**ISO/IEC 23090-9:2019(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**

CD stage

**Warning for WDs and CDs**

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

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

*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*](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*](https://www.iso.org/iso/model_document-rice_model.pdf)

© ISO/IEC 2019

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 4](#_Toc12888267)

[Introduction 5](#_Toc12888268)

[1 Scope 1](#_Toc12888269)

[2 Normative references 1](#_Toc12888270)

[3 Terms and definitions 1](#_Toc12888271)

[3.1 General 1](#_Toc12888272)

[3.2 Geometry coding related 3](#_Toc12888273)

[3.3 Attribute coding related 3](#_Toc12888274)

[4 Abbreviations 4](#_Toc12888275)

[5 Conventions 4](#_Toc12888276)

[5.1 General 4](#_Toc12888277)

[5.2 Arithmetic operators 4](#_Toc12888278)

[5.3 Logical operators 5](#_Toc12888279)

[5.4 Relational operators 5](#_Toc12888280)

[5.5 Bit-wise operators 5](#_Toc12888281)

[5.6 Assignment operators 6](#_Toc12888282)

[5.7 Range notation 6](#_Toc12888283)

[5.8 Mathematical functions 6](#_Toc12888284)

[5.8.1 Definition of simpleAtan2 7](#_Toc12888285)

[5.8.2 Definition of popCnt 8](#_Toc12888286)

[5.8.3 Definition of iLog2 8](#_Toc12888287)

[5.8.4 Definition of iSqrt 8](#_Toc12888289)

[Definition of 9](#_Toc12888290)

[5.8.5 divExp2RoundHalfInf 9](#_Toc12888291)

[5.9 Vector operations 10](#_Toc12888292)

[5.10 Order of operation precedence 10](#_Toc12888293)

[5.11 Variables, syntax elements, and tables 11](#_Toc12888294)

[5.12 Text description of logical operations 12](#_Toc12888295)

[5.13 Processes 13](#_Toc12888296)

[6 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships 13](#_Toc12888297)

[6.1 Bitstream formats 13](#_Toc12888298)

[6.2 Source, decoded, and output picture formats 13](#_Toc12888299)

[6.3 Geometry octree 13](#_Toc12888300)

[6.3.1 Scan order of child nodes 13](#_Toc12888301)

[6.4 Neighbour relationships 13](#_Toc12888302)

[6.4.1 Neighbour dependent geometry octree child node scan order inverse mapping process 13](#_Toc12888303)

[6.4.2 Neighbour depending geometry occupancy map permutation process 14](#_Toc12888304)

[7 Syntax and semantics 15](#_Toc12888305)

[7.1 Method of specifying syntax in tabular form 15](#_Toc12888306)

[7.2 Specification of syntax functions and descriptors 16](#_Toc12888307)

[7.3 Syntax in tabular form 17](#_Toc12888308)

[7.3.1 General 17](#_Toc12888309)

[7.3.2 Payload and byte alignment syntax 18](#_Toc12888310)

[7.3.3 Geometry payload syntax 21](#_Toc12888311)

[7.3.4 Attribute payload syntax 23](#_Toc12888312)

[7.4 Semantics 24](#_Toc12888313)

[7.4.1 General 24](#_Toc12888314)

[7.4.2 Payload and byte alignment semantics 24](#_Toc12888315)

[7.4.3 Geometry payload semantics 29](#_Toc12888316)

[7.4.4 Attribute payload semantics 34](#_Toc12888317)

[8 Decoding process 35](#_Toc12888318)

[8.1 General decoding process 35](#_Toc12888319)

[8.2 Geometry decoding process 36](#_Toc12888320)

[8.2.1 General geometry decoding process 36](#_Toc12888321)

[8.2.2 Octree node decoding process 36](#_Toc12888322)

[8.2.3 Geometry Trisoup decoding process 37](#_Toc12888323)

[8.3 Attribute decoding 47](#_Toc12888324)

[8.3.1 RAHT decoding process 47](#_Toc12888325)

[8.3.2 Lifting decoding process 51](#_Toc12888326)

[8.3.3 Predictive Lifting decoding process 64](#_Toc12888327)

[8.4 Slice concatentation process 67](#_Toc12888328)

[9 Parsing process 67](#_Toc12888329)

[9.1 General 67](#_Toc12888330)

[9.2 Definition of readPayloadBit 69](#_Toc12888331)

[9.3 General inverse binarisation processes 70](#_Toc12888332)

[9.3.1 Parsing of fixed-length codes 70](#_Toc12888333)

[9.3.2 Parsing of k-th order exp-Golomb codes 70](#_Toc12888334)

[9.3.3 Parsing of truncated unary codes 70](#_Toc12888335)

[9.3.4 Mapping process for signed codes 71](#_Toc12888336)

[9.4 Bitwise geometry octree occupancy parsing process 71](#_Toc12888337)

[9.4.1 General process 71](#_Toc12888338)

[9.4.2 Initialisation process 71](#_Toc12888339)

[9.4.3 Inverse binarization process 72](#_Toc12888340)

[9.4.4 Definition of readOccBin() 72](#_Toc12888341)

[9.4.5 ctxMapIdx and ctxIdx derivation processes 73](#_Toc12888342)

[9.4.6 Context map update process 76](#_Toc12888343)

[9.4.7 Occupancy prediction process using neighbouring octree nodes 76](#_Toc12888344)

[9.5 Dictionary-based parsing 79](#_Toc12888345)

[9.5.1 General process 79](#_Toc12888346)

[9.5.2 Initializing lut0 80](#_Toc12888347)

[9.5.3 Initializing lut1 81](#_Toc12888348)

[9.5.4 Definition of decodeLut0Index() 81](#_Toc12888349)

[9.5.5 Definition of pushLut0() 82](#_Toc12888350)

[9.5.6 Definition of updateLut0() 82](#_Toc12888351)

[9.5.7 Definition of lut0ComputeMostFrequentSymbols() 83](#_Toc12888352)

[9.5.8 Definition of pushLut1() 83](#_Toc12888353)

[9.5.9 Dictionary encoded value semantics 83](#_Toc12888354)

[9.6 CABAC parsing process 84](#_Toc12888355)

[9.6.1 General 84](#_Toc12888356)

[9.6.2 Definition of readBin() 84](#_Toc12888357)

[9.6.3 Context variables 86](#_Toc12888358)

[9.6.4 Arithmetic decoding engine 87](#_Toc12888359)

[9.6.5 Arithmetic encoding engine (informative) 89](#_Toc12888360)

[Annex A Profiles and levels 93](#_Toc12888361)

[A.1 Overview of profiles and levels 93](#_Toc12888362)

[Annex B Type-length-value bytestream format 94](#_Toc12888363)

[Annex C Bibliography 96](#_Toc12888364)

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

There are no normative references in this document.

# 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 cloud

sequence of point cloud frames

3.1.2

point cloud frame

set of 3D points specified by their Cartesian coordinates (x,y,z) and optionally a fixed set of corresponding attributes at a particular time instance

3.1.3

Cartesian coordinates

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

3.1.4

geometry

set of Cartesian coordinates associated with a point cloud frame

3.1.5

attribute

scalar or vector property optionally associated with each point in a point cloud such as colour, reflectance, surface normal, time stamps, material ID, etc.

3.1.6

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.

3.1.7

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)

3.1.8

shall

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

3.1.9

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.

3.1.10

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

3.1.11

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.

3.1.12

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

syntax structure

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

3.1.17

bounding box

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

3.1.18

3D tile

rectangular cuboid inside a bounding box.

3.1.19

slice

unit of bitstream that can be decoded independently from another slice.

## Geometry coding related

3.2.1

position

(x,y,z) coordinates 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.

3.2.4

octree

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

3.2.5

node

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

3.2.6

root node

node of the octree with no parent

3.2.7

leaf node

terminating node of the octree having no children

3.2.8

level

number of hops from the root to the node.

3.2.9

occupied node

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

3.2.10

occupancy code

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

3.2.11

Morton code

non-negative 3d-bit integer obtained by interleaving the bits of the non-negative d-bit integers x, y, and z.

## Attribute coding related

3.3.3

Component

array or single sample from one of the three arrays (x,y,z) that compose a point position or the array or a single sample of one of the arrays (A\_1 through A\_D) that compose a point attribute.

3.3.4

luma

An adjective specifying that a sample array or single sample is representing the monochrome signal related to the primary colours. The symbol or subscript used for luma is Y or L.

NOTE 1 to the entry:  The term luma is used rather than the term luminance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term luminance. The symbol L is sometimes used instead of the symbol Y to avoid confusion with the symbol y as used for vertical location.

3.3.5

chroma

An adjective specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours. The symbols used for a chroma array or sample are Cb and Cr.

NOTE 1 to the entry: The term chroma is used rather than the term chrominance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term chrominance.

# 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

TPS Tile 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.

## 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

Input of the process is the variable a and b.

Output of the 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:

the variable c and idx are derived as follow.

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 popCnt

Input of the process an integer variable x.

Output of the process is the number of 1-valued bits present in the binary representation of x.

### Definition of iLog2

Input to the process is a variable x.

Output of the process is a 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

Input to the process is a variable pIn.

Output of the process is a variable pOut.

A variable x and n are derived as follows.

x is initialized to 0 and n is initialized to 8.

The following apply:

while( n < = 64 & & x = = 0 ){

if( pIn >= ( 1<< (64 − n) ) )

x = ( tableSqrt[ pIn >> (64 − n) ] << (32 − (n/2) )  −  (n = = 8 ? 1 : 0 )

n += 8

}

[Ed. The derivation process should be clarified to align with the software.

The value of tableSqrt[ k ] with k = 0..255 is defined in Table 2.

Table 2 — the value of tableSqrt[ i+j ]



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | i | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 |
| 20 | 5 | 5 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 | 7 |
| 40 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 60 | 8 | 8 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 80 | 9 | 9 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 100 | 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 120 | 11 | 11 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 140 | 12 | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 160 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| 180 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 15 | 15 | 15 |
| 200 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 220 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 240 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

Finally, pOut is derived as follows.

x = (x + pIn/x) >> 1

pOut = ( (x+pIn)/(x+1) ) >> 1

### Definition of divExp2RoundHalfInf

Inputs of this process are:

a variable scalar.

a variable shift.

The output of the process is a 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

Inputs of this process are:

a variable scalar.

a variable shift.

The output of the process is a 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, u ).

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

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

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bit string form** | | | | **Integer form** |
| **s** | **t** | **u** | **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 | un ... u1 u0 | sn tn un  ... s1 t1 u1  s0 t0 u0 | ... |

### Definition of QpToQstep

Inputs of this process is:

a variable quantization parameter qp.

The output of the process is a variable quantization step qstep computed as follows:

if ( qp < 4) qp = 4;

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

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

## 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:

## Vector operations

The following mathematical functions are defined:

The function c[3] = CrossProduct ( a[3], b[3] ) 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[3], b[3] ) 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 4 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 4 – Operation precedence from highest (at top of table) to lowest (at bottom of table)

|  |
| --- |
| **operations (with operands x, y, and z)** |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| 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 xx 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 5.8) 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

<To do>

## Source, decoded, and output point cloud formats

<To do>

### Data partitioning

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. Geometry and attribute information of each slice can be encoded or decoded independently.

A tile is a group of slices with bounding box information. The bounding box information of each tile is specified in tile inventory. A tile may overlap another tile in the bounding box. Each slice contains an index that identifies to which tile it belongs

### 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

### Scan order of child nodes

## 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.

Input to this process are

an index, inIdx, in the neighbour dependent permuted child node scan order, and

the neighbourhood occupancy pattern, neighbourPattern.

Output by 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

where values of childScanMap are given by Table 5.

Table 5 — Values of childScanMap[ i + j ]

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

NOTE — Values are represented in octal form

TO DO: add change in the mapping depending on the mask\_planar[2], as in the code

mapGeometryOccupancy(uint8\_t occupancy, uint8\_t neighPattern, int mask\_planar\_z)

{

if ( mask\_planar\_z && !(mask\_planar\_z & 1) ) {

occupancy = kOccMapMirrorXY[occupancy];

}

etc.

### Neighbour depending geometry occupancy map permutation process

Inputs to this process are

a neighbourhood occupancy pattern neighbourPattern

a decoded occupancy map value occMap

Output from 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 )

}

where values of childScanMap[ ] are given by Table 5.

# 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.

– 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.

## 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.

### Payload and byte alignment syntax

#### Sequence parameter set syntax

|  |  |
| --- | --- |
| seq\_parameter\_set( ) { | Descriptor |
| **reserved\_profile\_compatibility\_23bits** [Ed. assign bits from this when there is a profile defined] | u(23) |
| **unique\_point\_positions\_constraint\_flag** | u(1) |
| **level\_idc** | u(8) |
| **sps\_bounding\_box\_present\_flag** | u(1) |
| if( sps\_bounding\_box\_present\_flag ) { |  |
| **sps\_bounding\_box\_offset\_x** | se(v) |
| **sps\_bounding\_box\_offset\_y** | se(v) |
| **sps\_bounding\_box\_offset\_z** | se(v) |
| **sps\_bounding\_box\_scale\_factor** | ue(v) |
| **sps\_bounding\_box\_size\_width** | ue(v) |
| **sps\_bounding\_box\_size\_height** | ue(v) |
| **sps\_bounding\_box\_size\_depth** | ue(v) |
| } |  |
| **sps\_source\_scale\_factor** [Ed. TMC13 v7 uses float, but integer is preferred.] | u(32) |
| **sps\_seq\_parameter\_set\_id** | ue(v) |
| **sps\_num\_attribute\_sets** | ue(v) |
| for( i = 0; i< sps\_num\_attribute\_sets; i++ ) { |  |
| **attribute\_dimension**[ i ] | ue(v) |
| **attribute\_instance\_id**[ i ] | ue(v) |
| **attribute\_bitdepth**[ i ] | ue(v) |
| **attribute\_cicp\_colour\_primaries**[ i ] | ue(v) |
| **attribute\_cicp\_transfer\_characteristics**[ i ] | ue(v) |
| **attribute\_cicp\_matrix\_coeffs**[ i ] | ue(v) |
| **attribute\_cicp\_video\_full\_range\_flag**[ i ] | u(1) |
| **known\_attribute\_label\_flag**[ i ] | u(1) |
| if( known\_attribute\_label\_flag[ i ] ) |  |
| **known\_attribute\_label**[ i ] | ue(v) |
| else |  |
| **attribute\_label\_four\_bytes**[ i ] | u(32) |
| } |  |
| **log2\_max\_frame\_idx** | u(5) |
| **axis\_coding\_order** | u(3) |
| **sps\_bypass\_stream\_enabled\_flag** | u(1) |
| **sps\_extension\_present\_flag** | u(1) |
| if( sps\_extension\_present\_flag ) |  |
| while( more\_data\_in\_byte\_stream( ) ) |  |
| **sps\_extension\_data\_flag** | u(1) |
| byte\_alignment( ) |  |
| **}** |  |

