# Slice geometry

## General

Clause 9 specifies the coding of slice geometry and the reconstruction of point positions, storing the reconstructed geometry in the arrays PointPos and PointAng.

## Occupancy tree

### General

Subclause 9.2 specifies the parsing and reconstruction of point positions from a coded occupancy tree. It applies when geom\_tree\_type is 0.

An occupancy tree represents the slice geometry as a tree of occupancy tree nodes. Parsing or traversing a coded occupancy tree implicitly generates a representation of the slice geometry.

### Coded occupancy tree

#### General tree structure

Individual point positions are represented in the occupancy tree either by the position of leaf nodes, or by direct nodes that encode node-relative positions.

An occupancy tree node shall identify the presence of at least one point contained within the volume of an axis-aligned cuboid. The volume is defined in the slice's coordinate system by an inclusive lower corner and an exclusive upper corner . The volume edge lengths are non-negative integer powers of two. A node's size, nodeSize, is synonymous with the volume dimensions .

The occupancy tree shall be formed of one or more tree levels. Every tree level consists of tree nodes with non-overlapping volumes. All tree nodes within a tree level shall have identical volume dimensions.

The occupancy tree shall contain a single root node. The root node shall be the only node in the top level of the tree. The volume identified by the root node shall have a lower corner at position ( 0, 0, 0 ), coincident with the slice origin. The upper corner shall be at an integer position equal to the root node size.

With each subsequent tree level, starting from the top tree level, the node volume dimensions are halved along one or more coded axes. The coded axes in each tree level are enumerated in the GDU header, as specified by occtree\_coded\_axis.

The location of a node, nodeLoc, within a tree level is related to the spatial position of the node volume's lower corner in the slice coordinate system by:

Two tree nodes are spatially adjacent if their volumes share a face.

Unless an early termination condition applies (9.2.6.5), tree nodes with a volume greater than the unit cube shall have one or more child nodes. Depending upon the number of coded axes, these nodes shall have at most eight, four, two, or one child nodes, as illustrated for cubic nodes in Figure 5.

形状

描述已自动生成

Figure 5 — Arrangement of child nodes depending upon coded axes.

Leaf nodes, in the absence of geometry scaling (9.2.14), represent indivisible volumes with dimensions equal to the unit cube. When duplicate point coding is enabled, a leaf node may represent more than one point. In such cases, all points represented by the leaf node shall have identical positions.

#### Tree traversal order

The coded occupancy tree shall be traversed in breadth-first order. Traversal shall start from the top tree level. All nodes in a tree level shall be sequentially traversed before proceeding to the next level. Within a tree level, nodes shall be traversed in ascending Morton order of node location.

The traversal order for an example tree is illustrated in Figure 6. Each tree level progressively refines the slice geometry. Starting from the root node, a, the node traversal order is from a to y. Occupied nodes are shaded.

图示

描述已自动生成

Figure 6 — Occupancy tree traversal order. Illustrated for three tree levels.

#### Node occupancy bitmap

The structure of the occupancy tree is coded as a sequence of node occupancy bitmaps.

Each node occupancy bitmap shall enumerate the child nodes for a single node. Each set bit position identifies the relative location of a child node in the next level of the occupancy tree as specified by the expression OccLocC[ bitIdx ][ 𝑘 ] and Table 13. The count of set bits in the bitmap shall be the number of child nodes.

When a node has fewer than three coded axes, bits in the occupancy bitmap that do not enumerate a valid child node location shall be unset.

OccLocC[bitIdx][k] := Bit(bitIdx, 2 − k)

The tree level location of a child node is related to its parent's by:

Table 13 — Identification of valid relative child node location relLoc from set bits in an occupancy bitmap

| Coded Axes | | | Bit position (bitIdx) in occupancy bitmap | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | T | V | 7 (MSB) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (LSB) |
| 1 | 1 | 1 | ( 1, 1, 1 ) | ( 1, 1, 0 ) | ( 1, 0, 1 ) | ( 1, 0, 0 ) | ( 0, 1, 1 ) | ( 0, 1, 0 ) | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 1 | 1 | 0 |  | ( 1, 1, 0 ) |  | ( 1, 0, 0 ) |  | ( 0, 1, 0 ) |  | ( 0, 0, 0 ) |
| 1 | 0 | 1 |  |  | ( 1, 0, 1 ) | ( 1, 0, 0 ) |  |  | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 0 | 1 | 1 |  |  |  |  | ( 0, 1, 1 ) | ( 0, 1, 0 ) | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 1 | 0 | 0 |  |  |  | ( 1, 0, 0 ) |  |  |  | ( 0, 0, 0 ) |
| 0 | 1 | 0 |  |  |  |  |  | ( 0, 1, 0 ) |  | ( 0, 0, 0 ) |
| 0 | 0 | 1 |  |  |  |  |  |  | ( 0, 0, 1 ) | ( 0, 0, 0 ) |
| 0 | 0 | 0 |  |  |  |  |  |  |  | ( 0, 0, 0 ) |

#### Terminal nodes

In the coded occupancy tree, leaf nodes are immediately encoded by their parent (terminal) node.

When geom\_dup\_point\_counts\_enabled is 1, terminal nodes shall encode the duplicate point counts for the leaf nodes they contain.

### Occupancy tree syntax element semantics

occtree\_depth\_minus1 plus 1 specifies the maximum number of tree levels present in the coded occupancy tree. When occtree\_coded\_axis\_list\_present is 0, the root node size is a cubic volume with edge lengths equal to Exp2( occtreeMaxDepthMinus1 + triSoupNodeSizeLog2 + 1 ).

The number of coded tree levels coded is equal to occtreeMaxDepthMinus1 plus 1. The value of occtreeMaxDepthMinus1 is set equal to occtree\_depth\_minus1 when trisoup\_enable\_flag is equal to 0, otherwise when trisoup\_enable\_flag is equal to 1 the value of occtreeMaxDepthMinus1 is set as

occtreeMaxDepthMinus1 = occtree\_depth\_minus1 - 1

In the latter case, occupancy tree coding is performed until reaching not the unit cube size but node size triSoupNodeSizeLog2 instead. The TriSoup process (9.4) is performed for each node with node size triSoupNodeSizeLog2.

When trisoup\_enable\_flag is equal to 1, the value triSoupNodeSizeLog2 is set to  trisoup\_node\_size\_log2\_minus2 + 2 . Otherwise, when trisoup\_enable\_flag is equal to 0, the value triSoupNodeSizeLog2 is set to 0.

occtree\_coded\_axis[ dpth ][ 𝑘 ] specifies whether (when 1) or not (when 0) a subdivision along the 𝑘-th STV axis is coded by tree nodes at depth dpth. occtree\_coded\_axis shall be used to determine the node volume dimensions in each level of the occupancy tree. When occtree\_coded\_axis[ dpth ][ 𝑘 ] is not present, it shall be inferred to be 1.

It is a requirement of bitstream conformance that:

* There shall be at least one coded axis in every tree level specified by occtree\_coded\_axis; i.e. MaxVec( occtree\_coded\_axis[ dpth ] ) == 1.
* The log2 dimensions of the root node shall be less than or equal to MaxSliceDimLog2.
* The largest log2 dimension of the root node shall be greater than  occtreeMaxDepthMinus1 + triSoupNodeSizeLog2 – 4.

occtree\_stream\_cnt\_minus1 plus 1 specifies the maximum number of entropy streams used to code the occupancy tree. When occtree\_stream\_cnt\_minus1 is greater than zero, each of the bottom occtree\_stream\_cnt\_minus1 tree levels shall be conveyed in a separate entropy stream; the parsing state shall be memorized and restored according to 11.6.

The expression OcctreeEntropyStreamDepth is the depth of the last tree level that is coded in the first entropy stream.

OcctreeEntropyStreamDepth := occtreeMaxDepthMinus1 − occtree\_stream\_cnt\_minus1

occtree\_end\_of\_entropy\_stream is a non-coded syntax element used to specify the termination point for the arithmetic decoder at the end of an entropy stream. The syntax element has no value.

occtree\_lvl\_point\_cnt\_minus1[ dpth ] plus 1 indicates, when present, the number of points that can be partially decoded (See Annex D) from the root node to the end of the tree level at depth dpth. occtree\_lvl\_point\_cnt\_minus1[ 0 ] shall be inferred to be 0. occtree\_lvl\_point\_cnt\_minus1[ occtreeMaxDepthMinus1 ] shall be inferred to be slice\_num\_points\_minus1.

### Node dimensions per tree level

The log2 node dimensions at depth dpth are specified by the expression OccLvlNodeSizeLog2[ dpth ][ 𝑘 ]. They are derived from the list of coded axes:

* In the bottom tree level, at depth occtreeMaxDepthMinus1 + 1, the node dimensions shall be equal to the cube with the size triSoupNodeSizeLog2.
* The log2 node dimensions in any tree level shallower than the bottom tree level shall be the count of the respective coded axes, proceeding from the bottom tree level.

OccLvlNodeSizeLog2[dpth][k] := lvl ≤ occtreeMaxDepthMinus1  
 ? OccLvlNodeSizeLog2[dpth + 1][k] + occtree\_coded\_axis[dpth][k]  
 : triSoupNodeSizeLog2

### State representation

#### State variables

The occupancy tree is specified in terms of the following state variable:

* The sparse array OccNodePresent that identifies nodes present in the occupancy tree. Each element OccNodePresent[ dpth ][ ns ][ nt ][ nv ] equal to either 1 or −1 indicates the presence of a node at location ( ns, nt, nv ) and depth dpth. Elements equal to −1 identify leaf nodes that are not coded at that depth (9.2.2.4). Unset elements of OccNodePresent are inferred to be 0.

Traversal of the occupancy tree is specified in terms of the following state variables:

* The array OccNodeCnt; OccNodeCnt[ dpth ] is the cumulative count of nodes present at depth dpth.
* The array OccNodeLoc; OccNodeLoc[ dpth ][ nodeIdx ][ 𝑘 ] identifies the 𝑘-th location component of the nodeIdx-th coded node in the traversal order of the tree level at depth dpth.

#### The root node

At the start of the occupancy tree syntax structure, the arrays OccNodePresent, OccNodeLoc and OccNodeCnt are initialized to represent the root node at location ( 0, 0, 0 ); all other elements are cleared.

OccNodePresent[0][0][0][0] = 1  
OccNodeLoc[0][0][0] = OccNodeLoc[0][0][1] = OccNodeLoc[0][0][2] = 0  
OccNodeCnt[0] = 1

### Occupancy tree node coding

#### General

Subclause 9.2.6 specifies the semantics of the NodeIdx-th coded node at tree depth Dpth.

#### Syntax element semantics

occ\_single\_child equal to 1 specifies that the coded node has a single child. occ\_single\_child equal to 0 specifies that the coded node may generate multiple child nodes. When occ\_single\_child is not present, it shall be inferred to be 0.

occupancy\_idx[ 𝑘 ] specifies the 𝑘-th component of the relative child node location for the only child of the coded node.

occupancy\_bit and occupancy\_byte specify the child nodes of the coded node as neighbourhood-permuted node occupancy bitmaps. The syntax elements shall be coded as specified by 9.2.10 and 9.2.9, respectively.

occ\_dup\_point\_cnt[ 𝑖 ] plus 1 specifies the number of points represented by the 𝑖-th coded child leaf node. All points represented by a child have identical positions. When occ\_dup\_point\_cnt[ 𝑖 ] is not present in a terminal node, it is inferred to be 0.

When unique\_point\_positions\_constraint is 1, it is a requirement of bitstream conformance that occ\_dup\_point\_cnt[ 𝑖 ] shall be 0.

#### Node, parent, grandparent and child tree-level locations

The tree-level location ( Ns, Nt, Nv ) of the coded node is specified by the expression Nloc[ 𝑘 ].

Nloc[k] := OccNodeLoc[Dpth][NodeIdx][k]

Ns := Nloc[0]  
Nt := Nloc[1]  
Nv := Nloc[2]

The parent node has a location ( NsP, NtP, NvP ) in the tree level at depth Dpth − 1. It is specified by the expression NlocP[ 𝑘 ].

NlocP[k] := Dpth ? Nloc[k] >> occtree\_coded\_axis[Dpth − 1][k] : 0

NsP := NlocP[0]  
NtP := NlocP[1]  
NvP := NlocP[2]

The grandparent node has a location ( NsG, NtG, NvG ) in the tree level at depth Dpth − 2. It is specified by the expression NlocG[ 𝑘 ].

NlocG[k] := Dpth > 1 ? NlocP[k] >> occtree\_coded\_axis[Dpth − 2][k] : 0

NsG := NlocG[0]  
NtG := NlocG[1]  
NvG := NlocG[2]

The corresponding location ( NsC, NtC, NvC ) in the tree level at depth Dpth + 1 for the coded node is specified by the expression NlocC[ 𝑘 ].

NlocC[k] := Nloc[k] << AxisCoded[k]

NsC := NlocC[0]  
NtC := NlocC[1]  
NvC := NlocC[2]

#### Node size

The expressions NodeSizeLog2[ 𝑘 ] and ChildNodeSizeLog2[ 𝑘 ] specify the log2 dimensions of the coded node and its children, respectively.

NodeSizeLog2[k] := OccLvlNodeSizeLog2[Dpth][k]

ChildNodeSizeLog2[k] := OccLvlNodeSizeLog2[Dpth + 1][k]

When geom\_scaling\_enabled is 0, QuantizedNodeSizeLog2[ 𝑘 ] is equal to NodeSizeLog2[ 𝑘 ].

#### Whether the node is a terminal node

A node is a terminal node, as specified by the expression TerminalNode, if its children are leaf nodes, or it is a direct node.

TerminalNode := MaxVec(ChildNodeSizeLog2) == 0  
 || geom\_scaling\_enabled && MaxVec(QuantizedChildNodeSizeLog2) == 0  
 || occtree\_direct\_coding\_mode && occ\_direct\_node

#### Coded axes

A node shall only code an axis for child locations when specified by the expression AxisCoded[ 𝑘 ]. An axis is coded when:

* it is specified to be coded in the tree level by the coded axis list, and
* if geometry subtree scaling is enabled, the corresponding dimension of the quantized node size is greater than 1.

AxisCoded[k] := occtree\_coded\_axis[Dpth][k]  
 && (¬geom\_scaling\_enabled || QuantizedNodeSizeLog2[k] > 0)

The expression CodedAxisCnt is the number of axes coded by the node.

CodedAxisCnt := AxisCoded[0] + AxisCoded[1] + AxisCoded[2]

The location of child nodes along each coded axis may be constrained by planar occupancy coding (9.2.11.4). A free axis is a coded axis that is not constrained so, as specified by the expression OccFreeAxis[ 𝑘 ]. The number of free axes is specified by the expression OccFreeAxisCnt.

When planar occupancy coding is disabled, OccFreeAxisCnt is equal to CodedAxisCnt.

OccFreeAxis[k] := AxisCoded[k] && (¬occtree\_planar\_enabled || PlanarFreeAxis[k])

OccFreeAxisCnt := OccFreeAxis[0] + OccFreeAxis[1] + OccFreeAxis[2]

#### Limits to the number of child nodes

The number of child nodes a node can contain is constrained by the tree and node syntax.

The maximum number of child nodes is specified by the expression OccMaxChildren. Unless occ\_single\_child is 1, the limit shall be the number of child node locations that can be identified by the free axes. Otherwise, the limit shall be 1 when occ\_single\_child is 1.

OccMaxChildren := occ\_single\_child ? 1 : Exp2(OccFreeAxisCnt)

The minimum number of child nodes is specified by the expression OccMinChildren. A node shall contain at least one child node unless:

* planar occupancy coding specifies that there shall be at least two child nodes (9.2.11.3), or
* occ\_single\_child is both present and equal to 0, in which case there shall be at least two child nodes.

OccMinChildren :=  
 OccMaybeSingleChild && ¬occ\_single\_child ? 2 :  
 occtree\_planar\_enabled ? PlanarMinChildren : 1

#### Presence of occ\_single\_child

The presence of occ\_single\_child is specified by the expression OccMaybeSingleChild. It shall be present in the occupancy node syntax when all the following conditions are true:

* No nodes are present in the occupied neighbourhood pattern (9.2.7.4).
* OccupancyIsPredictable is 0 or there is at least one free axis to code child node locations.
* Planar occupancy coding does not specify that there shall be at least two child nodes (9.2.11.3).

OccMaybeSingleChild :=  
 ¬OccNeighPat && OccFreeAxisCnt > 0 && PlanarMinChildren == 1

#### Presence of occupancy\_bit and occupancy\_byte

The node occupancy bitmap shall be coded using either occupancy\_bit or occupancy\_byte when specified by the expression OccMapPresent. One of the two syntax elements shall be present when both of the following conditions are true:

* The maximum number of child nodes is greater than 1.
* The locations of child nodes are not completely prescribed by constraints on occupancy. i.e. the maximum number of child nodes is greater than the minimum number of child nodes.

OccMapPresent := OccMaxChildren > 1 && OccMinChildren != OccMaxChildren

#### Node occupancy bitmap

This subclause specifies the node occupancy bitmap by the expression OccupancyMap.

When occupancy\_bit syntax elements are present (OccMapPresent is 1), the node occupancy bitmap is specified by bitwise occupancy coding (9.2.10.2).

if (OccMapPresent && occtree\_bitwise\_coding)  
 OccupancyMap = OccBitMap

When occupancy\_byte is present (OccMapPresent is 1), the node occupancy bitmap shall be rearranged from the neighbourhood-permuted bitmap (9.2.8) coded by occupancy\_byte.

if (OccMapPresent && ¬occtree\_bitwise\_coding)  
 OccupancyMap = OccFromNpOcc(occupancy\_byte)

When constraints on occupancy require there to be a single child node, each component 𝑘 of the child location shall be specified by whichever of occ\_plane\_pos[ 𝑘 ] or occupancy\_idx[ 𝑘 ] are present, or shall be 0 if neither is present.

if (OccMaxChildren == 1 && OccMinChildren == 1) {  
 occupancyIdx[k] :=  
 OccFreeAxis[k] && occupancy\_idx[k] || ¬PlanarFreeAxis[k] && occ\_plane\_pos[k]  
  
 OccupancyMap = 1 << Morton[occupancyIdx]  
}

In this case, an axis cannot be both a free axis and eligible for planar occupancy coding.

When constraints on occupancy require there to be two child nodes, there shall be one child node at both permitted locations along the free axis.

if (OccMaxChildren == 2 && OccMinChildren == 2) {  
 baseIdx[k] := ¬PlanarFreeAxis[k] && occ\_plane\_pos[k]  
  
 if (OccFreeAxis[0]) OccupancyMap = 0x11 << Morton[baseIdx]  
 if (OccFreeAxis[1]) OccupancyMap = 0x05 << Morton[baseIdx]  
 if (OccFreeAxis[2]) OccupancyMap = 0x03 << Morton[baseIdx]  
}

#### Child node count

The number of child nodes is equal to the number of set bits in the node occupancy bitmap, as specified by the expression OccChildCnt.

OccChildCnt := PopCnt(OccupancyMap)

#### Insertion of non-terminal child nodes

Unless the coded node is a terminal node, its child nodes shall be inserted into the state representation of the occupancy tree and included in the traversal list of the next tree level. The node occupancy bitmap shall be scanned to enumerate the child nodes.

if (¬TerminalNode)  
 for (occBitIdx = 0; occBitIdx < 8; occBitIdx++) {  
 if (¬Bit(OccupancyMap, occBitIdx))  
 continue  
  
 cs = NsC + OccLocC[occBitIdx][0]  
 ct = NtC + OccLocC[occBitIdx][1]  
 cv = NvC + OccLocC[occBitIdx][2]  
 OccNodePresent[Dpth + 1][cs][ct][cv] = 1  
  
 childNodeIdx = OccNodeCnt[Dpth + 1]  
 OccNodeCnt[Dpth + 1]++  
  
 OccNode[Dpth + 1][childNodeIdx][0] = cs  
 OccNode[Dpth + 1][childNodeIdx][1] = ct  
 OccNode[Dpth + 1][childNodeIdx][2] = cv  
 }

#### Points represented by child leaf nodes

When the node is a non-direct terminal node, points represented by child leaf nodes shall be scaled (9.2.14.6) and appended to the output point list. The node occupancy bitmap shall be scanned to enumerate the child nodes.

When geometry scaling is disabled, this condition is equivalent to the child node size being equal to the unit cube.

When adjacent child occupancy contextualization is enabled, the child leaf nodes shall be inserted into the state representation of the occupancy tree for use by other nodes in the same tree level; but they shall be excluded from traversal in the next tree level. The child leaf nodes shall not be included in the occupied neighbourhood pattern for any node in the next tree level.

if (TerminalNode && ¬occ\_direct\_node)  
 for (child = 0, occBitIdx = 0; occBitIdx < 8; occBitIdx++) {  
 if (¬Bit(OccupancyMap, occBitIdx))  
 continue  
  
 cs = NsC + OccLocC[occBitIdx][0]  
 ct = NtC + OccLocC[occBitIdx][1]  
 cv = NvC + OccLocC[occBitIdx][2]  
 if (occtree\_adjacent\_child\_enabled)  
 OccNodePresent[Dpth + 1][cs][ct][cv] = −1  
  
 for (i = 0; i < occ\_dup\_point\_cnt[child] + 1; i++, PointCnt++) {  
 PointPos[PointCnt][0] = OccPosScaleK(0, cs)  
 PointPos[PointCnt][1] = OccPosScaleK(1, ct)  
 PointPos[PointCnt][2] = OccPosScaleK(2, cv)  
 }  
  
 child++  
 }

#### Definition of ChildIdx

This subclause specifies the expression ChildIdx[ pt ] that is the child index of point pt for a given node.

ChildIdx[pt] = (pt[2] & mask[2] ? 1 : 0) +   
 ((pt[1] & mask[1] ? 1 : 0) << 1) +  
 ((pt[0] & mask[0] ? 1 : 0) << 2)  
where  
 mask[k] = (NodeSizeLog2[k] != ChildNodeSizeLog2[k]) ? 1 << ChildNodeSizeLog2[k] : 0

#### Derivation of OccupancyIsPredictable

OccupancyIsPredictable specifies whether the current node satisfies conditions to enable inter prediction of occupancy of child nodes.

occupancyIsPredictable = numSiblingsMispredcted <= 5 && PredOccBitMap > 0  
where  
 numSimblingsMispredicted = PopCnt(PredOccBitMap ^ OccupancyMap)

#### Derivation of PredOccBitMap

The occupancy of the regions collocated to the child nodes of the current node is stored in the variable PredOccBitMap.

PredOccBitMap is derived as follows

PredOccBitMap1 = 0  
if(enableInterPredFromRef)  
 for(i = 0;i < 8; i++)  
 if(PredCounts1[i])  
 PredOccBitMap1 |= 1 << i  
PredOccBitMap2 = 0  
if(enableInterPredFromRef2)  
 for(i = 0;i < 8; i++)  
 if(PredCounts2[i])  
 PredOccBitMap2 |= 1 << i  
PredOccBitMap = predDir ? PredOccBitMap2 : PredOccBitMap1

#### Derivation of nStart, nEnd, nStart2 and nEnd2

When *enableInterPredFromRef* is 1, the variables nStart and nEnd for a node that is the i-th child node of its parent node specify the start and end indices of a sorted reference frame that contains points belonging to the reference frame that are collocated with the node. Let nStart\_parent denote the nStart variable of the parent node.

When the node is the root node, nStart is 0 and nEnd is the total number of points in the slice.

Otherwise, (node is not the root node), nStart and nEnd are derived as follows:

nStart = nStart\_parent  
for(j = 0; j < i; j++)  
 nStart += PredCounts[j]  
nEnd = nStart + PredCounts[i]

When *enableInterPredFromRef2* is 1, the variables nStart2 and nEnd2 for a node that is the i-th child node of its parent node specify the start and end indices of the second sorted reference frame that contains points belonging to the second reference frame that are collocated with the node. nStart2 and nEnd2 are derived from the total number of points in the slice of second reference frame or *PredCounts2*, in the same way as nStart and nEnd.

[Ed. (YZ): In order to avoid duplication of description, the phrase "in the same way as ..." is used. No sure if it is permitted to describe in this way.]

#### Derivation of PredCounts and PredCounts2

When *enableInterPredFromRef* is 1, the array PredCounts[i] denotes the start indices of the positions of the octree node in the reference frame collocated with the i-th child node.

The frame buffers RefFramePos and RefFramePos2 are set equal to PredPointCloud and SecondPredPointCloud, respectively, at the beginning of the octree slice.

First, the number of points in reference frame that are collocated with each child node, colRefPoints[i] is calculated. colRefPoints[0..7] is initialized to 0.

for(i = nStart; i < nEnd; i++) {  
 pt = RefFramePos[i]  
 colRefPoints[ChildIdx[pt]]++  
}

The array PredCounts is derived as follows:

PredCounts[0] = 0  
for(i = 1; i < 8; i++)  
 PredCounts[i] = PredCounts[i-1] + colRefPoints[i-1]

When *enableInterPredFromRef2* is 1, the array PredCounts2[i] denotes the start indices of the positions of the octree node in the second reference frame collocated with the i-th child node. PredCounts2 is derived from the number of points in second reference frame that are collocated with each child node, in the same way as PredCounts.

[Ed. (YZ): In order to avoid duplication of description, the phrase "in the same way as ..." is used. No sure if it is permitted to describe in this way.]

#### Update of RefFramePos and RefFramePos2 at the beginning of coding a node

When *enableInterPredFromRef1* is 1, the points in the reference frame buffer RefFramePos are sorted based on the child node regions.

nIdx[0] = nStart  
for(i = 1; i < 8; i++)  
 nIdx[i] = nStart + PredCounts[i]  
lastIdx = 0  
for (int i = 0; i < Radix; i++) {  
 lastIdx += PredCounts[i]  
 while (nIdx[i] != lastIdx) {  
 radix = ChildIdx[RefFramePos[nIdx[i]]]  
 Swap(RefFramePos[nIdx[i]], RefFramePos[nIdx[radix]]);  
 nIdx[radix]++  
 }  
}

When *enableInterPredFromRef2* is 1, the points in the second reference frame buffer RefFramePos2 are sorted based on the child node regions.

nIdx[0] = nStart2  
for(i = 1; i < 8; i++)  
 nIdx[i] = nStart2 + PredCounts2[i]  
lastIdx = 0  
for (int i = 0; i < Radix; i++) {  
 lastIdx += PredCounts2[i]  
 while (nIdx[i] != lastIdx) {  
 radix = ChildIdx[RefFramePos2[nIdx[i]]]  
 Swap(RefFramePos2[nIdx[i]], RefFramePos2[nIdx[radix]]);  
 nIdx[radix]++  
 }  
}

#### Derivation of *enableInterPredFromRef* and *enableInterPredFromRef2*

When slice\_biprediction is 1 and frame\_merge\_enabled is 0, *enableBipredDerive* is derived as follows:

enableBipredDerive = slice\_biprediction && !frame\_merge\_enabled && (nodeSizeLog2[0] >= 1) && (nodeSizeLog2[1] >= 1) && (nodeSizeLog2[2] >= 1)

The variables *enableInterPredFromRef* and *enableInterPredFromRef2* specify whether (1) or not (0) the first and second reference frames are used in inter prediction. *enableInterPredFromRef* and *enableInterPredFromRef2* are derived as:

enableInterPredFromRef = enableBipredDerive || !predDir  
enableInterPredFromRef2 = enableBipredDerive || predDir

#### Derivation of *predDir*

The variables predDir for a node that is the i-th child node of its parent node specify the inter prediction is from the first reference frame (0) or the second reference frame (1). predDir is initilized to 0 for the root node.

Let predDir\_parent denote the predDir variable of the parent node. predDir is derived as follows:

predDir = predDir\_parent  
if (enableBiPredDerive)  
 if (!PredCounts2[i])  
 predDir = 0  
 else if (!PredCounts[i])  
 predDir = 1  
 else  
 predDir = predFailureCount != predFailureCount2 ? (predFailureCount >=   
 predFailureCount2) : predDir\_parent  
 Where  
 predFailureCount = enableInterPredFromRef ? PopCnt(PredOccBitMap ^ OccupancyMap) : 0  
 predFailureCount2 = enableInterPredFromRef2 ? PopCnt(PredOccBitMap2 ^ OccupancyMap) :   
 0

### Occupied neighbourhood patterns

#### General

Coding of the node occupancy bitmap depends upon the existence and arrangement of up to six spatially adjacent tree nodes within an availability window. The occupied neighbourhood pattern for an occupancy tree node shall identify the spatial arrangement of these adjacent nodes from the 64 possible combinations. Examples of occupied neighbourhood patterns are illustrated in Figure 7.

图示

描述已自动生成

Figure 7 — Characteristic occupied neighbourhood patterns.

#### Neighbour availability

Nodes are grouped into availability windows by their spatial location within their tree level. Nodes can form part of the occupied neighbourhood pattern of adjacent nodes within the same window. Nodes shall not form part of the occupied neighbourhood pattern of any node in a different window.

The size of the availability window is specified by the expression OccAvailWinLog2[ 𝑘 ]:

* Unless occtree\_neigh\_window\_log2\_minus1 is 0, each availability window shall span 𝑛×𝑛×𝑛 node locations, 𝑛 = Exp2( occtree\_neigh\_window\_log2\_minus1 + 1 ), within the tree level. The availability windows form a contiguous grid starting from the location ( 0, 0, 0 ).
* Otherwise, the availability window for any node shall be restricted to its sibling nodes.

OccAvailWinLog2[k] := occtree\_neigh\_window\_log2\_minus1  
 + (occtree\_neigh\_window\_log2\_minus1 > 0 || Dpth > 0 && occtree\_coded\_axis[Dpth − 1][k])

Only the Main profile permits an element of occtree\_coded\_axis to be 0 when occtree\_neigh\_window\_log2\_minus1 is 0.

The expression OccNeighAvail[ ns ][ nt ][ nv ] specifies whether the node at location ( ns, nt, nv ) is within the same availability window as the coded node ( Ns, Nt, Nv ).

OccNeighAvail[ns][nt][nv] :=  
 (ns ^ Ns) >> OccAvailWinLog2[0] == 0  
 && (nt ^ Nt) >> OccAvailWinLog2[1] == 0  
 && (nv ^ Nv) >> OccAvailWinLog2[2] == 0

#### Presence of another coded node within the availability window

The expression OccNeigh[ ns ][ nt ][ nv ] identifies whether there exists a node with tree location ( ns, nt, nv ) and depth Dpth that is not a leaf node and is within the availability window of the coded node ( Ns, Nt, Nv ).

