

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

DIS stage

Warning for WDs and CDs

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

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

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

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

© ISO/IEC 2020

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

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Foreword	5
Introduction.....	6
1 Scope	1
2 Normative references	1
3 Terms and definitions.....	1
3.1 General	1
3.2 Geometry coding related.....	3
3.3 Attribute coding related.....	4
4 Abbreviations.....	4
5 Conventions	5
5.1 General	5
5.2 Numerical representation.....	5
5.3 Arithmetic operators.....	5
5.4 Logical operators.....	6
5.5 Relational operators.....	6
5.6 Bit-wise operators.....	6
5.7 Assignment operators.....	6
5.8 Range notation.....	7
5.9 Mathematical functions.....	7
5.9.1 Definition of iAtan2	7
5.9.2 Definition of popCnt	8
5.9.3 Definition of iLog2.....	8
5.9.4 Definition of iSqrt.....	8
5.9.5 Definition of inverse square root function invSqrt.....	9
5.9.6 Definition of divExp2RoundHalfInf	11
5.9.7 Definition of divExp2RoundHalfUp	11
5.9.8 Conversion of a tuple to 3D Morton code (TupleToMorton)	11
5.9.9 Conversion of 3D Morton codes to a tuple (MortonToTuple)	12
5.9.10 Definition of QpToQstep	12
5.10 Vector operations.....	12
5.11 Order of operation precedence	13
5.12 Variables, syntax elements, and tables.....	13
5.13 Text description of logical operations	15
5.14 Processes.....	16
6 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships.....	16
6.1 Bitstream formats	16
6.2 Source, decoded, and output point cloud formats.....	16
6.2.1 Data partitioning	17
6.2.2 Frame index attribute component.....	17
6.3 Geometry octree.....	17
6.4 Neighbour relationships	17
6.4.1 Neighbour dependent geometry octree child node scan order inverse mapping process	17
6.4.2 Neighbour depending geometry occupancy map permutation process	18
7 Syntax and semantics	19
7.1 Method of specifying syntax in tabular form	19
7.2 Specification of syntax functions and descriptors.....	20
7.3 Syntax in tabular form	21

7.3.1	General	21
7.3.2	Data unit and byte alignment syntax.....	21
7.3.3	Geometry data unit syntax.....	25
7.3.4	Attribute data unit syntax	29
7.4	Semantics.....	31
7.4.1	General	31
7.4.2	Data unit and byte alignment semantics.....	32
7.4.3	Geometry data unit semantics.....	40
7.4.4	Attribute data unit semantics	49
8	Decoding process.....	51
8.1	General decoding process.....	51
8.2	Geometry decoding process.....	51
8.2.1	General geometry decoding process	51
8.2.2	Octree decoding process.....	51
8.2.3	Geometry Trisoup decoding process.....	53
8.2.4	Planar coding mode	62
8.2.5	Angular coding mode	66
8.3	Attribute decoding	68
8.3.1	Region adaptive hierachical transform decoding process.....	68
8.3.2	LoD with Lifting Transform decoding process.....	77
8.3.3	LoD with Predicting Transform decoding process.....	87
8.4	Slice concatenation process	88
9	Parsing process	89
9.1	General	89
9.2	Chunked bytestream parsing process.....	91
9.2.1	General.....	91
9.2.2	Syntax	92
9.2.3	Semantics	92
9.3	Definition of readDataUnitBit.....	93
9.4	Definition of readAeStreamBit	93
9.5	Definition of readBypassStreamBit	93
9.6	General inverse binarisation processes.....	94
9.6.1	Parsing of fixed-length codes	94
9.6.2	Parsing of k-th order exp-Golomb codes	94
9.6.3	Parsing of truncated unary codes.....	94
9.6.4	Mapping process for signed codes	94
9.7	Bit-wise geometry octree occupancy parsing process	95
9.7.1	General process.....	95
9.7.2	Initialisation process	95
9.7.3	Determination of planar masks used in the inverse binarization process	95
9.7.4	Occupancy_idx[] parsing process	96
9.7.5	Inverse binarization process.....	96
9.7.6	Definition of readOccBin().....	98
9.7.7	ctxMapIdx and ctxIdx derivation processes	98
9.7.8	Context map update process	100
9.7.9	Occupancy prediction process using neighbouring octree nodes	101
9.8	Inferred Direct Coding Mode parsing process	103
9.8.1	General process.....	103
9.8.2	Determination of the angular context idcmIdxAngular	103
9.8.3	Inverse binarization process.....	104
9.9	Dictionary-based parsing	104
9.9.1	General process.....	104
9.9.2	Initializing lut0.....	106
9.9.3	Initializing lut1.....	106
9.9.4	Definition of decodeLut0Index()	106

9.9.5	Definition of pushLut0()	107
9.9.6	Definition of updateLut0()	107
9.9.7	Definition of lut0ComputeMostFrequentSymbols()	107
9.9.8	Definition of pushLut1()	107
9.10	CABAC parsing process	108
9.10.1	General	108
9.10.2	Definition of readBin()	108
9.10.3	Context variables	110
9.10.4	Arithmetic decoding engine	111
9.10.5	Arithmetic encoding engine (informative)	113
9.11	Parsing state memorization process	115
9.12	Parsing state restoration process	115
	Annex A Profiles and levels	116
A.1	Overview of profiles and levels	116
A.2	Requirements on decoder capability	116
A.3	Profiles	116
A.3.1	General	116
A.3.2	Main profile	116
A.4	Levels	117
A.4.1	Level limits	117
	Annex B Type-length-value bytestream format	118

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).

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).

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.

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.

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

1 Scope

This document specifies geometry-based point cloud compression.

2 Normative references

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

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

3 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/>

3.1 General

3.1.1

point

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

3.1.2

point cloud frame

set of points at a particular time instance

3.1.3

point cloud

sequence of point cloud frames

3.1.4

Cartesian co-ordinates

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

3.1.5

geometry

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

3.1.6

attribute

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

3.1.7

may

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

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

3.1.8

must

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

3.1.9

shall

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

3.1.10

should

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

3.1.11

informative

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

3.1.12

byte

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

3.1.13

byte-aligned

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

3.1.14

unspecified

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

3.1.15

syntax element

element of data represented in the *bitstream*.

3.1.16

bitstream

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

3.1.17

coded point cloud frame

a coded representation of a point cloud frame

3.1.18

syntax structure

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

3.1.19

bounding box

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

3.1.20

3D tile

rectangular cuboid inside a bounding box.

3.1.21

slice

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

3.2 Geometry coding related

3.2.1

position

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

3.2.2

octree

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

3.2.3

node

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

3.2.4

root node

node of the octree with no parent

3.2.5

leaf node

terminating node of the octree having no children

3.2.7

level

number of hops from the root to the node.

3.2.8

occupied node

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

3.2.9

occupancy code

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

3.2.10

Morton code

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

3.3 Attribute coding related

3.3.1

Colour

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

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

3.3.2

Reflectance

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

3.3.3

Frame index

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

3.3.4

Material ID

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

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

3.3.5

Transparency

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

3.3.7

Normals

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

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

4 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

5 Conventions

5.1 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.

5.2 Numerical representation

The following numerical representation format are defined.

binary representation	formatted as 0bXXX... where each digit X is 0 or 1
octal representation	formatted as 0oXXX... where each digit X is 0 to 7
decimal representation	formatted as XXX... where each digit X is 0 to 9
hexadecimal representation	formatted as 0xXXX... where each digit X is 0 to 9 or a to f

5.3 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
x^y	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.
$\frac{x}{y}$	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\sum_{i=x}^y f(i)$	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 $x \geq 0$ and $y > 0$.

5.4 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`.

5.5 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.

5.6 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.

5.7 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 $x = x - (-3)$.

5.8 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 .

5.9 Mathematical functions

The following mathematical functions are defined:

$$\text{Abs}(x) = \begin{cases} x & ; \quad x \geq 0 \\ -x & ; \quad x < 0 \end{cases}$$

$\text{Ceil}(x)$ the smallest integer greater than or equal to x .

$$\text{Clip1}_V(x) = \text{Clip3}(0, (1 \ll \text{BitDepth}_V) - 1, x)$$

$$\text{Clip1}_C(x) = \text{Clip3}(0, (1 \ll \text{BitDepth}_C) - 1, x)$$

$$\text{Clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases}$$

$\text{Floor}(x)$ the largest integer less than or equal to x .

$$\text{Min}(x, y) = \begin{cases} x & ; \quad x \leq y \\ y & ; \quad x > y \end{cases}$$

$$\text{Max}(x, y) = \begin{cases} x & ; \quad x \geq y \\ y & ; \quad x < y \end{cases}$$

$$\text{Sign}(x) = \begin{cases} 1 & ; \quad x > 0 \\ 0 & ; \quad x = 0 \\ -1 & ; \quad x < 0 \end{cases}$$

$$\text{Sqrt}(x) = \sqrt{x}$$

$$\text{Swap}(x, y) = (y, x)$$

5.9.1 Definition of iAtan2

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

The output of this process is the variable t .

The derivation process for $t = i\text{Atan2}(a, b)$ is defined as follows.

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

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

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

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

Otherwise, following steps apply:

```
c = Abs((b << 8) / a)
if (c <= 256)
    idx = c / 12
else
    idx = c > 40 ? 40 : c

t = atanLut[idx]
if (a < 0 && b > 0)
    t += 402
else if (a < 0 && b < 0)
    t += 804
else if (a > 0 && b < 0)
    t += 1206
```

The array atanLut is defined in Table 1.

Table 1 — the value of atanLut[i+j]

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

5.9.2 Definition of popCnt

The input to this process is the integer variable x.

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

5.9.3 Definition of iLog2

The input to this process is the variable x.

The output of this process is the variable y.

The function iLog2 is defined as follows:

```
y = Floor(Log(x) ÷ Log(2))
```

where Log() is the natural logarithmic function.

5.9.4 Definition of iSqrt

The input to this process is the variable pIn.

The output of this process is the variable pOut.

The variables x and n are derived as follows.


```

for (n = 8; n <= 64; n+= 8) {
  if (pIn >= (1 << (64 - n))) {
    x = (tableSqrt[pIn >> (64 - n)] << (32 - (n / 2))) - (n == 8 ? 1 : 0)
    break;
  }
}

```

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

Finally, pOut is derived as follows.

```

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

```

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	11	11	11	11	11	11	11	11	11	11
100	10	10	11	11	11	11	11	11	11	11	12	12	12	12	12	12	12	12	12	12
120	11	11	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	13	13
140	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	14	14
160	13	13	14	14	14	14	14	14	14	14	15	15	15	15	15	15	15	15	15	15
180	14	14	15	15	15	15	15	15	15	15	16	16	16	16	16	16	16	16	16	16
200	15	15	16	16	16	16	16	16	16	16	17	17	17	17	17	17	17	17	17	17
220	16	16	17	17	17	17	17	17	17	17	18	18	18	18	18	18	18	18	18	18
240	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19

5.9.5 Definition of inverse square root function invSqrt

The input to this process is the variable pIn.

The output of this process is the variable pOut.

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

```

shift = -3;
pInScaled = pIn;

```

```

while (pIn & 0xffffffff00000000) {
    pInScaled >>= 2;
    nShift--;
}
while (!(pInScaled & 0xc0000000)) {
    pInScaled <<= 2;
    nShift++;
}

```

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

```

idx = (pInScaled >> 25) - 32;
invSqrtApprox = threeTimesR[idx] - ((rCubed[idx] × pInScaled) >> 32);

```

A second approximation, `invSqrtApprox2`, is obtained as follows:

```

s = (invSqrtApprox × pInScaled) >> 32;
s = 0x30000000 - ((invSqrtApprox × s) >> 32);
invSqrtApprox2 = (invSqrtApprox × s) >> 32;

```

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

```

if (nShift >= 0)
    pOut = invSqrtApprox2 << nShift;
else
    pOut = invSqrtApprox2 >> (-nShift);

```

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

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

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

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

5.9.6 Definition of divExp2RoundHalfInf

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

The output of this process is the variable value approximating $\text{scalar}/2^{\text{shift}}$, computed as follows:

```

if (!shift) {
    value = scalar;
} else {
    s0 = 1 << (shift - 1);
    value = scalar >= 0 ? (s0 + scalar) >> shift : -((s0 - scalar) >> shift)
}

```

5.9.7 Definition of divExp2RoundHalfUp

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

The output of this process is the variable value approximating $\text{scalar}/2^{\text{shift}}$, computed as follows:

```

if (!shift) {
    value = scalar;
} else {
    s0 = 1 << (shift - 1);
    value = (s0 + scalar) >> shift;
}

```

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

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

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

$$m = \sum_i 2^{3i+2} [s \& 2^i] + 2^{3i+1} [t \& 2^i] + 2^{3i} [v \& 2^i]$$

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

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

Bit string form				Integer form
<i>s</i>	<i>t</i>	<i>v</i>	<i>m</i>	<i>m</i>
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
<i>s_n ... s₁ s₀</i>	<i>t_n ... t₁ t₀</i>	<i>v_n ... v₁ v₀</i>	<i>s_n t_n v_n ... s₁ t₁ v₁ s₀ t₀ v₀</i>	...

5.9.9 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:

$$\begin{aligned} s &= \sum_i 2^i [m \& 2^{3i+2}] \\ t &= \sum_i 2^i [m \& 2^{3i+1}] \\ u &= \sum_i 2^i [m \& 2^{3i}] \end{aligned}$$

5.9.10 Definition of QpToQstep

The inputs to this process are:

the variable *qP*, representing the quantization parameter.

the variable *isFirstComp*

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

```

if (isFirstComp)
    qpBdOffset = 6 × (attribute_bitdepth_minus1[ash_attr_sps_attr_idx] - 7)
else
    qpBdOffset = 6 × (attribute_secondary_bitdepth_minus1[ash_attr_sps_attr_idx] - 7)

qP' = Clip3(4, 51 + qpBdOffset, qP);
qstep = levelScale[qP' % 6] << (qP' / 6);

```

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

5.10 Vector operations

The following mathematical functions are defined:

The function $c[i] = \text{CrossProduct}(a[i], b[i])$ with $i = 0 \dots 2$ is defined as follows:

$$\begin{aligned} c[0] &= a[1] \times b[2] - a[2] \times b[1] \\ c[1] &= a[2] \times b[0] - a[0] \times b[2] \\ c[2] &= a[0] \times b[1] - a[1] \times b[0] \end{aligned}$$

The function $c = \text{InnerProduct}(a[i], b[i])$ with $i = 0 \dots 2$ is defined as follows:

$$c = a[0] \times b[0] + a[1] \times b[1] + a[2] \times b[2]$$

5.11 Order of operation precedence

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

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

Table 6 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

Table 6 – Operation precedence from highest (at top of table) to lowest (at bottom of table)

operations (with operands x, y, and z)
"x++", "x--"
"!x", "-x" (as a unary prefix operator)
x^y
"x × y", "x / y", "x ÷ y", " $\frac{x}{y}$ ", "x % y"
"x + y", "x - y" (as a two-argument operator), " $\sum_{i=x}^y f(i)$ "
"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"

5.12 Variables, syntax elements, and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower-case letters with underscore characters), and one descriptor for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the

values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e., not bold) type.

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

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

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

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

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

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

5.13 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

5.14 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.

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

6.1 Bitstream formats

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

6.2 Source, decoded, and output point cloud formats

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

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

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

- Geometry information – cartesian co-ordinates of the occupied point in 3-dimensional space (0 1 2, also known as XYZ).
- Single stimulus (Luma only, Reflectance).
- Colour, for example Green, Blue and Red (GBR, also known as RGB).
- Arrays representing other unspecified monochrome or multi-stimulus attribute samplings (for example, Frame index, Transparency).

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

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

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

6.2.1 Data partitioning

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

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

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

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

6.2.2 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.

6.3 Geometry octree

When the geometry octree is used, then the geometry encoding proceeds as follows. First, a cubical axis-aligned bounding box B is defined by the two extreme points $(0, 0, 0)$ and $(2^d, 2^d, 2^d)$. An octree structure is then built by recursively subdividing B. At each stage, a cube is subdivided into 8 sub-cubes. An 8-bit code, named an occupancy code, is then generated by associating a 1-bit value with each sub-cube in order to indicate whether it contains points (i.e., full and has value 1) or not (i.e., empty and has value 0). Only full sub-cubes with a size greater than 1 (i.e., non-voxels) are further subdivided. Since points may be duplicated, multiple points may be mapped to the same sub-cube of size 1 (i.e., the same voxel). In order to handle such a situation, the number of points for each sub-cube of dimension 1 is also arithmetically encoded. The same arithmetic encoder is used to encode all the information put into the bitstream.

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

6.4 Neighbour relationships

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

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

The inputs to this process are

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

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

The output index is determined as follows

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

Values of the array childScanMap are given by Table 7.

Table 7 — Values of childScanMap[i + j]

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

6.4.2 Neighbour depending geometry occupancy map permutation process

The inputs to this process are

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

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

The output is derived as follows

```
occMapP = 0
for (srcIdx = 0; srcIdx < 8; srcIdx++) {
    dstIdx = (childScanMap[neighbourPattern] >> (srcIdx × 3)) & 7
    occMapP = occMapP | (((occMap >> srcIdx) & 1) << dstIdx)
}
```

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

7 Syntax and semantics

7.1 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	

7.2 Specification of syntax functions and descriptors

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

byte_aligned() is specified as follows:

- If the current position in the bitstream is on a byte boundary, i.e. the next bit in the bitstream is the first bit in a byte, the return value of byte_aligned() is equal to TRUE.
- Otherwise, the return value of byte_aligned() is equal to FALSE.

more_data_in_byte_stream(), which is specified as follows:

- If more data follow in the byte stream, the return value of more_data_in_byte_stream() is equal to TRUE.

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

- ae(v): adaptive arithmetic entropy-coded syntax element.
- de(v): dictionary coded syntax element.
- s(n): signed integer using n bits plus sign bit.
- se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first.
- u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read_bits(n) interpreted as a binary representation of an unsigned integer with most significant bit written first.
- ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first.

7.3 Syntax in tabular form

7.3.1 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.

7.3.2 Data unit and byte alignment syntax

7.3.2.1 Sequence parameter set syntax

<code>seq_parameter_set() {</code>	Descriptor
main_profile_compatibility_flag	u(1)
reserved_profile_compatibility_22bits	u(22)
unique_point_positions_constraint_flag	u(1)
level_idc	u(8)
sps_seq_parameter_set_id	ue(v)
sps_bounding_box_present_flag	u(1)
if(sps_bounding_box_present_flag) {	
for(k = 0; k < 3; k++)	
sps_bounding_box_offset_xyz[k]	se(v)
sps_bounding_box_offset_log2_scale	ue(v)
for(k = 0; k < 3; k++)	
sps_bounding_box_size_xyz[k]	ue(v)
}	
sps_source_scale_factor_numerator_minus1	ue(v)
sps_source_scale_factor_denominator_minus1	ue(v)
sps_num_attribute_sets	ue(v)
for(i = 0; i < sps_num_attribute_sets; i++) {	
attribute_instance_id[i]	ue(v)
attribute_dimension_minus1[i]	ue(v)
attribute_bitdepth_minus1[i]	ue(v)
if(attribute_dimension_minus1[i] > 0)	
attribute_secondary_bitdepth_minus1[i]	ue(v)
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_flag	u(1)
if(sps_extension_flag)	
while(more_data_in_byte_stream())	
sps_extension_data_flag	u(1)
byte_alignment()	
}	

7.3.2.2 Tile inventory syntax

tile_inventory() {	D
	e
	s
	c
	r
	i
	p
	t
	o
	r
tile_frame_idx	t
	b
	u
num_tiles	u
	(
	1
	6
)
tile_bounding_box_bits	u
	(
	8
)
for(i = 0; i < num_tiles; i++) {	
for(k = 0; k < 3; k++)	
tile_bounding_box_offset_xyz[i][k]	s
	(
	v
)
for(k = 0; k < 3; k++)	
tile_bounding_box_size_xyz[i][k]	u
	(
	v
)
}	
byte_alignment()	
}	

7.3.2.3 Geometry parameter set syntax

geometry_parameter_set() {	Descriptor
gps_geom_parameter_set_id	ue(v)
gps_seq_parameter_set_id	ue(v)
gps_gsh_box_present_flag	u(1)
if(gps_gsh_box_present_flag){	
gps_gsh_box_log2_scale_present_flag	u(1)
if(!gps_gsh_box_log2_scale_present_flag)	
gps_gs_box_log2_scale	ue(v)
}	
unique_geometry_points_flag	u(1)
geometry_planar_mode_flag	u(1)
if(geometry_planar_mode_flag){	
geom_planar_mode_th_idcm	ue(v)
geom_planar_mode_th[0]	ue(v)
geom_planar_mode_th[1]	ue(v)
geom_planar_mode_th[2]	ue(v)
geometry_angular_mode_flag	u(1)
}	
if(geometry_angular_mode_flag){	
for(k = 0; k < 3; k++)	
geom_angular_origin_xyz[k]	se(v)
number_lasers_minus1	ue(v)
for(i = 0; i <= number_lasers_minus1; i++) {	
laser_angle[i]	se(v)
laser_correction[i]	se(v)
}	
planar_buffer_disabled	u(1)
implicit_qtbt_angular_max_node_min_dim_log2_to_split_v	se(v)
implicit_qtbt_angular_max_diff_to_split_v	se(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_minus4	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_flag	u(1)
if(gps_extension_flag)	
while(more_data_in_byte_stream())	
gps_extension_data_flag	u(1)
byte_alignment()	
}	

7.3.2.4 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)
if(attr_coding_type == 0) { //RAHT	
raht_prediction_enabled_flag	u(1)
if (raht_prediction_enabled_flag) {	
raht_prediction_threshold0	ue(v)
raht_prediction_threshold1	ue(v)
}	
}	
else if (attr_coding_type <= 2) {	
lifting_num_pred_nearest_neighbours_minus1	ue(v)
lifting_search_range_minus1	ue(v)
for(k = 0; k < 3; k++)	
lifting_neighbour_bias_xyz[k]	ue(v)
if (attr_coding_type == 2)	
lifting_scalability_enabled_flag	u(1)
if (! lifting_scalability_enabled_flag) {	
lifting_num_detail_levels_minus1	ue(v)
if (lifting_num_detail_levels_minus1 > 0) {	
lifting_lod_regular_sampling_enabled_flag	u(1)
for(idx = 0; idx < num_detail_levels_minus1; idx++)	
{	
if (lifting_lod_regular_sampling_enabled_flag)	
lifting_sampling_period_minus2[idx]	ue(v)
else	
lifting_sampling_distance_squared_scale_minus1[idx]	ue(v)

if (idx != 0)	
lifting_sampling_distance_squared_offset [idx]	ue(v)
}	
}	
}	
if(attr_coding_type == 1) {	
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)
}	
}	
aps_extension_flag	u(1)
if(aps_extension_flag)	
while(more_data_in_byte_stream())	
aps_extension_data_flag	u(1)
byte_alignment()	
}	

7.3.2.5 Frame boundary marker syntax

frame_boundary_marker() {	Descriptor
/* this syntax structure is intentionally empty */	
}	

7.3.2.6 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)
}	

7.3.3 Geometry data unit syntax

7.3.3.1 General geometry data unit syntax

geometry_data_unit () {	Descriptor
geometry_data_unit_header()	
geometry_data_unit_data()	
}	