#### Tile inventory syntax

|  |  |
| --- | --- |
| tile\_inventory( ) { | Descriptor |
| **num\_tiles** | ue(v) |
| for( i = 0; i < num\_tiles; i++ ) { |  |
| **tile\_bounding\_box\_offset\_x**[ i ] | se(v) |
| **tile\_bounding\_box\_offset\_y**[ i ] | se(v) |
| **tile\_bounding\_box\_offset\_z**[ i ] | se(v) |
| **tile\_bounding\_box\_size\_width**[ i ] | ue(v) |
| **tile\_bounding\_box\_size\_height**[ i ] | ue(v) |
| **tile\_bounding\_box\_size\_depth**[ i ] | 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\_box\_present\_flag** | u(1) |
| if( gps\_box\_present\_flag ){ |  |
| **gps\_gsh\_box\_log2\_scale\_present\_flag** | u(1) |
| if( gps\_gsh\_box\_log2\_scale\_present\_flag = = 0 ) |  |
| **gps\_gsh\_box\_log2\_scale** | ue(v) |
| } |  |
| **unique\_geometry\_points\_flag** | u(1) |
| **geometry\_planar\_mode\_flag** | u(1) |
| **geom\_planar\_mode\_th\_IDCM** | ue(v) |
| 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) |
| } |  |
| **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) |
| **gps\_implicit\_geom\_partition\_flag** | u(1) |
| if( gps\_implicit\_geom\_partition\_flag ) { |  |
| **gps\_max\_num\_implicit\_qtbt\_before\_ot** | ue(v) |
| **gps\_min\_size\_implicit\_qtbt** | ue(v) |
| } |  |
| **gps\_extension\_present\_flag** | u(1) |
| if( gps\_extension\_present\_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** | ue(v) |
| **aps\_attr\_chroma\_qp\_offset** | se(v) |
| **aps\_slice\_qp\_delta\_present\_flag** | u(1) |
| LodParametersPresent = ( attr\_coding\_type = = 0 | | attr\_coding\_type = = 2 ) ? 1 : 0 |  |
| if( LodParametersPresent) { |  |
| **lifting\_num\_pred\_nearest\_neighbours** | ue(v) |
| **lifting\_search\_range\_minus1** | ue(v) |
| **lifting\_num\_detail\_levels\_minus1**  [Ed. The V7.0 code use the variable without minus1. It should be aligned] | ue(v) |
| for( k = 0; k < 3; k++ ) |  |
| **lifting\_neighbour\_bias**[ k ] | ue(v) |
| if ( attr\_coding\_type = = 2 ) |  |
| **lifting\_scalability\_enabled\_flag** | u(1) |
| if ( ! lifting\_scalability\_enabled\_flag ) { |  |
| **lifting\_lod\_regular\_sampling\_enabled\_flag** | u(1) |
| for( idx = 0; idx <= num\_detail\_levels\_minus1; idx++ ) { |  |
| if ( lifting\_lod\_decimation\_enabled\_flag ) |  |
| **lifting\_sampling\_period**[ idx ] | ue(v) |
| else |  |
| **lifting\_sampling\_distance\_squared**[ idx ] | ue(v) |
| } |  |
| } |  |
| if( attr\_coding\_type = = 0 ) { |  |
| **lifting\_adaptive\_prediction\_threshold** | ue(v) |
| **lifting\_intra\_lod\_prediction\_num\_layers** | ue(v) |
| **lifting\_max\_num\_direct\_predictors** | ue(v) |
| **inter\_component\_prediction\_enabled\_flag** | u(1) |
| } |  |
| } |  |
| if( attribute\_coding\_type = = 1 ) { //RAHT |  |
| **raht\_prediction\_enabled\_flag** | u(1) |
| **raht\_depth\_minus1** | ue(v) |
| } |  |
| **aps\_extension\_present\_flag** | u(1) |
| if( aps\_extension\_present\_flag ) |  |
| while( more\_data\_in\_byte\_stream( ) ) |  |
| **aps\_extension\_data\_flag** | u(1) |
| byte\_alignment( ) |  |
| } |  |

#### Frame boundary marker

|  |  |
| --- | --- |
| 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 payload syntax

#### General geometry slice syntax

|  |  |
| --- | --- |
| general\_geometry\_slice\_bitstream( ) { | Descriptor |
| geometry\_slice\_header( ) |  |
| geometry\_slice\_data( ) |  |
| } |  |

#### Geometry slice header syntax

|  |  |
| --- | --- |
| geometry\_slice\_header( ) { | Descriptor |
| **gsh\_geometry\_parameter\_set\_id** | ue(v) |
| **gsh\_tile\_id** | ue(v) |
| **gsh\_slice\_id** | ue(v) |
| **frame\_idx** | u(n) |
| if( gps\_box\_present\_flag ) { |  |
| if( gps\_gsh\_box\_log2\_scale\_present\_flag ) |  |
| **gsh\_box\_log2\_scale** | ue(v) |
| **gsh\_box\_origin\_x** | ue(v) |
| **gsh\_box\_origin\_y** | ue(v) |
| **gsh\_box\_origin\_z** | ue(v) |
| } |  |
| if ( gps\_implicit\_geom\_partition\_flag ) { |  |
| **gsh\_log2\_max\_nodesize\_x** | ue(v) |
| **gsh\_log2\_max\_nodesize\_y\_minus\_x** | se(v) |
| **gsh\_log2\_max\_nodesize\_z\_minus\_y** | se(v) |
| } else { |  |
| **gsh\_log2\_max\_nodesize** | ue(v) |
| } |  |
| **gsh\_num\_points** | ue(v) |
| if( geom\_scaling\_enabled\_flag ) {  [Ed: this should be last in the gsh?] |  |
| **geom\_slice\_qp\_offset** | se(v) |
| **geom\_octree\_qp\_offsets\_enabled\_flag** | u(1) |
| if( geom\_octree\_qp\_offsets\_enabled\_flag ) |  |
| **geom\_octree\_qp\_offsets\_depth** | ue(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Geometry slice data syntax

|  |  |
| --- | --- |
| geometry\_slice\_data( ) { | Descriptor |
| depthX = depthY = depthZ = 0; |  |
| for( depth = 0; depth < MaxGeometryOctreeDepth; depth++ ) { |  |
| for( nodeIdx = 0; nodeIdx < NumNodesAtDepth[ depth ]; nodeIdx++ ) { |  |
| xN = NodeX[ depthX ][ nodeIdx ] |  |
| yN = NodeY[ depthY ][ nodeIdx ] |  |
| zN = NodeZ[ depthZ ][ nodeIdx ] |  |
| geometry\_node( depthX, depthY, depthZ, partitionSkip, nodeIdx, xN, yN, zN ) |  |
| } |  |
| if ( !(partitionSkip & 4) ) |  |
| depthX = depthX + 1; |  |
| if ( !(partitionSkip & 2) ) |  |
| depthY = depthY + 1; |  |
| if ( !(partitionSkip & 1) ) |  |
| depthZ = depthZ + 1; |  |
| } |  |
| if ( log2\_trisoup\_node\_size > 0 ) |  |
| geometry\_trisoup\_data( ) |  |
| } |  |

#### Geometry node syntax

|  |  |  |
| --- | --- | --- |
| geometry\_node( depthX, depthY, depthZ, partitionSkip, nodeIdx, xN, yN, zN ) { | Descriptor | |
| if( depth = = GeomScalingDepth && geom\_octree\_qp\_offsets\_enabled\_flag) { |  | |
| **geom\_node\_qp\_offset\_eq0\_flag** [Ed. sense mismatch between m50924 and CE implementation] | 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 < MaxGeometryOctreeDepth ) { |  | |
| 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( EffectiveDepthX >= MaxNodeSizeXLog2 − 1 &&  EffectiveDepthY >= MaxNodeSizeYLog2 – 1 &&  EffectiveDepthZ >= MaxNodeSizeZLog2 – 1 ) {  [Ed. This used to be >= MaxGeometryOctreeDepth, it doesn't appear to make sense using NodeSizeLog2]  [Ed. Software simplifies this with isLeafNode] |  | |
| 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++ )  [Ed. move into geometry\_planar\_mode] |  | |
| if (eligible\_planar\_flag[axisIdx]) |  | |
| geometry\_planar\_mode\_data(child, axisIdx ) |  | |
| } |  | |
| if( DirectModeFlagPresent ) |  | |
| geometry\_direct\_mode\_data( 0 ) |  | |
| } |  | |
| } |  | |

#### Geometry trisoup data syntax

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

#### Single occupancy data syntax

|  |  |
| --- | --- |
| single\_occupancy(nodeIdx ) { | Descriptor |
| if ( !is\_planar\_flag [nodeIdx][0] || !is\_planar\_flag [nodeIdx][1]   || !is\_planar\_flag [nodeIdx][2] ) {  [Ed: elsewhere is\_planar\_flag[child][] is used ... nodeIdx != child] |  |
| 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(child, axisIdx ) { | Descriptor |
| **is\_planar\_flag** [ child ][axisIdx] | ae(v) |
| if( is\_planar\_0\_flag [ child ] [axisIdx] ) |  |
| **plane\_position** [ child ] [axisIdx] | ae(v) |
| } |  |

#### Direct mode data syntax

|  |  |
| --- | --- |
| geometry\_direct\_mode\_data(child) { | Descriptor |
| **direct\_mode\_flag** | ae(v) |
| if( direct\_mode\_flag ) { |  |
| **num\_direct\_points\_gt1** | ae(v) |
| if( !geom\_unique\_points\_flag && !num\_direct\_points\_gt1) { |  |
| **not\_duplicated\_point\_flag** |  |
| if( !not\_duplicated\_point\_flag ) { |  |
| **num\_direct\_points\_eq2\_flag** |  |
| if( num\_direct\_points\_eq2\_flag) |  |
| **num\_points\_direct\_mode\_minus3** |  |
| } |  |
| } |  |
| for( i = 0; i <= num\_direct\_different\_points\_minus1; i++ ){ |  |
| if(ChildNodeSizeXLog2>=1 && (!**is\_planar\_flag** [child ][0] || partitionSkip & 4)) |  |
| **point\_offset\_x**[ i ][ 0 ] | ae(v) |
| for( j = 1; j < EffectiveChildNodeSizeXLog2; j++ ) |  |
| **point\_offset\_x**[ i ][ jx ] | ae(v) |
| if(ChildNodeSizeYLog2>=1 && (!**is\_planar\_flag** [child ][1] || partitionSkip & 2)) |  |
| **point\_offset\_y**[ i ][ 0 ] | ae(v) |
| for( j = 1; j < EffectiveChildNodeSizeYLog2; j++ ) |  |
| **point\_offset\_y**[ i ][ jy ] | ae(v) |
| if(ChildNodeSizeZLog2>=1 && (!**is\_planar\_flag** [child ][2] || partitionSkip & 1)) |  |
| **point\_offset\_z**[ i ][ 0 ] | ae(v) |
| for( j = 1; j < EffectiveChildNodeSizeZLog2; j++ ) |  |
| **point\_offset\_z**[ i ][ j ] | ae(v) |
| } |  |
| } |  |
| } |  |

### Attribute payload syntax

#### General attribute slice syntax

|  |  |
| --- | --- |
| general\_attribute\_slice\_bitstream( ) { | Descriptor |
| attribute\_slice\_header( ) |  |
| attribute\_slice\_data( ) |  |
| } |  |

#### Attribute slice header syntax

|  |  |
| --- | --- |
| attribute\_slice\_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\_delta\_present\_flag ) { |  |
| **ash\_attr\_qp\_delta\_luma** | se(v) |
| **ash\_attr\_qp\_delta\_chroma** | se(v) |
| } |  |
| **ash\_attr\_layer\_qp\_delta\_present\_flag** | u(1) |
| if ( ash\_attr\_layer\_qp\_delta\_present\_flag ) { |  |
| numLayers = aps.attr\_encoding =  = 1 ? aps.raht\_depth + 1 : aps.num\_detail\_levels + 1 |  |
| for( i = 0; i < numLayers; i++ ){ |  |
| **ash\_attr\_layer\_qp\_delta\_luma[i]** | se(v) |
| **ash\_attr\_layer\_qp\_delta\_chroma[i]** | se(v) |
| } |  |
| } |  |
| **ash\_attr\_region\_qp\_delta\_present\_flag** | u(1) |
| if ( ash\_attr\_region\_qp\_delta\_present\_flag ) { |  |
| **ash\_attr\_qp\_region\_box\_origin\_x** | ue(v) |
| **ash\_attr\_qp\_region\_box\_origin\_y** | ue(v) |
| **ash\_attr\_qp\_region\_box\_origin\_z** | ue(v) |
| **ash\_attr\_qp\_region\_box\_width** | ue(v) |
| **ash\_attr\_qp\_region\_box\_height** | ue(v) |
| **ash\_attr\_qp\_region\_box\_depth** | ue(v) |
| **ash\_attr\_region\_qp\_delta** | se(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Attribute slice data syntax

|  |  |
| --- | --- |
| attribute\_slice\_data( ) { | Descriptor |
| *dimension* = attribute\_dimension[ ash\_attr\_sps\_attr\_idx ] |  |
| **zerorun** | ae(v) |
| for( i = 0; i < PointCount; i++ ) { |  |
| if( attr\_coding\_type = = 0 &&  maxPredDiff[ i ] >= lifting\_adaptive\_prediction\_threshold &&  MaxNumPredictors > 1 ) { |  |
| **predIndex**[ i ] | ae(v) |
| } |  |
| if( zerorun > 0 ) { |  |
| for( k = 0; k < *dimension* ; k++ ) |  |
| values[ k ][ i ] = 0 |  |
| zerorun −= 1 |  |
| } |  |
| else { |  |
| attribute\_coding( *dimension*, *i* ) | ae(v) |
| **zerorun** | ae(v) |
| } |  |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Quantized value bitstream syntax

|  |  |
| --- | --- |
| attribute\_coding( dimension, pointIdx ) { | Descriptor |
| for ( k = 0; k < *dimension*; k++ ) { |  |
| **isZero**[ k ] | ae(v) |
| if ( isZero[ k ] = = 1 ) |  |
| values[ k ][ pointIdx ] = 0 |  |
| else { |  |
| **isOne**[ k ] | ae(v) |
| if ( isOne[ k ] = = 1 ) |  |
| values[ k ][ pointIdx ] = 1 |  |
| else { |  |
| **values**[ k ][ pointIdx ] | de(v) |
| if (values[ k ][ pointIdx ] = = 255 ) { |  |
| **remaining\_values**[ k ][ pointIdx ] | ae(v) |
| values[ k ][ pointIdx ] += remaining\_values[ k ][ pointIdx ] |  |
| } |  |
| values[ k ][ pointIdx ] += 2 |  |
| } |  |
| } |  |
| for( d = 1, k=1; k < *dimension*; k++ ) |  |
| if( values[ k ][ pointIdx ] != values[ 0 ][ pointIdx ] ) |  |
| d = 0 |  |
| for( k = 0; k < *dimension* ; k++ ) |  |
| values[ k ][ pointIdx ] += d |  |
| } |  |

## 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.

### Payload and byte alignment semantics

#### Sequence parameter set semantics

**reserved\_profile\_compatibility\_23bits** shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for reserved\_profile\_compatibility\_23bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of reserved\_profile\_compatibility\_23bits.

**unique\_point\_positions\_constraint\_flag** eachsto sunique\_point\_positions\_constraint\_flag equal to 0 anysto the

unique\_point\_positions\_constraint\_flag

**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\_bounding\_box\_present\_flag** equal to 1 indicates the source bounding box offset and the size information is signalled in the SPS. sps\_bounding\_box\_present\_flag equal to 0 indicates the source bounding box information is not signalled.

**sps\_bounding\_box\_offset\_x** indicates the x offset of the source bounding box in the cartesian coordinates. When not present, the value of sps\_bounding\_box\_offset\_x is inferred to be 0.

**sps\_bounding\_box\_offset\_y** indicates the y offset of the source bounding box in the cartesian coordinates. When not present, the value of sps\_bounding\_box\_offset\_y is inferred to be 0.

**sps\_bounding\_box\_offset\_z** indicates the z offset of the source bounding box in the Cartesian coordinates. When not present, the value of sps\_bounding\_box\_offset\_z is inferred to be 0.