OccNodePresent[ Dpth ][ ns ][ nt ][ nv ] equal to −1 identifies a leaf node.

OccNeigh[ns][nt][nv] := OccNeighAvail[ns][nt][nv] && OccNodePresent[Dpth][ns][nt][nv] == 1

#### Occupied neighbourhood pattern

The occupied neighbourhood pattern for the coded node located at ( Ns, Nt, Nv ) in the tree level at depth Dpth is specified by the expression OccNeighPat. It is a linear combination of spatially adjacent nodes coded in the same tree level that are available and adjoin the coded node by a face. Leaf nodes shall not be included in the occupied neighbourhood pattern.

An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood pattern equal to 0.

OccNeighPat := (uN << 5) | (dN << 4) | (bN << 3) | (fN << 2) | (rN << 1) | lN  
 where  
 rN := OccNeigh[Ns + 1][Nt][Nv]  
 lN := OccNeigh[Ns − 1][Nt][Nv] && (¬occtree\_adjacent\_child\_enabled || lNadj)  
 bN := OccNeigh[Ns][Nt + 1][Nv]  
 fN := OccNeigh[Ns][Nt − 1][Nv] && (¬occtree\_adjacent\_child\_enabled || fNadj)  
 uN := OccNeigh[Ns][Nt][Nv + 1]  
 dN := OccNeigh[Ns][Nt][Nv − 1] && (¬occtree\_adjacent\_child\_enabled || dNadj)

When adjacent child contextualization is enabled (occtree\_adjacent\_child\_enabled is 1), a tree node that adjoins the left (Ns − 1), front (Nt − 1) or bottom (Nv − 1) face shall not be included in the occupied neighbourhood pattern unless it contains at least one child node that also adjoins the same face. Their inclusion is specified by the expressions lNadj, fNadj and dNadj, equivalent to the following:

lNadj = fNadj = dNadj = 0  
for (s = 0; s ≤ occtree\_coded\_axis[Dpth][0]; s++)  
 for (t = 0; t ≤ occtree\_coded\_axis[Dpth][1]; t++)  
 for (v = 0; v ≤ occtree\_coded\_axis[Dpth][2]; v++) {  
 lNadj |= OccNodePresent[Dpth + 1][NsC − 1][NtC + t][NvC + v] ≠ 0  
 fNadj |= OccNodePresent[Dpth + 1][NsC + s][NtC − 1][NvC + v] ≠ 0  
 dNadj |= OccNodePresent[Dpth + 1][NsC + s][NtC + t][NvC − 1] ≠ 0  
 }

The exclusion of a node from the occupied neighbourhood pattern due to adjacent child contextualization is illustrated in Figure 8. The child node of the left neighbour does not adjoin the left face of the coded node N resulting in its parent node being excluded from the occupied neighbourhood pattern; OccNeighPat is 2. The child node does adjoin N and its parent node is not excluded; OccNeighPat is 3.

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| N | Coded node |
|  | Child node at location 𝑚 in left neighbour |

Figure 8 — Effect of adjacent child contextualization on an occupied neighbourhood pattern. (Top) Exclusion. (Bottom) Inclusion.

#### Reduced occupied neighbourhood pattern

The occupied neighbourhood pattern shall be reduced to one of a smaller set of patterns as specified by the expression OccNeighPatR.

When occtree\_neigh\_window\_log2\_minus1 is greater than 0, the smaller set of patterns is specified by Table 14 as a mapping of spatial rotations and reflections in the arrangement of the six spatially adjacent neighbours to produce nine unique arrangements.

Otherwise, when occtree\_neigh\_window\_log2\_minus1 is 0, the smaller set of patterns is specified by Table 15 as a mapping of adjacent siblings that produces six arrangements.

OccNeighPatR := occtree\_neigh\_window\_log2\_minus1 > 0  
 ? NeighPat64to9[OccNeighPat]  
 : NeighPat64to6[OccNeighPat]

Table 14 — Reduction of occupied neighbourhood pattern 𝑗 + 𝑖 to nine patterns, NeighPat64to9[ 𝑗 + 𝑖 ]

| 𝑗 | 𝑖 | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **0** | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 3 | 1 | 3 | 3 | 4 |
| **16** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **32** | 1 | 2 | 2 | 3 | 2 | 5 | 5 | 6 | 2 | 5 | 5 | 6 | 3 | 6 | 6 | 7 |
| **48** | 1 | 3 | 3 | 4 | 3 | 6 | 6 | 7 | 3 | 6 | 6 | 7 | 4 | 7 | 7 | 8 |

Table 15 — Reduction of occupied neighbourhood pattern 𝑗 + 𝑖 to six patterns, NeighPat64to6[ 𝑗 + 𝑖 ]

| 𝑗 | 𝑖 | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **0** | 0 | 5 | 5 | na | 5 | 1 | 1 | na | 5 | 1 | 1 | na | na | na | na | na |
| **16** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **32** | 2 | 3 | 3 | na | 3 | 7 | 7 | na | 3 | 7 | 7 | na | na | na | na | na |
| **48** | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na | na |
| 1. The specification of values 5 and 7 aligns with further reductions performed in bitwise occupancy coding. | | | | | | | | | | | | | | | | |

#### Advanced occupied neighbourhood patterns

[Ed. (YZ): the implementation shall be further check to ensure it consistent with the spec text.]

The advanced occupied neighbourhood patterns for one node located at ( *Ns*, *Nt*, *Nv* ) in the tree level at depth *Dpth* are the occupancy combinations of the decoded spatially adjacent nodes in the tree level at depth *Dpth* and neighbouring child nodes in the tree level at depth *Dpth*+1. Leaf nodes shall not be included in the advanced occupied neighbourhood patterns. For each child node, one advanced occupied neighbourhood pattern *OccAdvNeiPati* is derived for the child node. The derived (*OccAdvNeiPat0*, *OccAdvNeiPat1*, *OccAdvNeiPat2*, *OccAdvNeiPat3*, *OccAdvNeiPat4*, *OccAdvNeiPat5*, *OccAdvNeiPat6*, *OccAdvNeiPat7*) are used to construct contextual information *CI* to be used in Dynamic OBUF according to 9.2.10.6.10.

For one child node, the expression *occL* specifies the occupancy of the 4 neighbouring child nodes neighbouring current node on the left, *occF* specifies the occupancy of the 4 neighbouring child nodes neighbouring current node on the front, and *occB* specifies the occupancy of the 4 neighbouring child nodes neighbouring current coded node on the bottom. An example of occL, occF and occB for the first child node is illustrated in Figure9.

图片包含 游戏机, 体育

描述已自动生成

Figure 9 — *occL*, *occF*, *occB* for the first child node

occL, occF and occB are calculated from the occupancy of child nodes of three spatially adjacent nodes (*occLeft*, *occFront* and *occBottom*) on the left, front and bottom of current node.

occL := occLeft >> 4  
occF := ((occFront >> 2) & 3) | ((occFront >> 4) & 12)  
occB := ((occBottom >> 1) & 1) | ((occBottom >> 2) & 2) | ((occBottom >> 3) & 4) | ((occBottom >> 4) & 8

The expression *occOfLFB* specifies the occupancy of child nodes of three spatially adjacent nodes on the left, front and bottom.

occOfLFB := occLeft | occFront | occBottom

The expression edgeSets specifies the occupancy of 6 neighbouring child nodes, *fLF0*, *fLF1*, *fLB0*, *fLB1*, *fFB0* and *fFB1*. *fLF0* and *fLF1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occF*. *fLB0* and *fLB1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occB*. *fFB0* and *fFB1* share the same faces with both the neighbouring child nodes sets specified by *occL* and *occF*. An example of edgeSets the first child node is illustrated in Figure 10.

图片包含 游戏机

描述已自动生成

**Figure 10** — **example of *edgeSets* for the first child node**

The expression *neighPatternRBT* specifies the occupancy of 3 spatially adjacent nodes on the right, back and top of the current node. An example of *neighPatternRBT* for the node and their order to construct *neighPatternRBT* is illustrated in Figure 11. *neighPatternRBT* is calculated by accessing occupancy OccNeighPat(9.2.7.4) of six spatially adjacent nodes of the node.

neighPatternRBT := ((neighPattern >> 3) & 4) | ((neighPattern >> 2) & 2) | (neighPattern & 1)

图片包含 矩形

描述已自动生成

Figure 11 — example of three adjacent nodes of *neighPatternRBT* of node N

The expression *neighPatternLFB* specifies the occupancy of 3 spatially adjacent nodes on the bottom, front and left of the current node. *neighPatternLFB* is calculated by accessing occupancy OccNeighPat(9.2.7.4) of six spatially adjacent nodes of the node.

neighPatternLFB =((neighPattern & 0b110) >> 1)| ((neighPattern & 16) >> 2)

The expression neighb20 specifies of the occupancy of 20 spatially adjacent nodes of the current node which adjoin the current node by an edge or a vertex and not a face. An example of neighb20 for the node is illustrated in Figure 12. neighb20 is calculated by accessing OccNeigh (9.2.7.3) and constructed by an order starting from node 0 to node 19.

neighb20 = 0

for(i = 0; i < 20; i++)

neighb20 |= OccNeigh[Ns + LUTds[i]][Nt + LUTdt[i]][Nv + LUTdv[i]]<<i

where  
 LUTds[20] := {-1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1}  
 LUTdt[20] := {-1, -1, -1, 0, 0, 1, 1, 1, -1, -1, 1, 1, -1, -1, -1, 0, 0, 1, 1, 1}

LUTdv[20] := {-1, 0, 1, -1, 1, -1, 0, 1, -1, 1, -1, 1, -1, 0, 1, -1, 1, -1, 0, 1}

图片包含 游戏机, 钟表

描述已自动生成

**Figure 12 — Example of neighbouring nodes for *neighb20* of node N**

##### Derivation of OccAdvNeiPat0

The expression *OccAdvNeiPat0* specifies advanced occupied neighbourhood pattern constructed to code the first child node *b0*of the current node.

The expression *NN0* specifies the number of occupied child nodes in the three sets of neighbouring child nodes specified by *occL*, *occF* and *occB*.

* If *NN0* is larger than , then the occupied neighbour nodes of child node *b0* are not sparse and *OccAdvNeiPat0* comprises 19 bits. *OccAdvNeiPat0* is derived by evoking 9.2.7.5.2.
* Otherwise, the occupied neighbour nodes of child node *b0* are sparse and *OccAdvNeiPat0* comprises 16 bits. *OccAdvNeiPat0* is derived by evoking 9.2.7.5.3.

##### Derivation of OccAdvNeiPat0 with non-sparse neighbour nodes

When the occupied neighbour nodes of child node *b0* are not sparse, *OccAdvNeiPat0* is derived as follows.

* If more than one sets within the three sets specified by *occL*, *occF* and *occB* are occupied, *OccAdvNeiPat0*[0] is set as 1 and *OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] indicates which sets are occupied.

OccAdvNeiPat0[1]OccAdvNeiPat0[2] = occL && occF && occB ? 00 :   
(!occF ? 01 : (!occL ? 10 : 11))

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 100,
    - *OccAdvNeiPat0*[3]*…OccAdvNeiPat0*[5] indicates the occupancy the 3 neighboring child nodes sharing the same face with *b0* in the three sets specified by *occL*, *occF* and *occB* in order;
    - *OccAdvNeiPat0*[6]*OccAdvNeiPat0*[7] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB*;
    - *OccAdvNeiPat0*[8]*OccAdvNeiPat0*[9] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occF*;
    - *OccAdvNeiPat0*[10]*OccAdvNeiPat0*[11] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occL*;

[Ed. (YZ): The order of “2 neighboring child nodes” shall be further defined.]

* + - *OccAdvNeiPat0*[12]…*OccAdvNeiPat0*[14] is set as neighPatternRBT; *OccAdvNeiPat0*[15]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[15]OccAdvNeiPat0[16]OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
neighb20[8]neighb20[3]neighb20[1]neighb20[0]

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 101,
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occB and occL*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB and occL*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occB and occL*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[1]  
  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15]OccAdvNeiPat0[16]  
OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

* + if *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 110,
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occB and occF*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB and occF*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occB and occF*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[0]  
  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15]OccAdvNeiPat0[16]  
OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

* + Otherwise (*OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 111),
    - *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 neighboring child nodes sharing the same face with *b0* in the the sets specified by *occL and occF*;
    - *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] indicates the occupancy of the 4 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occL and occF*;
    - *OccAdvNeiPat0*[9]*OccAdvNeiPat0*[10] indicates the occupancy of the 2 neighboring child nodes sharing the same vertex with *b0* in the the sets specified by *occL and occF*;
    - *OccAdvNeiPat0*[11]…*OccAdvNeiPat0*[18] is calculated as follows.

OccAdvNeiPat0[11] = neighPatternRBT[2]  
  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15]OccAdvNeiPat0[16]  
OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

* Otherwise, *OccAdvNeiPat0*[0] is set as 0 and *OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] indicates which set is occupied.

OccAdvNeiPat0[1]OccAdvNeiPat0[2] = occL ? 00 : (occF ? 01 : 10)

* + If *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 000,
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occL*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occL*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occL*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 13.

OccAdvNeiPat0[7]OccAdvNeiPat0[8]OccAdvNeiPat0[9]OccAdvNeiPat0[10] =  
00edgeSets[2]edgeSets[3]  
  
OccAdvNeiPat0[11] =neighPatternRBT[0] || neighPatternRBT[1] || neighPatternRBT[2]  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] OccAdvNeiPat0[16]OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
 neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

图片包含 游戏机, 钟表

描述已自动生成

**Figure 13 — Example of edge sub-set of current child node when only the set specified by *occL* is occupied**

* + If *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 001,
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occF*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occF*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occF*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 14.

OccAdvNeiPat0[7]OccAdvNeiPat0[8] =edgeSets[4]edgeSets[5]  
  
OccAdvNeiPat0[9]OccAdvNeiPat0[9]OccAdvNeiPat0[11] =  
neighPatternRBT[0]neighPatternRBT[1]neighPatternRBT[2]  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] OccAdvNeiPat0[16]OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
 neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

QR 代码

描述已自动生成

**Figure 14 — Example of edge sub-set of current child node when only the set specified by *occF* is occupied**

* + Otherwise (*OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1]*OccAdvNeiPat0*[2] is equal to 010),
    - *OccAdvNeiPat0*[3] indicates the occupancy of the neighboring child node sharing the same face with *b0* in the three sets specified by *occB*;
    - *OccAdvNeiPat0*[4]*OccAdvNeiPat0*[5] indicates the occupancy of the 2 neighboring child nodes sharing the same edge with *b0* in the the set specified by *occB*;
    - *OccAdvNeiPat0*[6] indicates the occupancy of the neighboring child node sharing the same vertex with *b0* in the the set specified by *occB*;
    - *OccAdvNeiPat0*[7]…*OccAdvNeiPat0*[18] is calculated as follows. An example of neighbouring child nodes used to calculate *OccAdvNeiPat0*[7]*OccAdvNeiPat0*[8] is illustrated in Figure 15.

OccAdvNeiPat0[7]OccAdvNeiPat0[8] =edgeSets[0]edgeSets[1]  
  
OccAdvNeiPat0[9]OccAdvNeiPat0[9]OccAdvNeiPat0[11] =  
neighPatternRBT[0]neighPatternRBT[1]neighPatternRBT[2]  
OccAdvNeiPat0[12]OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] OccAdvNeiPat0[16]OccAdvNeiPat0[17]OccAdvNeiPat0[18] =  
 neighb20[8]neighb20[3]neighb20[1]neighb20[0]neighb20[18]neighb20[19]neighb20[11]

图片包含 游戏机

描述已自动生成

**Figure 15 — Example of edge sub-set of current child node when only the set specified by *occB* is occupied**

##### Derivation of OccAdvNeiPat0 with sparse neighbour nodes

When the occupied neighbour nodes of child node *b0* are sparse, *OccAdvNeiPat0* is derived as follows.

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occL* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 01;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occL;*
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the bottom and front of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3]OccAdvNeiPat0[4] = neighPatternLFB[2]neighPatternLFB[1]

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occF* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 10;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occF;*
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the bottom and left of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3]OccAdvNeiPat0[4] = neighPatternLFB[4]neighPatternLFB[1]

[Ed. (YZ): May be “neighPatternLFB[2]neighPatternLFB[0]”, need to further check with proponent.]

* If *NN0* is equal to 1 and the neighbouring child nodes set specified by *occB* is occupied,
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 11;
  + *OccAdvNeiPat0*[2] indicates the occupancy of the neighboring child node sharing same face with *b0* in the *occB*;
  + *OccAdvNeiPat0*[3]*OccAdvNeiPat0*[4] indicates the occupancy of the 2 spatially adjacent nodes on the front and left of the current node and adjoin the current node by a face.

OccAdvNeiPat0[3]OccAdvNeiPat0[4] = neighPatternLFB[2]neighPatternLFB[1]

[Ed. (YZ): May be “neighPatternLFB[1]neighPatternLFB[0]”, need to further check with proponent.]

* Otherwise (*NN0* is equal to 0),
  + *OccAdvNeiPat0*[0]*OccAdvNeiPat0*[1] is set as 00;
  + *OccAdvNeiPat0*[2]…*OccAdvNeiPat0*[4] indicates the occupancy of the 3 spatially adjacent nodes on the bottom, front and left of the coded node and adjoin the coded node by a face.

OccAdvNeiPat0[2]OccAdvNeiPat0[3]OccAdvNeiPat0[4] =  
 neighPatternLFB[2]neighPatternLFB[1]neighPatternLFB[0]

* *OccAdvNeiPat0*[5]…*OccAdvNeiPat0*[8] is calculated as follows.

OccAdvNeiPat0[5]OccAdvNeiPat0[6]OccAdvNeiPat0[7]OccAdvNeiPat0[8] = neighb20[1]  
 neighb20[3]neighb20[8]neighb20[0]

* If at least one of the 3 adjacent parent nodes on the bottom, front and left of the current node is occupied,
  + If the first child node within the set of neighbouring child nodes specified by *occOfLFB* is occupied, *OccAdvNeiPat0*[9]…*OccAdvNeiPat0*[12] is calculated as follows. *OccAdvNeiPat0*[10]…*OccAdvNeiPat0*[12] indicates the occupancy of the next 3 farther neighboring child nodes as illustrated in Figure 16.

OccAdvNeiPat0[9] = 1  
OccAdvNeiPat0[10] = occBottom & 1

OccAdvNeiPat0[11] = occFront & 1

OccAdvNeiPat0[12] = occLeft & 1

图片包含 图标

描述已自动生成

**Figure 16 — Example of farther neighbouring child nodes**

* + Otherwise,
    - *OccAdvNeiPat0*[9] is set as 0;
    - *OccAdvNeiPat0*[10] indicates whether the 3 edge sub-sets of two farther neighboring nodes of current child node *b0* is occupied or not;

OccAdvNeiPat0[10] = edgeSets[5] || edgeSets[4] || edgeSets[3] || edgeSets[2] ||  
 edgeSets[1] || edgeSets[0]

* + - *OccAdvNeiPat0*[11]*OccAdvNeiPat0*[12] is calculated as follows.

OccAdvNeiPat0[11] = (occLeft & 4) || (occFront & 2) || (occBottom & 4)  
OccAdvNeiPat0[12] = (occLeft & 2) || (occFront & 16) || (occBottom & 16)

* *OccAdvNeiPat0*[13]…*OccAdvNeiPat0*[15] is calculated as follows.

OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] = neighb20[18]   
neighb20[19]neighb20[11]

* Otherwise, *OccAdvNeiPat0*[9]…*OccAdvNeiPat0*[15] is calculated as follows.

OccAdvNeiPat0[9] = !(edgeSets[5]edgeSets[4])

OccAdvNeiPat0[10] = !(edgeSets[3]edgeSets[2])

OccAdvNeiPat0[11] = !(edgeSets[1]edgeSets[0])  
OccAdvNeiPat0[12] = 0

OccAdvNeiPat0[13]OccAdvNeiPat0[14]OccAdvNeiPat0[15] = neighb20[18]   
neighb20[19]neighb20[11]

##### Derivation of OccAdvNeiPat1

The expression *OccAdvNeiPat1* specifies advanced occupied neighbourhood pattern constructed to code the second child node *b1*of the current node.

The two neighbouring child nodes sets specified by *occL* and *occF* for coding the second child node *b1*is illustrated in Figure 17.

图片包含 游戏机, 建筑, 窗户

描述已自动生成

**Figure 17 Two sets of neighboring child nodes for** **b1**

*OccAdvNeiPat1* is derived as follows.

* *OccAdvNeiPat1*[0] is set as the occupancy of the first child node *b0*.
* If the nerighouring child nodes set specified by *occF* is occupied, the occupied neighbour nodes of child node *b1* are not sparse.

OccAdvNeiPat1 = (b0 << 18) | (!(occF & 2) << 17) | (!occL << 16) | (occL ? AdvNeiFL :   
 AdvNeiFNL) | neighb20[11] << 2 | neighb20[16] << 1 | neighb20[19]

where   
AdvNeiFL := ((occL & 2) << 15) | ((neighPatternRBT & 4) << 14) | (!(occF & 1)  
 << 13) | (!(occF & 8) << 12) | (!(occL & 1) << 11) | (!(occL & 8) << 10) |  
 (!(occF & 4) << 9) | (!(occL & 4) << 8) | ((neighPatternRBT & 1) << 7) |  
 neighb20[9] << 6 | neighb20[4] << 5 | neighb20[1] << 4 | neighb20[2] << 3  
AdvNeiFNL := (!(neighPatternRBT & 4) << 15) | (!(occF & 1) << 14) | (!(occF &  
 8) << 13) | (!(occF & 4) << 12) | neighb20[9] << 11 | neighb20[4] << 10 |  
 neighb20[1] << 9 | neighb20[2] << 8 | (!(occBottom & 2) << 7) | (!(occFront &  
 2) << 6) | (!(occLeft & 2) << 5) | ((neighPatternRBT & 3) << 3)

* Otherwise, the occupied neighbour nodes of child node *b1* are sparse.

OccAdvNeiPat1 = (b0 << 18) | (!(occL & 2) << 17) | ((neighPatternRBT & 4) << 16) | (!(occL & 1) << 15) | (!(occL & 8) << 14) | (!(occL & 4) << 13) | ((neighPatternRBT  
& 1) << 12) | neighb20[1] << 11 | neighb20[4] << 10 | neighb20[9] << 9 | neighb20[2]  
<< 8 | (occOfLFB & 2 ? AdvBFL : AdvELF) | (!(occB) << 3) | neighb20[11] << 2 |  
neighb20[16] << 1 | neighb20[19]

where   
AdvBFL := (1 << 7) | (!(occBottom & 2) << 6) | (!(occFront & 2) << 5) |   
 ((occLeft & 2) << 4)

AdvELF := (!(edgeSets & 53) << 6) | (((occLeft & 8) || (occFront & 32)) << 5) |   
(((occLeft & 1) || (occFront & 1)) << 4)

##### Derivation of OccAdvNeiPat2

The expression *OccAdvNeiPat2* specifies advanced occupied neighbourhood pattern constructed to code the third child node *b2*of the current node. *OccAdvNeiPat2* is derived as follows.

* *OccAdvNeiPat2*[0] is set as the occupancy of the first child node *b0*.
* If the nerighouring child nodes set specified by *occB* is occupied, the occupied neighbour nodes of child node *b2* are not sparse.

OccAdvNeiPat2 = (b0 << 18) | (!(occB & 2) << 17) | (!occL << 16) | (occL ? AdvNeiBL :  
 AdvNeiBNL) | neighb20[0] << 3 | neighb20[18] << 2 | neighb20[19] << 1 | neighb20[11]

where   
AdvNeiBL := (!(occL & 4) << 15) | (!(neighPatternRBT & 2) << 14) | (!b1 << 13)  
 | (!(occB & 8) << 12) | (!(occL & 8) << 11) | (!(occL & 1) << 10) | (!(occB &  
 1) << 9) | neighb20[10] << 8 | neighb20[6] << 7 | neighb20[3] << 6 | (!(occB &  
 4) << 5) | (!(occL & 2) << 4)  
AdvNeiBNL := (!(neighPatternRBT & 4) << 15) | (!(occF & 1) << 14) | (!(occF &  
 8) << 13) | (!(occF & 4) << 12) | neighb20[9] << 11 | neighb20[4] << 10 |  
 neighb20[1] << 9 | neighb20[2] << 8 | (!(occBottom & 2) << 7) | (!(occFront &  
 2) << 6) | (!(occLeft & 2) << 5) | ((neighPatternRBT & 3) << 3)

* Otherwise, the occupied neighbour nodes of child node *b2* are sparse.

OccAdvNeiPat2 = (b0 << 18) | (!(occL & 4) << 17) | (!(neighPatternRBT & 2) << 16) |  
 (!(b0) << 15) | (!(occL & 8) << 14) | (!(occL & 1) << 13) | (!(occL & 2) << 12) |  
 neighb20[3] << 11 | neighb20[6] << 10 | neighb20[10] << 9 | neighb20[5] << 8 |  
 (occOfLFB & 4 ? AdvNeiLBF : AdvNeiLBE)

where   
AdvNeiLBF := (1 << 7) | (!(occLeft & 4) << 6) | (!(occBottom & 4) << 5) |  
 (!(occFront & 4) << 4)  
AdvNeiLBE := (((occLeft & 1) || (occBottom & 1)) << 6) | (((occLeft & 8) ||  
 (occBottom & 64)) << 5) | (!(edgeSets & 3) << 4)

##### Derivation of OccAdvNeiPat3

The expression *OccAdvNeiPat3* specifies advanced occupied neighbourhood pattern constructed to code the fourth child node *b3*of the current node. The expression *NN3* specifies the number of occupied child nodes in the set of neighbouring child nodes specified by *occL* and the set of coded child nodes (*b0*, *b1*, *b2*). *OccAdvNeiPat3* is derived as follows.

* If *NN3* is larger than , then the occupied neighbour nodes of child node *b3* are not sparse.

OccAdvNeiPat3 = (!b2 << 16) | (!(b1) << 15) | (!(occL & 8) << 14) | (neighPatternRBT  
 << 11) | (!b0 << 10) | (!(occL & 4) << 9) | (!(occL & 2) << 8) | (occL & 1) << |  
 neighb20[11] << 6 | neighb20[6] << 5 | neighb20[4] << 4 | neighb20[0] << 3 |  
 neighb20[16] << 2 | neighb20[19] << 1 | neighb20[18]

* Otherwise, the occupied neighbour nodes of child node *b3* are sparse.

OccAdvNeiPat3 = (!((b2 << 2) | (b1 << 1) | b0) << 17) | (((b2 << 2) | (b1 << 1) |  
 b0) ? AdvNei3Occp : AdvNei3L) | ((neighPatternRBT & 6) << 13) | neighb20[4] << 12 |  
 neighb20[6] << 11 | neighb20[11] << 10 | neighb20[7] << 9 | (occOfLFB & 8 ?  
 AvdNei3BFL : AvdNei3LE) | (!occB << 4) | (!occF << 3) | neighb20[18] << 2 |  
 neighb20[19] << 1 | neighb20[16]

where   
AdvNei3Occp := (!!((b2 << 2) | (b1 << 1) | b0) + !!((b2 << 1) | b1) + !!b2) <<  
 15  
AdvNei3L := (!!(occL >> 1) + !!(occL >> 2) + !!(occL >> 3)) << 15  
AvdNei3BFL := (1 << 8) | (!(occBottom & 8) << 7) | (!(occFront & 8) << 6) |  
 (!(occLeft & 8) << 5)  
AvdNei3LE := ((occLeft & 6) << 6) | (!(edgeSets & 50) << 5)

##### Derivation of OccAdvNeiPat4

The expression *OccAdvNeiPat4* specifies advanced occupied neighbourhood pattern constructed to code the fifth child node *b4*of the current node.

The expression *occLeftChilds* specifies the set of the 4 child nodes *b0*, *b1*, *b2* and *b3*. *occRightChilds* specifies the set of another 4 child nodes *b4*, *b5*, *b6* and *b7*.

occLeftChilds = b3 << 3 | b2 << 2 | b1 << 1 | b0

occRightChilds = b7 << 3 | b6 << 2 | b5 << 1 | b4

The expression *NN4* specifies the number of occupied child nodes in the sets of neighbouring child nodes specified by *occL*, *occB* and *occLeftChilds*. *OccAdvNeiPat4* is derived as follows.

* If *NN4* is larger than , then the occupied neighbour nodes of child node *b4* are not sparse.
  + The expression *NLFB* specifies the number NLFB of occupied sets among the sets specified by *occF* and *occB*, *occLeftChilds*.

NLFB = !!occLeftChilds + !!occF + !!occB

* + *OccAdvNeiPat4* is derived as follows.

OccAdvNeiPat4 = 0  
OccAdvNeiPat4 |= (NLFB == 3 ? AdvNei4NLFB3 : NLFB == 2 ? AdvNei4NLFB2 :   
 AdvNei4NLFB1)

where   
AdvNei4NLFB3 := (8 << 15) | (!(occB & 4) << 17) | (!(occF & 4) << 16) |  
 ((occLeftChilds & 1) << 15) | (!(neighPatternRBT & 1) << 14) | (!(occB &  
 1) << 13) | (!(occB & 8) << 12) | (!(occF & 1) << 11) | (!(occF & 8) <<  
 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8) | (!(occB  
 & 2) << 7) | (!(occF & 2) << 6) | (!(neighPatternRBT >> 1) << 4) |  
 neighb20[15] << 3 | neighb20[13] << 2 | neighb20[8] << 2 | neighb20[12]  
AdvNei4NLFB2 := (occLeftChilds && occB ? AdvNei4BChild : occF && occB ?  
 AdvNei4FB : AdvNei4Lchild) | neighb20[15] << 5 | neighb20[13] << 4 |  
 neighb20[8] << 3 | neighb20[12] << 2 | neighb20[16] << 1 | neighb20[18]  
AdvNei4NLFB1 := (occLeftChilds ? AdvNei4Child : occF ? AdvNei4F : AdvNei4B)  
 | ((neighPatternRBT >> 1) << 6) | neighb20[15] << 5 | neighb20[13] << 4 |  
 neighb20[8] << 3 | neighb20[12] << 2 | neighb20[16] << 1 | neighb20[18]  
AdvNei4BChild := (4 << 15) | (!(occB & 4) << 14) | (!(occLeftChilds & 1) <<  
 13) | (!(neighPatternRBT & 1) << 12) | (!(occB & 1) << 11) | (!(occB & 8)  
 << 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8)) |  
 (!(occB & 2) << 7) | (!(occLeftChilds & 8) << 6)  
AdvNei4FB := (5 << 15) | (!(occB & 4) << 14) | (!(occF & 4) << 13) |  
 (!(neighPatternRBT & 1) << 12) | (!(occB & 1) << 11) | (!(occB & 8) << 10)  
 | (!(occF & 1) << 9) | (!(occF & 8) << 8) | (!(occB & 2) << 7) | (!(occF &  
 2) << 6)  
AdvNei4Lchild := (6 << 15) | (!(occF & 4) << 14) | (!(occLeftChilds & 1) <<  
 13) | (!(neighPatternRBT & 1) << 12) | (!(occF & 1) << 11) | (!(occF & 8)  
 << 10) | (!(occLeftChilds & 2) << 9) | (!(occLeftChilds & 4) << 8) |  
 (!(occF & 2) << 7) | (!(occLeftChilds & 8) << 6)  
AdvNei4Child := (0 << 15) | ((occLeftChilds & 1) << 14) |  
 (!(neighPatternRBT & 1) << 13) | ((occLeftChilds & 6) << 12) |  
 (!(occLeftChilds & 8) << 10) | ((edgeSets & 12) << 8)

* Otherwise, the occupied neighbour nodes of child node *b4* are sparse. *OccAdvNeiPat4* is derived as follows.