7.3.3.2 Geometry data unit header syntax

geometry_data_unit_header() {	Descriptor
gsh_num_points_minus1	u(24)
gsh_geometry_parameter_set_id	ue(v)
gsh_tile_id	ue(v)
gsh_slice_id	ue(v)
frame_idx	u(v)
if(gps_gsh_box_present_flag) {	
if(gps_gsh_box_log2_scale_present_flag)	
gsh_box_log2_scale	ue(v)
for(k = 0; k < 3; k++)	
gsh_box_origin_xyz[k]	ue(v)
}	
if (gps_implicit_geom_partition_flag) {	
gsh_log2_root_node_size_s	ue(v)
gsh_log2_root_node_size_t_minus_s	se(v)
gsh_log2_root_node_size_v_minus_t	se(v)
} else {	
gsh_log2_root_node_size	ue(v)
}	
gsh_num_entropy_streams_minusQ	ue(v)
if(gsh_num_entropy_streams_minusQ) {	
gsh_entropy_stream_len_bits	u(6)
for(i = 0; i < 2 + gsh_num_entropy_streams_minusQ; i++)	
gsh_entropy_stream_len[i]	u(v)
}	
if(geom_scaling_enabled_flag) {	
geom_slice_qp_offset	se(v)
geom_octree_qp_offsets_depth	ue(v)
}	
byte_alignment()	
}	

7.3.3.3 Geometry data unit data syntax

geometry_data_unit_data() {	Descriptor
depthS = depthT = depthV = 0	
for(depth = 0; depth < MaxGeometryOctreeDepth; depth++) {	
for(nodeIdx = 0; nodeIdx < NumNodesAtDepth[depth]; nodeIdx++) {	
sN = NodeS[depthS][nodeIdx]	
tN = NodeT[depthT][nodeIdx]	
vN = NodeV[depthV][nodeIdx]	

geometry_node(depthS, depthT, depthV, partitionSkip, nodeIdx, sN, tN, vN)	
}	
if(! (partitionSkip & 4))	
depthS = depthS + 1	
if(! (partitionSkip & 2))	
depthT = depthT + 1	
if(! (partitionSkip & 1))	
depthV = depthV + 1	
}	
if(log2_trisoup_node_size > 0)	
geometry_trisoup_data()	
}	

7.3.3.4 Geometry node syntax

geometry_node(depthS, depthT, depthV, partitionSkip, nodeIdx, sN, tN, vN) {	Descriptor
if(depth == GeomScalingDepth) {	
geom_node_qp_offset_eq0_flag	ae(v)
if(! geom_node_qp_offset_eq0_flag) {	
geom_node_qp_offset_sign_flag	ae(v)
geom_node_qp_offset_abs_minus1	ae(v)
}	
}	
if(EffectiveDepth < 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(EffectiveDepthS >= RootNodeSizeSLog2 - 1 && EffectiveDepthT >= RootNodeSizeTLog2 - 1 && EffectiveDepthV >= RootNodeSizeVLog2 - 1) {	
if(!unique_geometry_points_flag)	
for(child = 0; child < GeometryNodeChildrenCnt; child++) {	
num_points_eq1_flag[child]	ae(v)
if(!num_points_eq1_flag)	
num_points_minus2[child]	ae(v)
}	
} else {	

if(geometry_planar_mode_flag) {	
for(child = 0; child < GeometryNodeChildrenCnt; child++)	
for(axisIdx = 0; axisIdx <= 2; axisIdx++)	
if(eligible_planar_flag[axisIdx])	
geometry_planar_mode_data(child, axisIdx)	
}	
if(DirectModeFlagPresent)	
geometry_direct_mode_data(0)	
}	
}	

7.3.3.5 Single occupancy data syntax

single_occupancy(nodeIdx) {	Descriptor
if (!is_planar_flag[nodeIdx][0] !is_planar_flag[nodeIdx][1] !is_planar_flag[nodeIdx][2]) {	
if(NeighbourPattern == 0) {	
if(possibly_planar[nodeIdx][0] && possibly_planar[nodeIdx][1] && possibly_planar[nodeIdx][2]) {	
single_occupancy_flag	ae(v)
if(single_occupancy_flag) {	
if(! is_planar_flag[nodeIdx][0])	
occupancy_idx[0]	ae(v)
if(! is_planar_flag[nodeIdx][1])	
occupancy_idx[1]	ae(v)
if(! is_planar_flag[nodeIdx][2])	
occupancy_idx[2]	ae(v)
}	
}	
}	
}	
}	

7.3.3.6 Planar mode data syntax

geometry_planar_mode_data(child, axisIdx) {	Descriptor
is_planar_flag[child][axisIdx]	ae(v)
if(is_planar_flag[child][axisIdx])	
plane_position[child][axisIdx]	ae(v)
}	

7.3.3.7 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	ae(v)
if(!not_duplicated_point_flag) {	
num_direct_points_eq2_flag	ae(v)
if(num_direct_points_eq2_flag)	
num_points_direct_mode_minus3	ae(v)
}	
}	
for(i = 0; i <= num_direct_different_points_minus1; i++){	
if(ChildNodeSizeSLog2 >= 1 && (!is_planar_flag[child][0] partitionSkip & 4))	
point_offset_s[i][0]	ae(v)
for(j = 1; j < EffectiveChildNodeSizeSLog2; j++)	
point_offset_s[i][j]	ae(v)
if(ChildNodeSizeTLog2 >= 1 && (!is_planar_flag[child][1] partitionSkip & 2))	
point_offset_t[i][0]	ae(v)
for(j = 1; j < EffectiveChildNodeSizeTLog2; j++)	
point_offset_t[i][j]	ae(v)
if(ChildNodeSizeVLog2 >= 1 && (!is_planar_flag[child][2] partitionSkip & 1))	
point_offset_v[i][0]	ae(v)
for(j = 1; j < EffectiveChildNodeSizeVLog2; j++)	
point_offset_v[i][j]	ae(v)
}	
}	
}	

7.3.3.8 Geometry trisoup data syntax

geometry_trisoup_data() {	Descriptor
trisoup_sampling_value_minus1	ae(v)
num_unique_segments_minus1	ae(v)
for(i = 0; i <= num_unique_segments_minus1; i++)	
segment_indicator[i]	ae(v)
num_vertices_minus1	ae(v)
for(i = 0; i <= num_vertices_minus1; i++)	
vertex_position[i]	ae(v)
}	

7.3.4 Attribute data unit syntax

7.3.4.1 General attribute data unit syntax

attribute_data_unit () {	Descriptor
attribute_data_unit_header()	
attribute_data_unit_data()	
}	

7.3.4.2 Attribute data unit header syntax

attribute_data_unit_header() {	Descriptor
ash_attr_parameter_set_id	ue(v)
ash_attr_sps_attr_idx	ue(v)
ash_attr_geom_slice_id	ue(v)
if (aps_slice_qp_delta_present_flag) {	
ash_attr_qp_delta_luma	se(v)
if(attribute_dimension_minus1[ash_attr_sps_attr_idx] > 0)	
ash_attr_qp_delta_chroma	se(v)
}	
ash_attr_layer_qp_delta_present_flag	u(1)
if (ash_attr_layer_qp_delta_present_flag) {	
ash_attr_num_layer_qp_minus1	ue(v)
for(i = 0; i < NumLayerQp; i++){	
ash_attr_layer_qp_delta_luma[i]	se(v)
if(attribute_dimension_minus1[ash_attr_sps_attr_idx] > 0)	
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) {	
for(k = 0; k < 3; k++)	
ash_attr_qp_region_origin_xyz[k]	ue(v)
for(k = 0; k < 3; k++)	
ash_attr_qp_region_size_xyz[k]	ue(v)
ash_attr_region_qp_delta	se(v)
}	
byte_alignment()	
}	

7.3.4.3 Attribute data unit data syntax

attribute_data_unit_data() {	Descriptor
AttrDim = attribute_dimension_minus1[ash_attr_sps_attr_idx] + 1	
all_residual_values_equal_to_zero_run	ae(v)
for(i = 0; i < PointCount; i++) {	
if(attr_coding_type == 1 && MaxPredDiff[i] >= lifting_adaptive_prediction_threshold && MaxNumPredictors > 1) {	
pred_index[i]	ae(v)
}	
if(all_residual_values_equal_to_zero_run > 0) {	
for(k = 0; k < AttrDim; k++)	
residual_values[k][i] = 0	
all_residual_values_equal_to_zero_run -= 1	
}	
else {	
attribute_coding(i)	
all_residual_values_equal_to_zero_run	ae(v)
}	
}	
byte_alignment()	
}	

7.3.4.4 Attribute value syntax

attribute_coding(pointIdx) {	Descriptor
for (k = 0; k < AttrDim; k++) {	
residual_values_equal_to_zero[k]	ae(v)
if (residual_values_equal_to_zero[k] == 1)	
residual_values[k][pointIdx] = 0	
else {	
residual_values_equal_to_one[k]	ae(v)
if (residual_values_equal_to_one[k] == 1)	
residual_values[k][pointIdx] = 1	
else {	
residual_values[k][pointIdx]	de(v)
if (residual_values[k][pointIdx] == 255)	
remaining_values[k][pointIdx]	ae(v)
residual_values[k][pointIdx] += 2	
}	
}	
}	

for(d = 1, k = 1; k < AttrDim; k++)	
if(residual_values[k][pointIdx] != residual_values[0][pointIdx])	
d = 0	
for(k = 0; k < AttrDim; k++)	
residual_values[k][pointIdx] += d	
}	

7.4 Semantics

7.4.1 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.

7.4.2 Data unit and byte alignment semantics

7.4.2.1 Sequence parameter set semantics

main_profile_compatibility_flag equal to 1 specifies that the bitstream conforms to the Main profile. **main_profile_compatibility_flag** equal to 0 specifies that the bitstream conforms to a profile other than the Main profile.

reserved_profile_compatibility_22bits shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for **reserved_profile_compatibility_22bits** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **reserved_profile_compatibility_22bits**.

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

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

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

sps_seq_parameter_set_id provides an identifier for the SPS for reference by other syntax elements. In the value of **sps_seq_parameter_set_id** shall be 0 in bitstreams conforming to this version of this Specification. The value other than 0 for **sps_seq_parameter_set_id** is reserved for future use by ISO/IEC.

sps_bounding_box_present_flag equal to 1 indicates that a bounding box is present in the sequence parameter set. **sps_bounding_box_present_flag** equal to 0 indicates that the size of the bounding box is undefined.

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

sps_bounding_box_offset_log2_scale indicates the scaling factor to scale the quantized x, y, and z source bounding box offsets. When not present, the value of **sps_bounding_box_offset_log2_scale** is inferred to be 0.

sps_bounding_box_size_xyz[k] indicates the k-th component of the width, height, and depth, respectively, of the source bounding box in Cartesian co-ordinates.

sps_source_scale_factor_numerator_minus1 plus 1 indicates the scale factor numerator of the source point cloud.

sps_source_scale_factor_denominator_minus1 plus 1 indicates the scale factor denominator of the source point cloud.

sps_num_attribute_sets indicates the number of coded attributes in the bitstream. The value of **sps_num_attribute_sets** shall be in the range of 0 to 63.

attribute_dimension_minus1[i] plus 1 specifies the number of components of the i-th attribute.

attribute_instance_id[i] specifies the instance id for the i-th attribute.

NOTE – The value of the **attribute_instance_id** identifies the attribute when two or more attribute having the **attribute_label_four_bytes** value is in the bitstream. For example, it is useful for the point cloud having multiple color from the different view point.

attribute_bitdepth_minus1[i] plus 1 specifies the bitdepth for first component of the i-th attribute signal(s).

attribute_secondary_bitdepth_minus1[i] plus 1 specifies the bitdepth for secondary component of the i-th attribute signal(s).

attribute_cicp_colour_primaries[i] indicates the chromaticity co-ordinates of the colour attribute source primaries of the i-th attribute. The semantics are as specified for the code point **ColourPrimaries** in ISO/IEC 23091-2.

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 L_c 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 L_o with a nominal real-valued range of 0 to 1. The semantics are as specified for the code point **TransferCharacteristics** in ISO/IEC 23091-2.

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. The semantics are as specified for the code point **MatrixCoefficients** in ISO/IEC 23091-2.

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. The semantics are as specified for the code point **VideoFullRangeFlag** in ISO/IEC 23091-2.

known_attribute_label_flag[i] equal to 1 specifies **known_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 8 describes the list of supported attributes and their relationship with attribute_label_four_bytes[i].

Table 8 — attribute_label_four_bytes

attribute_label_four_bytes[i]	Attribute type
0	Colour
1	Reflectance
2	Frame index
3	Material ID
4	Transparency
5	Normals
6 .. 255	Reserved
256 .. 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.

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

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

Table 9 — Definition of XyzToStv[k] according to the value of axis_coding_order

axis_coding_order	XyzToStv[k]		
	0	1	2
0	2	1	0
1	0	1	2
2	0	2	1
3	2	0	1
4	2	1	0
5	1	2	0
6	1	0	2
7	0	1	2

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

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

sps_bypass_stream_enabled_flag equal to 1 specifies that the bypass coding mode may be used on reading the bitstream. **sps_bypass_stream_enabled_flag** equal to 0 specifies that the bypass coding mode is not used on reading the bitstream.

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

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

7.4.2.2 Tile inventory semantics

tile_frame_idx contains an identifying number that may be used to identify the purpose of the tile inventory.

num_tiles specifies the number of tile bounding boxes present in the tile inventory.

tile_bounding_box_bits specifies the bitdepth to represent the bounding box information for the tile inventory.

tile_bounding_box_offset_xyz[tileId][k] and **tile_bounding_box_size_xyz**[tileId][k] specify a bounding box encompassing slices identified by **gsh_tile_id** equal to tileId.

tile_bounding_box_offset_xyz[tileId][k] is the k-th component of the (x, y, z) origin co-ordinate of the tile bounding box relative to **sps_bounding_box_offset**[k].

tile_bounding_box_size_xyz[tileId][k] is the k-th component of the tile bounding box width, height, and depth, respectively.

7.4.2.3 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_flag** equal 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_gs_box_log2_scale indicates a scale factor to be applied to the slice origin of all slices that reference the current GPS.

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

geometry_planar_mode_flag equal to 1 indicates that the planar coding mode is activated. **geometry_planar_mode_flag** equal to 0 indicates that the planar coding mode is not activated.

geom_planar_mode_th_idcm specifies the value of the threshold of activation for the direct coding mode. **geom_planar_mode_th_idcm** is an integer in the range 0 to 127 inclusive. When not present, **geom_planar_mode_th_idcm** is inferred to be 127.

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

geometry_angular_mode_flag equal to 1 indicates that the angular coding mode is activated. **geometry_angular_mode_flag** equal to 0 indicates that the angular coding mode is not activated.

geom_angular_origin_xyz[k] specifies the k -th component of the (x, y, z) co-ordinate of the origin used in the processing of the angular coding mode. When not present, **geom_angular_origin_x**, **geom_angular_origin_y**, and **geom_angular_origin_z** are inferred to be 0.

The array **geomAngularOrigin**, with values **geomAngularOrigin[k]**, for $k = 0 \dots 2$, represents the values of **geom_angular_origin_xyz** permuted into the coded geometry axis order as follows:

```
geomAngularOrigin[XyzToStv[k]] = geom_angular_origin_xyz[k], for  $k = 0 \dots 2$ 
```

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

laser_angle[i], for i in the range 1 .. **number_lasers**, specifies the tangent of the elevation angle of the i -th laser relative to the horizontal plane defined by the first and second coded axes.

laser_correction[i], for i in the range 1 .. **number_lasers**, specifies the correction, along the second internal axis, of the i -th laser position relative to the **geomAngularOrigin[2]**. When not present, **laser_correction[i]** is inferred to be 0.

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

implicit_qtbt_angular_max_node_min_dim_log2_to_split_v specifies the log2 value of a node size below which horizontal split of nodes is preferred over vertical split. When not present, **implicit_qtbt_angular_max_diff_to_split_v** specifies is inferred to be 0.

implicit_qtbt_angular_max_diff_to_split_v specifies the log2 value of the maximum vertical over horizontal node size ratio allowed to a node. When not present, **implicit_qtbt_angular_max_node_min_dim_log2_to_split_v** is inferred to be 0.

neighbour_context_restriction_flag equal to 0 indicates that geometry node occupancy of the current node is coded with the contexts determined from neighbouring nodes which is located inside the parent node of the current node. **neighbour_context_restriction_flag** equal to 1 indicates that geometry node occupancy of the current node is coded with the contexts determined from neighbouring nodes which is located inside or outside the parent node of the current node.

inferred_direct_coding_mode_enabled_flag equal to 1 indicates that **direct_mode_flag** may be present in the geometry node syntax. **inferred_direct_coding_mode_enabled_flag** equal to 0 indicates that **direct_mode_flag** is not present in the geometry node syntax.

bitwise_occupancy_coding_flag equal to 1 indicates that geometry node occupancy is encoded using bit-wise contextualisation of the syntax element `occupancy_map`. **bitwise_occupancy_coding_flag** equal to 0 indicates that geometry node occupancy is encoded using the dictionary encoded syntax element `occupancy_byte`.

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

log2_neighbour_avail_boundary specifies the variable `NeighbAvailabilityMask` as follows.

When `neighbour_context_restriction_flag` is equal to 1, `NeighbAvailabilityMask` is set equal to 1. Otherwise, `neighbour_context_restriction_flag` equal to 0, `NeighbAvailabilityMask` is set equal to $1 \ll \text{log2_neighbour_avail_boundary}$.

log2_intra_pred_max_node_size specifies the octree node size eligible for occupancy intra prediction.

log2_trisoup_node_size specifies the variable `TrisoupNodeSize` as the size of the triangle nodes as follows.

```
TrisoupNodeSize = 1 << log2_trisoup_node_size
```

When `log2_trisoup_node_size` is equal to 0, the geometry bitstream includes only the octree coding syntax. When `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_minus4 plus 4 specifies the base value of the geometry position quantization parameter. When not present, `geom_base_qp_minus4` is inferred to be 0.

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_ot` and `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.

gps_min_size_implicit_qtbt specifies the minimal size of implicit QT and BT partitions.

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

gps_extension_data_flag may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all `gps_extension_data_flag` syntax elements.

7.4.2.4 Attribute parameter set semantics

aps_attr_parameter_set_id provides an identifier for the APS for reference by other syntax elements. The value of **aps_attr_parameter_set_id** shall be in the range of 0 to 15, inclusive.

aps_seq_parameter_set_id specifies the value of **sps_seq_parameter_set_id** for the active SPS. The value of **aps_seq_parameter_set_id** shall be in the range of 0 to 15, inclusive.

attr_coding_type indicates that the coding type for the attribute in Table 11 for the given value of **attr_coding_type**. The value of **attr_coding_type** shall be equal to 0, 1, or 2 in bitstreams conforming to this version of this Specification. Other values of **attr_coding_type** are reserved for future use by ISO/IEC. Decoders conforming to this version of this Specification shall ignore reserved values of **attr_coding_type**.

Table 11 — Interpretation of attr_coding_type

attr_coding_type	coding type	decoding process
0	Region Adaptive Hierarchical Transform (RAHT)	8.3.1
1	LoD with Predicting Transform	8.3.3
2	LoD with Lifting Transform	8.3.2

aps_attr_initial_qp specifies the initial value of the variable **SliceQp** for each slice referring to the APS. The value of **aps_attr_initial_qp** shall be in the range of 4 to 51, 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_chroma** syntax elements are present in the ASH. **aps_slice_qp_delta_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.

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 disabled in the RAHT decoding process.

raht_prediction_threshold0 specifies the threshold to terminate the transform weight prediction from neighbour points. The value of **raht_prediction_threshold0** shall be in the range of 0 to 19.

raht_prediction_threshold1 specifies the threshold to skip the transform weight prediction from neighbour points. The value of **raht_prediction_threshold1** shall be in the range of 0 to 19.

lifting_num_pred_nearest_neighbours_minus1 plus 1 specifies the maximum number of nearest neighbours to be used for prediction. The value of **lifting_num_pred_nearest_neighbours** shall be in the range of 1 to xx.

The value of **NumPredNearestNeighbours** is set equal to **lifting_num_pred_nearest_neighbours**

lifting_search_range_minus1 plus 1 specifies the search range used to determine nearest neighbours to be used for prediction and to build distance-based levels of detail.

The variable **LiftingSearchRange** is derived as follows:

LiftingSearchRange = **lifting_search_range_minus1** + 1

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

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

```
liftingNeighbourBiasStv[XyzToStv[k]] = lifting_neighbour_bias_xyz[k]
```

lifting_scalability_enabled_flag equal to 1 specifies that the attribute decoding process allows the pruned octree decode result for the input geometry points. lifting_scalability_enabled_flag equal to 0 specifies that the attribute decoding process requires the complete octree decode result for the input geometry points. When not present, the value of lifting_scalability_enabled_flag is inferred to be 0. When the value of log2_trisoup_node_size is greater than 0, the value of lifting_scalability_enabled_flag shall be 0.

lifting_num_detail_levels_minus1 specifies the number of levels of detail for the attribute coding. The value of lifting_num_detail_levels_minus1 shall be in the range of 0 to xx.

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

```
LevelDetailCount = lifting_num_detail_levels_minus1 + 1
```

lifting_lod_regular_sampling_enabled_flag equal to 1 specifies levels of detail are built by using a regular sampling strategy. lifting_lod_regular_sampling_enabled_flag equal to 0 specifies that a distance-based sampling strategy is used instead.

lifting_sampling_period_minus2[idx] plus 2 specifies the sampling period for the level of detail idx. The value of lifting_sampling_period_minus2[] shall be in the range of 0 to xx.

lifting_sampling_distance_squared_scale_minus1[idx] plus 1 specifies the scaling factor for the derivation of the square of the sampling distance for the level of detail idx. The value of lifting_sampling_distance_squared_scale_minus1[idx] shall be in the range of 0 to xx. When lifting_sampling_distance_squared_scale_minus1[idx] is not present in the bitstream, it is inferred to be 0.

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

The variable LiftingSamplingDistanceSquared[idx] for idx = 0 .. num_detail_level_minus1 – 1, specifying the sampling distance for the level of detail idx, are derived as follows:

```
LiftingSamplingDistanceSquared[0] = lifting_sampling_distance_squared_scale_minus1[0] + 1
for (idx = 1; idx < num_detail_level_minus1; idx++) {
    LiftingSamplingDistanceSquared[idx] =
        (lifting_sampling_distance_squared_scale_minus1[idx] + 1)
        × LiftingSamplingDistanceSquared[idx - 1]
        + lifting_sampling_distance_squared_offset[idx]
}
```

lifting_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.

The variable AdaptivePredictionThreshold specifying the threshold to switch to adaptive predictor selection mode is set equal to lifting_adaptive_prediction_threshold

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

The variable **IntraLodPredNumLayers** specifying the number of LoD layer where intra lod prediction is enabled is set equal to **lifting_intra_lod_prediction_num_layers**.

lifting_max_num_direct_predictors specifies the maximum number of predictors predictor 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 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.

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

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

7.4.2.5 Frame boundary marker semantics

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

7.4.2.6 Byte alignment semantics

alignment_bit_equal_to_one shall be equal to 1.

alignment_bit_equal_to_zero shall be equal to 0.