**sps\_bounding\_box\_scale\_factor** indicates the scale factor the source bounding box in the Cartesian coordinates. When not present, the value of sps\_bounding\_box\_scale\_factor is inferred to be 1.

**sps\_bounding\_box\_size\_width** indicates the width of the source bounding box in the Cartesian coordinates. When not present, the value of sps\_bounding\_box\_size\_width is inferred to be 1.

**sps\_bounding\_box\_size\_height** indicates the height of the source bounding box in the Cartesian coordinates. When not present, the value of sps\_bounding\_box\_size\_height is inferred to be 1.

**sps\_bounding\_box\_size\_depth** indicates the depth of the source bounding box in the Cartesian coordinates. When not present, the value of sps\_bounding\_box\_size\_depth is inferred to be 1.

**sps\_source\_scale\_factor** indicates the scale factor of the source point cloud.

**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\_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.

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

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

**attribute\_bitdepth**[i]specifies the bitdepth of the i-th attribute signal(s).

**attribute\_cicp\_colour\_primaries**[ i ]indicates the chromaticity coordinates of the colour attribute source primaries of the i-th attribute.

**attribute\_cicp\_transfer\_characteristics**[i ]either indicates the reference opto-electronic transfer characteristic function of the colour attribute as a function of a source input linear optical intensity Lc with a nominal real-valued range of 0 to 1 or indicates the inverse of the reference electro-optical transfer characteristic function as a function of an output linear optical intensity Lo with a nominal real-valued range of 0 to 1.

**attribute\_cicp\_matrix\_coeffs**[ i ]describes the matrix coefficients used in deriving luma and chroma signals from the green, blue, and red, or Y, Z, and X primaries.

**attribute\_cicp\_video\_full\_range\_flag**[ i ]specifies indicates the black level and range of the luma and chroma signals as derived from E′Y, E′PB, and E′PR or E′R, E′G, and E′B real-valued component signals.

**known\_attribute\_label\_flag**[ i ] equal to 1 specifies know\_attribute\_label is signalled for the i-th attribute. known\_attribute\_label\_flag[ i ] equal to 0 specifies attribute\_label\_four\_bytes is signalled for the i-th attribute.

**known\_attribute\_label**[ i ] equal to 0 specifies the attribute is colour. known\_attribute\_label[i] equal to 1 specifies the attribute is reflectance. known\_attribute\_label[ i ] equal to 2 specifies the attribute is frame index.

**attribute\_label\_four\_bytes**[ i ]indicates the known attribute type with the 4 bytes code. Table 6 describes the list of supported attributes and their relationship with attribute\_label\_four\_bytes[ i ].

Table 6 — attribute\_label\_four\_bytes

|  |  |
| --- | --- |
| attribute\_label\_four\_bytes[ i ] | Attribute type |
| 0 | Colour |
| 1 | Reflectance |
| 2 | Frame index |
| 0xffffffff | unspecified |

**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 RecPic[ pointIdx ][ axis ] with axis = 0 .. 2.

Table 7 — Mapping of output X, Y, and Z axis labels to indicies axis of RecPic[ i ][ axis ]

|  |  |  |  |
| --- | --- | --- | --- |
| axis\_coding\_order | X | Y | Z |
| 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 |

**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\_extension\_present\_flag** equal to 1 specifies that the sps\_extension\_data syntax structure is present in the SPS syntax structure. sps\_extension\_present\_flag equal to 0 specifies that this syntax structure is not present. When not present, the value of sps\_extension\_present\_flag is inferred to be equal to 0.

**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 a profile specified in Annex A.

#### Tile inventory syntax

**num\_tiles** specifies the number of tiles signalled for the bitstream. When not present, num\_tiles is inferred to be 0.

**tile\_bounding\_box\_offset\_x**[ i ]indicates the x offset of the i-th tile in the cartesian coordinates. When not present, the value of tile\_bounding\_box\_offset\_x[ 0 ] is inferred to be sps\_bounding\_box\_offset\_x.

**tile\_bounding\_box\_offset\_y**[ i ]indicates the y offset of the i-th tile in the cartesian coordinates. When not present, the value of tile\_bounding\_box\_offset\_y[ 0 ] is inferred to be sps\_bounding\_box\_offset\_y.

**tile\_bounding\_box\_offset\_z**[ i ] indicates the z offset of the i-th tile in the Cartesian coordinates. When not present, the value of tile\_bounding\_box\_offset\_z[ 0 ] is inferred to be sps\_bounding\_box\_offset\_z.

**tile\_bounding\_box\_size\_width**[ i ] indicates the width of the i-th tile in the Cartesian coordinates. When not present, the value of tile\_bounding\_box\_size\_width[ 0 ] is inferred to be sps\_bounding\_box\_size\_width.

**tile\_bounding\_box\_size\_height**[ i ] indicates the height of the i-th tile in the Cartesian coordinates. When not present, the value of tile\_bounding\_box\_size\_height[ 0 ] is inferred to be sps\_bounding\_box\_size\_height.

**tile\_bounding\_box\_size\_depth**[ i ] indicates the depth of the i-th tile in the Cartesian coordinates. When not present, the value of tile\_bounding\_box\_size\_depth[ 0 ] is inferred to be sps\_bounding\_box\_size\_depth.

#### 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\_box\_present\_flag** equal to 1 specifies an additional bounding box information is provided in a geometry header that references the current GPS. gps\_bounding\_box\_present\_flagequal to 0 specifies that additional bounding box information is not signalled in the geometry header.

**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\_gsh\_box\_log2\_scale** indicates the common scale factor of bounding box origin for all slices that references 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…127 .

**geom\_planar\_mode\_th[i]**, for i in the rang 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 .

**neighbour\_context\_restriction\_flag** equal to 0 indicates that octree occupancy coding uses contexts determined from six neighbouring parent nodes. neighbour\_context\_restriction\_flag equal to 1 indicates that octree coding uses contexts determined from sibling nodes only.

**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 bitwise 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 bitwise 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.

[Ed.(ON): Should the syntax log2\_neighbour\_avail\_boundary be sent if neighbour\_context\_restriction\_flag is 0?]

**log2\_intra\_pred\_max\_node\_size** specifies the octree nodesize 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 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.

**geom\_base\_qp** specifies the base value of the geometry position quantization parameter. When not present, geom\_base\_qp is inferred to be equal to 4.

**gps\_implicit\_geom\_partition\_flag** equal to 1 specifies that the implicit geometry partition is enabled for the sequence or slice. gps\_implicit\_geom\_partition\_flag equal to 0 specifies that the implicit geometry partition is disabled for the sequence or slice. If gps\_implicit\_geom\_partition\_flag equals to 1, the following two parameters gps\_max\_num\_implicit\_qtbt\_before\_otand gps\_min\_size\_implicit\_qtbt are signaled.

**gps\_max\_num\_implicit\_qtbt\_before\_ot** specifies the maximal number of implicit QT and BT partitions before OT partitions. The variable K is then initialized by

K = gps\_max\_num\_implicit\_qtbt\_before\_ot.

**gps\_min\_size\_implicit\_qtbt** specifies the minimal size of implicit QT and BT partitions. The variable M is then initialized by

M = gps\_min\_size\_implicit\_qtbt.

**gps\_extension\_present\_flag** equal to 1 specifies that the gps\_extension\_data syntax structure is present in the GPS syntax structure. gps\_extension\_present\_flag equal to 0 specifies that this syntax structure is not present. When not present, the value of gps\_ extension\_present\_flag is inferred to be equal to 0.

**gps\_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 a profile specified in Annex A.

#### 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 7 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 7 — Interpretation of attr\_coding\_type

|  |  |
| --- | --- |
| attr\_coding\_type | coding type |
| 0 | Predicting Weight Lifting |
| 1 | Region Adaptive Hierarchical Transform (RAHT) |
| 2 | Fix Weight Lifting |

[Ed. Need to define the consistent name for Predicting and Lifting Transform]

**aps\_attr\_initial\_qp** specifies the initial value of the variable SliceQp for each slice referring to the APS. The initial value of SliceQp is modified at the attribute slice segment layer when a non-zero value of slice\_qp\_delta\_luma or slice\_qp\_delta\_luma are decoded. The value of aps\_attr\_initial\_qp shall be in the range of 0 to 52, inclusive.

**aps\_attr\_chroma\_qp\_offset** specifies the offsets to the initial quantization parameter signalled by the syntax aps\_attr\_initial\_qp.

**aps\_slice\_qp\_delta\_present\_flag** equal to 1 specifies that the ash\_attr\_qp\_delta\_luma and ash\_attr\_qp\_delta\_chromachromachromachroma syntax elements are present in the ASH. aps\_slice\_qp\_present\_flag equal to 0 specifies that the ash\_attr\_qp\_delta\_luma and ash\_attr\_qp\_delta\_chroma syntax elements are not present in the ASH.

**lifting\_num\_pred\_nearest\_neighbours** 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.

**lifting\_neighbour\_bias**[ k ] specifies a bias used to weight the k-th components in the calculation of the euclidean distance between two points as part of the nearest neighbour derivation process.

When not present, the value of lifting\_scalability\_enabled\_flag is inferred to be equal to 0. the value of

**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 value of lifting\_search\_range is derived as follows:

lifting\_search\_range = lifting\_search\_range\_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**[ idx ]specifies the sampling period for the level of detail idx. The value of lifting\_sampling\_period[ ] shall be in the range of 0 to xx.

**lifting\_sampling\_distance\_squared**[ idx ]specifies the square of the sampling distance for the level of detail idx. The value of lifting\_sampling\_distance\_squared[ ] shall be in the range of 0 to xx.

**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.

**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 num\_detail\_levels\_minus1 plus 1 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 lifting\_num\_detail\_levels\_minus1 plus 1.

**lifting\_max\_num\_direct\_predictors** specifies the maximum number of predictorspredictor to be used for direct prediction. The value of lifting\_max\_num\_direct\_predictors shall be range of 0 to lifting\_num\_pred\_nearest\_neighbours.

The value of the variable MaxNumPredictors that is used in the decoding process as follows:

MaxNumPredictors = lifting\_max\_num\_direct\_predictors + 1

**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.

**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\_depth\_minus1** specifies the number of levels minus 1 of detail for RAHT. The value of depthRAHTshall be in the range of 0 to xx.

**aps\_extension\_present\_flag** equal to 1 specifies that the aps\_extension\_data syntax structure is present in the APS syntax structure. aps\_extension\_present\_flag equal to 0 specifies that this syntax structure is not present. When not present, the value of aps\_ extension\_present\_flag is inferred to be equal to 0.

**aps\_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 a profile specified in Annex A.

#### Frame boundary marker

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 payload semantics

#### General geometry slice semantics

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

The variables NodeX[ depthX ][ idx ], NodeY[ depthY ][ idx ], and NodeZ[ depthZ ][ idx ] represent the x, y, and z 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 depthX, depthY and depthZ specify respectively the depth in x, y and z dimensions. The variable partitionSkip specifies the partition type and direction as in Table 8. The variable partitionSkip is represented in binary form with three bits , which specify respectively whether to skip partition along x, y and z dimension.

Table 8— Interpretation of partitionSkip

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Partition** | **QT along x-y axes** | **QT along x-z axes** | **QT along y-z axes** | **OT** |
| partitionSkip | b001 | b010 | b100 | b000 |
| **Partition** | **BT along x axis** | **BT along y axis** | **BT along z axis** |  |
| partitionSkip | b011 | b101 | b110 |  |

The variables NodeX, NodeY, NodeZ, NumNodesAtDepth, and GeometryNodeOccupancyCnt are initialised as follows:

NodeX[ 0 ][ 0 ] = NodeY[ 0 ][ 0 ] = NodeZ[ 0 ][ 0 ] = 0

NumNodesAtDepth[ 0 ] = 1

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

#### Geometry slice header semantics

**gsh\_geometry\_parameter\_set\_id** specifies the value of the gps\_geom\_parameter\_set\_id of the active GPS.

**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.

**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\_box\_log2\_scale** specifies the scaling factor of bounding box origin for the slice.

**gsh\_box\_origin\_x** specifies the x value of bounding box origin that scaled by gsh\_box\_log2\_scale value.

**gsh\_box\_origin\_y** specifies the y value of bounding box origin that scaled by gsh\_box\_log2\_scale value

**gsh\_box\_origin\_z** specifies the z value of bounding box origin that scaled by gsh\_box\_log2\_scale value.

The variable slice\_origin\_x, slice\_origin\_y, and slice\_origin\_z are derived as follows:

If gps\_gsh\_box\_log2\_scale\_present\_flag is equal to 0,

origin\_scale is set equal to gsh\_box\_log2\_scale

Otherwise ( gps\_gsh\_box\_log2\_scale\_present\_flag is equal to 1 ),

origin\_scale is set equal to gps\_gsh\_box\_log2\_scale

If gps\_box\_present\_flag is equal to 0,

the value of slice\_origin\_x and slice\_origin\_y and slice\_origin\_z are inferred to be 0.

Othersise (gps\_box\_present\_flag is equal to 1), the following applies:

slice\_origin\_x = gsh\_box\_origin\_x << origin\_scale

slice\_origin\_y = gsh\_box\_origin\_x << origin\_scale

slice\_origin\_z = gsh\_box\_origin\_x << origin\_scale

**gsh\_log2\_max\_nodesize\_x** specifies the bounding box size in the x dimension, i.e., MaxNodesizeXLog2 that is used in the decoding process as follows.

MaxNodeSizeXLog2 = gsh\_log2\_max\_nodesize\_x

MaxNodeSizeX = 1 << MaxNodeSizeXLog2

**gsh\_log2\_max\_nodesize\_y\_minus\_x** specifies the bounding box size in the y dimension, i.e., MaxNodesizeYLog2 that is used in the decoding process as follows:

MaxNodeSizeYLog2 = gsh\_log2\_max\_nodesize\_y\_minus\_x + MaxNodeSizeXLog2.

MaxNodeSizeY = 1 << MaxNodeSizeYLog2.

**gsh\_log2\_max\_nodesize\_z\_minus\_y** specifies the bounding box size in the z dimension, i.e., MaxNodesizeZLog2 that is used in the decoding process as follows.

MaxNodeSizeZLog2 = gsh\_log2\_max\_nodesize\_z\_minus\_y + MaxNodeSizeYLog2

MaxNodeSizeZ = 1 << MaxNodeSizeZLog2

If gps\_implicit\_geom\_partition\_flag equals to 1, gsh\_log2\_max\_nodesize is derived as follows.

gsh\_log2\_max\_nodesize = max{ MaxNodeSizeXLog2, MaxNodeSizeYLog2, MaxNodeSizeZLog2}

**gsh\_log2\_max\_nodesize** specifies the size of the root geometry octree node when gps\_implicit\_geom\_partition\_flag is equal to 0. The variables MaxNodeSize, and MaxGeometryOctreeDepth are derived as follows.