OccAdvNeiPat4=0

if (NN4 == 1){

if (NN4 == 1){

OccAdvNeiPat4 |= occLeftChilds ? ((1 << 14) | (!(occLeftChilds & 1) << 13) |  
 (!(neighPatternLFB & 4) << 12) | (!(neighPatternLFB & 2) << 11)) : occF ? ((2  
 << 14) | (!(occF & 1) << 13) | (!(neighPatternLFB & 4) << 12) |  
 (!(neighPatternLFB & 1) << 11)) : ((3 << 14) | (!(occB & 1) << 13) |  
 (!(neighPatternLFB & 2) << 12) | (!(neighPatternLFB & 1) << 11))

}  
else {

OccAdvNeiPat4 |= (0 << 14) | (neighPatternLFB << 11)

}

OccAdvNeiPat4 |= neighb20[8] << 10 | neighb20[13] << 9 | neighb20[15] << 8  
 | neighb20[12] << 7

OccAdvNeiPat4 |= (neighPatternLFB ? AdvNei4LFB : AdvNei4ELFB) | neighb20[16] << 2  
 | neighb20[18] << 1 | neighb20[19] << 0

}

where

AdvNei4ELFB := (!(edgeSets & 48) << 6) | (!(edgeSets & 12) << 5) | (!(edgeSets &  
 3) << 4)

AdvNei4LFB := (occOfLFB & 16) ? AdvNei4LFBfar : AdvNei4LFBfarer

AdvNei4LFBfar := (1 << 6) | (!(occBottom & 16) << 5) | (!(occFront & 16) << 4) |  
 (!(occLeft & 16) << 3)

AdvNei4LFBfarer := (!edgeSets << 5) | (((occLeft & 64) || (occFront & 8) ||  
 (occBottom & 8)) << 4) | (((occLeft & 32) || (occFront & 64) || (occBottom & 32))  
 << 3)

##### Derivation of OccAdvNeiPat5

The expression *OccAdvNeiPat5* specifies advanced occupied neighbourhood pattern constructed to code the fifth child node *b5*of the current node. *OccAdvNeiPat5* is derived as follows.

* If the nerighouring child nodes set specified by *occF* is occupied, the occupied neighbour nodes of child node *b5* are not sparse.

OccAdvNeiPat5 = (b4 << 18) | (!(occF & 8) << 17) | (!occLeftChilds << 16) |  
 (occLeftChilds ? AdvNei5occFChilds : AdvNei5fartherBFL) | neighb20[18] << 2 |  
 neighb20[19] << 1 | neighb20[11]

where  
AdvNei5occFChilds := (!(occLeftChilds & 2) << 15) | (!(neighPatternRBT & 4) <<  
 14) | (!(neighPatternRBT & 1) << 13) | (!(occF & 2) << 12) | (!(occF & 4) <<  
 11) | (!(occLeftChilds & 1) << 10) | (!(occLeftChilds & 8) << 9) | (!(occF &  
 1) << 8) | (!(occLeftChilds & 4) << 7) | neighb20[16] << 6 | neighb20[13] << 5  
 | neighb20[9] << 4 | neighb20[14]   
AdvNei5fartherBFL := (!(neighPatternRBT & 4) << 15) | (!(neighPatternRBT & 1)   
 << 14) | (!(occF & 2) << 13) | (!(occF & 4) << 12) | (!(occF & 1) << 11) |  
 neighb20[16] << 10 | neighb20[13] << 9 | neighb20[9] << 8 | neighb20[14] << 7  
 | (!(occBottom & 32) << 6) | (!(occFront & 32) << 5) | (!(occLeft & 32) << 4)  
 | (!(neighPatternRBT & 2) << 3)

* Otherwise, the occupied neighbour nodes of child node *b5* are sparse.

OccAdvNeiPat5 = (!b4 << 18) | (!(occLeftChilds & 2) << 17) | (!(neighPatternRBT & 4)  
 << 16) | (!(neighPatternRBT & 1) << 15) | (!(occLeftChilds & 1) << 14) |  
 (!(occLeftChilds & 8) << 13) | (!(occL & 4) << 12) | neighb20[9] << 11 |neighb20[13]  
 << 10 |neighb20[16] << 9 | neighb20[14] << 8 | (occOfLFB & 32 ? AdvNei5BFL :  
 AdvNei5EFL) | (!occB << 3) | neighb20[18] << 2 | neighb20[19] << 1 | neighb20[11]

Where

AdvNei5BFL := (1 << 7) | (!(occBottom & 32 ) << 6) | (!(occFront & 32) << 5) |  
 (!(occLeft & 32) << 4)  
 AdvNei5EFL := (!(edgeSets & 60) << 6) | (((occLeft & 128) || (occFront & 2)) <<  
 5) | (((occLeft & 16) || (occFront & 16)) << 4)

##### Derivation of OccAdvNeiPat6

The expression *OccAdvNeiPat6* specifies advanced occupied neighbourhood pattern constructed to code the fifth child node *b6*of the current node. *OccAdvNeiPat6* is derived as follows.

* If the nerighouring child nodes set specified by *occB* is occupied, the occupied neighbour nodes of child node *b6* are not sparse.

OccAdvNeiPat6 = (b4 << 18) | (!(occB & 8) << 17) | (!occLeftChilds << 16) |  
 (occLeftChilds ? AdvNei6occBChilds : AdvNei6occBLFB)

where  
AdvNei6occBChilds := (!(occLeftChilds & 4) << 15) | (!(neighPatternRBT & 1) << 14) |  
 (!(neighPatternRBT & 2) << 13) | (!b5 << 12) | (!(occB & 2) << 11) |  
 (!(occLeftChilds & 1) << 10) | (!(occLeftChilds & 8) << 9) | (!(occB & 4) << 8) |  
 neighb20[18] << 7 | neighb20[15] << 6 | neighb20[10] << 5 | (!(occB & 1) << 4) |  
 (!(occLeftChilds & 2) << 3) | neighb20[17] << 2 | neighb20[0] << 1 | neighb20[11]  
AdvNei6occBLFB := (!(neighPatternRBT & 4) << 15) | (!(neighPatternRBT & 1) << 14) |  
 (!(occF & 2) << 13) | (!(occF & 4) << 12) | (!(occF & 1) << 11) |  
 neighb20[16] << 10 | neighb20[13] << 9 | neighb20[9] << 8 | neighb20[14] << 7  
 | (!(occBottom & 32) << 6) | (!(occFront & 32) << 5) | (!(occLeft & 32) << 4)  
 | (!(neighPatternRBT & 2) << 3)

* Otherwise, the occupied neighbour nodes of child node *b6* are sparse.

OccAdvNeiPat6 = (!b4 << 18) | (!(occLeftChilds & 4) << 17) | (!(neighPatternRBT & 1) <<  
 16) | (!b5 << 15) | (!(occLeftChilds & 8) << 14) | (!(occLeftChilds & 1) << 13) |  
 (!(occLeftChilds & 2) << 12) | neighb20[17] << 11 | neighb20[18] << 10 | neighb20[15] <<  
 9 | neighb20[10] << 8 | (occOfLFB & 64 ? AdvNei6LBF : AdvNei6LBFE) | (!occF << 3) |  
 neighb20[19] << 2 | neighb20[16] << 1 | neighb20[11]

where

AdvNei6LBF := (1 << 7) | (!(occLeft & 64) << 6) | (!(occBottom & 64) << 5) |  
 (!(occFront & 64) << 4)

AdvNei6LBFE := (((occLeft & 1) || (occBottom & 1)) << 6) | (((occLeft & 8) ||  
 (occBottom & 64)) << 5) | (!(edgeSets & 3) << 4)

##### Derivation of OccAdvNeiPat7

The expression *OccAdvNeiPat7* specifies advanced occupied neighbourhood pattern constructed to code the fourth child node *b7*of the current node. The expression *NN7* specifies the number of occupied child nodes in the set of neighbouring child nodes specified by *occLeftChilds* and *occRightChilds*. *OccAdvNeiPat7* is derived as follows.

[Ed. (YZ): since b7 is not reconstructed, not sure how to determine occRightChilds. Need to further check with proponent.]

* If *NN7* is larger than , then the occupied neighbour nodes of child node *b7* are not sparse.

OccAdvNeiPat7 = (!b6 << 16) | (!b5 << 15) | (!(occLeftChilds & 8) << 14) |  
 (neighPatternRBT << 11) | (!b4 << 10) | neighb20[11] << 9 | (!(occLeftChilds & 4) << 8) |  
 neighb20[16] << 7 | (!(occLeftChilds & 2) << 6) | neighb20[18] << 5 | ((occLeftChilds &  
 1) << 4) | neighb20[19] << 3 | neighb20[0] << 2 | neighb20[17] << 1 | neighb20[10]

* Otherwise, the occupied neighbour nodes of child node *b7* are sparse.

OccAdvNeiPat7 = (!occup << 17) | (occup ? AdvNei7occRight : AdvNei7occLeft) |  
 (!(neighPatternRBT & 4) << 13) | neighb20[11] << 12 | neighb20[16] << 11 | neighb20[18]  
 << 10 | neighb20[19] << 9 | (occOfLFB & 128 ? AdvNei7LFB : AdvNei7LoccFoccB) | (!occB <<  
 4) | (!occF << 3) | neighb20[7] << 2 | neighb20[17] << 1 | neighb20[10]

where  
occup = occRightChilds & 7  
AdvNei7occRight := ((!!occup + !!(occup >> 1) + !!(occup >> 2)) << 15) | (!(neighPatternRBT & 2) << 14)  
AdvNei7occLeft := (!!(occRightChilds >> 1) + !!(occRightChilds >> 2)  
 + !!(occRightChilds >> 3) << 15) | (!(neighPatternRBT & 1) << 14)  
AdvNei7LFB := (1 << 8) | (!(occLeft & 128) << 7) | (!(occFront & 128) << 6)  
 | (!(occBottom & 128) << 5)  
AdvNei7LoccFoccB := ((occLeft & 96) << 1) | (((occF & 3) || (occB & 6)) << 5)

### Neighbourhood-permuted node occupancy bitmap

The neighbourhood-permuted node occupancy bitmap is a rearrangement of the bits forming the node occupancy bitmap. It is used in the coding of occupancy\_byte and occupancy\_bit. The permutation shall be selected according to the occupied neighbourhood pattern.

The permutations for every occupied neighbourhood pattern are specified by Table 16. Each entry is a base eight value with digits numbered from right to left. The 𝑖-th digit is the bit position in the node occupancy bitmap of the 𝑖-th bit in the neighbourhood-permuted bitmap as specified by the expression OccBitIdxFromNpBit[ 𝑖 ].

OccBitIdxFromNpBit[i] := (OccArrangement[OccNeighPat] >> i × 3) & 7

The expression OccFromNpOcc[ npocc ] is the node occupancy bitmap derived from the neighbourhood-permuted node occupancy bitmap npocc.

OccFromNpOcc[npocc] :=  
 OccFromNpOcc = 0  
 for (i = 0; i < 8; i++)  
 OccFromNpOcc |= Bit(npocc, i) << OccBitIdxFromNpBit[i]

Two example derivations of OccupancyMap from a single occupancy\_byte by OccFromNpOcc are illustrated in Figure 9. Each derivation uses a different occupied neighbourhood pattern, OccNeighPat. Bit of occupancy\_byte is permuted to bit of OccupancyMap when OccNeighPat is 17; in this case, OccBitIdxFromNpBit[ 4 ] would be 1.

occupancy\_bit codes the bits of the neighbourhood-permuted node occupancy bitmap in a different order. i.e. occupancy\_bit[ 𝑖 ] does not correspond to bit of occupancy\_byte.

图示, 工程绘图

描述已自动生成

Figure 9 — Example relationships between the node occupancy bitmap OccupancyMap and neighbourhood-permuted node occupancy bitmap as coded by occupancy\_byte.

Table 16 — Arrangements for neighbourhood-permuted node occupancy bitmaps by occupied neighbourhood pattern as OccArrangement[ 𝑖 + 𝑗 ]

| 𝑖 | 𝑗 | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| **0** | 765432108 | 103254768 | 765432108 | 765432108 | 541076328 | 103254768 |
| **6** | 541076328 | 327610548 | 327610548 | 327610548 | 765432108 | 541076328 |
| **12** | 327610548 | 765432108 | 103254768 | 765432108 | 260437158 | 203164758 |
| **18** | 465702138 | 571346028 | 041526378 | 012345678 | 450167238 | 236701458 |
| **24** | 627340518 | 236701458 | 674523018 | 450167238 | 735162408 | 674523018 |
| **30** | 012345678 | 674523018 | 371526048 | 312075648 | 574613028 | 460257138 |
| **36** | 150437268 | 103254768 | 541076328 | 327610548 | 736251408 | 327610548 |
| **42** | 765432108 | 541076328 | 624073518 | 765432108 | 103254768 | 765432108 |
| **48** | 371526048 | 647520318 | 021346578 | 574613028 | 263704158 | 574613028 |
| **54** | 736251408 | 135702468 | 405162738 | 150437268 | 312075648 | 753164208 |
| **60** | 736251408 | 371526048 | 517340628 | 765432108 |  | |

### Dictionary coding of occupancy\_byte

#### General

The occupancy\_byte syntax element shall be coded as symbols by one of nine instances of this dictionary codec. Coding shall proceed according to the syntax and semantics of the occupancy\_byte\_symbol syntax structure.

Each dictionary instance comprises a list of thirty-two most probable symbols (occupancy\_byte values), a list of sixteen recently coded symbols, a histogram of symbol counts and state variables used to control updates to the dictionary state.

#### Syntax of a dictionary coded symbol

|  |  |
| --- | --- |
| occupancy\_byte\_symbol( ) { | Descriptor |
| occ\_histogram\_hit | ae(v) |
| if( occ\_histogram\_hit ) |  |
| occ\_histogram\_index | ae(v) |
| else { |  |
| occ\_recent\_hit | ae(v) |
| if( occ\_recent\_hit ) |  |
| occ\_recent\_index | ae(v) |
| else |  |
| occ\_symbol\_escape | ae(v) |
| } |  |
| } |  |

#### Syntax element semantics of a dictionary coded symbol

occ\_histogram\_hit specifies whether (when 1) or not (when 0) the coded symbol is present in the list of most probable symbols.

occ\_histogram\_index specifies the index of the coded symbol in the list of most probable symbols.

occ\_recent\_hit specifies whether (when 1) or not (when 0) the coded symbol is present in the most recently coded symbol list. When occ\_recent\_hit is not present, it shall be inferred to be 0.

occ\_recent\_index specifies the index of the coded symbol in the most recently coded symbol list.

occ\_symbol\_escape specifies the value of the decoded symbol when occ\_histogram\_hit and occ\_recent\_hit are both 0.

#### State variables

The dictionary codec is specified in terms of the following state variables; the index dictIdx identifies an instance of the dictionary codec:

* A 9×256 element array DictsHistogram of symbol occurrence histograms per dictionary instance; DictsHistogram[ dictIdx ][ sym ] is the cumulative count for the symbol sym.
* A 9×32 element array DictsMostProb of thirty-two most probable symbols per dictionary instance; DictsMostProb[ dictIdx ][ 𝑖 ] is the 𝑖-th most probable symbol.
* A 9×16 element array DictsRecent of sixteen recently coded symbols per dictionary instance; DictsRecent[ dictIdx ][ 𝑖 ] is a recently coded symbol that was not coded using the most probable symbols list.
* A 9 element array DictsMostProbAge; DictsMostProbAge[ dictIdx ] is the count of symbols since the last generation of the dictionary's most probable symbol list.
* A 9 element array DictsMostProbMaxAge; DictsMostProbMaxAge[ dictIdx ] is the maximum allowed age in symbols of the dictionary's most probable symbol list.
* A 9 element array DictsNextEvictIdx; DictsNextEvictIdx[ dictIdx ] is the index of the next element to be evicted from the dictionary's DictsRecent array.

#### Initial state

The dictionary state shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed by the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), the dictionary state variables shall be initialized:

* Elements of DictsMostProb shall be initialized according to Table 17.
* Elements of DictsHistogram shall be set to 1 if they identify a symbol present in the corresponding most probable symbol sub-array. i.e. DictsHistogram[ dictIdx ][ 𝑖 ] = 1 if 𝑖 ∈ DictsMostProb[ dictIdx ]. All other elements shall be set to 0.
* Elements of DictsRecent, DictsRecent[ dictIdx ][ 𝑖 ], shall be set to 𝑖.
* Elements of DictsNextEvictIdx and DictsMostProbAge shall be set to 0.
* Elements of DictsMostProbMaxAge shall be set to 16.

if (¬slice\_entropy\_continuation && ¬slice\_inter \_entropy\_continuation) {  
 … /\* Initialize DictsMostProb using Table 17 \*/  
  
 for (dictIdx = 0; dictIdx < 9; dictIdx++) {  
 for (i = 0; i < 16; i++)  
 DictsRecent[dictIdx][i] = i  
  
 for (i = 0; i < 32; i++) {  
 symbol = DictsMostProb[dictIdx][i]  
 DictsHistogram[dictIdx][symbol] = 1  
 }  
  
 DictsNextEvictIdx[dictIdx] = 0  
 DictsMostProbAge[dictIdx] = 0  
 DictsMostProbMaxAge[dictIdx] = 16  
 }  
}

Table 17 — Initial values of DictsMostProb[ dictIdx ][ 𝑖 ]

| 𝑖 | dictIdx | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **0** | 5 | 85 | 128 | 64 | 85 | 16 | 170 | 170 | 255 |
| **1** | 17 | 255 | 32 | 128 | 170 | 64 | 10 | 255 | 223 |
| **2** | 34 | 170 | 64 | 192 | 255 | 17 | 42 | 128 | 239 |
| **3** | 68 | 64 | 16 | 204 | 119 | 80 | 8 | 160 | 251 |
| **4** | 160 | 80 | 192 | 136 | 127 | 128 | 138 | 136 | 127 |
| **5** | 136 | 252 | 80 | 68 | 254 | 68 | 15 | 168 | 247 |
| **6** | 12 | 223 | 160 | 170 | 87 | 32 | 255 | 204 | 119 |
| **7** | 80 | 84 | 48 | 200 | 223 | 85 | 2 | 240 | 253 |
| **8** | 192 | 117 | 68 | 85 | 95 | 81 | 14 | 250 | 63 |
| **9** | 21 | 68 | 8 | 196 | 117 | 84 | 136 | 192 | 191 |
| **10** | 10 | 247 | 136 | 255 | 245 | 192 | 11 | 238 | 221 |
| **11** | 48 | 221 | 176 | 4 | 213 | 48 | 175 | 162 | 254 |
| **12** | 3 | 4 | 2 | 8 | 247 | 51 | 32 | 234 | 238 |
| **13** | 170 | 192 | 240 | 80 | 93 | 4 | 238 | 223 | 95 |
| **14** | 168 | 128 | 144 | 160 | 234 | 34 | 47 | 138 | 175 |
| **15** | 162 | 174 | 17 | 240 | 69 | 240 | 191 | 254 | 240 |
| **16** | 204 | 253 | 208 | 208 | 238 | 1 | 34 | 10 | 85 |
| **17** | 85 | 204 | 224 | 76 | 21 | 136 | 239 | 186 | 187 |
| **18** | 14 | 240 | 112 | 221 | 221 | 170 | 245 | 8 | 244 |
| **19** | 81 | 69 | 19 | 140 | 191 | 255 | 174 | 251 | 250 |
| **20** | 35 | 127 | 255 | 244 | 253 | 204 | 3 | 2 | 170 |
| **21** | 69 | 213 | 85 | 72 | 187 | 196 | 63 | 127 | 245 |
| **22** | 84 | 5 | 51 | 93 | 16 | 160 | 95 | 125 | 117 |
| **23** | 176 | 119 | 170 | 168 | 251 | 12 | 223 | 247 | 34 |
| **24** | 51 | 238 | 84 | 250 | 171 | 208 | 253 | 85 | 126 |
| **25** | 65 | 175 | 162 | 32 | 17 | 69 | 168 | 171 | 51 |
| **26** | 138 | 160 | 238 | 252 | 5 | 191 | 142 | 32 | 93 |
| **27** | 200 | 87 | 204 | 187 | 174 | 119 | 246 | 197 | 243 |
| **28** | 212 | 136 | 1 | 223 | 125 | 21 | 206 | 221 | 207 |
| **29** | 11 | 16 | 76 | 238 | 239 | 95 | 13 | 87 | 234 |
| **30** | 50 | 244 | 138 | 243 | 12 | 2 | 162 | 42 | 59 |
| **31** | 15 | 23 | 187 | 84 | 241 | 206 | 250 | 239 | 236 |

#### Selection of a dictionary instance

A dictionary instance shall be selected for each coded occupancy\_byte syntax element according to the reduced occupied neighbourhood pattern, OccNeighPatR.

The following expressions are aliases used in the specification of operations on the selected dictionary instance:

OccDictHistogram[i] := DictsHistogram[OccNeighPatR][i]  
  
OccDictMostProb[i] := DictsMostProb[OccNeighPatR][i]  
  
OccDictRecent[i] := DictsRecent[OccNeighPatR][i]  
  
OccDictMostProbAge := DictsMostProbAge[OccNeighPatR]  
  
OccDictMostProbMaxAge := DictsMostProbMaxAge[OccNeighPatR]  
  
OccDictNextEvictIdx := DictsNextEvictIdx[OccNeighPatR]

#### The value for occupancy\_byte

The decoded value of the syntax element shall be:

* when occ\_histogram\_hit is 1: OccDictMostProb[ occ\_histogram\_index ];
* when occ\_recent\_hit is 1: OccDictRecent[ occ\_recent\_index ];
* otherwise (neither occ\_histogram\_hit nor occ\_recent\_hit is 1): occ\_symbol\_escape.

#### Update of dictionary state after each coded symbol

##### List of most recently coded symbols

After each coded occupancy\_byte syntax element when occ\_histogram\_hit is 0, the syntax element value shall be used to update the list of most recently coded symbols.

If the syntax element value is already present in the list, its position in the list shall be exchanged with the symbol scheduled to be evicted (at index OccDictNextEvictIdx).

for (i = 0; i < 16; i++)  
 if (OccDictRecent[i] == occupancy\_byte) {  
 OccDictRecent[i] = OccDictRecent[OccDictNextEvictIdx]  
 break  
 }

The syntax element value shall be inserted into the list, replacing the symbol at index OccDictNextEvictIdx.

OccDictRecent[OccDictNextEvictIdx] = occupancy\_byte

After updating the list of most recently coded symbols, the eviction index shall be incremented modulo 16.

OccDictNextEvictIdx = (OccDictNextEvictIdx + 1) % 16

##### Histogram of symbol counts

After each coded occupancy\_byte syntax element, the histogram of symbol occurrences shall be updated and the age of the most probable symbol list shall be incremented.

OccDictHistogram[occupancy\_byte]++  
OccDictMostProbAge++

When the histogram count of symbols equal to occupancy\_byte reaches 1 024, all counts in the histogram shall be halved and any fractional parts discarded.

if (OccDictHistogram[occupancy\_byte] == 1024)  
 for (i = 0; i < 256; i++)  
 OccDictHistogram[i] >>= 1

#### Generation of the most probable symbol list

When OccDictMostProbAge is equal to OccDictMostProbMaxAge, the most probable symbol list shall be recalculated from the histogram of symbol counts.

The most probable symbol list shall be a stable descending ordering of the OccDictHistogram array. Each element OccDictMostProb[ 𝑖 ] shall be the index of the 𝑖-th largest element in the OccDictHistogram array. The ordering shall be such that the following conditions are true:

* OccDictHistogram[ OccDictMostProb[ 𝑖 ] ] ≥ OccDictHistogram[ OccDictMostProb[ 𝑖 + 1 ] ], and
* OccDictMostProb[ 𝑖 ] < OccDictMostProb[ 𝑖 + 1 ] when OccDictHistogram[ OccDictMostProb[ 𝑖 ] ] is equal to OccDictHistogram[ OccDictMostProb[ 𝑖 + 1 ] ].

The age of the generated most probable symbol list shall be 0.

OccDictMostProbAge = 0

The maximum age of the generated most probable symbol list shall be the next value in the bounded geometric progression specified by the expression OccDictMostProbMaxAgeNext:

OccDictMostProbMaxAgeNext := Min(5 × OccDictMostProbMaxAge >> 2, 1024)  
OccDictMostProbMaxAge = OccDictMostProbMaxAge

#### Resetting the histogram of symbol counts

The histogram of symbol counts shall be reset immediately after the first recalculation of the most probable symbol list in each level of the occupancy tree. Counts for symbols that are present in the most probable symbol list shall be set to 1; all other counts shall be set to 0.

if (OccDictMostProbAge == 0)  
 if (… /\* First occurrence in tree level \*/) {  
 for (i = 0; i < 256; i++)  
 OccDictHistogram[i] = 0  
  
 for (i = 0; i < 32; i++) {  
 symbol = OccDictMostProb[i]  
 OccDictHistogram[symbol] = 1  
 }  
 }

#### Determination of CtxIdxDictHg for a bin of occ\_histogram\_index

Contextualization depends upon the reduced neighbourhood pattern, the bin index and the MSBs of the occ\_histogram\_index syntax element value.

Table 18 specifies the value ctxIdxInc for the bin. If the ctxIdxInc is 'bypass', the value of CtxIdxDictHg shall be 'bypass'. Otherwise, the value of CtxIdxDictHg shall be 5 × OccNeighPatR + ctxIdxInc.

Table 18 — Values of ctxIdxInc for bins of the syntax element occ\_histogram\_index

| MSBs of binarized occ\_histogram\_index | BinIdx | | | | |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| '000' | 0 | 1 | 2 | 3 | 4 |
| '001' | 0 | 1 | 2 | bypass | bypass |
| '01' | 0 | 1 | bypass | bypass | bypass |
| '1' | 0 | bypass | bypass | bypass | bypass |

### Bitwise occupancy coding

#### General

Subclause 9.2.10 applies when occtree\_bitwise\_coding is 1.

The neighbourhood-permuted node occupancy bitmap shall be coded as a sequence of individual occupancy\_bit syntax elements. Coding uses constraints on occupancy to infer the value of certain occupancy\_bit syntax elements.

Entropy coding of each coded bit is contextualized by a combination of the coded bit index, previously coded occupancy\_bit syntax elements, the reduced occupied neighbourhood pattern, the number of spatially adjacent child nodes in neighbouring nodes and a ternary prediction based upon the presence of neighbouring nodes.

#### Correspondence between the node occupancy bitmap and occupancy\_bit

Bits of the neighbourhood-permuted node occupancy bitmap shall be coded in the order specified by Table 19. Each occupancy\_bit[ cbIdx ] syntax element codes the bit OccBitCodingOrder[ cbIdx ].

The expression OccBitIdx[ cbIdx ] is the bit position in the node occupancy bitmap of the bit coded by occupancy\_bit[ cbIdx ]. For example, when OccNeighPat is 17, occupancy\_bit[ 6 ] corresponds to the second bit () of the node occupancy bitmap.

OccBitIdx[cbIdx] := OccBitIdxFromNpBit[OccBitCodingOrder[cbIdx]]

The expression OccBitLocC[ cbIdx ][ 𝑘 ] is the node-relative child location represented by occupancy\_bit[ cbIdx ].

OccBitLocC[cbIdx][k] := OccLocC[OccBitIdx[cbIdx]][k]

The expression OccBitMap is the node occupancy bitmap.

OccBitMap :=  
 OccBitMap = 0  
 for (cbIdx = 0; cbIdx < 8; cbIdx++)  
 OccBitMap = OccBitMap | (occupancy\_bit[cbIdx] << OccBitIdx[cbIdx])

Table 19 — Order for coding bits of the neighbourhood-permuted node occupancy bitmap as occupancy\_bit[ cbIdx ]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| cbIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| OccBitCodingOrder[ cbIdx ] | 1 | 7 | 5 | 3 | 2 | 6 | 4 | 0 |

#### Presence of occupancy\_bit

An occupancy bitmap bit shall not be coded if its value can be inferred to be set or unset. The expression OccBitPresent[ cbIdx ] specifies whether occupancy\_bit[ cbIdx ] is present.

OccBitPresent[cbIdx] := ¬(OccBitInferUnset[cbIdx] || OccBitInferSet[cbIdx])

#### Inference of an unset bit

An occupancy bitmap bit shall be inferred to be 0 when either:

* the bit represents an invalid child according to the node's coded axes (9.2.2.3), or
* the bit represents a child within an unoccupied plane signalled by planar occupancy coding.

When the expression OccBitInferUnset[ cbIdx ] is equal to 1, occupancy\_bit[ cbIdx ] shall be inferred to be 0.