7.4.3 Geometry data unit semantics

7.4.3.1 General geometry data unit semantics

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

The variables **NodeS[depthS][idx]**, **NodeT[depthT][idx]**, and **NodeV[depthV][idx]** represent the s, t, and v co-ordinates of the idx-th node in decoding order at the given depth. The variable **NumNodesAtDepth[depth]** represents the number of nodes to be decoded at the given depth. The variables **depthS**, **depthT** and **depthV** specify respectively the depth in s, t and v dimensions. The variable **partitionSkip** specifies the partition type and direction as in Table 12. The variable **partitionSkip** is

represented in binary form with three bits $b_s b_t b_v$, which specify respectively whether to skip partition along s, t and v dimension.

Table 12— Interpretation of partitionSkip

Partition	QT along s-t axes	QT along s-v axes	QT along t-v axes	OT
partitionSkip	0b001	0b010	0b100	0b000
Partition	BT along s axis	BT along t axis	BT along v axis	
partitionSkip	0b011	0b101	0b110	

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

```
NodeS[0][0] = NodeT[0][0] = NodeV[0][0] = 0
NumNodesAtDepth[0] = 1
GeometryNodeOccupancyCnt[-1][0][0][0] = 8
```

7.4.3.2 Geometry data unit header semantics

gsh_num_points_minus1 plus 1 specifies the maximum number of coded points in the slice. It is a requirement of bitstream conformance that $\text{gsh_num_points_minus1} + 1$ is greater than or equal to the number of decoded points in the slice.

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 $\log_2 \text{max_frame_idx} + 1$ 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 the slice bounding box origin. When not present, `gsh_box_log2_scale` is inferred to be equal to `gps_gs_box_log2_scale`.

gsh_box_origin_xyz[k] specifies the k-th component of the quantized (x, y, z) co-ordinate of the slice bounding box origin. When not present, the values of `gsh_box_origin_xyz[k]` are each inferred to be 0.

The array `SliceOriginStv`, with values `SliceOriginStv[k]`, $k = 0 \dots 2$, represents the scaled values of `gsh_box_origin_xyz` permuted into the coded geometry axis order as follows:

```
SliceOriginStv[XyzToStv[k]] = gsh_box_origin_xyz[k] << gsh_box_log2_scale
```

gsh_log2_root_node_size_s specifies, when present, the first component, s, of the geometry tree root node size.

gsh_log2_root_node_size_t_minus_s specifies, when present, the second component, t, of the geometry tree root node size.

gsh_log2_root_node_size_v_minus_t specifies, when present, the third component, v, of the geometry tree root node size.

gsh_log2_root_node_size specifies, when present, the size of the root geometry tree node.

The variables **RootNodeSizeSLog2**, **RootNodeSizeTLog2**, and **RootNodeSizeVLog2** are defined as follows:

- When **gps_implicit_geom_partition_flag** is equal to 1, the following applies:

```
RootNodeSizeSLog2 = gsh_log2_root_node_size_s
RootNodeSizeTLog2 = gsh_log2_root_node_size_t_minus_s + RootNodeSizeSLog2
RootNodeSizeVLog2 = gsh_log2_root_node_size_v_minus_t + RootNodeSizeTLog2
```

- Otherwise, **gps_implicit_geom_partition_flag** equal to 0, the following applies:

```
RootNodeSizeSLog2 = 1 << gsh_log2_root_node_size
RootNodeSizeTLog2 = 1 << gsh_log2_root_node_size
RootNodeSizeVLog2 = 1 << gsh_log2_root_node_size
```

The variables **RootNodeSizeS**, **RootNodeSizeT**, **RootNodeSizeV**, and **MaxGeometryOctreeDepth** are initialized as follows:

```
RootNodeSizeS = 1 << RootNodeSizeSLog2
RootNodeSizeT = 1 << RootNodeSizeTLog2
RootNodeSizeV = 1 << RootNodeSizeVLog2
MaxRootNodeDimLog2 = Max(RootNodeSizeSLog2, RootNodeSizeTLog2, RootNodeSizeVLog2)
minRootNodeDimLog2 = Min(RootNodeSizeSLog2, RootNodeSizeTLog2, RootNodeSizeVLog2)
MaxGeometryOctreeDepth = MaxRootNodeDimLog2 - log2_trisoup_node_size
```

The variables **QtBtK** and **QtBtM** are derived as follows:

- When **log2_trisoup_node_size** is equal to 0, the following applies

```
QtBtK = Min(gps_max_num_implicit_qtbt_before_ot, MaxRootNodeDimLog2 - minRootNodeDimLog2)
if (MaxRootNodeDimLog2 == minRootNodeDimLog2)
    QtBtM = 0;
else
    QtBtM = Min(gps_min_size_implicit_qtbt, minRootNodeDimLog2)
```

- Otherwise, **log2_trisoup_node_size** is greater than 0, the following applies:

```
QtBtK = MaxRootNodeDimLog2 - minRootNodeDimLog2
QtBtM = 0
```

gsh_num_entropy_streams_minusQ indicates the number of entropy streams used to convey the geometry slice data. If **gsh_num_entropy_streams_minusQ** is equal to 0, the geometry slice data is conveyed in a single entropy stream. Otherwise (**gsh_num_entropy_streams_minusQ** is greater than 0), the geometry slice data is conveyed in **gsh_num_entropy_streams_minusQ** + 2 streams. It is a requirement of bitstream conformance that **gsh_num_entropy_streams_minusQ** is equal to 0 when **log2_trisoup_node_size** is greater than 0.

The variable **EntropyStreamCnt** represents the number of entropy streams present in the current data unit:

```
if (!gsh_num_entropy_streams_minusQ)
    EntropyStreamCnt = 1
else
    EntropyStreamCnt = gsh_num_entropy_streams_minusQ + 2
```

The variable **GeomEntropyStreamDepth** is derived as follows:

```
GeomEntropyStreamDepth = MaxGeometryOctreeDepth - gsh_num_entropy_streams_minusQ - 2
```

gsh_entropy_stream_len_bits specifies the number of bits used to represent each of the `gsh_entropy_stream_len[i]` syntax elements.

gsh_entropy_stream_len[i] specifies the length in bytes of the *i*-th entropy stream.

geom_slice_qp_offset specifies an offset to the base geometry quantisation parameter `geom_base_qp_minus4`. When not present, `geom_slice_qp_offset` is inferred to be 0.

geom_octree_qp_offsets_depth specifies, when present, the depth of the geometry octree when `geom_node_qp_offset_eq0_flag` is present in the geometry node syntax.

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

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

```
GeomScalingDepth = geom_scaling_enabled_flag ? geom_octree_qp_offsets_depth : 0
```

7.4.3.3 Geometry slice data semantics

The process to derive the variable `partitionSkip` is specified from here. The input of the process are variables `depth`, `depthS`, `depthT` and `depthV`. The output of the process is the value of `partitionSkip`. The process to derive the variable `partitionSkip` proceeds as follows.

```
partitionSkip = 0
NodeSizeSLog2 = RootNodeSizeSLog2 - depthS
NodeSizeTLog2 = RootNodeSizeTLog2 - depthT
NodeSizeVLog2 = RootNodeSizeVLog2 - depthV
MinNodeDimLog2 = Min(NodeSizeSLog2, NodeSizeTLog2, NodeSizeVLog2)
MaxNodeDimLog2 = Max(NodeSizeSLog2, NodeSizeTLog2, NodeSizeVLog2)
If (MinNodeDimLog2 == MaxNodeDimLog2)
    QtBtM = 0

if (QtBtK > depth || M == MinNodeDimLog2) {
    if (NodeSizeSLog2 < MaxNodeDimLog2)
        partitionSkip |= 4
    if (NodeSizeTLog2 < MaxNodeDimLog2)
        partitionSkip |= 2
    if (NodeSizeVLog2 < MaxNodeDimLog2)
        partitionSkip |= 1
}
else if (geometry_angular_mode_flag) {
    minDim = implicit_qtbt_angular_max_node_min_dim_log2_to_split_v
    maxDiff = implicit_qtbt_angular_max_diff_to_split_v
    if (minDim + maxDiff > 0){
        maxNodeDimLog2ST = Max(NodeSizeSLog2, NodeSizeTLog2)
        if (NodeSizeSLog2 < maxNodeDimLog2ST)
            partitionSkip |= 4
        if (NodeSizeTLog2 < maxNodeDimLog2ST)
            partitionSkip |= 2
        if (MinNodeDimLog2 <= minDim && NodeSizeVLog2 >= maxNodeDimLog2ST + maxDiff)
            partitionSkip |= 1
        if (maxNodeDimLog2ST > minDim + maxDiff && NodeSizeVLog2 >= maxNodeDimLog2ST)
            partitionSkip |= 1
    }
}
```

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

7.4.3.4 Geometry node semantics

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

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

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

```
sNp = sN >> 1
tNp = tN >> 1
vNp = vN >> 1
```

The variables NodeSizeLog2 and ChildNodeSizeLog2 are derived as follows:

```
NodeSizeLog2 = MaxRootNodeDimLog2 - depth
ChildNodeSizeLog2 = NodeSizeLog2 - 1
```

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

```
ScalingNodeSizeLog2[0] = NodeSizeSLog2
ScalingNodeSizeLog2[1] = NodeSizeTLog2
ScalingNodeSizeLog2[2] = NodeSizeVLog2
minScalingNodeDimLog2 = Min(NodeSizeSLog2, NodeSizeTLog2, NodeSizeVLog2)
```

The variable NeighbourPattern is derived as follows:

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

```
rN = GeometryNodeOccupancyCnt[depth][sN + 1][tN][vN] != 0
lN = GeometryNodeOccupancyCnt[depth][sN - 1][tN][vN] != 0
bN = GeometryNodeOccupancyCnt[depth][sN][tN + 1][vN] != 0
fN = GeometryNodeOccupancyCnt[depth][sN][tN - 1][vN] != 0
uN = GeometryNodeOccupancyCnt[depth][sN][tN][vN + 1] != 0
dN = GeometryNodeOccupancyCnt[depth][sN][tN][vN - 1] != 0
```

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

```
lN = ((sN + 1) & NeighbAvailabilityMask == 1 ? 0 : lN
rN = ((sN + 1) & NeighbAvailabilityMask == 0 ? 0 : rN
fN = ((tN + 1) & NeighbAvailabilityMask == 1 ? 0 : fN
bN = ((tN + 1) & NeighbAvailabilityMask == 0 ? 0 : bN
dN = ((vN + 1) & NeighbAvailabilityMask == 1 ? 0 : dN
uN = ((vN + 1) & NeighbAvailabilityMask == 0 ? 0 : uN
```

- If adjacent_child_contextualization_enabled_flag is equal to 1, the following applies.

```
lNadj = fNadj = dNadj = 0
for (sNc = sN × 2; sNc < sN × 2 + 2; sNc++){
  for (tNc = tN × 2; tNc < tN × 2 + 2; tNc++){
    for (vNc = vN × 2; vNc < vN × 2 + 2; vNc++) {
      lNadj |= GeometryNodeOccupancyCnt[depth + 1][sN × 2 - 1][tNc][vNc]
      fNadj |= GeometryNodeOccupancyCnt[depth + 1][sNc][tN × 2 - 1][vNc]
      dNadj |= GeometryNodeOccupancyCnt[depth + 1][sNc][tNc][vN × 2 - 1]
    }
  }
}
lN &= lNadj
fN &= fNadj
dN &= dNadj
```

- Finally, the variable NeighbourPattern is set as follows:

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

geom_node_qp_offset_eq0_flag equal to 1 specifies that the current node's quantization parameter is offset from the slice quantization parameter. **geom_node_qp_offset_eq0_flag** equal to 0 specifies that the current node quantization parameter inherits the quantization parameter of the parent node.

geom_node_qp_offset_sign_flag specifies, when present, the sign of nodeQpOffset as follows:

- If **geom_node_qp_offset_sign_flag** is equal to 0, the corresponding nodeQpOffset has a negative value.
- Otherwise, **geom_node_qp_offset_sign_flag** is equal to 1, the corresponding nodeQpOffset has a positive value.

geom_node_qp_offset_abs_minus1 plus 1 specifies, when present, the absolute difference between the current node's quantization parameter, nodeQp, and the slice quantisation parameter.

The variable nodeQpOffset is derived as follows:

```
if (geom_node_qp_offset_eq0_flag)
    nodeQpOffset = 0
else
    nodeQpOffset = (2 × geom_node_qp_offset_sign_flag - 1) × (geom_node_qp_offset_abs_minus1 + 1)
```

The variable NodeQp is derived as follows:

- When depth is equal to GeomScalingDepth:

```
NodeQp = geom_base_qp_minus4 + 4 + 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.

It is a requirement of bitstream conformance that NodeQp is less than or equal to $\text{minScalingNodeDimLog2} \times 6 + 9$.

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

```
EffectiveChildNodeSizeLog2 = ChildNodeSizeLog2 - (NodeQp - 4) / 6
EffectiveDepth = depth + (NodeQp - 4) / 6
EffectiveDepthS = depthS + (NodeQp - 4) / 6
EffectiveDepthT = depthT + (NodeQp - 4) / 6
EffectiveDepthV = depthV + (NodeQp - 4) / 6
```

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:

```
childCnt = 0
for (childIdx = 0; childIdx < 8; childIdx++) {
    if (!(OccupancyMap & (1 << childIdx)))
        continue
    GeometryNodeChildren[childCnt++] = childIdx
}
GeometryNodeChildrenCnt = childCnt
GeometryNodeOccupancyCnt[depth][sN][tN][vN] = childCnt
```

The variable `DirectModeFlagPresent` is derived as follows:

- When all of the following conditions are true, `DirectModeFlagPresent` is set equal to 1:
 - `inferred_direct_coding_mode_enabled_flag` is equal to 1
 - `proba_planar[0] * proba_planar[1] * proba_planar[2]` is less than or equal to `127 * 127 * geom_planar_mode_th_IDCM`
 - `NodeSizeLog2` is greater than 1
 - `GeometryNodeOccupancyCnt[depth - 1][sNp][tNp][vNp]` is less than or equal to 2
 - `GeometryNodeOccupancyCnt[depth][sN][tN][vN]` is equal to 1
 - `NeighbourPattern` is equal to 0
 - (`geometry_angular_mode_flag` is equal to 0) OR (`geometry_angular_mode_flag` is equal to 1 AND `idcm4angular[child]` is equal to 1)
- Otherwise, `DirectModeFlagPresent` is set equal to 0.

The determination of the probabilities `proba_planar[]` is performed as described in 8.2.4.6.

num_points_eq1_flag[*child*] equal to 1 indicates that the current child node contains a single point. `num_points_eq1_flag` equal to 0 indicates that the current child node contains at least two points. When not present, the value of `num_points_eq1_flag` is inferred equal to 1.

num_points_minus2[*child*] plus 2 indicates the number of points represented by the current child node.

The array `GeometryNodeDupPoints[child]` identifies the number of duplicate points in each child of the current leaf node. When `num_points_eq1_flag` is equal to 0, `GeometryNodeDupPoints[child]` is set equal to `1 + num_points_minus2[child]`. Otherwise, `GeometryNodeDupPoints[child]` is set equal to 0.

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

7.4.3.5 Single occupancy data semantics

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

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

7.4.3.6 Planar mode data semantics

is_planar_flag[child][axisIdx] equal to 1 indicates that the current child node is planar in the direction perpendicular to the axisIdx-th axis. **is_planar_flag[child][axisIdx]** equal 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 co-ordinates. **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 co-ordinates.

7.4.3.7 Direct mode data semantics

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

When **direct_mode_flag** is equal to 0, the following applies:

```
nodeIdx = NumNodesAtDepth[depth + 1]
for (child = 0; child < GeometryNodeChildrenCnt; child++) {
    childIdx = GeometryNodeChildren[child]
    s = NodeS[depth + 1][nodeIdx] = 2 × sN + (childIdx & 4 ==== 1)
    t = NodeT[depth + 1][nodeIdx] = 2 × tN + (childIdx & 2 ==== 1)
    v = NodeV[depth + 1][nodeIdx] = 2 × vN + (childIdx & 1 ==== 1)
    NodeQpMap[depth + 1][nodeIdx] = NodeQp
    GeometryNodeOccupancyCnt[depth + 1][s][t][v] = 1
    nodeIdx++
}
NumNodesAtDepth[depth + 1] = nodeIdx
```

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 s, t and v co-ordinates. **num_direct_points_gt1** equal to 1 indicates that there are at least two points in the current child node with different s, t or v co-ordinates.

not_duplicated_point_flag equal to 0 indicates that all points in the current child node have the same s, t and v co-ordinates. **not_duplicated_point_flag** equal to 1 indicates that at least two points in the current child node have different s, t or v co-ordinates. When not present, the value of **not_duplicated_point_flag** is inferred equal to 1.

The variable **duplicatedPointFlag** is derived as the negation of **not_duplicated_point_flag** as follows

```
duplicatedPointFlag = !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 there 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 (duplicatedPointFlag && !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 s, t or v co-ordinate 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 (duplicatedPointFlag)
    num_direct_different_points_minus1 = 0
```

The variables **ChildNodeSizeSLog2**, **ChildNodeSizeTLog2** and **ChildNodeSizeVLog2** specify the s, t, and v components of the child node size, and are determined by implicit QT and BT partitions as follows.

```
if (!(partitionSkip & 4)
    ChildNodeSizeSLog2 = NodeSizeSLog2 - 1;
else
    ChildNodeSizeSLog2 = NodeSizeSLog2;
if (!(partitionSkip & 2)
    ChildNodeSizeTLog2 = NodeSizeTLog2 - 1;
else
    ChildNodeSizeTLog2 = NodeSizeTLog2;
if (!(partitionSkip & 1)
    ChildNodeSizeVLog2 = NodeSizeVLog2 - 1;
else
    ChildNodeSizeVLog2 = NodeSizeVLog2;
```

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

When **point_offset_s[i][0]** is not present, the value of **point_offset_s[i][0]** is inferred by the plane position **plane_position[child][0]**.

When **point_offset_t[i][0]** is not present, the value of **point_offset_t[i][0]** is inferred by the plane position **plane_position[child][1]**.

When **point_offset_v[i][0]** is not present, the value of **point_offset_v[i][0]** is inferred by the plane position **plane_position[child][2]**.

The variables **PointOffsetS[i]**, **PointOffsetT[i]**, and **PointOffsetV[i]** are derived as follows:

```
PointOffsetS[i] = PointOffsetT[i] = PointOffsetV[i] = 0;
for (j = 0; j < EffectiveChildNodeSizeSLog2; j++)
    PointOffsetS[i] += point_offset_s[i][j] << j;

for (j = 0; j < EffectiveChildNodeSizeTLog2; j++)
```



```
PointOffsetT[i] += point_offset_t[i][j] << j;
for (j = 0; j < EffectiveChildNodeSizeVLog2; j++)
    PointOffsetV[i] += point_offset_v[i][j] << j;
```

7.4.3.8 Geometry trisoup data semantics

trisoup_sampling_value_minus1 plus 1 specifies the step size for the point sampling on the triangle surface in the trisoup decoding process specified in 8.2.3.3

num_unique_segments specifies the number of segment indicators.

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_minus1 plus 1 specifies the number of vertices.

vertex_position[i] indicates the position of the vertex along the edge. The value of **vertex_position**[i] shall be in the range of 0 to (1 << log2_trisoup_node_size) – 1, inclusive.

7.4.4 Attribute data unit semantics

7.4.4.1 General attribute data unit semantics

7.4.4.2 Attribute data unit header semantics

ash_attr_parameter_set_id specifies the value of the **aps_attr_parameter_set_id** of the active APS.

ash_attr_sps_attr_idx specifies the order of attribute set in the active SPS. The value of **ash_attr_sps_attr_idx** shall be in the range of 0 to **sps_num_attribute_sets** in the active SPS.

ash_attr_geom_slice_id specifies the value of the **gsh_slice_id** of the active Geometry Slice Header.

ash_attr_layer_qp_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_num_layer_qp_minus1 plus 1 specifies the number of layer in which **ash_attr_qp_delta_luma** and **ash_attr_qp_delta_chroma** are signalled. When **ash_attr_num_layer_qp** is not signalled, the value of **ash_attr_num_layer_qp** is inferred to be 0. The value of **NumLayerQp** is derived as follows:

$$\text{NumLayerQp} = \text{num_layer_qp_minus1} + 1$$

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_attr_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 .. NumLayerQPNumQPLayer – 1 are derived as follows:

```
for (i = 0; i < NumLayerQPNumQPLayer; i++) {
    SliceQpY[i] = InitialSliceQpY + ash_attr_layer_qp_delta_luma[i]
    SliceQpC[i] = InitialSliceQpC + ash_attr_layer_qp_delta_chroma[i]
}
```

ash_attr_region_qp_delta_present_flag equal to 1 indicates that a QP offset is applied to a spatial region within the current slice. ash_attr_region_qp_delta_present_flag equal to 0 indicates that no spatial adaptation of QP is performed for the current slice.

ash_attr_qp_region_origin_xyz[*k*] and **ash_attr_qp_region_size_xyz[*k*]** specify, when present, the spatial region within the current slice where ash_attr_region_qp_delta is applied.

ash_attr_qp_region_origin_xyz[*k*] is the *k*-th component of the (*x*, *y*, *z*) origin co-ordinate relative to the slice origin.

ash_attr_qp_region_size_xyz[*k*] is the *k*-th component of the region width, height, and depth, respectively.

The arrays AttrRegionQpOriginStv and AttrRegionQpSizeStv, with values AttrRegionQpOriginStv[*k*] and AttrRegionQpSizeStv[*k*], for *k* = 0 .. 2, represents the values of ash_attr_qp_region_origin_xyz and ash_attr_qp_region_size_xyz respectively permuted into the coded geometry axis order as follows:

```
AttrRegionQpOriginStv[XyzToStv[k]] = ash_attr_qp_region_origin_xyz[k]
AttrRegionQpSizeStv[XyzToStv[k]] = ash_attr_qp_region_size_xyz[k]
```

ash_attr_region_qp_delta specifies the QP offset to be applied within the region defined by ash_attr_qp_region_origin_xyz and ash_attr_qp_region_size_xyz.

7.4.4.3 Attribute slice data semantics

all_residual_values_equal_to_zero_run specifies the number of occurrence of the pattern which indicates that each residual values of all dimension are equal to zero.

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

The variable MaxPredDiff[*i*] is calculated as follows:

Let \mathfrak{N}_i be the set of the *k*-nearest neighbours of the current point *i* and let $(\tilde{a}_j)_{j \in \mathfrak{N}_i}$ be their decoded/reconstructed attribute values. The number of nearest neighbours, *k*, shall be range of 1 to lifting_num_pred_nearest_neighbours. The decoded/reconstructed attribute value of neighbours are derived according to the Predictive Lifting decoding process (8.3.3).