MaxNodeSize = 1 << gsh\_log2\_max\_nodesize

MaxGeometryOctreeDepth = gsh\_log2\_max\_nodesize − log2\_trisoup\_node\_size

The variables K and M are then updated as follows.

gsh\_log2\_min\_nodesize = min{ MaxNodeSizeXLog2, MaxNodeSizeYLog2, MaxNodeSizeZLog2}

if (K > (gsh\_log2\_max\_nodesize - gsh\_log2\_min\_nodesize))

K = gsh\_log2\_max\_nodesize - gsh\_log2\_min\_nodesize;

if (M > gsh\_log2\_min\_nodesize)

M = gsh\_log2\_min\_nodesize;

if (gsh\_log2\_max\_nodesize == gsh\_log2\_min\_nodesize)

M = 0;

if (log2\_trisoup\_node\_size != 0) {

K = gsh\_log2\_max\_nodesize - gsh\_log2\_min\_nodesize;

M = 0;

}

**gsh\_num\_points** specifies the number of coded points in the slice.

**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 equal to 0.

**geom\_octree\_qp\_offsets\_enabled\_flag** equal to 1 specifies that geom\_node\_qp\_offset\_eq0\_flag may be present in the geometry node syntax. geom\_octree\_qp\_offsets\_enabled\_flag equal to 0 specifies that geom\_node\_qp\_offset\_eq0\_flag is not present in the geometry node syntax. When not present, geom\_octree\_qp\_offsets\_enabled\_flag is inferred to be equal to 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. [Ed. refine]

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

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

#### Geometry slice data semantics

The process to derive the variable partitionSkip is specified from here. The input of the process are varibales K, M, depth, depthX, depthY and depthZ. The output of the process is the value of partitionSkip. The process to derive the variable partitionSkip proceeds as follows.

partitionSkip = 0

NodeSizeXLog2 = MaxNodeSizeXLog2 – depthX

NodeSizeYLog2 = MaxNodeSizeYLog2 – depthY

NodeSizeZLog2 = MaxNodeSizeZLog2 – depthZ

MinNodeSizeLog2 = min{ NodeSizeXLog2, NodeSizeYLog2, NodeSizeZLog2}

MaxNodeSizeLog2 = max{ NodeSizeXLog2, NodeSizeYLog2, NodeSizeZLog2}

If (MinNodeSizeLog2 == MaxNodeSizeLog2)

M = 0

if (K > depth || M == MinNodeSizeLog2) {

if (NodeSizeXLog2 < MaxNodeSizeLog2)

partitionSkip |= 4

if (NodeSizeYLog2 < MaxNodeSizeLog2)

partitionSkip |= 2

if (NodeSizeZLog2 < MaxNodeSizeLog2)

partitionSkip |= 1

}

The parameter M prevents implicit QT and BT partitions when all dimensions are smaller than or equal to M.

#### 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. Nodes at depth gbh\_log2\_max\_node\_size are leaf nodes.

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

The variables xPn, yPn, and zPn indicating the position of the current node's parent node at depth − 1 are derived as follows:

The variables NodeSizeLog2 and ChildNodeSizeLog2 are derived as follows:

When depth is equal to GeomScalingDepth and nodeIdx is equal to 0, the array ScalingNodeSizeLog2 is derived as follows:

ScalingNodeSizeLog2[ 0 ] = NodeSizeXLog2

ScalingNodeSizeLog2[ 1 ] = NodeSizeYLog2

ScalingNodeSizeLog2[ 2 ] = NodeSizeZLog2.

The variable NeighbourPattern is derived as follows:

* For each node, the variables rN, lN, fN, bN, uN, and dN are derived as follows:
* If NeighbAvailabilityMask is not equal to 0, the following applies.
* If adjacent\_child\_contextualization\_enabled\_flag is equal to 1, the following applies.

for (xNc = xN × 2; xNc < xN × 2 + 2; xNc++){

for (yNc = yN × 2; yNc < yN × 2 + 2; yNc++){

for (zNc = zN × 2; zNc < zN × 2 + 2; zNc++) {

}

}

}

lN &= lNadj

fN &= fNadj

dN &= dNadj

* Finally, the variable NeighbourPattern is set as follows:

**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, depth is less than GeomScalingDepth, NodeQp is set equal to 4.

The variables EffectiveChildNodeSizeLog2 and EffectiveDepth are derived as follows:

EffectiveChildNodeSizeLog2 = ChildNodeSizeLog2 - ( NodeQp - 4 ) / 6

EffectiveDepth = depth + ( NodeQp - 4 ) / 6

**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 []** 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 **[]** as described in 9.5.4.

**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 MaxGeometryOctreeDepth, 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:

The variable childCnt is initialized to 0.

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

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

continue

GeometryNodeChildren[ childCnt++ ] = childIdx

}

GeometryNodeChildrenCnt = childCnt

GeometryNodeOccupancyCnt[ depth ][ xN ][ yN ][ zN ] = 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
    - EffectiveNodeSizeLog2 is greater than 1
    - GeometryNodeOccupancyCnt[ depth − 1 ][ xPn ][ yPn ][ zPn ] is less than or equal to 2
    - GeometryNodeOccupancyCnt[ depth ][ xN ][ yN ][ zN ] 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 **8.2.4.5** .

**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.

**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 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.

**is\_planar\_flag** [ child ] [axisIdx] equal 1 indicates that the current child node is planar in the direction perpendicular to the axisIdx -th axis (the x axis being the 0-th axis; the y axis being the 1st axis; the z axis being the 2nd axis). is\_planar\_flag [ child ] [axisIdx] equal 0 indicates that the current child node is not planar in the direction perpendicular to the i-th axis. When not present, the value of is\_planar\_flag [ child ] [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**[ child ] [axisIdx] equal 0 indicates that the position of the plane for the planar mode is the lower position relative to increasing i-th coordinates. plane\_position[ child ] [axisIdx] equal 1 indicates that the position of the plane for the planar mode is the higher position relative to increasing axisIdx-th coordinates.

**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 coordinates. 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.

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

nodeIdx = NumNodesAtDepth[ depth + 1 ]

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

childIdx = GeometryNodeChildren[ child ]

x = NodeX[ depth + 1 ][ nodeIdx ] = 2 × xN + ( childIdx & 4 = = 1 )

y = NodeY[ depth + 1 ][ nodeIdx ] = 2 × yN + ( childIdx & 2 = = 1 )

z = NodeZ[ depth + 1 ][ nodeIdx ] = 2 × zN + ( childIdx & 1 = = 1 )

NodeQpMap[ depth + 1 ][ nodeIdx ] = NodeQp

GeometryNodeOccupancyCnt[ depth + 1 ][ x ][ y ][ z ] = 1

nodeIdx++

}

NumNodesAtDepth[ depth + 1 ] = nodeIdx

**num\_direct\_points\_gt1**equal to 0 indicates that there is one point in the current child node or that all points in the current child node have the same x, y and z coordinates. num\_direct\_points\_gt1 equal to 1 indicates that there are at leats two points in the current child node with different x, y or z coordinates .

**not\_duplicated \_point\_flag** equal to 0 indicates that all points in the current child node have the same x, y and z coordinates. not\_duplicated\_point\_flagequal to 1 indicates that at least two points in the current child node have different x, y or z coordinates. When not present, the value of not\_duplicated\_point\_flag is inferred equal to 1.

duplicated\_point\_flagis derived as the negation of **not\_duplicated\_point\_flag** as follows

duplicated\_point\_flag= ! not\_duplicated\_point\_flag

**num\_direct\_points\_eq2\_flag** equal to 1 indicates that there are two points in the current child node. num\_direct\_points\_eq2\_flag equal to 0 indicates that theres are at least three points in the current child node.

**num\_points\_direct\_mode\_minus3** plus 3 indicates the number of points in the current child node.

num\_direct\_points\_minus1 plus 1 indicates the number of points in the current child node. The variable num\_direct\_points\_minus1 is derived as follows

num\_direct\_points\_minus1 = 0

if ( num\_direct\_points\_gt1) {

num\_direct\_points\_minus1 = 1

if (duplicated\_point\_flag && !num\_direct\_points\_eq2\_flag)

num\_direct\_points\_minus1 = 2 + num\_points\_direct\_mode\_minus3

}

}

num\_direct\_different\_points\_minus1 plus 1 is the number of points having at least one different x, y or z coordinate in the current child node. The variable num\_direct\_different\_points\_minus1 is derived as follows

num\_direct\_different\_points\_minus1 = num\_direct\_points\_minus1

if ( duplicated\_point\_flag)

num\_direct\_different\_points\_minus1 = 0

The variables ChildNodeSizeXLog2, ChildNodeSizeYLog2 and ChildNodeSizeZLog2 specify the child node size in each dimension x, y or z respectively, and are determined by implicit QT and BT partitions as follows.

NodeSizeXLog2 = MaxNodeSizeXLog2 – depthX;

NodeSizeYLog2 = MaxNodeSizeYLog2 – depthY;

NodeSizeZLog2 = MaxNodeSizeZLog2 – depthZ;

if( !(partitionSkip & 4 )

ChildNodeSizeXLog2 = NodeSizeXLog2 – 1;

else

ChildNodeSizeXLog2 = NodeSizeXLog2;

if( !(partitionSkip & 2 )

ChildNodeSizeYLog2 = NodeSizeYLog2 – 1;

else

ChildNodeSizeYLog2 = NodeSizeYLog2;

if( !(partitionSkip & 1 )

ChildNodeSizeZLog2 = NodeSizeZLog2 – 1;

else

ChildNodeSizeZLog2 = NodeSizeZLog2;

**point\_offset\_x**[ i ][ j ], **point\_offset\_y**[ i ][ j ], and **point\_offset\_z**[ i ][ j ]indicate the j-th bit of the current child node's i-th point's respective x, y, and z co-ordinates relative to the origin of the child node identified by the index GeometryNodeChildren[ 0 ].

When point\_offset\_x[ i ][ 0 ] is not present because the condition (!**is\_planar\_flag** [child ][0] || partitionSkip & 4) is false, the value of point\_offset\_x[ i ][ 0 ] is inferred by the plane position **plane\_position** [ child  ] [0].

When point\_offset\_y[ i ][ 0 ] is not present because the condition (!**is\_planar\_flag** [child ][1] || partitionSkip & 2) is false, the value of point\_offset\_y[ i ][ 0 ] is inferred by the plane position **plane\_position** [ child  ] [1].

When point\_offset\_z[ i ][ 0 ] is not present because the condition (!**is\_planar\_flag** [child ][2] || partitionSkip & 1) is false, the value of point\_offset\_z[ i ][ 0 ] is inferred by the plane position **plane\_position** [ child  ] [2].

The variables PointOffsetX[ i ], PointOffsetY[ i ], and PointOffsetZ[ i ] are derived as follows:

PointOffsetX[ i ] = PointOffsetY[ i ] = PointOffsetZ[ i ] = 0;

for( j = 0; j < EffectiveChildNodeSizeXLog2; j++ ) {

PointOffsetX[ i ] += point\_offset\_x[ i ][ j ] << j;

}

for( j = 0; j < EffectiveChildNodeSizeYLog2; j++ ) {

PointOffsetY[ i ] += point\_offset\_y[ i ][ j ] << j;

}

for( j = 0; j < EffectiveChildNodeSizeZLog2; j++ ) {

PointOffsetZ[ i ] += point\_offset\_z[ i ][ j ] << j;

}

#### Geometry trisoup data semantics

**num\_unique\_segments** specifies the number of segment indicators.

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

**num\_vertices** specifies the number of vertices.

**vertex\_position**[ i ]indicates the position of the vertex along the edge.

### Attribute payload semantics

#### General attribute slice semantics

#### Attribute slice header semantics

**abh\_attr\_parameter\_set\_id** specifies the value of the aps\_attr\_parameter\_set\_id of the active APS.

**abh\_attr\_sps\_attr\_idx** specifies the order of attribute set in the active SPS. The value of abh\_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\_delta\_present\_flag** equal to 1 specifies that the ash\_attr\_layer\_qp\_delta\_luma and ash\_attr\_layer\_qp\_delta\_chroma syntax elements are present in current ASH. ash\_attr\_layer\_qp\_delta\_present\_flag equal to 0 specifies that the ash\_attr\_layer\_qp\_delta\_luma and ash\_attr\_layer\_qp\_delta\_chroma syntax elements are not present in current ASH.

**ash\_attr\_qp\_delta\_luma** specifies the luma delta qp from the initial slice qp in the active attribute parameter set. When ash\_attr\_qp\_delta\_luma is not signalled, the value of ash\_attr\_qp\_delta\_luma is inferred to be 0.

**ash\_attr\_qp\_delta\_chroma** specifies the chroma delta qp from the initial slice qp in the active attribute parameter set. When ash\_attr\_qp\_delta\_chroma is not signalled, the value of ash\_attr\_qp\_delta\_chroma is inferred to be 0.

The variables InitialSliceQpY and InitialSliceQpC are derived as follows:

InitialSliceQpY = aps\_attrattr\_initial\_qp + ash\_attr\_qp\_delta\_luma

InitialSliceQpC = aps\_attrattr\_initial\_qp + aps\_attr\_chroma\_qp\_offset + ash\_attr\_qp\_delta\_chroma

**ash\_attr\_layer\_qp\_delta\_luma** specifies the luma delta qp from the InitialSliceQpY in each layer. When ash\_attr\_layer\_qp\_delta\_luma is not signalled, the value of ash\_attr\_layer\_qp\_delta\_luma of all layers are inferred to be 0.

**ash\_attr\_layer\_qp\_delta\_chroma** specifies the chroma delta qp from the InitialSliceQpC in each layer. When ash\_attr\_layer\_qp\_delta\_chroma is not signalled, the value of ash\_attr\_layer\_qp\_delta\_chroma of all layers are inferred to be 0.

The variables SliceQpY[ i ] and SliceQpC[ i ] with i = 0…num\_layer − 1 are derived as follows:

for ( i = 0; i < num\_layer; i++) {

SliceQpY[ i ] = InitialSliceQpY + ash\_attr\_layer\_qp\_delta\_luma[ i ]

SliceQpC[ i ] = InitialSliceQpC + ash\_attr\_layer\_qp\_delta\_chroma[ i ]

}

Here, the variable of num\_layer is derived as follows with using attr\_coding\_type, num\_detail\_levels\_minus1 and raht\_depth\_minus1 in APS:

if ( ash\_attr\_layer\_qp\_delta\_present\_flag )

num\_layer = ( attr\_coding\_type = = 0 | | attr\_coding\_type = = 2 ) ?

num\_detail\_levels\_minus1 + 1 : raht\_depth\_minus1 + 1;

else

num\_layer = 1;

The value of SliceQpY[ i ] and SliceQpC[ i ] with i = 0…num\_layer – 1 shall be in the range of 4 to 51, inclusive.

[Ed(On): the QP range should be clarified to handle a N-bitdepth attribute. The current upper bound value 51 is for the 8bit attribute. Broder range may be needed for 10 or 16 bitdepth attribute]

[Ed(df): derivation of Qstep should happen much later in the decoding process]

The variable SliceQstepY[ i ] and SliceQstepC[ i ] with i = 0…num\_layer – 1 are derived as follows:

SliceQstepY[ i ] = QpToQstep( SliceQpY[ i ] )

SliceQstepC[ i ] = QpToQstep( SliceQpC[ i ] )

**ash\_attr\_region\_qp\_delta\_present\_flag** equal to 1 indicates the ash\_attr\_region\_qp\_delta and region bounding box origin and size are present in current ASH. ash\_attr\_region\_qp\_delta\_present\_flag equal to 0 indicates the ash\_attr\_region\_qp\_delta and region bounding box origin and size are not present in current ASH.

**ash\_attr\_qp\_region\_box\_origin\_x** indicates the x offset of the region bounding box relative to slice\_origin\_x. When not present, the value of ash\_attr\_qp\_region\_box\_origin\_x is inferred to be 0.

**ash\_attr\_qp\_region\_box\_origin\_y** indicates the y offset of the region bounding box relative to slice\_origin\_y. When not present, the value of ash\_attr\_qp\_region\_box\_origin\_y is inferred to be 0.

**ash\_attr\_qp\_region\_box\_origin\_z** indicates the z offset of the region bounding box relative to slice\_origin\_z. When not present, the value of ash\_attr\_qp\_region\_box\_origin\_z is inferred to be 0.

regionboxX = ash\_attr\_qp\_region\_box\_origin\_x

regionboxY = ash\_attr\_qp\_region\_box\_origin\_y

regionboxZ = ash\_attr\_qp\_region\_box\_origin\_z

**ash\_attr\_qp\_region\_box\_size\_width** indicates the width of the region bounding box. When not present, the value of ash\_attr\_qp\_region\_box\_size\_width is inferred to be 0.

