUniqSeg − 1  is derived as follows:

If the value of segment\_indicator[ i ] is not equal to 0, the following applies:

uniqSegVertex[ i ] is set equal to vertex\_position[ vertexCount ]

vertexCount += 1

Otherwise (the value of segment\_indicator[ i ] is equal to 0),

uniqSegVertex[i] is set equal to −1.

Finally, segVertex[ i ] with i = 0 .. NodeNum − 1 is derived as follows:

segVertex[i] = uniqSegVertex[uniqSegIdx[sortedSegIdx[i]]]

#### Derivation process for the reconstructed triangles

Inputs to the process are:

the array segStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2,

the array segEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2,

the array segVertex[ i ] with i = 0 .. NodeNum − 1

Outputs of the process are:

a variable numTriangles for the number of the decoded triangles,

an array recTriVertex[ tIdx ][ vertex ][ axis ] with tIdx = 0 .. numTriangles − 1, vertex = 0 .. 2, axis = 0 .. 2 for the vertex positions of the decoded triangles.

The variable numTriangles is initialized to 0.

This process invokes the processes from 8.2.3.2.1 to 8.2.3.2.3 with the variable nIdx = 0 .. PointCount − 1 as the node index.

##### Derivation process for the leaf vertex

Inputs to the process are:

the variable nIdx,

the array segVertex[ i ] with i = 0 .. NodeNum − 1,

the array segStPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2,

the array segEdPos[ i ][ axis ] with  i = 0 .. NodeNum − 1 , axis = 0 .. 2

Outputs of the process are:

a variable numVertex for the number of the leaf vertices,

an array leafVertices[ j ][ axis ] with  j = 0 .. numVertex − 1 , axis = 0 .. 2,

a variable bkWidth for the block width of the node

The following applies:

numVertex is initialized to 0.

for (k = 0; k < 12; k++){

If segVertex[ nIdx × 12+k ] is greater than 0, the following applies:

An array segDist[ axis ] with axis = 0..2  is derived as follows:

segDist[axis] = segEdPos[nIdx × 12+k][axis] − segStPos[nIdx × 12+k][axis]

A variable bkWidth is derived as follows:

bkWidth = Max(Max(segDist[0], segDist[1]), segDist[2])

A variable dist is derived as follows:

If segVertex[ nIdx × 12+k ] is equal to 0,

dist is set to 0.

Otherwise, if segVertex[ nIdx × 12+k ] is equal to (bkWidth − 1),

dist is set to (bkWidth << 8).

Otherwise (segVertex[ nIdx × 12+k ] is greater than 0 and less than (bkWidth − 1)),

dist is set to (segVertex[ nIdx × 12+k ] << 8) + 128.

leafVertices[ numVertex ][ axis ] with axis = 0 .. 2  is derives as follows:

leafVertices[numVertex][axis] = (segStPos[nIdx × 12+k][axis] << 8)

If segDist[ axis ] with axis = 0 .. 2  is greater than 0, the following applies.

leafVertices[numVertex][axis] += dist

Finally, numVertex is set equal to (numVertex +1).

}

##### Sorting process for leafVertices

Inputs to the process are:

the variable nIdx,

the variable bkWidth,

the variable numVertex,

the array leafVertices[ j ][ axis ] with  j = 0 .. numVertex − 1 , axis = 0 .. 2

Output of the process is

the sorted array leafVertices[ j ][ axis ] with  j = 0 .. numVertex − 1 , axis = 0 .. 2

This process is skipped if numVertex is less than 3.

An array centroid[ axis ] with axis = 0 .. 2 is derived as follows:

centroid[axis] = 0

for (j = 0; j < numVertex; j++)

centroid[axis] += leafVertices[j][axis]

centroid[axis] /= numVertex

An array variance[ axis ] with axis = 0 .. 2 is derived as follows:

variance[axis] = 0

for (j = 0; j < numVertex; j++)

variance[axis] += ((leafVertices[j][axis] − centroid[axis])^2) >> 8

A variable minVariance is derived as follows:

minVariance = Min(Min(variance[0], variance[1]), variance[2])

A variable mainAxis is derived as follows:

mainAxis = (minVariance == variance[0] ? 0 : (minVariance == variance[1] ? 1 : 2))

A array triSide[ j ][ axis ] with  j = 0 .. numVertex − 1 , axis = 0 .. 2 is derived as follows

triSide[j][axis] = leafVertices[j][axis] – ((PointPos[nIdx][axis]+ bkWidth/2) << 8)

An array theta[ j ] and tiebreaker[ j ] with  j = 0 .. numVertex − 1  are derived as follows:

theta[j] = iAtan2(triSide[j][mainAxis == 2 ? 1 : 2], triSide[j][mainAxis == 0 ? 1 : 0])

tiebreaker[j] = triSide[j][mainAxis]

where the function iAtan2( ) is defined in 5.9.1.

An array triSortIdx[ j ] with  j = 0 .. numVertex − 1  is derived as follows:

triSortIdx[j] = (theta[j] << 16 + tiebreaker[j]) × −1

Finally, the array leafVertices[ j ] is sorted based on the value of triSortIdx[ j ] for  j = 0. numVertex − 1.

sort(leafVertices[j], triSortIdx[j])

where sort( a[ ], b[ ] ) is a process to reorder the content of the 1D array a[ ] depending on the value of 1D array b[ ] in the ascending order.

##### Derivation process for reconstructed triangle vertex

Inputs to the process are:

the variable numVertex,

the array leafVertices[ j ][ axis ] with  j = 0 .. numVertex − 1 , axis = 0 .. 2,

the variable numTriangles

Outputs of the process are

the modified variable numTriangles

the array recTriVertex[ k ][ vertex ][ axis ] with vertex = 0 .. 2, axis = 0 .. 2 for the vertices of the k-th decoded triangles.

This process is skipped if numVertex is less than 3.

A variable triStart is derived as follows:

triStart = (numVertex − 3) × (numVertex – 2) / 2

For the variable triIndex = 0 .. (numVertex – 2) , the following applies:

An array triOrder[ axis ] with axis = 0 .. 2  is derived as follows:

triOrder[axis] = polyTriangles[triStart+triIndex][axis]

recTriVertex[ numTriangles ][ vertex ][ axis ] with vertex = 0 .. 2, axis = 0 .. 2 is derived as follows:

recTriVertex[numTriangles][vertex][axis] = leafVertices[triOrder[vertex]][axis]

numTriangles is set to (numTriangles+1).

where the value of polyTrianges[ i ][ axis ] is defined in Table 15.

Table 15 — value of polyTriangles[ i ][ axis ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **i** | | | | | | | | | | | | | | | | | | | |
| **axis** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** |
| **0** | 0 | 0 | 2 | 0 | 2 | 4 | 0 | 2 | 4 | 0 | 0 | 2 | 4 | 6 | 2 | 0 | 2 | 4 | 6 | 0 |
| **1** | 1 | 1 | 3 | 1 | 3 | 0 | 1 | 3 | 5 | 2 | 1 | 3 | 5 | 0 | 4 | 1 | 3 | 5 | 7 | 2 |
| **2** | 2 | 2 | 0 | 2 | 4 | 2 | 2 | 4 | 0 | 4 | 2 | 4 | 6 | 2 | 6 | 2 | 4 | 6 | 0 | 4 |
|  | i | | | | | | | | | | | | | | | | | | | |
| **axis** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** |
| **0** | 4 | 0 | 2 | 4 | 6 | 8 | 2 | 6 | 0 | 2 | 4 | 6 | 8 | 0 | 4 | 8 | 0 | 2 | 4 | 6 |
| **1** | 6 | 1 | 3 | 5 | 7 | 0 | 4 | 8 | 1 | 3 | 5 | 7 | 9 | 2 | 6 | 0 | 1 | 3 | 5 | 7 |
| **2** | 0 | 2 | 4 | 6 | 8 | 2 | 6 | 2 | 2 | 4 | 6 | 8 | 0 | 4 | 8 | 4 | 2 | 4 | 6 | 8 |
|  | i | | | | | | | | | | | | | | |
| **axis** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** |
| **0** | 8 | 10 | 2 | 6 | 10 | 0 | 2 | 4 | 6 | 8 | 10 | 0 | 4 | 8 | 0 |
| **1** | 9 | 0 | 4 | 8 | 2 | 1 | 3 | 5 | 7 | 9 | 11 | 2 | 6 | 10 | 4 |
| **2** | 10 | 2 | 6 | 10 | 6 | 2 | 4 | 6 | 8 | 10 | 0 | 4 | 8 | 0 | 8 |

#### Points derivation process on the triangles

Inputs to this process are:

the variable numTriangles,

the array recTriVertex[ tIdx ][ vertex ][ axis ] with tIdx = 0 .. numTriangles − 1, vertex = 0 .. 2, and axis = 0 .. 2

Outputs of the process are:

a variable numPtsOnTriangle for the number of the decoded points on the reconstructed triangles,

an array ptsOnTriangle[ k ][ axis ] with k = 0 .. numPtsOnTriangle − 1 , axis = 0 .. 2

The variable numPtsOnTriangle is initialized to 0.

A variable bbSize is set to ( 1 << geom\_max\_node\_size\_log2 ) – 1.

For the variable k = 0 .. numTriangles − 1, the following applies:

An array recTV[ vertex ][ axis ] with vertex = 0 .. 2, axis = 0 .. 2 is set to recTriVertex[ k ][ vertex ][ axis ]

The three vertices of recTV[ vertex ][ axis ] are added to ptsOnTriangle[ ][ axis ] with axis = 0 .. 2 as follows:

for (vertex = 0; vertex < 3; vertex++)

ptsOnTriangle[numPtsOnTriangle++][axis] = Clip3(recTV[vertex][axis], 0, bbSize)

For the variable rDir = 0 .. 2, g1 = 0 .. bbSize – 1 , g2 = 0 .. bbSize – 1 , and sign = 0 .. 1, the following applies:

A variable rSign is derived as follows:

rSign = sign > 0 ? 256: −256

A variable rayStart is derived as follows:

rayStart = sign > 0 ? −256 : (bbSize+1) << 8

An array rayOrigin[ axis ] with axis = 0 .. 2  is derived as follows:

rayOrigin[0] = (rDir == 0) ? rayStart : g1 << 8

rayOrigin[1] = (rDir == 1) ? rayStart : g1 << 8

rayOrigin[2] = (rDir == 2) ? rayStart : g2 << 8

An array rayVector[ axis ] with axis = 0 .. 2  is derived as follows:

rayVector[0] = (rDir == 0) ? rSign : 0

rayVector[1] = (rDir == 1) ? rSign : 0

rayVector[2] = (rDir == 2) ? rSign : 0

An array interSection[ axis ] is derived by the process in 8.2.3.3.1 with the input recTV[ vertex ][ axis ], rayOrigin[ axis ], and rayVector[ axis ] with vertex = 0 .. 2, axis = 0 .. 2 .

If all the values of interSection[ axis ] with axis = 0 .. 2  are greater than 0 and less than or equal to bbSize, the following applies:

ptsOnTriangle[ numPtsOnTriangle ][ axis ] with axis = 0 .. 2  is set equal to  interSection[ axis ]

numPtsOnTriangle is set to (numPtsOnTriangle+1)

##### Derivation process of the intersection between triangle and vector

Inputs to the process are:

three triangle vertices positions v0[ axis ], v1[ axis ], and v2[ axis ] with axis = 0 .. 2,

the start position of the vector rayOrg[ axis ] with axis = 0 .. 2,

the direction of the vector rayVec[ axis ] with axis = 0 .. 2.

Output of the process is the array interSection[ axis ] with axis = 0 .. 2.

The array interSection[ axis ] with axis = 0 .. 2 is initialized to − 1.

An array edge1[ axis ], edge2[ axis ], and rOV[ axis ] with axis = 0 .. 2 are derived as follows:

edge1[axis] = v1[axis] − v0[axis]

edge2[axis] = v2[axis] − v0[axis]

rOV[axis] = rayOrg[axis] − rayVec[axis]

An array cp1[ axis ] with axis = 0 .. 2 is derived as follows.

cp1[axis] = CrossProduct(rayVec[axis], edge2[axis])

A variable r1 is calculated as follows:

r1 = InnerProduct(edge1[axis], cp1[axis]) / 256

If r1 is equal to 0, the process ends.

Otherwise (r1 is not equal to 0), the following applies:

The variable r2 is calculated as follows:

r2 = InnerProduct(rOV[axis], cp1[axis]) / r1

If r2 is less than 0 or greater than 256, the process ends.

Otherwise (r2 is greater than or equal to 0 and r2 is less than or equal to 256), the following applies:

An array cp2[ axis ] with axis = 0 .. 2 is derived as follows:

cp2[axis] = CrossProduct(rOV[axis], edge1[axis])

A variable r3 is derived as follows:

r3 = InnerProduct(rayVec[axis], cp2[axis]) / r1

If r3 is less than 0 or (r2+r3) is greater than 256, the process ends.

Otherwise (r3 is greater than or equal to 0 and (r2+r3) is less than or equal to 256), the following applies:

A variable rScale is calculated as follows:

rScale = InnerProduct(edge2[axis], cp2[axis]) / r1

If rScale is less than or equal to 0, the process ends.

Otherwise (rScale is greater than 0), interSection[ axis ] with axis = 0 .. 2 is derived as follows:

interSection[axis] = Max(0, (rayOrg[axis]+((rayVec[axis] × rScale) >> 8) – 128) >> 8)

#### Update process of the decoded geometry points

Inputs to the process are:

the variable numPtsOnTriangle,

the array ptsOnTriangle[ k ][ axis ] with k = 0 .. numPtsOnTriangle − 1  and axis = 0 .. 2

For a variable p with p = 0 .. numPtsOnTriangle − 1 , if the values of ptsOnTriangle[ p ][ axis ] are equal to the values of ptsOnTriangle[ q ][ axis ] with q = 0 .. numPtsOnTriangle − 1 , axis = 0 .. 2 and q ! = p, the following applies:

ptsOnTriangle[q][axis] with axis = 0..2  is removed from the array.

numPtsOnTriangle−−

The process is repeated until the values of ptsOnTriangle[p][axis] with p = 0..numPtsOnTriangle − 1 , axis = 0..2  are unique from the ptsOnTriangle[q][axis] with q = 0..numPtsOnTriangle − 1 , axis = 0..2 .

Finally, the following applies:

PointCount = numPtsOnTriangle

PointPos[i][axis] with i = 0..PointCount − 1 , axis = 0..2 is modified as follows.

PointPos[i][axis] = ptsOnTriangle[i][axis]

### Planar coding mode

#### Eligiblity of a node for planar coding mode

For an axis index axisIdx in the range 0 .. 2, the value of eligible\_planar\_flag[axisIdx] for a current node is determined as follows

if (depth == GeomScalingDepth − 1)

eligible\_planar\_flag[axisIdx] = 0

else if (localDensity >= 3 × 1024)

eligible\_planar\_flag[axisIdx] = 0

else {

eligible\_planar\_flag[axisIdx] =  
 planeRate[axisIdx] >= geom\_planar\_mode\_th[probable\_order[axisIdx]]

}

The variable localDensity is an estimate of the mean number of occupied child nodes in a node. localDensity is initialized to the value localDensity = 1024\*4 when starting the geometry decoding process.

The variable planeRate[axisIdx], for axisIdx in the range 0  2, is an estimate of the probability for a node to be planar in the direction perpendicular to the axisIdx-th axis. planeRate[axisIdx] is initialized to the value planeRate[axisIdx] = 128 \* 8 when starting the geometry decoding process.

After decoding occupancy\_map or occupancy\_byte of a current node, the values of localDensity and planeRate[axisIdx] are updated by

localDensity = ((localDensity << 8) − localDensity + 1024 × GeometryNodeChildrenCnt) >> 8

if (isNodePlanar[axisIdx])

planeRate[axisIdx] = ((planeRate[axisIdx] << 8) − planeRate[axisIdx] + 256 + 128) >> 8

else

planeRate[axisIdx] = ((planeRate[axisIdx] << 8) − planeRate[axisIdx] + 128) >> 8

where isNodePlanar[axisIdx] is equal to 1 if the current node is planar in the direction perpendicular to the axisIdx-th axis, and is equal to 0 otherwise.

The three values of probable\_order[] are deduced from the ordering of the three-entry array planeRate[] as defined in Table 16.

Table 16 — Determination of the values of probable\_order[] from planeRate[]

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition** | **probable\_order[ 0 ]** | **probable\_order[ 1 ]** | **probable\_order[ 2 ]** |
| planeRate[0]≥ planeRate[1] ≥ planeRate[2] | 0 | 1 | 2 |
| planeRate[0]≥ planeRate[2] > planeRate[1] | 0 | 2 | 1 |
| planeRate[1]>planeRate[0] ≥ planeRate[2] | 1 | 0 | 2 |
| planeRate[1]> planeRate[2] > planeRate[0] | 2 | 0 | 1 |
| planeRate[2]> planeRate[0] ≥ planeRate[1] | 1 | 2 | 0 |
| planeRate[2]> planeRate[1] > planeRate[0] | 2 | 1 | 0 |

#### Buffer tracking the closest nodes along an axis

[Ed: this needs further rewording/reworking. covers definition and update process]

The arrays PlanarPrevPos, PlanarPlaneOffset, IsPlanar record information about previously decoded geometry tree nodes for use in the determination of ctxIdx for the syntax elements is\_planar\_flag and plane\_position. When either geometry\_planar\_mode\_flag is equal to 0 or planar\_buffer\_disabled\_flag is equal to 1, the arrays are not used by the decoding process.

In this process, the variable axisIdx is used to represent one of the three coded axes, the variable axisPos represents the position of a node along the axisIdx-th axis. Values of axisPos are in the range 0 .. 0x3fff.

The array IsPlanarNode, with values IsPlanarNode[ axisIdx ][ axisPos ] indicates if the most recently decoded node with an axisIdx-th position component equal to axisPos is planar in the plane perpendicular to the axisIdx-th axis.

The array PlanarPrevPos, with values PlanarPrevPos[ axisIdx ][ axisPos ][ k ] identifies the k-th component of the most recently decoded node with an axisIdx-th position component equal to axisPos. [Ed. note? for any value of axisIdx, only two of three values of k are used]

The array PlanarPlaneOffset, with values PlanarPlaneOffset[ axisIdx ][ axisPos ] indicates the value of plane\_position for the most recently decoded node with an axisIdx-th position component equal to axisPos.

At the start of each geometry tree level, each element of the arrays PlanarPrevPos and IsPlanarNode is initialized to 0.

After decoding each geometry\_planar\_mode\_data syntax structure with parameters childIdx and axisIdx, the arrays PlanarPrevPos, PlanarPlane and IsPlanar are updated as follows:

* The array childPos is set to the spatial position of the corresponding child node as follows:

childPos[0] = 2 × sN + (childIdx & 4 == 1)

childPos[1] = 2 × tN + (childIdx & 2 == 1)

childPos[2] = 2 × vN + (childIdx & 1 == 1)

* The variable axisPos representing a position along the axisIdx-th axis is derived as follows:

axisPos = childPos[axisIdx] & 0x3fff

* The array entries corresponding to the child node are updated as follows:

for (k = 0; k < 3; k++)

if (k != axisIdx)

PlanarPrevPos[axisIdx][axisPos][k] = (childPos[k] >> 1) & 0x7f

if (is\_planar\_flag[childIdx][axisPos])

PlanarPlane[axisIdx][axisPos] = plane\_position[childIdx][axisIdx]

IsPlanar[axisIdx][axisPos] = is\_planar\_flag[childIdx][axisIdx]



#### Common variables used in the determination of ctxIdx for the syntax elements is\_planar\_flag and plane\_position

The inputs to this process are:

* the variable childIdx identifying the child of the current node,
* the variable axisIdx identifying the axis normal to a plane, and
* the position (sN, tN, vN) of the current node within a geometry tree level.

The outputs of this proces are the variables axisPos. axisOffset, dist, neighOccupied.