```
minValue = maxValue =  $\tilde{a}_0$ 
for (j = 0; j < k; j++) {
    minValue = Min(minValue,  $\tilde{a}_j$ )
    maxValue = Max(maxValue,  $\tilde{a}_j$ )
}
```

```
MaxPredDiff[i] = maxValue - minValue;
```

7.4.4.4 Quantized value bitstream syntax

residual_values_equal_to_zero[k] equal to 1 indicates that **residual_values**[k][i] is equal to 0. **residual_values_equal_to_zero** equal to 0 indicates that **residual_values**[k][i] is not equal to 0.

residual_values_equal_to_one[k] equal to 1 indicates that **residual_values**[k][i] equal to 1. **residual_values_equal_to_one** equal to 0 indicates that **residual_values**[k][i] is larger than 2.

residual_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.

8 Decoding process

8.1 General decoding process

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

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

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

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

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

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

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

8.2 Geometry decoding process

8.2.1 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_minus1** inclusive, and axis ranging from 0 to 2 inclusive.

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

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

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

8.2.2 Octree decoding process

8.2.2.1 General

8.2.2.2 Octree node decoding process

The inputs to this process are:

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

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

If both EffectiveDepth is less than 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];
    s = 2 × sN + (childIdx & 4) ==== 1;
    t = 2 × tN + (childIdx & 2) ==== 1;
    v = 2 × vN + (childIdx & 1) ==== 1;
    for (i = 0; i < GeometryNodeDupPoints[child] + 1 ; i++, PointCount++) {
        PointPos[PointCount][0] = geomScale(s, 0);
        PointPos[PointCount][1] = geomScale(t, 1);
        PointPos[PointCount][2] = geomScale(v, 2);
    }
    if (direct_mode_flag) {
        if (!duplicatedPointFlag) {
            for (i = 0; i <= num_direct_points_minus1; i++, PointCount++) {
                PointPos[PointCount][0] = geomScale((s << EffectiveChildNodeSizeLog2) +
                PointOffsetS[i], 0);
                PointPos[PointCount][1] = geomScale((t << EffectiveChildNodeSizeLog2) +
                PointOffsetT[i], 1);
                PointPos[PointCount][2] = geomScale((v << EffectiveChildNodeSizeLog2) +
                PointOffsetV[i], 2);
            }
        }
        else {
            for (i = 0; i <= num_direct_points_minus1; i++, PointCount++) {
                PointPos[PointCount][0] = geomScale((s << EffectiveChildNodeSizeLog2) +
                PointOffsetS[0], 0);
                PointPos[PointCount][1] = geomScale((t << EffectiveChildNodeSizeLog2) +
                PointOffsetT[0], 1);
                PointPos[PointCount][2] = geomScale((v << EffectiveChildNodeSizeLog2) +
                PointOffsetV[0], 2);
            }
        }
    }
}
```

}

It is a requirement of bitstream conformance that PointCount is less than or equal to gsh_num_points.

8.2.2.3 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 is 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 scale factor sF is derived as follows:

```
sF = geomLevelScale[qP % 6] << (qP / 6)
```

The output variable pos is derived as follows:

```
highPartS = highPart << ScalingNodeSizeLog2[cIdx]
lowPartS = (lowPart × sF + (1 << 19)) >> 20
pos = highPartS | Min(lowPartS, (1 << ScalingNodeSizeLog2[cIdx]) - 1)
```

8.2.3 Geometry Trisoup decoding process

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

This process modifies the following:

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

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

8.2.3.1 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 $\text{PointCount} \times 12 - 1$.

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

8.2.3.1.1 Derivation process for sorted segment index

Outputs of this process are:

the array `segStPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$

the array `segEdPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$

an array `sortedSegIdx[i]` with $i = 0 \dots \text{NodeNum} - 1$ for the sorted segment index.

`segStPos[i][axis]` and `segEdPos[i][axis]` with $i = 0 \dots \text{PointCount} - 1$, $\text{axis} = 0 \dots 2$ are derived as follows.

```
for (k = 0; k < 12; k++) {
    segStPos[i * 12 + k][axis] =
        PointPos[i][axis] + segStOffsetTable[k][axis] * TrisoupNodeSize
    segEdPos[i * 12 + k][axis] =
        PointPos[i][axis] + segEdOffsetTable[k][axis] * TrisoupNodeSize
}
```

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

Table 13 — `segStOffsetTable[k][axis]`

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

Table 14 — `segEdOffsetTable[k][axis]`

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

An array `stPos1D[i]` with $i = 0 \dots \text{NodeNum} - 1$ 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 $i = 0 \dots \text{NodeNum} - 1$.

```
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.

8.2.3.1.2 Derivation process for unique segment index

Input to this process are:

the array `segStPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$,

the array `segEdPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$,

the array `sortedSegIdx[i]` with $i = 0 \dots \text{NodeNum} - 1$.

Outputs of this process are:

a variable `numUniqSeg` for the number of unique segments,

an array `uniqSegIdx[i]` with $i = 0 \dots \text{NodeNum} - 1$ for the unique segment index,

an array `uniqSegStPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$ for the start position of an unique segment,

an array `uniqSegEdPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$ 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 $\text{axis} = 0 \dots 2$ 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 $i = 1 \dots \text{NodeNum}$, the following applies:

If `segStPos[i][axis]` is not equal to `uniqSegStPos[uIdx][axis]` with $\text{axis} = 0 \dots 2$ or `segEdPos[i][axis]` is not equal to `uniqSegEdPos[uIdx][axis]` with $\text{axis} = 0 \dots 2$, the following applies:

`uniqSegStPos[uIdx][axis]` and `uniqSegEdPos[uIdx][axis]` with $\text{axis} = 0 \dots 2$ are derived as follows:

```
uniqSegStPos[uIdx][axis] = segStPos[sortedSegIdx[i]][axis]
uniqSegEdPos[uIdx][axis] = segEdPos[sortedSegIdx[i]][axis]
```

`uIdx` is set equal to `(uIdx + 1)`.

`uniqSegIdx[i]` is updated as follows:

```
uniqSegIdx[sortedSegIdx[i]] = uIdx - 1
```

Finally, `numUniqSeg` is derived as follows,

```
numUniqSeg = uIdx
```

8.2.3.1.3 Derivation process for unique segment vertex

Inputs to the process are:

the variable `numUniqSeg`,

the array `uniqSegIdx[i]` with $i = 0 \dots \text{NodeNum} - 1$,

the array `sortedSegIdx[i]` with $i = 0 \dots \text{NodeNum} - 1$.

Output of the process is

the array `segVertex[i]` with $i = 0 \dots \text{NodeNum} - 1$

A variable `vertexCount` is initialized equal to 0.

An array `uniqSegVertex[i]` with $i = 0 \dots \text{numUniqSeg} - 1$ 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 $i = 0 \dots \text{NodeNum} - 1$ is derived as follows:

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

8.2.3.2 Derivation process for the reconstructed triangles

Inputs to the process are:

the array `segStPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$,

the array `segEdPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$,

the array `segVertex[i]` with $i = 0 \dots \text{NodeNum} - 1$

Outputs of the process are:

a variable `numTriangles` for the number of the decoded triangles,

an array `recTriVertex[tIdx][vertex][axis]` with $tIdx = 0 \dots \text{numTriangles} - 1$, $\text{vertex} = 0 \dots 2$, $\text{axis} = 0 \dots 2$ 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 $nIdx = 0 \dots \text{PointCount} - 1$ as the node index.

8.2.3.2.1 Derivation process for the leaf vertex

Inputs to the process are:

the variable `nIdx`,

the array `segVertex[i]` with $i = 0 \dots \text{NodeNum} - 1$,

the array `segStPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$,

the array `segEdPos[i][axis]` with $i = 0 \dots \text{NodeNum} - 1$, $\text{axis} = 0 \dots 2$

Outputs of the process are:

- a variable numVertex for the number of the leaf vertices,
- an array leafVertices[j][axis] with $j = 0 \dots \text{numVertex} - 1$, $\text{axis} = 0 \dots 2$,
- 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 $\text{axis} = 0 \dots 2$ 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 $\text{axis} = 0 \dots 2$ is derived as follows:

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

If segDist[axis] with $\text{axis} = 0 \dots 2$ is greater than 0, the following applies.

`leafVertices[numVertex][axis] += dist`

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

```
}
```

8.2.3.2.2 Sorting process for leafVertices

Inputs to the process are:

- the variable nIdx,
- the variable bkWidth,
- the variable numVertex,

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

Output of the process is

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

This process is skipped if numVertex is less than 3.

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

```
centroid[axis] = 0
for (j = 0; j < numVertex; j++)
    centroid[axis] += leafVertices[j][axis]
centroid[axis] /= numVertex
```

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

```
variance[axis] = 0
for (j = 0; j < numVertex; j++)
    variance[axis] += ((leafVertices[j][axis] - centroid[axis])^2) >> 8
```

A variable minVariance is derived as follows:

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

A variable mainAxis is derived as follows:

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

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

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

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

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

where the function iAtan2() is defined in 5.9.1.

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

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

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

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

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

8.2.3.2.3 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:

$$\text{triStart} = (\text{numVertex} - 3) \times (\text{numVertex} - 2) / 2$$

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

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

$$\text{triOrder}[\text{axis}] = \text{polyTriangles}[\text{triStart} + \text{triIndex}][\text{axis}]$$

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

$$\text{recTriVertex}[\text{numTriangles}][\text{vertex}][\text{axis}] = \text{leafVertices}[\text{triOrder}[\text{vertex}]][\text{axis}]$$

numTriangles is set to (numTriangles+1).

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

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

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

8.2.3.3 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)

8.2.3.3.1 Derivation process of the intersection between triangle and vector

Inputs to the process are:

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

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

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

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

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

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

```
edge1[axis] = v1[axis] - v0[axis]
edge2[axis] = v2[axis] - v0[axis]
rOV[axis] = rayOrg[axis] - rayVec[axis]
```

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

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

A variable $r1$ is calculated as follows:

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

If $r1$ is equal to 0, the process ends.

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

The variable $r2$ is calculated as follows:

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

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

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

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

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

A variable $r3$ is derived as follows:

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

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

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

A variable $rScale$ is calculated as follows:

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

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

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

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

8.2.3.4 Update process of the decoded geometry points

Inputs to the process are:

the variable numPtsOnTriangle,

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

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

```
ptsOnTriangle[q][axis] with axis = 0..2 is removed from the array.  
numPtsOnTriangle--
```

The process is repeated until the values of ptsOnTriangle[p][axis] with p = 0..numPtsOnTriangle – 1, axis = 0..2 are unique from the ptsOnTriangle[q][axis] with q = 0..numPtsOnTriangle – 1, axis = 0..2 .

Finally, the following applies:

```
PointCount = numPtsOnTriangle
```

PointPos[i][axis] with i = 0..PointCount – 1, axis = 0..2 is modified as follows.

```
PointPos[i][axis] = ptsOnTriangle[i][axis]
```

8.2.4 Planar coding mode

8.2.4.1 Eligibility of a node for planar coding mode

For an axis index axisIdx in the range 0 .. 2, the value of eligible_planar_flag[axisIdx] for a current node is determined as follows

```
if (depth == GeomScalingDepth - 1)  
    eligible_planar_flag[axisIdx] = 0  
else if (localDensity >= 3 × 1024)  
    eligible_planar_flag[axisIdx] = 0  
else {  
    eligible_planar_flag[axisIdx] =  
        planeRate[axisIdx] >= geom_planar_mode_th[probable_order[axisIdx]]  
}
```

The variable localDensity is an estimate of the mean number of occupied child nodes in a node. localDensity is initialized to the value localDensity = 1024*4 when starting the geometry decoding process.

The variable planeRate[axisIdx], for axisIdx in the range 0 .. 2, is an estimate of the probability for a node to be planar in the direction perpendicular to the axisIdx-th axis. planeRate[axisIdx] is initialized to the value planeRate[axisIdx] = 128 * 8 when starting the geometry decoding process.

After decoding occupancy_map or occupancy_byte of a current node, the values of localDensity and planeRate[axisIdx] are updated by

```
localDensity = ((localDensity << 8) - localDensity + 1024 × GeometryNodeChildrenCnt) >> 8
```

```

if (isNodePlanar[axisIdx])
    planeRate[axisIdx] = ((planeRate[axisIdx] << 8) - planeRate[axisIdx] + 256 + 128) >> 8
else
    planeRate[axisIdx] = ((planeRate[axisIdx] << 8) - planeRate[axisIdx] + 128) >> 8

```

where `isNodePlanar[axisIdx]` is equal to 1 if the current node is planar in the direction perpendicular to the `axisIdx`-th axis, and is equal to 0 otherwise.

The three values of `probable_order[]` are deduced from the ordering of the three-entry array `planeRate[]` as defined in Table 16.

Table 16 — Determination of the values of `probable_order[]` from `planeRate[]`

Condition	<code>probable_order[0]</code>	<code>probable_order[1]</code>	<code>probable_order[2]</code>
<code>planeRate[0] ≥ planeRate[1] ≥ planeRate[2]</code>	0	1	2
<code>planeRate[0] ≥ planeRate[2] > planeRate[1]</code>	0	2	1
<code>planeRate[1] > planeRate[0] ≥ planeRate[2]</code>	1	0	2
<code>planeRate[1] > planeRate[2] > planeRate[0]</code>	2	0	1
<code>planeRate[2] > planeRate[0] ≥ planeRate[1]</code>	1	2	0
<code>planeRate[2] > planeRate[1] > planeRate[0]</code>	2	1	0

8.2.4.2 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 co-ordinate 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 co-ordinate 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 .. $(1 << (\text{depth} + 1)) - 1$.

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 *t* axis, and `secondary_axisIdx` equal to 1 specifies the *v* axis. When `axisIdx` is equal to 1, `secondary_axisIdx` equal to 0 specifies the *s* axis, and `secondary_axisIdx` equal to 1 specifies the *v* axis. When `axisIdx` is equal to 2, `secondary_axisIdx` equal to 0 specifies the *s* axis, and `secondary_axisIdx` equal to 1 specifies the *t* 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] = statusKnown
            buffer_closest_node_position[axisIdx][coord][candidateIdx][1] = statusKnown
        }

```

```

    buffer_closest_node_status[axisIdx][coord][candidateIdx] = statusUnknown
}

```

where statusKnown is equal to 1 and statusUnknown is equal to 0.

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[axisIdx][XXX]` are provided by the Table 17.

Table 17 — the values of other_axis[axisIdx][XXX]

axisIdx	other_axis[axisIdx][0]	other_axis[axisIdx][1]
0	1	2
1	0	2
2	0	1

The variable `coord_child[]` is the co-ordinates 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 co-

ordinate (sN, tN, vN) of the lower left corner of the child node by `coord_child[0] = sN`, `coord_child[1] = tN` and `coord_child[2] = vN`.

8.2.4.3 Determination of `planarIdx` for the coding of the planar mode flag

When `planar_buffer_disabled` is not equal to 1, 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 co-ordinate 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 neighbour 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))
```

Otherwise, when `planar_buffer_disabled` is equal to 1, the context index `planarIdx`, for a child node and an axis index `axisIdx`, is set to

```
planarIdx = axisIdx
```

8.2.4.4 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, in case `planar_buffer_disabled` is not equal to 1,

```
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 co-ordinate position, within its parent, along the `axisIdx`-th axis, and set equal to 0 if the child node is located at the lower co-ordinate position.

Otherwise, in case `planar_buffer_disabled` is equal to 1, the planar status is set to

```
status[child][axisIdx] = UNKNOWN_STATUS
```

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
}
```

8.2.4.5 Determination of `planePosIdxAngular` for the coding of the vertical plane position

The determination of `planePosIdxAngular` for the arithmetic coding of `plane_position[child][2]` is obtained as follows.

In case `geometry_angular_mode_flag` is equal to 0, i.e. the angular coding mode is not used, the value of `planePosIdxAngular` is set equal to `planePosIdx`.

In case `geometry_angular_mode_flag` is equal to 1, the value of `planePosIdxAngular` is determined from `contextAngular` by

```
if (contextAngular == -1)
    planePosIdxAngular = planePosIdx
else
    planePosIdxAngular = 36 + contextAngular[child]
```

The determination of `contextAngular[child]` for the arithmetic coding of `plane_position[child][2]` is performed as described in section 8.2.5.3.

8.2.4.6 Determination of the probability `proba_planar[]` of good plane position prediction

The information `proba_planar[]` on the probability of good plane position prediction is used in the determination of the direct coding mode activation flag `DirectModeFlagPresent`. The value of `proba_planar[axisIdx]`, for an axis index in the range 0 .. 2, is in the range 1 .. 127 and is deduced as follows for each child node

```
proba_planar[axisIdx] = 127
if (is_planar_flag[child][axisIdx]) {
    if (axisIdx <= 1)
        p = p0[planePosIdx] >> 9
    else
        p = p0[planePosIdxAngular] >> 9
    if (plane_position[child][axisIdx])
        p = 128 - p
    if (p < 1)
        p = 1
    if (p > 127)
        p = 127
    proba_planar[axisIdx] = p
}
```

where `p0[planePosIdx]` (respectively `p0[planePosIdxAngular]`) is the probability, provided by the CABAC before decoding the bit `plane_position[child][axisIdx]`, of having a zero associated with the context. This probability `p0[planePosIdx]` (respectively `p0[planePosIdxAngular]`) is provided as a 16-bit unsigned integer in the range 0 .. 0xffff.

Note that `proba_planar[axisIdx]` does not need to depend on the child node because the direct mode is activated if there is only one occupied child node in the current node.

8.2.5 Angular coding mode

8.2.5.1 Determination of the angular eligibility for a node

The following process applies to a child node `Child` to determine the angular eligibility `angular_eligible[Child]` of the child node. If `geometry_angular_mode_flag` is equal to 0, `angular_eligible[Child]` is set to equal to 0. Otherwise, the following applies

```
midNodeS = 1 << (ChildNodeSizeSLog2 - 1)
midNodeT = 1 << (ChildNodeSizeTLog2 - 1)
sLidar = Abs((sNchild - geomAngularOrigin[0] + midNodeS) << 8) - 128)
tLidar = Abs((tNchild - geomAngularOrigin[1] + midNodeT) << 8) - 128)
rL1 = (sLidar + tLidar) >> 1
deltaAngleR = deltaAngle × rL1
midNodeV = 1 << (ChildNodeSizeVLog2 - 1)
if (deltaAngleR <= (midNodeV << 26))
    angular_eligible[Child] = 0
else
    angular_eligible[Child] = 1
```

where `deltaAngle` is the minimum angular distance between the lasers determined by

$$\text{deltaAngle} = \text{Min}\{ \text{Abs}(\text{laser_angle}[i] - \text{laser_angle}[j]) ; 0 \leq i < j < \text{number_lasers} \},$$

and where (`sNchild`, `tNchild`, `vNchild`) specifying the position of the geometry octree child node `Child` in the current slice.

8.2.5.2 IDCM angular eligibility. Laser index `laserIndex` associated with a node

The following process applies to a child node `Child` to determine the IDCM angular eligibility `idcm4angular[Child]` and the laser index `laserIndex[Child]` associated with the child node.

If the angular eligibility `angular_eligible[Child]` is equal to 0, then `idcm4angular[Child]` is set to 0 and `laserIndex[Child]` index is set to a preset value `UNKOWN_LASER`.

Otherwise, if the angular eligibility `angular_eligible[Child]` is equal to 1, the following applies as a continuation of the process described in 8.2.5.1. Firstly, the inverse `rInv` of the radial distance of the child node from the Lidar is determined

```
r2 = sLidar × sLidar + tLidar × tLidar
rInv = invSqrt(r2)
```

then an angle `theta32` is determined for the child node.

```
vLidar = ((vNchild - geomAngularOrigin[2] + midNodeT) << 1) - 1
theta = vLidar × rInv
theta32 = theta >= 0 ? theta >> 15 : -((-theta) >> 15)
```

Finally, the angular eligibility and the associated laser to the child node are determined as follows, based on the parent node `Parent` of the child node.

```
laserIndex[Child] = UNKOWN_LASER
idcm4angular[Child] = 0
if (laserIndex[Parent] == UNKOWN_LASER || deltaAngleR <= (midNodeV << (26 + 2))) {
    minDelta = 1 << (18 + 7)
    for (j = 0; j < number_lasers; j++) {
        delta = Abs(laser_angle[j] - theta32)
        if (delta < minDelta) {
            minDelta = delta
        }
    }
}
```

```

        laserIndex[Child] = j
    }
}
else
    idcm4angular[Child] = 1

```

8.2.5.3 Determination of the context contextAngular for planar coding mode

The following process applies to a child node Child to determine the angular context contextAngular[Child] associated with the child node.

If the laser index laserIndex[Child] is equal to UNKOWN_LASER, then contextAngular[Child] is set to a preset value UNKOWN_CONTEXT. Otherwise, if the laser index laserIndex[Child] is not equal to UNKOWN_LASER, the following applies as a continuation of the process described in 8.2.5.2.

Firstly, two angular differences m and M relative to a lower plane and an upper plane are determined.

```

thetaLaserDelta = laser_angle[laserIndex[Child]] - theta32
Hr = laser_correction[laserIndex[Child]] × rInv;
thetaLaserDelta += Hr >= 0 ? -(Hr >> 17) : ((-Hr) >> 17)
vShift = (rInv << (ChildNodeSizeVLog2 + 1)) >> 17
m = Abs(thetaLaserDelta - vShift)
M = Abs(thetaLaserDelta + vShift)

```

Then, the angular context is deduced from the two angular differences.