OccBitInferUnset[cbIdx] :=  
 ¬AxisCoded[0] && OccBitLocC[cbIdx][0]  
 || ¬AxisCoded[1] && OccBitLocC[cbIdx][1]  
 || ¬AxisCoded[2] && OccBitLocC[cbIdx][2]  
 || ¬PlanarFreeAxis[0] && OccBitLocC[cbIdx][0] ^ occ\_plane\_pos[0]  
 || ¬PlanarFreeAxis[1] && OccBitLocC[cbIdx][1] ^ occ\_plane\_pos[1]  
 || ¬PlanarFreeAxis[2] && OccBitLocC[cbIdx][2] ^ occ\_plane\_pos[2]

#### Inference of a set bit

An occupancy bitmap bit occupancy\_bit[ cbIdx ] shall be inferred to be 1, as specified by the expression OccBitInferSet[ cbIdx ], when:

* the bit is the last present bit and all previous coded bits are 0, or
* the bit is the penultimate present bit, all previous coded bits are 0 and the node is required to have two child nodes, or
* the bit is in a plane identified as occupied by planar occupancy coding, the bit is the last bit in the plane and all previous bits in the plane are 0.

OccBitInferSet[cbIdx] :=  
 PlanarEligible[0] && PopCnt(OccKnownZero & (0x0F << 4 × OccBitLocC[cbIdx][0])) == 3  
 || PlanarEligible[1] && PopCnt(OccKnownZero & (0x33 << 2 × OccBitLocC[cbIdx][1])) == 3  
 || PlanarEligible[2] && PopCnt(OccKnownZero & (0x55 << 1 × OccBitLocC[cbIdx][2])) == 3  
 || cbIdx == 6 && PopCnt(OccKnown) == 0 && OccMinChildren == 2  
 || cbIdx == 7 && PopCnt(OccKnown) == 0

The expression OccKnownMask is a bit mask that identifies bits of the node occupancy bitmap that have a known value prior to coding occupancy\_bit[ cbIdx ].

OccKnownMask :=  
 OccKnownMask = 0  
 for (i = 0; i < cbIdx; i++)  
 OccKnownMask |= 1 << OccBitIdx[i]  
 for (i = 0; i < 8; i++)  
 OccKnownMask |= OccBitInferUnset[i] << OccBitIdx[i]

The expression OccKnown is the partially coded node occupancy bitmap comprising the bits coded prior to occupancy\_bit[ cbIdx ].

OccKnown :=  
 OccKnown = 0  
 for (i = 0; i < cbIdx; i++)  
 OccKnown |= occupancy\_bit[i] << OccBitIdx[i]

The expression OccKnownZero is a bitmap of occupancy bits that are known to be 0.

OccKnownZero := (0xFF ^ OccKnown) & OccKnownMask

#### Contextualization

##### General

Contextualization of occupancy\_bit syntax elements is a two-stage process. First, context discriminators are used to select a demi-CPM. Then, the demi-CPM is used to select the CPM that codes the syntax element.

A demi-CPM is an 8-bit unsigned integer that models the probability of a coded zero-valued occupancy\_bit syntax element.

The values 0, 128 and 256 represent the probability of a zero bin as impossible, equiprobable and certain, respectively. The values 0 and 256 can never be attained due to the operation of the probability models' update process.

##### State variables

Context selection is specified in terms of the following state variable:

* The array OccCtxSel; OccCtxSel[ selNeigh ][ cbIdx ][ selSib ][ selAdj ][ selPred ] is a demi-CPM, contextualized by selNeigh, cbIdx, selSib, selAdj and selPred.

##### Initial state

The demi-CPMs shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed according to the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), all elements of OccCtxSel shall be set to 127.

##### Determination of CtxIdxOccBit for the syntax element occupancy\_bit

The expression OccCtxSelVar specifies the demi-CPM for the syntax element occupancy\_bit[ CbIdx ] using:

* SelNeigh, the reduced occupied neighbourhood context discriminator (9.2.10.6.6);
* SelSib, the sibling occupancy context discriminator (9.2.10.6.7);
* SelAdj, the adjacent child neighbour context discriminator (9.2.10.6.8);
* SelPred, the neighbour-predicted occupancy context discriminator (9.2.10.6.9).

OccCtxSelVar := OccCtxSel[SelNeigh][CbIdx][SelSib][SelAdj][SelPred]

The CPM index, CtxIdxOccBit, shall be the value of the demi-CPM exclusive of the bottom three bits:

CtxIdxOccBit := OccCtxSelVar >> 3

##### Update after each coded occupancy\_bit syntax element

After each coded occupancy\_bit syntax element, its demi-CPM shall be updated. The update specified by Table 20 supplies a value for incrementing or decrementing the probability of a zero bin based upon the upper four bits of the demi-CPM's value:

if (OccBitPresent[CbIdx])  
 if (occupancy\_bit[CbIdx])  
 OccCtxSelVar += OccCtxSelUpdate[255 – OccCtxSelVar >> 4]  
 else  
 OccCtxSelVar −= OccCtxSelUpdate[OccCtxSelVar >> 4]

Table 20 — Values of OccCtxSelUpdate[ 𝑖 ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| OccCtxSelUpdate[ 𝑖 ] | 0 | 1 | 1 | 2 | 4 | 7 | 9 | 11 | 14 | 16 | 19 | 23 | 22 | 18 | 13 | 6 |

##### Reduced occupied neighbourhood context discriminator

The reduced occupied neighbourhood context discriminator shall distinguish between different reduced occupied neighbourhood patterns (OccNeighPatR) depending upon the coded bit index (CbIdx) as specified by Table 21 as the expression SelNeigh.

Table 21 — Discriminated values SelNeigh for occupancy\_bit[ CbIdx ] and OccNeighPatR

| CbIdx | OccNeighPatR | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 .. 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 4 .. 5 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 4 |
| 6 | 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

##### Sibling occupancy context discriminator

The sibling occupancy context discriminator shall distinguish between arrangements of previously coded/inferred siblings for the node coded by occupancy\_bit[ CbIdx ] as specified by the expression SelSib:

* If there are no nodes present in the occupied neighbourhood pattern, discrimination shall be by the number of present child nodes identified by the syntax elements occupancy\_bit[ 𝑖 ] with 𝑖 < CbIdx.
* If there is at least one node present in the occupied neighbourhood pattern, discrimination shall be by the combination of present child nodes identified by the syntax elements occupancy\_bit[ 𝑖 ] with 𝑖 < CbIdx.

SelSib := OccNeighPat ? occPrevBits : PopCnt(occPrevBits)

The expression occPrevBits is the concatenation of occupancy\_bit[ 𝑖 ] for 𝑖 < CbIdx.

occPrevBits :=  
 occPrevBits = 0  
 for (i = 0; i < CbIdx; i++)  
 occPrevBits |= occupancy\_bit[i] << i

##### Adjacent child neighbour context discriminator

The adjacent child neighbour context discriminator for child the node coded by occupancy\_bit[ CbIdx ] is specified by the expression SelAdj. When adjacent child neighbour contextualization is enabled (occtree\_adjacent\_child\_enabled is 1), it distinguishes between contexts by:

* the number of child nodes from available, previously coded nodes in the same tree level (9.2.7.2) that adjoin the coded child by a face; and
* whether any of the available, previously coded nodes in the same tree level that adjoin the coded child node do not have a child node that also adjoins the coded child.

An example is illustrated in Figure 10. The child node of the coded node N is adjoined by a single child node. There are two available previously coded nodes that adjoin , one of which does not contain a child node that also adjoins .

SelAdj := occtree\_adjacent\_child\_enabled  
 ? 2 × Min(2, adjCntC) + ((cbIdx ≤ 4 || adjCntC == 1) && adjUnocc)  
 : 0

The expression adjOccN[ 𝑘 ] identifies whether there is a spatially adjacent node along the 𝑘-th axis within the occupied neighbourhood availability window. Values for the expressions ds, dt and dv are specified in Table 22 for each axis 𝑘.

adjOccN[k] := ¬OccBitLocC[CbIdx][k] && OccNeigh[Ns + ds][Nt + dt][Nv + dv]

The expression adjOccC[ 𝑘 ] identifies whether there is a spatially adjacent child node along the 𝑘-th axis within the occupied neighbourhood availability window. Values for the expressions ds, dt and dv are specified in Table 22 for each axis, 𝑘.

adjOccC[k] := adjOccN[k] && OccNodePresent[Dpth + 1][cs + ds][ct + dt][cv + dv] ≠ 0  
 where  
 cs := NsC + OccBitLocC[CbIdx][0]  
 ct := NtC + OccBitLocC[CbIdx][1]  
 cv := NvC + OccBitLocC[CbIdx][2]

Table 22 — Relative neighbour locations ( ds, dt, dv ) used in the computation of adjOccN[ 𝑘 ] and adjOccC[ 𝑘 ]

| 𝑘 | ds | dt | dv |
| --- | --- | --- | --- |
| 0 | −1 | 0 | 0 |
| 1 | 0 | −1 | 0 |
| 2 | 0 | 0 | −1 |

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| N | Coded node |
|  | Contextualized child with OccBitIdx[ cbIdx ] = 0 |

Figure 10 — Example of adjacent child neighbour context discrimination.

The expressions adjCntN and adjCntC are the number of spatially adjacent nodes and child nodes, respectively, that are within the occupied neighbourhood availability window.

adjCntN := adjOccN[0] + adjOccN[1] + adjOccN[2]  
adjCntC := adjOccC[0] + adjOccC[1] + adjOccC[2]

The expression adjUnocc identifies whether there exists a spatially adjacent node within the occupied neighbourhood availability window that does not have a child node spatially adjacent to the coded child.

adjUnocc := adjCntN ≠ adjCntC

##### Neighbour-predicted-occupancy context discriminator

###### General

The neighbour-predicted-occupancy context discriminator shall, for eligible nodes (9.2.10.6.9.2), distinguish between three predictions for the presence of the child node coded by occupancy\_bit[ CbIdx ]. The discriminator is specified by the expression SelPred. The three predictions are that the node is present, not present, or that it is unpredictable.

SelPred := SelPredEligible ? OccIntraPred : 0

###### Eligibility

The discriminator shall only form a prediction for eligible nodes as specified by the expression SelPredEligible. Eligible nodes shall have both:

* three free axes and
* a maximum log2 node dimension less than occtree\_intra\_pred\_max\_nodesize\_log2.

SelPredEligible :=  
 OccFreeAxisCnt == 3 && MaxVec(NodeSizeLog2) < occtree\_intra\_pred\_max\_nodesize\_log2

###### Occupancy prediction

Occupancy prediction generates a ternary prediction for the presence of a child node identified by occupancy\_bit[ CbIdx ] of a coded node. The prediction is specified by the expression OccIntraPred. It is based upon how many of the nodes that neighbour the coded node also adjoin the volume of the identified child node by a face, edge or corner (as illustrated by Figure 11):

* A child node shall be predicted to be not present if there are two or fewer adjoining nodes.
* A child node shall be predicted to be present if there is at least a threshold number of adjoining nodes. The threshold is specified by the expression OccIntraThreshold. The threshold is four nodes unless there are more than 13 neighbouring nodes; in which case the threshold is 5 nodes.
* Otherwise, the presence is unpredictable.

The size of the child node volume is half the size of the neighbour nodes' in each dimension.

OccIntraPred := (OccAdjCnt ≤ 2) + 2 × (OccAdjCnt ≥ OccIntraThreshold)  
OccIntraThreshold := 4 + (OccNeighCnt ≥ 14)

图片包含 体育, 游戏机, 桌子

描述已自动生成

Figure 11 — Nodes that adjoin a child node by a face, edge or corner.

The expression OccAdj[ ds ][ dt ][ dv ] identifies whether a neighbouring node with a relative tree location ( ds, dt, dv ) to the coded node would adjoin the identified child volume.

OccAdj[ds][dt][dv] := (OccBitIdx[CbIdx] & adjMask) == adjLoc  
 where  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

The expression OccAdjCnt is the number of neighbours that adjoin the identified child volume.

OccAdjCnt := SumN26[neighAdj]  
 where  
 neighAdj[ds][dt][dv] := OccNeigh[Ns + ds][Nt + dt][Nv + dv] && OccAdj[ds][dt][dv]

The expression OccNeighCnt is the number of nodes that neighbour the coded node.

OccNeighCnt := SumN26[neighRel]  
 where  
 neighRel[ds][dt][dv] := OccNeigh[Ns + ds][Nt + dt][Nv + dv]

The expression SumN26[ expr ] sums the result of applying expr to the relative tree location of each of the 26 possible neighbouring nodes.

SumN26[expr] :=  
 SumN26 = 0  
 for (ds = −1; ds ≤ 1; ds++)  
 for (dt = −1; dt ≤ 1; dt++)  
 for (dv = −1; dv ≤ 1; dv++)  
 if (ds ≠ 0 && dt ≠ 0 && dv ≠ 0)  
 SumN26 += expr[ds][dt][dv]

##### Coding occupancy bits using OBUF

[Ed. (YZ): the implementation shall be further check to ensure it consistent with the spec text.]

The occupancy bits of octree nodes are coded as follows.

* A first buffer is created for all OBUF instances according to 12.3. The OBUF buffer size obufBufferSize is set as 20000, the buffer depth obufLeafDepth of fully deployed trees is set as 4.
* The array of OBUF ACPMs is created according to 12.2.2.
* All OBUF instances are created according to 12.2 by using the octree buffer created, and the OBUF instances are divided into two types, sparse OBUF instances and non-sparse OBUF instances. Each type of OBUF instances has specific sizes nBit1 and nBit2 to specify the sizes of contextual information info1 and info2 used as input when calling the OBUF instance.
* The sizes nBit1[j][i] and nBit2[j][i] for sparse OBUF instances are set according to Table 23, and the sizes nBit1[j][i] and nBit2[j][i] for non-sparse OBUF instances are set according to Table 24, where index i represents the child node in current coded node, and index j represents intra prediction method (j = 0) or inter prediction method (j = 1) for octree coding.

**Table 23**  —  **the size values (nBit1, nBit2) [ j][ i ] of sparse OBUF instance**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | i | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0** | (4,12) | (7,12) | (7,12) | (6,12) | (4,12) | (7,12) | (7,12) | (6,12) |
| **1** | (4,12) | (7,12) | (7,12) | (6,12) | (4,12) | (7,12) | (7,12) | (6,12) |

**Table 24**  — **the size values (nBit1, nBit2) [ j][ i ] of non-sparse OBUF instance**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | i | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0** | (6,13) | (6,13) | (6,13) | (6,11) | (6,13) | (6,13) | (6,13) | (6,11) |
| **1** | (6,13) | (6,13) | (6,13) | (6,11) | (6,13) | (6,13) | (6,13) | (6,11) |

* Optionally, for each child node, if the corresponding table among Table 25, Table 26, Table 27, Table 28, Table 29, Table 30, Table 31 and Table 32 is provided, the context array *ctxIdxMap*[][] of the created OBUF instances is initialized according to the provided table. Otherwise, for this child node, *ctxIdxMap*[][] is initialized with all the values set as 127.
* The array *ctxIdxMap*[ ][ ] corresponds to 8-bit context indices pointing (after right shift by 3) to OBUF ACPMs of the array *obufCtxArray*[ ].

Table 25 — Initial values of *ctxIdxMap*[ j][0] for child node under intra octree coding

| j | | *ctxIdxMap*[ j][0] | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 127 | | 17 | 82 | 38 | 127 | 105 | 141 | 81 |
| 8 .. 15 | 127 | | 15 | 45 | 43 | 116 | 105 | 152 | 115 |
| 16 .. 23 | 127 | | 53 | 21 | 20 | 127 | 127 | 127 | 37 |
| 24 .. 31 | 127 | | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 32 .. 39 | 171 | | 186 | 170 | 240 | 182 | 209 | 223 | 240 |
| 40 .. 47 | 44 | | 101 | 101 | 74 | 65 | 66 | 134 | 199 |
| 48 ..55 | 47 | | 27 | 141 | 113 | 126 | 61 | 240 | 151 |
| 56 ..63 | 45 | | 68 | 113 | 101 | 47 | 84 | 153 | 234 |

**Table 26 — Initial values of *ctxIdxMap*[ j][**1**]for child node under intra octree coding**

| j | *ctxIdxMap*[ j][ 1] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 240 | 240 | 222 | 240 | 175 | 181 | 127 | 127 |
| 8 .. 15 | 120 | 152 | 132 | 116 | 57 | 127 | 127 | 127 |
| 16 .. 23 | 105 | 185 | 127 | 87 | 105 | 116 | 65 | 69 |
| 24 .. 31 | 66 | 105 | 58 | 43 | 44 | 49 | 18 | 15 |
| 32 .. 39 | 228 | 240 | 138 | 240 | 178 | 198 | 114 | 152 |
| 40 .. 47 | 173 | 240 | 204 | 127 | 70 | 141 | 127 | 127 |
| 48 ..55 | 184 | 192 | 105 | 116 | 121 | 181 | 35 | 46 |
| 56 ..63 | 58 | 87 | 114 | 73 | 51 | 15 | 101 | 40 |

**Table 27 — Initial values of *ctxIdxMap*[ j][**2**]for child node under intra octree coding**

| j | *ctxIdxMap*[ j][ 2] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 194 | 240 | 173 | 190 | 115 | 129 | 87 | 87 |
| 8 .. 15 | 168 | 161 | 116 | 92 | 127 | 127 | 26 | 96 |
| 16 .. 23 | 160 | 106 | 96 | 127 | 86 | 109 | 105 | 127 |
| 24 .. 31 | 116 | 68 | 80 | 27 | 116 | 116 | 46 | 19 |
| 32 .. 39 | 240 | 240 | 205 | 114 | 215 | 194 | 134 | 78 |
| 40 .. 47 | 225 | 182 | 191 | 141 | 122 | 127 | 58 | 127 |
| 48 ..55 | 200 | 214 | 124 | 89 | 188 | 161 | 91 | 59 |
| 56 ..63 | 126 | 126 | 74 | 152 | 80 | 96 | 59 | 127 |

**Table 28 — Initial values of *ctxIdxMap*[ j][**3**] for child node under intra octree coding**

| j | *ctxIdxMap*[ j][3] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 59 | 121 | 160 | 210 | 171 | 211 | 240 | 231 |
| 8 .. 15 | 127 | 56 | 149 | 125 | 127 | 115 | 230 | 204 |
| 16 .. 23 | 55 | 127 | 78 | 192 | 127 | 182 | 197 | 218 |
| 24 .. 31 | 35 | 39 | 15 | 72 | 96 | 87 | 151 | 139 |
| 32 .. 39 | 46 | 141 | 152 | 240 | 114 | 162 | 240 | 240 |
| 40 .. 47 | 87 | 69 | 127 | 96 | 44, | 67 | 129 | 155 |
| 48 ..55 | 53 | 105 | 141 | 73 | 96 | 105 | 198 | 128 |
| 56 ..63 | 15 | 35 | 96 | 57 | 127 | 96 | 127 | 96 |

**Table 29 — Initial values of *ctxIdxMap*[ j][**4**] for child node under intra octree coding**

| j | *ctxIdxMap*[ j][4] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 23 | 30 | 130 | 66 | 139 | 127 | 30 | 105 |
| 8 .. 15 | 113 | 127 | 87 | 127 | 127 | 127 | 127 | 127 |
| 16 .. 23 | 166 | 146 | 70 | 15 | 209 | 116 | 141 | 90 |
| 24 .. 31 | 114 | 138 | 71 | 15 | 127 | 127 | 127 | 127 |
| 32 .. 39 | 204 | 240 | 198 | 219 | 232 | 240 | 142 | 240 |
| 40 .. 47 | 151 | 139 | 87 | 127 | 209 | 190 | 43 | 141 |
| 48 ..55 | 141 | 181 | 116 | 127 | 240 | 210 | 88 | 127 |
| 56 ..63 | 73 | 170 | 65 | 61 | 140 | 194 | 48 | 65 |

**Table 30— Initial values of *ctxIdxMap*[ j][**5**] for child node under intra octree coding**

| j | *ctxIdxMap*[ j][ 5] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 240 | 99 | 240 | 69 | 189 | 96 | 105 | 80 |
| 8 .. 15 | 154 | 233 | 152 | 141 | 127 | 152 | 127 | 127 |
| 16 .. 23 | 166 | 48 | 57 | 15 | 97 | 41 | 43 | 15 |
| 24 .. 31 | 127 | 116 | 127 | 127 | 127 | 85 | 127 | 127 |
| 32 .. 39 | 235 | 214 | 177 | 154 | 240 | 240 | 161 | 61 |
| 40 .. 47 | 219 | 185 | 152 | 208 | 157 | 90 | 127 | 127 |
| 48 ..55 | 117 | 138 | 69 | 30 | 154 | 80 | 62 | 15 |
| 56 ..63 | 141 | 121 | 127 | 127 | 127 | 41 | 127 | 105 |

**Table 31— Initial values of *ctxIdxMap*[ j][**6**] for child node under intra octree coding**

| j | *ctxIdxMap*[ j][ 6] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 227 | 199 | 188 | 103 | 212 | 141 | 205 | 55 |
| 8 .. 15 | 240 | 240 | 210 | 141 | 178 | 70 | 127 | 127 |
| 16 .. 23 | 240 | 84 | 139 | 73 | 139 | 60 | 127 | 59 |
| 24 .. 31 | 161 | 127 | 127 | 127 | 80 | 65 | 127 | 127 |
| 32 .. 39 | 201 | 195 | 127 | 69 | 175 | 80 | 87 | 39 |
| 40 .. 47 | 115 | 240 | 127 | 175 | 116 | 168 | 127 | 127 |
| 48 ..55 | 115 | 96 | 42 | 23 | 65 | 65 | 49 | 15 |
| 56 ..63 | 96 | 141 | 127 | 127 | 105 | 127 | 127 | 127 |

**Table 32— Initial values of *ctxIdxMap*[ j][**7**] for child node under intra octree coding**

| j | *ctxIdxMap*[ j][ 7] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 141 | 141 | 139 | 146 | 127 | 144 | 177 | 218 |
| 8 .. 15 | 127 | 63 | 127 | 115 | 127 | 164 | 240 | 194 |
| 16 .. 23 | 127 | 127 | 73 | 97 | 127 | 190 | 186 | 128 |
| 24 .. 31 | 73 | 16 | 15 | 88 | 116 | 127 | 80 | 161 |
| 32 .. 39 | 127 | 116 | 116 | 240 | 42 | 166 | 161 | 230 |
| 40 .. 47 | 96 | 47 | 127 | 127 | 58 | 88 | 116 | 109 |
| 48 ..55 | 105 | 116 | 15 | 61 | 15 | 80 | 73 | 155 |
| 56 ..63 | 15 | 15 | 15 | 45 | 36 | 73 | 57 | 121 |

* Then, the advanced occupied neighbourhood pattern *OccAdvNeiPati* (9.2.7.6) is used as the contextual information *CI* for entropy coding occupancy of child node of current coded node as follows.
  + *CI* is split into a primary part *CI1* and a second part *CI2* to derive the contextural information info1 and info2 for dynamic OBUF (12.5). *CI1* is a first representation of occupancy information of the set of neighboring nodes of current child node andcontains the first nBit1 bits of *OccAdvNeiPati*. *CI2* is a finer representation of the occupancy information and contains the remaining bits of *OccAdvNeiPati*. *CI1* is set as the first contextual information info1, and *CI2* is set as the second contextual information info2.
  + For example, to code child node of the coded node,
    - if the occupied neighbour nodes of a child node are not sparse, the contextual information is set as below,

CI1= OccAdvNeiPat0 >> 13

CI2= OccAdvNeiPat0 & 0x1FFF

* + - Otherwise, the contextual information is set as below.

CI1= OccAdvNeiPat0 >> 12

CI2= OccAdvNeiPat0 & 0x0FFF

* Then, each bit of child node occupancy of current node is decoded according to 12.4 by calling the OBUF instances of octree, and usinginfo1,info2, *ctxIdxMap*[ ][ ] and *obufCtxArray*[ ]. The type (sparse or non-sparse) of called OBUF instances depends on the first bit of *OccAdvNeiPati* (9.2.7.6), which indicates if the occupied neighbour nodes of coded child node are sparse or not.

### Planar occupancy coding

#### General

Subclause 9.2.11 applies when occtree\_planar\_enabled is 1.

Planar occupancy coding decomposes the node occupancy bitmap into axis-aligned planes. Each coded axis has two perpendicular planes that child nodes can occupy as illustrated by Table 23. For each planar-eligible coded axis (9.2.11.5), planar occupancy coding specifies whether one of the two planes is unoccupied. Plane occupancy is then used by bitwise occupancy coding to constrain and infer the coding of bits in the node occupancy bitmap.

There shall be at least one child node in each occupied plane.

The definition of an occupancy tree node requires that at least one plane is occupied along each coded axis.

For example, if a node has three planar-eligible coded axes, there is a total of six axis-aligned planes. Along the S axis (𝑘 = 0), information about the occupied state of the two T-V planes is coded.

Table 23 — Plane, perpendicular to each planar axis, 𝑘

| 𝑘 | Planar axis | Plane axes |
| --- | --- | --- |
| 0 | S | T-V |
| 1 | T | S-V |
| 2 | V | S-T |

#### Syntax element semantics

planar\_copy\_modespecifies, when present, whether (when 1) or not (when 0) the the values of occ\_plane\_pos[ 𝑘 ] indicating the occupancy of the child nodes are copied from the corresponding node in the reference frame.

multi\_planar\_flagspecifies, when present, whether the positions of child nodes in the node occupancy bitmap shall locate at the intersection of planes (equal to 1) or not (equal to 0).

occ\_single\_plane[ 𝑘 ] specifies, when present, whether (when 1) the locations of child nodes in the node occupancy bitmap shall occupy a single plane or (when 0) both planes perpendicular to the 𝑘-th axis. When equal to 1, the location of the single plane is specified by occ\_plane\_pos[ 𝑘 ]. When not present, the child nodes can be located in either or both planes perpendicular to the 𝑘-th axis. The number of occupied planes is illustrated in Table 24.

Table 24 — Interpretation of PlanarEligible[ 𝑘 ] and occ\_single\_plane[ 𝑘 ]

| PlanarEligible[ 𝑘 ] | occ\_single\_plane[ 𝑘 ] | № occupied planes | PlanarMinPlanes[ 𝑘 ] |
| --- | --- | --- | --- |
| 0 | not present | 1 or 2 | 1 |
| 1 | 0 | 2 | 2 |
| 1 | 1 | 1 | 1 |

occ\_plane\_pos[ 𝑘 ] specifies the node-relative location along the 𝑘-th axis for the occupied plane specified by occ\_single\_plane[ 𝑘 ] equal to 1.

Examples of planar occupancy are illustrated in Figure 12. Each entry shows the child node indices for a node with three coded axes. If an axis 𝑘 is eligible for planar coding and occ\_single\_plane[ 𝑘 ] is 0, the two occupied planes are marked with a dotted line. If occ\_single\_plane[ 𝑘 ] is 1, the unoccupied plane is marked by hatching and a dashed (red) line, with its child indices in grey; the occupied plane is not marked for clarity. In the case where three axes are eligible and each has occ\_single\_plane equal to 1, there is only a single child node present; its location is fully constrained by the planes.

图示, 示意图

描述已自动生成

Figure 12 — Example planar constraints on node occupancy bitmap.

#### Minimum number of child nodes

Planar occupancy coding can require that the coded node has a minimum of either one or two child nodes, as specified by the expression PlanarMinChildren.

A node shall have at least two child nodes if, for any planar-eligible axis 𝑘, occ\_single\_plane[ 𝑘 ] specifies that there shall be a minimum of two occupied planes; otherwise, the node shall have at least one child node.

PlanarMinChildren := MaxVec(PlanarMinPlanes)

#### Free axes

A free axis is a coded axis whose occupancy is not constrained to a single plane by occ\_single\_plane, as specified by the expression PlanarFreeAxis[ 𝑘 ].

PlanarFreeAxis[k] := ¬PlanarEligible[k] || PlanarMinPlanes[k] == 2

In Figure 12 a free axis is an axis with either two marked planes or no marked planes. In the case where three axes are eligible, of which two have occ\_single\_plane equal to 1, PlanarMinChildren is 2.

#### Per-axis eligibility

##### Condition

Only certain axes are eligible for planar occupancy coding. Eligibility for the 𝑘-th axis is specified by the expression PlanarEligible[ 𝑘 ]. Eligibility shall be determined after any applicable update to the eligibility state (9.2.11.5.5).

An axis is not eligible for planar coding when either planar occupancy coding is disabled, or the axis is not coded, or geo\_disable\_planar\_idcm\_angular is equal to 1 and occ\_direct\_node is 1. Otherwise, the determination of eligibility depends upon the use of the angular coding and whether the node is eligible for angular contextualization (9.2.13.7.2) as specified in Table 25.

PlanarEligible[k] :=  
 occtree\_planar\_enabled && AxisCoded[k]  
 && (geom\_angular\_enabled ? PlanarEligibleByAng[k] : PlanarEligibleByDensity[k])

PlanarEligibleByAng[k] :=  
 AngularEligible ? k == 2 || k == AzimuthAxis  
 : 0

Table 25 — Method to determine eligibility for an axis

| Axis | 𝑘 | Angular coding disabled | Angular coding enabled | |
| --- | --- | --- | --- | --- |
| AngularEligible == 0 | AngularEligible == 1 |
| S | 0 | PlanarEligibleByDensity[ 0 ] | Not eligible | AzimuthAxisIsS |
| T | 1 | PlanarEligibleByDensity[ 1 ] | Not eligible | AzimuthAxisIsQ |
| V | 2 | PlanarEligibleByDensity[ 2 ] | Not eligible | Eligible |

Axes whose eligibility is determined by the expression PlanarEligibeByDensity[ 𝑘 ] are eligible if:

* the density of the points in the tree level at depth *dpth* is sparse enough.

PlanarEligibleByDensity[k]:= PointDensity[dpth ] < 13

The expression *PointDensity*[*dpth* ] is a factor that identifies the density of the points in the tree level at depth *dpth*:

PointDensity[dpth ]:= (slice\_num\_points\_minus1 + 1 – DirectNodePointCnt) × 10 / OccNodeCnt[ dpth ]

##### State variable and update

When octree\_planar\_neigh\_prediction\_enabled is 1, eligibility is specified in terms of the following state variables:

* The sparse array *NodeOccMap* of node occupancy bitmaps; *NodeOccMap*[ *dpth* ][ *ns* ][ *nt* ][ *nv* ] is the coded node occupancy bitmap of the node located at ( *ns*, *nt*, *nv* ) in the tree level at depth *dpth*.

At the end of each occupancy\_tree\_node syntax structure, the state shall be updated for the next tree level:

NodeOccMap[Dpth][Ns][Nt][Nv] = OccupancyMap.