**ash\_attr\_qp\_region\_box\_size\_height** indicates the height of the region bounding box. When not present, the value of ash\_attr\_qp\_region\_box\_size\_height is inferred to be 0.

**ash\_attr\_qp\_region\_box\_size\_depth** indicates the depth of the region bounding box. When not present, the value of ash\_attr\_qp\_region\_box\_size\_depth is inferred to be 0.

regionboxWidth = ash\_attr\_qp\_region\_box\_size\_width

regionboxHeight = ash\_attr\_qp\_region\_box\_size\_height

regionboxDepth = ash\_attr\_qp\_region\_box\_size\_depth

**ash\_attr\_region\_qp\_delta** specifies the delta qp from the SliceQpY[i] and SliceQpC[i] (with i = 0…num\_layer – 1) of the region specified by ash\_attr\_qp\_region\_box. When not present, the value of ash\_attr\_region\_qp\_delta is inferred to be 0.

regionboxDeltaQp = ash\_attr\_region\_qp\_delta

#### Attribute slice data semantics

**zerorun** specifies the number of 0 prior to residual.

**predIndex**[ i ] specifies the predictor index to decode the i-th point value of the attribute. The value of predIndex[ i ] shall be range of 0 to max\_num\_direct\_predictors.

The variable maxPredDiff is calculated as follows…[Ed: tbd]

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 num\_pred\_nearest\_neighbours.

minValue = maxValue =

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

minValue = Min( minValue, )

maxValue = Max( maxValue, )

}

maxPredDiff = maxValue − minValue;

#### Quantized value bitstream syntax

**isZero**[ k ] equal to 1 indicates that residual value[k][i] is equal to 0. isZero equal to 0 indicates that residual value[k][i] is not equal to 0.

**isOne**[ k ] equal to 1 indicates that residual value[k][i] equal to 1. isOne equal to 0 indicates that residual value[k][i] is larger than 2.

**values**[k][i]describes the k-th dimension and the i-th point value of the attribute.

**remaining\_values**[k][i] 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.

# Decoding process

## General decoding process

The input to this process is a sequence of typed payload 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 [Ed: definition: geometry or attribute payloads with the same slice\_id] of the current picture:

1. Point positions are decoded using the geometry payload of the current slice as specified in 8.2.
2. Point attributes are decoded for each attribute payload 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 ranging from 0 to gsh\_num\_points − 1 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 in the packed geometry octree. and
* a spatial location (xN, yN, zN) 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 MaxGeometryOctreeDepth − 1, 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 MaxGeometryOctreeDepth − 1, 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 ];

x = 2 × xN + ( childIdx & 4 ) = = 1 ;

y = 2 × yN + ( childIdx & 2 ) = = 1 ;

z = 2 × zN + ( childIdx & 1 ) = = 1 ;

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

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

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

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

}

if( direct\_mode\_flag ) {

if (!duplicated\_point\_flag) {

for( i = 0; i <= num\_direct\_points\_minus1; i++, PointCount++ ) {

PointPos[ PointCount ][ 0 ] = geomScale( ( x << EffectiveChildNodeSizeLog2 ) + PointOffsetX[ i ] , 0);

PointPos[ PointCount ][ 1 ] = geomScale( ( y << EffectiveChildNodeSizeLog2 ) + PointOffsetY[ i ] , 1);

PointPos[ PointCount ][ 2 ] = geomScale( ( z << EffectiveChildNodeSizeLog2 ) + PointOffsetZ[ i ] , 2);

}

}

else {

for( i = 0; i <= num\_direct\_points\_minus1; i++, PointCount++ ) {

PointPos[ PointCount ][ 0 ] = geomScale( ( x << EffectiveChildNodeSizeLog2 ) + PointOffsetX[ 0 ] , 0);

PointPos[ PointCount ][ 1 ] = geomScale( ( y << EffectiveChildNodeSizeLog2 ) + PointOffsetY[ 0 ] , 1);

PointPos[ PointCount ][ 2 ] = geomScale( ( z << EffectiveChildNodeSizeLog2 ) + PointOffsetZ[ 0 ] , 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 ) / 6.

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 list geomLevelScale is specified as geomLevelScale[ i ] = { 659445, 741374, 831472, 933892, 1048576, 1175576 } with i = 0 .. 5.

The output variable pos is derived as follows:

highPartS = highPart << ScalingNodeSizeLog2[cIdx]

lowPartS = (lowPart \* (geomLevelScale[qP % 6] << (qP / 6) ) + (1 << 19)) >> 20

pos = highPart | lowPart

### 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 PointNum as the number of the decoded geometry points,

the array PointPos[ i ][ axis ] with i = 0...PointNum – 1 , axis = 0..2, for the decoded geometry point positions

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 PointNum × 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..PointNum − 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 9 and Table 10, respectively.

Table 9 — segStOffsetTable[ k ][ axis ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | k | | | | | | | | | | | |
| axis | 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 10 — segEdOffsetTable [ k ][ axis ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | k | | | | | | | | | | | |
| axis | 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..PointNum − 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 5.8.1.

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 11.

Table 11 — 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.

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:

PointNum = numPtsOnTriangle

PointPos[ i ][ axis ] with i = 0..PointNum − 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 (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 12.

Table 12 — Determination of the values of probable\_order[] from planeRate[]

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 in along an axis

The determination of planarIdx (respectively planePosIdx) for the arithmetic coding of is\_planar\_flag[child][ axisIdx] (respectively plane\_position[child][ axisIdx]) is performed based on the planar status of and the distance from the closest already decoded node with same depth and same axisIdx -th coordinate as the current node’s child node. A limited number of candidate nodes for the closest nodes are tracked by two buffers

buffer\_closest\_node\_position[axisIdx] [coord] [candidateIdx][ secondary\_axisIdx],

buffer\_closest\_node\_status[axisIdx] [coord] [candidateIdx],

where axisIdx is an axis index in the range 0…2, and where candidateIdx is a candidate node index in the range 0… nb\_candidates-1. The value nb\_candidates specifies the number of candidate nodes tracked by the buffer and is set to nb\_candidates = 4 .

The value of the variable coord specifies the coordinate of the candidate nodes along the axisIdx -th axis at the spatial precision of the current depth plus 1 which is the depth of the child nodes. For a given value depth of the depth in octree, coord is in the range 0… (( 1 << (depth+1) ) - 1) .

The value of the variable secondary\_axisIdx specifies a secondary axis index in the range 0…1. When axisIdx is equal to 0, secondary\_axisIdx equal to 0 specifies the y axis, and secondary\_axisIdx equal to 1 specifies the z axis. When axisIdx is equal to 1, secondary\_axisIdx equal to 0 specifies the x axis, and secondary\_axisIdx equal to 1 specifies the z axis. When axisIdx is equal to 2, secondary\_axisIdx equal to 0 specifies the x axis, and secondary\_axisIdx equal to 1 specifies the y axis.

The two buffers are initialized, at the start of the geometry decoding process and also each time the variable depth specifying the octree depth is incremented, as follows

for(axisIdx = 0 ; axisIdx <= 2 ; axisIdx++ )

for(coord = 0 ; coord < (1 << (depth+1)) ; coord + + )

for(candidateIdx = 0; candidateIdx < nb\_candidates; candidateIdx + + ) {

buffer\_closest\_node\_position[axisIdx] [coord] [candidateIdx][0] = FOO\_VALUE

buffer\_closest\_node\_position[axisIdx] [coord] [candidateIdx][1] = FOO\_VALUE

buffer\_closest\_node\_status[axisIdx] [coord] [candidateIdx] = UNKNOWN\_STATUS

}

where FOO\_VALUE and UNKNOWN\_STATUS are arbitrary values different from 0,1 and each other.

The two buffers are updated after each decoding of the planar mode syntax of a child node, e.g. after each call of geometry\_planar\_mode\_data(child, axisIdx). First the closest candidate node whose index is closestIdx, index determined as described in 8.2.4.3, is pushed as the right-most candidate in the buffer. Second, all candidate nodes are pushed left, losing the left-most candidate mode in the process. Finally, the child node is pushed as the right-most candidate.

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1][0] = buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [closestIdx][0]

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1][1] = buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [closestIdx][1]

buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1] = buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [closestIdx]

for(candidateIdx = 0; candidateIdx < nb\_candidates-1; candidateIdx + + ) {

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [candidateIdx][0] = buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [candidateIdx+1][0]

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [candidateIdx][1] = buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [candidateIdx+1][1]

buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [candidateIdx] = buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [candidateIdx+1]

}

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1][0] = coord\_child[other\_axis[axisIdx][0]]

buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1][1] = coord\_child[other\_axis[axisIdx][1]]

buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [nb\_candidates-1] = child\_node\_status[axisIdx]

The planar status child\_node\_status[] of the child node is determined as follows

if (is\_planar\_flag [ child ] [axisIdx]){

child\_node\_status[axisIdx] = plane\_position [ child ] [axisIdx]

} else {

child\_node\_status[axisIdx] = NOT\_PLANAR

}

where planar NOT\_PLANAR is a value different from 0,1 and UNKNOWN\_STATUS.

The values of other\_axis[][] are provided by the Table 13.

Table 13 — the values of other\_axis[][]

|  |  |  |
| --- | --- | --- |
| axisIdx | other\_axis[axisIdx][0] | other\_axis[axisIdx][1] |
| 0 | 1 | 2 |
| 1 | 0 | 2 |
| 2 | 0 | 1 |

The variable coord\_child[] is the coordinates of the child node with the spatial precision at depth + 1 which is the depth of the child node. The value of coord\_child[axisIdx] are obtained from the unscaled coordinate ( xN, yN, zN) of the lower left corner of the child node by coord\_child[0] = xN, coord\_child[1] = yN and coord\_child[2] = zN .

#### Determination of planarIdx for the coding of the planar mode flag

The determination of planarIdx for the arithmetic coding of is\_planar\_flag[child][axisIdx] is performed based firstly on axisIdx, secondly based on the occupancy of the neighbouring nodes of the current node along the axisIdx-th axis, and thirdly on the distance from the closest already decoded node with same depth and same axisIdx -th coordinate as the current node child node.

The index closestIdx the closest candidate node is determined as the left-most index idx that minimizes the distance d[axisIdx][idx] defined as follows

d[child][axisIdx][idx] = abs( buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [Idx][0] - coord\_child[other\_axis[axisIdx][0]] ) + abs( buffer\_closest\_node\_position[axisIdx] [coord\_child[axisIdx]] [Idx][1] - coord\_child[other\_axis[axisIdx][1]] )

and the minimal distance is d\_min[child] [axisIdx] = d[child] [axisIdx][closestIdx].

The planar parent neighbouring configuration neigh\_planar[child] [axisIdx] is deduced from the occupancy of the neighbouring node N of the current (parent) node such that the node N is a neighbourg along the axisIdx-th axis and is adjacent to the child node. neigh\_planar[child] [axisIdx] is set to 1 if the node N is occupied, 0 otherwise.

The context index planarIdx , for a child node and an axis index axisIdx, is then determined by

planarIdx = axisIdx + 3 \* ( neigh\_planar[child] [axisIdx] + (d\_min[child] [axisIdx]<=2 ? 0 : 2 ) )

#### Determination of planePosIdx for the coding of the plane position

The determination of planePosIdx for the arithmetic coding of plane\_position[child][axisIdx] is performed based on

* axisIdx
* planar parent neighbouring configuration neigh\_planar[child] [axisIdx]
* the planar status status[child][axisIdx] of the closest already decoded node whose index is closestIdx
* the distance d\_min[child] [axisIdx] from the closest already decoded node
* and fourthly on the position pos[child][idx] along the axisIdx-th axis of the child inside its parent.

The planar status is determined by

status[child] [axisIdx] = buffer\_closest\_node\_status [axisIdx] [coord\_child[axisIdx]] [closestIdx]

and pos[child][axisIdx] is set equal to 1 if the child node is located at the higher coordinate position, within its parent, along the axisIdx-th axis, and set equal to 0 if the child node is located at the lower coordinate position.

The context index planePosIdx, for a child node and an axis index axisIdx, is then determined by

if (status[child][axisIdx] == UNKNOWN\_STATUS || status[child][axisIdx] == NOT\_PLANAR) {

planePosIdx = 0

} else {

discrete\_dist = (d\_min[child] [axisIdx]<=2 ? 0 : 1 ) + (d\_min[child] [axisIdx]<=16 ? 0 : 1 )

planePosIdx = axisIdx + 3\* (neigh\_planar[child] [axisIdx]+2\* discrete\_dist)

planePosIdx += 18 \* pos[child][idx]

planePosIdx += 1

}

#### 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])

{

p = p0[planePosIdx] >> 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] 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] 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.

## Attribute decoding

Inputs to this process are:

the attribute parameter set and the associated bitstream,

the number of decoded geometry points, PointNum,

the series of the decoded geometry point PointPos[i][3], where i is in the range of 0 to PointNum − 1.

Output of the process is a series of the decoded point PointAttr[i][attrCnt], where i is in the range of 0 to PointNum − 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.

When attr\_coding\_type is equalt to 1, the RAHT decoding process in clause 8.3.1 is invoked.

Otherwise, if attr\_coding\_type is equal to 0, the Predicting Weight Liting decoding process in clause 8.3.3 is invoked.

Otherwise (if attr\_coding\_type is equal to 2), the Fixed Weight Liting decoding process in clause 8.3.2 is invoked.

### 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 .. AttrCnt − 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 initialised to 0.

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).

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[ x ][ y ][ z ][ 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 ( xPt, yPt, zPt ) are set equal to PointPos[ i ][ j ], with j = 0 .. 2 respectively.
* If Weights[ 0 ][ xPt ][ yPt ][ zPt ] is equal to 1, the following applies:

for( cIdx = 0; cIdx < AttrCnt; cIdx++ )

PointAttr[ i ][ cIdx ] = DivExp2RoundHalfInf( recon[ xPt ][ yPt ][ zPt ], 15 )

* Otherwise, Weights[ 0 ][ xPt ][ yPt ][ zPt ] is greater than 1, the following process is used to reconstruct samples PointAttr[ i + j ][ cIdx ] for j = i .. Weights[ 0 ][ xPt ][ yPt ][ zPt ] − 1:
  + - The ( AttrCnt )×( 2 ) sized array x is initialised as follows:

for ( cIdx = 0; cIdx < AttrCnt; cIdx++ )

x[ cIdx ][ 0 ] = recon[ xPt ][ yPt ][ zPt ]

* + - For each wi in the descending range Weights[ 0 ][ xPt ][ yPt ][ zPt ] − 1 .. 1, the following applies:
      * The scaling process for RAHT coefficients is invoked for each component cIdx in the range 0 .. AttrCnt − 1, with the single-element coeff set equal to value[ cIdx ][ CoeffIdx ], the 3D transform level lvl set equal to 0, and the variable cIdx as inputs. The array element x[ 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 .. AttrCnt − 1, the following applies:
        + The inverse two-point transform process is invoked with the array x[ 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 x[ cIdx ][ 0 ] is replaced by r[ 0 ]
        + The output PointAttr[ i + wi ][ cIdx ] is derived as follows:

PointAttr[ i + wi ][ cIdx ] = DivExp2RoundHalfInf( x[ 1 ], 15 )

* + - The ouput PointAttr[ i ][ cIdx ] for cIdx = 0 .. AttrCnt − 1 is derived as follows:

PointAttr[ i ][ cIdx ] = DivExp2RoundHalfInf( x[ 0 ], 15 )

#### RAHT weights derivation process

The outputs of this process are:

* the array Weights, with entries Weights[ lvl ][ x ][ y ][ z ] equal to the number of points represented by a coefficient at position ( x, y, z ) 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( lvl = 0, done = 0; !done; )

for( j = 0; j < 3; j++, lvl++ )

for( i = 0; i < PointCount; i++ ) {

x = PointPos[ i ][ 0 ] >> ( ( lvl + 0 ) / 3 )

y = PointPos[ i ][ 1 ] >> ( ( lvl + 1 ) / 3 )

z = PointPos[ i ][ 2 ] >> ( ( lvl + 2 ) / 3 )

Weights[ lvl ][ x ][ y ][ z ] += 1;

if (Weights[ lvl ][ x ][ y ][ z ] == PointCount)

done = 1;

}

The variable NumRahtLevels is set equal to lvl / 3 + 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 × lvl) ); mIdx++ ) {

( xN, yN, zN ) = MortonToTuple( mIdx )

if( Weights[ lvl + 3 ][ xN ][ yN ][ zN ] = = 0 )

continue

nodesX[ nIdx ] = 2 × xN

nodesY[ nIdx ] = 2 × yN

nodesZ[ nIdx ] = 2 × zN

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 ( xTn , yTn , zTn ) indicating the location of a transform tree node are initialised with the values of nodesX[ nIdx ], nodesY[ nIdx ], and nodesZ[ nIdx ] respectively.