```

contextAngular[Child] = m > M
diff = Abs(m - M)
if (diff >= rInv >> 15) contextAngular[Child] += 2;
if (diff >= rInv >> 14) contextAngular[Child] += 2;
if (diff >= rInv >> 13) contextAngular[Child] += 2;
if (diff >= rInv >> 12) contextAngular[Child] += 2;

```

8.3 Attribute decoding

Inputs to this process are:

the attribute parameter set and the associated bitstream,

Output of the process is a series of the decoded point PointAttr[i][cldx], where i is in the range of 0 to PointCount – 1 and cldx is in the range of 0 to AttrDim – 1.

The attributes may have multiple components.

Each attribute component has been transform coded by a spatial transform, quantized, and entropy coded, to produce its bitstream. The attribute decoder must invert this process for each attribute component, to produce a decoded attribute component.

When attr_coding_type is equal to 0, RAHT decoding process in clause 8.3.1 is invoked.

Otherwise, if attr_coding_type is equal to 1, LoD with Predicting Transform decoding process in clause 8.3.3 is invoked.

Otherwise (attr_coding_type is equal to 2), LoD with Lifting Transform decoding process in clause 8.3.2 is invoked.

8.3.1 Region adaptive hierarchical transform decoding process

8.3.1.1 General

The output of this process is the array `PointsAttr` with elements `PointsAttr[i][cldx]` with $i = 0 \dots \text{PointCount} - 1$, and $\text{cldx} = 0 \dots \text{AttrDim} - 1$. Each element with index i of `PointsAttr` is associated with a position given by the array `PointPos` with the same index i .

The variable `CoeffIdx`, specifying a current position in the decoded values array, is initialized to 0.

If `PointCount` equal to 1, the following applies:

- The variable `NumRahtLevels`, specifying the number of 3D transform levels, is set equal to 1
- The array `PointRegionboxDeltaQp`, specifying the value of delta QP per point based on region, are derived according to the RAHT region-wise qp derivation process (8.3.1.3).
- The scaling process for RAHT coefficients (8.3.1.6) is invoked for each component `cldx` in the range $0 \dots \text{AttrDim} - 1$, with the single-element `coeff` set equal to `value[cldx][CoeffIdx]`, the position (sTn, tTn, vTn) set equal to $(0, 0, 0)$, the 3D transform level `lvl` set equal to 0, and the variable `cldx` as inputs. The reconstructed samples of the output array `PointAttr[0][cldx]` is set equal to the single-element output array of scaled transform coefficients `d`.

Otherwise, the following applies:

The array `Weights`, specifying transform coefficient weights, and the variable `NumRahtLevels`, specifying the number of 3D transform levels, are derived according to the RAHT weights derivation process (8.3.1.2).

The array `PointRegionboxDeltaQp`, specifying the value of delta QP per point based on region, are derived according to the RAHT region-wise qp derivation process (8.3.1.3).

Reconstruction proceeds level by level from the root of the transform tree to the leaves, each using the reconstruction of the previous level.

For each 3D transform level in the descending range $\text{lvl} = \text{NumRahtLevels} - 1 \dots 0$, 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 $3 \times \text{lvl}$, and `inheritDc` as inputs. The output is the array `recon` with elements `recon[s][t][v][cldx]`.
- 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][cldx]` are derived as follows with $i = 0 \dots \text{PointCount} - 1$:

- The point position variables (sPt, tPt, vPt) are set equal to `PointPos[i][j]`, with $j = 0 \dots 2$ respectively.
- If `Weights[0][sPt][tPt][vPt]` is equal to 1, the following applies:

```
for (cIdx = 0; cIdx < AttrDim; cIdx++)
    PointAttr[i][cIdx] = DivExp2RoundHalfInf(recon[sPt][tPt][vPt], 15)
```

- Otherwise, $\text{Weights}[0][sPt][tPt][vPt]$ is greater than 1, the following process is used to reconstruct samples $\text{PointAttr}[i+j][cIdx]$ for $j = i \dots \text{Weights}[0][sPt][tPt][vPt] - 1$:

- The $(\text{AttrDim}) \times (2)$ sized array xxx is initialized as follows:

```
for (cIdx = 0; cIdx < AttrDim; cIdx++)
    xxx[cIdx][0] = recon[xPt][yPt][zPt]
```

- For each w_i in the descending range $\text{Weights}[0][sPt][tPt][vPt] - 1 \dots 1$, the following applies:
 - The scaling process for RAHT coefficients is invoked for each component $cIdx$ in the range $0 \dots \text{AttrDim} - 1$, with the single-element coeff set equal to residual_values[cIdx][CoeffIdx], the 3D transform level lvl set equal to 0, and the variable cIdx as inputs. The array element $\text{xxx}[cIdx][1]$ is set equal to the single-element output array of scaled transform coefficients d.
 - CoeffIdx is incremented by 1.
 - For each component $cIdx$ in the range $0 \dots \text{AttrDim} - 1$, the following applies:
 - The inverse two-point transform process is invoked with the array $\text{xxx}[cIdx][j]$ with $j = 0 \dots 1$, and the array w equal to $\{w_i, 1\}$ as inputs. The output is the two-element array r.
 - The value of $\text{xxx}[cIdx][0]$ is replaced by $r[0]$
 - The output $\text{PointAttr}[i + w_i][cIdx]$ is derived as follows:

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

- The output $\text{PointAttr}[i][cIdx]$ for $cIdx = 0 \dots \text{AttrDim} - 1$ is derived as follows:

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

8.3.1.2 RAHT weights derivation process

The outputs of this process are:

- the array Weights, with entries $\text{Weights}[lvl][s][t][v]$ equal to the number of points represented by a coefficient at position (s, t, v) at the lvl 'th 1D level of the RAHT transform,
- the variable NumRahtLevels indicating the number of 3D levels in the transform tree.

The elements of the array Weights are derived as follows:

```
for (i = 0; i < PointCount; i++) {
    s = PointPos[i][0]
    t = PointPos[i][1]
    v = PointPos[i][2]
    Weights[0][s][t][v] += 1;
}

for (lvl = 1, done = 0; !done;)
    for (j = 0; j < 3; j++, lvl++)
        for (i = 0; i < PointCount; i++) {
```

```

s = PointPos[i][0] >> ((lvl + 0) / 3)
t = PointPos[i][1] >> ((lvl + 1) / 3)
v = PointPos[i][2] >> ((lvl + 2) / 3)

Weights[lvl][s][t][v] += 1;
if (Weights[lvl][s][t][v] == PointCount)
    done = 1;
}

```

The variable NumRahtLevels is set equal to lvl / 3.

8.3.1.3 RAHT region-wise qp derivation process

The outputs of this process are the array PointRegionboxDeltaQp, with entries PointRegionboxDeltaQp[lvl][s][t][v] equal to the value of delta QP per point based on region represented by a coefficient at position (s, t, v) at the lvl'th 1D level of the RAHT transform.

The output array PointRegionboxDeltaQp is initialize to -1. The variable RegionQpBitShift is set to equal to 4.

```

for (i = 0; i < PointCount; i++) { s = PointPos[i][0]
    t = PointPos[i][1]
    v = PointPos[i][2]

    PointRegionboxDeltaQp[0][s][t][v] = 0

    if (!ash_attr_region_qp_delta_present_flag)
        continue
    isPointInRegion = 1
    for (k = 0; k < 3; k++)
        isPointInRegion &=
            AttrRegionQpOrigin[k] <= PointPos[i][k]
            && PointPos[i][k] < AttrRegionQpOrigin[k] + AttrRegionQpSize[k]

    if (isPointInRegion)
        PointRegionboxDeltaQp[0][s][t][v] = ash_attr_region_qp_delta << RegionQpBitShift
}

for (lvl = 1, lvl <= (NumRahtLevels - 1) * 3; lvl++){
    for (i = 0; i < PointCount; i++) {
        s = PointPos[i][0] >> ((lvl + 0) / 3)
        t = PointPos[i][1] >> ((lvl + 1) / 3)
        v = PointPos[i][2] >> ((lvl + 2) / 3)

        if (PointRegionboxDeltaQp[lvl][s][t][v] == -1){
            prevS = (lvl % 3 == 0)? s + 1: s;
            prevT = (lvl % 3 == 2)? t + 1: t;
            prevV = (lvl % 3 == 1)? v + 1: v;
            lQp = PointRegionboxDeltaQp[lvl - 1][s][t][v];
            rQp = PointRegionboxDeltaQp[lvl - 1][prevS][prevT][prevV];

            if (lQp == -1)
                PointRegionboxDeltaQp[lvl][s][t][v] = rQp;
            else if (rQp == -1)
                PointRegionboxDeltaQp[lvl][s][t][v] = lQp;
            else
                PointRegionboxDeltaQp[lvl][s][t][v] = ((lQp + rQp) >> 1);
        }
    }
}

```

8.3.1.4 Reconstruction process for a single 3D RAHT level

The inputs to this process are:

- a variable lvl indicating the current 1D transform level.

- a variable `inheritDc` indicating if DC coefficients should be inherited from a previous reconstruction level.

The outputs of this process are the array `recon` of reconstructed values and an updated variable `CoeffIdx`.

An array, `nodes`, of occupied transform tree nodes in the current level with elements `nodes[idx][dim]` is derived using a Morton order traversal of the array `Weights` as follows:

```
for (mIdx = 0, nIdx = 0; mIdx < (1 << (3 × NumRahtLevels - 3 - lvl)); mIdx++) {
    (sN, tN, vN) = MortonToTuple(mIdx)
    if (Weights[lvl + 3][sN][tN][vN] == 0)
        continue
    nodesS[nIdx] = 2 × sN
    nodesT[nIdx] = 2 × tN
    nodesV[nIdx] = 2 × vN
    nIdx++
}
```

The variable `numNodesInLvl` is set equal to `nIdx`.

For each occupied transform tree node with `nIdx = 0 .. numNodesInLvl - 1`, the following steps apply:

The position variables (`sTn`, `tTn`, `vTn`) indicating the location of a transform tree node are initialized with the values of `nodesS[nIdx]`, `nodesT[nIdx]`, and `nodesV[nIdx]` respectively.

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

```
for (childIdx = 0; childIdx < 8; childIdx++) {
    (ds, dt, dv) = MortonToTuple(childIdx)
    if (inheritDc && childIdx == 0)
        continue
    if (Weights[lvl][sTn + ds][tTn + dt][vTn + dv] == 0)
        continue
    for (cIdx = 0; cIdx < AttrDim; cIdx++)
        coeff[cIdx][childIdx] = residual_values[cIdx][CoeffIdx]
    CoeffIdx++
}
```

For each component of the attribute, the following ordered steps are performed:

- The reconstruction process for a 2×2×2 transform tree node is invoked with the node position (`sTn`, `tTn`, `vTn`), and the eight-element array `coeff[cIdx][childIdx]` with `childIdx = 0 .. 7` as inputs. The output is the eight-element array `r`.
- The array of reconstructed values, `recon`, is updated as follows:

```
for (childIdx = 0; childIdx < 8; childIdx++) {
    (ds, dt, dv) = MortonToTuple(childIdx)
    recon[sTn + ds][tTn + dt][vTn + dv][cIdx] = r[childIdx]
}
```

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

The inputs to this process are:

- a position (`sTn`, `tTn`, `vTn`) and 1D level, `lvl`, specifying the location of a transform tree node in the RAHT transform tree,
- a variable `cIdx` specifying the index of an attribute component,
- an array, `coeff`, of packed quantized transform coefficients.

The output of this process is an eight-element array, *r*, of reconstructed values

The scaling process for RAHT coefficients is invoked with the eight-element array *coeff*, the position (*sTn*, *tTn*, *vTn*), the 3D transform level *lvl* set equal to *lvl* / 3, and the variable *cIdx* as inputs. The output is an eight-element array of scaled transform coefficients *d*.

If *RahtPredictionEnabled* is equal to 1, the following applies:

- The transform prediction upsampling process is invoked with the position (*sTn*/2, *tTn*/2, *vTn*/2) and the variable *lvl* set equal to *lvl* + 3. The output is the eight-element array *p* of upsampled prediction values.
- The forward transform process for 2×2×2 blocks is invoked with the position (*sTn*, *tTn*, *vTn*) and level *lvl* of the current transform tree node, and the array *p* of upsampled prediction values. The output is the eight-element array *q* of transformed prediction values.

The scaled transform coefficients *d*, the transformed prediction values *q*, and an inherited DC value are summed to produce the transform coefficient array *e* as follows:

```
for (i = inheritDc; i < 8; i++)
    e[i] = d[i] << 15

if (inheritDc) {
    e[0] = DivExp2RoundHalfInf(PrevRecon[sTn / 2][tTn / 2][vTn / 2][cIdx], 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 (*sTn*, *tTn*, *vTn*) and level *lvl* of the current transform tree node, and the array *e* of transform coefficients. The output is the eight-element array *r* of inverse transformed values.

8.3.1.6 Scaling process for RAHT coefficients

The inputs to this process are:

- an *n*-element array *coeff* of quantized coefficients
- a position (*sTn*, *tTn*, *vTn*) specifying the location of a transform tree node in the RAHT transform tree
- a variable *lvl* indicating the 3D transform level of the coefficients
- a variable *cIdx* specifying the index of an attribute component

The output is an *n*-element array of scaled transform coefficients *d*.

The variable *qlayer* is set equal to Min(NumLayerQP – 1, NumRahtLevels – *lvl* – 1).

The scaled transform coefficient *d*[*i*][*cIdx*] with *i* = 0 .. *n* – 1, and *cIdx* = 0 .. AttrDim – 1 is derived as follows:

```
for (i = 0, childIdx = 0; childIdx < 8 && i < n; childIdx++) {
    (ds, dt, dv) = MortonToTuple(childIdx)
    if (Weights[lvl][sTn + ds][tTn + dt][vTn + dv] == 0)
        continue
    deltaRegionQp = PointRegionboxDeltaQp[lvl][sTn + ds][tTn + dt][vTn + dv]
    >> RegionQpBitShift
```

```

qstepY = QpToQstep(SliceQpY[qlayer] + deltaRegionQp, 1)
qstepC = QpToQstep(SliceQpC[qlayer] + deltaRegionQp, 0)
for (cIdx = 0; cIdx < AttrDim; cIdx++)
    d[i][cIdx] = DivExp2RoundHalfUp(coeff[i][cIdx] ×
        (!cIdx ? qstepY : qstepC), 8)
    i++
}

```

8.3.1.7 Transform prediction upsampling process

The inputs to this process are:

- a position (sTn, tTn, vTn) and 1D level, lvl, specifying the location of a transform tree node in the RAHT transform tree, and
- a variable cIdx specifying the index of an attribute component.

The output of this process are:

- an eight-element array p of upsampled values.
- the array of NeighCount, with entries NeighCount[lvl][s][t][v] equal to the number of valid neighbour transform tree node where more than equal to one point exist represented by a coefficient at position (s, t, v) at the lvl'th 1D level of the RAHT transform.

NeighCount[lvl][s][t][v] is initialized as 0. For each row in Table 18, the following applies:

If lvl / 3 is not equal to NumRahtLevels – 1 and NeighCount[lvl + 3][sTn / 2][tTn / 2][vTn / 2] is less than raht_prediction_threshold0, for each child position childIdx in the range 0 to 7, inclusive, the following applies:

```

for (childIdx = 0; childIdx < 8; childIdx++)
    p[childIdx] = 0

```

Otherwise, for each row in Table 18, the following applies:

```

cs = sTn + ds
ct = tTn + dt
cv = vTn + dv
if (Weights[lvl][cs][ct][cv] > 0)
    NeighCount[lvl][sTn][tTn][vTn] += 1

```

If NeighCount[lvl][sTn][tTn][vTn] is less than raht_prediction_threshold1, for each child position childIdx in the range 0 to 7, inclusive, the following applies:

```

for (childIdx = 0; childIdx < 8; childIdx++)
    p[childIdx] = 0

```

Otherwise, the following applies:

The upsampled 2×2×2 block located at the position (sTn, tTn, vTn) is derived as follows. For each row in Table 18, the following applies:

```

cs = sTn + ds
ct = tTn + dt
cv = vTn + dv
if (Weights[lvl][cs][ct][cv] > 0) {
    neighVal = Recon[cs][ct][cv][cIdx]
    value = DivFp(neighVal, iSqrt(Weights[lvl][cs][ct][cv] << 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++) {
  (ds, dt, dv) = MortonToTuple(childIdx)
  pred = DivFp(sumDc[childIdx], sumWn[childIdx], 15)
  pred *= iSqrt(Weights[lvl - 3][2 × sTn + ds][2 × tTn + dt][2 × vTn + dv] << 30)
  p[childIdx] = DivExp2RoundHalfInf(pred, 15)
}

```

Table 18 — Weighting matrix for determining upsampled child position weights, `wn[childIdx]`, for various neighbour position offsets (`dx`, `dy`, `dz`)

Neighbour offset			wn[childIdx]							
ds	dt	dv	0	1	2	3	4	5	6	7
0	0	0	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										

8.3.1.8 Forward transform process for 2×2×2 blocks

The inputs to this process are:

- a position (`sTn`, `tTn`, `vTn`) and level, `lvl`, specifying the position of a transform tree node,
- an eight-element array, `p`, of values to be transformed.

The output of this process is an eight-element array, `q`, of transformed values.

For each row of Table 19 in sequential order, the array p is modified by transforming a pair of values by invoking the forward two-point transform process 8.3.1.9 with the input array x equal to $\{p[i], p[j]\}$, and the array w equal to $\{w_i, w_j\}$. The output y updates the array $p[i] = y[0]$, $p[j] = y[1]$.

The output array q is derived as $q[s] = p[t]$ with $s = 0 \dots 7$ and the value of t derived from s according to Table 20.

Table 19 — Ordering of coefficients and respective weights for use in the forward and inverse (reverse order) two-point transform processes

i	j	w_i	w_j
0	1	$w[lvl][sTn + 0][tTn + 0][vTn]$	$w[lvl][sTn + 0][tTn + 0][vTn + 1]$
2	3	$w[lvl][sTn + 0][tTn + 1][vTn]$	$w[lvl][sTn + 0][tTn + 1][vTn + 1]$
4	5	$w[lvl][sTn + 1][tTn + 0][vTn]$	$w[lvl][sTn + 1][tTn + 0][vTn + 1]$
6	7	$w[lvl][sTn + 1][tTn + 1][vTn]$	$w[lvl][sTn + 1][tTn + 1][vTn + 1]$
4	6	$w[lvl + 1][sTn + 1][tTn][vTn]$	$w[lvl + 1][sTn + 1][tTn + 1][vTn]$
0	2	$w[lvl + 1][sTn + 0][tTn][vTn]$	$w[lvl + 1][sTn + 0][tTn + 1][vTn]$
0	4	$w[lvl + 2][sTn + 0][tTn][vTn]$	$w[lvl + 2][sTn + 1][tTn + 0][vTn]$

Table 20 — Indexes of transform coefficients in decoding order (s)

s	0	1	2	3	4	5	6	7
t	0	4	6	2	7	5	3	1

8.3.1.9 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)
```

8.3.1.10 Inverse transform process for 2×2×2 blocks

The inputs to this process are:

- a position (sTn, tTn, vTn) and level, lvl , specifying the position of a transform tree node, and

- an eight-element array, e , of transform coefficients.

The output of this process is an eight-element array, r , of inverse transformed values.

The output array r is initialized as $r[t] = e[s]$ with $s = 0 \dots 7$ and the value of t derived from s according to Table 20.

For each row of Table 19 in reverse order, the array r is modified by transforming a pair of values by invoking the inverse two-point transform process 8.3.1.11 with the input array x equal to $\{r[i], r[j]\}$, and the array w equal to $\{w_i, w_j\}$. The output y updates the array $r[i] = y[0]$, $r[j] = y[1]$.

8.3.1.11 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)
```

8.3.2 LoD with Lifting Transform decoding process

Inputs of this process are:

a variable `minGeomNodeSizeLog2` specifying the number of octree layers that are skipped to decode.

The output of the process is

a series of the decoded attribute values `attributeValues[i][a]`, where i is in the range of 0 to `PointCount` – 1, inclusive, and a in the range of 0 to `AttrDim` – 1, inclusive.

First a variable `PointNumInSlice` is set to `gsh_num_points` in the active slice.

NOTE 1 – When `lifting_scalability_enabled_flag` is equal to 1, `PointCount` may be smaller than `PointNumInSlice` due to `minGeomNodeSizeLog2` larger than 0.

This process invokes the sub-processes in the following order.

The level of detail generation process in clause 8.3.2.1 is invoked. The output of this process are stored in `indexes[i]`, `neighbours[i][n]`, `neighboursCount[i]`, `neighboursDistance2[i][n]`, and `pointCountPerLevelOfDetail[l]`, where i is in the range of 0 to `PointCount` – 1, inclusive, n in the range of 0 to `NumPredNearestNeighbours` – 1, inclusive, l is in the range of 0 to `LevelDetailCount` – 1, inclusive.

The prediction weight derivation process in 8.3.2.4 is invoked with the parameters `neighbours`, `neighboursCount` and `neighboursDistance2`. The output of this process is stored in

`predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

The quantization weights derivation process in 8.3.2.5 is invoked with the parameters `indexes`, `neighbours`, `neighboursCount`, `predictionWeights`, and `pointNumPerLoD`. The output of this process is stored in `quantizationWeights[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

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

The inverse lifting process in 8.3.2.7 is invoked with the parameters `unquantAttributeCoefficients`, `quantizationWeights`, `predictionWeights` and `pointCountPerLevelOfDetail`. This process updates the attribute coefficients `unquantAttributeCoefficients[i][j]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `j` in the range of 0 to `AttrDim - 1`, inclusive.

The reconstructed attributes values are obtained as follows.

```
for (i = 0; i < PointCount; i++) {
    for (j = 0; j < AttrDim; j++) {
        value = divExp2RoundHalfInf(unquantAttributeCoefficients[i][j], 8);
        if (AttrDim == 0) {
            maxAttribute = (1 << (attribute_bitdepth_minus1[ash_attr_sps_attr_idx] + 1)) - 1
        }
        else {
            maxAttribute = (1 << (attribute_secondary_bitdepth_minus1[ash_attr_sps_attr_idx] +
1)) - 1
        }
        attributeValues[i][j] = Clip3(value, 0, maxAttribute);
    }
}
```

8.3.2.1 Level of Detail Generation

The input of the process is

a available `minGeomNodeSizeLog2` specifying the number of octree layers that are skipped to decode.

The outputs of the process are

an array of point indexes `indexes[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

a series of nearest neighbours indexes `neighbours[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

an array of nearest neighbours count `neighboursCount[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

an array of nearest neighbours squared distances `neighboursDistance2[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

an array `pointCountPerLevelOfDetail[l]`, where `l` is in the range of 0 to `LevelDetailCount - 1`, inclusive.

An array of distances `sampling[l]`, where `l` is in the range of 0 to `LevelDetailCount - 2`, inclusive, is derived as followings:

```
if (lifting_lod_regular_sampling_enabled_flag) {
```

```

    for (lod = 0; lod < LevelDetailCount - 1; lod++)
        sampling[lod] = lifting_sampling_period_minus2[lod] + 2
}
else {
    for (lod = 0; lod < LevelDetailCount - 1; lod++)
        sampling[lod] = LiftingSamplingDistanceSquared[lod]
}

```

Depending on the value of `lifting_lod_regular_sampling_enabled_flag`, the level of detail generation process re-organizes the points into a set of refinement levels $(R_l)_{l=0\dots L-1}$, according to a the set of Euclidian distances (i.e., `lifting_lod_regular_sampling_enabled_flag` equals 0) or sampling period (i.e., `lifting_lod_regular_sampling_enabled_flag` equals 1) specified by the array `sampling[l]`.

If `lifting_lod_regular_sampling_enabled_flag` equals 0, the array `sampling[l]` represents squared sampling distances verifying the following condition:

```
sampling[l-1] < sampling[l]
```

If `lifting_lod_regular_sampling_enabled_flag` equals 1, the array `sampling[l]` represents sampling periods verifying the following condition:

`sampling[l] > 1`. If `lifting_scalability_enabled_flag` equals 1, the level of detail degeneration process re-organizes the points into a set of refinement levels $(R_l)_{l=0\dots L-1}$, according to octree nodes based on geometry. Depending on the value of `samplingFromLastFlag`, the first point in the node (i.e., `samplingFromLastFlag` equals 0) or the last point in the node (i.e., `samplingFromLastFlag` equals 1) is sampled.

First, the point sorting process based on Morton code in clause 5.9.8 is invoked. Let `Order[i]` be the array of point indexes sorted according to their Morton codes and `McodeUnsorted` the array of unsorted Morton codes.