The array childPos is set to the spatial position of the corresponding child node as follows:

childPos[0] = 2 × sN + (childIdx & 4 == 1)

childPos[1] = 2 × tN + (childIdx & 2 == 1)

childPos[2] = 2 × vN + (childIdx & 1 == 1)

The variable axisOffset identifies the position of a plane relative to the current node that intersects the corresponding child node normal to the axisIdx-th axis:

axisOffset = childPos[axisIdx] & 1

The variable axisPos indicates the position of the child node along the axisIdx-th axis:

axisPos = childPos[axisIdx] & 0x3fff

The variable dist represents the Manhattan distance between childPos and the most recently decoded [Ed. decoded is perhaps the wrong term?] child node position with the same value of axisPos along the axisIdx-th axis. It is derived as follows:

dist = 0

for (k = 0; k < 3; k++)

if (k != axisIdx) {

a = PlanarPrevPos[axisIdx][axisPos][k]

b = (childPos[axisIdx][k] >> 1) & 0x7f

dist += Abs(a - b)

}

The variable neighOccupied indicates whether there is an occupied node that both neighbours the current node along the axisIdx-th axis and is spatially adjacent to the current child node. It is derived as follows:

neighPatternBitIdx = 2 × axisIdx + axisOffset ^ (axisIdx ? 0 : 1)

neighOccupied = (NeighbourPattern >> neighPatternBitIdx) & 1

#### Determination of ctxIdx for the syntax element is\_planar\_flag

[Ed: this needs further rewording/reworking]

The inputs to this process are:

* the variable childIdx identifying the child of the current node,
* the variable axisIdx identifying the axis normal to a plane, and
* the position (sN, tN, vN) of the current node within a geometry tree level.

The output of this proces is the variable ctxIdx.

When planar\_buffer\_disabled\_flag is equal to 1, the value of ctxIdx is set equal to axisIdx and no further processing is performed. Otherwise, the remainder of this clause applies.

The variables neighOccupied and dist are determined according to the common variable determination process 8.2.4.3.

The context index ctxIdx is derived as follows:

ctxIdx = axisIdx + 3 × neighOccupied + (dist <= 1 ? 0 : 6)

#### Determination of ctxIdx for the syntax element plane\_position

[Ed: this needs further rewording/reworking]

The inputs to this process are:

* the variable childIdx identifying the child of the current node,
* the variable axisIdx identifying the axis normal to a plane, and
* the position (sN, tN, vN) of the current node within a geometry tree level.

The output of this proces is the variable ctxIdx.

When planar\_buffer\_disabled\_flag is equal to 1, the value of ctxIdx is set equal to 0 and no further processing is performed by this process. Otherwise, the remainder of this clause applies.

The variables axisOffset, neighOccupied, and dist are determined according to the common variable determination process 8.2.4.3.

The context index ctxIdx is derived as follows:

if (!IsPlanar[axisIdx][axisPos]])

ctxIdx = 0

else {

distCtxInc = (dist > 1) + (dist > 8)

ctxIdx = axisIdx + 3 × neighOccupied + 6 × distCtxInc + 18 × axisOffset + 1

}

#### Determination of planePosIdxAzimuthalS and planePosIdxAzimuthalT for the coding of the horizontal plane positions

The determination of planePosIdxAngularS for the arithmetic coding of plane\_position[ child ][ 0 ] and of planePosIdxAngularT for the arithmetic coding of plane\_position[ child ][ 1 ] is obtained as follows.

When geometry\_angular\_mode\_flag is equal to 0, the values of both planePosIdxAzimuthalS and planePosIdxAzimuthalT are set equal to planePosIdx. Otherwise, the following applies:

if (contextAzimuthalS[child] == −1)

planePosIdxAzimuthalS = planePosIdx

else

planePosIdxAzimuthalS = 36 + contextAzimuthalS[child]

if (contextAzimuthalT[child] == −1)

planePosIdxAzimuthalT = planePosIdx

else

planePosIdxAzimuthalT = 36 + contextAzimuthalT[child]

The determination of contextAngular[ child ] for the arithmetic coding of plane\_position[ child ][ 2 ] is performed as described in XREF.

#### Determination of planePosIdxAngular for the coding of the vertical plane position

The determination of planePosIdxAngular for the arithmetic coding of plane\_position[ child ][ 2 ] is obtained as follows.

When geometry\_angular\_mode\_flag is equal to 0, the value of planePosIdxAngular is set equal to planePosIdx. Otherwise, the following applies:

if (contextAngular[child] == −1)

planePosIdxAngular = planePosIdx

else

planePosIdxAngular = 36 + contextAngular[child]

The determination of contextAngular[child] for the arithmetic coding of plane\_position[ child ][ 2 ] is performed as described in section 8.2.5.3.

#### Determination of the probability proba\_planar[] of good plane position prediction

The information proba\_planar[] on the probability of good plane position prediction is used in the determination of the direct coding mode activation flag DirectModeFlagPresent. The value of proba\_planar[axisIdx], for an axis index in the range 0 .. 2, is in the range 1 .. 127 and is deduced as follows for each child node

proba\_planar[axisIdx] = 127

if (is\_planar\_flag[child][axisIdx]) {

if (axisIdx <= 1)

p = p0[planePosIdx] >> 9

else

p = p0[planePosIdxAngular] >> 9

if (plane\_position[child][axisIdx])

p = 128 – p

if (p < 1)

p = 1

if (p > 127)

p = 127

proba\_planar[axisIdx] = p

}

where p0[ planePosIdx ] (respectively p0[ planePosIdxAngular ]) is the probability, provided by the CABAC before decoding the bit plane\_position[ child ][ axisIdx ], of having a zero associated with the context. This probability p0[ planePosIdx ] (respectively p0[ planePosIdxAngular ]) is provided as a 16-bit unsigned integer in the range 0 .. 0xffff .

Note that proba\_planar[ axisIdx ] does not need to depend on the child node because the direct mode is activated if there is only one occupied child node in the current node.

### Angular coding modes (XXX planar)

#### Determination of the angular eligiblity for a node

The following process applies to a child node Child to determine the angular elibiligility angular\_eligible[ Child ] of the child node. If geometry\_angular\_mode\_flag is equal to 0, angular\_eligible[ Child ] is set to equal to 0. Otherwise, the following applies

midNodeS = 1 << (Max(1, ChildNodeSizeLog2[0]) − 1)

midNodeT = 1 << (Max(1, ChildNodeSizeLog2[1]) − 1)

sLidar = Abs(((sNchild − geomAngularOrigin[0] + midNodeS) << 8) – 128)

tLidar = Abs(((tNchild − geomAngularOrigin[1] + midNodeT) << 8) – 128)

rL1 = (sLidar + tLidar) >> 1

deltaAngleR = deltaAngle × rL1

midNodeV = 1 << (Max(1, ChildNodeSizeLog2[2]) − 1)

if (deltaAngleR <= (midNodeV << 26))

angular\_eligible[Child] = 0

else

angular\_eligible[Child] = 1

where deltaAngle is the minimum angular distance between the lasers determined by

deltaAngle = Min{ Abs(LaserAngle[i] – LaserAngle[j]) ; 0 ≤ i < j < number\_lasers },

and where (sNchild, tNchild, vNchild) specifying the position of the geometry octree child node Child in the current slice.

#### Laser index laserIndex associated with a node

The following process applies to a child node Child to determine the laser index laserIndex[ Child ] associated with the child node.

If the angular elibiligility angular\_eligible[ Child ] is equal to 0, then laserIndex[ Child ] index is set to a preset value UNKOWN\_LASER.

Otherwise, if the angular elibiligility angular\_eligible[Child] is equal to 1, the following applies as a continuation of the process described in 8.2.5.1. Firstly, the inverse rInv of the radial distance of the child node from the Lidar is determined

r2 = sLidar × sLidar + tLidar × tLidar

rInv = invSqrt(r2)

then an angle theta32 is determined for the child node.

vLidar = ((vNchild − geomAngularOrigin[2] + midNodeT) << 1) − 1

theta = vLidar × rInv

theta32 = theta >= 0 ? theta >> 15 : −((−theta) >> 15)

Finally, the angular elibility and the associated laser to the child node are determined as follows, based on the parent node Parent of the child node.

laserIndex[Child] = UNKOWN\_LASER

if (laserIndex[Parent] == UNKOWN\_LASER || deltaAngleR <= (midNodeV << (26 + 2))) {

minDelta = 1 << (18 + 7)

for (j = 0; j < number\_lasers; j++) {

delta = Abs(LaserAngle[j] − theta32)

if (delta < minDelta) {

minDelta = delta

laserIndex[Child] = j

}

}

}

#### Determination of the contexts contextAzimuthalS and contextAzimuthalT for planar coding mode

The following process applies to a child node Child to determine the two azimuthal contexts contextAzimuthalS[ child ] and contextAzimuthalT[ child ] associated with the child node.

The following applies as a continuation of the process described in 8.2.5.2.

Firstly, two angles are decuced from the child node position relative to the angular origin

sPos = sNchild − geomAngularOrigin[0]

tPos = tNchild − geomAngularOrigin[1]

phiNode = iAtan2hp(tPos + midNodeT, sPos + midNodeS);

phiNode0 = iAtan2hp(tPos, sPos);

Secondly, an azimuthal predictor is obtained from the array phiBuffer

predPhi = phiBuffer[laserIndex[Child]]

if (predPhi == 0x80000000)

predPhi = phiNode

The two azimuthal contexts are initialized as follows

contextAzimuthalS[Child] = -1

contextAzimuthalT[Child] = -1

Then, if the predictor predPhi is not equal to 0x80000000, the following applies to refine the two azimuthal contexts

Nshift = ((predPhi - phiNode) \* InvDeltaPhi[laserIndex] + 536870912) >> 30

predPhi -= DeltaPhi[laserIndex] \* Nshift

angleL = phiNode0 - predPhi

angleR = phiNode - predPhi

contextAnglePhi = (angleL >= 0 && angleR >= 0) || (angleL < 0 && angleR < 0) ? 2 : 0

angleL = Abs(angleL)

angleR = Abs(angleR)

if (angleL > angleR) {

contextAnglePhi++

int temp = angleL

angleL = angleR

angleR = temp

}

if (angleR > (angleL << 1))

contextAnglePhi += 4

if (angleR > (angleL << 2))

contextAnglePhi += 4

if (angleR > (angleL << 4))

contextAnglePhi += 4

if (Abs(sPos) <= Abs(tPos))

contextAzimuthalS = contextAnglePhi

else

contextAzimuthalT = contextAnglePhi

#### Determination of the context contextAngular for planar coding mode

The following process applies to a child node Child to determine the angular context contextAngular[ Child ] associated with the child node.

If the laser index laserIndex[Child] is equal to UNKOWN\_LASER, then contextAngular[ Child ] is set to a preset value UNKOWN\_CONTEXT. Otherwise, if the laser index laserIndex[ Child ] is not equal to UNKOWN\_LASER, the following applies as a continuation of the process described in 8.2.5.2.

Firstly, two angular differences thetaLaserDeltaBot and thetaLaserDeltaTop relative to a lower plane and an upper plane are determined.

thetaLaserDelta = LaserAngle[laserIndex[Child]] − theta32

Hr = LaserCorrection[laserIndex[Child]] × rInv;

thetaLaserDelta += Hr >= 0 ? −(Hr >> 17) : ((−Hr) >> 17)

vShift = (rInv << ChildNodeSizeLog2[2]) >> 20

thetaLaserDeltaTop = thetaLaserDelta − vShift

thetaLaserDeltaBot = thetaLaserDelta + vShift

Then, the angular context is deduced from the two angular differences.

contextAngular[Child] = thetaLaserDelta < 0

if (thetaLaserDeltaTop >= 0 || thetaLaserDeltaBot < 0)

contextAngular[Child] += 2

### Geometry predictive tree decoding processes

#### Decoding a sequence of predictive trees

The invocation of this process populates the array PointPos with the decoded point positions of one or more predictive trees.

The variable PtnNodeIdx is a counter used to iterate over parsed predictive tree nodes in a depth-first order. It is initialized to 0 at the start of the decoding process and incremented during the recusrive traversal of the tree.

The variable PointCount indicating the number of points decoded is initialized to 0. It is incremented during the recursive traversal of the tree for each point decoded.

The process to decode a single predictive tree (8.2.6.2) is repeatedly invoked until PointCount is equal to geom\_num\_points\_minus1 + 1.

#### Decoding a single predictive tree

An invocation of this process updates the array PointPos with the decoded point positions of a single predictive tree.

The decoding of a single predictive tree is initiated by invoking the recursive position decoding process (8.2.6.3) with the inputs curDepth set equal to 0, and aPos0, aPos1, and aPos2 each set equal to {0, 0, 0} [Ed. An undefined position?].

#### Recursive position decoding process

The inputs to this process are:

* the variable curDepth, indicating the distance in nodes between the current node and the root node of the current predictive tree.
* the arrays aPos0, aPos1, and aPos2, with values aPosX[ k ], k = 0 .. 2, and X = 0 .. 2. Each array contains the position associated with the X-th generation ancestor node in the depth-first tree traversal order.

An invocation of this process updates the variable PointCount indicating the number of points decoded and the array PointPos with the decoded point positions of the current node and its sub-tree.

The variable curNodeIdx indicating the index of the current node in the depth-first tree traversal order is derived from the couter PtnNodeIdx as follows:

curNodeIdx = PtnNodeIdx++

The array predPos is set equal to the output of the position prediction process (8.2.6.4) invoked with predMode set equal to ptn\_pred\_mode[ curNodeIdx ], and the ancestor positions aPos0, aPos1, and aPos2 as inputs.

It is a requirement of bitstream conformance that ptn\_pred\_mode[ curNodeIdx ] is less than or equal to curDepth.

The array nodePos represents the decoded point position associated with the current node.

for (k = 0; k < 3; k++)

nodePos[k] = predPos[k] + PtnResidual[curNodeIdx][k]

The decoded point position is output for each point represented by the current node.

for (i = 0; i < ptnPointCount[curNodeIdx]; i++, PointCount++)

for (k = 0; k < 3; k++)

PointPos[PointCount][k] = nodePos[k]

The sub-trees of the current node, with childIdx = 0 .. ptn\_child\_cnt[ curNodeIdx ], are decoded in turn by recursively invoking the current process with the input variables aPos0, aPos1, and aPos2 set equal to the current values of nodePos, aPos0, and aPos1 respectively.

#### Position prediction process

The inputs to this process are:

* the variable predMode indicating the prediction mode of current node,
* the arrays aPos0, aPos1, and aPos2, with values aPosX[ k ], k = 0 .. 2, and X = 0 .. 2. Each array contains the position associated with the X-th generation ancestor node in the depth-first tree traversal order.

The output from this process is the array predPos, with values predPos[ k ], k = 0 .. 2, indicating the predicted point position associated with the nodeIdx-th tree node.

When predMode is equal to 0, the predicted point position is 0:

for (k = 0; k < 3; k++)

predPos[k] = 0

When predMode is equal to 1, the predicted point position is the position associated with the first ancestor.

for (k = 0; k < 3; k++)

predPos[k] = aPos0[k]

When predMode is equal to 2, the predicted point position is a linear combination of the positions associated with the first two ancestors

for (k = 0; k < 3; k++)

predPos[k] = aPos0[k] + aPos0[k] − aPos1[k]

Otherwise, predMode is equal to 3, the predicted point position is a linear combination of the positions associated with all three ancestors

for (k = 0; k < 3; k++)

predPos[k] = aPos0[k] + aPos[k] − aPos2[k]

## Attribute decoding

Inputs to this process are:

the attribute parameter set and the associated bitstream,

Output of the process is a series of the decoded point PointAttr[ i ][ cIdx ], where i is in the range of 0 to PointCount − 1 and cIdx is in the range of 0 to AttrDim− 1.

The attributes may have multiple components.

Each attribute component has been transform coded by a spatial transform, quantized, and entropy coded, to produce its bitstream. The attribute decoder must invert this process for each attribute component, to produce a decoded attribute component.

The array initAttr, with elements initAttr[ cIdx ], cIdx = 0 .. AttrDim− 1, represents the initial attribute value of the cIdx-th attribute component. It is derived as follows:

* If the current attribute data unit is a defaulted attribute data unit, defAttr[ cIdx ] is set equal to defattr\_value[ cIdx ].
* Otherwise, if attr\_default\_value is present for the current attribute, defAttr[ cIdx ] is set equal to attr\_default\_value[ cIdx ].
* Otherwise, the following applies: [Ed: define attrIdx or a way to access active attribute parameters]

defAttr[0] = 1 << attribute\_bitdepth\_minus1[attrIdx]

for (cIdx = 1; cIdx < AttrDim; cIdx++)

defAttr[cIdx] = 1 << attribute\_bitdepth\_secondary\_minus1[attrIdx]

The array PointAttr is initialised by setting PointAttr[ i ][ cIdx ] equal to defAttr[ cIdx ] for all elements i = 0 .. PointCount − 1.

When attr\_coding\_type is equalt to 0, RAHT decoding process in clause 8.3.1 is invoked.

Otherwise, if attr\_coding\_type is equal to 1, LoD with Predicting Transform decoding process in clause 8.3.3 is invoked.

Otherwise, if attr\_coding\_type is equal to 2, LoD with Lifting Transform decoding process in clause 8.3.2 is invoked.

Otherewise (attr\_coding\_type is equal to 3), PointAttr[ i ][ cIdx ] is set equal to raw\_attr\_value[ i ][ cIdx ], for i = 0 .. PointCount − 1, and cIdx = 0 .. AttrDim− 1. [Ed: check decoding order]

### Region adaptive hierachical transform decoding process

#### General

The output of this process is the array PointsAttr with elements PointsAttr[ i ][ cIdx ] with i = 0 .. PointCount − 1, and cIdx = 0 .. AttrDim − 1. Each element with index i of PointsAttr is associated with a position given by the array PointPos with the same index i.

The variable CoeffIdx, specifying a current position in the decoded values array, is initialized to 0.

If PointCount equal to 1, the following applies:

* The variable NumRahtLevels, specifying the number of 3D transform levels, is set equal to 1
* The array PointRegionboxDeltaQp, specifying the value of delta QP per point based on region, are derived according to the RAHT region-wise qp derivation process (8.3.1.3).
* The scaling process for RAHT coefficients (8.3.1.6) is invoked for each component cIdx in the range 0 .. AttrDim − 1, with the single-element coeff set equal to value[ cIdx ][ CoeffIdx ], the position ( sTn, tTn, vTn ) set equal to ( 0, 0, 0 ), the 3D transform level lvl set equal to 0, and the variable cIdx as inputs. The reconstructed samples of the output array PointAttr[ 0 ][ cIdx ] is set equal to the single-element output array of scaled transform coefficients d.

Otherwise, the following applies:

The array Weights, specifying transform coefficient weights, and the variable NumRahtLevels, specifying the number of 3D transform levels, are derived according to the RAHT weights derivation process (8.3.1.2).

The array PointRegionboxDeltaQp, specifying the value of delta QP per point based on region, are derived according to the RAHT region-wise qp derivation process (8.3.1.3).

Reconstruction proceeds level by level from the root of the transform tree to the leaves, each using the reconstruction of the previous level.

For each 3D transform level in the descending range lvl = NumRahtLevels − 1 .. 0, the following applies:

* The variable inheritDc is derived according to the transform level. For the first 3D transform level, inheritDc is set equal to 0. Otherwise, for subsequent transform levels, inheritDc is set equal to 1.
* The variable RahtPredictionEnabled is derived as follows:

RahtPredictionEnabled = inheritDc && raht\_prediction\_enabled\_flag.

* The reconstruction process for a single RAHT level is invoked with the variable lvl set equal to 3 × lvl, and inheritDc as inputs. The output is the array recon with elements recon[ s ][ t ][ v ][ cIdx ].
* The array PrevRecon, specifying DC coefficients reconstructed from a transform level for use in a subsequent level is set equal to the array recon.