An ( AttrCnt )×( 8 ) element array of transform coefficients is derived as follows:

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

( dx, dy, dz) = MortonToTuple( childIdx )

if( inheritDc && childIdx = = 0 )

continue

if( Weights[ lvl ][ xTn + dx ][ yTn + dy ][ zTn + dz ] = = 0 )

continue

for( cIdx = 0; cIdx < AttrCnt; cIdx++ )

coeff[ cIdx ][ childIdx ] = value[ 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 ( xTn, yTn, zTn ), 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++ ) {

( dx, dy, dz) = MortonToTuple( childIdx )

recon[ xTn + dx ][ yTn + dy ][ zTn + dz ][ cIdx ] = r[ childIdx ]

}

#### Reconstruction process for a 2×2×2 transform tree node

The inputs to this process are:

* a position ( xTn, yTn, zTn ) 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 quantised 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 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 ( xTn/2, yTn/2, zTn/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 ( xTn, yTn, zTn ) 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[ xTn / 2 ][ yTn / 2 ][ zTn / 2 ][ cIdx ], 15 )

e[ 0 ] <<= 15

}

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 ( xTn, yTn, zTn ) 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 quantised coefficients
* 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( num\_layer − 1, NumRahtLevels − lvl ).

If cIdx is equal to 0, the variable qs is set equal to SliceQstepY[ qlayer ]. Otherwise, cIdx is greater than 0, the variable qs is set equal to SliceQstepC[ qlayer ].

For each element coeff[ i ] with i = 0 .. n − 1, the scaled transform coefficient d[ i ] is derived as follows:

d[ i ] = DivExp2RoundHalfUp( coeff[ i ] × qs, 8 )

#### Transform prediction upsampling process

The inputs to this process are:

* a position ( xTn, yTn, zTn ) 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 is an eight-element array p of upsampled values.

The upsampled 2×2×2 block located at the position ( xTn, yTn, zTn ) is derived as follows. For each row in Table 14, the following applies:

cx = nx + dx

cy = ny + dy

cz = nz + dz

if( Weights[ lvl ][ cx ][ cy ][ cz ] > 0 ) {

neighVal = Recon[ cx ][ cy ][ cz ][ cIdx ]

value = DivFp( neighVal, iSqrt( Weights[ lvl ][ cx ][ cy ][ cz ] << 30 ), 15 )

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

sumDc[ childIdx ] += DivExp2RoundHalfInf( value × wn[ childIdx ], 15 )

sumWn[ childIdx ] += wn[ childIdx ]

}

}

For each child position childIdx in the range 0 to 7, inclusive, and sumW[ childIdx ] > 0, as follows:

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

( dx, dy, dz ) = MortonToTuple( childIdx )

pred = DivFp(sumDc[childIdx], sumWn[childIdx], 15)

pred ×= iSqrt( Weights[ lvl − 3 ][ 2×xTn + dx ][ 2×yTn + dy ][2×zTn + dz ] << 30 )

p[ childIdx ] = DivExp2RoundHalfInf( pred, 15 )

}

Table 14 — Weighting matrix for determining upsampled child position weights, wn[ childIdx ], for various neighbour position offsets ( dx, dy, dz )

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Neighbour offset** | | | **wn[ childIdx ]** | | | | | | | |
| **dx** | **dy** | **dz** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 0 | 0 | 0 | a | a | a | a | a | a | a | a |
| 1 | 0 | 0 | b | b | b | b | 0 | 0 | 0 | 0 |
| −1 | 0 | 0 | b | b | b | b | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | b | b | 0 | 0 | b | b |
| 0 | −1 | 0 | b | b | 0 | 0 | b | b | 0 | 0 |
| 0 | 0 | 1 | 0 | b | 0 | b | 0 | b | 0 | b |
| 0 | 0 | −1 | b | 0 | b | 0 | b | 0 | b | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | c | c |
| −1 | 1 | 0 | 0 | 0 | c | c | 0 | 0 | 0 | 0 |
| 1 | −1 | 0 | 0 | 0 | 0 | 0 | c | c | 0 | 0 |
| −1 | −1 | 0 | c | c | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | c | 0 | 0 | 0 | c |
| 0 | −1 | 1 | 0 | c | 0 | 0 | 0 | c | 0 | 0 |
| 0 | 1 | −1 | 0 | 0 | c | 0 | 0 | 0 | c | 0 |
| 0 | −1 | −1 | c | 0 | 0 | 0 | c | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | c | 0 | c |
| −1 | 0 | 1 | 0 | c | 0 | c | 0 | 0 | 0 | 0 |
| 1 | 0 | −1 | 0 | 0 | 0 | 0 | c | 0 | c | 0 |
| −1 | 0 | −1 | c | 0 | c | 0 | 0 | 0 | 0 | 0 |
| Where a = 24518, b = 5536, c = 2937 | | | | | | | | | | |

#### Forward transform process for 2×2×2 blocks

The inputs to this process are:

* a position ( xTn, yTn, zTn ) 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 15 in sequential order, the array p is modified by transforming a pair of values by invoking the forward two-point transform process with the input array x equal to { p [ i ], p[ j ] }, and the array w equal to { wi, wj }. The output 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 16.

Table 15 — 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 ][ xTn + 0 ][ yTn + 0 ][ zTn ] | w[ lvl ][ xTn + 0 ][ yTn + 0 ][ zTn + 1 ] |
| 2 | 3 | w[ lvl ][ xTn + 0 ][ yTn + 1 ][ zTn ] | w[ lvl ][ xTn + 0 ][ yTn + 1 ][ zTn + 1 ] |
| 4 | 5 | w[ lvl ][ xTn + 1 ][ yTn + 0 ][ zTn] | w[ lvl ][ xTn + 1 ][ yTn + 0 ][ zTn + 1 ] |
| 6 | 7 | w[ lvl ][ xTn + 1 ][ yTn + 1 ][ zTn] | w[ lvl ][ xTn + 1 ][ yTn + 1 ][ zTn + 1 ] |
| 4 | 6 | w[ lvl + 1 ][ xTn + 1 ][ yTn ][ zTn] | w[ lvl + 1 ][ xTn + 1 ][ yTn + 1 ][ zTn ] |
| 0 | 2 | w[ lvl + 1 ][ xTn + 0 ][ yTn ][ zTn] | w[ lvl + 1 ][ xTn + 0 ][ yTn + 1 ][ zTn ] |
| 0 | 4 | w[ lvl + 2 ][ xTn + 0 ][ yTn ][ zTn] | w[ lvl + 2 ][ xTn + 1 ][ yTn + 0 ][ zTn ] |

Table 16 — 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 ) / ( w[ 0 ] + w[ 1 ] ) )

b = iSqrt( ( w[ 1 ] << 30 ) / ( w[ 0 ] + w[ 1 ] ) )

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 ( xTn, yTn, zTn ) 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 initialised as r[ t ] = e[ s ] with s = 0 .. 7 and the value of t derived from s according to Table 13.

For each row of Table 12 in reverse order, the array r is modified by transforming a pair of values by invoking the inverse two-point transform process with the input array x equal to { r[ i ], r[ j ] }, and the array w equal to { wi, wj }. The output 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 ) / ( w[ 0 ] + w[ 1 ] ) )

b = iSqrt( ( w[ 1 ] << 30 ) / ( w[ 0 ] + w[ 1 ] ) )

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 )

### Lifting decoding process

Inputs of this process are:

two variables minAttribute and maxAttribute specifying the minimum and maximum allowed attribute values.

a variable attrCnt specifying the attribute dimension.

a series of quantized attribute coefficients quantAttributeCoefficients[ i ][ a ], where i is in the range of 0 to PointNum − 1, inclusive, and a in the range 0 to attrCnt − 1, inclusive.

a series of 3D points PointPos[ i ][ axis], where i is in the range of 0 to PointNum − 1, inclusive, and axis in the range 0 to 2, inclusive.

a variable levelDetailCount specifying the number of level of detail to be generated.

an array of distances/periods sampling [ l ], where l is in the range of 0 to levelDetailCount − 1, inclusive.

a variable searchRange specifyingthe search rangefor level of detail generation and the nearest neighbour search.

a variable numPredNearestNeighbours indicating the maximum number of nearest neighbours per point.

a variable FixedPointWeightShift specifying the fixed-point representation precision for prediction and quantization weights.

a variable FixedPointAttributeShift specifying the fixed-point representation precision for attribute values.

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 PointNum − 1, inclusive, and a in the range of 0 to attrCnt − 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 with the parameters PointPos, levelDetailCount, sampling, searchRange and numPredNearestNeighbours. 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 PointNum − 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, neighboursDistance2 and FixedPointWeightShift. The output of this process is stored in predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 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. The output of this process is stored in quantizationWeights[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

The inverse quantization process in 8.3.2.6 is invoked with the parameters indexes, neighbours, neighboursCount, predictionWeights and FixedPointWeightShift. The output of this process is stored in unquantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

The inverse lifting process in 8.3.2.7 is invoked with the parameters attrCnt, unquantAttributeCoefficients, FixedPointWeightShift, FixedPointAttributeShift, quantizationWeights, predictionWeights and pointCountPerLevelOfDetail. This process updates the attribute coefficients unquantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

The reconstructed attributes values are obtained as follows.

for (i = 0; i < PointNum; i++ ) {

for (j = 0; j < attrCnt; j++ ) {

value = divExp2RoundHalfInf(unquantAttributeCoefficients[ i ][ j ], FixedPointAttributeShift);

attributeValues[ i ][ j ] = Clip(value , minAttribute, maxAttribute);

}

}

#### Level of Detail Generation

Inputs of this process are:

a series of 3D points PointPos[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range 0 to 2, inclusive.

a variable levelDetailCount specifying the number of level of detail to be generated.

an array of distances sampling [ l ], where l is in the range of 0 to levelDetailCount − 1, inclusive.

a variable searchRange specifyingthe search rangefor level of detail generation and the nearest neighbour search.

a variable numPredNearestNeighbours indicating the maximum number of nearest neighbours per point.

The outputs of the process are

an array of point indexes indexes[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

a series of nearest neighbours indexes neighbours[ i ][ n ], where i is in the range of 0 to PointNum − 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 PointNum − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointNum − 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, inclusive.

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 1, 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 0, the array sampling [ l ] represents sampling periods verifying the following condition:

sampling [ l ] > 1.

First, the point sorting process based on Morton code in clause **Error! Reference source not found.** is invoked with the parameter PointPos. 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 = PointNum

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

unprocessedPointIndexes[ i ] = Order[ i ]

}

assignedPointCount = 0;

for (lod = 1; lod <= levelDetailCount; lod++) {

unprocessedPointCountPerLevelOfDetail[lod] = 0;

}

unprocessedPointCountPerLevelOfDetail[0] = PointNum;

for (lod = 0; unprocessedPointCount > 0 && lod <= levelDetailCount; lod++) {

nonAssignedPointCount = 0;

startIndex = assignedPointCount;

if (lod = = levelDetailCount) {

-- --

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

assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i];

}

} else {

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

foundAssignedPointWithinDistanceFlag = 0;

if ( lifting\_lod\_regular\_sampling\_enabled\_flag== 1) {

foundAssignedPointWithinDistanceFlag = (i % sampling[ lod ]) != 0;

} else {

currentPos = PointPos[ unprocessedPointIndexes[ i ] ];

k = 0;

j= nonAssignedPointCount – 1;

while (j >= 0 && k++ < lifting\_scalability\_enabled\_flag ? 1 : lifting\_search\_range) {

neighbourPos = PointPos[nonAssignedPointIndexes [j]];

if (lifting\_scalability\_enabled\_flag){

d2 = (currentPos[0] – neighbourPos[ 0 ])/(1<<nodeSizeLog2)   
 +(currentPos[1] – neighbourPos[ 1 ])/(1<<nodeSizeLog2)  
 +(currentPos[1] – neighbourPos[ 2 ])/(1<<nodeSizeLog2)

if (d2 = = 0) {

foundAssignedPointWithinDistanceFlag = 1;

break;

}

} else {

d2 = Norm2(

currentPos[0] – neighbourPos[0],

currentPos[1] – neighbourPos[1],

currentPos[2] – neighbourPos[2]));

if (d2 <= sampling[ lod ]) {

foundAssignedPointWithinDistanceFlag = 1;

break;

}

}

}

j− −;

}

if (foundAssignedPointWithinDistance == 1)

assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i];

else

nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i];

}

}

endIndex = assignedPointCount

currentLayer = levelDetailCount – l

if( lifting\_scalability\_enabled\_flag && lod = = 1 && (endIndex – startIndex) > startIndex ){

computeNearestNeighbours(PointPos, searchRange , 0, startIndex0startI, currentLayer, assignedPointIndexes, McodeUnsorted, numPredNearestNeighbours, nonAssignedPointCount, nonNssignedPointIndexes, lod -1 );

}

computeNearestNeighbours(PointPos, searchRange , startIndex, endIndex, currentLayer, assignedPointIndexes, McodeUnsorted, numPredNearestNeighbours, nonAssignedPointCount, nonNssignedPointIndexes, lod );

unprocessedPointCountPerLevelOfDetail[lod+1] = nonAssignedPointCount;

unprocessedPointCount = nonAssignedPointCount;

unprocessedPointIndexes = nonAssignedPointIndexes

}

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

indexes[ unprocessedPointCount− 1 – i ] = assignedPointIndexes[ i ]

}

for (lod = 0; lod <= levelDetailCount; lod++) {

pointCountPerLevelOfDetail[lod] = unprocessedPointCountPerLevelOfDetail[levelDetailCount − lod];

}

#### Definition of computeNearestNeighbours()

Inputs of this process are:

a series of the decoded geometry point PointPos[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive.

a variable searchRange specifyingthe search rangefor the nearest neighbour search.

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

a variable liftingIntraLodPredictionNumLayers specifying the number of LoD layer where intra lod prediction is enabled

an array of point indexes assignedPointIndexes[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

an array of Morton codes McodeUnsorted[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

a variable numPredNearestNeighbours indicating the maximum number of nearest neighbours per point.

a variable nonAssignedPointCount specifying the number of non-assigned points.

an array of point indexes nonNssignedPointIndexes[ i ], where i is in the range of 0 to PointNum − 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 PointNum − 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 PointNum − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointNum − 1, inclusive, and n in the range of 0 to numPredNearestNeighbours − 1, inclusive.