Next, the following procedure is applied in order to compute both the level of detail reordering and the points nearest neighbours.

```

unprocessedPointCount = PointCount
for (i = 0; i < unprocessedPointCount; i++) {
    unprocessedPointIndexes[i] = Order[i]
}
for (lod = 1; lod < LevelDetailCount; lod++)
    unprocessedPointCountPerLevelOfDetail[lod] = 0;
unprocessedPointCountPerLevelOfDetail[0] = PointCount

```

If `lifting_scalability_enabled_flag` is equal to 0, the following is applied.

```

endIndex = 0
assignedPointCount = 0
for (lod = 0; unprocessedPointCount > 0 && lod < LevelDetailCount; lod++) {
    nonAssignedPointCount = 0
    startIndex = assignedPointCount
    if (lod == LevelDetailCount - 1) {
        for (i = 0; i < unprocessedPointCount; i++)
            assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]
    } else {
        nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[0]
        for (i = 1; i < unprocessedPointCount; i++) {
            foundAssignedPointWithinDistanceFlag = 0
            if ((lifting_lod_regular_sampling_enabled_flag == 1) {
                foundAssignedPointWithinDistanceFlag = (i % sampling[lod]) != 0
            } else {
                for (axis = 0; axis < 3; axis++)
                    currentPos[axis] = PointPos[unprocessedPointIndexes[i]][axis]
                k = 0
                while (k++ < LiftingSearchRange) {
                    for (axis = 0; axis < 3; axis++)

```

```

        d[axis] = currentPos[axis] -
PointPos[nonAssignedPointIndexes[nonAssignedPointCount - 1]][axis]
        d2 = InnerProduct(d[], d[])
        if (d2 <= sampling[lod]) {
            foundAssignedPointWithinDistanceFlag = 1
            break
        }
    }
}
if (foundAssignedPointWithinDistance == 1)
    assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]
else
    nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]
}
}
endIndex = assignedPointCount
computeNearestNeighbours(
    startIndex, endIndex,
    lod, assignedPointIndexes,
    McodeUnsorted, nonAssignedPointCount,
    nonAssignedPointIndexes)
unprocessedPointCountPerLevelOfDetail[lod+1] = nonAssignedPointCount
unprocessedPointCount = nonAssignedPointCount
unprocessedPointIndexes = nonAssignedPointIndexes //NOTE the left and the right are
pointer of the array
}

```

Otherwise (lifting_scalability_enabled_flag is equal to 1), the following is applied;

```

endIndex = 0
assignedPointCount = 0
for (lod = minGeomNodeSizeLog2; unprocessedPointCount > 0; lod++) {
    startIndex = assignedPointCount
    nonAssignedPointCount = 0
    samplingFromLastFlag = lod & 1
    for (i = 0; i < unprocessedPointCount; i++) {
        currVoxelIndex = McodeUnsorted[unprocessedPointIndexes[i]] >> (3*(lod+1))
        if (samplingFromLastFlag == 0){
            if (i == 0)
                nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]
            else {
                prevVoxelIndex = McodeUnsorted[unprocessedPointIndexes[i-1]] >> (3*(lod+1))

                if (currVoxelIndex > prevVoxelIndex)
                    nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]
                else
                    assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]
            }
        } else {
            if (i == (unprocessedPointCount - 1))
                nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]
            else {
                nextVoxelIndex = McodeUnsorted[unprocessedPointIndexes[i+1]] >> (3*(lod+1))
                if (currVoxelIndex < nextVoxelIndex)
                    nonAssignedPointIndexes[nonAssignedPointCount++] = unprocessedPointIndexes[i]
                else
                    assignedPointIndexes[assignedPointCount++] = unprocessedPointIndexes[i]
            }
        }
    }
}
endIndex = assignedPointCount
if (startIndex != endIndex) {
    numOfPointInSkipped = PointNumInSlice - PointCount
    if ((endIndex - startIndex) > (startIndex + numOfPointInSkipped)){
        for (loop = 0; loop < lod - minGeomNodeSizeLog2; loop++){
            computeNearestNeighbours(
                PointCount - unprocessedPointCountPerLevelOfDetail[loop],
                PointCount - unprocessedPointCountPerLevelOfDetail[loop+1],
                loop + minGeomNodeSizeLog2, assignedPointIndexes,

```



```

        McodeUnsorted, nonAssignedPointCount,
        nonAssignedPointIndexes)
    }
}
computeNearestNeighbours(
    startIndex, endIndex,
    lod, assignedPointIndexes,
    McodeUnsorted, nonAssignedPointCount,
    nonAssignedPointIndexes)
unprocessedPointCountPerLevelOfDetail[lod+1] = nonAssignedPointCount
unprocessedPointCount = nonAssignedPointCount
unprocessedPointIndexes = nonAssignedPointIndexes
}

```

Then, the following procedure is applied:

```

for (i = 0; i < PointCount; i++)
    indexes[PointCount- 1 - i] = assignedPointIndexes[i]

for (lod = 0; lod < LevelDetailCount; lod++)
    pointCountPerLevelOfDetail[lod] = unprocessedPointCountPerLevelOfDetail[LevelDetailCount
- 1 - lod]

```

8.3.2.2 Definition of computeNearestNeighbours()

Inputs of this process are:

two variables `startIndex` and `endIndex` indicating the range of points for which the nearest neighbours should be computed

a variable `currentLayer` specifying LoD layer number, where a series of the decoded geometry point belong

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

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

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

an array of point indexes `nonAssignedPointIndexes[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

The outputs of the process are

a series of nearest neighbours indexes `neighbours[i][j]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `j` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

an array of nearest neighbours counts `neighboursCount[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

an array of nearest neighbours squared distances `neighboursDistance2[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

The nearest neighbours of the points are computing as follows.

```

if (nonAssignedPointCount == 0) {
    for (i = startIndex; i < endIndex; i++)
        neighboursCount[assignedPointIndexes[i]] = 0
} else {

```

```

j = 0
for (i = startIndex; i < endIndex; i++) {
    currentIndex = assignedPointIndexes[i]
    currentMortonCode = McodeUnsorted[currentIndex]
    currentPos = PointPos[currentIndex]
    while (j < nonAssignedPointCount &&
           currentMortonCode >= McodeUnsorted[nonAssignedPointIndexes[j]])
        j++
    }
    j = Min(nonAssignedPointCount - 1, j)
    j0 = Max(0, j - LiftingSearchRange)
    j1 = Min(nonAssignedPointCount, j + LiftingSearchRange + 1)
    neighboursCount[currentIndex] = 0
    k = 0
    for (k = j0; k < j1 ; k++) {
        neighbourIndex = nonAssignedPointIndex[k]
        neighbourPos = PointPos[neighbourIndex]
        if (lifting_scalability_enabled_flag){
            for (axis = 0; axis < 3; axis++){
                currentPos[axis] = (currentPos[axis] >> currentLayer) << currentLayer
                neighbourPos[axis] = (neighbourPos[axis] >> currentLayer) << currentLayer
            }
        }
        for (axis = 0; axis < 3; axis++){
            d[axis] = liftingNeighbourBiasStv[axis] * (currentPos[axis] - neighbourPos[axis])
        }
        d2 = InnerProduct(d[], d[])
        if (Abs(k - j) <= 3)
            insertIndex = k - j > 0 ? ((k - j) << 1) - 1 : (j - k) << 1;
        else if (k > j)
            insertIndex = 7 + k - j;
        else
            insertIndex = LiftingSearchRange + 4 + j - k;
        if (neighboursCount[currentIndex] < NumPredNearestNeighbours) {
            p = neighboursCount[currentIndex]
            neighbours[currentIndex][p] = neighbourIndex;
            neighboursDistance2[currentIndex][p] = d2
            neighboursInsertIndex[currentIndex][p] = insertIndex;
            neighboursCount[currentIndex]++
            sortNeighbours(neighboursCount[currentIndex],
                           neighbours[currentIndex],
                           neighboursDistance2[currentIndex] ,
                           neighboursInsertIndex[currentIndex])
        } else if (d2 < neighboursDistance2[currentIndex][NumPredNearestNeighbours-1]) {
            neighbours[currentIndex][NumPredNearestNeighbours-1] = neighbourIndex
            neighboursDistance2[currentIndex][NumPredNearestNeighbours-1] = d2
            neighboursInsertIndex[currentIndex][NumPredNearestNeighbours - 1] = insertIndex
            sortNeighbours(NumPredNearestNeighbours,
                           neighbours[currentIndex],
                           neighboursDistance2[currentIndex] ,
                           neighboursInsertIndex[currentIndex]);
        }
    }
}
if (currentLayer >= LevelDetailCount - IntraLodPredNumLayers) {
    j1 = Min(endIndex, k + LiftingSearchRange)
    for (k = i + 1; k < j1; k++) {
        neighbourIndex = assignedPointIndex[k]
        neighbourPos = PointPos[neighbourIndex]
        for (axis = 0; axis < 3; axis++){
            d[axis] = liftingNeighbourBiasStv[axis] * (currentPos[axis] - neighbourPos[axis])
        }
        d2 = InnerProduct(d[], d[])
        insertIndex = 2 * LiftingSearchRange + (k - i);
        if (neighboursCount[currentIndex] < NumPredNearestNeighbours) {
            p = neighboursCount[currentIndex]
            neighbours[currentIndex][p] = neighbourIndex
            neighboursDistance2[currentIndex][p] = d2
            neighboursInsertIndex[currentIndex][p] = insertIndex
            neighboursCount[currentIndex]++
            sortNeighbours(neighboursCount[currentIndex],
                           neighbours[currentIndex],
                           neighboursDistance2[currentIndex] ,
                           neighboursInsertIndex[currentIndex]);
        }
    }
}

```

```

    } else if (d2 < neighboursDistance2[currentIndex][NumPredNearestNeighbours - 1]) {
        neighbours[currentIndex][NumPredNearestNeighbours - 1] = neighbourIndex
        neighboursDistance2[currentIndex][NumPredNearestNeighbours - 1] = d2
        neighboursInsertIndex[currentIndex][NumPredNearestNeighbours - 1] = insertIndex
        sortNeighbours(NumPredNearestNeighbours,
            neighbours[currentIndex],
            neighboursDistance2[currentIndex] ,
            neighboursInsertIndex[currentIndex])
    }
}
}
}

```

8.3.2.3 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.

8.3.2.4 Prediction weights derivation process

The inputs of this process are:

a series of nearest neighbours indexes neighbours[i][j], where i is in the range of 0 to PointCount – 1, inclusive, and j in the range of 0 to NumPredNearestNeighbours – 1, inclusive.

an array of nearest neighbours counts neighboursCount[i], where i is in the range of 0 to PointCount – 1, inclusive.

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

The output is:

an array of prediction predictionWeights[i][n], where i is in the range of 0 to PointCount – 1, inclusive, and n in the range of 0 to NumPredNearestNeighbours – 1, inclusive.

The prediction weights derivation process proceeds as follows:

```

MaxWeightValue = 1 << 8;
for (i = 0; i < PointCount; i++) {
    while (neighboursCount[i] > 1 &&
        neighboursDistance2[i][0] > 0 &&
        (neighboursDistance2[neighbourCount[i] - 1][0] >> 8) >

```

```

        neighboursDistance2[i][0]) {
    neighboursCount[i]--;
}
if (neighboursCount[i] < 2 || neighboursDistance2[i][0] == 0) {
    neighboursCount[i] = 1;
    predictionWeights[i][0] = MaxWeightValue;
} else {
    bitCount = iLog2(neighboursDistance2[i][0]) + 2;
    shiftDistance = bitCount > 8 ? bitCount - 8 : 0;
    biasDistance = ((1 << shift) >> 1);
    if (neighboursCount[i] == 2) {
        d0 = (neighboursDistance2[i][0] + biasDistance) >> shiftDistance;
        d1 = (neighboursDistance2[i][1] + biasDistance) >> shiftDistance;
        sum = d1 + d0;
        sumDiv2 = sum >> 1;
        w1 = ((d0 << 8) + 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 << 31) + sumDiv2) / sum;
        biasWeight = 1 << (shift - 1);
        w2 = (d0d1 × r + biasWeight) >> 23;
        w1 = (d0d2 × r + biasWeight) >> 23;
        predictionWeights[i][0] = MaxWeightValue - (w1 + w2);
        predictionWeights[i][1] = w1;
        predictionWeights[i][2] = w2;
    }
}
}
}

```

8.3.2.5 Quantization weights derivation process

The inputs of this process are:

an array of point indexes `indexes[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

a series of nearest neighbours indexes `neighbours[i][j]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `j` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

an array of nearest neighbours counts `neighboursCount[i]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive.

an array of prediction `predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

an array of the number of the decoded points per LoD `pointNumPerLoD[k]`, where `k` is in the range of 0 to `LevelDetailCount - 1`, inclusive.

The output is:

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

The quantization weights derivation procedure proceeds as follows.

If `lifting_scalability_enabled_flag` is equal to 0, the following is applied:

```
for (i = 0; i < PointCount; i++)
    quantizationWeights[i] = 1 << 8

for (i = PointCount - 1; i >= 0; i--) {
    index = indexes[i]
    for (p = 0; p < neighboursCount[index]; p++) {
        neighbour = neighbours[index][p]
        quantizationWeights[neighbour] += divExp2RoundHalfInf(
            predictionWeights[neighbour] × quantizationWeights[neighbour],
            8)
    }
}

for (i = 0; i < PointCount; i++)
    quantizationWeights[i] = iSqrt(quantizationWeights[i])
```

Otherwise (`lifting_scalability_enabled_flag` is equal to 1), the following is applied:

```
index = 0
startIndex = 0
for (lodIndex = 0; lodIndex < lodCount; lodIndex++) {
    for (i = 0; i < pointNumPerLoD[lodIndex]; i++)
        quantizationWeights[index++] =
            ((PointNumInSlice - startIndex)/pointNumPerLoD[lodIndex])) × (1 << 8)
    startIndex += pointNumPerLoD[lodIndex]
}
```

8.3.2.6 Inverse quantization process

Inputs of this process are:

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

The output of the process is

a series of the unquantized attribute coefficients `unquantAttributeCoefficients[i][a]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `a` in the range of 0 to `AttrDim - 1`, inclusive.

The inverse quantization process proceeds as follows.

```
endIndex = pointCountPerLevelOfDetail[0]
for (i = 0, d = 0; i < PointCount; i++) {
    if (i == endIndex) {
        endIndex = pointCountPerLevelOfDetail[++d];
        layerQpY = d < NumLayerQP ? SliceQpY[d] : SliceQpY[NumLayerQP - 1];
        layerQpC = d < NumLayerQP ? SliceQpC[d] : SliceQpC[NumLayerQP - 1];
    }

    regionBoxDeltaQp = 0;
    if (ash_attr_region_qp_delta_present_flag == 1){
        isPointInRegion = 1
        for (k = 0; k < 3; k++)
            isPointInRegion &=
                AttrRegionQpOrigin[k] <= PointPos[i][k]
                && PointPos[i][k] < AttrRegionQpOrigin[k] + AttrRegionQpSize[k]

        if (isPointInRegion)
            regionBoxDeltaQp = RegionboxDeltaQp
    }

    gstepY = QpToQstep(layerQpY + regionBoxDeltaQp, 1);
    gstepC = QpToQstep(layerQpC + regionBoxDeltaQp, 0);
    for (a = 0; a < AttrDim; a++)
```

```

    unquantAttributeCoefficients[i][a] = residual_values[a][i] × (!a ? qstepY : qstepC);
}

```

8.3.2.7 Inverse lifting

Inputs of this process are:

a series of attribute coefficients `attributeCoefficients[i][j]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `j` in the range of 0 to `AttrDim - 1`, inclusive.

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

an array of prediction `predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

The process updates the attributes coefficients `attributeCoefficients`. It proceeds as follows.

```

for (lod = 1; lod < LevelDetailCount; lod++) {
    startIndex = pointCountPerLevelOfDetail[lod - 1];
    endIndex = pointCountPerLevelOfDetail[lod];
    inverseUpdate(startIndex, endIndex, attributeCoefficients, quantizationWeights and
predictionWeights);
    inversePrediction(startIndex, endIndex, attributeCoefficients, and predictionWeights);
}

```

8.3.2.8 Definition of `inverseUpdate()`

Inputs of this process are:

a series of attribute coefficients `attributeCoefficients[i][j]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `j` in the range of 0 to `AttrDim - 1`, inclusive.

an array of prediction `predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount - 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours - 1`, inclusive.

The process updates the attribute coefficients `attributeCoefficients`. It proceeds as follows.

```

for (i = 0; i < startIndex; i++) {
    updateWeights[i] = 0;
    for (j = 0; j < AttrDim; j++)
        updates[i][j] = 0
}

for (i = 0; i < (endIndex - startIndex); i++) {
    index = predictorCount - i - 1 + startIndex;
    currentQuantWeight = quantizationWeights[index];
    for (p = 0; p < neighboursCount[index]; p++) {
        neighbourIndex = neighbours[index][p];
        weight = predictionWeights[index][p] × currentQuantWeight;
        updateWeights[neighbourIndex] += weight;
        for (j = 0; j < AttrDim; j++)
            updates[neighbourIndex][j] += weight × attributeCoefficients[index][j];
    }
}

for (i = 0; i < startIndex; i++) {
    if (updateWeights[i] > 0) {
        bias = updateWeights[i] >> 1;
        for (j = 0; j < AttrDim; j++)
            attributeCoefficients[index][j] -= (updates[i][j] + bias) / updateWeights[i];
    }
}

```

8.3.2.9 Definition of inversePrediction()

Inputs of this process are:

a series of attribute coefficients `attributeCoefficients[i][j]`, where `i` is in the range of 0 to `PointCount – 1`, inclusive, and `j` in the range of 0 to `AttrDim – 1`, inclusive.

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

an array of prediction `predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount – 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours – 1`, inclusive.

an array `pointCountPerLevelOfDetail[l]`, where `l` is in the range of 0 to `LevelDetailCount – 1`, inclusive.

The process updates the attribute coefficients `attributeCoefficients`. It proceeds as follows.

```
pointCount = endIndex - startIndex;
for (i = 0; i < pointCount; i++) {
    index = predictorCount - i - 1 + startIndex;
    for (j = 0; j < AttrDim; j++) {
        predicted = 0;
        for (p = 0; p < neighboursCount[index]; p++) {
            neighbourIndex = neighbours[index][p];
            predicted += predictionWeights[index][p] × attributeCoefficients[neighbourIndex][j];
        }
        attributeCoefficients[neighbourIndex][j] += divExp2RoundHalfInf(predicted, 8);
    }
}
```

8.3.3 LoD with Predicting Transform decoding process

The output of the process is

a series of the decoded attribute values `attributeValues[i][j]`, where `i` is in the range of 0 to `PointCount – 1`, inclusive, and `j` in the range of 0 to `AttrDim – 1`, inclusive.

This process invokes the sub-processes in the following order.

The level of detail generation process in clause 8.3.2.1 is invoked. The output of this process are stored in `indexes[i]`, `neighbours[i][n]`, `neighboursCount[i]`, `neighboursDistance2[i][n]`, and `pointCountPerLevelOfDetail[l]`, where `i` is in the range of 0 to `PointCount – 1`, inclusive, `n` in the range of 0 to `NumPredNearestNeighbours – 1`, inclusive, `l` is in the range of 0 to `LevelDetailCount`, inclusive.

The Prediction weight derivation process in 8.3.2.4 is invoked with the parameters `neighbours`, `neighboursCount` and `neighboursDistance2`. The output of this process is stored in `predictionWeights[i][n]`, where `i` is in the range of 0 to `PointCount – 1`, inclusive, and `n` in the range of 0 to `NumPredNearestNeighbours – 1`, inclusive.

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

The reconstructed attributes values are obtained as follows.

```
q = 0;
for (i = 0; i < PointCount; i++) {
    currentIndex = indexes[i];
    for (j = 0; j < AttrDim; j++) {
```

```

    minPredAttribute[j] = 0;
    maxPredAttribute[j] = 0;
    predicted[j] = 0;
}
for (p = 0; p < neighboursCount[index]; p++) {
    neighbourIndex = neighbours[index][p];
    for (j = 0; j < AttrDim; j++) {
        if (p == 0 || minPredAttribute[j] > attributeValues[neighbourIndex][j])
            minPredAttribute[j] = attributeValues[neighbourIndex][j];
        if (p == 0 || maxPredAttribute[j] < attributeValues[neighbourIndex][j])
            maxPredAttribute[j] = attributeValues[neighbourIndex][j];
    }
}
maxDiff = maxPredAttribute[0] - minPredAttribute[0];
for (j = 1; j < AttrDim; j++)
    maxDiff = Max(maxDiff, maxPredAttribute[j] - minPredAttribute[j]);
if (maxDiff > AdaptivePredictionThreshold)
    predMode = pred_index[i];
else
    predMode = 0;
if (predMode > 0) {
    neighbourIndex = neighbours[index][predMode - 1];
    for (j = 1; j < AttrDim; j++)
        predicted[j] = attributeValues[neighbourIndex][j];
} else {
    for (j = 0; j < AttrDim; j++) {
        for (p = 0; p < neighboursCount[index]; p++) {
            neighbourIndex = neighbours[index][p];
            weight = predictionWeights[index][p];
            predicted[j] += weight × attributeValues[neighbourIndex][j];
        }
        predicted[j] = divExp2RoundHalfInf(predicted[j], 8);
    }
}
for (j = 0; j < AttrDim; j++)
    res[j] = divExp2RoundHalfInf(unquantAttributeCoefficients[currentIndex][j], 8);
for (j = 0; j < AttrDim; j++) {
    attributeValue = predicted[j] + res[j] + (j > 0 ? res[0] : 0);
    if (AttrDim == 0)
        maxAttribute = (1 << (attribute_bitdepth_minus1[ash_attr_sps_attr_idx] + 1)) - 1
    else
        maxAttribute = (1 << (attribute_secondary_bitdepth_minus1[ash_attr_sps_attr_idx] +
1)) - 1
    attributeValues[currentIndex][j] = Clip(attributeValue, 0, maxAttribute);
}
}

```

8.4 Slice concatenation process

The outputs of this process are:

- the modified array RecPic with elements RecPic[pointIdx][attrIdx] representing points in the reconstructed point cloud frame, and
- the modified variable RecPicPointCount representing the number of points in the reconstructed point cloud frame.

RecPicPointCount is initialized to 0.

The points and attributes from the current slice are concatenated with the reconstructed point cloud frame as follows:

```

for (pointIdx = 0; pointIdx <= gsh_num_points_minus1; pointIdx++, RecPicPointCount++) {
    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];
}

```

9 Parsing process

9.1 General

This process is invoked when the descriptor of a syntax element in the syntax tables in 7.3 is equal to $u(n)$, $ue(v)$, $se(v)$, $ae(v)$, or $de(v)$.

The output of this process is a syntax element value.

The array `DataUnitBytes`, with elements `DataUnitBytes[i]`, $i = 0 \dots \text{DataUnitLength} - 1$, represents a coded data unit as a sequence of bytes. When parsing the first syntax element of a data unit, `DataUnitBytes` is set equal to the byte array provided by an encapsulation format (such as Annex B) or by an external means. The function `readDataUnitBit()` provides access to the bitstream as described in 9.2.

When `sps_bypass_stream_enabled_flag` is equal to 1, each data unit represents a header part and one or more sequences of chunk interleaved substreams. Parsing of the geometry slice and attribute slice syntax structures proceeds as follows:

- At the start of parsing the data unit, the variable `entropyStreamIdx` is initialized to 0.
- The variable `ChunkSeqLen` is derived as follows:
 - When parsing the geometry slice syntax, if `entropyStreamIdx` is less than `EntropyStreamCnt - 1`, `ChunkSeqLen` is set equal to `gsh_entropy_stream_len[entropyStreamIdx]`.
 - Otherwise, `ChunkSeqLen` is set equal to `DataUnitLength - (DataUnitReadIdx >> 3)`
- The arrays `AeByteStream` and `BypassBitStream` represent streams of non-bypass arithmetic coded bins and directly coded bypass bins respectively.
- The chunk interleaved substreams parsing process (9.2) is invoked with the input variable `ChunkSeqLen` and the output arrays `AeByteStream` and `BypassBitstream` as follows:
 - At the start of parsing the `geometry_slice_data` syntax structure.
 - At the start of parsing the `geometry_node_syntax` structure when `GeomEntropyStreamCnt` is greater than 1, `nodeIdx` is equal to 0, and the variable `depth` is greater than or equal to `GeomEntropyStreamDepth`.
 - At the start of parsing the `attribute_slice_data` syntax structure.
- `entropyStreamIdx` is incremented by 1.