The reconstructed samples of the output array PointAttr[ i ][ cIdx ] are derived as follows with i = 0 .. PointCount − 1:

* The point position variables ( sPt, tPt, vPt ) are set equal to PointPos[ i ][ j ], with j = 0 .. 2 respectively.
* If Weights[ 0 ][ sPt ][ tPt ][ vPt ] is equal to 1, the following applies:

for (cIdx = 0; cIdx < AttrDim; cIdx++)

PointAttr[i][cIdx] = DivExp2RoundHalfInf(recon[sPt][tPt][vPt], 15)

* Otherwise, Weights[ 0 ][ sPt ][ tPt ][ vPt ] is greater than 1, the following process is used to reconstruct samples PointAttr[ i + j ][ cIdx ] for j = i .. Weights[ 0 ][ sPt ][ tPt ][ vPt ] − 1:
  + - The ( AttrDim )×( 2 ) sized array xxx is initialized as follows:

for (cIdx = 0; cIdx < AttrDim; cIdx++)

xxx[cIdx][0] = recon[xPt][yPt][zPt]

* + - For each wi in the descending range Weights[ 0 ][ sPt ][ tPt ][ vPt ] − 1 .. 1, the following applies:
      * The scaling process for RAHT coefficients is invoked for each component cIdx in the range 0 .. AttrDim − 1, with the single-element coeff set equal to coeffLevel[ cIdx ][ CoeffIdx ], the 3D transform level lvl set equal to 0, and the variable cIdx as inputs. The array element xxx[ cIdx ][ 1 ] is set equal to the single-element output array of scaled transform coefficients d.
      * CoeffIdx is incremented by 1.
      * For each component cIdx in the range 0 .. AttrDim − 1, the following applies:
        + The inverse two-point transform process is invoked with the array xxx[ cIdx ][ j ] with j = 0 .. 1, and the array w equal to { wi, 1 } as inputs. The output is the two-element array r.
        + The value of xxx[ cIdx ][ 0 ] is replaced by r[ 0 ]
        + The output PointAttr[ i + wi ][ cIdx ] is derived as follows:

PointAttr[i + wi][cIdx] = DivExp2RoundHalfInf(xxx[1], 15)

* + - The ouput PointAttr[ i ][ cIdx ] for cIdx = 0 .. AttrDim − 1 is derived as follows:

PointAttr[i][cIdx] = DivExp2RoundHalfInf(xxx[0], 15)

#### RAHT weights derivation process

The outputs of this process are:

* the array Weights, with entries Weights[ lvl ][ s ][ t ][ v ] equal to the number of points represented by a coefficient at position ( s, t, v ) at the lvl'th 1D level of the RAHT transform,
* the variable NumRahtLevels indicating the number of 3D levels in the transform tree.

The elements of the array Weights are derived as follows:

for (i = 0; i < PointCount; i++) {

s = PointPos[i][0]

t = PointPos[i][1]

v = PointPos[i][2]

Weights[0][s][t][v] += 1;

}

for (lvl = 1, done = 0; !done;)

for (j = 0; j < 3; j++, lvl++)

for (i = 0; i < PointCount; i++) {

s = PointPos[i][0] >> ((lvl + 0) / 3)

t = PointPos[i][1] >> ((lvl + 1) / 3)

v = PointPos[i][2] >> ((lvl + 2) / 3)

Weights[lvl][s][t][v] += 1;

if (Weights[lvl][s][t][v] == PointCount)

done = 1;

}

The variable NumRahtLevels is set equal to lvl / 3.

#### RAHT region-wise qp derivation process

The outputs of this process are the array PointRegionboxDeltaQp, with entries PointRegionboxDeltaQp[ lvl ][ s ][ t ][ v ][ k ] equal to the value of delta QP per attribute component of each point based on region represented by a coefficient at position ( s, t, v ) at the lvl'th 1D level of the RAHT transform.

The output array PointRegionboxDeltaQp is initialize to −1. The variable RegionQpBitShift is set to equal to 4.

for (i = 0; i < PointCount; i++) { s = PointPos[i][0]

t = PointPos[i][1]

v = PointPos[i][2]

for (k = 0; k < Min(2, AttrDim); k++)

PointRegionboxDeltaQp[0][s][t][v][k] = 0

for (n = 0; n < ash\_attr\_region\_cnt; n++) {

isPointInRegion = 1

for (k = 0; k < 3; k++)

isPointInRegion &=

AttrRegionQpOriginStv[n][k] <= PointPos[i][k]

&& PointPos[i][k] < AttrRegionQpOriginStv[n][k] + AttrRegionQpSizeStv[n][k]

if (!isPointInRegion)

continue

for (k = 0; k < 3; k++)

PointRegionboxDeltaQp[0][s][t][v][k] = ash\_attr\_region\_qp\_offset[n][k] << RegionQpBitShift

}

}

for (lvl = 1, lvl <= (NumRahtLevels − 1) × 3; lvl++){

for (i = 0; i < PointCount; i++) {

s = PointPos[i][0] >> ((lvl + 0) / 3)

t = PointPos[i][1] >> ((lvl + 1) / 3)

v = PointPos[i][2] >> ((lvl + 2) / 3)

if (PointRegionboxDeltaQp[lvl][s][t][v][0] == −1){

prevS = (lvl % 3 == 0)? s + 1: s;

prevT = (lvl % 3 == 2)? t + 1: t;

prevV = (lvl % 3 == 1)? v + 1: v;

for (k = 0; k < Min(2, AttrDim); k++) {

lQp = PointRegionboxDeltaQp[lvl − 1][s][t][v][k];

rQp = PointRegionboxDeltaQp[lvl − 1][prevS][prevT][prevV][k];

if (lQp == −1)

PointRegionboxDeltaQp[lvl][s][t][v][k] = rQp;

else if (rQp == −1)

PointRegionboxDeltaQp[lvl][s][t][v][k] = lQp;

else

PointRegionboxDeltaQp[lvl][s][t][v][k] = ((lQp + rQp) >> 1);

}

}

}

}

#### Reconstruction process for a single 3D RAHT level

The inputs to this process are:

* a variable lvl indicating the current 1D transform level.
* a variable inheritDc indicating if DC coefficients should be inherited from a previous reconstruction level.

The outputs of this process are the array recon of reconstructed values and an updated variable CoeffIdx.

An array, nodes, of occupied transform tree nodes in the current level with elements nodes[ idx ][ dim ] is derived using a Morton order traversal of the array Weights as follows:

for (mIdx = 0, nIdx = 0; mIdx < (1 << (3 × NumRahtLevels − 3 − lvl)); mIdx++) {

(sN, tN, vN) = MortonToTuple(mIdx)

if (Weights[lvl + 3][sN][tN][vN] == 0)

continue

nodesS[nIdx] = 2 × sN

nodesT[nIdx] = 2 × tN

nodesV[nIdx] = 2 × vN

nIdx++

}

The variable numNodesInLvl is set equal to nIdx.

For each occupied transform tree node with nIdx = 0 .. numNodesInLvl − 1, the following steps apply:

The position variables ( sTn, tTn, vTn ) indicating the location of a transform tree node are initialized with the values of nodesS[ nIdx ], nodesT[ nIdx ], and nodesV[ nIdx ] respectively.

An ( AttrDim )×( 8 ) element array of transform coefficients is derived as follows:

for (childIdx = 0; childIdx < 8; childIdx++) {

(ds, dt, dv) = MortonToTuple(childIdx)

if (inheritDc && childIdx == 0)

continue

if (Weights[lvl][sTn + ds][tTn + dt][vTn + dv] == 0)

continue

for (cIdx = 0; cIdx < AttrDim; cIdx++)

coeff[cIdx][childIdx] = CoeffLevel[cIdx][CoeffIdx]

CoeffIdx++

}

For each component of the attribute, the following ordered steps are performed:

* The reconstruction process for a 2×2×2 transform tree node is invoked with the node position ( sTn, tTn, vTn ), and the eight-element array coeff[ cIdx ][ childIdx ] with childIdx = 0 .. 7 as inputs. The output is the eight-element array r.
* The array of reconstructed values, recon, is updated as follows:

for (childIdx = 0; childIdx < 8; childIdx++) {

(ds, dt, dv) = MortonToTuple(childIdx)

recon[sTn + ds][tTn + dt][vTn + dv][cIdx] = r[childIdx]

}

#### Reconstruction process for a 2×2×2 transform tree node

The inputs to this process are:

* a position ( sTn, tTn, vTn ) and 1D level, lvl, specifying the location of a transform tree node in the RAHT transform tree,
* a variable cIdx specifying the index of an attribute component,
* an array, coeff, of packed quantized transform coefficients.

The output of this process is an eight-element array, r, of reconstructed values

The scaling process for RAHT coefficients is invoked with the eight-element array coeff, the position ( sTn, tTn, vTn ), the 3D transform level lvl set equal to lvl / 3, and the variable cIdx as inputs. The output is an eight-element array of scaled transform coefficients d.

If RahtPredictionEnabled is equal to 1, the following applies:

* The transform prediction upsampling process is invoked with the position ( sTn/2, tTn/2, vTn/2 ) and the variable lvl set equal to lvl + 3. The output is the eight-element array p of upsampled prediction values.
* The forward transform process for 2×2×2 blocks is invoked with the position ( sTn, tTn, vTn ) and level lvl of the current transform tree node, and the array p of upsampled prediction values. The output is the eight-element array q of transformed prediction values.

The scaled transform coefficients d, the transformed prediction values q, and an inherited DC value are summed to produce the transform coefficient array e as follows:

for (i = inheritDc; i < 8; i++)

e[i] = d[i] << 15

if (inheritDc) {

e[0] = DivExp2RoundHalfInf(PrevRecon[sTn / 2][tTn / 2][vTn / 2][cIdx], 13) << 13

}

for (i = 1; i < 8; i++)

e[i] += RahtPredictionEnabled ? q[i] : 0

The inverse transform process for 2×2×2 blocks is invoked with the position ( sTn, tTn, vTn ) and level lvl of the current transform tree node, and the array e of transform coefficients. The output is the eight-element array r of inverse transformed values.

#### Scaling process for RAHT coefficients

The inputs to this process are:

* an n-element array coeff of quantized coefficients
* a position ( sTn, tTn, vTn ) specifying the location of a transform tree node in the RAHT transform tree
* a variable lvl indicating the 3D transform level of the coefficients
* a variable cIdx specifying the index of an attribute component

The output is an n-element array of scaled transform coefficients d.

The variable qlayer is set equal to Min( NumQpLayers − 1, NumRahtLevels − lvl − 1).

The scaled transform coefficient d[ i ][ cIdx ] with i = 0 .. n − 1, and cIdx = 0 .. AttrDim − 1 is derived as follows:

for (i = 0, childIdx = 0; childIdx < 8 && i < n; childIdx++) {

(ds, dt, dv) = MortonToTuple(childIdx)

if (Weights[lvl][sTn + ds][tTn + dt][vTn + dv] == 0)

continue

for (k = 0; k < Min(2, AttrDim); k++)

deltaRegionQp[k] = PointRegionboxDeltaQp[lvl][sTn + ds][tTn + dt][vTn + dv][k]

>> RegionQpBitShift

qstepY = QpToQstep(SliceQpY[qlayer] + deltaRegionQp[0], 1)

qstepC = QpToQstep(SliceQpY[qlayer] + SliceQpOffsetC[qlayer] + deltaRegionQp[0] + deltaRegionQp[1], 0)

for (cIdx = 0; cIdx < AttrDim; cIdx++)

d[i][cIdx] = DivExp2RoundHalfUp(coeff[i][cIdx] ×

(!cIdx ? qstepY : qstepC), 8)

i++

}

#### Transform prediction upsampling process

The inputs to this process are:

* a position ( sTn, tTn, vTn ) and 1D level, lvl, specifying the location of a transform tree node in the RAHT transform tree, and
* a variable cIdx specifying the index of an attribute component.

The output of this process are:

* an eight-element array p of upsampled values.
* the array of NeighCount, with entries NeighCount[ lvl ][ s ][ t ][ v ] equal to the number of valid neighbour transform tree node where more than equal to one point exist represented by a coefficient at position ( s, t, v ) at the lvl'th 1D level of the RAHT transform.

NeighCount[ lvl ][ s ][ t ][ v ] is initialized as 0. For each row in Table 18, the following applies:

If lvl / 3 is not equal to NumRahtLevels − 1 and NeighCount[ lvl + 3][ sTn /  2][ tTn / 2][ vTn / 2] is less than raht\_prediction\_threshold0, for each child position childIdx in the range 0 to 7, inclusive, the following applies:

for (childIdx = 0; childIdx < 8; childIdx++)

p[childIdx] = 0

Otherwise, for each row in Table 18, the following applies:

cs = sTn + ds

ct = tTn + dt

cv = vTn + dv

if (Weights[lvl][cs][ct][cv] > 0)

NeighCount[lvl][sTn][tTn][vTn] += 1

If NeighCount[ lvl ][ sTn ][ tTn ][ vTn ] is less than raht\_prediction\_threshold1, for each child position childIdx in the range 0 to 7, inclusive, the following applies:

for (childIdx = 0; childIdx < 8; childIdx++)

p[childIdx] = 0

Otherwise, the following applies:

The upsampled 2×2×2 block located at the position ( sTn, tTn, vTn ) is derived as follows. For each row in Table 18, the following applies:

cs = sTn + ds

ct = tTn + dt

cv = vTn + dvif (Weights[lvl][cs][ct][cv] > 0) {

wt = Weights[lvl][cs][ct][cv]

wShift = wt > 4194304 ? 11 : wt > 262144 ? 9 : wt > 16384 ? 7 : wt > 1024 ? 5 : 0

neighVal = Recon[cs][ct][cv][cIdx]

value = DivExp2RoundHalfInf((neighVal >> wShift) \* (invSqrt(wt) >> 25 - wShift), 15)

for (childIdx = 0; childIdx < 8; childIdx++) {

sumDc[childIdx] += value × wn[childIdx] << 15

sumWn[childIdx] += wn[childIdx]

}

}

For each child position childIdx in the range 0 to 7, inclusive, where sumWn[ childIdx ] > 0, the upsampled output sample is determined as follows:

(ds, dt, dv) = MortonToTuple(childIdx)

pred = DivExp2RoundHalfInf(sumDc[childIdx] × wnRecip[sumWn[childIdx] − 4], 15)

pred ×= iSqrt(Weights[lvl − 3][2 × sTn + ds][2 × tTn + dt][2 × vTn + dv] << 30)

p[childIdx] = DivExp2RoundHalfInf(pred, 15)

Table 18 — Weighting matrix for determining upsampled child position weights, wn[ childIdx ], for various neighbour position offsets ( dx, dy, dz )

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Neighbour offset** | | | **wn[ childIdx ]** | | | | | | | |
| **ds** | **dt** | **dv** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 1 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| −1 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 2 |
| 0 | −1 | 0 | 2 | 2 | 0 | 0 | 2 | 2 | 0 | 0 |
| 0 | 0 | 1 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 |
| 0 | 0 | −1 | 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| −1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | −1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| −1 | −1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | −1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | −1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | −1 | −1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| −1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | −1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| −1 | 0 | −1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Table 18 — Values of wnRecip[ i + j ]

| **j** | **i** | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **0** | 8192 | 6554 | 5461 | 4681 | 4096 | 3641 | 3277 | 2979 | 2731 | 2521 | 2341 | 2185 |
| **12** | 2048 | 1928 | 1820 | 1725 | 1638 | 1560 | 1489 | 1425 | 1365 | 1311 | 1260 | 1214 |
| **24** | 1170 |  |  |  |  |  |  |  |  |  |  |  |

#### Forward transform process for 2×2×2 blocks

The inputs to this process are:

* a position ( sTn, tTn, vTn ) and level, lvl, specifying the position of a transform tree node,
* an eight-element array, p, of values to be transformed.

The output of this process is an eight-element array, q, of transformed values.

For each row of Table 19 in sequential order, the array p is modified by transforming a pair of values by invoking the forward two-point transform process 8.3.1.9 with the input array x equal to { p[ i ], p[ j ] }, and the array w equal to { wi, wj }. The output y updates the array p[ i ] = y[ 0 ], p[ j ] = y[ 1 ].

The output array q is derived as q[ s ] = p[ t ] with s = 0 .. 7 and the value of t derived from s according to Table 20.

Table 19 — Ordering of coefficients and respective weights for use in the forward and inverse (reverse order) two-point transform processes

|  |  |  |  |
| --- | --- | --- | --- |
| **i** | **j** | **wi** | **wj** |
| 0 | 1 | w[ lvl ][ sTn + 0 ][ tTn + 0 ][ vTn ] | w[ lvl ][ sTn + 0 ][ tTn + 0 ][ vTn + 1 ] |
| 2 | 3 | w[ lvl ][ sTn + 0 ][ tTn + 1 ][ vTn ] | w[ lvl ][ sTn + 0 ][ tTn + 1 ][ vTn + 1 ] |
| 4 | 5 | w[ lvl ][ sTn + 1 ][ tTn + 0 ][ vTn] | w[ lvl ][ sTn + 1 ][ tTn + 0 ][ vTn + 1 ] |
| 6 | 7 | w[ lvl ][ sTn + 1 ][ tTn + 1 ][ vTn] | w[ lvl ][ sTn + 1 ][ tTn + 1 ][ vTn + 1 ] |
| 4 | 6 | w[ lvl + 1 ][ sTn + 1 ][ tTn ][ vTn] | w[ lvl + 1 ][ sTn + 1 ][ tTn + 1 ][ vTn ] |
| 0 | 2 | w[ lvl + 1 ][ sTn + 0 ][ tTn ][ vTn] | w[ lvl + 1 ][ sTn + 0 ][ tTn + 1 ][ vTn ] |
| 0 | 4 | w[ lvl + 2 ][ sTn + 0 ][ tTn ][ vTn] | w[ lvl + 2 ][ sTn + 1 ][ tTn + 0 ][ vTn ] |

Table 20 — Indexes of transform coefficients in decoding order (s)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **s** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **t** | 0 | 4 | 6 | 2 | 7 | 5 | 3 | 1 |

#### Forward two-point transform process

The inputs to this process are:

* a two-element array, x, of values to be transformed, and
* a two-element array, w, of corresponding weights.

The output of this process is a two-element array, y, of transformed values.

This process has no effect if both elements of w are equal to zero.

The transform coefficients a and b are derived as follows:

a = iSqrt(w[0] << 30) \* invSqrt(w[0] + w[1]) >> 40

b = iSqrt(w[1] << 30) \* invSqrt(w[0] + w[1]) >> 40

The output is determined as follows:

y[0] = DivExp2RoundHalfInf(x[0] × a, 15) + DivExp2RoundHalfInf(x[1] × b, 15)

y[1] = DivExp2RoundHalfInf(x[1] × a, 15) − DivExp2RoundHalfInf(x[0] × b, 15)

#### Inverse transform process for 2×2×2 blocks

The inputs to this process are:

* a position ( sTn, tTn, vTn ) and level, lvl, specifying the position of a transform tree node, and
* an eight-element array, e, of transform coefficients.

The output of this process is an eight-element array, r, of inverse transformed values.

The output array r is initialized as r[ t ] = e[ s ] with s = 0 .. 7 and the value of t derived from s according to Table 20.

For each row of Table 19 in reverse order, the array r is modified by transforming a pair of values by invoking the inverse two-point transform process 8.3.1.11 with the input array x equal to { r[ i ], r[ j ] }, and the array w equal to { wi, wj }. The output y updates the array r[ i ] = y[ 0 ], r[ j ] = y[ 1 ].

#### Inverse two-point transform process

The inputs to this process are:

* a two-element array, x, of transform coefficient, and
* a two-element array, w, of corresponding weights.

The output of this process is a two-element array, y, of inverse transformed values.

This process has no effect if both elements of w are equal to zero.

The transform coefficients a and b are derived as follows:

a = iSqrt(w[0] << 30) \* invSqrt(w[0] + w[1]) >> 40

b = iSqrt(w[1] << 30) \* invSqrt(w[0] + w[1]) >> 40

The output is determined as follows:

y[0] = DivExp2RoundHalfInf(x[0] × a, 15) − DivExp2RoundHalfInf(x[1] × b, 15)

y[1] = DivExp2RoundHalfInf(x[1] × a, 15) + DivExp2RoundHalfInf(x[0] × b, 15)

### LoD with Lifting Transform decoding process

Inputs of this process are:

a variable minGeomNodeSizeLog2 specifing the number of octree layers that are skipped to decode.

The output of the process is

a series of the decoded attribute values attributeValues[ i ][ a ], where i is in the range of 0 to PointCount − 1, inclusive, and a in the range of 0 to AttrDim − 1, inclusive.

First a variable PointNumInSlice is set to gsh\_ num\_points in the active slice.