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 − searchRange);

j1 = Min(nonAssignedPointCount, j + searchRange + 1);

neighboursCount[ currentIndex ] = 0;

k = 0;

for ( k = j0; k < j1 ; k++) {

neighbourIndex = nonAssignedPointIndex[k];

neighbourPos = PointPos[neighbourIndex];

d2 = Norm2(

lifting\_neighbour\_bias[ 0 ] × (currentPos[0] – neighbourPos[0]),

lifting\_neighbour\_bias[ 1 ] × (currentPos[1] – neighbourPos[1]),

lifting\_neighbour\_bias[ 2 ] × (currentPos[2] – neighbourPos[2]));

insertIndex = ( k − j > 0 ) ? ( ( ( k – j ) << 1 ) – 1 ) : ( ( j – k ) << 1 );

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 < liftingIntraLodPredictionNumLayers) {

j1 = Min(endIndex, k + searchRange);

for ( k = i + 1; k < j1; k++) {

neighbourIndex = assignedPointIndex[k];

neighbourPos = PointPos[neighbourIndex];

d2 = Norm2(

lifting\_neighbour\_bias[ 0 ] × (currentPos[0] – neighbourPos[0]),

lifting\_neighbour\_bias[ 1 ] × (currentPos[1] – neighbourPos[1]),

lifting\_neighbour\_bias[ 2 ] × (currentPos[2] – neighbourPos[2]));

if ( lifting\_scalability\_enabled\_flag ){

d2 = Norm2(

(currentPos[0]>>nodeSizeLog2) – (neighbourPos[ 0 ] >>nodeSizeLog2),

(currentPos[1] >>nodeSizeLog2)– (neighbourPos[1] >>nodeSizeLog2),

(currentPos[2] >>nodeSizeLog2) – (neighbourPos[2] >>nodeSizeLog2));

}

insertIndex = ((k – i) << 1) – 1;

ifi (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 ]);

}

}

}

}

#### 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 PointNum − 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 PointNum − 1, inclusive.

an array of nearest neighbours squared distances neighboursDistance2[ i ][ n ], where i is in the range of 0 to PointNum − 1, inclusive, and n in the range of 0 to numPredNearestNeighbours − 1, inclusive.

a variable FixedPointWeightShift specifying the fixed-point representation precision for prediction weights.

The output is:

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 1, inclusive, and n in the range of 0 to numPredNearestNeighbours − 1, inclusive.

The prediction weights derivation process proceeds as follows:

MaxWeightValue = 1 << FixedPointWeightShift;

FixedPointWeightShiftSum = 4 × FixedPointWeightShift − 1;

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

while (neighboursCount[ i ] > 1 &&

neighboursDistance2[ i ][ 0 ] > 0 &&

(neighboursDistance2[ neighbourCount[ i ] −1 ][ 0 ] >> FixedPointWeightShift) >

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 > FixedPointWeightShift ? bitCount − FixedPointWeightShift : 0;

biasDistance = ((1 << shift) >> 1);

if (neighboursCount[ i ]== 2) {

d0 = (neighboursDistance2[ i ][ 0 ] + biasDistance) >> shiftDistance;

d1 = (neighboursDistance2[ i ][ 1 ]+ biasDistance) >> shiftDistance;

sum = d1 + d0;

sumDiv2 = sum >> 1;

w1 = ((d0 <<FixedPointWeightShift) + sumDiv2) / sum;

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;

sumDiv2 = sum >> 1;

r = ((1 << kFixedPointWeightShiftSum) + sumDiv2) / sum;

shiftWeight = FixedPointWeightShiftSum − FixedPointWeightShift;

biasWeight = 1 << (shift − 1);

w2 = (d0d1 × r + biasWeight) >> shiftWeight;

w1 = (d0d2 × r + biasWeight) >> shiftWeight;

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 PointNum − 1, inclusive.

a series of nearest neighbours indexes neighbours[ i ][ j ], where i is in the range of 0 to PointNum − 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 PointNum − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 1, inclusive, and n in the range of 0 to numPredNearestNeighbours − 1, inclusive.

a variable FixedPointWeightShift specifying the fixed-point representation precision for quantization weights.

The output is:

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

The quantization weights derivation procedure proceeds as follows.

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

quantizationWeights[ i ] = 1 << FixedPointWeightShift;

}

for (i = PointNum−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 ],

FixedPointWeightShift);

}

}

for (i = 0; i < PointNum; i++)

quantizationWeights[ i ] = iSqrt(quantizationWeights[ i ]);

if ( lifting\_scalability\_enabled\_flag ){

index = 0

pointNumSumLoD = 0

for (lodIndex = 0; lodIndex < lodCount; lodIndex++){

pointNumSumLoD += pointNumPerLoD[lodIndex]

for (i = 0; i < pointNumPerLoDpointNumPerLoD[lodIndex]; i++){

quantizationWeights[index ] = 1 << kFixedPointWeightShift

if ( numCancelLayer = = 0 && lodIndex != (lodCount −− 1))

quantizationWeights[index] \*= (PointNum / pointNumPerLoDpointNumPerLoD[lodIndex])

if (numCancelLayer < 2){

if( lodIndex = = (lodCount − 2 + numCancelLayer) &&

(PointNum – pointNumSumLoD) < pointNumPerLoD[lodIndex] )

quantizationWeights[index] = 1 << kFixedPointWeightShift

}

index++;

}

}

}

where pointNumPerLoDpointNumPerLoD[lodIndex] stores the number of points in the lodIndex layer of the octree structure in the decoded geometry.

#### Inverse quantization process

Inputs of this process are:

a variable attrCnt specifying the attribute dimension.

a series of quantized attribute coefficients quantAttributeCoefficients[ i ][ a ], where i is in the range of 0 to PointNum − 1, inclusive, and a in the range 0 to attrCnt − 1, inclusive.

a variable FixedPointWeightShift specifying the fixed-point representation precision for quantization weights.

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

a variable SliceQpY[ l ] with l=0..num\_layer − 1 specifying the inverse quantization parameter for the luma coefficient,

a variable SliceQpC[ l ] with l=0..num\_layer − 1 specifying the inverse quantization parameter for the chroma coefficient,

a variable num\_layer specifying the number of SliceQpY and SliceQpC,

a series of the decoded geometry point PointPos[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive.

a variable regionboxDeltaQp specifying the region box delta quantization parameter.

a variable regionboxX, regionboxY, regionboxZ specifying the region box origin.

a variable regionboxWidth, regionboxHeight, regionboxDepth specifying the region box size.

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 PointNum − 1, inclusive, and a in the range of 0 to attrCnt − 1, inclusive.

The inverse quantization process proceeds as follows.

endIndex = pointCountPerLevelOfDetail[ 0 ];

for (i=0, d = 0; i< PointNum; i++) {

if ( i == endIndex ) {

endIndex = pointCountPerLevelOfDetail[ ++d ];

layerQpYlayer = ( d < num\_layer ) ? SliceQpY[ d ] : SliceQpY[ num\_layer – 1 ];

layerQpClayer = ( d < num\_layer ) ? SliceQpC[ d ] : SliceQpC[ num\_layer – 1 ];

}

isPointContainedRegionBox =

( regionboxX <= PointPos[ i ][ 0 ] ) && ( PointPos[ i ][ 0 ] < ( regionboxX + regionboxWidth ) )

&& ( regionboxY <= PointPos[ i ][ 1 ] ) && ( PointPos[ i ][ 1 ] < ( regionboxY + regionboxHeight ) )

&& ( regionboxZ <= PointPos[ i ][ 2 ] ) && ( PointPos[ i ][ 2 ] < ( regionboxZ + regionboxDepth ) ) ;

qpY = layerQpY + ( isPointContainedRegionBox ? regionboxDeltaQp : 0 ) ;

qpC = layerQpC + ( isPointContainedRegionBox ? regionboxDeltaQp : 0 ) ;

QstepY = QpToQstep( qpY );

QstepC = QpToQstep( qpC );

for (a=0; a< attrCnt; a++)

unquantAttributeCoefficients [ i ][ a ] = quantAttributeCoefficients[ i ][ a ] ×

( a == 0 ? QstepY : QstepC);)

}

#### Inverse lifting

Inputs of this process are:

a variable attrCnt specifying the attribute dimension.

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 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 variable attrCnt specifying the attribute dimension.

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 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 < attrCnt; j++)

updates[ i ][ j ] = 0

}

pointCount = endIndex − startIndex; //[Ed(ON). the variable name should be reconsidered as PointCount is used for the num of the points in the slice]

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

index = predictorCount − i − 1 + startIndex;

currentQuantWeight = quantizationWeights[index];

for (p = 0; p < neighboursCount[index]; p++) {

neighbourIndex = neighbours[index][p];

weight = predictionWeights[index][p] × currentQuantWeight;

updateWeights[neighbourIndex] += weight;

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

updates[neighbourIndex][j] += weight × attributeCoefficients[index][j];

}

}

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

if (updateWeights[i] > 0) {

bias = updateWeights[i] >> 1;

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

attributeCoefficients[index][j] −= (updates[i][j] + bias) / updateWeights[i];

}

}

#### Definition of inversePrediction()

Inputs of this process are:

a variable attrCnt specifying the attribute dimension.

a series of attribute coefficients attributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

an array of quantization weights quantizationWeights[ i ], where i is in the range of 0 to PointNum − 1, inclusive.

an array of prediction predictionWeights[ i ][ n ], where i is in the range of 0 to PointNum − 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.

a variable FixedPointWeightShift specifying the fixed-point representation precision for quantization weights.

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 < attrCnt; 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, FixedPointWeightShift);

}

}

### Predictive Lifting decoding process

Inputs of this process are:

two variables minAttribute and maxAttribute specifying the minimum and maximum allowed attribute values.

a variable attrCnt specifying the attribute dimension.

a variable adaptivePredictionThreshold specifying the threshold to switch to adaptive predictor selection mode.

an array predictorIndexes[q] predictor indexes, where i is in the range of 0 to Npred − 1, inclusive.

[ed. KM define Npred during the decoding process]

a series of quantized attribute coefficients quantAttributeCoefficients[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range 0 to attrCnt − 1, inclusive.

a series of 3D points PointPos[ i ][ j ], where i is in the range of 0 to PointNum − 1, inclusive, and j in the range 0 to 2, inclusive.

a variable levelDetailCount specifying the number of level of detail to be generated.

an array of distances samplingDistanceSquare[ l ], where l is in the range of 0 to levelDetailCount − 1, inclusive.

a variable searchRange specifyingthe search rangefor level of detail generation and the nearest neighbour search.

a variable numPredNearestNeighbours indicating the maximum number of nearest neighbours per point.

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 PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 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 with the parameters PointPos, levelDetailCount, samplingDistanceSquare , searchRange and numPredNearestNeighbours. 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 PointNum − 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 PointNum − 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 PointNum − 1, inclusive, and j in the range of 0 to attrCnt − 1, inclusive.

The reconstructed attributes values are obtained as follows.

q = 0;

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

currentIndex = indexes[i];

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

maxDiff = Max(maxDiff, maxPredAttribute[j] – minPredAttribute[j]);

if (maxDiff > adaptivePredictionThreshold)

predMode = predictorIndexes[ q++ ];

else

predMode = 0;

if ( predMode > 0 ) {

neighbourIndex = neighbours[ index ][ predMode −1 ];

for (j = 1; j < attrCnt; j++)

predicted[j] = attributeValues[ neighbourIndex][ j ];

} else {

for (j = 0; j < attrCnt; 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], FixedPointWeightShift);

}

}

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

res[j] = divExp2RoundHalfInf(unquantAttributeCoefficients[ currentIndex ][ j ],

FixedPointAttributeShift);

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

attributeValue = predicted[j] + res[j] + (j > 0 ? res[0] : 0);

attributeValues[ currentIndex][ j ] = Clip(attributeValue, minAttribute, maxAttribute);

}

}

## Slice concatenation process

The inputs to this process are the arrays pointPos and pointAttr, with elements pointPos[ pointIdx ][ axis ] representing decoded point positions in the current slice and pointAttr[ pointIdx ][ cIdx ] representing corresponding decoded point attributes.

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 < gsh\_num\_points; pointIdx++, RecPicPointCound++) {

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

RecPic[ RecPicPointCount ][ axis ] = pointPos[ pointIdx ][ 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 PayloadBytes represents a coded payload as a sequence of bytes. When parsing the first syntax element of a payload, PayloadBytes is set equal to the byte array provided by an encapsulation format (such as Annex B) or by an external means. The function readPayloadBit( ) provides access to the bitstream as described in 9.2.

The output syntax element value is parsed according to the processes corresponding to the syntax element’s descriptor and name in Table 17 and Table 18.

Table 17 — Descriptor passing process

|  |  |  |
| --- | --- | --- |
| Descriptor | Process | Channel read method |
| u(n) | 9.4.1 | readPayloadBit( ) |
| ue(v) | 9.4.2 | readPayloadBit( ) |
| se(v) | 9.4.2, 9.4.4 | readPayloadBit( ) |
| ae(v) | 9.7.1 | readBin( ) |
| de(v) | 9.6.1 | readBin( ) |

Table 18 — Syntax element specific parsing processes

| **Syntax structure** | **Syntax element** | **Parsing process** |
| --- | --- | --- |
| geometry\_node( ) | geom\_node\_qp\_offset\_eq0\_flag | 9.3.1 (FL), numBins = 1 |
|  | geom\_node\_qp\_offset\_sign\_flag | 9.3.1 (FL), numBins = 1 |
|  | geom\_node\_qp\_offset\_abs\_minus1 | 9.3.2 (EGk), k = 0 |
|  | single\_occupancy\_flag | 9.4.1 (FL), numBins = 1 |
|  | occupancy\_idx[] | 9.4.1 (FL), numBins = 3 |
|  | occypancy\_map | 9.5.5 |
|  | occupancy\_byte | 9.6.1 |
|  | num\_points\_eq1\_flag[ ] | 9.4.1 (FL), numBins = 1 |
|  | num\_points\_minus2[ ] | 9.4.2 (EGk), k = 0 |
|  | is\_planar\_flag [ ] [ ] | 9.4.1 (FL), numBins = 1 |
|  | plane\_position[ ] [ ] | 9.4.1 (FL), numBins = 1 |
|  | direct\_mode\_flag | 9.4.1 (FL), numBins = 1 |
|  | num\_direct\_points\_gt1 | 9.4.1 (FL), numBins = 1 |
|  | not\_duplicated\_point\_flag | 9.3.1 (FL), numBins = 1 |
|  | num\_direct\_points\_eq2\_flag | 9.3.1 (FL), numBins = 1 |
|  | num\_points\_direct\_mode\_minus3 | 9.3.2 (EGk), k = 0 |
|  | point\_offset\_x[ ][ ] point\_offset\_y[ ][ ] point\_offset\_z[ ][ ] | 9.4.1 (FL), numBins = 1 |
| geometry\_trisoup\_data( ) | num\_unique\_segments[ ] | 9.4.2 (EGk), k = 0 |
|  | segment\_indicator[ ] | 9.4.3 (TU), maxVal = 257 |
|  | num\_vertices[ ] | 9.4.2 (EGk), k = 0 |
|  | vertex\_position[ ] | 9.4.3 (TU),  maxVal = ( 1 << trisoup\_node\_size\_log2 ) + 1 |
| attribute\_slice\_data( ) | zerorun | 9.4.3 (TU),  maxVal = TBD |
|  | predIndex | 9.4.3 (TU),  maxVal = max\_num\_direct\_predictors |
| attribute\_coding( ) | isZero | 9.4.1 (FL), numBins = 1 |
|  | isOne | 9.4.1 (FL), numBins = 1 |
|  | remaining\_values[ ][ ] | 9.4.2 (EGk), k = 0 |
| dictionary\_encoded\_value( ) | dict\_lut0\_hit\_flag | 9.4.1 (FL), numBins = 1 |
|  | dict\_lut1\_hit\_flag | 9.4.1 (FL), numBins = 1 |
|  | dict\_lut0\_idx | XXXREF |
|  | dict\_lut1\_idx | 9.4.1 (FL), numBins = 4 |
|  | dict\_direct\_value | 9.4.1 (FL), numBins = 8 |

## Definition of readPayloadBit

The inputs to this process are the current payload byte array PayloadBytes and the associated read position PayloadReadIdx.