When `GeomEntropyStreamCnt` is greater than 1, the parsing state may be memorized or restored when starting to parse the `geometry_node_syntax` structure (7.3.3.4) as follows:

- The parsing state memorization process (9.11) is invoked when `nodeIdx` is equal to 0 and `depth` is equal to `GeomEntropyStreamDepth`.
- The parsing state restoration process (9.12) is invoked when `nodeIdx` is equal to 0 and `depth` is greater than or equal to `GeomEntropyStreamDepth`.

The output syntax element value is parsed according to the processes corresponding to the syntax element's descriptor and name in Table 21 and Table 22.

Table 21 — Descriptor passing process

Descriptor	Process	Channel read method
u(n)	9.6.1	readDataUnitBit()
ue(v)	9.6.2	readDataUnitBit()
s(n)	9.6.1, 9.6.4	readDataUnitBit()
se(v)	9.6.2, 9.6.4	readDataUnitBit()
ae(v)	9.10.1	readBin()
de(v)	9.9.1	readBin()

Table 22 — Syntax element specific parsing processes

Syntax structure	Syntax element	Parsing process
geometry_node()	geom_node_qp_offset_eq0_flag	9.6.1 (FL), numBins = 1
	geom_node_qp_offset_sign_flag	9.6.1 (FL), numBins = 1
	geom_node_qp_offset_abs_minus1	9.6.2 (EGk), k = 0
	single_occupancy_flag	9.6.1 (FL), numBins = 1
	occupancy_idx[]	9.6.1 (FL), numBins = 3
	occupancy_map	9.7.5
	occupancy_byte	9.9.1
	num_points_eq1_flag[]	9.6.1 (FL), numBins = 1
	num_points_minus2[]	9.6.2 (EGk), k = 0
	is_planar_flag[][]	9.6.1 (FL), numBins = 1
	plane_position[][]	9.6.1 (FL), numBins = 1
	direct_mode_flag	9.6.1 (FL), numBins = 1
	num_direct_points_gt1	9.6.1 (FL), numBins = 1
	not_duplicated_point_flag	9.6.1 (FL), numBins = 1
	num_direct_points_eq2_flag	9.6.1 (FL), numBins = 1
	num_points_direct_mode_minus3	9.6.2 (EGk), k = 0
	point_offset_s[][] point_offset_t[][] point_offset_v[][]	9.6.1 (FL), numBins = 1
geometry_trisoup_data()	trisoup_sampling_value_minus1	9.4.2 (EGk), k = 0
	num_unique_segments_minus1[]	9.6.2 (EGk), k = 0
	segment_indicator[]	9.6.1 (FL), numBins = 1
	num_vertices_minus1[]	9.6.2 (EGk), k = 0
	vertex_position[]	9.6.3 (TU), maxVal = (1 << trisoup_node_size_log2) + 1
attribute_slice_data()	all_residual_values_equal_to_zero_run	9.6.3 (TU), maxVal = TBD
	pred_index	9.6.3 (TU),

Syntax structure	Syntax element	Parsing process
		maxVal = MaxNumPredictors
attribute_coding()	residual_values_equal_to_zero	9.6.1 (FL), numBins = 1
	residual_values_equal_to_one	9.6.1 (FL), numBins = 1
	remaining_values[][]	9.6.2 (EGk), k = 0
dictionary_encoded_value()	dict_lut0_hit_flag	9.6.1 (FL), numBins = 1
	dict_lut1_hit_flag	9.6.1 (FL), numBins = 1
	dict_lut0_idx	XXXREF
	dict_lut1_idx	9.6.1 (FL), numBins = 4
	dict_direct_value	9.6.1 (FL), numBins = 8

9.2 Chunked bytestream parsing process

9.2.1 General

The input to this process is the variable ChunkSeqLen representing the length in bytes of a sequence of chunks.

The output of this process are:

- The array AeByteStream consisting of bytes of an arithmetic coded data stream.
- The variable AeStreamReadIdx, representing the read position of the AeByteStream.
- The array BypassBitStream consisting of bits of a bypass data stream.
- The variable BypassStreamReadIdx, representing the read position of the BypassBitStream.

A chunked bytestream sequence consists of one or more chunks. With the exception of the last chunk in a sequence, all chunks are 256 bytes in length. The final chunk may be truncated to $\text{ChunkSeqLen} \% 256$ bytes. Each chunk contains data from one or both of the arithmetic coded data stream and a bypass bin data stream.

The variables AeStreamReadIdx and AeBypassStreamReadIdx are both initialized to 0.

The arrays AeByteStream and BypassBitStream are assembled according to the following syntax (9.2.2) and semantics (9.2.3).

9.2.2 Syntax

9.2.2.1 Chunked bytestream sequence syntax

ae_chunk_sequence() {	Descriptor
for(chunkOffset = 0; chunkOffset < ChunkSeqLen; chunkOffset += 256)	
ae_chunk(Min(256, chunkSeqLen – chunkSeqOffset))	
}	

9.2.2.2 Chunked bytestream chunk syntax

<code>ae_chunk(chunkLen) {</code>	Descriptor
chunk_num_ae_bytes	<code>u(8)</code>
<code>for(i = 0; i < num_ae_bytes; i++)</code>	
chunk_ae_byte[i]	<code>u(8)</code>
<code>for(j = 0; j < chunkLen - 1; j++, i++) {</code>	
<code>if(i < chunkLen - 2)</code>	
chunk_bypass_byte[j]	<code>u(8)</code>
<code>else {</code>	
chunk_bypass_5bits	<code>u(5)</code>
chunk_bypass_num_flushed_bits	<code>u(3)</code>
<code>}</code>	
<code>}</code>	

9.2.3 Semantics

9.2.3.1 Chunked bytestream sequence semantics

This clause is intentionally empty.

9.2.3.2 Chunked bytestream chunk semantics

The contents of each chunk is concatenated to the arrays `AeByteStream` and `BypassBitStream`.

chunk_num_ae_bytes indicates the number of `chunk_ae_byte` and `chunk_bypass_byte` elements present in a chunk. When not present, the value of `chunk_num_ae_bytes` is inferred to be 0. It is a requirement of bitstream conformance that `chunk_num_ae_bytes` is less than `chunkLen`.

chunk_ae_byte[i] specifies the *i*-th byte of the arithmetically encoded symbol sub-stream of the current chunk. Each `chunk_ae_byte[i]` is appended to the `AeByteStream` array as follows:

```
for (i = 0; i < chunk_num_ae_bytes; i++)
    AeByteStream[AeStreamLen++] = chunk_ae_byte[i]
```

chunk_bypass_byte[j] specifies the *j*-th byte of the bypass symbol sub-stream of the current chunk. Each `chunk_bypass_byte` is appended to the `BypassBitStream` array as follows:

```
numChunkBypassBytes = Max(0, chunkLen - 2 - chunk_num_ae_bytes)
for (j = 0; j < numChunkBypassBytes; j++)
    for (b = 7; b >= 0; b--)
        BypassBitStream[BypassBitStreamLen++] = (chunk_bypass_byte[j] >> b) & 1
```

chunk_bypass_5bits specifies the values of five bypass bits at the end of the bypass symbol sub-stream of the current chunk. Each bit is appended to the `BypassBitStream` array as follows:

```
for (b = 4; b >= 0; b--)
    BypassBitStream[BypassBitStreamLen++] = (chunk_bypass_5bits >> b) & 1
```

chunk_bypass_num_flushed_bits specifies the number of bypass bits to be discarded from the end of the `BypassBitStream`.

```
BypassBitstreamLen -= chunk_bypass_num_flushed_bits
```

9.3 Definition of readDataUnitBit

The inputs to this process are the current data unit byte array DataUnitBytes and the associated read position DataUnitReadIdx.

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

On the first invocation of this process for the current data unit, the variable DataUnitReadIdx is set equal to 0.

The output value bitVal is determined as follows:

```
byteIdx = DataUnitReadIdx >> 3
bitMask = 0x80 >> (DataUnitReadIdx & 7)
bitVal = DataUnitBytes[byteIdx] & bitMask != 0
```

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

9.4 Definition of readAeStreamBit

If sps_bypass_stream_enabled_flag is equal to 0, this process is equivalent to invoking readDataUnitBit (9.3).

Otherwise, sps_bypass_stream_enabled_flag equal to 1, the outputs of this process are the next bit read from the AeByteStream array, and the updated AeByteStream read position.

The output value bitVal is determined as follows:

```
byteIdx = AeStreamReadIdx >> 3
bitMask = 0x80 >> (AeStreamReadIdx & 7)
bitVal = AeByteStream[byteIdx] & bitMask != 0
```

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

9.5 Definition of readBypassStreamBit

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

The output value bitVal is determined as follows:

```
bitVal = BypassBitStream[BypassBitStreamReadIdx]
```

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

9.6 General inverse binarisation processes

9.6.1 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()
```

9.6.2 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 $k + \text{prefix}$ 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
```

9.6.3 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++
```

9.6.4 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 output is set equal to `unsignedVal >> 1`,

Otherwise, if `unsignedVal` is odd, the output is set equal to `(unsignedVal + 1) >> 1`.

Table 23 illustrates an example of the mapping process.

Table 23 — Conversion of unsigned values for signed syntax elements (informative)

Unsigned value	Signed value
0	0
1	−1
2	1
3	−2
4	2
5	−3
6	3
...	...

9.7 Bit-wise geometry octree occupancy parsing process

9.7.1 General process

The parsing and inverse binarization of the arithmetically coded syntax element `occupancy_map` is described in 9.7.5

The decoding of each arithmetically encoded bin in `occupancy_map` involves a context selection process that makes use of a dynamic map (the array `CtxMap`) to select a context (9.7.7) based upon the occupancy state of neighbouring nodes, predicted occupancy values ((9.7.9) and previously decoded bins. After decoding a bin, `CtxMap` is updated based upon the decoded bin value (9.7.8).

At the start of decoding a geometrydata unit, `CtxMap` is initialized according to 9.7.2.

NOTE — While the described process updates `CtxMap` after decoding each bin, there is no dependency by subsequent bins on the updated value.

9.7.2 Initialisation process

This process is invoked at the start of each geometry data unit.

The output from this process is the initialized array `CtxMap` with entries `CtxMap[i]` for i in the range 0 to 1499×3 set equal to 127.

9.7.3 Determination of planar masks used in the inverse binarization process

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 $i = 0 \dots 7$, 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 $i = 0 \dots 7$, 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.

9.7.4 Occupancy_idx[] parsing process

When `occupancy_idx[axisIdx]`, for `axisIdx` in the range $0 \dots 2$, 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))
```

9.7.5 Inverse binarization process

This process reconstructs a value of the syntax element `occupancy_map`. The input to this process is the variables `NeighbourPattern` and the planar information `mask_planar[]` and `mask_planar_fixed0[]` associated with the current node.

The output from this process is the syntax element value, constructed as follows:

```
value = 0
min_non_zero_node = NeighbourPattern == 0 ? 2 : 1
for (axisIdx = 0; axisIdx <= 2; axisIdx++)
    min_non_zero_plane[axisIdx] = NeighbourPattern == 0 && mask_planar[axisIdx] ? 2 : 1

initialize_counters_for_zeros()
for (BinIdx = 0; BinIdx < 8; BinIdx++) {
    binIsInferred0 =
        ((mask_planar[0] >> bitCodingOrder[BinIdx]) & 1)
        || ((mask_planar[1] >> bitCodingOrder[BinIdx]) & 1)
        || ((mask_planar[2] >> bitCodingOrder[BinIdx]) & 1)

    if (binIsInferred0) {
        bin = 0
        continue
    }

    determine_binIsInferred1()
    if (binIsInferred1)
        bin = 1
    else {
        bin = readOccBin()
        if (!bin)
            update_counters_for_zeros()
    }
    value = value | (bin << bitCodingOrder[BinIdx])
}
```

where `bitCodingOrder[BinIdx]` is defined by Table 24, and `readOccBin()` is specified by 9.7.6,

Table 24 — Values of bitCodingOrder[i]

i	0	1	2	3	4	5	6	7
value	1	7	5	3	2	6	4	0

The variable `binIsInferred0` is set equal to 1 when the value of the bin can be deduced to be 0 from the planar information associated with the node, e.g. when the bin corresponds to the occupancy bit of a child node that belongs to a plane known to be unoccupied from the planar information. Otherwise, `binIsInferred0` is set equal to 0.

If `binIsInferred0` equal 0, the variable `binIsInferred1` is set equal to 1 when the value of the bin can be deduced to be 1 from the planar information, the minimum number `min_non_zero_node` of non-zero bins in the node, and the minimum number `min_non_zero_plane[axisIdx]` of non-zero bins in the occupied plane along the `axisIdx`-th axis (would the node be palnar along this axis). Otherwise, `binIsInferred1` is set equal to 0.

The value of `binIsInferred1` is determined based on counters `coded0[axisIdx][planePos]` that counts the number of occupancy bits already known to be zero for a plane position `planePos` (either equal to 0 for the lower plane, or equal to 1 for the upper plane) along the `axisIdx`-th axis. The counters are initialized at the start of the inverse binarization process as follows

```
initialize_counters_for_zeros() {
    for (axisIdx = 0; axisIdx <= 2; axisIdx++)
        for (planePos = 0; planePos <= 1; planePos++)
            coded0[axisIdx][planePos] = 0
    for (i = 0; i < 8; i++) {
        if ((mask_planar[0] >> i) & 1 || ((mask_planar[1] >> i) & 1 || ((mask_planar[2] >> i)
& 1) {
            coded0[0][(mask_planar_fixed0[0] >> i) & 1]++
            coded0[1][(mask_planar_fixed0[1] >> i) & 1]++
            coded0[2][(mask_planar_fixed0[2] >> i) & 1]++
        }
    }
}
```

Thus, the counters `coded0[][]` are initialized counting the number of occupancy bits known to be zero from the planar information. Each time a bin is decoded by `readOccBin()` and this decoded bin is equal to 0, the counters `coded0[][]` are updated by

```
update_counters_for_zeros() {
    coded0[0][(mask_planar_fixed0[0] >> bitCodingOrder[BinIdx]) & 1]++
    coded0[1][(mask_planar_fixed0[1] >> bitCodingOrder[BinIdx]) & 1]++
    coded0[2][(mask_planar_fixed0[2] >> bitCodingOrder[BinIdx]) & 1]++
}
```

When `binIsInferred0` equal 0, the determination of the value of `binIsInferred1` performed as follows

```
determine_binIsInferred1() {
    for (axisIdx = 0; axisIdx <= 2; axisIdx++) {
        mask0 = mask_planar_fixed0[axisIdx] >> bitCodingOrder[BinIdx]) & 1
        binIsOne[axisIdx] =
            (eligible_planar_flag[axisIdx]
            && coded0[axisIdx][mask0] >= 4 - min_non_zero_plane[axisIdx])
            || coded0[axisIdx][0] + coded0[axisIdx][1] >= 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 $4 - \text{min_non_zero_plane}[\text{axisIdx}]$

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.

9.7.6 Definition of readOccBin()

The inputs to this process are the variables BinIdx, and PartialSynVal.

The output from this process is the value of the decoded bin.

The process for a decoding a single bin is as follows:

The variables ctxMapIdx and ctxIdx are determined according to the derivation process 9.7.7 with the variables NeighbourPattern, BinIdx, and PartialSynVal as input.

The arithmetic decoding process 9.10.2 for a single bin is invoked for the syntax element occupancy_map with the variable ctxIdx as input. The output binVal is the value of the decoded bin.

The context map update process 9.7.8 is invoked with the variable ctxMapIdx and the decoded bin value.

9.7.7 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 and ctxIdx.

The variable idxPred is set as follows:

If NodeMaxDimLog2 is greater than or equal to log2_intra_pred_max_node_size, the variable idxPred is set equal to 0.

Otherwise, NodeMaxDimLog2 is less than log2_intra_pred_max_node_size, the variable idxPred is set equal to the output of the occupancy prediction process using neighbouring octree nodes (9.7.9) when invoked with the current node and childIdx set equal to the output of the neighbour dependent geometry octree child node scan order Inverse mapping process (6.4.1) with the inputs neighbourPattern and inIdx set equal to bitCodingOrder[binIdx] where values of bitCodingOrder[] are given in Table 24.

The variable idxAdj is set as follows:

If adjacent_child_contextualization_enabled_flag is equal to 1, the following applies:

The variables adjOcc and adjUnocc are initialized to 0.

The variables sC, tC, and vC identifying the position of the child node associated with binIdx at depth + 1 are initialized as follows

```
sC = 2 × sN + ((bitCodingOrder[binIdx] >> 2) & 1)
tC = 2 × tN + ((bitCodingOrder[binIdx] >> 1) & 1)
vC = 2 × vN + (bitCodingOrder[binIdx] & 1)
```

The following procedure is performed for each of the s, t, and v axes by substituting the variables aN, aC, nPmas, sCn, tCn, vCn, sNn, tNn, and vNn of the corresponding row of Table 25.

```
// if child is adjacent to a causally-valid neighbour
if (!(aC & 1)) {
  if (NeighbourPattern & nPmask)
    adjOcc += GeometryNodeOccupancyCnt[depth + 1][sCn][tCn][vCn]
  else
    // if neighbour is available but not present
    if ((aN + 1) & NeighbAvailabilityMask != 1)
      if (GeometryNodeOccupancyCnt[depth][sNn][tNn][vNn] == 0)
        adjUnocc = 1
}
```

Table 25 — Variable substitutions for the computation of adjOcc and adjUnocc

axis	aN	aC	nPmask	sCn	tCn	vCn	sNn	tNn	vNn
s	sN	sC	2	sC-1	tC	vC	sN-1	tN	vN
t	tN	tC	4	sC	tC-1	vC	sN	tN-1	vN
v	vN	vC	16	sC	tC	vC-1	sN	tN	vN-1

The variable idxAdj is derived as follows

```
idxAdj = adjUnocc + 2 × Min(2, adjOcc)
if (binIdx > 4)
  idxAdj = ctxIdxAdjReduc567[idxAdj]
```

Table 26 — Values of ctxIdxAdjReduc567[i]

i	0	1	2	3	4	5
ctxIdxAdjReduc567[i]	0	0	1	2	3	3

The variable ctxIdxMapIdx is set equal to $3 \times \text{idxAdj} + \text{idxPred}$.

The output variable ctxMapIdx is derived as follows:

If NeighbourPattern is equal to 0, ctxIdxMapOffset is set equal to popcnt(partialSynVal).

Otherwise, NeighbourPattern is not equal to 0, the following applies:

```
if (neighbour_context_restriction_flag)
  pattern = neighbourPattern64to9[NeighbourPattern];
else
  pattern = neighbourPattern64to6[NeighbourPattern];

if (binIdx == 7)
  pattern = 1;
else if (binIdx == 6)
  pattern = neighbourPattern9to3[pattern];
else if (binIdx > 3)
  pattern = neighbourPattern9to5[pattern];
ctxIdxMapOffset = ((pattern - 1) << binIdx) + partialSynVal + binIdx + 1;
```

Finally, the output variable ctxIdx is set as follows

```
ctxMapIdx = ctxIdxMapIdx × 1499 + ctxIdxMapOffset
ctxIdx = CtxMap[ctxMapIdx] >> 3
```

Table 27 — Values of neighbourPattern64to9[j + i]

j	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	1	1	2	2	3	1	2	2	3	1	3	3	4
16	1	2	2	3	2	5	5	6	2	5	5	6	3	6	6	7
32	1	2	2	3	2	5	5	6	2	5	5	6	3	6	6	7
48	1	3	3	4	3	6	6	7	3	6	6	7	4	7	7	8

Table 28 — Values of neighbourPattern64to6[j + i]

j	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	5	5	na	5	1	1	na	5	1	1	na	na	na	na	na
16	2	3	3	na	3	7	7	na	3	7	7	na	na	na	na	na
32	2	3	3	na	3	7	7	na	3	7	7	na	na	na	na	na
48	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na

Table 29 — Specification of neighbourPattern9to5[i]

i	0	1	2	3	4	5	6	7	8	
neighbourPattern9to5[i]	0	1	2	3	11	22	3	4	44	

Table 30 — Specification of neighbourPattern9to3[i]

i	0	1	2	3	4	5	6	7	8
neighbourPattern9to3[i]	0	1	11	22	22	11	22	2	2

9.7.8 Context map update process

This process updates the context mapping table for the syntax element occupancy_map.

Input to this process are the variable ctxMapIdx and a decoded bin value.

The context mapping CtxMap[ctxMapIdx] is updated as follows:

```
stateVal = CtxMap[ctxMapIdx]
if (binVal)
  CtxMap[ctxMapIdx] += ctxMapTransition[(255 - stateVal) >> 4]
else
  CtxMap[ctxMapIdx] -= ctxMapTransition[stateVal >> 4]
```

Where values of ctxMapTransition are given by Table 31.

Table 31 — Values of ctxMapTransition[i]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	0	1	1	2	4	7	9	11	14	16	19	23	22	18	13	6

9.7.9 Occupancy prediction process using neighbouring octree nodes

The occupancy prediction process generates a tri-state occupancy prediction of a single child node based on the occupancy state of nodes neighbouring the parent node.

Input to this process are

the variables sN, tN, vN, and depth, identifying a node in the geometry octree, and

the variable childIdx identifying a child node position according to the geometry octree child traversal order for occupancy prediction.

Output from this process is the predicted occupancy state for the specified child node.

A list of neighbouring occupied blocks is determined as follows:

```
for (i = 0; i < 25; i++) {
    s = sN + dS[i]
    t = tN + dT[i]
    v = vN + dV[i]
    if (available(sN, tN, vN, s, t, v))
        occupied[i] = GeometryNodeOccupancyCnt[depth][s][t][v] != 0
    else
        occupied[i] = 0
}
```

Where the function available(sN, tN, vN, s, t, v) evaluates to true if all of the following conditions are true:

```
log2_neighbour_avail_boundary > 0
(s ^ sN) >> log2_neighbour_avail_boundary == 0
(t ^ tN) >> log2_neighbour_avail_boundary == 0
(v ^ vN) >> log2_neighbour_avail_boundary == 0
```

And where the values of the neighbour position offsets dS[], dT[], and dV[] are given in Table 32.

If the sum of occupied[i], with i = 0 .. 25, is less than 8, the output predicted occupancy state is set equal to zero and no further processing occurs.

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

$$score = \sum_{i=0}^{25} predictionScore[scoreIdx[childIdx][i]][occupied[i]]$$

Where the values of scoreIdx[][], and predictionScore[][] are given by Table 32 and Table 33.

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

$$numOccupied = \sum_{i=0}^{25} occupied[i]$$

```
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 34.