#### Previous coded node for contextualization of occ\_plane\_pos

##### General

Subclause 9.2.11.6 does not apply when occtree\_planar\_buffer\_disabled is 1.

Planar contextualization of occ\_plane\_pos[ 𝑘 ] can use the following information about the previous planar-eligible coded node that is located in the same plane (9.2.11.6.2) as the coded node:

* The zone within the plane that the node resides (9.2.11.6.3).
* The values for occ\_single\_plane[ 𝑘 ] and occ\_plane\_pos[ 𝑘 ].

##### Identification of the plane

The plane normal to the 𝑘-th axis of a coded node is identified by its location along the axis modulo .

PlanarNodeAxisLoc[k] := Nloc[k] & 0x3FFF

##### Zone within a plane

The plane normal to the 𝑘-th axis of a coded node is partitioned into zones according to the norm of the node location within the plane. The expression PlanarNodeZone[ 𝑘 ] identifies the zone for the coded node.

PlanarNodeZone[k] :=  
 k == 0 ? Max(Nt & 0xF8, Nv & 0xF8) >> 3 :  
 k == 1 ? Max(Ns & 0xF8, Nv & 0xF8) >> 3 :  
 k == 2 ? Max(Ns & 0xF8, Nt & 0xF8) >> 3 : na

Figure 13 illustrates the partitioning of an S-T plane (𝑘 = 0) according to node location.

图示

描述已自动生成

Figure 13 — S-T plane divided into zones.

##### State variables

Information about previous planar-eligible coded nodes is specified in terms of the following state variables; the indexes 𝑘 and axisLoc identify the location of a plane along the 𝑘-th axis:

* The array PrevPlanarNodeZone; PrevPlanarNodeZone[ 𝑘 ][ axisLoc ] is the plane zone of the previous planar-eligible node in the identified plane.
* The array PrevOccSinglePlane; PrevOccSinglePlane[ 𝑘 ][ axisLoc ] is the value of occ\_single\_plane[ 𝑘 ] for the previous planar-eligible node in the identified plane.
* The array PrevOccPlanePos; PrevOccPlanePos[ 𝑘 ][ axisLoc ] is the value of occ\_plane\_pos[ 𝑘 ] for the previous planar-eligible node in the identified plane.

##### Initial state

At the start of every occupancy tree level, every element of PrevOccSinglePlane shall be initialized to 0.

##### State update at the end of each node

After each occupancy\_tree\_node syntax structure, the state shall be updated for each planar-eligible axis:

for (k = 0; k < 3; k++)  
 if (PlanarEligible[k]) {  
 PrevPlanarNodeZone[k][PlanarNodeAxisLoc[k]] = PlanarNodeZone[k]  
 PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]] = occ\_single\_plane[k]  
  
 if (occ\_single\_plane[k])  
 PrevOccPlanePos[k][PlanarNodeAxisLoc[k]] = occ\_plane\_pos[k]  
 }

#### Determination of CtxIdxPlanePos for occ\_plane\_pos[ 𝑘 ]

##### Case for angular-ineligible axes

Contextualization of occ\_plane\_pos[ 𝑘 ] for nodes not eligible for angular contextualization (AngularEligible is 0) is specified by the expression CtxIdxPlanePos. When octree\_planar\_neigh\_prediction\_enabled is 1, CtxIdxPlanePos is set equal to *CtxIdxNeighPlanePos* using the planar information of neighbors (9.2.11.7.2).

CtxIdxPlanePos :=octree\_planar\_neigh\_prediction\_enabled ? CtxIdxNeighPlanePos:  
 (occtree\_planar\_buffer\_disabled || ¬PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]]  
 ? adjPlaneCtxInc  
 : 12 × k + 4 × adjPlaneCtxInc + 2 × zoneCtxInc + prevPlanePosCtxInc + 3)  
.

The expression adjPlaneCtxInc discriminates by whether nodes have adjoining neighbours on a single side along the 𝑘-th axis, and if so, on which of the two sides they are present. Adjoining neighbours are:

* those along the 𝑘-th axis identified by the corresponding bits of the occupied neighbourhood pattern (adjNeighHL); and
* when the node is in the lower 𝑘-th axis plane of its parent, the sibling nodes in the corresponding upper plane (identified by OccPlaneMask[ 1 ][ 𝑘 ]). The bit masks OccPlaneMask[ planeLoc ][ 𝑘 ] that identify planes in an occupancy bitmap are specified by Table 26.

adjPlaneCtxInc := (adjNeighHL | sibPlaneH << 1) % 3  
 where  
 adjNeighHL := (OccNeighPat >> 2 × k) & 3  
 sibPlaneH := (Nloc[k] & 1) ≠ 1 && (OccupancyMapP & OccPlaneMask[1][k]) ≠ 0

Whenever occtree\_coded\_axis[ Dpth − 1 ][ 𝑘 ] is 0, sibPlaneH is always 0.

Table 14 — Bit masks OccPlaneMask[ planeLoc ][ 𝑘 ] that identify planes of a node occupancy bitmap

|  |  |  |  |
| --- | --- | --- | --- |
| 𝑘 | 0 | 1 | 2 |
| OccPlaneMask[ 0 ][ 𝑘 ] | 0x0F | 0x33 | 0x55 |
| OccPlaneMask[ 1 ][ 𝑘 ] | 0xF0 | 0xCC | 0xAA |

If occtree\_planar\_buffer\_disabled is 0, contextualization uses information about the previous planar-eligible node in the plane identified by the coded node location (9.2.11.6.2).

The expression zoneCtxInc discriminates by whether the coded node is within ±1 zones of the identified previous node.

zoneCtxInc := Abs(a − b) > 1  
 where  
 a := PrevPlanarNodeZone[k][PlanarNodeAxisLoc[k]]  
 b := PlanarNodeZone[k]

The expression prevPlanePosCtxInc discriminates by the occupied plane position of the identified previous node.

prevPlanePosCtxInc := PrevOccPlanePos[k][PlanarNodeAxisLoc[k]]

##### Neighbourhood planar information based prediction

The expression adjOccC[ *n*] identifies the occupancy bitmaps of the spatial adjacent nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 27.

adjOccC[n] := OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0 ? NodeOccMap[Dpth][Ns + ds][Nt + dt][Nv + dv] : 0

The expression adjOccSinglePlane[ *k*] identifies the occupied neighbourhood planar pattern along the 𝑘-th axis. It is a linear combination of spatial adjacent nodes coded in the same tree level that are available and adjoin the coded node. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 27. An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood planar pattern equal to 0.

adjOccSinglePlane[k]:= 0

for (n = 0; n < 7; n++)   
if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] == 0)  
 adjOccSinglePlane[k] <<= 1  
else {  
 plane0 = (adjOccC[n] & OccPlaneMask[0][k]) ≠ 0  
 plane1 = (adjOccC[n] & OccPlaneMask[1][k]) ≠ 0  
 hasSinglePlane = plane0 ^ plane1  
 adjOccSinglePlane[k] |= hasSinglePlane  
}

The expression adjOccPlanePos[ *k*] identifies the occupied neighbourhood planar position pattern along the 𝑘-th axis. It is a linear combination of spatial adjacent nodes coded in the same tree level that are available and adjoin the coded node. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 27. An occupancy tree node with no spatially adjacent nodes has an occupied neighbourhood planar position pattern equal to 0.

adjOccPlanePos[k] :=0

for (n = 0; n < 7; n++)  
if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] == 0)  
 adjOccPlanePos[k] << =1  
else{  
 plane0 = (adjOccC[n] & OccPlaneMask[0][k]) ≠ 0  
 plane1 = (adjOccC[n] & OccPlaneMask[1][k]) ≠ 0  
 hasSinglePlane = plane0 ^ plane1  
 adjOccPlanePos[k] |= hasSinglePlane & OccPlaneMask[1][k]  
 adjOccPlanePos[k] <<= 1  
}

Table 27 — Relative locations (*ds*, *dt*, *dv*) used in the computation of adjOccC[ *n*], adjOccSinglePlane[ *k*], adjOccPlanePos[ *k*] and neighAvailable

| n | *ds* | *dt* | *dv* |
| --- | --- | --- | --- |
| 0 | −1 | 0 | 0 |
| 1 | 0 | −1 | 0 |
| 2 | 0 | 0 | −1 |
| 3 | -1 | -1 | 0 |
| 4 | -1 | 1 | -1 |
| 5 | 0 | -1 | -1 |
| 6 | -1 | -1 | -1 |

The expression adjOccNeighPat identifies the bitmap for the spatial nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 28.

adjOccNeighPat: =0  
for (n = 0; n < 12; n++){  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0)  
 adjOccNeighPat |= 1  
 adjOccNeighPat <<= 1   
}

Table 28 — Relative locations ( ds, dt, dv ) used in the computation of adjOccNeighPat

| n | ds | dt | dv |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | 0 | -1 |
| 6 | 1 | -1 | 0 |
| 7 | 0 | 1 | 1 |
| 8 | 0 | 1 | -1 |
| 9 | 0 | -1 | 1 |
| 10 | -1 | 1 | 0 |
| 11 | -1 | 0 | 1 |

The expression neighAvailable identifies whether there are spatial nodes within the availability windows. The relative location values *ds, dt* and *dv* of the  *n*-th adjacent node for the expression are specified in Table 27.

neighAvailable:= false  
for (n = 0; n < 7; n++){  
 if (OccNodePresent[Dpth][Ns + ds][Nt + dt][Nv + dv] ≠ 0)  
 neighAvailable |= adjOccC[n] ≠ 0}

The expression primaryCtx discriminates by spatial nodes’ geometry primary information. The expression minorCtx discriminates by spatial nodes’ geometry minor information.

primaryCtx: = neighAvailable ? zoneCtxInc << 7 + prevPlanePosCtxInc << 6 + ((adjOccPlanePos[k] & 0x70) >> 4) << 3+ ((adjOccSinglePlane[k]&0x70) >> 4) :  
zoneCtxInc << 6 + prevPlanePosCtxInc << 5 + adjPlaneCtxInc << 3 + ((adjOccNeighPat & 0xe00) >> 9) + (0x01 << 7)

minorCtx: = neighAvailable ? adjPlaneCtxInc << 8 + (adjOccPlanePos[k] & 0xf) << 4 + (adjOccSinglePlane[k] & 0xf) :   
(adjOccNeighPat & 0x1ff) +(0x01 << 9)

*CtxIdxNeighPlanePos* is derived from the Dynamic\_OBUF based primaryCtx and minorCtx.

CtxIdxNeighPlanePos := Dynamic\_OBUF(primaryCtx, minorCtx)

[Ed. (YZ): Application of dynamic\_OBUF to be checked and fixed with the correct function.]

##### Case for angular-eligible axes

Contextualization of occ\_plane\_pos[ 𝑘 ] for nodes eligible for angular contextualization (AngularEligible is 1) is specified by 9.2.13.7.

#### Multiple axes eligibility

Eligibility for planar coding of multiple axes is specified by the expression MultiPlanarEligible.

NumEligibleAxes indicates the number of axes that are eligible for planar coding. It is used to determine the value of MultiPlanarEligible .

NumEligibleAxes = PlanarEligible[0] + PlanarEligible[1] + PlanarEligible[2]

MultiPlanarEligible = numEligibleAxes > 1 ? 1 : 0

#### occ\_single\_plane inference

When multi\_planar\_flag is present or when planar\_copy\_mode is present, inferences on the value of occ\_single\_plane[ 𝑘 ] can be made.

PlanarInferred[ 𝑘 ] indicates that occ\_single\_plane[ 𝑘 ] can be inferred (equal to 1) or not (equal to 0).

When planar\_copy\_mode equals to 1 or multi\_planar\_flag equals to 1, occ\_single\_plane[ 𝑘 ] can be inferred if 𝑘-th axis is eligible for planar coding.

if(planar\_copy\_mode)  
 for(k = 0; k < 3; k++)  
 PlanarInferred[k] = 1  
if(multi\_planar\_flag){

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

PlanarInferred[k] = PlanarEligible[k]

When planar\_copy\_mode equals to 0 and multi\_planar\_flag equals to 0, occ\_single\_plane[ 𝑘 ] can be inferred if 𝑘-th axis is eligible for planar coding and present NumEligibleAxes – 1 occ\_single\_plane[ j ] which equals to 1 and j < 𝑘.

if(¬multi\_planar\_flag){

count = 0;

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

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

if(PlanarEligible[j]

count += occ\_single\_plane[j] ? 1 :0

PlanarInferred[k] = (count == NumEligibleAxes – 1) && PlanarEligible[k] ? 1 :0

}

}

If PlanarInferred[ 𝑘 ] equal to 1, occ\_single\_plane[ 𝑘 ] will be inferred as follows:

if(PlanarInferred[k])

occ\_single\_plane[k] = planar\_copy\_mode ? PlaneRef[k] : multi\_planar\_flag

#### Determination of AllowPlanarCopyMode for planar\_copy\_mode

Determination of signalling of planar\_copy\_mode is specified by the expression AllowPlanarCopyMode.

AllowPlanarCopyMode = isInter && OccupancyIsPredictable && planarEligibile  
 where   
 planarEligible = PlanarEligible[0] || PlanarEligible[1] || PlanarEligible[2]

[Ed. (YZ): Derivation of PredOccBitMap when biprediction is enabled to be added on top of the description of uni-prediction.]

#### occ\_plane\_pos inference

When planar\_copy\_mode is present, inferences on the value of occ\_plane\_pos[ 𝑘 ] may be made.

PlanarPosInferred[ 𝑘 ] indicates that occ\_plane\_pos [ 𝑘 ] can be inferred (equal to 1) or not (equal to 0).

When AllowPlanarCopyMode equals to 1, occ\_plane\_pos[ 𝑘 ] can be inferred if planar copy mode is indicated or if the 𝑘-th axis is last eligible axis for planar coding and planar coding of previous axes match that of the reference frame.

prevDirMatch = true  
for(k = 0; k < 3; k++) {  
 if(planar\_copy\_mode)  
 PlanarPosInferred[k] = 1  
 else if(AllowPlanarCopyMode && k == lastDirIdx && prevDirMatch && PlaneRef[k])  
 PlanarPosInferred[k] = 1  
 else  
 PlanarPosInferred[k] = 0  
 prevDirMatch = prevDirMatch & (occ\_single\_plane[k] == PlaneRef[k]) && (occ\_plane\_pos[k] == PlanePosRef[k])  
}  
 where  
 lastDirIdx = PlanarEligible[2] ? 2 : (PlanarEligible[1] ? 1 : 0)

If PlanarPosInferred[ 𝑘 ] equal to 1, occ\_plane\_pos[ 𝑘 ] will be inferred as follows:

if(PlanarPosInferred[k])  
 occ\_plane\_pos[k] = planar\_copy\_mode ? PlanePosRef[k] : (PlanePosRef[k] ? 0 : 1)

#### Determination of CtxIdxPlanarCopyMode for planar\_copy\_mode

Contextualization of planar\_copy\_mode is specified by the expression CtxIdxPlanarCopyMode.

CtxIdxPlanarCopyMode = 8 \* (4 \* (eligibilityPlanarSum – 1) + matchDirSum) + planarRefSum  
 where  
 eligibilityPlanarSum = PlanarEligible[0] + PlanarEligible[1] + PlanarEligible[2]  
 matchedDirSum = matchedDir[0] + matchedDir[1] + matchedDir[2]  
 planeRefSum = PlaneRef[0] << 2 + PlaneRef[1] << 1 + PlaneRef[0]

The variable matchedDir[] is used to determine whether the planar information in the reference frame is equal to the planar information of the closest zone in the planar buffer.

matchedDir[k] = 0  
if(PlanarEligible[k])  
 matchedDir[k] = PrevOccSinglePlane[k][PlanarNodeAxisLoc[k]] == PlanarRef[k]   
 && PrevOccPlanePos[k][PlanarNodeAxisLoc[k]] == PlanePosRef[k]

#### Derivation of PlanarRef and PlanePosRef

The variables PlanarRef and PlanePosRef denote the planar information of the collocated node in the reference frame.

PlanarRef[0] = (PredOccBitMap & 0x0f) ≠ (PredOccBitMap & 0xf0)  
PlanarRef[1] = (PredOccBitMap & 0x33) ≠ (PredOccBitMap & 0xcc)  
PlanarRef[2] = (PredOccBitMap & 0x55) ≠ (PredOccBitMap & 0xaa)

PlanarPosRef[0] = PlanarRef[0] && (PredOccBitMap & 0xf0) > 0  
PlanarPosRef[1] = PlanarRef[1] && (PredOccBitMap & 0xcc) > 0  
PlanarPosRef[2] = PlanarRef[2] && (PredOccBitMap & 0xaa) > 0

### Direct nodes

#### General

Subclause 9.2.12 applies when occtree\_direct\_coding\_mode is not 0.

Certain occupancy tree nodes may immediately code point positions as a direct node, instead of coding a node occupancy bitmap for subsequent traversal. A direct node can represent either two distinct point positions, or a single position that is identical for every represented point.

A direct node codes a position as a residual relative to the node position.

The number of points coded in direct nodes is counted cumulatively, *DirectNodePointCnt*. The variable is initialized to 0 At the start of every slice.

Direct coding is limited to nodes that are both eligible and not prohibited by the planar direct node rate limit. Eligibility shall be determined for each occupancy tree node based upon the degree of spatial isolation as specified in 9.2.12.3.

#### Syntax element semantics

occ\_direct\_node equal to 1 specifies that the occupancy tree node is a direct node that codes the position of at least one point. When occ\_direct\_node is not present, it shall be inferred to be 0.

direct\_point\_cnt\_eq2 equal to 1 specifies that the direct node codes two point positions. direct\_point\_cnt\_eq2 equal to 0 specifies that the occupancy tree node codes a single point position for one or more points.

direct\_dup\_point\_cnt plus 1 specifies, when present, the number of points the direct node represents when direct\_point\_cnt\_eq2 is 0. When direct\_dup\_point\_cnt is not present, it shall be inferred to be 0.

direct\_joint\_prefix[ 𝑘 ] specifies a sequence of identical MSBs in the 𝑘-th component of two coded position residuals. The MSB position of the syntax element value indicates the number of position bits coded by the syntax element and does not form part of the reconstructed point position.

direct\_joint\_diff\_bit[ 𝑘 ] specifies, when direct\_joint\_prefix[ 𝑘 ] is present, the value of a bit in the binary representation of the 𝑘-th component of the two coded position residuals. The bit is the most significant non-identical bit of the two coded residual components. Its value is that for the first point. When direct\_joint\_diff\_bit[ 𝑘 ] is not present, it shall be inferred to be 0.

direct\_rem[ dnPt ][ 𝑘 ], direct\_rem\_st\_ang[ dnPt ] and direct\_rem\_v\_ang[ dnPt ] specify the remaining position bits of the dnPt-th point. When present, direct\_rem codes the 𝑘-th component, direct\_rem\_st\_ang either the 𝑠- or 𝑡-component depending upon the node location and direct\_rem\_v\_ang codes the 𝑣-component. When not present, they shall be inferred to be 0.

direct\_v\_ang\_resid\_abs[ dnPt ] and direct\_v\_ang\_resid\_sign[ dnPt ] together specify, when present, the residual of the 𝑣-component of dnPt-th point, for a prediction made from a selected beam’s elevation and vertical offset. When not present, they shall be inferred to be 0. The residual of the 𝑣-component of dnPt-th point is specified by the expression DirectVAngResid[ dnPt ].

DirectVAngResid[dnPt] :=  
 (1 − 2 × direct\_v\_ang\_resid\_sign[dnPt]) × direct\_v\_ang\_resid\_abs[dnPt]

beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ] together specify the index of an enumerated beam relative to a per-node prediction. The residual between the enumerated beam and the per node prediction is specified by the expression BeamIdxResid[ dnPt ].

BeamIdxResid[dnPt] := (1 − 2 × beam\_idx\_resid\_sign[dnPt]) × beam\_idx\_resid\_abs[dnPt]

The beam is used in the contextualization of the syntax elements direct\_rem\_st\_ang[ dnPt ], direct\_rem\_v\_ang[ dnPt ] and in 𝑣-component of dnPt-th point prediction to be used with DirectVAngResid[ dnPt ].

#### Eligibility

##### Decision for each occupancy tree node

Only certain occupancy tree nodes are eligible to be direct nodes. They are specified by the expression DirectModeEligible. An eligible node:

* is not the root node;
* is not the root node of a fully quantized subtree (9.2.14.2.6); and
* meets one of the following mode-dependent conditions:
  + When occtree\_direct\_coding\_mode is 1: if there are no nodes in the occupied neighbourhood pattern of the parent node, the coded node has no siblings and the parent node has at most one sibling.
  + When occtree\_direct\_coding\_mode is 2: if there are no nodes in the occupied neighbourhood pattern of the parent node.
  + When occtree\_direct\_coding\_mode is 3: if the coded node has at least one sibling.

DirectModeEligible := occtree\_direct\_coding\_mode > 0  
 && Dpth > 0  
 && MaxVec(QuantizedNodeSizeLog2) > 0 && ¬OccupancyIsPredictable  
 && ( (interNonAngular && DirectMode1Eligible) ||   
 (¬interNonAngular   
 && (occtree\_direct\_coding\_mode ≠ 1 || DirectMode1Eligible)  
 && (occtree\_direct\_coding\_mode ≠ 2 || DirectMode2Eligible)  
 && (occtree\_direct\_coding\_mode ≠ 3 || DirectMode3Eligible)   
 )  
 )  
 where,  
 interNonAngular = slice\_inter\_prediction && ¬geom\_angular\_enabled

DirectMode1Eligible := OccNeighPatEq0[Dpth − 1][NsP][NtP][NvP]  
 && OccNodeChildCnt[Dpth − 1][NsP][NtP][NvP] == 1  
 && (Dpth < 2 || OccNodeChildCnt[Dpth − 2][NsG][NtG][NvG] ≤ 2)

DirectMode2Eligible := OccNeighPatEq0[Dpth − 1][NsP][NtP][NvP]

DirectMode3Eligible := OccNodeChildCnt[Dpth − 1][NsP][NtP][NvP] > 1

When occtree\_inter\_angular\_direct\_coding\_enabled is equal to 1, an eligible node to be direct nodes meets the condition when DnEligibleByAng is equal to 1. (9.2.13.8.1).

DirectModeEligible := DnEligibleByAng

##### Presence of occ\_direct\_node

The syntax element occ\_direct\_node shall only be present in occupancy tree nodes that are both eligible for direct coding and not prohibited by the rate limit for direct nodes that applies when planar occupancy coding is enabled.

The direct node rate limit mask is specified by the expression DnPresenceMask[ 𝑖 ], 𝑖 ∈ 0 .. 31.

DnPresenceMask[i] := occtree\_planar\_enabled && occtree\_direct\_coding\_mode == 1  
 ? dnRate × i % 32 + (dnRate ≥ 32)  
 : 1  
 where  
 dnRate := occtree\_direct\_node\_rate\_minus1 + 1

The expression DirectNodePresent specifies the presence of the syntax element.

DirectNodePresent := DirectModeEligible && DnPresenceMask[(Dpth + DnEligibleCnt) % 32]

##### State variables

Eligibility is specified in terms of the following state variables; the indexes dpth, ns, nt and nv identify a node with location ( ns, nt, nv ) in the tree level at depth dpth:

* The sparse array OccNeighPatEq0; OccNeighPatEq0[ dpth ][ ns ][ nt ][ nv ] identifies whether the identified node has no nodes present in its occupied neighbourhood pattern.
* The sparse array OccNodeChildCnt; OccNodeChildCnt[ dpth ][ ns ][ nt ][ nv ] is the number of child nodes of the identified node.
* The variable DnEligibleCnt, a cumulative count of eligible nodes in a tree level.

##### Initial state

At the start of every occupancy\_tree syntax structure, the OccNodeChildCnt array shall be cleared; all elements of OccNodeChildCnt are unset.

##### State update at the start of every occupancy tree level

At the start of every occupancy\_tree\_level syntax structure, the count of eligible nodes DnEligibleCnt shall be set to zero.

##### State update after each coded occupancy tree node

This subclause applies at the end of every occupancy\_tree\_node syntax structure.

The number of child nodes and the presence of any nodes in the occupied neighbourhood pattern are recorded for use in subsequent eligibility decisions.

OccNodeChildCnt[Dpth][Ns][Nt][Nv] = direct\_node ? 0 : OccChildCnt  
OccNeighPatEq0[Dpth][Ns][Nt][Nv] = OccNeighPat == 0

If the node is eligible for direct coding, irrespective of the presence of occ\_direct\_node, the count of eligible nodes shall be incremented.

if (DirectModeEligible)  
 DnEligibleCnt++

#### Points represented by direct nodes

##### General

The unscaled positions for the points coded by the direct node are specified by the expression DnPtPos[ dnPt ][ 𝑘 ]. They are the concatenation of:

* the (quantized) node position,
* any bit corresponding to an occupied plane as determined by planar occupancy coding,
* any bits from joint direct position coding, and
* any remaining bits.

When geometry subtree scaling is enabled, direct nodes code partially quantized positions relative to the quantized node position.

DnPtPos[dnPt][k] := Nloc[k] << QuantizedNodeSizeLog2[k] | DnPtPosRem[dnPt][k]  
  
DnPtPosRem[dnPt][k] := DnPlanarPos[k] | DnJointPos[dnPt][k] | DnRemPos[dnPt][k]

DnPtPosS[dnPt] := DnPtPos[dnPt][0]  
DnPtPosT[dnPt] := DnPtPos[dnPt][1]  
DnPtPosV[dnPt] := DnPtPos[dnPt][2]

##### Output

At the end of the direct node, the coded points shall be scaled (9.2.14.6) and appended to the output point list:

if (occ\_direct\_node) {  
 for (dnPt = 0; dnPt ≤ direct\_point\_cnt\_eq2; dnPt++, PointCnt++, DirectNodePointCnt++)  
 for (k = 0; k < 3; k++)  
 PointPos[PointCnt][k] = OccPosScaleK(k, DnPtPos[dnPt][k])  
  
 for (i = 0; i < direct\_dup\_point\_cnt; i++, PointCnt++, DirectNodePointCnt++)  
 for (k = 0; k < 3; k++)  
 PointPos[PointCnt][k] = PointPos[PointCnt − 1][k]  
}

##### Planar-inferred position bits

When an axis is eligible for planar occupancy coding and it has a single occupied plane, the MSB of the position residual for that axis 𝑘 is specified by DnPlanarPos[ 𝑘 ], equal to the occupied plane location.

DnPlanarPos[k] := ¬PlanarFreeAxis[k] ? occ\_plane\_pos[k] << DnBitsAfterPlanar[k] : 0

The number of bits coded by each position residual exclusive of any bit derived from planar occupancy coding is specified for each component by the expression DnBitsAfterPlanar[ 𝑘 ].

DnBitsAfterPlanar[k] := QuantizedNodeSizeLog2[k] − ¬PlanarFreeAxis[k]

##### Joint coded position bits

The position residuals shall be jointly coded for an axis when the direct node codes two positions, joint coding is enabled and it has a residual bit to code. When angular coding is enabled, components coded by direct\_rem\_st\_ang and either direct\_rem\_v\_ang or direct\_v\_ang\_resid\_abs and direct\_v\_ang\_resid\_sign shall not be jointly coded.

DnJointCoded[k] := occtree\_direct\_joint\_coding\_enabled && direct\_point\_cnt\_eq2  
 && DnBitsAfterPlanar[k] > 0  
 && (¬geom\_angular\_enabled || k == (1 ^ AzimuthAxis))

The joint-coded bits for the two positions are specified by the expression DnJointPos[ dnPt ][ 𝑘 ]. For each component 𝑘, they comprise the common MSB (9.2.12.5.2) and the first divergent bit (9.2.12.5.3), if any. Joint coding is exclusive of any planar-inferred position bit.

DnJointPos[dnPt][k] :=  
 DnJointCoded[k] ? DnJointPosCommon[k] | DnJointPosDiffBit[dnPt][k] : 0

##### Remaining bits

The number of bits coded by each position residual exclusive of any bits derived from planar occupancy coding or joint coding is specified for each component by the expression DnRemBits[ 𝑘 ].

DnRemBits[k] := DnBitsAfterPlanar[k] − DnJointCommonBits[k] − DnJointDiffBits[k]

The expression DnRemPos[ dnPt ][ 𝑘 ] specifies the position bits coded by the syntax elements direct\_rem, direct\_rem\_st\_ang and direct\_rem\_v\_ang.

DnRemPos[dnPt][k] :=  
 geom\_angular\_enabled && k == AzimuthAxis ? direct\_rem\_st\_ang[dnPt] :  
 geom\_angular\_enabled && k == 2 ? occtree\_angular\_extension\_enabled ?  
 DirectRemVAngPred[dnPt] :  
 direct\_rem\_v\_ang[dnPt]  
 : direct\_rem[dnPt][k]

#### Joint coding of point positions

##### Point order

The coded order of two jointly coded points shall satisfy the constraint DnPtPosConstraint equal to 1:

* The 𝑠-coordinate of the first point shall be less than or equal to that of the second point.
* If the 𝑠-coordinates are equal, the 𝑡-coordinate of the first point shall be less than or equal to that of the second point.
* If both the 𝑠- and 𝑡-coordinates are equal, the 𝑣-coordinate of the first point shall be less than or equal to that of the second point

DnPtPosConstraint :=  
 DnPtPosS[0] < DnPtPosS[1]  
 || DnPtPosT[0] < DnPtPosT[1] && dnPtPosSameS  
 || DnPtPosV[0] ≤ DnPtPosV[1] && dnPtPosSameS && dnPtPosSameT  
 where  
 dnPtPosSameS := DnPtPosS[0] == DnPtPosS[1]  
 dnPtPosSameT := DnPtPosT[0] == DnPtPosT[1]

##### Common position bits

The number of position bits coded by direct\_joint\_prefix[ 𝑘 ] is specified by DnJointPrefixBits[ 𝑘 ].

DnJointPrefixBits[k] := DnJointCoded[k] ? IntLog2(direct\_joint\_prefix[k]) : 0

The number of bits coded by the 𝑘-th component of each position residual, exclusive of any bits derived from planar occupancy coding or a joint-coded prefix, is specified by the expression DnBitsAfterJointPrefix[ 𝑘 ].

DnBitsAfterJointPrefix[k] := DnBitsAfterPlanar[k] − DnJointCommonBits[k]

The expression DnJointPosPrefix[ 𝑘 ] specifies the value of the position bits coded by direct\_joint\_prefix[ 𝑘 ].