NOTE 1 – When lifting\_scalability\_enabled\_flag is equal to 1, PointCount may be smaller than PointNumInSlice due to minGeomNodeSizeLog2 larger than 0.

This process invokes the sub-processes in the following order.

The level of detail generation process in clause 8.3.2.1 is invoked.The output of this process are stored in indexes[ i ], neighbours[ i ][ n ], neighboursCount[ i ], neighboursDistance2[ i ][ n ], and pointCountPerLevelOfDetail[l], where i is in the range of 0 to PointCount − 1, inclusive, n in the range of 0 to NumPredNearestNeighbours − 1, inclusive, l is in the range of 0 to LevelDetailCount − 1, inclusive.

The prediction weight derivation process in 8.3.2.4 is invoked with the parameters neighbours, neighboursCount and neighboursDistance2. The output of this process is stored in predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The quantization weights derivation process in 8.3.2.5 is invoked with the parameters indexes, neighbours, neighboursCount, predictionWeights, and pointNumPerLoD. The output of this process is stored in quantizationWeights[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

The inverse quantization process in 8.3.2.6 is invoked with the parameters indexes, neighbours, neighboursCount and predictionWeights. The output of this process is stored in unquantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

The inverse lifting process in 8.3.2.7 is invoked with the parameters unquantAttributeCoefficients, quantizationWeights, predictionWeights and pointCountPerLevelOfDetail. This process updates the attribute coefficients unquantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

The reconstructed attributes values are obtained as follows.

for (i = 0; i < PointCount; i++) {

for (j = 0; j < AttrDim; j++) {

value = divExp2RoundHalfInf(unquantAttributeCoefficients[i][j], 8);

if (AttrDim == 0) {

maxAttribute = (1 << (attribute\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] + 1)) − 1

}

else {

maxAttribute = (1 << (attribute\_secondary\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] + 1)) − 1

}

attributeValues[i][j] = Clip3(value , 0, maxAttribute);

}

}

#### Level of Detail Generation

The input of the process is

a vailable minGeomNodeSizeLog2 specifing the number of octree layers that are skipped to decode.

The outputs of the process are

an array of point indexes indexes[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

a series of nearest neighbours indexes neighbours[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of nearest neighbours count neighboursCount[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array pointCountPerLevelOfDetail[l], where l is in the range of 0 to LevelDetailCount − 1, inclusive.

An array of distances sampling[ l ], where l is in the range of 0 to LevelDetailCount − 2, inclusive, is derived as followings:

if (lifting\_lod\_regular\_sampling\_enabled\_flag) {

for (lod = 0; lod < LevelDetailCount − 1; lod++)

sampling[lod] = lifting\_sampling\_period\_minus2[lod] + 2

}

else {

for (lod = 0; lod < LevelDetailCount − 1; lod++)

sampling[lod] = LiftingSamplingDistanceSquared[lod]

}

Depending on the value of lifting\_lod\_regular\_sampling\_enabled\_flag, the level of detail generation process re-organizes the points into a set of refinement levels , according to a the set of Euclidian distances (i.e., lifting\_lod\_regular\_sampling\_enabled\_flag equals 0) or sampling period (i.e., lifting\_lod\_regular\_sampling\_enabled\_flag equals 1) specified by the array sampling[ l ].

If lifting\_lod\_regular\_sampling\_enabled\_flag equals 0, the array sampling[ l ] represents squared sampling distances verifying the following condition:

sampling[l−1] < sampling[l]

If lifting\_lod\_regular\_sampling\_enabled\_flag equals 1, the array sampling[ l ] represents sampling periods verifying the following condition:

sampling[ l ] > 1.If lifting\_scalability\_enabled\_flag equals 1, the level of detail degneration process re-organizes the points into a set of refinement levels , according to octree nodes based on geometry. Depeding on the value of samplingFromLastFlag, the first point in the node (i.e., samplingFromLastFlag equals 0) or the last point in the node (i.e., samplingFromLastFlag equals 1) is sampled.

First, if lifting\_morton\_sort\_skip\_enabled\_flag equal to 0, the point sorting process based on Morton code in clause 5.9.8 is invoked. Let Order[i] be the array of point indexes sorted according to their Morton codes and McodeUnsorted the array of unsorted Morton codes.

Next, the following procedure is applied in order to compute both the level of detail reordering and the points nearest neighbours.

unprocessedPointCount = PointCount

for (i = 0; i < unprocessedPointCount; i++) {

unprocessedPointIndexes[i] = lifting\_morton\_sort\_skip\_enabled\_flag ? i : Order[i]

}

for (lod = 1; lod < LevelDetailCount; lod++)

unprocessedPointCountPerLevelOfDetail[lod] = 0;

unprocessedPointCountPerLevelOfDetail[0] = PointCount

If lifting\_scalability\_enabled\_flag is equal to 0, the following is applied.

endIndex = 0

assignedPointCount = 0

for (lod = 0; unprocessedPointCount > 0 && lod < LevelDetailCount; lod++) {

nonAssignedPointCount = 0

startIndex = assignedPointCount

if (lod == LevelDetailCount – 1) {

for (i = 0; i < unprocessedPointCount; i++)

assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]

} else {

nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[0]

for (i = 1; i < unprocessedPointCount; i++) {

foundAssignedPointWithinDistanceFlag = 0

if (lifting\_lod\_regular\_sampling\_enabled\_flag == 1) {

foundAssignedPointWithinDistanceFlag = (i % sampling[lod]) != 0

} else {

for (axis = 0; axis < 3; axis++)

currentPos[axis] = PointPos[unprocessedPointIndexes[i]][axis]

k = 0

while (k++ < LiftingSearchRange) {

for (axis = 0; axis < 3; axis++)

d[axis] = currentPos[axis] – PointPos[nonAssignedPointIndexes[nonAssignedPointCount – 1]][axis]

d2 = InneProduct(d[], d[])

if (d2 <= sampling[lod]) {

foundAssignedPointWithinDistanceFlag = 1

break

}

}

}

if (foundAssignedPointWithinDistance == 1)

assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]

else

nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]

}

}

endIndex = assignedPointCount

computeNearestNeighbours(

startIndex, endIndex,

lod, assignedPointIndexes,

McodeUnsorted, nonAssignedPointCount,

nonAssignedPointIndexes)

unprocessedPointCountPerLevelOfDetail[lod+1] = nonAssignedPointCount

unprocessedPointCount = nonAssignedPointCount

unprocessedPointIndexes = nonAssignedPointIndexes //NOTE the left and the right are pointer of the array

}

Otherwise (lifting\_scalability\_enabled\_flag is equal to 1), the following is applied;

endIndex = 0

assignedPointCount = 0

for (lod = minGeomNodeSizeLog2; unprocessedPointCount > 0; lod++) {

startIndex = assignedPointCount

nonAssignedPointCount = 0

samplingFromLastFlag = lod & 1

endPointInLastOctet= -1

for (i = 0; i < unprocessedPointCount; i++) {

currVoxelIndex = McodeUnsorted[unprocessedPointIndexes[i]] >> (3×(lod+1))

voxelIndexOfNextPoint = currVoxelIndex

if(i < unprocessedPointCount - 1)

voxelIndexOfNextPoint = McodeUnsorted[unprocessedPointIndexes[i+1]] >> (3\*(lod+1))

if (i ==(unprocessedPointCount – 1) || currVoxelIndex < voxelIndexOfNextPoint){

for (axis = 0; axis < 3; axis++)

centroid[axis] = 0

for (t = endPointInLastOctet+1; t <=i; t++){

for (axis = 0; axis < 3; axis++){

centroid[axis] += PointPos[unprocessedPointIndexes[t]][axis]>>nodeSizeLog2

}

n = i - endPointInLastOctet

uppIndex = unprocessedPointIndexes[endPointInLastOctet+1]

for (minDist=0, axis = 0; axis < 3; axis++){

minDist += Abs(n\*(PointPos[uppIndex][axis]>>nodeSizeLog2) – centroid[axis])

for (t = 2+endPointInLastOctet; t <=i ; t++){

uppIndex = unprocessedPointIndexes[t]

for (d2=0, axis = 0; axis < 3; axis++){

d2 += Abs(n\*(PointPos[uppIndex][axis]>>nodeSizeLog2) – centroid[axis])

if ((samplingFromLastFlag&&(d2== MinD)) || d2 < minDist){

minDist = d2

retainedPos = t

}

}

for (t=0; t < i - endPointInLastOctet; t++){

if(t == retainedPos)

nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[t]

else

assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[t]

}

endPointInLastOctet = i

}

}

endIndex = assignedPointCount

if (startIndex != endIndex) {

numOfPointInSkipped = PointNumInSlice – PointCount

if ((endIndex – startIndex) > (startIndex + numOfPointInSkipped)){

for (loop = 0; loop < lod − minGeomNodeSizeLog2; loop++){

computeNearestNeighbours(

PointCount − unprocessedPointCountPerLevelOfDetail[loop],

PointCount − unprocessedPointCountPerLevelOfDetail[loop+1],

loop + minGeomNodeSizeLog2, assignedPointIndexes,

McodeUnsorted, nonAssignedPointCount,

nonAssignedPointIndexes)

}

}

}

computeNearestNeighbours(

startIndex, endIndex,

lod , assignedPointIndexes,

McodeUnsorted, nonAssignedPointCount,

nonAssignedPointIndexes)

unprocessedPointCountPerLevelOfDetail[lod+1] = nonAssignedPointCount

unprocessedPointCount = nonAssignedPointCount

unprocessedPointIndexes = nonAssignedPointIndexes

}

Then, the following procedure is applied:

for (i = 0; i < PointCount; i++)

indexes[PointCount− 1 – i] = assignedPointIndexes[i]

for (lod = 0; lod < LevelDetailCount; lod++)

pointCountPerLevelOfDetail[lod] = unprocessedPointCountPerLevelOfDetail[LevelDetailCount − 1 − lod]

#### Definition of computeNearestNeighbours()

Inputs of this process are:

two variables startIndex and endIndex indicating the range of points for which the nearest neighbours should be computed

a variable currentLayer specifying LoD layer number, where a series of the decoded geometry point belong

an array of point indexes assignedPointIndexes[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of Morton codes McodeUnsorted[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

a variable nonAssignedPointCount specifying the number of non-assigned points.

an array of point indexes nonAssignedPointIndexes[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

The outputs of the process are

a series of nearest neighbours indexes neighbours[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of nearest neighbours counts neighboursCount[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The variable maxNNRange is set to lifting\_max\_nn\_range\_minus1 + 1.

The nearest neighbours of the points are computing as follows.

if (nonAssignedPointCount == 0) {

for (i = startIndex; i < endIndex; i++)

neighboursCount[assignedPointIndexes[i]] = 0

} else {

j = 0

for (i = startIndex; i < endIndex; i++) {

currentIndex = assignedPointIndexes[i]

currentMortonCode = McodeUnsorted[currentIndex]

currentPos = PointPos[currentIndex]

while (j < nonAssignedPointCount &&

currentMortonCode >= McodeUnsorted[nonAssignedPointIndexes[j])

j++

}

j = Min(nonAssignedPointCount − 1, j)

j0 = Max(0, j − LiftingSearchRange)

j1 = Min(nonAssignedPointCount, j + LiftingSearchRange + 1)

neighboursCount[currentIndex] = 0

k = 0

for (k = j0; k < j1 ; k++) {

neighbourIndex = nonAssignedPointIndex[k]

neighbourPos = PointPos[neighbourIndex]

if (lifting\_scalability\_enabled\_flag){

for (axis = 0; axis < 3; axis++)

currentPos[axis] = (currentPos[axis] >> currentLayer) << currentLayer

neighbourPos[axis] = (neighbourPos[axis] >> currentLayer) << currentLayer

}

}

for (axis = 0; axis < 3; axis++)

d[axis] = liftingNeighbourBiasStv[axis]×(currentPos[axis] – neighbourPos[axis])

d2 = Abs(d[0]) + Abs(d[1]) + Abs(d[2])

if (Abs(k − j) <= 3)

insertIndex = k − j > 0 ? ((k − j) << 1) − 1 : (j − k) << 1;

else if (k > j)

insertIndex = 7 + k − j;

else

insertIndex = LiftingSearchRange + 4 + j − k;

if (neighboursCount[currentIndex] < NumPredNearestNeighbours) {

p = neighboursCount[currentIndex]

neighbours[currentIndex][p] = neighbourIndex;

neighboursDistance2[currentIndex][p] = d2

neighboursInsertIndex[currentIndex][p] = insertIndex;

neighboursCount[currentIndex]++

sortNeighbours(neighboursCount[currentIndex],

neighbours[currentIndex],

neighboursDistance2[currentIndex] ,

neighboursInsertIndex[currentIndex])

} else if (d2 < neighboursDistance2[currentIndex ][NumPredNearestNeighbours−1) {

neighbours[currentIndex ][NumPredNearestNeighbours−1 = neighbourIndex

neighboursDistance2[currentIndex ][NumPredNearestNeighbours−1 = d2

neighboursInsertIndex[currentIndex][NumPredNearestNeighbours − 1] = insertIndex

sortNeighbours(NumPredNearestNeighbours,

neighbours[currentIndex],

neighboursDistance2[currentIndex] ,

neighboursInsertIndex[currentIndex]);

} }

if (currentLayer >= LevelDetailCount − IntraLodPredNumLayers) {

j1 = Min(endIndex, k + LiftingSearchRange)

for (k = i + 1; k < j1; k++) {

neighbourIndex = assignedPointIndex[k]

neighbourPos = PointPos[neighbourIndex]

for (axis = 0; axis < 3; axis++)

d[axis] = liftingNeighbourBiasStv[axis]×(currentPos[axis] – neighbourPos[axis])

d2 = Abs(d[0]) + Abs(d[1]) + Abs(d[2])

insertIndex = 2 × LiftingSearchRange + (k − i);

if (neighboursCount[currentIndex] < NumPredNearestNeighbours) {

p = neighboursCount[currentIndex]

neighbours[currentIndex][p] = neighbourIndex

neighboursDistance2[currentIndex][p] = d2

neighboursInsertIndex[currentIndex][p] = insertIndex

neighboursCount[currentIndex]++

sortNeighbours(neighboursCount[currentIndex],

neighbours[currentIndex],

neighboursDistance2[currentIndex] ,

neighboursInsertIndex[currentIndex]);

} else if (d2 < neighboursDistance2[currentIndex][NumPredNearestNeighbours – 1]) {

neighbours[currentIndex][NumPredNearestNeighbours – 1] = neighbourIndex

neighboursDistance2[currentIndex][NumPredNearestNeighbours – 1] = d2

neighboursInsertIndex[currentIndex][NumPredNearestNeighbours − 1] = insertIndex

sortNeighbours(NumPredNearestNeighbours,

neighbours[currentIndex],

neighboursDistance2[currentIndex] ,

neighboursInsertIndex[currentIndex])

}

}

}

}

if (lifting\_scalability\_enabled\_flag) {

maxNNDistance = pow(2, currentLayer) \* pow(2, currentLayer) \* 3 \* maxNNRange

for (i = startIndex; i < endIndex; ++i) {

currentIndex = assignedPointIndexes[i]

for (j = 1; j < neighborCount[ currentIndex ] ; j++) {

if (neighboursDistance2[ currentIndex ][j] > maxNNDistance) {

neighboursCount[ currentIndex ]= j

break;

}

}

}

}

#### Definition of sortNeighbours()

Inputs of this process are:

a variable neighboursCount indicating the number of nearest neighbours for the current point. neighboursCount i is in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of nearest neighbours indexes neighbours[ n ], where n in the range of 0 to neighboursCount − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ n ], where n in the range of 0 to neighboursCount − 1, inclusive.

an array of nearest neighbours insert index neighboursInsertIndex[ n ], where n in the range of 0 to neighboursCount − 1, inclusive.

The process sortNeighbours() sorts the arrays neighbours[ n ],,],, neighboursDistance2[ n ] and neighboursInsertIndex[ n ], according to the increasing values of neighboursDistance2[ n ]. Herein, when two more than neighbours[ n ] have same neighboursDistance2[ n ], neighbours[ n ] where smaller neighboursInsertIndex[ n ] is assigned is sorted by priority.

#### Prediction weights derivation process

The inputs of this process are:

a series of nearest neighbours indexes neighbours[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of nearest neighbours counts neighboursCount[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The output is:

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The prediction weights derivation process proceeds as follows:

maxWeightValue = 1 << 8;

for (i = 0; i < PointCount; i++) {

while (neighboursCount[i] > 1 &&

neighboursDistance2[i][0] > 0 &&

(neighboursDistance2[neighbourCount[i] − 1][0] >> 8) >

neighboursDistance2[i][0]) {

neighboursCount[i]−−;

}

if (neighboursCount[i]< 2 || neighboursDistance2[i][0]== 0) {

neighboursCount[i]= 1;

predictionWeights[i][0] = maxWeightValue;

} else {

bitCount = iLog2(neighboursDistance2[i][0]) + 2;

shiftDistance = bitCount > 8 ? bitCount − 8 : 0;

biasDistance = ((1 << shift) >> 1);

if (neighboursCount[i]== 2) {

d0 = (neighboursDistance2[i][0] + biasDistance) >> shiftDistance;

d1 = (neighboursDistance2[i][1] + biasDistance) >> shiftDistance;

sum = d1 + d0;

w1 = Div(d0, sum, 8);

predictionWeights[i][0] = maxWeightValue − w1;

predictionWeights[i][1] = w1;

} else {

neighboursCount[i] = 3;

d0 = (neighboursDistance2[i][0] + biasDistance) >> shiftDistance;

d1 = (neighboursDistance2[i][1] + biasDistance) >> shiftDistance;

d2 = (neighboursDistance2[i][2] + biasDistance) >> shiftDistance;

d0d1 = d0 × d1;

d0d2 = d0 × d2;

d1d2 = d1 × d2;

sum = d1d2 + d0d1 + d0d2;

w2 = Div(d0d1, sum, 8)

w1 = Div(d0d2, sum, 8)

predictionWeights[i][0] = maxWeightValue − (w1 + w2);

predictionWeights[i][1] = w1;

predictionWeights[i][2] = w2;

}

}

}

#### Quantization weights derivation process

The inputs of this process are:

an array of point indexes indexes[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

a series of nearest neighbours indexes neighbours[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of nearest neighbours counts neighboursCount[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array of the number of the decoded points per LoD pointNumPerLoD[ k ], where k is in the range of 0 to LevelDetailCount − 1, inclusive.

The output is:

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

The quantization weights derivation procedure proceeds as follows.

If lifting\_scalability\_enabled\_flag is equal to 0, the following is applied:

for (i = 0; i < PointCount; i++)

quantizationWeights[i] = 1 << 8

for (i = PointCount − 1; i >= 0; i−−) {

index = indexes[i]

for (p = 0; p < neighboursCount[index]; p++) {

neighbour = neighbours[index][p]

quantizationWeights[neighbour] += divExp2RoundHalfInf(

predictionWeights[neighbour] ×quantizationWeights[neighbour],

8)

}

}

Otherwise (lifting\_scalability\_enabled\_flag is equal to 1), the following is applied:

index = 0

startIndex = 0

for (lodIndex = 0; lodIndex < lodCount; lodIndex++) {

for (i = 0; i < pointNumPerLoD[lodIndex]; i++)

quantizationWeights[index++] = (PointNumInSlice/pointNumPerLoD[lodIndex])) × (1 << 8)

startIndex += pointNumPerLoD[lodIndex]

}

#### Inverse quantization process

[Ed. This process seems to be missing the quantization weighting aspect]

Inputs of this process are:

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

The output of the process is

a series of the unquantized attribute coefficients unquantAttributeCoefficients[ i ][ a ], where i is in the range of 0 to PointCount − 1, inclusive, and a in the range of 0 to AttrDim − 1, inclusive.