The outputs of this process are the next bit read from the payload, and the updated payload read position.

On the first invocation of this process for the current payload, the variable PayloadReadIdx is set equal to 0.

The output value bitVal is determined as follows:

byteIdx = PayloadReadIdx >> 3

bitMask = 0x80 >> ( PayloadReadIdx & 7 )

bitVal = PayloadBytes[ byteIdx ] & bitMask != 0

After determining bitVal, the variable PayloadReadIdx is incremented by one.

## Definition of readBypassBit

The inputs to this process are the current payload byte array PayloadBytes, its length PayloadLength, and the associated bypass sub-stream read position payloadBypassReadIdx.

The outputs of this process are the next bypass bit read from the payload, and an updated payload read position.

On the first invocation of this process for the current payload, the variable payloadBypassReadIdx is set equal to PayloadLength.

The output value bitVal is determined as follows:

byteIdx = payloadBypassReadIdx >> 3

bitMask = 0x80 >> (payloadBypassReadIdx & 7)

bitVal = PayloadBytes[byteIdx] & bitMask != 0

After determining bitVal, the variable payloadBypassReadIdx is decremented 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 19 illustrates an example of the mapping process.

Table 19 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 |
| ... | ... |

## Bitwise geometry octree occupancy parsing process

[Ed: todo— decide if these subclauses should be grouped together or not]

### General process

The parsing and inverse binarization of the arithmetically coded syntax element occupancy\_map is described in 9.5.5

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 (9.5.7) based upon the occupancy state of neighbouring nodes, predicted occupancy values ((9.5.9) and previously decoded bins. After decoding a bin, CtxMap is updated based upon the decoded bin value (9.5.8).

At the start of decoding a geometry payload, CtxMap is initialised according to 9.5.2.

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 payload.

The output from this process is the initialised 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

ADD inference of mask by QTBT !

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 20, and readOccBin() is specified by 9.5.6,

Table 20 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] [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 9.4.5 with the variables NeighbourPattern, BinIdx, and PartialSynVal as input.

The arithmetic decoding process 9.7.2 for a single bin is invoked for the syntax element occupancy\_map with the variable ctxIdx as input. [Ed:. to determine binVal?]

The context map update process 9.5.8 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 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: [Ed. XXX this needs to take into account the effective node size!]

If log2\_intra\_pred\_max\_node\_size is equal to zero, the variable idxPredidxPred is set equal to 0.

Otherwise, log2\_intra\_pred\_max\_node\_size is not equal to zero, the variable idxPred is set equal to the output of the occupancy prediction process using neighbouring octree nodes (9.5.9) 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 17.

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 initialised to 0.

The variables xC, yC, and zC identifying the position of the child node associated with binIdx at depth + 1 are initialised as follows

xC = 2 × xN + ((bitCodingOrder[ binIdx ] >> 2) & 1)

yC = 2 × yN + ((bitCodingOrder[ binIdx ] >> 1) & 1)

zC = 2 × zN + (bitCodingOrder[ binIdx ] & 1)

The following procedure is performed for each of the x, y, and z axes by substituting the variables aN, aC, nPmas, xCn, yCn, zCn, xNn, yNn, and zNn of the corresponding row of Table 21.

// if child is adjacent to a causally-valid neighbour

if ( !(aC & 1) ) {

if (NeighboutPattern & nPmask)

adjOcc += GeometryNodeOccupancyCnt[ depth + 1 ][ xCn ][ yCn ][ zCn ]

else

// if neighbour is available but not present

if ((aN + 1) & NeighbAvailabilityMask != 1)

if (GeometryNodeOccupancyCnt[ depth ][ xNn ][ yNn ][ zNn ] = = 0)

adjUnocc = 1

}

Table 21 — Variable substitutions for the computation of adjOcc and adjUnocc

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| axis | aN | aC | nPmask | xCn | yCn | zCn | xNn | yNn | zNn |
| x | xN | xC | 2 | xC−1 | yC | zC | xN−1 | yN | zN |
| y | yN | yC | 4 | xC | yC−1 | zC | xN | yN−1 | zN |
| z | zN | zC | 16 | xC | yC | zC−1 | xN | yN | zN−1 |

The variable idxAdj is derived as follows

idxAdj = adjUnocc + 2 × Min(2, adjOcc)

if (binIdx > 4) {

idxAdj = ctxIdxAdjReduc567[ idxAdj ]

}

Table 22 — 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 23 — 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 24 — Values of neighbourPattern64to6[ j + i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | i | | | | | | | | | | | | | | | |
| j | 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 25 — 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 26 — 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 27.

Table 27 — 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 xN, yN, zN, 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 blocks is determined as follows:

for (i = 0; i < 25; i++) {

x = xN + dX[ i ]

y = yN + dY[ i ]

z = zN + dZ[ i ]

if (available(xN, yN, zN, x, y, z))

occupied[ i ] = GeometryNodeOccupancyCnt[ depth ][ x ][ y ][ z ] != 0

else

occupied[ i ] = 0

}

Where the function available(xN, yN, zN, x,y, z) evaluates to true if all of the following conditions are true:

log2\_neighbour\_avail\_boundary > 0

(x ^ xN) >> log2\_neighbour\_avail\_boundary = = 0

(y ^ yN) >> log2\_neighbour\_avail\_boundary = = 0

(z ^ zN) >> log2\_neighbour\_avail\_boundary = = 0

And where the values of the neighbour position offsets dX[ ], dY[ ], and dz[ ] are given in Table 28.

If the sum of occupied[i],i =0 to 25 inclusive, is less than 8, the output predicted occupancy state is set equal to zero and no further processing occurs.[Ed: Sigma\_{i=0}^{25} occupied[i] < 8?]

An occupancy score for the child node is determined as follows:

Where the values of scoreIdx[ ][ ], and predictionScore[ ][ ] are given by Table 28 and Table 29.

The output predicted occupancy state, prediction, is set according to the following:

thresholdIdx = Min(numOccupied − 8, 4);

if (score <= predictionThreshold[ thresholdIdx ][ 0 ])

prediction = 1;

else if (score >= predictionThreshold[ thresholdIdx ][ 1 ])

prediction = 2;

else

prediction = 0;

Where the value of predictionThreshold[ ][ ] is given by Table 30.

Table 28 — Values of dX[ i ], dY[ i ], dZ[ i ], and scoreIdx[ bitIdx ][ i ] for intra occupancy prediction

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **scoreIdx[ childIdx ][ i ]** | | | | | | | |
| **i** | **dX[ i ]** | **dY[ i ]** | **dZ[ i ]** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 0 | −1 | −1 | −1 | 2 | 4 | 4 | 6 | 4 | 6 | 6 | 7 |
| 1 | −1 | −1 | 0 | 1 | 1 | 3 | 3 | 3 | 3 | 5 | 5 |
| 2 | −1 | −1 | 1 | 4 | 2 | 6 | 4 | 6 | 4 | 7 | 6 |
| 3 | −1 | 0 | −1 | 1 | 3 | 1 | 3 | 3 | 5 | 3 | 5 |
| 4 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 |
| 5 | −1 | 0 | 1 | 3 | 1 | 3 | 1 | 5 | 3 | 5 | 3 |
| 6 | −1 | 1 | −1 | 4 | 6 | 2 | 4 | 6 | 7 | 4 | 6 |
| 7 | −1 | 1 | 0 | 3 | 3 | 1 | 1 | 5 | 5 | 3 | 3 |
| 8 | −1 | 1 | 1 | 6 | 4 | 4 | 2 | 7 | 6 | 6 | 4 |
| 9 | 0 | −1 | −1 | 1 | 3 | 3 | 5 | 1 | 3 | 3 | 5 |
| 10 | 0 | −1 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 2 |
| 11 | 0 | −1 | 1 | 3 | 1 | 5 | 3 | 3 | 1 | 5 | 3 |
| 12 | 0 | 0 | −1 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 |
| 13 | 0 | 0 | 1 | 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 14 | 0 | 1 | −1 | 3 | 5 | 1 | 3 | 3 | 5 | 1 | 3 |
| 15 | 0 | 1 | 0 | 2 | 2 | 0 | 0 | 2 | 2 | 0 | 0 |
| 16 | 0 | 1 | 1 | 5 | 3 | 3 | 1 | 5 | 3 | 3 | 1 |
| 17 | 1 | −1 | −1 | 4 | 6 | 6 | 7 | 2 | 4 | 4 | 6 |
| 18 | 1 | −1 | 0 | 3 | 3 | 5 | 5 | 1 | 1 | 3 | 3 |
| 19 | 1 | −1 | 1 | 6 | 4 | 7 | 6 | 4 | 2 | 6 | 4 |
| 20 | 1 | 0 | −1 | 3 | 5 | 3 | 5 | 1 | 3 | 1 | 3 |
| 21 | 1 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| 22 | 1 | 0 | 1 | 5 | 3 | 5 | 3 | 3 | 1 | 3 | 1 |
| 23 | 1 | 1 | −1 | 6 | 7 | 4 | 6 | 4 | 6 | 2 | 4 |
| 24 | 1 | 1 | 0 | 5 | 5 | 3 | 3 | 3 | 3 | 1 | 1 |
| 25 | 1 | 1 | 1 | 7 | 6 | 6 | 4 | 6 | 4 | 4 | 2 |

Table 29 — Values of predictionScore[ i ][ occupied ]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **i** | | | | | | | |
| **occupied** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 0 | −4 | −24 | 48 | 80 | 56 | 112 | 88 | 48 |
| 1 | 108 | 156 | 80 | 32 | 72 | 16 | 44 | 72 |

Table 30 — Values of predictionThreshold[ i ][ j ]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **i** | | | | |
| **occupied** | **0** | **1** | **2** | **3** | **4** |
| 0 | 1612 | 1560 | 1586 | 1534 | 1534 |
| 1 | 1742 | 1716 | 1690 | 1716 | 1664 |

## Dictionary-based parsing

### General process

This process is invoked when parsing syntax elements with descriptor ae(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 9.6.2 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 9.6.3.

All the binary arithmetic contexts are initialized by invoking the process in clause 9.7.4.2.

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]);

}

### Dictionary encoded value semantics

<TBD>

## 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 9.7.3.2 and 9.7.4.2 are invoked when starting parsing 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 15.

### 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 31.

If the value of ctxIdx is not equal to the value 'bypass', the following applies:

* The arithmetic decoding process 9.7.4.3 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 9.7.3.3 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 9.7.4.4 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 readBypassBit process 9.3 is invoked to determine the value of the decoded bin.

Table 31 — Values of ctxTbl and ctxIdx for binarised 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 |
| occypancy\_map | 1 | 0 … 31 (9.5.7) |
| num\_points\_eq1\_flag[ ] | 2 | 0 |
| num\_points\_minus2[ ] | 3 | prefix:0 suffix: bypass |
| is\_planar\_flag [ ] [ ] | ? | 0 ... 11 planarIdx (8.2.4.3) |
| plane\_position[ ] [ ] | ? | 0 ... 36 planePosIdx (8.2.4.4) |
| direct\_mode\_flag | 4 | 0 |
| num\_direct\_points\_gt1 | ? | 0 |
| not\_duplicated\_point\_flag | 2 | 0 |
| num\_direct\_points\_eq2\_flag | ? | 0 |
| num\_points\_direct\_mode\_minus3 | 3 | prefix:0 suffix: bypass |
| point\_offset\_x[ ][ ] point\_offset\_y[ ][ ] point\_offset\_z[ ][ ] | na | bypass |
| num\_unique\_segments[ ] | na | prefix: bypass suffix: bypass |
| segment\_indicator[ ] | 6 | BinIdx |
| num\_vertices[ ] | na | prefix: bypass suffix: bypass |
| vertex\_position[ ] | 7 | BinIdx |
| zerorun | 8 | 0 … 2 |
| predIndex | 9 | Min(BinIdx, 1) |
| isZero | 10 |  |
| isOne | 11 |  |
| remaining\_values[ ][ ] | 12 | 0 .… 6 |
| 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 32 — 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 |
| values[ ][ ], k2 = = 0 | 18 | 19 | 20 | 21 | 22 |
| values[ ][ ], k2 = = 1 | 23 | 24 | 25 | 26 | 27 |

### 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.

[Ed: todo -- Contexts[][] array definition]

#### Initialisation of context variables

The outputs of this process are initialised CABAC state variables.

All context variables of the arithmetic decoding engine are initialised 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 33.

Table 33 — 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 initialised arithmetic decoding engine variables ivlLow, ivlRange, and ivlCode.

At the start of the decoding of any data unit, the arithmetic decoding state shall be initialised as follows:

ivlLow = 0;

ivlRange = 0xffff;

ivlCode = 0;

for (i = 0; i < 15; i++) {

ivlCode <<= 1;

ivlCode += readPayloadBit( );

}

#### 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) | readPayloadBit( )) & 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 payload bytestream.

This informative clause describes an arithmetic encoding engine that matches the arithmetic decoding engine described in 9.7.4. The encoding engine is essentially symmetric with the decoding engine, i.e., procedures are called in the same order. Table 34 illustrates the correspondence between decoding and encoding processes.

Table 34 — Correspondence between decoder and encoder arithmetic coding processes

|  |  |  |
| --- | --- | --- |
| **Process** | **Decoder** | **Encoder** |
| Initialisation | 9.7.4.2 | 9.7.5.2 |
| Symbol coding | 9.7.4.3 | 9.7.5.3 |
| Renormalisation | 9.7.4.5 | 9.7.5.4 |
| Termination | — | 9.7.5.5 |

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) symbol of a payload.

The outputs of this process are the arithmetic encoding engine variables ivlLow, ivlRange, and ivlCarry, initialised 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 payload 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( );

1. Profiles and levels
   1. Overview of profiles and levels

<To do>

1. Type-length-value bytestream format

[Ed: Decide if this should be normative it is just the format used by the software]

**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 payloads 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.

[Ed: todo geometry must become before attribute?]

tlv\_type identifies the syntax structure represented by tlv\_payload\_byte[ ] according to Table 35

Table 35 — Mapping of tlv\_type and associated payload to syntax table

|  |  |  |
| --- | --- | --- |
| tlv\_type | Syntax table | Description |
| 0 | 7.3.1.1 | Sequence parameter set |
| 1 | 7.3.1.2 | Geometry parameter set |
| 2 | 7.3.2.1 | Geometry payload |
| 3 | 7.3.1.3 | Attribute parameter set |
| 4 | 7.3.3.1 | Attribute payload |
| 5 | 7.3.2.2 | Tile inventory |
| 6 | 7.3.2.5 | Frame boundary marker |

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 payload 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 PayloadBytes is set equal to tlv\_payload\_byte[ ],

the variable NumPayloadBytes is set equal to tlv\_num\_payload\_bytes,

the parsing process in Table 18 corresponding to tlv\_type is invoked.

1. Bibliography

[1] ISO/IEC 23091, Information Technology — MPEG Coding-independent code-points