DnJointPosPrefix[k] :=  
 (direct\_joint\_prefix[k] ^ Exp2(DnJointPrefixBits[k])) << DnBitsAfterJointPrefix[k]

##### First divergent position bit

The existence of a divergent bit in the 𝑘-th component of the two jointly coded positions is specified by the expression DnJointPosDiffBits[ 𝑘 ]. A jointly coded divergent position bit exists if direct\_joint\_prefix[ 𝑘 ] is present and does not complete the coding of the 𝑘-th position component.

DnJointDiffBits[k] := DnJointCoded[k] ? DnBitsAfterPlanar[k] > DnJointPrefixBits[k] : 0

The value of the jointly coded divergent bit is specified for the 𝑘-th component of each point by the expression DnJointPosDiffBit[ dnPt ][ 𝑘 ].

DnJointPosDiffBit[dnPt][k] := bit << DnRemBits[k]  
 where bit :=  
 DnJointDiffBitPresent[k] ? direct\_joint\_diff\_bit[k] ^ dnPt :  
 DnJointDiffBitInferred[k] ? dnPt  
 : 0 /\* DnJointDiffBits[k] is 0 \*/

##### Presence of the syntax element direct\_joint\_diff\_bit

The presence of direct\_joint\_diff\_bit[ 𝑘 ] is specified by DnJointDiffBitPresent[ 𝑘 ].

DnJointDiffBitPresent[k] := DnJointDiffBits[k] > 0 && ¬DnJointDiffBitInferred[k]

It is present when a divergent bit position exists in the 𝑘-th component of the two jointly coded positions and the bit value cannot be inferred from the ordering constraint on jointly coded positions. The inference condition is specified by the expression DnJointDiffBitInferred[ 𝑘 ].

DnJointDiffBitInferred[k] :=  
 k == 0 && DnJointDiffBits[0] == 1  
 || k == 1 && DnJointDiffBits[1] == 1 && DnJointDiffBits[0] == 0  
 || k == 2 && DnJointDiffBits[2] == 1 && DnJointDiffBits[0] + DnJointDiffBits[1] == 0

#### Parsing of direct\_joint\_prefix

The binarization of direct\_joint\_prefix[ 𝑘 ] interleaves the prefix and suffix bins of the binarized exp-Golomb code (11.4.3). Every non-zero prefix bin shall be followed by a single suffix bin.

The binarization shall have no more than DnRemBits[ 𝑘 ] suffix bins.

maxVal = Exp2(DnRemBits[k])

The syntax element is parsed as follows; the variable BinIdx is the count of coded bins used for contextualization.

value = 1  
for (BinIdx = 0; value < maxVal && AeReadBin() == 1; BinIdx++)  
 BinIdx++  
 value = (value << 1) + AeReadBin()

### Angular coding

#### General

Subclause 9.2.13 applies when geom\_angular\_enabled is 1.

Angular coding in the occupancy tree specifies the contextualization for the planar occupied plane location occ\_plane\_pos, the direct position remainders direct\_rem\_st\_ang and direct\_rem\_v\_ang, the absolute value of the direct position vertical residual direct\_v\_ang\_resid\_abs and the sign of the direct position vertical residual direct\_v\_ang\_resid\_sign and the eligibility for direct coding. The contextualization uses rays cast from the angular origin.

#### Node position relative to the angular origin

The angular-origin-relative position of the occupancy tree node position is specified by the expression NposAng[ 𝑘 ].

NposAng[k] := AngPosScaleK(k, Nloc[k] << QuantizedNodeSizeLog2[k]) − AngularOrigin[k]

NposAngS := NposAng[0]  
NposAngT := NposAng[1]  
NposAngV := NposAng[2]

#### Azimuth coded axis

Contextualization using a beam's azimuth applies to either the S or T axis. The index of the contextualized axis is specified by the expression AzimuthAxis. It is determined using the angular-origin-relative node position.

AzimuthAxis := Abs(NposAngS) > Abs(NposAngT)

AzimuthAxisIsS := AzimuthAxis == 0  
AzimuthAxisIsT := AzimuthAxis == 1

#### State variables

Contextualization of occ\_plane\_pos[ AzimuthAxis ], beam\_idx\_resid\_abs[ dnPt ], beam\_idx\_resid\_sign[ dnPt ] and direct\_rem\_st\_ang is specified in terms of the following state variables; the index beamIdx identifies an SPS enumerated beam:

* The array BeamPrevPhiValid; BeamPrevPhiValid[ beamIdx ] indicates whether a value has been recorded by BeamPrevPhi[ beamIdx ].
* The array BeamPrevPhi; BeamPrevPhi[ beamIdx ] records the azimuth of the identified beam computed from the most recently coded syntax element direct\_rem\_st\_ang.
* The array BeamPrevIdxResid; BeamPrevIdxResid[ beamIdx ] records the residual BeamIdxResid[ dnPt ] between the enumerated beam DnBeamIdx[dnPt] and the per node prediction from estimated beam index DnBeamIdxEst, as specified from the most recently coded syntax elements beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ].

#### Initial state

At the start of every occupancy tree, all elements of BeamPrevPhiValid and all elements of BeamPrevIdxResid shall have the value 0.

#### Closest beam to a point

This subclause specifies the selection of the beam that can emit the closest rays to an angular-origin-relative point ( as, at, av ) by the expression BeamIdxEst[ as ][ at ][ av ]. The selection shall use rays cast from the angular origin.

When angular extension shall not be used or when performing beam selection for planar occupancy coding (9.2.13.7.3), the expression preciseBeamSelection is set equal to 0 and the selection is made without application of vertical beam displacements.

When angular extension shall be used and not performing beam selection for planar occupancy coding (9.2.13.7.3), the expression preciseBeamSelection is set equal to 1 and the selection is made by taking into account the vertical beam displacements.

BeamIdxEst[as][at][av] := num\_beams\_minus1 ?  
 (preciseBeamSelection ? PreciseBeamIdx : BeamIdxFromGrad[aGrad]) : 0

When preciseBeamSelection is equal to 0, the selection shall be performed by comparing the gradient of a ray specified by the expression aGrad to the elevation gradients of the enumerated beams. When preciseBeamSelection is equal to 1, the selection shall be performed by comparing for each enumerated beam the gradient of a ray specified by the expression displacedPointRayGrad[i] to the elevation gradient of the corresponding enumerated beam. According to the expression preciseBeamSelection, the ray is cast from the angular origin and passes either (when equal to 0) through the point ( as, at, av ), or (when equal to 1) through the point to which vertical beam displacement is applied ( as, at, av  + BeamOffsetV[i]).

Unless performing beam selection for planar occupancy coding (9.2.13.7.3), the expressions aGrad and preciseBeamSelection and displacedPointRayGrad[i] shall be defined as:

aGrad := av × IntRecipSqrt(rs × rs + rt × rv) >> 14  
 where  
 rs := as << 8  
 rt := at << 8

preciseBeamSelection := occtree\_angular\_extension\_enabled

displacedPointRayGrad[i] := ((av << 3) + BeamOffsetV[i])  
 × IntRecipSqrt(rs × rs + rt × rt) >> 17  
 where  
 rs := as << 8  
 rt := at << 8

When performing beam selection for planar occupancy coding (9.2.13.7.3), the expression aGrad shall be defined as:

aGrad := (2 × av − 1) × IntRecipSqrt(rs × rs + rt × rt) >> 15  
 where  
 rs := (as << 8) − 128  
 rt := (at << 8) − 128

preciseBeamSelection := 0

When preciseBeamSelection is equal to 0, the beam search is specified by the expression BeamIdxFromGrad. If the gradient rayGrad is half-way between that of two beams, the beam with the lower index shall be chosen.

BeamIdxFromGrad[rayGrad] :=  
 for (i = 1; i < num\_beams\_minus1; i++)  
 if (BeamElev[i] > rayGrad)  
 break  
 if (rayGrad − BeamElev[i − 1] ≤ BeamElev[i] − rayGrad)  
 i−−  
 BeamIdxFromGrad = i

When preciseBeamSelection is equal to 1, the precise beam search is specified by the expression PreciseBeamIdx. If the smallest value of gradientDistance[i] is obtained for at least two beams, the beam with the lower index shall be chosen.

PreciseBeamIdx :=  
 PreciseBeamIdx = 0  
 for (i = 1; i < num\_beams\_minus1 + 1; i++)  
 if (gradientDistance[i] < gradientDistance[PreciseBeamIdx])  
 PreciseBeamIdx = i  
 where  
 gradientDistance[i] := abs(displacedPointRayGrad[i] - BeamElev[i])

#### Application to occupied plane location coding

##### Node dimensions and midpoint

The expression ScaledNodeSize[ 𝑘 ] specifies the scaled volume dimensions of the coded node.

ScaledNodeSize[k] := AngPosScaleK(k, Exp2(QuantizedNodeSizeLog2[k]))

ScaledNodeSizeS := ScaledNodeSize[0]  
ScaledNodeSizeT := ScaledNodeSize[1]  
ScaledNodeSizeV := ScaledNodeSize[2]

The expression ScaledHalfNodeSize[ 𝑘 ] specifies the midpoint within the scaled volume of the coded node.

ScaledHalfNodeSize[k] := AngPosScaleK(k, Exp2(QuantizedNodeSizeLog2[k]) >> 1)

ScaledHalfNodeSizeS := ScaledHalfNodeSize[0]  
ScaledHalfNodeSizeT := ScaledHalfNodeSize[1]  
ScaledHalfNodeSizeV := ScaledHalfNodeSize[2]

When geometry scaling is disabled, ScaledNodeSize[ 𝑘 ] and ScaledHalfNodeSize[ 𝑘 ] are equal to Exp2( NodeSizeLog2[ 𝑘 ] ) and Exp2( NodeSizeLog2[ 𝑘 ] >> 1 ) respectively.

The midpoint coordinates of the coded node relative to the angular origin are specified by the expression NposAngMid[ 𝑘 ].

NposAngMid[k] := NposAng[k] + ScaledHalfNodeSize[k]

NposAngMidS := NposAngMid[0]  
NposAngMidT := NposAngMid[1]  
NposAngMidV := NposAngMid[2]

##### Eligibility

Only certain nodes are eligible for angular contextualization of the occupied plane location and inter direct coding. They are specified by the expression AngularEligible. Eligibility prevents the use of angular contextualization when a node is sufficiently large to be intersected by rays from multiple beams.

AngularEligible := geom\_angular\_enabled  
 && (¬num\_beams\_minus1 || CathetusV > (ScaledHalfNodeSizeV << 26))

Eligibility is specified in terms of:

* the smallest difference in elevation gradient between any two beams, BeamMinDeltaGrad;
* the V-axis distance subtended (CathetusV) by a ray with elevation gradient BeamMinDeltaGrad, over the distance in the S-T plane from the angular origin to the node midpoint; and
* the 𝑣-component of the scaled node size.

An example of an eligible node is illustrated in Figure 14.

手机屏幕截图

中度可信度描述已自动生成

Figure 14 — Illustration of an eligible node.

CathetusV := BeamMinDeltaGrad × ((rs + rt) >> 1)  
 where  
 rs := Abs((NposAngMidS << 8) − 128)  
 rt := Abs((NposAngMidT << 8) − 128)

The expression BeamMinDeltaGrad is the smallest difference in elevation gradient between any two beams.

BeamMinDeltaGrad :=  
 BeamMinDeltaGrad = BeamElev[0]  
 for (i = 1; i ≤ num\_beams\_minus1; i++) {  
 delta = BeamElev[i] – BeamElev[i − 1]  
 if (delta < BeamMinDeltaGrad)  
 BeamMinDeltaGrad = delta  
 }

Only certain nodes are eligible for azimuth direct coding. They are specified by the expression AzimuthEligible.

AzimuthEligible := AngularEligible && ((HalfDeltaAziAng << 1) ≤ phiStep)  
where  
 phiStep:= 6588397 / BeamStepsPerRev[PlanarBeamIdx]

HalfDeltaAziAng := Abs(azimuthLowMId - azimuthMid)  
where  
 azimuthLowMId := AzimuthAxis ? IntAtan2(NposAngMidT, NposAngS)  
 : IntAtan2(NposAngT, NposAngMidS)  
 azimuthMid := IntAtan2(NposAngMidT, NposAngMidS)

##### Beam selection

Every node that is eligible for angular contextualization (AngularEligible == 1) shall select a beam for that purpose. The selected beam is specified by PlanarBeamIdx. It is either the same beam as used for the parent node or the closest beam to the node midpoint (9.2.13.6).

PlanarBeamIdx :=  
 inheritBeam ? NodeBeamIdx[Dpth − 1][NsP][NtP][NvP]  
 : BeamIdxEst[NposAngMidS][NposAngMidT][NposAngMidV]

The beam selection is inherited from the parent node when both the parent node has a valid beam and the node is not too small, as specified by the expression inheritBeam.

inheritBeam :=  
 NodeBeamValid[Dpth − 1][NsP][NtP][NvP] && CathetusV > (ScaledHalfNodeSizeV << 28)

##### State variables

Beam selection is specified in terms of the following state variables; the indexes dpth, ns, nt and nv identify a node with location ( ns, nt, nv ) in the tree level at depth dpth:

* The sparse array NodeBeamValid; NodeBeamValid[ dpth ][ ns ][ nt ][ nv ] indicates whether (when 1) the identified node has a valid beam selection. Unset elements are inferred to be 0.
* The sparse array NodeBeamIdx; NodeBeamIdx[ dpth ][ ns ][ nt ][ nv ] is the selected beam, if valid, for the identified node.

##### Initial state

At the start of every occupancy tree, elements of NodeBeamValid shall be unset.

##### State update

After selecting a beam (9.2.13.7.3), the beam index and its validity shall be recorded for inheritance by child nodes.

NodeBeamIdx[Dpth][Ns][Nt][Nv] = PlanarBeamIdx  
NodeBeamValid[Dpth][Ns][Nt][Nv] = 1

##### Contextualization of occ\_plane\_pos[ AzimuthAxis ]

The syntax element occ\_plane\_pos[ AzimuthAzis ] shall be contextualized according to 9.2.13.9 with the argument phiRangeMulLog2 equal to 2 and the arguments beamIdx, phiNominal, phiLow and phiHigh as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = PlanarBeamIdx

The arguments phiLow and phiHigh are the azimuthal angles for the angular-origin-relative coordinates of the lower corner of the occupancy tree node in the S-T plane for phiLow, and of the scaled node midpoint in the S-T plane for phiHigh. The argument phiNominal is a nominal azimuthal angle use during contextualization and it is set equal to phiLow.

phiLow = IntAtan2(NposAngT, NposAngS)  
phiHigh = IntAtan2(NposAngMidT, NposAngMidS)  
phiNominal = phiLow

##### Contextualization of occ\_plane\_pos[ 2 ]

The syntax element occ\_plane\_pos[ 2 ] shall be contextualized according to 9.2.13.10 with the argument elvIntvlGradDivLog2 equal to 2 and the arguments beamIdx, elvIntvlPosS, elvIntvlPosT, elvIntvlMidV and elvIntvlLenV as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = PlanarBeamIdx

The arguments elvIntvlPosS, elvIntvlPosT and elvIntvlMidV are the angular-origin-relative coordinates of the scaled node midpoint. The argument elvIntvlLenV is the length of the V-axis interval represented by the syntax element

elvIntvlPosS = NposAngMidS  
elvIntvlPosT = NposAngMidT  
elvIntvlMidV = NposAngMidV  
elvIntvlLenV = ScaledNodeSizeV

#### Application to direct node position coding

##### Eligibility

Only certain occupancy tree nodes are eligible to be direct nodes. They are specified by the expression DirectModeEligible (9.2.12.3.1).

When occtree\_inter\_angular\_direct\_coding\_enabled is equal to 1, eligibility for direct nodes shall be performed by the expression DnEligibleByAng.

DnEligibleByAng:= occtree\_inter\_angular\_direct\_coding\_enabled ?  (geom\_dup\_point\_counts\_enabled ?  
 AngularEligible && AzimuthEligible: AngularEligible || AzimuthEligible): 0

Only certain occupancy tree nodes are eligible to be inter direct nodes. They are specified by the expression interPredEligible.

The reference node indicates the colocated node in the reference frameof the current node.

*interNumPredPoints* indicates the number of points within the reference node of the current node.

interPredMode is true, when the reference node has at most one point or all points position are same.

interPredEligible is true, when the follwing conditions are true:

* *inter\_prediction\_enabled* is true and
* interPredMode is true and
* interNumPredPoints larger than 0

interPredEligible:=  
 inter\_prediction\_enabled&& interPredMode && interNumPredPoints >=0

[Ed. (HH): Using inter\_prediction\_enabled instead of occtree\_inter\_angular\_direct\_coding\_enabled might result misguiding inter direct coding. – This needs to be discussed.]

##### Beam selection

Context selection for direct\_rem\_st\_ang[ dnPt ] and direct\_rem\_v\_ang[ dnPt ] and 𝑣-component of dnPt-th point prediction to be used with DirectVAngResid[ dnPt ] shall be performed using the beam specified by DnBeamIdx[ dnPt ]. For eligible node (9.2.12.3), an estimated beam index is determined for the dnPt-th point within the current node. The estimated beam index is specified by the expression DnBeamIdx[ dnPt ].

It is a requirement of bitstream conformance that DnBeamIdx[ dnPt ] shall be in the range 0 .. num\_beams\_minus1.

DnBeamIdx[dnPt] := (interPredEligible ? interDnBeamIdxEst:DnBeamIdxEst) + BeamIdxResid[dnPt]

Every direct node shall determine an estimated beam index as specified by the expression DnBeamIdxEst. The beam is estimated from the angular-origin-relative, scaled partial point position specified by dnBeamPosAng[ 𝑘 ].

DnBeamIdxEst := BeamIdxEst[beamPosAngS][beamPosAngT][beamPosAngV]  
 where  
 beamPosAngS := dnBeamPosAng[0]  
 beamPosAngT := dnBeamPosAng[1]  
 beamPosAngV := dnBeamPosAng[2]

dnBeamPosAng[k] := NposAng[k] + AngPosScaleK(k, DnPlanarPos[k] + dnPlanarPosHalf)  
 where  
 dnPlanarPosHalf := Exp2(DnBitsAfterPlanar[k]) >> 1

When interPredEligible is equal to 1, every point shall determine an estimated beam index as specified by the expression interDnBeamIdxEst. The beam is estimated from the angular-origin-relative, scaled reference predictor point position specified by interDnBeamPosAng[ 𝑘 ].

inter  
DnBeamIdxEst := BeamIdxEst[beamPosAngS][beamPosAngT][beamPosAngV]  
 where  
 beamPosAngS := interDnBeamPosAng[0]  
 beamPosAngT := interDnBeamPosAng[1]  
 beamPosAngV := interDnBeamPosAng[2]

interDnBeamPosAng[k]:= AngPosScaleK(k, interDnPlanarPos[k])  
 where  
 interDnPlanarPos:= PredGMPointCloud[*predDnPt*]   
 *predDnPt*:= dnPt< predEnd? dnPt : predEnd -1

##### Scaled angular-origin-relative point positions and partial point positions

Contextualization of direct\_rem\_st\_ang and direct\_rem\_v\_ang depends upon approximately scaled partially coded positions. A partially coded position is specified in terms of the MSBs of the complete coded position.

Approximate scaling accumulates applications of geometry scaling to each bit in the partially coded position.

The expression DnPtPosAng[ dnPt ][ 𝑘 ] is the 𝑘-th component of the scaled, coded point position relative to the angular origin.

DnPtPosAng[dnPt][k] := AngPosScaleK(k, DnPtPos[dnPt][k]) − AngularOrigin[k]

DnPtPosAngS[dnPt] := DnPtPosAng[dnPt][0]  
DnPtPosAngT[dnPt] := DnPtPosAng[dnPt][1]

The expression DnPartialPosAng[ dnPt ][ remBits ][ 𝑘 ] is the 𝑘-th component of the approximately scaled partially coded position for the dnPt-th coded point of the direct node. It excludes the remBits LSBs of DnPtPos[ dnPt ][ 𝑘 ].

DnPartialPosAng[dnPt][remBits][k] :=  
 DnPartialPosAng = NposAng[k]  
 for (i = remBits; i < QuantizedNodeSizeLog2[k]; i++)  
 DnPartialPosAng += AngPosScaleK(k, DnPtPosRem[dnPt][k] & Exp2(i))

##### Binarization of direct\_rem\_st\_ang and direct\_rem\_v\_ang

The direct\_rem\_st\_ang and direct\_rem\_v\_ang syntax elements shall be entropy coded using fixed-length binarization. The length in bins shall be equal to the log2 quantized node size less any bit inferred by the presence of a single occupied plane.

The expression DnRemAngBitsST is the length in bins of the syntax element direct\_rem\_st\_ang.

DnRemAngBitsST := DnBitsAfterPlanar[AzimuthAxis]

The expression DnRemAngBitsV is the length in bins of the syntax element direct\_rem\_v\_ang.

DnRemAngBitsV := DnBitsAfterPlanar[2]

##### Contextualization of direct\_rem\_st\_ang

Each bin, identified by BinIdx, of direct\_rem\_st\_ang[ dnPt ] shall be contextualized according to 9.2.13.9 with the argument phiRangeMulLog2 equal to 1 and the arguments beamIdx, phiNominal, phiLow and phiHigh as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = DnBeamIdx[dnPt]

The expression PhiLowST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the lower end of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it is computed from coordinates in the S-T plane, otherwise it may be estimated.

The expression PhiHighST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the higher end of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it has no utility, otherwise it may be estimated.

The expression PhiMidST[ remBits ] is the azimuthal angle for angular-origin-relative coordinates of the mid point of the interval in the S-T plane, according to the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ]. When occupancy tree angular extension is not enabled it is computed from coordinates in the S-T plane, otherwise it may be estimated.

The expressions PhiPosS[ remBits ] and PhiPosT[ remBits ] are the angular-origin-relative coordinates of the lower end of the interval coded by a bin in the S-T plane, the bin index being expressed by the number of remaining bins remBits for direct\_rem\_st\_ang[ dnPt ].

PhiPosS[remBits] := AzimuthAxisIsS ? DnPartialPosAngS[dnPt][remBits] : DnPtPosAngS[dnPt]  
  
PhiPosT[remBits] := AzimuthAxisIsT ? DnPartialPosAngT[dnPt][remBits] : DnPtPosAngT[dnPt]

PhiLowST[remBits] :=  
 occtree\_angular\_extension\_enabled && remBits < DnRemAngBitsST ?  
 (prevBitIs1 ? PhiMidST[remBits + 1] : PhiLowST[remBits + 1]) :  
 IntAtan2(phiIntvlLowT, phiIntvlLowS)  
 where  
 prevBitIs1 := (direct\_rem\_st\_ang[dnPt] & (1 << remBits)  
 phiIntvlLowS := PhiPosS[remBits]  
 phiIntvlLowT := PhiPosT[remBits]

PhiHighST[remBits] :=  
 occtree\_angular\_extension\_enabled && remBits < DnRemAngBitsST ?  
 (prevBitIs1 ? PhiHighST[remBits + 1] : PhiMidST[remBits + 1]) :  
 IntAtan2(phiIntvlHighT, phiIntvlHighS)  
 where  
 prevBitIs1 := (direct\_rem\_st\_ang[dnPt] & (1 << remBits)  
 binIdx := DnRemAngBitsST - remBits  
 phiIntvlLen := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> binIdx)  
 phiIntvlHighS = phiPosS[remBits] + (AzimuthAxisIsS ? phiIntvlLen : 0)  
 phiIntvlHighT = phiPosT[remBits] + (AzimuthAxisIsT ? phiIntvlLen : 0)

PhiMidST[remBits] :=  
 occtree\_angular\_extension\_enabled  
 && (Abs(PhiLowST[remBits] - PhiHighST[remBits]) < phiMidInterpThreshold ?  
 PhiLowST[remBits] + PhiHighST[remBits] >> 1 :  
 IntAtan2(phiIntvlMidT, phiIntvlMidS)  
 where  
 binIdx := DnRemAngBitsST - remBits  
 phiMidInterpThreshold := 1 << 13  
 phiHalfIntvlLen := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> 1 + binIdx)  
 phiIntvlMidS = PhiPosS[remBits] + (AzimuthAxisIsS ? phiHalfIntvlLen : 0)  
 phiIntvlMidT = PhiPosT[remBits] + (AzimuthAxisIsT ? phiHalfIntvlLen : 0)

The expression phiPartialSTBits is the number of remaining bins for direct\_rem\_st\_ang[ dnPt ], including the bin identified by BinIdx.

phiPartialSTBits := DnRemAngBitsST – BinIdx

The expression phiOneFourth and phiThreeFourth are azimuthal angles estimated for angular-origin-relative coordinates of points estimated around one fourth and around three fourth of the interval in the S-T plane.

phiOneFourth := phiIntvlLow  
 + ((offset0 – offset1) × (phiIntvlMid - phiIntvlLow) >> phiPartialSTBits)  
phiThreeFourth := phiIntvlMid  
 + ((offset0 + offset1) × (phiIntvlMid - phiIntvlLow) >> phiPartialSTBits)  
 where  
 phiIntvlLow := PhiLowST[ phiPartialSTBits ]  
 phiIntvlMid := PhiMidST[ phiPartialSTBits ]  
 offset0 := AngPosScaleK(AzimuthAxis, Exp2(DnRemAngBitsST) >> 1 + BinIdx) - 1  
 offset1 := phiPartialSTBits > 1 ? phiPartialSTBits > 2 ? 0 : 1 : 2

When occupancy tree angular extension is enabled, the argument phiLow is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of a point estimated around one fourth of the interval in the S-T plane, otherwise phiLow is set equal to an azimuthal angle for angular-origin-relative coordinates of the lower end of the interval in the S-T plane. When occupancy tree angular extension is enabled, the argument phiHigh is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of a point estimated around three fourth of the interval in the S-T plane, otherwise phiHigh is set equal to an azimuthal angle for angular-origin-relative coordinates of the mid point of the interval in the S-T plane. When occupancy tree angular extension is enabled, the argument phiNominal is set equal to an azimuthal angle estimated for angular-origin-relative coordinates of the mid point of the interval in the S-T plane, otherwise phiNominal is set equal to phiLow.

phiLow = occtree\_angular\_extension\_enabled ? phiOneFourth : PhiLowST[phiPartialSTBits]  
phiHigh = occtree\_angular\_extension\_enabled ? phiThreeFourth : PhiMidST[phiPartialSTBits]  
phiNominal = occtree\_angular\_extension\_enabled ? PhiMidST[phiPartialSTBits] : phiLow

##### State update after each direct\_rem\_st\_ang syntax element

After each coded direct\_rem\_st\_ang[ dnPt ] syntax element, the azimuth of the coded point shall be recorded as the azimuth of the selected beam.

BeamPrevPhi[DnBeamIdx[dnPt]] = PhiLowST[ 0]  
BeamPrevPhiValid[DnBeamIdx[dnPt]] = 1

##### Contextualization of direct\_rem\_v\_ang

Each bin, identified by BinIdx, of direct\_rem\_v\_ang[ dnPt ] shall be contextualized according to 9.2.13.10 with the arguments beamIdx, elvIntvlPosS, elvIntvlPosT, elvIntvlLenV, elvIntvlMidV and elvIntvlGradDivLog2 as specified in this subclause.

The argument beamIdx is the index of the selected beam.

beamIdx = DnBeamIdx[dnPt]

The arguments elvIntvlPosS and elvIntvlPosT are the angular-origin-relative coordinates in the S-T plane of the approximately scaled position of the coded point. The argument elvIntvlLenV is the length of the interval represented by the syntax element before division by Exp2( elvIntvlGradDivLog2 ).

elvIntvlPosS = AzimuthAxisIsS ? DnPartialPosAngS[dnPt][0] : DnPtPosAngS[dnPt]  
elvIntvlPosT = AzimuthAxisIsT ? DnPartialPosAngT[dnPt][0] : DnPtPosAngT[dnPt]  
elvIntvlLenV = AngPosScaleV(Exp2(DnRemAngBitsV))  
elvIntvlGradDivLog2 = BinIdx

When geometry scaling is disabled or the scaling step size is a power of two, DtPartialPosAng[ dnPt ][ 0 ][ 𝑘 ] is equal to DnPtPosAng[ dnPt ][ 𝑘 ].

The argument elvIntvlMidV specifies the midpoint of the V-axis interval relative to the angular origin. The expression elvIntvlHalfLen is the half-interval range. The start of the V-axis interval, elvPartialPosV, is the partially coded and approximately scaled point 𝑣-coordinate.

elvPartialPosV := DnPartialPosAngV[dnPt][DnRemAngBitsV − BinIdx]  
elvIntvlHalfLen := AngPosScaleV(Exp2(DnRemAngBitsV) >> 1 + BinIdx)  
elvIntvlMidV = elvPartialPosV + elvIntvlHalfLen

##### Binarization of direct\_v\_ang\_resid\_abs and direct\_v\_ang\_resid\_sign

[Ed. (JT): Consider moving this subclause to an appropriate location.]

When angular coding extension shall be used, direct\_v\_ang\_resid\_abs syntax element is the absolute value of a residual of a prediction made from the selected beam elevation and vertical offset and shall be entropy coded using a concatenated truncated unary and 𝑘-th order exp-Golomb code 11.4.4 (TU+EGk) and direct\_v\_ang\_resid\_sign is its sign and shall be entropy coded when the absolute value of the residual is not null.

The prediction of the 𝑣-component of dnPt-th point relative to the angular origin is specified by the expression DirectVAngPred[ dnPt ]. It clips a prediction made from the selected beam to fit in V-axis interval relative to the angular origin in which 𝑣-component of dnPt-th shall be refined.

DirectVAngPred[dnPt] :=  
 Min(Max(directVAngPred0[dnPt], dnPosAng), dnPosAng + dnRemNodeSizeV - 1)  
 where  
 dnPosAngV := NposAng[2] + AngPosScaleK(2, DnPlanarPos[2])  
 dnRemNodeSizeV := 1 << DnResVAngPredBits  
 directVAngPred0[dnPt] :=  
 DivExp2Fz(radius \* BeamElev[beamIdx] - (BeamOffsetV[beamIdx] << 23), 26 )  
 where  
 radius := IntSqrt(rs × rs + rt × rt)  
 where  
 rs := elvIntvlPosS << 8  
 rt := elvIntvlPosT << 8

The arguments elvIntvlPosS and elvIntvlPosT are the angular-origin-relative coordinates in the S-T plane of the approximately scaled position of the coded point.

elvIntvlPosS = AzimuthAxisIsS ? DnPartialPosAngS[dnPt][0] : DnPtPosAngS[dnPt]  
elvIntvlPosT = AzimuthAxisIsT ? DnPartialPosAngT[dnPt][0] : DnPtPosAngT[dnPt]

The expression DnResVAngPredBits is the length in bits of the expression DirectRemVAngPred[ dnPt ].