The inverse quantization process proceeds as follows.

endIndex = pointCountPerLevelOfDetail[0]

for (i = 0, d = 0; i < PointCount; i++) {

if (i == endIndex) {

endIndex = pointCountPerLevelOfDetail[++d];

layerQpY = d < NumQpLayers ? SliceQpY[d] : SliceQpY[NumQpLayers – 1];

layerQpOffsetC = d < NumQpLayers ? SliceQpOffsetC[d] : SliceQpOffsetC[NumQpLayers – 1];

}

for (k = 0; k < Min(2, AttrDim; k++)

regionBoxDeltaQp[k] = 0;

for (n = 0; n < ash\_attr\_region\_cnt; n++) {

isPointInRegion = 1

for (k = 0; k < 3; k++)

isPointInRegion &=  
 AttrRegionQpOriginStv[n][k] <= PointPos[i][k]  
 && PointPos[i][k] < AttrRegionQpOriginStv[n][k] + AttrRegionQpSizeStv[n][k]

if (!isPointInRegion)

continue

for (k = 0; k < Min(2, AttrDim); k++)

regionBoxDeltaQp[k] = ash\_attr\_region\_qp\_offset[n][k]

}

qstepY = QpToQstep(layerQpY + regionBoxDeltaQp[0], 1);

qstepC = QpToQstep(layerQpY + layerQpOffsetC + regionBoxDeltaQp[0] + regionBoxDeltaQp[1], 0);

for (a = 0; a < AttrDim; a++)

unquantAttributeCoefficients[i][a] = coeffLevel[a][i] × (!a ? qstepY : qstepC);

}

#### Inverse lifting

Inputs of this process are:

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The process updates the attributes coefficients attributeCoefficients. It proceeds as follows.

for (lod = 1; lod < LevelDetailCount; lod++) {

startIndex = pointCountPerLevelOfDetail[lod – 1];

endIndex = pointCountPerLevelOfDetail[lod];

inverseUpdate(startIndex, endIndex, attributeCoefficients, quantizationWeights and predictionWeights);

inversePrediction(startIndex, endIndex, attributeCoefficients, and predictionWeights);

}

#### Definition of inverseUpdate()

Inputs of this process are:

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The process updates the attribute coefficients attributeCoefficients. It proceeds as follows.

for (i = 0; i < startIndex; i++) {

updateWeights[i] = 0;

for (j = 0; j < AttrDim; j++)

updates[i][j] = 0

}

for (i = 0; i < (endIndex − startIndex); i++) {

index = predictorCount − i − 1 + startIndex;

currentQuantWeight = quantizationWeights[index];

for (p = 0; p < neighboursCount[index]; p++) {

neighbourIndex = neighbours[index][p];

weight = divExp2RoundHalfInf(predictionWeights[index][p] × currentQuantWeight, 8);

updateWeights[neighbourIndex] += weight;

for (j = 0; j < AttrDim; j++)

updates[neighbourIndex][j] += weight × attributeCoefficients[index][j];

}

}

for (i = 0; i < startIndex; i++) {

if (updateWeights[i] > 0) {

for (j = 0; j < AttrDim; j++)

attributeCoefficients[index][j] −= Div(updates[i][j], updateWeights[i], 0);

}

}

#### Definition of inversePrediction()

Inputs of this process are:

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointCount − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

an array pointCountPerLevelOfDetail[l], where l is in the range of 0 to LevelDetailCount − 1, inclusive.

The process updates the attribute coefficients attributeCoefficients. It proceeds as follows.

pointCount = endIndex − startIndex;

for (i = 0; i < pointCount; i++) {

index = predictorCount − i − 1 + startIndex;

for (j = 0; j < AttrDim; j++) {

predicted = 0;

for (p = 0; p < neighboursCount[index]; p++) {

neighbourIndex = neighbours[index][p];

predicted += predictionWeights[index][p] × attributeCoefficients[neighbourIndex][j];

}

attributeCoefficients[neighbourIndex][j] += divExp2RoundHalfInf(predicted, 8);

}

}

### **L**oD with **Predicting Transform** decoding process

The output of the process is

a series of the decoded attribute values attributeValues[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

This process invokes the sub-processes in the following order.

The level of detail generation process in clause 8.3.2.1 is invoked.The output of this process are stored in indexes[ i ], neighbours[ i ][ n ], neighboursCount[ i ], neighboursDistance2[ i ][ n ], and pointCountPerLevelOfDetail[l], where i is in the range of 0 to PointCount − 1, inclusive, n in the range of 0 to NumPredNearestNeighbours − 1, inclusive, l is in the range of 0 to LevelDetailCount, inclusive.

The Prediction weight derivation process in 8.3.2.4 is invoked with the parameters neighbours, neighboursCount and neighboursDistance2. The output of this process is stored in predictionWeights[ i ][ n ], where i is in the range of 0 to PointCount − 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours − 1, inclusive.

The inverse quantization process in 8.3.2.6 is invoked with the parameters indexes, neighbours, neighboursCount and predictionWeights. The output of this process is stored in unquantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointCount − 1, inclusive, and j in the range of 0 to AttrDim − 1, inclusive.

The reconstructed attributes values are obtained as follows.

q = 0;

for (i = 0; i < PointCount; i++) {

currentIndex = indexes[i];

for (j = 0; j < AttrDim; j++) {

minPredAttribute[j] = 0;

maxPredAttribute[j] = 0;

predicted[j] = 0;

}

for (p = 0; p < neighboursCount[index]; p++) {

neighbourIndex = neighbours[index][p];

for (j = 0; j < AttrDim; j++) {

if (p == 0 || minPredAttribute[j] > attributeValues[neighbourIndex][j])

minPredAttribute[j] = attributeValues[neighbourIndex][j];

if (p == 0 || maxPredAttribute[j] < attributeValues[neighbourIndex][j])

maxPredAttribute[j] = attributeValues[neighbourIndex][j];

}

}

maxDiff = maxPredAttribute[0] – minPredAttribute[0];

for (j = 1; j < AttrDim; j++)

maxDiff = Max(maxDiff, maxPredAttribute[j] – minPredAttribute[j]);

if (maxDiff > AdaptivePredictionThreshold)

predMode = pred\_index[i];

else

predMode = 0;

if (predMode > 0) {

neighbourIndex = neighbours[index][predMode −1];

for (j = 1; j < AttrDim; j++)

predicted[j] = attributeValues[neighbourIndex][j];

} else {

for (j = 0; j < AttrDim; j++) {

for (p = 0; p < neighboursCount[index]; p++) {

neighbourIndex = neighbours[index][p];

weight = predictionWeights[index][p];

predicted[j] += weight × attributeValues[neighbourIndex][j];

}

predicted[j] = divExp2RoundHalfInf(predicted[j], 8);

}

}

for (j = 0; j < AttrDim; j++)

res[j] = divExp2RoundHalfInf(unquantAttributeCoefficients[currentIndex][j], 8);

for (j = 0; j < AttrDim; j++) {

attributeValue = predicted[j] + res[j] + (j > 0 ? res[0] : 0);

if (AttrDim == 0)

maxAttribute = (1 << (attribute\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] + 1)) − 1

else

maxAttribute = (1 << (attribute\_secondary\_bitdepth\_minus1[ash\_attr\_sps\_attr\_idx] + 1)) − 1

attributeValues[currentIndex][j] = Clip(attributeValue, 0, maxAttribute);

}

}

## Slice concatenation process

The outputs of this process are:

* the modified array RecPic with elements RecPic[ pointIdx ][ attrIdx ] representing points in the reconstructed point cloud frame, and
* the modified variable RecPicPointCount representing the number of points in the reconstructed point cloud frame.

RecPicPointCount is initialized to 0.

The points and attributes from the current slice are concatenated with the reconstructed point cloud frame as follows:

for (pointIdx = 0; pointIdx <= geom\_num\_points\_minus1; pointIdx++, RecPicPointCound++) {

for (axis = 0; axis < 3; axis++)

RecPic[RecPicPointCount][axis] = PointPos[pointIdx][axis] + SliceOriginStv[axis];

for (cIdx = 0; cIdx < NumAttributeComponents; cIdx++)

RecPic[RecPicPointCount][3 + cIdx] = pointAttr[pointIdx][cIdx];

}

# Parsing process

## General

This process is invoked when the descriptor of a syntax element in the syntx tables in 7.3 is equal to u(n), ue(v), se(v), ae(v), or de(v).

The output of this process is a syntax element value.

The array DataUnitBytes, with elements DataUnitBytes[ i ], i = 0 .. DataUnitLength − 1, represents a coded data unit as a sequence of bytes. When parsing the first syntax element of a data unit, DataUnitBytes is set equal to the byte array provided by an encapsulation format (such as Annex B) or by an external means. The function readDataUnitBit( ) provides access to the bitstream as described in 9.2.

When sps\_bypass\_stream\_enabled\_flag is equal to 1, each data unit represents a header part and one or more sequences of chunk interleaved substreams. Parsing of the geometry data unit and attribute data unit syntax structures proceeds as follows:

* At the start of parsing the data unit, the variable entropyStreamIdx is initialized to 0.
* The variable ChunkSeqLen is derived as follows:
  + - When parsing the geometry data unit syntax, if entropyStreamIdx is less than EntropyStreamCnt − 1, ChunkSeqLen is set equal to gsh\_entropy\_stream\_len[ entropyStreamIdx ].
    - Otherwise, ChunkSeqLen is set equal to DataUnitLength − ( DataUnitReadIdx >> 3 )
* The arrays AeByteStream and BypassBitStream represent streams of non-bypass arithmetic coded bins and directly coded bypass bins respectively.
* The chunk interleaved substreams parsing process (9.2) is invoked with the input variable ChunkSeqLen and the output arrays AeByteStream and BypassBitstream as follows:
  + - At the start of parsing the geometry\_data\_unit syntax structure.
    - At the start of pasing the geometry\_node\_syntax structure when nodeIdx is equal to 0, and the variable depth is greater than GeomEntropyStreamDepth.
    - At the start of parsing the attribute\_data\_unit syntax structure.
* entropyStreamIdx is incremented by 1.

When GeomEntropyStreamCnt is greater than 1, the parsing state may be memorized or restored when starting to parse the geometry\_node syntax structure (7.3.3.4) as follows:

* The parsing state memorization process (9.11) is invoked when nodeIdx is equal to 0 and depth is equal to GeomEntropyStreamDepth.
* The parsing state restoration process (9.12) is invoked when nodeIdx is equal to 0 and depth is greater than GeomEntropyStreamDepth.

The output syntax element value is parsed according to the processes corresponding to the syntax element’s descriptor and name in Table 21 and Table 22.

Table 21 — Descriptor passing process

|  |  |  |
| --- | --- | --- |
| **Descriptor** | **Process** | **Channel read method** |
| u(n) | 9.6.1 | readDataUnitBit( ) |
| ue(v) | 9.6.2 | readDataUnitBit( ) |
| s(n) | 9.6.1, 9.6.4 | readDataUnitBit( ) |
| se(v) | 9.6.2, 9.6.4 | readDataUnitBit( ) |
| oid(v) | 9.6.5 | readDataUnitBit( ) |
| ae(v) | 9.10.1 | readBin( ) |
| de(v) | 9.9.1 | readBin( ) |

Table 22 — Syntax element specific parsing processes

| **Syntax structure** | **Syntax element** | **Parsing process** |
| --- | --- | --- |
| geometry\_node( ) | geom\_node\_qp\_offset\_eq0\_flag | 9.6.1 (FL), numBins = 1 |
|  | geom\_node\_qp\_offset\_sign\_flag | 9.6.1 (FL), numBins = 1 |
|  | geom\_node\_qp\_offset\_abs\_minus1 | 9.6.2 (EGk), k = 0 |
|  | single\_occupancy\_flag | 9.6.1 (FL), numBins = 1 |
|  | occupancy\_idx[] | 9.6.1 (FL), numBins = 3 |
|  | occypancy\_map | 9.7.5 |
|  | occupancy\_byte | 9.9.1 |
|  | num\_points\_eq1\_flag[ ] | 9.6.1 (FL), numBins = 1 |
|  | num\_points\_minus2[ ] | 9.6.2 (EGk), k = 0 |
|  | is\_planar\_flag[ ][ ] | 9.6.1 (FL), numBins = 1 |
|  | plane\_position[ ][ ] | 9.6.1 (FL), numBins = 1 |
|  | direct\_mode\_flag | 9.6.1 (FL), numBins = 1 |
|  | direct\_point\_cnt\_eq2\_flag | 9.6.1 (FL), numBins = 1 |
|  | direct\_dup\_point\_cnt\_eq0\_flag | 9.6.1 (FL), numBins = 1 |
|  | direct\_dup\_point\_cnt\_eq1\_flag | 9.6.1 (FL), numBins = 1 |
|  | direct\_dup\_point\_cnt\_minus2 | 9.6.2 (EGk), k = 0 |
|  | laser\_residual\_eq0\_flag | 9.6.1 (FL), numBins = 1 |
|  | laser\_residual\_sign | 9.6.1 (FL), numBins = 1 |
|  | laser\_residual\_eq1\_flag | 9.6.1 (FL), numBins = 1 |
|  | laser\_residual\_eq2\_flag | 9.6.1 (FL), numBins = 1 |
|  | laser\_residual\_minus3 | 9.6.2 (EGk), k = 1 |
|  | point\_offset | 9.6.1 (FL), numBins = 1 |
| geometry\_trisoup\_data( ) | segment\_indicator[ ] | 9.6.1 (FL), numBins = 1 |
|  | vertex\_position[ ] | 9.6.3 (TU),  maxVal = ( 1 << trisoup\_node\_size\_log2 ) + 1 |
| attribute\_slice\_data( ) | all\_residual\_values\_equal\_to\_zero\_run | 9.6.3 (TU),  maxVal = TBD |
|  | pred\_index | 9.6.3 (TU),  maxVal = MaxNumPredictors |
| attribute\_coding( ) | coeff\_eq0\_flag | 9.6.1 (FL), numBins = 1 |
|  | coeff\_abs\_level\_eq1\_flag | 9.6.1 (FL), numBins = 1 |
|  | coeff\_abs\_level\_low | XXX |
|  | coeff\_abs\_level\_remaining | 9.6.2 (EGk), k = 0 |
|  | coeff\_sign\_flag | 9.6.1 (FL), numBins = 1 |
| dictionary\_encoded\_value( ) | dict\_lut0\_hit\_flag | 9.6.1 (FL), numBins = 1 |
| geometry\_predtree\_node( ) | ptn\_point\_cnt\_gt1\_flag | 9.6.1 (FL), numBins = 1 |
| ptn\_point\_cnt\_minus2 | 9.6.2 (EGk), k = 0 |
| ptn\_child\_cnt[ ] | 9.6.1 (FL), numBins = 2 |
| prn\_pred\_mode[ ] | 9.6.1 (FL), numBins = 2 |
| ptn\_residual\_eq0\_flag[ ] | 9.6.1 (FL), numBins = 1 |
| ptn\_residual\_sign\_flag[ ] | 9.6.1 (FL), numBins = 1 |
| ptn\_residual\_abs\_log2[ ] | 9.6.1 (FL), numBins = 5 |
| ptn\_residual\_abs\_remaining[ k ] | 9.6.1 (FL), numBins = ptn\_residual\_abs\_log2[ k ] − 1 |
|  | dict\_lut1\_hit\_flag | 9.6.1 (FL), numBins = 1 |
|  | dict\_lut0\_idx | XXXREF |
|  | dict\_lut1\_idx | 9.6.1 (FL), numBins = 4 |
|  | dict\_direct\_value | 9.6.1 (FL), numBins = 8 |

## Chunked bytestream parsing process

### General

The input to this process is the variable ChunkSeqLen representing the length in bytes of a sequence of chunks.

The output of this process are:

* The array AeByteStream consisting of bytes of an arithmetic coded data stream.
* The variable AeStreamReadIdx, representing the read position of the AeByteStream.
* The array BypassBitStream consisting of bits of a bypass data stream.
* The variable BypassStreamReadIdx, representing the read position of the BypassBitStream.

A chunked bytestream sequence consists of one or more chunks. With the exception of the last chunk in a sequence, all chunks are 256 bytes in length. The final chunk may be truncated to ChunkSeqLen % 256 bytes. Each chunk contains data from one or both of the arithmetic coded data stream and a bypass bin data stream.

The variables AeStreamReadIdx and AeBypassStreamReadIdx are both initialized to 0.

The arrays AeByteStream and BypassBitSteram are assembled according to the following syntax (9.2.2) and semantics (9.2.3).

### Syntax

#### Chunked bytestream sequence syntax

|  |  |
| --- | --- |
| ae\_chunk\_sequence( ) { | **Descriptor** |
| for( chunkOffset = 0; chunkOffset < ChunkSeqLen; chunkOffset += 256 ) |  |
| ae\_chunk( Min( 256, chunkSeqLen − chunkSeqOffset ) ) |  |
| } |  |

#### Chunked bytestream chunk syntax

|  |  |
| --- | --- |
| ae\_chunk( chunkLen ) { | **Descriptor** |
| **chunk\_num\_ae\_bytes** | u(8) |
| for( i = 0; i < num\_ae\_bytes; i++ ) |  |
| **chunk\_ae\_byte**[ i ] | u(8) |
| if( i < chunkLen − 1 ) { |  |
|  |  |
|  |  |
| } |  |
| for( j = 0, i++; i < chunkLen − 1; j++, i++ ) { |  |
| **chunk\_bypass\_byte**[ j ] | u(8) |
| } |  |
| } |  |

### Semantics

#### Chunked bytestream sequence semantics

This clause is intentionally empty.

#### Chunked bytestream chunk semantics

The contents of each chunk is concatenated to the arrays AeByteStream and BypassBitStream.

**chunk\_num\_ae\_bytes** indicates the number of chunk\_ae\_byte and chunk\_bypass\_byte elements present in a chunk. When not present, the value of chunk\_num\_ae\_bytes is inferred to be 0. It is a requirement of bitstream conformance that chunk\_num\_ae\_bytes is less than chunkLen.

**chunk\_ae\_byte**[ i ] specifies the i-th byte of the arithmetically encoded symbol sub-stream of the current chunk. Each chunk\_ae\_byte[ i ] is appended to the AeByteStream array as follows:

for (i = 0; i < chunk\_num\_ae\_bytes; i++)

AeByteStream[AeStreamLen++] = chunk\_ae\_byte[i]

**chunk\_bypass\_byte**[ j ] specifies the j-th whole byte of the bypass symbol sub-stream of the current chunk. The elements of chunk\_bypass\_byte are appended to the BypassBitStream array in reverse order as follows:

numChunkBypassBytes = Max(0, chunkLen − 2 − chunk\_num\_ae\_bytes)

for (j = numChunkBypassBytes - 1; j >= 0; j--)

for (b = 7; b >= 0; b−−)

BypassBitStream[BypassBitStreamLen++] = (chunk\_bypass\_byte[j] >> b) & 1

**chunk\_bypass\_5bits** specifies the values of five bypass bits at the end of the bypass symbol sub-stream of the current chunk. Each bit is appended to the BypassBitStream array as follows:

for (b = 4; b >= 0; b−−)

BypassBitStream[BypassBitStreamLen++] = (chunk\_bypass\_5bits >> b) & 1

**chunk\_bypass\_num\_flushed\_bits** specifies the number of bypass bits to be discarded from the end of the BypassBitStream.

BypassBitstreamLen −= chunk\_bypass\_num\_flushed\_bits

## Definition of readDataUnitBit

The inputs to this process are the current data unit byte array DataUnitBytes and the associated read position DataUnitReadIdx.

The outputs of this process are the next bit read from the data unit, and the updated data unit read position.

On the first invocation of this process for the current data unit, the variable DataUnitReadIdx is set equal to 0.

The output value bitVal is determined as follows:

byteIdx = DataUnitReadIdx >> 3

bitMask = 0x80 >> (DataUnitReadIdx & 7)

bitVal = DataUnitBytes[byteIdx] & bitMask != 0

After determining bitVal, the variable DataUnitReadIdx is incremented by one.

## Definition of readAeStreamBit

If sps\_bypass\_stream\_enabled\_flag is equal to 0, this process is equivalent to invoking readDataUnitBit (9.3).

Otherwise, sps\_bypass\_stream\_enabled\_flag equal to 1, the outputs of this process are the next bit read from the AeByteStream array, and the updated AeByteStream read position.

The output value bitVal is determined as follows:

byteIdx = AeStreamReadIdx >> 3

bitMask = 0x80 >> (AeStreamReadIdx & 7)

bitVal = AeByteStream[byteIdx] & bitMask != 0

After determining bitVal, the variable AeReadIdx is incremented by one.