Table 32 — Values of dS[i], dT[i], dV[i], and scoreIdx[bitIdx][i] for intra occupancy prediction

i	dS[i]	dT[i]	dV[i]	scoreIdx[childIdx][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 33 — Values of predictionScore[i][occupied]

occupied	i							
	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 34 — Values of predictionThreshold[i][j]

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

9.8 Inferred Direct Coding Mode parsing process

9.8.1 General process

The parsing and inverse binarization of the arithmetically coded syntax element plane_position[][2] is described in 9.8.3.

9.8.2 Determination of the angular context idcmIdxAngular

The process to determine the context idcmIdxAngular[i][j] for coding the bin point_offset_v[i][j] associated with j-th bit of the i-th point belong to a the child node that undergoes Inferred Direct Coding Mode is described in this section.

This process is performed after point_offset_s[i][] and point_offset_t[i][] are decoded such that PointOffsetS[i] and PointOffsetT[i] are known. The s and t position, relative to the Lidar, of the point i is derived by

```
posSLidar[ i ] = sNchild - geomAngularOrigin[ 0 ] + PointOffsetS[ i ]
posTLidar[ i ] = tNchild - geomAngularOrigin[ 1 ] + PointOffsetT[ i ]
```

where (sNchild, tNchild, vNchild) specifying the position of the geometry octree child node Child in the current slice.

The inverse rInv of the radial distance of the point from the Lidar is determined by

```
sLidar = (posSLidar[ i ] << 8) - 128
tLidar = (posTLidar[ i ] << 8) - 128
r2 = sLidar × sLidar + tLidar × tLidar
rInv = invSqrt(r2)
```

The corrected laser angle ThetaLaser of the laser associated with the child nodeChild is deduced by

```
Hr = laser_correction[laserIndex[Child]] × rInv
ThetaLaser = laser_angle[laserIndex[Child]] + (Hr >= 0 ? -(Hr >> 17) : ((-Hr) >> 17))
```

Assuming that the bits point_offset_v[i][j2] for j2 = 0 .. j - 1, are known, the point is known to belong to a virtual vertical interval whose half size is provided by

```
halfIntervalSize[ j ] = (1 << (EffectiveChildNodeSizeVLog2 - 1)) >> j
```

and a partial v point position posVLidarPartial[i][j], that provides the lower end of the interval, is deduced by

```
PointOffsetVpartial = 0;
for (j2 = 0; j2 < j; j2++)
    PointOffsetVpartial[i] += point_offset_v[i][j2] << j2
PointOffsetVpartial[i] <= (EffectiveChildNodeSizeVLog2 - j)
    posVlidarPartial[i][j] = vNchild - geomAngularOrigin[2] + PointOffsetVpartial[i]
```

A relative laser position $\theta_{\text{LaserDeltaVirtualInterval}}$ relative to the middle of the virtual interval is computed by

```
vLidar = ((posVlidarPartial[i][j] + halfIntervalSize[j]) << 1) - 1
theta = zLidar × rInv
theta32 = theta >= 0 ? theta >> 15 : -((-theta) >> 15)
thetaLaserDeltaVirtualInterval = ThetaLaser - theta32;
```

Two absolute angular differences m and M of the laser relative to a lower and an upper v position in the virtual interval are determined.

```
vShift = ((rInv << EffectiveChildNodeSizeVLog2) >> 17) >> j
m = Abs(thetaLaserDeltaVirtualInterval - vShift);
M = Abs(thetaLaserDeltaVirtualInterval + vShift);
```

Then, the angular context is deduced from the two absolute angular differences.

```
idcmIdxAngular[i][j] = m > M
diff = Abs(m - M)
if (diff >= rInv >> 15) idcmIdxAngular[i][j] += 2
if (diff >= rInv >> 14) idcmIdxAngular[i][j] += 2
if (diff >= rInv >> 13) idcmIdxAngular[i][j] += 2
if (diff >= rInv >> 12) idcmIdxAngular[i][j] += 2
```

9.8.3 Inverse binarization process

When Inferred Direct Coding Mode is applied to a child node Child , the bits $\text{point_offset_v}[i][j]$ of the i -th point in the child node, for j in the range $0 \dots \text{EffectiveChildNodeSizeVLog2}$ or in the range $1 \dots \text{EffectiveChildNodeSizeVLog2}$ in case the first bit is inferred by the plane position $\text{plane_position}[\text{Child}][2]$, are decoded applying the following process.

If $\text{geometry_angular_mode_flag}$ is equal to 0, then the bit $\text{point_offset_v}[i][j]$ is decoded using the bypass decoding process.

Otherwise, if $\text{geometry_angular_mode_flag}$ is equal to 0, the bit $\text{point_offset_v}[i][0]$ is bypass decoded when not inferred by the plane position, and the bits $\text{point_offset_v}[i][j]$ are decoded using the context $\text{idcmIdxAngular}[i][j]$ for $j > 0$.

9.9 Dictionary-based parsing

9.9.1 General process

This process is invoked when parsing syntax elements with descriptor $\text{ae}(v)$.

This process involves:

- An array of values $\text{lut0}[k]$ storing the most frequent symbols, where k is in the range of 0 to 31, inclusive.
- An array of values $\text{lut0Histogram}[k]$ storing the symbols occurrences, where k is in the range of 0 to 255, inclusive.

- Two variables `lut0UpdatePeriod` and `lut0SymbolsUntilUpdate` storing the update period for `lut0` and the number of symbols remaining until the next update, respectively.
- A variable `lut0Reset` specifying whether `lut0` should be reset during the next `lut0` update or not.
- An array of values `lut1[k]` storing the last 16 decoded symbols, where `k` is in the range of 0 to 15, inclusive.
- A variable `lut1IndexLastSymbol` storing the index of the last decoded symbol.
- A static binary arithmetic context `ctxStatic`.
- A set of adaptive binary arithmetic contexts `ctxLut0Hit`, `ctxLut1Hit`, and `ctxSymbolBit`.
- An array of adaptive binary arithmetic contexts `ctxLut0Index` of size 5 if `limitedContextMode` equals 1, and 31 otherwise (i.e., `limitedContextMode` equals 0).

Inputs to this process are

a variable `limitedContextMode` specifying whether a limited number of contexts is used or not.

a variable `lut0MaxOccurrence` specifying the maximum allowed occurrence value in `lut0Histogram[k]`.

two variables `lut0InitialUpdatePeriod` and `lut0MaxUpdatePeriod` specifying the initial update period and the maximum update period for the for `lut0`, respectively.

an array of values `lut0Initilization[k]` specifying the initial `lut0` values, where `k` is in the range of 0 to 31, inclusive.

`lut0` is initialized by invoking the initialization process in clause 9.9.2 with the parameters `limitedContextMode` and `lut0Initilization`.

`lut0UpdatePeriod`, `lut0SymbolsUntilUpdate` and `lut0Reset` are initialized as follows:

`lut0UpdatePeriod = lut0InitialUpdatePeriod`

`lut0SymbolsUntilUpdate = lut0InitialUpdatePeriod`

`lut0Reset = 0`

`lut1` is initialized by invoking the initialization process in clause 9.9.3.

All the binary arithmetic contexts are initialized by invoking the process in clause 9.10.4.2.

Output from this process is an 8-bit syntax element value, constructed as follows.

```
lut0_hit_flag = readBin(ctxLut0Hit);
if (lut0_hit_flag) {
    index = decodeLut0Index(limitedContextMode, ctxLut0Index);
    value = lut0[index];
    pushLut0(value);
} else {
    lut1_hit_flag = readBin(ctxLut1Hit);
    if (lut1_hit_flag) {
        index = 0;
        for (i = 0; i < 4; i++)
            index |= readBin(ctxStatic) << i;
        value = lut1[index];
    } else {
```

```

    value = 0;
    for (i = 0; i < 8; i++)
        value |= readBin(ctxSymbolBit) << i;
    }
    pushLut1(value);
    pushLut0(value);
}

```

9.9.2 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;

```

9.9.3 Initializing lut1

`lut1` is initialized according to the following process.

```

for (k = 0; k < 16; k++)
    lut1[k] = k;

```

9.9.4 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]);
}

```

9.9.5 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();

```

9.9.6 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;
}

```

9.9.7 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.

9.9.8 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]);

```

9.10 CABAC parsing process

9.10.1 General

This process is invoked when parsing syntax elements with descriptor `ae(v)`.

The input to this process is a request for the value of a syntax element.

The output of this process is the value of the syntax element.

The initialization processes 9.10.3.2 and 9.10.4.2 are invoked when starting to parse of any of the following syntax structures:

- `geom_slice_data` (7.3.3.3)
- `attribute_slice_data` (7.3.4.3)

The parsing of the syntax element proceeds according to the corresponding process listed in Table 22.

9.10.2 Definition of `readBin()`

The inputs to this process are the variable `binIdx` and an associated syntax element.

The outputs of this process is the value of the decoded bin and an updated context variable.

The values `ctxTbl` and `ctxIdx` are determined according to the entries for the associated syntax element in Table 35.

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

- The arithmetic decoding process 9.10.4.3 for a single bin is invoked to determine the value of the decoded bin with the context variable `Contexts[ctxTbl][ctxIdx]` as input.
- The context map update process 9.10.3.3 is invoked with the context variable `Contexts[ctxTbl][ctxIdx]` and the decoded bin value.

Otherwise, the value of `ctxIdx` is equal be the value 'bypass', the following applies:

- If `sps_bypass_stream_enabled_flag` is equal to 0, the arithmetic decoding process 9.10.4.4 for a single bypass bin is invoked to determine the value of the decoded bin. Otherwise, `sps_bypass_stream_enabled_flag` is equal to 1, the `readBypassStreamBit` process 9.4 is invoked to determine the value of the decoded bin.

Table 35 — Values of `ctxTbl` and `ctxIdx` for binarized `ae(v)` coded syntax elements

Syntax element	<code>ctxTbl</code>	<code>ctxIdx</code>
<code>geom_node_qp_offset_eq0_flag</code>	28	0
<code>geom_node_qp_offset_sign_flag</code>	29	0
<code>geom_node_qp_offset_abs_minus1</code>	30	prefix: 0 sufix: bypass
<code>single_occupancy_flag</code>	0	0
<code>occupancy_idx[]</code>	na	bypass
<code>occupancy_map</code>	1	0 .. 31 (9.7.7)
<code>num_points_eq1_flag[]</code>	2	0

Syntax element	ctxTbl	ctxIdx
num_points_minus2[]	3	prefix:0 suffix: bypass
is_planar_flag[][]	31	0 .. 11: planarIdx (8.2.4.3)
plane_position[][0]	32	0 .. 36: planePosIdx (8.2.4.4)
plane_position[][1]	33	0 .. 36: planePosIdx (8.2.4.4)
plane_position[][2]	34	0 .. 46: planePosIdxAngular (8.2.4.5)
direct_mode_flag	4	0
num_direct_points_gt1	35	0
not_duplicated_point_flag	2	0
num_direct_points_eq2_flag	36	0
num_points_direct_mode_minus3	3	prefix:0 suffix: bypass
point_offset_s[][] point_offset_t[][]	na	bypass
point_offset_v[][]	na 37	bypass or 0 .. 10: idcmIdxAngular (9.8.3)
trisoup_sampling_value_minus1	na	bypass
num_unique_segments_minus1[]	na	prefix: bypass suffix: bypass
segment_indicator[]	6	BinIdx
num_vertices_minus1[]	na	prefix: bypass suffix: bypass
vertex_position[]	7	BinIdx
all_residual_values_equal_to_zero_run	8	0 .. 2
pred_index	9	Min(BinIdx, 1)
residual_values_equal_to_zero	10	$\sum_{i=0}^{k-1} 2^i (1 + residual_values_equal_to_zero[i])$
residual_values_equal_to_one	11	$\sum_{i=0}^{k-1} 2^i (1 + (residual_values_equal_to_zero[i] residual_values_equal_to_one[i]))$
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 36 — Values of ctxTblD[n] for de(v) coded syntax elements

Syntax element	n				
	0	1	2	3	4
occupancy_byte	13	14	15	16	17
values[][], k2 = 0	18	19	20	21	22

values[][], k2 = 1	23	24	25	26	27
----------------------	----	----	----	----	----

9.10.3 Context variables

9.10.3.1 General

A context variable is a 16-bit unsigned integer value that models the probability of a zero bin.

NOTE — The values 0, 0x8000, and 0x10000 represent the probability of a zero bin as impossible, equi-probable, and certain respectively. The values 0 and 0x10000 can never be attained due to the operation of the context update process.

Adaptive contexts are updated after decoding each bin, according to a probability look-up table. The update table supplies a value for incrementing or decrementing the probability of a zero bin based upon the upper eight bits of the context's current value.

The array Contexts, with values Contexts[ctxTbl][ctxIdx], represents individual context variables used by the CABAC parsing process. The values of ctxIdx for each value of ctxTbl are specified in Table .

9.10.3.2 Initialisation of context variables

The outputs of this process are initialized CABAC state variables.

All context variables of the arithmetic decoding engine are initialized to the value 0x8000.

9.10.3.3 Context variable update process

The inputs to this process are the variable binVal representing the value of a decoded bin, and a context variable ctx.

The output of this process is the updated context variable.

The context variable is updated as follows:

```
if (binVal)
    ctx -= CtxUpdateDelta[ctx >> 8];
else
    ctx += CtxUpdateDelta[255 - (ctx >> 8)];
```

where values of CtxUpdateDelta[] are given in Table 37.

Table 37 — Values of CtxUpdateDelta[i + j]

j	i											
	0	1	2	3	4	5	6	7	8	9	10	11
0	0	2	5	8	11	15	20	24	29	35	41	47
12	53	60	67	74	82	89	97	106	114	123	132	141
24	150	160	170	180	190	201	211	222	233	244	256	267
36	279	291	303	315	327	340	353	366	379	392	405	419
48	433	447	461	475	489	504	518	533	548	563	578	593
60	609	624	640	656	672	688	705	721	738	754	771	788
72	805	822	840	857	875	892	910	928	946	964	983	1001
84	1020	1038	1057	1076	1095	1114	1133	1153	1172	1192	1211	1231
96	1251	1271	1291	1311	1332	1352	1373	1393	1414	1435	1456	1477

j	i											
	0	1	2	3	4	5	6	7	8	9	10	11
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								

9.10.4 Arithmetic decoding engine

9.10.4.1 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.

9.10.4.2 Initialisation process

The outputs of this process are the initialized arithmetic decoding engine variables ivlLow, ivlRange, and ivlCode.

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

```

ivlLow = 0;
ivlRange = 0xffff;
ivlCode = 0;
for (i = 0; i < 15; i++) {
    ivlCode <= 1;
    ivlCode += readAeStreamBit();
}

```

9.10.4.3 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;
}
```

9.10.4.4 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;
}
```

9.10.4.5 Arithmetic decoder state renormalisation process

Renormalisation stops the arithmetic decoding engine from losing accuracy. Renormalisation shall be applied while the range is less than or equal to a quarter of the total available 16-bit range (0x4000). Each renormalisation doubles the interval and reads a bit into the codeword.

The inputs to this process are the state variables ivlLow, ivlRange, and ivlCode.

The outputs of this process are the updated state variables ivlLow, ivlRange, and ivlCode.

While ivlRange is less than or equal to 0x4000, the following applies:

```
if ((ivlLow + ivlRange - 1) ^ ivlLow >= 0x8000) {
    ivlCode ^= 0x4000;
    ivlLow ^= 0x4000;
}
ivlRange <= 1;
ivlLow = (ivlLow << 1) & 0xffff;
ivlCode = ((ivlCode << 1) | readAeStreamBit()) & 0xffff;
```

9.10.5 Arithmetic encoding engine (informative)

9.10.5.1 General (informative)

This clause does not form an integral part of this Specification.

The inputs to this process are binary symbols that are to be encoded.

The outputs of this process are bits that are written to the data unit bytestream.

This informative clause describes an arithmetic encoding engine that matches the arithmetic decoding engine described in 9.10.4. The encoding engine is essentially symmetric with the decoding engine, i.e., procedures are called in the same order. Table 38 illustrates the correspondence between decoding and encoding processes.

Table 38 — Correspondence between decoder and encoder arithmetic coding processes

Process	Decoder	Encoder
Initialisation	9.10.4.2	9.10.5.2
Symbol coding	9.10.4.3	9.10.5.3
Renormalisation	9.10.4.5	9.10.5.4
Termination	—	9.10.5.5

The state of the arithmetic encoding engine is represented by the variables `ivlLow` indicating the bottom of the encoding interval, `ivlRange` indicating the width of the encoding interval, and `ivlCarry` tracking the number of unresolved straddle conditions during renormalisation.

9.10.5.2 Initialization process (informative)

This clause does not form an integral part of this Specification.

This process is invoked before encoding the first `ae(v)` coded syntax element of a data unit.

The outputs of this process are the arithmetic encoding engine variables `ivlLow`, `ivlRange`, and `ivlCarry`, initialized as follows:

```
ivlLow = 0;
ivlRange = 0xFFFF;
ivlCarry = 0;
```

With 16 bit accuracy, 0xFFFF corresponds to an interval width value of (almost) 1.

9.10.5.3 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;
}
```

9.10.5.4 Arithmetic encoder state renormalisation process (informative)

This clause does not form an integral part of this Specification.

The inputs to this process are the variables `ivlLow`, `ivlRange`.

The outputs of this process are zero or more bits written to the data unit bitstream and the updated variables `ivlLow`, `ivlRange`.

Renormalisation must cause `ivlLow` and `ivlRange` to be modified exactly as in the decoder. In addition, during renormalisation bits are output when `ivlLow` and `ivlLow + ivlRange` agree in their most significant bits, taking into account carries accumulated when a straddle condition is detected.

While `ivlRange` is less than or equal to `0x4000`, the following applies:

```
if ((ivlLow + ivlRange - 1) ^ ivlLow >= 0x8000) {
    ivlLow ^= 0x4000;
    ivlCarry++;
} else {
    writeBit((ivlLow >> 15) & 1);
    for (; ivlCarry > 0; ivlCarry--)
        writeBit((~ivlLow >> 15) & 1);
}
ivlRange <= 1;
ivlLow <= 1;
ivlLow &= 0xFFFF;
```

9.10.5.5 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();
```

9.11 Parsing state memorization process

This process records the elements and values of the following arrays and variables for restoration by the parsing state restoration process (9.12):

- The array Contexts from the CABAC parsing process (9.10)
- The array CtxMap from the bit-wise geometry octree occupancy parsing process (9.7)
- The arrays and variables lut0, lut0Histogram, lut0UpdatePeriod, lut0SymbolsUntilUpdate, lut0Reset, lut1, lut1IndexLastSymbol from the dictionary-based parsing process (9.9)
- The array planeRate and variable localDensity from the planar coding mode (8.2.4)

9.12 Parsing state restoration process

This process restores the elements and values of the following arrays and variables to those previously recorded by the parsing state memorization process (9.11):

- The array Contexts from the CABAC parsing process (9.10)
- The array CtxMap from the bit-wise geometry octree occupancy parsing process (9.7)
- The arrays and variables lut0, lut0Histogram, lut0UpdatePeriod, lut0SymbolsUntilUpdate, lut0Reset, lut1, lut1IndexLastSymbol from the dictionary-based parsing process (9.9)
- The array planeRate and variable localDensity from the planar coding mode (8.2.4)

Annex A

Profiles and levels

A.1 Overview of profiles and levels

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

NOTE 1 – This Specification does not include individually selectable “options” at the decoder, as this would increase interoperability difficulties.

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

NOTE 2 – Encoders are not required to make use of any particular subset of features supported in a profile.

Each level specifies a set of limits on the values that may be taken by the syntax elements of this Specification. The level definition is used with all profiles. For any given profile, a level generally corresponds to a particular decoder processing load and memory capability.

The profiles that are specified in clause A.3 are also referred to as the profiles specified in Annex A.

A.2 Requirements on decoder capability

Capabilities of decoders conforming to this Specification are specified in terms of the ability to decode bitstreams conforming to the constraints of profiles and levels specified in this annex. When expressing the capabilities of a decoder for a specified profile, the level supported for that profile should also be expressed.

Specific values are specified in this annex for the syntax elements `main_profile_compatibility_flag` and `level_idc`. All other values of `main_profile_compatibility_flag` and `level_idc` are reserved for future use by ISO/IEC.

NOTE – Decoders should infer that a reserved value of `level_idc` between the values specified in this Specification indicates intermediate capabilities between the specified levels.

A.3 Profiles

A.3.1 General

All constraints for SPSs, GPSs, and APSs that are specified are constraints for the parameter sets that are activated when the bitstream is decoded.

A.3.2 Main profile

Bitstreams conforming to the Main profile shall obey the following constraints:

- Active SPSs shall have `main_profile_compatibility_flag` equal to 1 only.
- The level constraints specified for the Main profile in clause A.4 shall be fulfilled.

Conformance of a bitstream to the Main profile is indicated by `main_profile_compatibility_flag` being equal to 1.

Decoders conforming to the Main profile at a specific level (identified by a specific value of `level_idc`) shall be capable of decoding all bitstreams for which all of the following conditions apply:

- The bitstream representation is indicated to conform to the Main profile.
- The bitstream representation is indicated to conform to a level that is lower than or equal to the specified level.

A.4 Levels

A.4.1 Level limits

For purposes of comparison of level capabilities, a particular level is considered to be a lower level than some other level when the value of the `level_idc` of the particular level is less than that of the other level.

Table A. 1 specifies limits for each level.

A level to which a bitstream conforms are indicated by the syntax elements `level_idc` as follows:

- `level_idc` shall be set equal to a value of 20 times the level number specified in Table A. 1.

Table A. 1 — Level limits

Level	Maximum number of pixels
4	1,100,000

Annex B

Type-length-value bytestream format

B.1 General

This annex specifies syntax and semantics of a byte stream format for use by applications that deliver some or all of the data units as an ordered stream of bytes without any requirement for further encapsulation in a file format.

The byte stream format consists of a sequence of type-length-value encapsulation structures that each represent a single coded syntax structure.

B.2 Syntax and semantics

B.2.1 Syntax

<code>tlv_encapsulation() {</code>	Descriptor
<code>tlv_type</code>	<code>u(8)</code>
<code>tlv_num_payload_bytes</code>	<code>u(32)</code>
<code>for(i = 0; i < tlv_num_payload_bytes; i++)</code>	
<code>tlv_payload_byte[i]</code>	<code>u(8)</code>
<code>}</code>	

B.2.2 Semantics

The order of TLV encapsulation structures shall follow the decoding order of the encapsulated syntax structures.

`tlv_type` identifies the syntax structure represented by `tlv_payload_byte[]` according to Table B. 1.

Table B. 1 — Mapping of `tlv_type` and associated data unit to syntax tables

<code>tlv_type</code>	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 data unit
3	7.3.1.3	Attribute parameter set
4	7.3.3.1	Attribute data unit
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 decoding process

Input to this process is an ordered stream of bytes consisting of a sequence of TLV encapsulation structures.

Output of this process is a sequence of syntax structures.

The decoder repeatedly parses tlv_encapsulation structures until the end of the bytestream has been encountered (as determined by unspecified means) and the last NAL unit in the byte stream has been decoded.

After parsing each tlv_encapsulation structure, the following occurs

- the array DataUnitBytes is set equal to tlv_payload_byte[],

- the variable DataUnitLength is set equal to tlv_num_payload_bytes,

- the parsing process in Table B. 1 corresponding to tlv_type is invoked.