DnResVAngPredBits := DnBitsAfterPlanar[2]

The expression DirectRemVAngPred[ dnPt ] specify the remaining position bits of the 𝑣-component of the dnPt-th point when angular coding extension shall be used.

DirectRemVAngPred[dnPt] :=  
 DirectVAngPred[dnPt] + DirectVAngResid[dnPt] - NposAng[2]

It is a requirement of bitstream conformance that DirectRemVAngPred[ dnPt ] shall be in the range 0 .. Exp2(DnResVAngPredBits) – 1.

##### State update after each **beam\_idx\_resid\_abs** and **beam\_idx\_resid\_sign** syntax elements

When angular extension coding shall be used, after each coded beam\_idx\_resid\_abs[ dnPt ] and beam\_idx\_resid\_sign[ dnPt ] syntax elements, the residual between the enumerated beam and the per node prediction specified by the expression BeamIdxResid[ dnPt ] shall be recorded as the most recently coded residual of the selected beam.

if(occtree\_angular\_extension\_enabled)  
 BeamPrevIdxResid[DnBeamIdxEst] = BeamIdxResid[ dnPt ]

#### Determination of CtxIdxAngPhi for a bin according to beam azimuth

Each contextualized bin represents a half-closed interval along the azimuth coding axis AzimuthAxis located along the other S-T plane axis at a distance from the angular origin. The bin values 1 and 0 identify the upper and lower halves of the interval respectively.

Contextualization discriminates by the intersection of a predicted ray with one of eight parameterized azimuth ranges.

The process is parameterized by:

* beamIdx, the index of the beam that casts the predicted ray;
* phiNominal represents a nominal azimuthal angle for the coded point;
* phiLow and phiHigh represents (possibly estimated) azimuthal angles for a point in the first and second half of the half-closed interval;
* phiRangeMulLog2, a syntax element dependent constant used to define the azimuth discrimination ranges.

The predicted ray azimuth is derived from the azimuth of a reference point. The prediction quantizes the nominal azimuth to be an integer number of azimuth steps from the reference point.

The reference azimuth for the predicted ray is that of the previous direct-node-coded point recorded by the selected beam (9.2.13.8.5); if no previous point has been recorded by the selected beam, the reference azimuth is the nominal azimuth.

phiRef := BeamPrevPhiValid[beamIdx] ? BeamPrevPhi[beamIdx] : phiNominal

The expression phiPred specifies the predicted ray azimuth.

phiPred := phiRef − beamPhiStep[beamIdx] × phiSteps  
 where  
 phiSteps := DivExp2Up((phiRef − phiNominal) × BeamPhiStepRecip[beamIdx], 30)  
 beamPhiStep := 6588397 / BeamStepsPerRev[beamIdx]  
 beamPhiStepRecip := (BeamStepsPerRev[beamIdx] << 30) / 6588397

The context selection for the coded bin is specified by the expression CtxIdxAngPhi.

CtxIdxAngPhi := (a + 2 × b + 4 × c) × 3 + d  
 where  
 dPhiL := phiLow − phiPred  
 dPhiH := phiHigh − phiPred  
 a := (dPhiL ≥ 0) == (dPhiH ≥ 0)  
 b := Abs(dPhiL) > Abs(dPhiH)  
 c := Max(Abs(dPhiL), Abs(dPhiH)) > (Min(Abs(dPhiL),Abs(dPhiH)) << phiRangeMulLog2)  
 d := occtree\_angular\_extension\_enabled ?   
 (3 × phiStep < 4 × adPhiLH) + (phiStep < 2 × adPhiLH) : 0  
 where  
 phiStep := beamPhiStep[beamIdx]  
 adPhiLH := Abs(phiLow – phiHigh)

#### Determination of CtxIdxAngTheta for a bin according to beam elevation

Each contextualized bin represents a half-closed V-axis interval located at a radial distance in the S-T plane from the angular origin. The bin values 1 and 0 identify the upper and lower halves of the interval respectively.

Contextualization discriminates by the intersection of a predicted ray cast by a beam with one of four parameterized V-axis intervals.

The process is parameterized by:

* beamIdx, the index of the beam that casts the predicted ray;
* elvIntvlPosS and elvIntvlPosT, the coordinates of the V-axis interval in the S-T plane, relative to the angular origin;
* elvIntvlLenV, the length of the V-axis interval;
* elvIntvlMidV, the midpoint of the V-axis interval relative to the angular origin;
* elvIntvlGradDivLog2, the factor used to derive the sub-interval size.

The expression rRecip specifies the reciprocal radial distance of the V-axis interval in the S-T plane from the angular origin.

rRecip := IntRecipSqrt(rs × rs + rt × rt)  
 where  
 rs := (elvIntvlPosS << 8) − 128  
 rt := (elvIntvlPosT << 8) − 128

The expression elvIntvlMidGrad specifies the gradient of a ray cast from the angular origin that intersects the midpoint of the V-axis interval.

elvIntvlMidGrad := (2 × elvIntvlMidV − 1) × rRecip / Exp2(15)

The expressions esoTopGrad and esoBotGrad specify the gradients of rays cast from the angular origin that intersect the end points of a V-axis sub-interval.

esoIntvlGrad := elvIntvlLenV × rRecip >> 18 + elvIntvlGradDivLog2  
esoTopGrad := elvIntvlMidGrad + esoIntvlGrad  
esoBotGrad := elvIntvlMidGrad − esoIntvlGrad

The expression elvBeamGrad specifies the gradient of a ray cast from the angular origin that would intersect the predicted ray at the given radial distance.

elvBeamGrad := BeamGrad[beamIdx] + BeamOffsetV[beamIdx] × rRecip / Exp2(17)

The context selection for the coded bin is specified by the expression CtxIdxAngTheta, comparing the gradients of rays cast from the angular origin to those of the threshold points.

CtxIdxAngTheta := a + 2 × b  
 where  
 a := elvBeamGrad ≥ elvIntvlMidGrad  
 b := elvBeamGrad ≥ esoTopGrad || elvBeamGrad < esoBotGrad

#### Conversion from Cartesian to angular coordinates

This subclause specifies the conversion of decoded points' positions to angular coordinates.

A decoder is not required to perform the conversion unless angular coordinates are required for attribute decoding.

The angular-origin-relative position of every point is specified by the expression posAng[ ptIdx ][ 𝑘 ].

posAng[ptIdx][k] := PointPos[ptIdx][k] − AngularOrigin[k]

Every angular-origin-relative point position is converted to angular coordinates. The radial distance is the Euclidean distance to the point in the S-T plane; the azimuth angle is the anti-clockwise angle in an S-T plane between the positive S axis and the point; the indexed elevation is the index of the beam that casts the closest rays to the point (9.2.13.6). The conversion is:

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++){  
 as = posAng[ptIdx][0]  
 at = posAng[ptIdx][1]  
 av = posAng[ptIdx][2]  
 PointAng[ptIdx][0] = IntSqrt(as × as + at × at << 16) >> 8  
 PointAng[ptIdx][1] = IntAtan2(at << 8, as << 8) + 3294199 >> 8  
 PointAng[ptIdx][2] = BeamIdxEst[as][at][av]  
}

### Subtree scaling

#### Partially quantized coordinates

Subclause 9.2.14 applies when geom\_scaling\_enabled is 1.

Certain subtrees (9.2.14.2.1) can represent point positions using partially quantized coordinates that are parameterized by a scaling volume (9.2.14.2.2) and an applicable QP (9.2.14.5).

The partially quantized representation comprises two concatenated parts:

* An upper part that, when scaled, is the position of the scaling volume's lower corner, . Scaling is by the scaling volume size rounded down to the next power of two.
* A lower part that, when scaled, is a position relative to and within the scaling volume. Scaling is by the QP derived scale factor.

An example of the representation is illustrated by Figure 15. The coded point ● is represented within the scaling volume A. The scaling volume size is 12; rounded down to the next power of two as 8 (). The QP is 12; the derived scale factor (OccQpScale) is 3. The two parts of the point's partially quantized 𝑠-coordinate are marked U and L:

* The upper part, U, scaled by , is the position of A.
* The lower part, L, scaled by OccQpScale = 3 is the relative point position; for the S axis, 2 × 3 = 6. Scaling expands L by 1 bit (OccQpScaleLog2) to be 3 bits (OccScalingVolSizeLog2) in total.
* If U were 5, then would be 5 ×  = 40, and the point's 𝑠-coordinate would be 46.

图表, 图示

描述已自动生成

Key

|  |  |
| --- | --- |
| A | Scaling volume |
| U, L | Upper and lower parts of partially quantized representation |
|  | 𝑖-th bit |

Figure 15 — Example of partially quantized coordinates. Illustrated in 2D for the S-V plane.

#### Quantized subtrees

##### Identification

Subtrees that code partially quantized coordinates are started by:

* every node at depth OccQpSubtreeDepth;
* every direct node at a shallower depth than OccQpSubtreeDepth; in this case, the subtree consists only of the subtree root node as a direct node.

The lower parts of partially quantized point coordinates are represented by the location of a leaf node in the subtree or by a direct node that codes a subtree-relative position.

Identification of a subtree from a direct node at a shallower depth than OccQpSubtreeDepth happens after any planar occupancy coding for that node (9.2.14.2.5).

##### Subtree scaling volume

A scaling volume is identified by every quantized subtree. Its size depends upon the tree level in which the quantized subtree starts:

* If the quantized subtree starts at a shallower depth than OccQpSubtreeDepth and planar occupancy coding is enabled, the scaling volume dimensions are equal to those of the scaled child-node volume of the subtree root node.
* Otherwise, the scaling volume dimensions are identical to those of the scaled subtree root node volume.

The expression OccScalingVolSizeLog2[ 𝑘 ] specifies the integer log2 size of the scaling volume. The scaled dimensions (9.2.14.2.3) are RoundUp( OccVolScale × Exp2( OccScalingVolSizeLog2[ 𝑘 ] ) ).

OccScalingVolSizeLog2[k] :=  
 Dpth ≥ OccQpSubtreeDepth ? OccLvlNodeSizeLog2[OccQpSubtreeDepth[k] :  
 occtree\_planar\_enabled ? ChildNodeSizeLog2[k] : NodeSizeLog2[k]

For example, a quantized subtree root node identifies a volume with dimensions 32×16×32 and an applicable QP of 10 (OccQpScale = 2.5). The quantized node size would be 16×8×16 and the scaled node size 40×20×40.

##### Scaled occupancy tree node volume dimensions

A scaled node volume is the geometric expansion of the volume associated with a node in the general occupancy tree (9.2.2.1). The expansion is by a QP derived scale factor for the subtree, OccVolScale, and is relative to the lower corner, , of the subtree scaling volume. Scaling by OccVolScale rounds half-values up.

The quantized node size QuantizedNodeSize is a power-of-two contraction by OccQpScaleLog2 of the node size for ordinary nodes in the tree level.

OccQpScaleLog2 := OccQp / 8  
  
QuantizedNodeSizeLog2[k] := Max(0, NodeSizeLog2[k] − OccQpScaleLog2)  
  
QuantizedChildNodeSizeLog2[k] := Max(0, ChildNodeSizeLog2[k] − OccQpScaleLog2)

The scaled node volume dimensions are equivalent to the rounded expansion of the quantized node size by the subtree scale factor OccQpScale.

##### Quantized subtree nodes

A node in a quantized subtree shall identify the presence of at least one point contained within the scaled node volume.

Leaf nodes in the quantized subtree represent indivisible volumes with dimensions equal to the unit cube scaled by the subtree scale factor. Scaled point positions outside the unscaled subtree volume shall be clipped to be within it (hatched area in Figure 15). For example, point ▪︎ in the figure is clipped to ▫︎.

Two example subtrees are illustrated over three tree levels by Figure 16. The subtree root node volumes have dashed outlines; the coded nodes in each level have a thick, solid outline. The top subtree uses QP 12, the bottom QP 0. The position of the point ● is coded, starting from depth 𝑛, by the locations of the grey child nodes. The top subtree identifies its leaf node at depth 𝑛 + 1, the bottom subtree at depth 𝑛 + 2.

图示

描述已自动生成

Figure 16 — Example decomposition of two quantized subtrees.  
QP 12 (top) and QP 0 (bottom).

##### Direct nodes before OccQpSubtreeDepth

This subclause applies when planar occupancy coding is enabled.

Until a node is identified as a direct node through the coding of occ\_direct\_node, it shall not be considered to start a quantized subtree.

Coding operations that are specified to occur:

* before occ\_direct\_node shall use the ordinary node volume, equivalent to QP 0;
* after occ\_direct\_node shall use the scaled node volume derived from the direct node QP (DnQp),

i.e. the QP and scaled node volume dimensions change during the coding of the node.

This case is illustrated by Figure 17. When planar occupancy coding is enabled (left), the scaled node dimensions are initially equal to NodeSize, but changes to OccScalingVolSize after occ\_direct\_node.

手机屏幕截图

描述已自动生成

Figure 17 — Example subtree scaling volume for subtrees shallower than OccQpSubtreeDepth. (Left) planar occupancy coding enabled; (Right) disabled.

##### Fully quantized subtrees

Sufficiently large subtree QPs can cause a subtree root node to have a log2 quantized node size of 0×0×0. In this case, the subtree root node is a terminal node that:

* is ineligible to be a direct node (9.2.12.3.1)
* has its points represented by a single implicit child leaf node (9.2.6.3); the child node has the same dimensions as the root node – i.e. no axes are coded

#### Syntax element semantics

occ\_subtree\_qp\_offset\_present specifies whether (when 1) or not (when 0) per-subtree QP offsets are present in the tree level. QP offsets can only be present in a single tree level. When occ\_subtree\_qp\_offset\_present is not present, it shall be inferred to be 0.

occ\_subtree\_qp\_offset\_abs[ ns ][ nt ][ nv ] and occ\_subtree\_qp\_offset\_sign[ ns ][ nt ][ nv ] together specify an offset to the slice geometry QP used to scale point positions. The offset is specified by the expression OccSubtreeQpOffset[ ns ][ nt ][ nv ]. The offset applies to all points coded in the subtree of the node located at ( ns, nt, nv ) in the tree level at depth OccQpSubtreeDepth. When occ\_subtree\_qp\_offset\_sign[ ns ][ nt ][ nv ] is not present, it shall be inferred to be 0.

OccSubtreeQpOffset[ns][nt][nv] :=  
 (1 − 2 × occ\_subtree\_qp\_offset\_sign[ns][nt][nv]) × occ\_subtree\_qp\_offset\_abs[ns][nt][nv]

#### State variables

Occupancy tree scaling is specified in terms of the following state variable:

* The variable OccQpSubtreeDepth that identifies the tree level where occ\_subtree\_qp\_offset\_present is 1.

#### The subtree QP

##### General

The subtree QP is specified by the expression OccQp. It is determined for a coded node as:

* when geometry scaling is disabled: 0;
* when the node is a direct node at a depth shallower than OccQpSubtreeDepth: specified by 9.2.14.5.2;
* when the node is at a depth equal to or deeper than OccQpSubtreeDepth: specified by 9.2.14.5.3.

The provisions of 9.2.14.2.5 affect evaluations of OccQp during certain nodes.

OccQp :=  
 ¬geom\_scaling\_enabled ? 0 :  
 Dpth ≥ OccQpSubtreeDepth ? OccSubtreeQp  
 occ\_direct\_node ? DnQp : 0

##### Determination for a direct node before OccQpSubtreeDepth

The subtree QP is specified by the expression DnQp. It is the slice's direct-node QP limited by the smallest log2 scaling volume dimension.

DnQp := Min(dnSliceQp, OccSubtreeQpMax)  
 where  
 dnSliceQp := geom\_qp + occtree\_direct\_node\_qp\_offset << geom\_qp\_mul\_log2

##### Determination for nodes at or after OccQpSubtreeDepth

The subtree QP is specified by the expression OccSubtreeQp.

OccSubtreeQp := sliceQp + (OccSubtreeQpOffset[ss][st][sv] << geom\_qp\_mul\_log2)  
 where  
 sliceQp := geom\_qp + slice\_geom\_qp\_offset << geom\_qp\_mul\_log2  
 ss := OccSubtreeQpLoc[0]  
 st := OccSubtreeQpLoc[1]  
 sv := OccSubtreeQpLoc[2]

The expression OccSubtreeQpLoc is the subtree root node location in the tree level at depth OccQpSubtreeDepth for the node at ( Ns, Nt, Nv ).

OccSubtreeQpLoc[k] := Nloc[k] >> NodeSizeLog2[k] − OccScalingVolSizeLog2[k]

##### Maximum QP

The maximum QP for a subtree is specified by the expression OccSubtreeQpMax.

OccSubtreeQpMax := MinVec(OccScalingVolSizeLog2) × 8

It is a requirement of bitstream conformance that when occ\_subtree\_qp\_offset\_present is 1, OccSubtreeQp shall be in the range 0 .. OccSubtreeQpMax.

#### Scaling of a position component

This subclause specifies the scaling of the 𝑘-th partially quantized coordinate by the expressions:

* OccPosScaleK( 𝑘, posk ) that clips the scaled position to be within the subtree volume, and
* AngPosScaleK( 𝑘, posk ) that does not clip the scaled position.

When either geom\_scaling\_enabled or OccNodeQp is 0, the expressions OccPosScaleK( 𝑘, posk ) and AngPosScaleK( 𝑘, posk ) are both equal to posk.

The expressions upperPartQ and lowerPartQ represent the upper and lower parts of the partially quantized coordinate.

upperPartQ := posk >> OccScalingVolSizeLog2[k] − OccQpScaleLog2  
lowerPartQ := posk & Exp2(OccScalingVolSizeLog2[k] − OccQpScaleLog2) − 1

The 3-fractional-bit, fixed-point scale factor used to scale the low part is specified by the expression sF.

sF := 8 + (OccNodeQp & 7) << OccNodeQp / 8

The upper and lower parts are scaled as specified by the expressions upperPart and lowerPart.

upperPart := upperPartQ << OccScalingVolSizeLog2[k]  
lowerPart := DivExp2Up(lowerPartQ × sF, 3)

The scaled upper and lower parts are combined to produce the scaled position component:

OccPosScaleK(k, posk) := upperPart | Min(lowerPart, Exp2(OccScalingVolSizeLog2[k]) − 1)  
  
AngPosScaleK(k, posk) := upperPart + lowerPart

### Reference frame generation

#### General

Subclause 9.2.15 applies when inter\_prediction\_enabled is true. When biprediction\_enabled is 0, there are only I-frames or P-frames in the sequence. When biprediction\_enabled is 1 or 2, there may be I-frames, P-frames and B-frames in the sequence.

The following frame aliases are defined: PrevReconFrame is the previous reconstructed point cloud frame, PrevReconIPFrame is the previous reconstructed I-frame or P-frame, PrevReconBFrame is the previous reconstructed B-frame and PrevPrevReconIPFrame is the reconstructed I-frame or P-frame before the previously reconstructed I-frame or P-frame.

When the previous reconstructed point cloud frame is a B-frame (or not a B-frame), prevReconFrameIsBFrame is set to 1 (or 0).

The original reference point cloud frame PredPointCloud and (when slice\_biprediction is 1) the second reference point cloud frame SecondPredPointCloud are derived as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| biprediction\_enabled | slice\_biprediction | PredPointCloud | SecondPredPointCloud |
| 0 | - | PrevReconFrame | - |
| 1 | 0 | PrevReconIPFrame | - |
| 1 | 1 | prevReconFrameIsBFrame ? PrevReconBFrame  : PrevPrevReconIPFrame | PrevReconIPFrame |
| 2 | 1 | subclause 9.2.15.3 | subclause 9.2.15.3 |

The output of this process is the compensated reference point cloud frame CompensatedPointCloud and (when slice\_biprediction is 1) the second compensated reference point cloud frame SecondCompensatedPointCloud.

When global\_motion\_enabled is equal to 0, CompensatedPointCloud is set equal to PredPointCloud and (when slice\_biprediction is 1) SecondCompensatedPointCloud is set equal to SecondPredPointCloud. When global\_motion\_enabled is equal to 1, CompensatedPointCloud and (when slice\_biprediction is 1) SecondCompensatedPointCloud are derived by invoking subclause 9.2.15.2.

When slice\_biprediction is 1 and frame\_merge\_enabled is 1, the two compensated reference point cloud frames are merged into one by invoking subclause 9.2.15.4, and the merged compensated reference point cloud frame is used in inter prediction.

#### Motion compensation of reference frame

This subclause specifies the generation of a motion compensated reference frame using global motion parameters *ThresBot*, *ThresTop*, *AppliedGMRot*, *AppliedGMTrans* and partition type. The input of this process is the original reference point cloud, which is specified by OriPointCloud. The output of this process is the motion compensated point cloud, which is specified by *GMPointCloud*.

For the first reference point cloud frame, OriPointCloud is set equal to PredPointCloud. *ThresBot* is set equal to gm\_thres\_bot[0], *ThresTop* is set equal to gm\_thres\_top[0], *AppliedGMRot* is set equal to *GMMatrix* and *AppliedGMTrans* is set equal to gm\_trans. After the motion compensation, CompensatedPointCloud is set equal to GMPointCloud.

For the second reference point cloud frame, OriPointCloud is set equal to SecondPredPointCloud. *ThresBot* is set equal to gm\_thres\_bot[1], *ThresTop* is set equal to gm\_thres\_top[1], *AppliedGMRot* is set equal to *GMMatrix2* and *AppliedGMTrans* is set equal to gm\_trans2. After the motion compensation, SecondCompensatedPointCloud is set equal to GMPointCloud.

When MotionPartitionType is 0, GMPointCloud is derived by invoking subclause 9.2.15.2.1. When MotionPartitionType is 1, GMPointCloud is derived by invoking subclause 9.2.15.2.2.

##### Motion compensation using road/object partitioning

This subclause specifies the generation of a motion compensated reference frame using global motion parameters and road/object partitioning.

The compensated point cloud is derived based on the thresholds *ThresTop* and *ThresBot*.

for (pointIdx=0; pointIdx < OriPointCloud.getPointCount(); pointIdx++) {  
 pt = OriPointCloud[pointIdx]  
 if(pt[2] < ThresBot || pt[2] > ThresTop)  
 … /\* Apply global compensation to point pt and add compPt to GMPointCloud[pointIdx] (9.2.15.2.3) \*/\_  
 else  
 for (i=0; i < 3; i++)  
 GMPointCloud[pointIdx][i] = pt[i] >= 0 ? pt[i] : 0  
}

##### Motion compensation using cuboid partitioning

gm\_comp\_partition\_block[idx] specifies whether (when 1) or not (when 0) global motion compensation is to be applied to the idx-th partition block under cuboid partitioning.

This subclause specifies the generation of a motion compensated reference frame using global motion parameters and cuboid partitioning.

An intermediate compensated point cloud PredGMPointCloud is derived as follows:

for (pointIdx=0; pointIdx < OriPointCloud.getPointCount(); pointIdx++) {  
 pt = OriPointCloud[pointIdx]  
 … /\* Apply global compensation to point pt and add compPt to PredGMPointCloud[pointIdx] (9.2.15.2.3) \*/\_  
}

The compensated point cloud is derived based on gm\_comp\_partition\_block[blkIdx], which specifies whether motion compensation is applied for a block. When the point does not belong to a valid motion block, the value of blkIdx is -1.

compCloudPtIdx = 0  
for (n=0; n < PredGMPointCloud.getPointCount(); n++) {  
 pt = PredGMPointCloud[n]  
 … /\* Derive motion block index for point pt and set it to blkIdx (9.2.15.2.4)\*/\_  
 if(blkIdx ¬= -1 && gm\_comp\_partition\_block[blkIdx])  
 for (i=0; i < 3; i++)  
 GMPointCloud[compCloudPtIdx++][i] = pt[i]  
}  
for (n=0; n < OriPointCloud.getPointCount(); n++) {  
 pt = OriPointCloud[n]  
 … /\* Derive motion block index for point pt and set it to blkIdx (9.2.15.2.4)\*/\_  
 if(blkIdx ¬= -1 && ¬gm\_comp\_partition\_block[blkIdx])  
 for (i=0; i < 3; i++)  
 GMPointCloud[compCloudPtIdx++][i] = pt[i]  
}

##### Motion compensation for a point

This subclause specifies the generation of a motion compensated point compPt from point pt using the global motion matrix *AppliedGMRot* and global translation vector *AppliedGMTrans*.

for (i=0; i < 3; i++) {  
 compPt[i] = divExp2RoundHalfInPositiveShift (  
 rot[3\*i] \* offsetPt[0] + rot[3\*i + 1] \* offsetPt[1] + rot[3\*i + 2] \* offsetPt[2], 16, 1 << (15))+  
 tran[i] – minPosition[i]  
 compPt[i] = compPt[i] >= 0 ? compPt[i] : 0  
}  
 where,  
 rot[m] = AppliedGMRot[m]  
 tran[n] = AppliedGMTrans[n]  
 offsetPt[i] = pt[i] + minPosition[i]  
 where,  
 minPosition[i] = motion\_zero\_origin\_flag ? 0 : sps.seq\_origin\_xyz[i]

##### Deriving motion block index of point under cuboid partitioning

The motion block index blkIdx is derived for point pt under cuboid partitioning.

for(i=0; i > 3; i++)  
 idx[i] = motion\_block\_size[i] ? (pt[i]-Bbox.min[i]) / motion\_block\_size[i] : 0  
if (idx[0] < 0 || idx[0] >= NumMotionBlocksPerAxis[0] ||   
 idx[1] < 0 || idx[1] >= NumMotionBlocksPerAxis[1] ||   
 idx[2] < 0 || idx[2] >= NumMotionBlocksPerAxis[2])   
 blkIdx = -1  
else   
 blkIdx = (idx[0] \* LPUnumInAxis[1] + idx[1]) \* LPUnumInAxis[2] + idx[2]

#### Reference frames generation for B-frame in Hierarchical GOF structure

This subclause specifies the generation of reference frames for B-frame in Hierarchical GOF structure. The notional frame counters of the reconstructed I-frame or P-frame before the previously reconstructed I-frame or P-frame FrameCtrPrePreIP, the previously reconstructed I-frame FrameCtrPreIP or P-frame and the current frame FrameCtr are used to derived the reference relationship among the frames.

The size of group of frames GOFsize is determined by the frame distance between the previouly reconstructed I-frames or P-frames.

GOFsize := FrameCtrPreIP – FrameCtrPrePreIP

The hierarchical reference relationship among the frames with the notional frame counter value from FrameCtrPrePreIP to FrameCtrPreIP are derived based on a hierarchical reference GOF structure determined by GOFsize.

The notional frame counter of the B-frame to be coded *FrameCtrCur* is derived from the notional frame counter values of the reference frames *FrameCtrRef* and *FrameCtrSecondRef*.

FrameCtrCur = (FrameCtrRef + FrameCtrSecondRef) >> 1

*FrameCtrRef* and *FrameCtrSecondRef* are initialized by FrameCtrPrePreIP to FrameCtrPreI for the frames with the notional frame counter value from FrameCtrPrePreIP to FrameCtrPreIP. After one B-frame is determined, FrameCtrSecondRef is updated by FrameCtrCur to derived next B-frame, and then FrameCtrRef is updated by FrameCtrCur to derived next B-frame of the next B-frame. This process is iterated until the notional frame counter value difference between the reference frames is less than or equal to 1. An example of reference relationship among B-frames (GOFsize is 8) is shown in Table 17.

Table 17 — Example of reference relationship among point cloud frames in a Hierarchical GOF

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Notional frame counter | *a* | *a* +1 | *a* +2 | *a* +3 | *a* +4 | *a* +5 | *a* +6 | *a* +7 | *a* +8 |
| Notional frame counters offirst reference frame | × | *a* | *a* | *a*+2 | *a* | *a*+4 | *a*+4 | *a*+6 | *a* |
| Notional frame counters ofsecond reference frame | × | *a*+2 | *a*+4 | *a*+4 | *a*+8 | *a*+6 | *a*+8 | *a*+8 | *a* |
| Frame type (I/B/P) | I | B | B | B | B | B | B | B | P |

An example of decoding order of B-frames (GOFsize is 8) is shown in Table 18.

Table 18 — Example of decoding order of B-frames in a Hierarchical GOF

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Notional frame counter of  frame in decoding order | *a* | *a* +8 | *a* +4 | *a* +2 | *a* +1 | *a* +3 | *a* +6 | *a* +5 | *a* +7 |
| Frame type (I/B/P) | I | P | B | B | B | B | B | B | B |

For each B-frame, the reference frames are generated based on the derived reference relationship which is indicated by the notional frame counter values.

[Ed. (YZ): The table number may need a revisit to fix it.]

#### Frame merge mode for bi-prediction

When slice\_biprediction is 1 and frame\_merge\_enabled is 1, SecondCompensatedPointCloud is appended to CompensatedPointCloud. CompPntCnt and SecondCompPntCnt specify the numbers of points in CompensatedPointCloud and SecondCompensatedPointCloud before appending respectively.

for (pointIdx = 0; pointIdx < SecondCompPntCnt; pointIdx++) {  
 pt = SecondCompensatedPointCloud[pointIdx]   
 CompensatedPointCloud[pointIdx + CompPntCnt] = pt  
}

After merging the two compensated reference point cloud frames, only CompensatedPointCloud is used in inter prediction.

## Predictive tree

### General

Subclause 9.3 specifies the reconstruction of point positions from parsed predictive trees. It applies when geom\_tree\_type is 1.

The slice geometry can be represented by a sequence of one or more predictive trees of predictive tree nodes. Every tree node specifies a single position for one or more points. Traversal of a predictive tree is in depth-first order.

### Syntax element semantics

#### Predictive tree

slice\_ptree\_qp\_period\_log2\_offset specifies an offset to the GPS-signalled period at which predictive tree QP offsets are signalled. When slice\_ptree\_qp\_period\_log2\_offset is not present, it shall be inferred to be 0.

A QP offset is signalled once every Exp2( PtnQpInterval ) nodes.