## Definition of readBypassStreamBit

The outputs of this process are the next bypass bit read from the BypassBitStream array, and an updated BypassBitStream read position.

The output value bitVal is determined as follows:

bitVal = BypassBitStream[BypassBitsteramReadIdx]

After determining bitVal, the variable BypassBitStreamReadIdx is incremented by one.

## General inverse binarisation processes

### Parsing of fixed-length codes

The inputs to this process are the value numBits, indicating the number of bits that represent the syntax element, and the channel read function readBit( ).

The output from this process is an unsigned syntax element value, constructed as follows:

value = 0;

for (BinIdx = 0; BinIdx < numBits; BinIdx++)

value = (value << 1) + readBit()

### Parsing of k-th order exp-Golomb codes

The inputs to this process are the value k, indicating the order of the exp-Golomb code used to represent the syntax element, and the channel read function readBit( ).

The output from this process is an unsigned syntax element value, determined as follows:

First, a unary encoded prefix is determined as follows:

prefix = 0

for (BinIdx = 0; readBit() == 1; BinIdx++)

prefix++

Then, a suffix consisting of k + prefix bins is determined as follows

suffix = 0;

for (i = 0; i < k + prefix; i++)

suffix = (suffix << 1) + readBit();

Finally, the syntax element value is constructed as follows

value = ((1 << prefix) − 1) × k + suffix

### Parsing of truncated unary codes

The inputs to this process are the value maxVal, and the channel read function readBit( ).

The output from this process is an unsigned syntax element value, determined as follows:

value = 0

for (BinIdx = 0; value < maxVal && readBit() == 1; BinIdx++)

value++

### Mapping process for signed codes

Input to this process is an unsigned syntax element value, unsignedVal.

Output from this process is the signed syntax element value, determined as follows:

If unsignedVal is even, the outputis is set equal to unsignedVal >> 1,

Otherwise, if unsignedVal is odd, the output is set equal to (unsignedVal + 1) >> 1.

Table 23 illustrates an example of the mapping process.

Table 23 — Conversion of unsigned values for signed syntax elements (informative)

|  |  |
| --- | --- |
| **Unsigned value** | **Signed value** |
| 0 | 0 |
| 1 | −1 |
| 2 | 1 |
| 3 | −2 |
| 4 | 2 |
| 5 | −3 |
| 6 | 3 |
| ... | ... |

### Parsing of international object identifiers

#### International 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) |
| } |  |

#### International object identifier semantics

The coded representation of an international object identifier follows the distinguished encoding rules of ASN.1 as specified in Rec. ITU-T X.690 | ISO/IEC 8825-1.

**oid\_forbidden\_zero\_bit** shall be equal to 0.

**oid\_length** specifies the number of octets present in oid\_contents\_octets.

**oid\_contents\_octets**[ i ] are the contents octets of an object identifier value encoding as specified in Rec. ITU-T X.690 | ISO/IEC 8825-1.

## Bit-wise geometry octree occupancy parsing process

### General process

The parsing and inverse binarization of the arithmetically coded syntax element occupancy\_map is described in 9.7.5

The decoding of each arithmetically encoded bin in occupancy\_map involves a context selection process that makes use of a dynamic map (the array CtxMap) to select a context (9.7.7) based upon the occupancy state of neighbouring nodes, predicted occupancy values ((9.7.9) and previously decoded bins. After decoding a bin, CtxMap is updated based upon the decoded bin value (9.7.8).

At the start of decoding a geometrydata unit, CtxMap is initialized according to 9.7.2.

NOTE — While the described process updates CtxMap after decoding each bin, there is no dependency by subsequent bins on the updated value.

### Initialisation process

This process is invoked at the start of each geometry data unit.

The output from this process is the initialized array CtxMap with entries CtxMap[ i ] for i in the range 0 to 1499 × 3 set equal to 127.

### Determination of planar masks used in the inverse binarization process

[Ed. XXX this process seems to be missing the interaction with qtbt (now geom\_tree\_coded\_axis\_flag)]

Two 8-bit binary masks mask\_planar\_fixed0[axisIdx] and mask\_planar[axisIdx] are determined for the current node and for an axis index axisIdx.

The first mask mask\_planar\_fixed0[axisIdx] is constructed that such its i-th bit, for i = 0 .. 7, is set to 1 if the corresponding i-th child node belongs to the lower plane along the axisIdx-th axis. This bit is set 0 if the child node belongs to the upper plane.

If the node is not planar along the axisIdx-th axis, i.e. is\_planar\_flag[nodeIdx ][ axisIdx] is equal to 0, then mask\_planar[axisIdx] is set to 0.

Otherwise, if is\_planar\_flag[ nodeIdx ][ axisIdx ] is equal to 1, the node is planar along the axisIdx-th axis, the occupied plane position is known from plane\_position[ nodeIdx ][ axisIdx ], and the i-th bit, for i = 0 .. 7, of mask\_planar[ axisIdx ] is set to 0 if the corresponding i-th child node belongs to the occupied plane, 1 otherwise.

By construction of mask\_planar[axisIdx], its bits whose value is 1 do mask the occupancy bits corresponding to child nodes for which it is known, from the planar information, that they are not occupied.

### Occupancy\_idx[] parsing process

When occupancy\_idx[axisIdx], for axisIdx in the range 0 .. 2,is not present, the value of occupancy\_idx[axisIdx] is inferred by the corresponding plane position, if the latter is present, as follows,

if (is\_planar\_flag[nodeIdx ][axisIdx])

occupancy\_idx[axisIdx] = plane\_position[nodeIdx][axisIdx]

If all three values occupancy\_idx[axisIdx] are either present or inferred by the corresponding plane position ,the following applies:

OccupancyMap = 1 << (occupancy\_idx[2] | (occupancy\_idx[1] << 1) | (occupancy\_idx[0] << 2))

If single\_occupancy\_flag is equal to 0, two\_planar\_flag[nodeIdx] is equal to 1, and is\_planar\_flag[nodeIdx][axisIdx] is equal to 0, for an axis index axisIdx, then only two child nodes are occupied along the axisIdx-th axis. In this case, OccupancyMap is determined as follows

if (!single\_occupancy\_flag && two\_planar\_flag[nodeIdx]) {

if (!is\_planar\_flag[nodeIdx][0])

OccupancyMap =  
 (1 << (occupancy\_idx[2] | (occupancy\_idx[1] << 1)))  
 | (1 << (occupancy\_idx[2] | (occupancy\_idx[1] << 1) | 1 << 2))

if (!is\_planar\_flag[nodeIdx][1])

OccupancyMap =  
 (1 << (occupancy\_idx[2] | (occupancy\_idx[0] << 2)))  
 | (1 << (occupancy\_idx[2] | 1 << 1 | (occupancy\_idx[0] << 2)))

if (!is\_planar\_flag[nodeIdx][2])

OccupancyMap =  
 (1 << (occupancy\_idx[1] << 1 | (occupancy\_idx[0] << 2)))  
 | (1 << (1 | occupancy\_idx[1] << 1 | (occupancy\_idx[0] << 2)))

}

OccupancyMap = 1 << (occupancy\_idx[2] | (occupancy\_idx[1] << 1) | (occupancy\_idx[0] << 2))

### Inverse binarization process

This process reconstructs a value of the syntax element occupancy\_map.The input to this process is the variables NeighbourPattern and the planar information mask\_planar[] and mask\_planar\_fixed0[] associated with the current node.

The output from this process is the syntax element value, constructed as follows:

value = 0

min\_non\_zero\_node = NeighbourPattern == 0 ? 2 : 1

for (axisIdx = 0; axisIdx <= 2; axisIdx++)

min\_non\_zero\_plane[axisIdx] = NeighbourPattern == 0 && mask\_planar[axisIdx] ? 2 :1

initialize\_counters\_for\_zeros()

for (BinIdx = 0; BinIdx < 8; BinIdx++) {

binIsInferred0 =  
 ((mask\_planar[0] >> bitCodingOrder[BinIdx]) & 1)  
 || ((mask\_planar[1] >> bitCodingOrder[BinIdx]) & 1)  
 || ((mask\_planar[2] >> bitCodingOrder[BinIdx]) & 1)

if (binIsInferred0) {

bin = 0

continue

}

determine\_binIsInferred1()

if (binIsInferred1)

bin = 1

else {

bin = readOccBin()

if (!bin)

update\_counters\_for\_zeros()

}

value = value | (bin << bitCodingOrder[BinIdx])

}

where bitCodingOrder[ BinIdx ] is defined by Table 24, and readOccBin() is specified by 9.7.6,

Table 24 — Values of bitCodingOrder[i]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **value** | 1 | 7 | 5 | 3 | 2 | 6 | 4 | 0 |

The variable binIsInferred0 is set equal to 1 when the value of the bin can be deduced to be 0 from the planar information associated with the node, e.g. when the bin corresponds to the occupancy bit of a child node that belongs to a plane known to be unoccupied from the planar information. Otherwise, binIsInferred0 is set equal to 0.

If binIsInferred0 equal 0, the variable binIsInferred1 is set equal to 1 when the value of the bin can be deduced to be 1 from the planar information, the minimum number min\_non\_zero\_node of non-zero bins in the node, and the minimum number min\_non\_zero\_plane[axisIdx] of non-zero bins in the occupied plane along the axisIdx-th axis (would the node be palnar along this axis). Otherwise, binIsInferred1 is set equal to 0.

The value of binIsInferred1 is determined based on counters coded0[axisIdx][planePos] that counts the number of occupancy bits already known to be zero for a plane position planePos (either equal to 0 for the lower plane, or equal to 1 for the upper plane) along the axisIdx-th axis. The counters are initialized at the start of the inverse binarization process as follows

initialize\_counters\_for\_zeros() {

for (axisIdx = 0; axisIdx <= 2; axisIdx++)

for (planePos = 0; planePos <= 1; planePos++)

coded0[axisIdx][planePos] = 0

for (i = 0; i < 8; i++) {

if ((mask\_planar[0] >> i) & 1 || ((mask\_planar[1] >> i) & 1 || ((mask\_planar[2] >> i) & 1) {

coded0[0][(mask\_planar\_fixed0[0] >> i) & 1]++

coded0[1][(mask\_planar\_fixed0[1] >> i) & 1]++

coded0[2][(mask\_planar\_fixed0[2] >> i) & 1]++

}

}

}

Thus, the counters coded0[][] are initialized counting the number of occupancy bits known to be zero from the planar information. Each time a bin is decoded by readOccBin( ) and this decoded bin is equal to 0, the counters coded0[][] are updated by

update\_counters\_for\_zeros() {

coded0[0][(mask\_planar\_fixed0[0] >> bitCodingOrder[BinIdx]) & 1]++

coded0[1][(mask\_planar\_fixed0[1] >> bitCodingOrder[BinIdx]) & 1]++

coded0[2][(mask\_planar\_fixed0[2] >> bitCodingOrder[BinIdx]) & 1]++

}

When binIsInferred0 equal 0, the determaination of the value of binIsInferred1 performed as follows

determine\_ binIsInferred1() {

for (axisIdx = 0; axisIdx <= 2; axisIdx ++) {

mask0 = mask\_planar\_fixed0[axisIdx] >> bitCodingOrder[BinIdx]) & 1

binIsOne[axisIdx] =  
 (eligible\_planar\_flag[axisIdx]  
 && coded0[axisIdx][mask0] >= 4− min\_non\_zero\_plane[axisIdx])  
 || coded0[axisIdx][0] + coded0[axisIdx][0] >= 8 − min\_non\_zero\_node

}

binIsInferred0 = binIsOne[0] || binIsOne[1] || binIsOne[2]

}

In this process binIsOne[axisIdx] is equal to 1 when the bin can be deduced to be 1 from the planar information along the axisIdx-th axis; it is equal to 0 otherwise. This deduction can be performed because either the node the planarity of the node is known and already at least 4 − min\_non\_zero\_plane[axisIdx] bins are known to be or have been decoded to zero, or already at least 8 − min\_non\_zero\_node bins are known to be or have been decoded to zero.

### Definition of readOccBin()

The inputs to this process are the variables BinIdx, and PartialSynVal.

The output from this process is the value of the decoded bin.

The process for a decoding a single bin is as follows:

The variables ctxMapIdx and ctxIdx are determined according to the derivation process 9.7.7 with the variables NeighbourPattern, BinIdx, and PartialSynVal as input.

The arithmetic decoding process 9.10.2 for a single bin is invoked for the syntax element occupancy\_map with the variable ctxIdx as input. The output binVal is the value of the decoded bin.

The context map update process 9.7.8 is invoked with the variable ctxMapIdx and the decoded bin value.

### ctxMapIdx and ctxIdx derivation processes

Inputs to this process are,

* the variable NeighbourPattern, representing the occupancy of the neighbours of the current node’s parent neighbours,
* the planar information mask\_planar associated with the current node,
* the variable depth, indicating the current geometry tree depth,
* the variable binIdx, indicating the bin to be decoded, and
* the variable partialSynVal, representing the partially reconstructed value of the syntax element.

Output of by this process are the variables ctxMapIdx andctxIdx.

The variable idxPred is set as follows:

* If NodeMaxDimLog2 is greater than or equal to log2\_intra\_pred\_max\_node\_size, the variable idxPred is set equal to 0.
* If any of mask\_planar[ k ], k = 0 .. 2, are not equal to 0, the variable idxPred is set equal to 0.
* Otherwise, the variable idxPred is set equal to the output of the occupancy prediction process using neighbouring octree nodes (9.7.9) when invoked with the current node and childIdx set equal to the output of the neighbour dependent geometry octree child node scan order Inverse mapping process (6.4.1) with the inputs neighbourPattern and inIdx set equal to bitCodingOrder[ binIdx ] where values of bitCodingOrder[ ] are given in Table 24.

The variable idxAdj is set as follows:

If adjacent\_child\_contextualization\_enabled\_flag is equal to 1, the following applies:

The variables adjOcc and adjUnocc are initialized to 0.

The variables sC, tC, and vC identifying the position of the child node associated with binIdx at depth + 1 are initialized as follows

sC = 2 × sN + ((bitCodingOrder[binIdx] >> 2) & 1)

tC = 2 × tN + ((bitCodingOrder[binIdx] >> 1) & 1)

vC = 2 × vN + (bitCodingOrder[binIdx] & 1)

The following procedure is performed for each of the s, t, and v axes by substituting the variables aK, aN, aC, nPmask, sCn, tCn, vCn, sNn, tNn, and vNn of the corresponding row of Table 25.

// if child is adjacent to a causally-valid neighbour

if (!(aC & 1)) {

if (NeighboutPattern & nPmask) {

aD = !depth ? 1 : geom\_tree\_coded\_axis\_flag[depth - 1][aK] ? 1 : 2

adjOcc += GeometryNodeOccupancyCnt[depth + 1][sCn][tCn][vCn]

} else

// if neighbour is available but not present

if ((aN + 1) & NeighbAvailabilityMask != 1)

if (GeometryNodeOccupancyCnt[depth][sNn][tNn][vNn] == 0)

adjUnocc = 1

}

Table 25 — Variable substitutions for the computation of adjOcc and adjUnocc

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **axis** | **aK** | **aN** | **aC** | **nPmask** | **sCn** | **tCn** | **vCn** | **sNn** | **tNn** | **vNn** |
| s | 0 | sN | sC | 2 | sC − aD | tC | vC | sN−1 | tN | vN |
| t | 1 | tN | tC | 4 | sC | tC − aD | vC | sN | tN−1 | vN |
| v | 2 | vN | vC | 16 | sC | tC | vC − aD | sN | tN | vN−1 |

The variable idxAdj is derived as follows

idxAdj = adjUnocc + 2 × Min(2, adjOcc)

if (binIdx > 4)

idxAdj = ctxIdxAdjReduc567[idxAdj]

Table 26 — Values of ctxIdxAdjReduc567[ i ]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 |
| **ctxIdxAdjReduc567[ i ]** | 0 | 0 | 1 | 2 | 3 | 3 |

The variable ctxIdxMapIdx is set equal to 3 × idxAdj + idxPred.

The output variable ctxMapIdx is derived as follows:

If NeighbourPattern is equal to 0, ctxIdxMapOffset is set equal to popcnt( partialSynVal ).

Otherwise, NeighbourPattern is not equal to 0, the following applies:

if (neighbour\_context\_restriction\_flag)

pattern = neighbourPattern64to9[NeighbourPattern];

else

pattern = neighbourPattern64to6[NeighbourPattern];

if (binIdx == 7)

pattern = 1;

else if (binIdx == 6)

pattern = neighbourPattern9to3[pattern];

else if (binIdx > 3)

pattern = neighbourPattern9to5[pattern];

ctxIdxMapOffset = ((pattern − 1) << binIdx) + partialSynVal + binIdx + 1;

Finally, the output variable ctxIdx is set as follows

ctxMapIdx = ctxIdxMapIdx × 1499 + ctxIdxMapOffset

ctxIdx = CtxMap[ctxMapIdx] >> 3

Table 27 — Values of neighbourPattern64to9[ j + i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **i** | | | | | | | | | | | | | | | |
| **j** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **0** | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 3 | 1 | 3 | 3 | 4 |
| **16** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **32** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **48** | 1 | 3 | 3 | 4 | 3 | 6 | 6 | 7 | 3 | 6 | 6 | 7 | 4 | 7 | 7 | 8 |

Table 28 — Values of neighbourPattern64to6[ j + i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **j** | **i** | | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **0** | 0 | 5 | 5 | na | 5 | 1 | 1 | na | 5 | 1 | 1 | na | na | na | na | na |
| **16** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **32** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **48** | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na |

Table 29 — Specification of neighbourPattern9to5[ i ]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |
| **neighbourPattern9to5[ i ]** | 0 | 1 | 2 | 3 | 11 | 22 | 3 | 4 | 44 |  |

Table 30 — Specification of neighbourPattern9to3[ i ]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **neighbourPattern9to3[ i ]** | 0 | 1 | 11 | 22 | 22 | 11 | 22 | 2 | 2 |

### Context map update process

This process updates the context mapping table for the syntax element occupancy\_map.

Input to this process are the variable ctxMapIdx and a decoded bin value.

The context mapping CtxMap[ctxMapIdx] is updated as follows:

stateVal = CtxMap[ctxMapIdx]

if (binVal)

CtxMap[ctxMapIdx] += ctxMapTransition[(255 − stateVal) >> 4]

else

CtxMap[ctxMapIdx] −= ctxMapTransition[stateVal >> 4]

Where values of ctxMapTransition are given by Table 31.

Table 31 — Values of ctxMapTransition[ i ]

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

### Occupancy prediction process using neighbouring octree nodes

The occupancy prediction process generates a tri-state occupancy prediction of a single child node based on the occupancy state of nodes neighbouring the parent node.

Input to this process are

the variables sN, tN, vN, and depth, identifying a node in the geometry octree, and

the variable childIdx identifying a child node position according to the geometry octree child traversal order for occupancy prediction.

Output from this process is the predicted occupancy state for the specified child node.

A list of neighbouring occupied geometry tree nodes is determined as follows:

for (i = 0; i < 26; i++) {

s = sN + dS[i]

t = tN + dT[i]

v = vN + dV[i]

if (available(sN, tN, vN, s, t, v))

occupiedNeigh[i] = GeometryNodeOccupancyCnt[depth][s][t][v] != 0

else

occupiedNeigh[i] = 0

}

Where the function available( sN, tN, vN, s, t, v ) evaluates to true if all of the following conditions are true:

log2\_neighbour\_avail\_boundary > 0

(s ^ sN) >> log2\_neighbour\_avail\_boundary == 0

(t ^ tN) >> log2\_neighbour\_avail\_boundary == 0

(v ^ vN) >> log2\_neighbour\_avail\_boundary == 0

And where the values of the neighbour position offsets dS[ i ], dT[ i ], and dV[ i ] are given in Table 32.

A score is determined for the predicted child node as the sum of the score contributions of occupied neighbours:

score = 0

for (i = 0; i < 26; i++)

score += childScore[i][childIdx] && occupiedNeigh[i]

A threshold for predicting the predicted child node as occupied is determined according to the number of occupied neighbour nodes:

numOccupiedNeigh = 0

for (i = 0; i < 26; i++)

numOccupiedNeigh += occupiedNeigh[i]

thresholdOccupied = numOccupiedNeigh < 14 ? 4 : 5

The output predicted occupancy state, prediction, is set according to the following:

if (score <= 2)

prediction = 1

else if (score >= thresholdOccupied)

prediction = 2

else

prediction = 0

Table 32 — Values of dS[ i ], dT[ i ], dV[ i ], and childScore[ i ][ childIdx ] for intra occupancy prediction

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **childScore[ i ][ childIdx ]** | | | | | | | |
| **i** | **dS[ i ]** | **dT[ i ]** | **dV[ i ]** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 0 | −1 | −1 | −1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | −1 | −1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | −1 | −1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | −1 | 0 | −1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | −1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | −1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6 | −1 | 1 | −1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 7 | −1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | −1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 9 | 0 | −1 | −1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | −1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 11 | 0 | −1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 12 | 0 | 0 | −1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 13 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 14 | 0 | 1 | −1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 15 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 16 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 17 | 1 | −1 | −1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 18 | 1 | −1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 19 | 1 | −1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 20 | 1 | 0 | −1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 21 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 22 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 23 | 1 | 1 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 24 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 25 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |







## Inferred Direct Coding Mode parsing process

### General process

The parsing and inverse binarization of the syntax element point\_offset[ i ][ XXX ][ j ], associated with the j-th bit of the i-th point coded by a child node is described in 9.8.3.

### Bypassing a first componenet S or T of point\_offset

The process to determine the component 1 − flagComp (whose vlaue is 0 for S, and 21 for T) of the bin point\_offset[ i ][ 1 − flagComp ][ j ] associated with j-th bit of the the i-th point belonging to the child node that undergoes Inferred Direct Coding Mode is described in this section that applies only when is geometry\_angular\_mode\_flag equal to 1.

The value of flagComp is determiend as follows.

sNode0 = sNchild − geomAngularOrigin[0]

tNote0 = tNchild − geomAngularOrigin[1]

flagComp = Abs(sNode0) <= Abs(tNode0) ? 0 : 1

The component c = 1− flagComp of point\_offset[ i ][ c ][ j ] is bypass coded. The other two components are arithmetically coded as described in the following sections.

### Determination of the laser index

The process to determine the laser index laserIndex[ Child ] for a child node Child that undergoes Inferred Direct Coding Mode is described in this section. This applies only when is geometry\_angular\_mode\_flag equal to 1.

Firstly, the estimate laserIndexEstimate[ Child ] is computed by determining a node angle NodeTheta

sNode = (sNchild − geomAngularOrigin[0]) << 8

tNote = (tNchild − geomAngularOrigin[1]) << 8

r2 = sNode × sNode + tNote × tNote

rInvLaser = invSqrt(r2)

NodeTheta = ((vNchild − geomAngularOrigin[2]) \* rInvLaser) >> 14

and determining the closest laser angle LaserAngle[ laserIndexEstimate[ Child ] ] to this node angle

start = 0

end = number\_lasers - 1

for (int t = 0; t <= 4; t++) {

mid = (start + end) >> 1

if (LaserAngle[mid] > NodeTheta)

end = mid

else

start = mid

}

minDelta = Abs(thetaList[start] - NodeTheta)

laserIndexEstimate [Child]= start

for (j = start + 1; j <= end; j++) {

delta = Abs(LaserAngle[j] − NodeTheta)

if (delta < minDelta) {

minDelta = delta

laserIndexEstimate [Child] = j

}

}

Secondly, a laser index residual laserIndexResidual[ Child ] is deduced from the decoded values

laserIndexResidual[Child] = 0

if (!laser\_residual\_eq0\_flag) {

residue = 1

if (!laser\_residual\_eq1\_flag) {

residue = 2

if (!laser\_residual\_eq2\_flag) {

residue = 3 + laser\_residual\_minus3

}

}

if (laser\_residual\_sign)

laserIndexResidual[Child] = residue

else

laserIndexResidual[Child] = -residue

}

The laser index laserIndex[ Child ] is then obtained by the sum

### laserIndex[Child] = laserIndexEstimate[Child] + laserIndexResidual[Child]Determination of the azimuthal context idcmIdxAzimuthal

The process to determine the context idcmIdxAzimuthal[ i ][ j ] for coding the bin point\_offset[ i ][ flagComp ][ j ], for flagComp being either 0 or 1, associated with j-th bit of the the i-th point belonging to the child node that undergoes Inferred Direct Coding Mode is described in this section which is a continuation of the preceding sections XREF 9.8.2 and XREF 9.8.3.

The azimuthal context applies to the s component (flagComp equal to 0) or to the t composent (flagComp equal to 1). The other component 1− flagComp is bypass coded as described in XREF 9.8.2. It is assumed that point\_offset[ i ][ 1 − flagComp ][ j ] have been decoded such that sNode0 or tNode0 is updated using the actual i-th point coordinate as follows.

if (flagComp)

sNote0 = PointOffset[i][0] − geomAngularOrigin[0]

else

tNote0 = PointOffset[i][1] − geomAngularOrigin[1]

Then, an initial value of an azimuthal predictor predPhi is determined from the buffer phiBuffer[] .

phiNode = iAtan2hp(tNote0, sNode0)

predph = phiBuffer[laserIndex[Child]]

if (predPhi == 0x80000000)

predPhi = phiNode

nShift = ((predPhi − phiNode) \* InvDeltaPhi[laserIndex[Child]] + 536870912) >> 30

predPhi −= DeltaPhi[laserIndex[Child]] \* nShift

The azimuthal contexts idcmIdxAzimuthal[ i ][ j ] are determined iteratively by a loop on j. [Ed: is there an issue here if the first bit is inferred by planar?]

mask = EffectiveChildNodeSizeLog2[flagComp] > 0

? 1 << EffectiveChildNodeSizeLog2[flagComp] − 1)

: 0

for (j = 0; mask; j++, mask >>= 1) {

phiR = flagComp ? iAtan2hp (tNote0 + mask, sNote0) : iAtan2hp(tNote0, sNote0 + mask)

phiL = phiNode

angleL = phiL - predPhi

angleR = phiR - predPhi

contextAnglePhi = (angleL >= 0 && angleR >= 0) || (angleL < 0 && angleR < 0) ? 2 : 0

angleL = Abs(angleL)

angleR = Abs(angleR)

if (angleL > angleR) {

contextAnglePhi++

int temp = angleL

angleL = angleR

angleR = temp

}

if (angleR > (angleL << 1))

contextAnglePhi += 4

if (angleR > (angleL << 2))

contextAnglePhi += 4

if (angleR > (angleL << 4))

contextAnglePhi += 4

idcmIdxAzimuthal[i][j] = contextAnglePhi

// decode the bin point\_offset[i][flagComp][j] using idcmIdxAzimuthal[i][j]

if (point\_offset[i][flagComp][j]) {

if (flagComp )

tNote0 += mask

else

sNote0 += mask

phiNode = phiR

}

}

### Determination of the angular context idcmIdxAngular

The process to determine the context idcmIdxAngular[ i ][ j ] for coding the bin point\_offset[ i ][ 2 ][ j ] associated with j-th bit of the the i-th point belonging to a the child node that undergoes Inferred Direct Coding Mode is described in this section which is a continuation of the preceding section 9.8.2.

This process is performed after point\_offset[ i ][ 0 ][ j ] and point\_offset[ i ][ 1 ][ j ] are decoded such that PointOffset[ i ][ 0 ] and PointOffset[ i ][ 1 ] are known. The s and t position, relative to the Lidar, of the point i is derived by

posSlidar[i] = sNchild − geomAngularOrigin[0] + PointOffset[i][0]

posTlidar[i] = tNchild − geomAngularOrigin[1] + PointOffset[i][1]

where (sNchild, tNchild, vNchild) specifying the position of the geometry octree child node Child in the current slice.

The inverse rInv of the radial distance of the point from the Lidar is determined by

sLidar = (posSlidar[i] << 8) − 128

tLidar = (posTlidar[i] << 8) − 128

r2 = sLidar × sLidar + tLidar × tLidar

rInv = invSqrt(r2)

The corrected laser angle ThetaLaser of the laser associated with the child nodeChild is deduces by

Hr = LaserCorrection[laserIndex[Child]] × rInv

ThetaLaser = LaserAngle[laserIndex[Child]] + (Hr >= 0 ? −(Hr >> 17) : ((−Hr) >> 17))

Assuming that the bits point\_offset[ i ][ 2 ][ j2 ] for j2 = 0 .. j − 1, are known, the point is known to belong to a virtual vertical interval whose half size is provided by

halfIntervalSize[j] = (1 << (EffectiveChildNodeSizeLog2[2] − 1)) >> j

and a partial v point position posVlidarPartial[ i ][ j ], that provides the lower end of the interval, is deduced by

PointOffsetVpartial = 0;

for (j2 = 0; j2 < j; j2++)

PointOffsetVpartial[i] += point\_offset[i][2][j2] << j2

PointOffsetVpartial[i] <<= (EffectiveChildNodeSizeLog2[2] − j)

posVlidarPartial[i][j] = vNchild − geomAngularOrigin[2] + PointOffsetVpartial[i]

A relative laser position thetaLaserDeltaVirtualInterval relative to the middle of the virtual interval is computed by

vLidar = ((posVlidarPartial[i][j] + halfIntervalSize[j]) << 1) − 1

theta = zLidar × rInv

theta32 = theta >= 0 ? theta >> 15 : −((−theta) >> 15)

thetaLaserDeltaVirtualInterval = ThetaLaser − theta32;

Two angular differences, deltaVirtualIntervalTop and deltaVirtualIntervalBot, of the laser relative to a lower and an upper v position in the virtual interval are determined.

vShift = ((rInv << EffectiveChildNodeSizeLog2[2]) >> 18) >> j

deltaVirtualIntervalTop = thetaLaserDeltaVirtualInterval − vShift

deltaVirtualIntervalBot = thetaLaserDeltaVirtualInterval + vShift

Then, the angular context is deduced from the two angular differences.

idcmIdxAngular[i][j] = thetaLaserDeltaVirtualInterval < 0

if (deltaVirtualIntervalTop >= 0 || deltaVirtualIntervalBot < 0)

idcmIdxAngular[i][j] += 2

### Inverse binarization process

When Inferred Direct Coding Mode is applied to a child node Child, the bits point\_offset[ i ][ c ][ j ] of the c-th component of the i-th point in the child node, for j in the range 0 .. EffectiveChildNodeSizeLog2[ 2 ] or in the range 1 .. EffectiveChildNodeSizeLog2[ c ] in case the first bit is inferred by the plane position plane\_position[ Child ][ c ], are decoded applying the following process.

If geometry\_angular\_mode\_flag is equal to 0, then the bit point\_offset[ i ][ c ][ j ] is decoded using the bypass decoding process.

Otherwise, geometry\_angular\_mode\_flag is equal to 1, the following applies:

* The bit point\_offset[ i ][ 1 − flagComp ][ 0 ] is bypass decoded when not inferred by the plane position. The bit point\_offset[ i ][ flagComp ][ j ], for j > 0, are bypass decoded.
* The bit point\_offset[ i ][ flagComp ][ 0 ] is bypass decoded when not inferred by the plane position. The bit point\_offset[ i ][ flagComp ][ j ], for j > 0, are decoded using the context idcmIdxAzimuthal[ i ][ j ].
* The bit point\_offset[ i ][ 2 ][ 0 ] is bypass decoded when not inferred by the plane position, and the bits point\_offset[ i ][ 2 ][ j ] are decoded using the context idcmIdxAngular[ i ][ j ] for j > 0.

## Attribute coefficient parsing processes

### Parsing of coeff\_abs\_level\_low

The coeff\_abs\_level\_low syntax element is coded with a fixed length prefix and a variable length suffix.

The four bin intvlIdx indicates the interval used to code and interpret the suffix value. It is decoded as follows:

intvlIdx = readBin(0)

intvlIdx = (intvlIdx << 1) | readBin(1 + intvlIdx)

intvlIdx = (intvlIdx << 1) | readBin(3 + intvlIdx)

intvlIdx = (intvlIdx << 1) | readBin(7 + intvlIdx)

The suffix comprises invlBits[ intvlIdx ] bypass bins:

for (suffix = 0, i = 0; i < intvlBits[intvlIdx]; i++)

suffix = readBin(bypass)

The decoded attribute value is intvlStart[ intvlIdx ] + suffix.

Table 1 — the values of intvlStart[ intvlIdx ] and intvlBits[ intvlIdx ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intvlIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **intvlStart** | 0 | 1 | 2 | 3 | 4 | 6 | 8 | 12 | 16 | 24 | 32 | 48 | 64 | 96 | 128 | 192 |
| **intvlBits** | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 |

## Dictionary-based parsing

### General process

This process is invoked when parsing syntax elements with descriptor de(v).

This process involves:

* An array of values lut0[k] storing the most frequent symbols, where k is in the range of 0 to 31, inclusive.
* An array of values lut0Histogram[k] storing the symbols occurrences, where k is in the range of 0 to 255, inclusive.
* Two variables lut0UpdatePeriod and lut0SymbolsUntilUpdate storing the update period for lut0 and the number of symbols remaining until the next update, respectively.
* A variable lut0Reset specifying whether lut0 should be reset during the next lut0 update or not.
* An array of values lut1[k] storing the last 16 decoded symbols, where k is in the range of 0 to 15, inclusive.
* A variable lut1IndexLastSymbol storing the index of the last decoded symbol.
* A static binary arithmetic context ctxStatic.
* A set of adaptive binary arithmetic contexts ctxLut0Hit, ctxLut1Hit, and ctxSymbolBit.
* An array of adaptive binary arithmetic contexts ctxLut0Index of size 5 if limitedContextMode equals 1, and 31 otherwise (i.e., limitedContextMode equals 0).

Inputs to this process are

a variable limitedContextMode specifying whether a limited number of contexts is used or not.

a variable lut0MaxOccurrence specifying the maximum allowed occurrence value in lut0Histogram[k].

two variables lut0InitialUpdatePeriod and lut0MaxUpdatePeriod specifying the initial update period and the maximum update period for the for lut0, respectively.

an array of values lut0Initilization[ k ] specifying the initial lut0 values, where k is in the range of 0 to 31, inclusive.

lut0 is initialized by invoking the initialization process in clause 9.9.2 with the parameters limitedContextMode and lut0Initilization.

lut0UpdatePeriod, lut0SymbolsUntilUpdate and lut0Reset are initialized as follows:

lut0UpdatePeriod = lut0InitialUpdatePeriod

lut0SymbolsUntilUpdate = lut0InitialUpdatePeriod

lut0Reset = 0

lut1 is initialized by invoking the initialization process in clause 9.9.3.

All the binary arithmetic contexts are initialized by invoking the process in clause 9.10.4.2.

Output from this process is an 8-bit syntax element value, constructed as follows.

lut0\_hit\_flag = readBin(ctxLut0Hit);

if (lut0\_hit\_flag) {

index = decodeLut0Index(limitedContextMode, ctxLut0Index);

value = lut0[index];

pushLut0(value);

} else {

lut1\_hit\_flag = readBin(ctxLut1Hit);

if (lut1\_hit\_flag) {

index = 0;

for (i = 0; i < 4; i++)

index |= readBin(ctxStatic) << i;

value = lut1[index];

} else {

value = 0;

for (i = 0; i < 8; i++)

value |= readBin(ctxSymbolBit) << i;

}

pushLut1(value);

pushLut0(value);

}

### Initializing lut0

Inputs to this process are

a variable limitedContextMode specifying whether a limited number of contexts is used or not.

an array of values lut0Initilization[ k ], to initialize lut0 where k is in the range of 0 to 31, inclusive.

lut0 is initialized according to the following process.

for (k = 0; k < 32; k++)

lut0[k]= limitedContextMode == 1 ? lut0Initlization[k] : k;

### Initializing lut1

lut1 is initialized according to the following process.

for (k = 0; k < 16; k++)

lut1[k]= k;

### Definition of decodeLut0Index()

Inputs to this process is a variable limitedContextMode specifying whether a limited number of contexts is used or not.

Output from this process is a 5-bit index, constructed as follows.

if (limitedContextMode == 1) {

b0 = readBin(ctxLutIndex[0]);

if (b0) {

b1 = readBin(ctxStatic);

b2 = readBin(ctxStatic);

b3 = readBin(ctxStatic);

b4 = readBin(ctxStatic);

} else {

b1 = readBin (ctxLutIndex[1]);

if (b1) {

b2 = readBin(ctxStatic);

b3 = readBin(ctxStatic);

b4 = readBin(ctxStatic);

} else {

b2 = readBin(ctxLutIndex[2]);

if (b2) {

b3 = readBin(ctxStatic);

b4 = readBin(ctxStatic);

} else {

b3 = readBin(ctxLutIndex[3]);

b4 = readBin(ctxLutIndex[4]);

}

}

}

index = (b0 << 4) | (b1 << 3) | (b2 << 2) | (b3 << 1) | b4;

} else {

index = 0;

index = (index << 1) | readBin(ctxLutIndex[0]);

index = (index << 1) | readBin(ctxLutIndex[1 + index]);

index = (index << 1) | readBin(ctxLutIndex[3 + index]);

index = (index << 1) | readBin(ctxLutIndex[7 + index]);

index = (index << 1) | readBin(ctxLutIndex[15 + index]);

}

### Definition of pushLut0()

Inputs to this process are

an 8-bit variable symbol specifying the symbol to be pushed to lut0.

a variable maxOccurrence specifying the maximum allowed occurrence value in lut0Histogram[k].

This process updates lut0 and lut0Histogram as follows.

lut0Histogram[symbol]++

if (lut0Histogram[symbol] > lut0MaxOccurrence) {

for (k = 0; k < 256; k++)

lut0Histogram[k] = lut0Histogram[k] >> 1;

}

lut0SymbolsUntilUpdate−−;

if (lut0SymbolsUntilUpdate == 0)

updateLut0();

### Definition of updateLut0()

This process updates lut0UpdatePeriod, lut0 and lut0Histogram as follows.

lut0UpdatePeriod = Min((5 × lut0UpdatePeriod) >> 2, lut0MaxUpdatePeriod);

lut0SymbolsUntilUpdate = lut0UpdatePeriod;

lut0ComputeMostFrequentSymbols()

if (lut0Reset) {

lut0Reset = false;

for (k = 0; k < 256; k++)

lut0Histogram[k] = 0;

}

### Definition of lut0ComputeMostFrequentSymbols()

This process updates lut0 such that it contains the 32 most frequent symbols based on the occurrence values stored in lut0Histogram. If two symbols S1 and S2 have the same occurrence the one with the smallest value is preferred.

### Definition of pushLut1()

Input to this process is an 8-bit variable symbol specifying the symbol to be pushed to lut1.

This process updates lut1 and lut1IndexLastSymbol as follows.

index = −1

for (k = 0; k < 16; k++) {

if (lut1[index] == symbol) {

index = k;

break;

}

}

lut1IndexLastSymbol++

index0 = lut1IndexLastSymbol % 16;

symbol0 = lut1[index0];

if (index == −1)

lut1[index0] = symbol;

else

swap(lut1[index0], lut1[index]);

## CABAC parsing process

### General

This process is invoked when parsing syntax elements with descriptor ae(v).

The input to this process is a request for the value of a syntax element.

The output of this process is the value of the syntax element.

The initialization processes 9.10.3.2 and 9.10.4.2 are invoked when starting to parse of any of the following syntax structures:

* geometry\_slice\_data (7.3.3.3)
* attribute\_slice\_data (7.3.4.3)

The parsing of the syntax element proceeds according to the corresponding process listed in Table 22.

### Definition of readBin()

The inputs to this process are the variable binIdx and an associated syntax element.

The outputs of this process is the value of the decoded bin and an updated context variable.

The values ctxTbl and ctxIdx are determined according to the entries for the associated syntax element in Table 35.

If the value of ctxIdx is not equal to the value 'bypass', the following applies:

* The arithmetic decoding process 9.10.4.3 for a single bin is invoked to determine the value of the decoded bin with the context variable Contexts[ ctxTbl ][ ctxIdx ] as input.
* The context map update process 9.10.3.3 is invoked with the context variable Contexts[ ctxTbl ][ ctxIdx ] and the decoded bin value.

Otherwise, the value of ctxIdx is equal be the value 'bypass', the following applies:

* If sps\_bypass\_stream\_enabled\_flag is equal to 0, the arithmetic decoding process 9.10.4.4 for a single bypass bin is invoked to determine the value of the decoded bin. Otherwise, sps\_bypass\_stream\_enabled\_flag is equal to 1, the readBypassStreamBit process 9.4 is invoked to determine the value of the decoded bin.

Table 35 — Values of ctxTbl and ctxIdx for binarized ae(v) coded syntax elements

| **Syntax element** | **ctxTbl** | **ctxIdx** |
| --- | --- | --- |
| geom\_node\_qp\_offset\_eq0\_flag | 28 | 0 |
| geom\_node\_qp\_offset\_sign\_flag | 29 | 0 |
| geom\_node\_qp\_offset\_abs\_minus1 | 30 | prefix: 0 sufix: bypass |
| single\_occupancy\_flag | 0 | 0 |
| occupancy\_idx[] | na | bypass |
| occupancy\_map | 1 | 0 .. 31 (9.7.7) |
| num\_points\_eq1\_flag[ ] | 2 | 0 |
| num\_points\_minus2[ ] | 3 | prefix:0 suffix: bypass |
| is\_planar\_flag[ ][ ] | 31 | 0 .. 11: (8.2.4.3) |
| plane\_position[ ][0] | 32 | 0 .. 52: planePosIdxAzimuthalS (8.2.4.4) |
| plane\_position[ ][1] | 33 | 0 .. 52: planePosIdxAzimuthalT (8.2.4.4) |
| plane\_position[ ][2] | 34 | 0 .. 40: planePosIdxAngular (8.2.4.5) |
| direct\_mode\_flag | 4 | 0 |
| num\_direct\_points\_gt1\_flag | 35 | 0 |
| not\_duplicated\_point\_flag | 2 | 0 |
| num\_direct\_points\_eq2\_flag | 36 | 0 |
| num\_points\_direct\_mode\_minus3 | 3 | prefix:0 suffix: bypass |
| point\_offset[ ][ 0 ][ ] | na 45 | bypass or  0 .. 3: idcmIdxAzimuthal (9.8.3) |
| point\_offset[ ][ 1 ][ ] | na  46 | bypass or  0 .. 3: idcmIdxAzimuthal (9.8.3) |
| point\_offset[ ][ 2 ][ ] | na  37 | bypass or  0 .. 3: idcmIdxAngular (9.8.3) |
| segment\_indicator[ ] | 6 | BinIdx |
| vertex\_position[ ] | 7 | BinIdx |
| all\_residual\_values\_equal\_to\_zero\_run | 8 | 0 .. 2 |
| pred\_index | 9 | Min(BinIdx, 1) |
| coeff\_eq0\_flag[ k ] | 10 |  |
| coeff\_abs\_level\_eq1\_flag[ k ] | 11 |  |
| coeff\_abs\_level\_low | XXX | XXX |
| coeff\_abs\_level\_remaining | 12 | 0 .. 6 |
| coeff\_sign\_flag | na | bypass |
| ptn\_point\_cnt\_gt1\_flag | 38 | 0 |
| ptn\_point\_cnt\_minus2 | 39 | prefix: 0 suffix: bypass |
| ptn\_child\_cnt[ ] | 40 | 0 .. 2 (XREF) |
| prn\_pred\_mode[ ] | 41 | 0 .. 2 (XREF) |
| ptn\_residual\_eq0\_flag[ k ] | 42 | k |
| ptn\_residual\_sign\_flag[ k ] | 43 | k |
| ptn\_residual\_abs\_log2[ k ] | 44 | 0 .. 526 (XREF) |
| ptn\_residual\_abs\_remaining[ ] | na | bypass |
| dict\_lut0\_hit\_flag | ctxTblD[0] | 0 |
| dict\_lut1\_hit\_flag | ctxTblD[1] | 0 |
| dict\_lut0\_idx | ctxTblD[2] | 0 .. 4 |
| dict\_lut1\_idx | ctxTblD[3] | bypass |
| dict\_direct\_value | ctxTblD[4] | 0 |

Table 36 — Values of ctxTblD[ n ] for de(v) coded syntax elements

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Syntax element** | **n** | | | | |
| **0** | **1** | **2** | **3** | **4** |
| **occupancy\_byte** | 13 | 14 | 15 | 16 | 17 |

### Context variables

#### General

A context variable is a 16-bit unsigned integer value that models the probability of a zero bin.

NOTE — The values 0, 0x8000, and 0x10000 represent the probability of a zero bin as impossible, equi-probable, and certain respectively. The values 0 and 0x10000 can never be attained due to the operation of the context update process.

Adaptive contexts are updated after decoding each bin, according to a probability look-up table. The update table supplies a value for incrementing or decrementing the probability of a zero bin based upon the upper eight bits of the context's current value.

The array Contexts, with values Contexts[ ctxTbl ][ ctxIdx ], represents individual context variables used by the CABAC parsing process. The values of ctxIdx for each value of ctxTbl are specified in Table .

#### Initialisation of context variables

The outputs of this process are initialized CABAC state variables.

All context variables of the arithmetic decoding engine are initialized to the value 0x8000.

#### Context variable update process

The inputs to this process are the variable binVal representing the value of a decoded bin, and a context variable ctx.

The output of this process is the updated context variable.

The context variable is updated as follows:

if (binVal)

ctx −= CtxUpdateDelta[ctx >> 8];

else

ctx += CtxUpdateDelta[255 − (ctx >> 8)];

where values of CtxUpdateDelta[ ] are given in Table 37.

Table 37 — 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 | 1001 |
| **84** | 1020 | 1038 | 1057 | 1076 | 1095 | 1114 | 1133 | 1153 | 1172 | 1192 | 1211 | 1231 |
| **96** | 1251 | 1271 | 1291 | 1311 | 1332 | 1352 | 1373 | 1393 | 1414 | 1435 | 1456 | 1477 |
| **108** | 1498 | 1520 | 1541 | 1562 | 1584 | 1606 | 1628 | 1649 | 1671 | 1694 | 1716 | 1738 |
| **120** | 1760 | 1783 | 1806 | 1828 | 1851 | 1874 | 1897 | 1920 | 1935 | 1942 | 1949 | 1955 |
| **132** | 1961 | 1968 | 1974 | 1980 | 1985 | 1991 | 1996 | 2001 | 2006 | 2011 | 2016 | 2021 |
| **144** | 2025 | 2029 | 2033 | 2037 | 2040 | 2044 | 2047 | 2050 | 2053 | 2056 | 2058 | 2061 |
| **156** | 2063 | 2065 | 2066 | 2068 | 2069 | 2070 | 2071 | 2072 | 2072 | 2072 | 2072 | 2072 |
| **168** | 2072 | 2071 | 2070 | 2069 | 2068 | 2066 | 2065 | 2063 | 2060 | 2058 | 2055 | 2052 |
| **180** | 2049 | 2045 | 2042 | 2038 | 2033 | 2029 | 2024 | 2019 | 2013 | 2008 | 2002 | 1996 |
| **192** | 1989 | 1982 | 1975 | 1968 | 1960 | 1952 | 1943 | 1934 | 1925 | 1916 | 1906 | 1896 |
| **204** | 1885 | 1874 | 1863 | 1851 | 1839 | 1827 | 1814 | 1800 | 1786 | 1772 | 1757 | 1742 |
| **216** | 1727 | 1710 | 1694 | 1676 | 1659 | 1640 | 1622 | 1602 | 1582 | 1561 | 1540 | 1518 |
| **228** | 1495 | 1471 | 1447 | 1422 | 1396 | 1369 | 1341 | 1312 | 1282 | 1251 | 1219 | 1186 |
| **240** | 1151 | 1114 | 1077 | 1037 | 995 | 952 | 906 | 857 | 805 | 750 | 690 | 625 |
| **252** | 553 | 471 | 376 | 255 |  |  |  |  |  |  |  |  |

### Arithmetic decoding engine

#### General

The arithmetic decoding engine is a multi-context adaptive binary arithmetic decoder, performing binary renormalisation and producing binary outputs.

NOTE — The arithmetic decoding engine is based upon that of Dirac|SMPTE VC-2.

The arithmetic decoder state consists of the following variables:

* ivlLow, an integer representing the beginning of the current coding interval.
* ivlRange, an integer representing the size of the current coding interval.
* ivlCode, an integer within the interval[ ivlLow, ivlLow + ivlRange − 1 ], updated from the encoded bitstream.

#### Initialisation process

The outputs of this process are the initialized arithmetic decoding engine variables ivlLow, ivlRange, and ivlCode.

At the start of the decoding of any data unit, the arithmetic decoding state shall be initialized as follows:

ivlLow = 0;

ivlRange = 0xffff;

ivlCode = 0;

for (i = 0; i < 15; i++) {

ivlCode <<= 1;

ivlCode += readAeStreamBit();

}

#### Decoding process for a single binary value

The inputs to this process are the context variable ctx and the state variables ivlLow, ivlRange, and ivlCode.

The outputs of this process are the decoded binary value binVal, and the updated state variables ivlLow, and ivlRange.

The output binVal, and the updated state variables ivlRange, and ivlCode are determined as follows:

count = ivlCode − ivlLow;

rangeTimesProb = (ivlRange × ctx) >> 16;

binVal = count >= rangeTimeProb;

if (!binVal)

ivlRange = rangeTimesProb;

else {

ivlLow += rangeTimesProb;

ivlRange −= rangeTimesProb;

}

#### Decoding process for a single binary bypass value

The inputs to this process are the state variables ivlLow, ivlRange, and ivlCode.

The outputs of this process are the decoded binary value binVal, and the updated state variables ivlLow, and ivlRange.

The output binVal, and the updated state variables ivlRange, and ivlCode are determined as follows:

count = ivlCode − ivlLow;

rangeTimesProb = ivlRange >> 1;

binVal = count >= rangeTimeProb;

if (!binVal)

ivlRange = rangeTimesProb;

else {

ivlLow += rangeTimesProb;

ivlRange −= rangeTimesProb;

}

#### Arithmetic decoder state renormalisation process

Renormalisation stops the arithmetic decoding engine from losing accuracy. Renormalisation shall be applied while the range is less than or equal to a quarter of the total available 16-bit range (0x4000). Each renormalisation doubles the interval and reads a bit into the codeword.

The inputs to this process are the state variables ivlLow, ivlRange, and ivlCode.

The outputs of this process are the updated state variables ivlLow, ivlRange, and ivlCode.

While ivlRange is less than or equal to 0x4000, the following applies:

if ((ivlLow + ivlRange − 1) ^ ivlLow >= 0x8000) {

ivlCode ^= 0x4000;

ivlLow ^= 0x4000;

}

ivlRange <<= 1;

ivlLow = (ivlLow << 1) & 0xffff;

ivlCode = ((ivlCode << 1) | readAeStreamBit()) & 0xffff;

### Arithmetic encoding engine (informative)

#### General (informative)

This clause does not form an integral part of this Specification.

The inputs to this process are binary symbols that are to be encoded.

The outputs of this process are bits that are written to the data unit bytestream.

This informative clause describes an arithmetic encoding engine that matches the arithmetic decoding engine described in 9.10.4. The encoding engine is essentially symmetric with the decoding engine, i.e., procedures are called in the same order. Table 38 illustrates the correspondence between decoding and encoding processes.

Table 38 — Correspondence between decoder and encoder arithmetic coding processes

|  |  |  |
| --- | --- | --- |
| **Process** | **Decoder** | **Encoder** |
| Initialisation | 9.10.4.2 | 9.10.5.2 |
| Symbol coding | 9.10.4.3 | 9.10.5.3 |
| Renormalisation | 9.10.4.5 | 9.10.5.4 |
| Termination | — | 9.10.5.5 |

The state of the arithmetic encoding engine is represented by the variables ivlLow indicating the bottom of the encoding interval, ivlRange indicating the width of the encoding interval, and ivlCarry tracking the number of unresolved straddle conditions during renormalisation.

#### Initialization process (informative)

This clause does not form an integral part of this Specification.

This process is invoked before encoding the first ae(v) coded syntax element of a data unit.

The outputs of this process are the arithmetic encoding engine variables ivlLow, ivlRange, and ivlCarry, initialized as follows:

ivlLow = 0;

ivlRange = 0xFFFF;

ivlCarry = 0;

With 16 bit accuracy, 0xFFFF corresponds to an interval width value of (almost) 1.

#### Encoding process for a single binary value (informative)

This clause does not form an integral part of this Specification.

The inputs to this process are the context variable ctx, the value of binVal to be encoded, and the state variables ivlLow, and ivlRange.

The outputs of this process are the updated state variables ivlLow, and ivlRange.

Coding a binary value consists of, in order, scaling the interval[ ivlLow, ivlLow + ivlRange ], renormalising and outputting data.

rangeTimesProb = (ivlRange × ctx) >> 16;

if (!binVal)

ivlRange = rangeTimesProb;

else {

ivlLow += rangeTimesProb;

ivlRange −= rangeTimesProb;

}

#### Arithmetic encoder state renormalisation process (informative)

This clause does not form an integral part of this Specification.

The inputs to this process are the variables ivlLow, ivlRange.

The outputs of this process are zero or more bits written to the data unit bitstream and the updated variables ivlLow, ivlRange.

Renormalisation must cause ivlLow and ivlRange to be modified exactly as in the decoder. In addition, during renormalisation bits are output when ivlLow and ivlLow + ivlRange agree in their most significant bits, taking into account carries accumulated when a straddle condition is detected.

While ivlRange is less than or equal to 0x4000, the following applies:

if ((ivlLow + ivlRange − 1) ^ ivlLow >= 0x8000) {

ivlLow ^= 0x4000;

ivlCarry++;

} else {

writeBit((ivlLow >> 15) & 1);

for (; ivlCarry > 0; ivlCarry−−)

writeBit((~ivlLow >> 15) & 1);

}

ivlRange <<= 1;

ivlLow <<= 1;

ivlLow &= 0xFFFF;

#### Arithmetic encoding engine termination process (informative)

This clause does not form an integral part of this Specification.

After encoding, there may be insufficient bits for a decoder to determine the final encoded symbols, partly because further renormalisation is required — for example, MSBs may agree but the range may still be larger than 0x4000) — 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, finally byte aligning the output with padding bits.

while ((ivlLow + ivlRange − 1) ^ ivlLow < 0x8000) {

writeBit((ivlLow >> 15) & 1);

for (; ivlCarry > 0; ivlCarry−−)

writeBit((~ivlLow >> 15) & 1);

ivlRange <<= 1;

ivlLow <<= 1;

ivlLow &= 0xFFFF;

}

while ((ivlLow & 0x4000) && ((ivlLow + ivlRange − 1) & 0x4000)) {

carry++;

ivlLow ^= 0x4000;

ivlLow &= 0x7FFF;

ivlLow <<= 1;

ivlRange <<= 1;

}

writeBit((ivlLow >> 15) & 1);

for (; ivlCarry > 0; ivlCarry−−)

writeBit((~ivlLow >> 15) & 1);

byte\_align();

## Parsing state memorization process

This process records the elements and values of the following arrays and variables for restoration by the parsing state restoration process (9.12):

* + The array Contexts from the CABAC parsing process (9.10)
  + The array CtxMap from the bit-wise geometry octree occupancy parsing process (9.7)
  + The arrays and variables lut0, lut0Histogram, lut0UpdatePeriod, lut0SymbolsUntilUpdate, lut0Reset, lut1, lut1IndexLastSymbol from the dictionar-based parsing process (9.9)
  + The array planeRate and variable localDensity from the planar coding mode (8.2.4)

## Parsing state restoration process

This process restores the elements and values of the following arrays and variables to those previously recorded by the parsing state memorization process (9.11):

* + The array Contexts from the CABAC parsing process (9.10)
  + The array CtxMap from the bit-wise geometry octree occupancy parsing process (9.7)
  + The arrays and variables lut0, lut0Histogram, lut0UpdatePeriod, lut0SymbolsUntilUpdate, lut0Reset, lut1, lut1IndexLastSymbol from the dictionar-based parsing process (9.9)
  + The array planeRate and variable localDensity from the planar coding mode (8.2.4)

1. 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 decoder implementations.

NOTE 1 – This Specification 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.

NOTE 2 – Encoders are not required to make use of any particular subset of features supported in a profile.

Each level specifies a set of limits on the values that may be taken by the syntax elements of this Specification. The level definition is used with all profiles. For any given profile, a level generally corresponds to a particular decoder processing load and memory capability.

The profiles that are specified in clause A.3 are also referred to as the profiles specified in Annex A.

* 1. Requirements on decoder capability

Capabilities of decoders conforming to this Specification 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 specified profile, the level supported for that profile should also be expressed.

Specific values are specified in this annex for the syntax elements main\_profile\_compatibility\_flag and level\_idc. All other values of main\_profile\_compatibility\_flag and level\_idc are reserved for future use by ISO/IEC.

NOTE – Decoders should infer that a reserved value of level\_idc between the values specified in this Specification indicates intermediate capabilities between the specified levels.

* 1. Profiles
     1. General

All constraints for SPSs, GPSs, and APSs that are specified are constraints for the parameter sets that are activated when the bitstream is decoded.

* + 1. Main profile

Bitstreams conforming to the Main profile shall obey the following constraints:

– Active SPSs shall have main\_profile\_compatibility\_flag equal to 1 only.

* The level constraints specified for the Main profile in clause A.4 shall be fulfilled.

Conformance of a bitstream to the Main profile is indicated by main\_profile\_compatibility\_flag being equal to 1.

Decoders conforming to the Main profile at a specific level (identified by a specific value of level\_idc) shall be capable of decoding all bitstreams for which all of the following conditions apply:

* The bitstream representation is indicated to conform to the Main profile.
* The bitstream representation is indicated to conform to a level that is lower than or equal to the specified level.
  1. Levels
     1. Level limits

For purposes of comparison of level capabilities, a particular level is considered to be a lower level than some other level when the value of the level\_idc of the particular level is less than that of the other level.

Table A. 1 specifies limits for each level.

A level to which a bitstream conforms are indicated by the syntax elements level\_idc as follows:

– level\_idc shall be set equal to a value of 20 times the level number specified in Table A. 1.

Table A. 1 — Level limits

|  |  |  |
| --- | --- | --- |
| **Level** | **Max points in a slice** | **MaxRootNodeDimLog2** |
| 4 | 1,100,000 | 21 |

1. Type-length-value bytestream format

**B.1 General**

This annex specifies syntax and semantics of a byte stream format for use by applications that deliver some or all of the data units as an ordered stream of bytes without any requirement for further encapsulation in a file format.

The byte stream format consists of a sequence of type-length-value encapsulation structures that each represent a single coded syntax structure.

**B.2 Syntax and semantics**

**B.2.1 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) |
| } |  |

**B.2.2 Semantics**

The order of TLV encapsulation stuctures shall follow the decoding order of the encapsulated syntax structures.

**tlv\_type** identifies the syntax structure represented by tlv\_payload\_byte[ ] according to 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 |
| 1 | 7.3.2.5 | Geometry parameter set |
| 2 | 7.3.3.1 | Geometry data unit |
| 3 | 7.3.2.6 | Attribute parameter set |
| 4 | 7.3.4.1 | Attribute data unit |
| 5 | 7.3.2.4 | Tile inventory |
| 6 | 7.3.2.7 | Frame boundary marker |
| 7 | 7.3.5 | Defaulted attribute data unit |

**tlv\_num\_payload\_bytes** indicates the length in bytes of tlv\_payload\_byte[ ].

**tlv\_payload\_byte**[ i ] is the i-th byte of payload data.

**B.3 TLV decoding process**

Input to this process is an ordered stream of bytes consisting of a sequence of TLV encapsulation structures.

Output of this process is a sequence of syntax structures.

The decoder repeatedly parses tlv\_encapsulation structures until the end of the bytestream has been encountered (as determined by unspecified means) and the last NAL unit in the byte stream has been decoded.

After parsing each tlv\_ encapsulation structure, the following occurs

the array DataUnitBytes is set equal to tlv\_payload\_byte[ ],

the variable DataUnitLength is set equal to tlv\_num\_payload\_bytes,

the parsing process in Table B. 1 corresponding to tlv\_type is invoked.