PtnQpInterval := Exp2(ptree\_qp\_period\_log2 + slice\_ptree\_qp\_period\_log2\_offset)

ptn\_resid\_abs\_log2\_bits[ 𝑘 ] specifies the number of bins used for the fixed-length binarization of ptn\_resid\_abs\_log2[ 𝑘 ].

ptn\_radius\_min specifies the minimum angular radius coordinate for nodes where ptn\_pred\_mode is 0.

ptree\_end\_of\_slice specifies whether (when 0) or not (when 1) there is a subsequent predictive tree in the DU.

#### Predictive tree node

Nodes in the predictive trees are numbered and coded according to their position in the depth-first traversal order. The semantics specified in this subclause are for the nodeIdx-th coded node.

The syntax structure parameter depth is the tree depth of the node.

ptn\_qp\_offset\_abs[ nodeIdx ] and ptn\_qp\_offset\_sign[ nodeIdx ] together specify, when present and in accordance with PtnQpOffset[ nodeIdx ], an offset to the slice geometry QP used to scale point positions. ptn\_qp\_offset\_sign specifies whether (when 0) the offset's sign is positive or (when 1) negative. When ptn\_qp\_offset\_sign is not present, it shall be inferred to be 0.

PtnQpOffset[nodeIdx] :=  
 (1 − 2 × ptn\_qp\_offset\_sign[nodeIdx]) × ptn\_qp\_offset\_abs[nodeIdx]

The QP for a node is specified by the expression PtnQp[ nodeIdx ].

PtnQp[nodeIdx] :=  
 ¬geom\_scaling\_enabled ? 0 :  
 nodeIdx % PtnQpInterval ? PtnQp[nodeIdx − 1]  
 : sliceQp + PtnQpOffset[nodeIdx] << geom\_qp\_mul\_log2  
 where  
 sliceQp := geom\_qp + slice\_geom\_qp\_offset

It is a requirement of bitstream conformance that PtnQp[ 𝑖 ] shall be in the range 0 .. 167 for each node index 𝑖.

ptn\_dup\_point\_cnt[ nodeIdx ] plus 1 specifies the number of points represented by the node. When ptn\_dup\_point\_cnt[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_child\_cnt\_xor1[ nodeIdx ] xor 1 specifies the number of child nodes.

ptn\_inter\_flag[ nodeIdx ] equal to 1 specifies that position of the node is coded with inter prediction. When ptn\_inter\_flag[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_pred\_direction[ nodeIdx ] specifies whether (when 0) the first reference frame or (when 1) the second reference frame is used for inter prediction. When ptn\_pred\_direction[ nodeIdx ] is not present, it shall be inferred to be 0.

ptn\_inter\_pred\_mode[ nodeIdx ] specifies the method used to predict the node’s coded point position using inter prediction.

ptn\_pred\_mode[ nodeIdx ] specifies the method used to predict the node's coded point position.

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), ptn\_pred\_mode[ nodeIdx ] is not present and shall be inferred to be 1.

ptn\_pred\_idx[ nodeIdx ] specifies the predictor index in a prediction list for angular coordinates to be used to predict the node's coded point position when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1).

ptn\_phi\_mul\_abs\_prefix[ nodeIdx ], ptn\_phi\_mul\_abs\_minus2[ nodeIdx ], ptn\_phi\_mul\_abs\_minus9[ nodeIdx ] and ptn\_phi\_mul\_sign[ nodeIdx ] together specify, in accordance with PtnPhiMul[ nodeIdx ], an integer number of ptree\_ang\_azimuth\_step\_minus1 + 1 steps that form an azimuth prediction residual. ptn\_phi\_mul\_sign specifies whether (when 0) the step change is in the anticlockwise or (when 1) clockwise direction. Any of ptn\_phi\_mul\_sign, ptn\_phi\_mul\_abs\_minus2 or ptn\_phi\_mul\_abs\_minus9 that are not present shall be inferred to be 0.

PtnPhiMul[nodeIdx] := (1 − 2 × ptn\_phi\_mul\_sign[nodeIdx]) × absVal  
 where  
 absVal := ptn\_phi\_mul\_abs\_prefix[nodeIdx]  
 + ptn\_phi\_mul\_abs\_minus2[nodeIdx] + ptn\_phi\_mul\_abs\_minus9[nodeIdx]

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), the expression BoundPhiResid[ nodeIdx ] provides for the point being coded a maximum boundary for the absolute value of the azimuthal angle residual. This value is determined in accordance with the radius coordinate residual residAng[ 0 ] and the predicted radius coordinate PtnPred[0 ] for that point, the azimuthal angle step size ptree\_ang\_azimuth\_step\_minus1 and the azimuthal angle precision ptree\_ang\_azimuth\_pi\_bits\_minus11.

BoundPhiResid[nodeIdx] := DivExp2Fz(scaledR × phiStep, azimuthTwoPiLog2 + 1)  
 where  
 scaledR := (PtnPred[0] + residAng[0]) × 8  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 azimuthTwoPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 12

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided (ptree\_ang\_azimuth\_scaling\_enabled is 1), the expression PtnPhiStep[ nodeIdx ] provides a dynamically adjusted azimuthal angle step for the point being coded. When ptree\_ang\_azimuth\_scaling\_enabled is 0 it provides a constant azimuthal angle step. This value is determined in accordance with the unitary azimuthal angle step size obtained from ptree\_ang\_azimuth\_step\_minus1 using the expression BoundPhiResid[ nodeIdx ], the radius coordinate residual residAng[ 0 ] and the predicted radius coordinate PtnPred[0 ] for that point.

PtnPhiStep[nodeIdx] :=  
 (ptree\_ang\_azimuth\_scaling\_enabled &&  
 radius && BoundPhiResid[nodeIdx] == 0 && (azimuthPiLog2 - phiStepArcLog2) > 0) ?  
 (ptree\_ang\_azimuth\_step\_minus1 + 1) << (azimuthPiLog2 - phiStepArcLog2) :  
 (ptree\_ang\_azimuth\_step\_minus1 + 1)  
 where  
 radius := PtnPred[0] + residAng[0]  
 phiStepArcLog2 := IntLog2(radius × (ptree\_ang\_azimuth\_step\_minus1 + 1))  
 azimuthPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11

ptn\_radius\_resid\_abs[ nodeIdx ], and ptn\_radius\_resid\_sign[ nodeIdx ] together specify, in accordance with PtnRadiusResidual[ nodeIdx ], the first component of the first coordinate-prediction residual, when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. ptn\_radius\_resid\_sign[ nodeIdx] specifies whether (when 0) the residual's sign is positive or (when 1) negative. When ptn\_radius\_resid\_sign[ nodeIdx ]is not present, it shall be inferred to be 0.

PtnRadiusResidual[nodeIdx] := (1 − 2 × ptn\_radius\_resid\_sign[nodeIdx]) × absVal[k]  
 where  
 absVal[k] := ptn\_radius\_resid\_abs[nodeIdx]

ptn\_phi\_resid\_abs\_gt0[ nodeIdx ], ptn\_phi\_resid\_sign[ nodeIdx ], ptn\_phi\_resid\_abs\_gt1[ nodeIdx ] and ptn\_phi\_resid\_abs\_rem[ nodeIdx ] together specify, in accordance with PtnPhiResidual[ nodeIdx ], the 2nd component of the first coordinate-prediction residual, when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. ptn\_phi\_resid\_sign[ nodeIdx] specifies whether (when 0) the residual's sign is positive or (when 1) negative. Any of ptn\_phi\_resid\_abs\_gt0[ nodeIdx ], ptn\_phi\_resid\_sign[ nodeIdx ], ptn\_phi\_resid\_abs\_gt1[ nodeIdx ] or ptn\_phi\_resid\_abs\_rem[ nodeIdx ] that are not present shall be inferred to be 0.

PtnPhiResidual[nodeIdx] := (1 − 2 × ptn\_phi\_resid\_sign[nodeIdx]) × absVal[k]  
 where  
 absVal[k] := ptn\_phi\_resid\_abs\_gt0[nodeIdx]  
 + ptn\_phi\_resid\_abs\_gt1[nodeIdx]  
 + ptn\_phi\_resid\_abs\_rem[nodeIdx]

ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_log2[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_rem[ nodeIdx ][ 𝑘 ] and ptn\_resid\_sign[ nodeIdx ][ 𝑘 ] together specify, in accordance with PtnResidual[ nodeIdx ][ 𝑘 ], the 𝑘-th component of the first coordinate-prediction residual. ptn\_resid\_sign[ nodeIdx][ 𝑘 ] specifies whether (when 0) the residual's sign is positive or (when 1) negative. Any of ptn\_resid\_abs\_gt0[ nodeIdx ][ 𝑘 ], ptn\_resid\_sign[ nodeIdx ][ 𝑘 ], ptn\_resid\_abs\_log2[ nodeIdx ][ 𝑘 ] or ptn\_resid\_abs\_rem[ nodeIdx ][ 𝑘 ] that are not present shall be inferred to be 0.

PtnResidual[nodeIdx][k] := (1 − 2 × ptn\_resid\_sign[nodeIdx][k]) × absVal[k]  
 where  
 absVal[k] := ptn\_resid\_abs\_gt0[nodeIdx][k]  
 + (Exp2(ptn\_resid\_abs\_log2[nodeIdx][k]) >> 1)  
 + ptn\_resid\_abs\_rem[nodeIdx][k]

ptn\_sec\_resid\_abs[ nodeIdx ][ 𝑘 ] and ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] together specify, in accordance with PtnSecResidual[ nodeIdx ][ 𝑘 ], the 𝑘-th component of the second coordinate-prediction residual that applies after conversion from angular to Cartesian coordinates. ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] specifies whether (when 0) the residual's sign is positive or (when 1) negative. If ptn\_sec\_resid\_sign[ nodeIdx ][ 𝑘 ] is not present, it shall be inferred to be 0.

PtnSecResidual[nodeIdx][k] :=  
 (1 − 2 × ptn\_sec\_resid\_sign[nodeIdx][k]) × ptn\_sec\_resid\_abs[nodeIdx][k]

### Tree traversal for reconstruction of point positions

#### State variables

The reconstruction of point positions from predictive tree nodes is specified in terms of the following state variables:

* The array PtnStack, a stack of ancestor node indexes; PtnStack[ dpth ] is the node index of the ancestor at depth dpth in the predictive tree for the current node.

The array PtnPredList, a prediction list for angular coordinates; PtnPredList[ ptn\_pred\_idx[ nodeIdx ]][k] is used for predicting point's k-th angular coordinate when geom\_angular\_enabled is 1 and when adaptive quantization step size of the predictive geometry azimuth angle residuals is provided. The prediction list is dynamically updated after angular coordinates of points are coded (9.3.4.7).

* The variable PtnDepth, indicating the size of the stack of ancestor node indexes and equivalent to the depth of the current node in its predictive tree.
* The variable PtnIdx, the node index of the current node in the canonical traversal order.
* The variable PtnCnt, a count of nodes parsed from the bitstream.
* The variable PrevInterFrameRefIdx, providing the reference frame buffer index of the last inter decoded point.
* The variable PrevPhiResidSign, providing the sign of the last decoded non-null azimuthal angle residual.
* The variable PrevPhiMul, providing the value of the preceding decoded azimuthal angle step multiplier.
* The variable PrevRadiusResidSign, providing the sign of the last decoded non-null radius residual.

#### Initial state

The state variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul, PrevRadiusResidSign and InterFlagHist shall be initialized at the start of every GDU.

When slice\_entropy\_continuation is 1 or slice\_inter\_entropy\_continuation is 1, initialization shall be performed by the parsing state restoration process (11.6.2.2).

Otherwise (slice\_entropy\_continuation is 0 and slice\_inter\_entropy\_continuation is 0), the state variables PrevInterFrameRefIdx, PrevPhiResidSign, PrevPhiMul, PrevRadiusResidSign and InterFlagHist shall be set to 0.

#### Decoding a sequence of predictive trees

Each predictive tree is decoded according to the recursive application of 9.3.3.3, starting from its root node:

* At the start of each tree, the stack of ancestor node indexes is empty, PtnDepth = 0, and the array PtnPredList shall be initialized to 0.
* The root node of the first tree has node index PtnIdx = 0.
* The index of each subsequent tree's root node is the successor to the index of the last node in the preceding tree.

Predictive trees are decoded while PtnIdx is less than PtnCnt.

for (PtnIdx = 0; PtnIdx < PtnCnt; PtnIdx++) {  
 PtnDepth = 0  
 for (predIdx = 0; predIdx ≤ ptree\_ang\_max\_pred\_index; predIdx++)  
 for (k = 0; k < 2; k++)  
 PtnPredList[predIdx][k] = 0  
 … /\* decode predictive tree with root node index PtnIdx; see 9.3.3.3 \*/  
}

#### Recursive decoding of a subtree

This subclause specifies the reconstruction of a node with index PtnIdx and the remainder of its subtree:

* The point positions for the node are reconstructed and appended to the output point lists as specified by 9.3.4.
* When adaptive quantization step size of the predictive geometry azimuth angle is provided (when ptree\_ang\_azimuth\_scaling\_enabled is 1), the prediction list for angular coordinates shall be updated as specified by 9.3.3.7.
* When inter prediction is enabled (inter\_prediction\_enabled is 1), the inter prediction reference buffer shall be updated as specified by 9.3.3.10.
* The node's index is appended to the stack of ancestor node indexes.
* The subtrees for every child node are reconstructed in turn by the recursive application of this subclause: the index of the first child node is PtnIdx + 1; each subsequent child node index is the successor to the last node index of the preceding subtree.

PtnStack[PtnDepth++] = PtnIdx /\* push \*/

ptnChildCnt = ptn\_child\_cnt\_xor1[PtnNodeIdx] ^ 1  
for (i = 0; i < ptnChildCnt; i++) {  
 PtnIdx++  
 … /\* Recursively reconstruct the i−th child subtree (9.3.3.3) \*/  
}

After the node's subtree has been reconstructed, the node is removed from the stack of ancestor node indexes.

PtnDepth−− /\* pop \*/

#### Update of the state variable PrevPhiResidSign and PrevInterFrameRefIdx after each coded **ptn\_phi\_resid\_sign** syntax element

After each coded ptn\_phi\_resid\_sign syntax element, the state variable PrevPhiResidSign shall be set equal to the coded value of ptn\_phi\_resid\_sign.

After each coded ptn\_phi\_resid\_sign syntax element, the state variable PrevInterFrameRefIdx shall be set equal to (ptn\_inter\_pred\_mode > 1) when ptn\_inter\_flag is equal to 1.

#### Update of the state variables PrevPhiMul and PrevRadiusResidSign after each coded **ptn\_resid\_sign**[ nodeIdx ][0 ] syntax element

When ptree\_ang\_azimuth\_scaling\_enabled is 1, after each coded ptn\_resid\_sign[ nodeIdx ][0 ] syntax element (i.e. after each code radius residual sign), the state variable PrevPhiMul shall be set equal to the expression PtnPhiMul[ nodeIdx ], and the state variable PrevRadiusResidSign shall be set equal to the coded value of ptn\_resid\_sign[ nodeIdx ][0 ].

#### Dynamic update of the prediction list for angular coordinates

After the point position for a node is decoded, the prediction list for angular coordinates shall be updated according to the value of the decoded radius residual residAng[ 0 ] and the threshold ptree\_ang\_pred\_list\_radius\_resid\_threshold:

if (ptree\_ang\_azimuth\_scaling\_enabled) {  
 for (predIdx = predIdxRemoved; predIdx > 0; predIdx--)  
 for(k = 0; k < 2; k ++)  
 PtnPredList[predIdx][k] = PtnPredList[predIdx - 1][k]  
 for(k = 0; k < 2; k ++)  
 PtnPredList[0][k] = PtnAng[k]  
}  
 where  
 predIdxRemoved := flagNewObject ? ptree\_ang\_max\_pred\_index : ptnPredIdx;  
 where  
 ptnPredIdx := ptn\_pred\_idx[PtnIdx]  
 flagNewObject := Abs(residAng[0]) > ptree\_ang\_pred\_list\_radius\_resid\_threshold

#### Update of the state variable InterFlagHist after each coded **ptn\_inter\_flag**[ nodeIdx ] syntax element

When slice\_inter\_prediction is 1, after each coded ptn\_inter\_flag[ nodeIdx ] syntax element, the state variable InterFlagHist shall be set equal to the expression InterFlagHist << 1 & ptn\_inter\_flag[ nodeIdx ].

#### Generate inter prediction list for angular coordinates

After the point position for a node is decoded, the inter prediction list for angular coordinates shall be updated according to the value of the decoded azimuth and beam ID. The subclause 9.3.3.9.1 is invoked with PtnRefFrame and (when slice\_inter\_frame\_ref\_gmc[0] is 1) PtnAltRefFrame as input and inter predictor list PtnInterPredList as output. When slice\_biprediction is 1, subclause 9.3.3.9.1 is invoked with PtnSecondRefFrame and (when slice\_inter\_frame\_ref\_gmc[1] is 1) PtnSecondAltRefFrame as input and inter predictor list PtnSecondInterPredList as output.

##### Update of inter prediction list

The input to this subclause are reference frames RefFrame and in some case AltRefFrame and the output is the inter predictor list InterPredList.

cnt = 0  
for (qAzim = stAzim; qAzim <= MaxQAzim && cnt < 2; qAzim++)  
 if (PtnRefFrame[beamId][qAzim][0] != -1) {  
 for (k = 0; k < 3; k++)  
 InterPredList[cnt][k] = RefFrame[beamId][qAzim][k]  
 cnt++  
 }  
while(cnt < 2)  
 InterPredList[cnt++] = na  
if (global\_motion\_enabled)  
 for (qAzim = stAzim; qAzim <= MaxQAzim && cnt < 4; qAzim++)  
 if (AltRefFrame[beamId][qAzim][0] != -1) {  
 InterPredList[cnt][0] = AltRefFrame[beamId][qAzim][0]  
 if(slice\_inter\_frame\_ref\_gmc){  
 deltaPhi = AltRefFrame[beamId][qAzim][1] – phiP  
 InterPredList[cnt][1] = phiP  
 if(deltaPhi >= (phiStep >> 1) ||  
 deltaPhi <= -(phiStep >> 1)){  
 phiMul = Div((deltaPhi) + (phiStep >> 1), phiStep, 0)  
 InterPredList[cnt][1] += phiMul \* phiStep  
 }  
 }  
 else  
 InterPredList[cnt][1] = AltRefFrame[beamId][qAzim][1]  
 InterPredList[cnt][2] = AltRefFrame[beamId][qAzim][2]  
 cnt++  
 }  
 while(cnt < 4)  
 InterPredList[cnt++] = na  
 where  
 beamId := PtnAng[2]  
 stAzim := DivExp2Fz(PtnAng[1], inter\_azim\_scale\_log2)  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 phiP := PointAng[ptIdx][1]  
 where  
 ptIdx := PtnStack[PtnDepth – 1]

#### Generation of derived reference frame

Subclause 9.3.3.10 applies to generate the derived reference frame(s).

The original reference frame is specified by PtnOriRefFrame.

When biprediction\_enabled is 0, PtnOriRefFrame is set equal to PtnCurrFramePos.

When biprediction\_enabled is 1 or 2, the reconstructed frame PtnCurrFramePos is indicated by the notional frame counter to be used as the reference frame for subsequent point cloud frames. PtnOriRefFrame is determined as follows:

* when slice\_biprediction is 0, PtnOriRefFrame is set equal to the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 1 and slice\_biprediction is 1, if the previously reconstructed point cloud frame is a B-frame, PtnOriRefFrame is set equal to the previously reconstructed B-frame; otherwise, PtnOriRefFrame is set equal to the I-frame or P-frame before the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 2 and slice\_biprediction is 1, PtnOriRefFrame is set equal to the first reference frame derived by invoking subclause 9.2.15.3.

The first derived reference frame PtnRefFrame and (when global\_motion\_enabled is 1) the second derived reference frame PtnAltRefFrame are derived as follows:

* when global\_motion\_enabled is 0, PtnRefFrame is set equal to PtnOriRefFram;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc is 0, PtnAltRefFrame is set equal to PtnRefFrame and then PtnRefFrame is set equal to PtnOriRefFrame;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc is 1, the global motion compensation is applied to points in PtnOriRefFrame as follows:

if (global\_motion\_enabled)  
 for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnOriRefFrame[beamId][qAzim][0] != -1) {   
 … /\* Apply global compensation to point pt and add compensated point to PtnGlobFrame (9.3.3.13) \*/\_  
 }  
 where  
 pt[k] = PtnOriRefFrame[beamId][qAzim][k]

When resampling\_enabled is 1, the radius values of points in PtnOriRefFrame are updated using global motion compensated points as follows:

for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnOriRefFrame[beamId][qAzim][0] != -1) {  
 … /\* Update radius of point pt in PtnOriRefFrame (9.3.3.14) \*/\_  
 }  
 where  
 pt[k] = PtnOriRefFrame[beamId][qAzim][k]

PtnRefFrame is set equal to PtnOriRefFrame and PtnAltRefFrame is set equal to PtnGlobFrame.

#### Generation of derived reference frame for the second reference frame

When biprediction\_enabled is 1 or 2, and slice\_biprediction is 1, the second original reference frame PtnSecondOriRefFrame is determined as follows.

* when biprediction\_enabled is 1, PtnSecondOriRefFrame is set equal to the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 2, PtnSecondOriRefFrame is set equal to the second reference frame derived by invoking subclause 9.2.15.3.

The corresponding first derived reference frame PtnSecondRefFrame and (when global\_motion\_enabled is 1) the second derived reference frame PtnSecondAltRefFrame are derived as follows:

* when global\_motion\_enabled is 0, PtnSecondRefFrame is set equal to PtnSecondOriRefFram;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc2 is 0, PtnSecondAltRefFrame is set equal to PtnSecondRefFrame and then PtnSecondRefFrame is set equal to PtnSecondOriRefFrame;
* when global\_motion\_enabled is 1 and slice\_inter\_frame\_ref\_gmc2 is 1, the global motion compensation is applied to points in PtnSecondOriRefFrame as follows:

if (global\_motion\_enabled)  
 for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnSecondOriRefFrame[beamId][qAzim][0] != -1) {   
 … /\* Apply global compensation to point pt and add compensated point to PtnSecondGlobFrame (9.3.3.13) \*/\_  
 }  
 where  
 pt[k] = PtnSecondOriRefFrame[beamId][qAzim][k]

When resampling\_enabled is 1, the radius values of points in PtnSecondOriRefFrame are updated using global motion compensated points as follows:

for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 if(PtnSecondOriRefFrame[beamId][qAzim][0] != -1) {  
 … /\* Update radius of point pt in PtnSecondOriRefFrame (9.3.3.14) \*/\_  
 }  
 where  
 pt[k] = PtnSecondOriRefFrame[beamId][qAzim][k]

PtnSecondRefFrame is set equal to PtnSecondOriRefFrame and PtnSecondAltRefFrame is set equal to PtnGlobFrame.

#### Global motion compensation

[Ed. (YZ): The value derivation of AngularOriginRef to be added.]

The spherical to cartesian coordinate conversion of point is applied as follows, the slice-relative angular origin in STV coordinates of the reference frame is specified by the expression AngularOriginRef[ 𝑘 ]:

ptCart[k] := AngularOriginRef[k] + (  
 k == 0 ? DivExp2Fz(ρ × IntCos(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24):  
 k == 1 ? DivExp2Fz(ρ × IntSin(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24):  
 k == 2 ? DivExp2Fz(DivExp2Fz(BeamElev[i] × ρ, 15) − BeamOffsetV[i], 3): na)  
 where  
 ρ := pt[0] << ptree\_ang\_radius\_scale\_log2  
 φ := pt[1]  
 i := pt[2]

The global motion compensation of a point is applied based on the threshold values *ThresBot* and *ThresTop*, global motion *AppliedGMMatrix* and *AppliedGMTrans* as follows:

if(ptCart[2] < ThresBot || ptCart[2] > ThresTop)  
 for(k = 0; k < 3; k++)  
 ptCartRef[k] = DivExp2Fz(ptCartComp[k], 16) + AppliedGMTrans[k]  
 where  
 ptCartComp[k] := AppliedGMMatrix[k][0] \* ptCart[0] + AppliedGMMatrix[k][1] \* ptCart[1] + AppliedGMMatrix[k][2] \* ptCart[2]

For the first reference frame, *ThresBot* is set equal to gm\_thres\_bot, *ThresTop* is set equal to gm\_thres\_top, *AppliedGMMatrix* is set equal to GMMatrix and *AppliedGMTrans* is set equal to gm\_trans. When slice\_biprediction is 1, *ThresBot* is set equal to gm\_thres\_bot2, *ThresTop* is set equal to gm\_thres\_top2, *AppliedGMMatrix* is set equal to GMMatrix2 and *AppliedGMTrans* is set equal to gm\_trans2.

The compensated point is then converted back into angular domain as follows:

ptSph[0] = IntSqrt(ptC[0] × ptC[0] + ptC[1] × ptC[1] << 16) >> 8  
ptSph[1] = IntAtan2(ptC[1] << 8, ptC[0] << 8)  
ptSph[1] = (((ptSph[1] + 3294199) \* 5340354 + off) >> sh) - (1 << azimLog2)  
ptSph[2] = BeamIdxEst[ptC[0]][ptC[1]][ptC[2]]  
 where  
 ptC[k] = ptCartRef[k]  
 azimLog2 = ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11  
 sh = 44 – azimLog2  
 off = 1 << sh – 1

The compensated point is then updated to buffer PtnGlobFrame as follows:

if (PtnGlobFrame[beamId][qAzim][0] == -1 || PtnGlobFrame[beamId][qAzim][0] > ptSph[0])  
 for(k = 0; k < 3; k ++)  
 PtnGlobFrame[beamId][qAzim][k] = ptSph[k]

#### Update radius after compensation

The final derived reference frame is generated as follows:

if (PtnGlobFrame[beamId][qAzim][0] != -1) {  
 for(k = 0; k < 2; k++) {  
 ptA[k] = PtnGlobFrame[beamId][qAzim][k]  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 }  
 if (ptA[1] < pt[1]) {  
 for(qA = qAzim + 1; qA <= MaxQAzim; qA++)  
 if (PtnGlobFrame[beamId][qA][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 } else if (ptA[1] > pt[1])  
 for(qB = qAzim - 1; qB >= MinQAzim; qB--)  
 if(PtnGlobFrame[beamId][qB][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 }  
} else {  
 for(qA = qAzim + 1; qA <= MaxQAzim; qA++)  
 if (PtnGlobFrame[beamId][qA][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptA[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
 for(qB = qAzim - 1; qB >= MinQAzim; qB--)  
 if(PtnGlobFrame[beamId][qB][0] != -1) {  
 for(k = 0; k < 2; k++)  
 ptB[k] = PtnGlobFrame[beamId][qAzim][k]  
 break  
 }  
}

When both ptA and ptB are not initialized (there is no point PtnGlobFrame[beamId] corresponding to beamId) the radius is not updated. When only one of ptA and ptB is initialized, both ptA and ptB are set equal to the initialized point as follows:

if (qA == MaxQAzim + 1 && qB != MinQAzim – 1)  
 ptA = ptB  
else if (qA != MaxQAzim + 1 && qB == MinQAzim – 1)  
 ptB = ptA

Once points ptA and ptB are obtained, the radius is updated as follows:

if (!del0 || !del1)  
 pt[0] = ptA[0]  
else  
 pt[0] = ptA[0] + (1 – 2\*sgn) \* (abs(nr) + (abs(dr) >> 1)) / abs(dr)  
 where  
 nr = del0 \* (pt[1] – ptA[1])  
 dr = del1  
 sgn = !((nr > 0 && dr > 0) || (nr < 0 && dr < 0))  
 where  
 del0 = ptA[0] – ptB[0]  
 del1 = ptA[1] – ptB[1]

### Reconstruction of point coordinates

#### General

Subclause 9.3.4 specifies the reconstruction of the point positions for a node with index PtnIdx at depth PtnDepth of a predictive tree.

#### Reconstructed STV coordinates

The node's reconstructed STV coordinates are specified by the expression PtnPos[ 𝑘 ]. They are the sum of a prediction (predStv) and a scaled residual (residStv), with negative coordinates clipped to 0. The predicted position is:

* when angular geometry coding is disabled: derived from the reconstructed STV coordinates of ancestor nodes (9.3.4.6);
* when angular geometry coding is enabled: a conversion from the node's reconstructed angular coordinates (9.3.4.5).

PtnPos[k] := Max(0, predStv[k] + residStv[k])  
 where  
 predStv[k] := geom\_angular\_enabled ? PtnAngStv[k] : PtnPred[k]

The scaled STV coordinate residuals, specified by residStv[ 𝑘 ], are:

* 0, when second coordinate residual coding is disabled; or otherwise
* coded by the node's first residual when angular geometry coding is disabled, and by its second residual when enabled; and
* scaled by the 3-fractional-bit, fixed-point, geometry scale factor specified by the expression sf for the node's QP; scaling shall round to the nearest integer with half-values rounded up.

residStv[k] := DivExp2Up(resid[k] × sf, 3)  
 where  
 sf := 8 + (PtnQp[PtnIdx] & 7) << PtnQp[PtnIdx] / 8  
 resid[k] := geom\_angular\_enabled ? ptree\_sec\_resid\_disabled ? 0  
 : PtnSecResidual[PtnIdx][k]  
 : PtnResidual[PtnIdx][k]

#### Reconstructed RPI node coordinates

When angular geometry coding is enabled, the node's reconstructed RPI coordinates are specified by the expression PtnAng[ 𝑘 ]. They are the sum of an angular coordinate prediction (9.3.4.6) and a residual (residAng).

It is a requirement of bitstream conformance that PtnAng[ 2 ] shall be in the range 0 .. num\_beams\_minus1.

PtnAng[k] := PtnPred[k] + residAng[k]

The residual RPI coordinates specified by residAng[ 𝑘 ] are the sum of a 𝜑-component offset, phiOffset and values PtnResiAng of primary residual.

residAng[k] := (k == 1 ? phiOffset : 0) + PtnResiAng[k]  
 where  
 phiOffset := phiOffset0 + phiOffset1  
 PtnResiAng[k] :=  
 k == 0 && ptree\_ang\_azimuth\_scaling\_enabled ? PtnRadiusResidual[PtnIdx] :  
 k == 1 && ptree\_ang\_azimuth\_scaling\_enabled ? scaledPhiAng :  
 PtnResidual[PtnIdx][k]  
 where  
 phiOffset0 := PtnPhiMul[PtnIdx] × PtnPhiStep[ nodeIdx ]  
 phiOffset1 := ptree\_ang\_azimuth\_scaling\_enabled ? (  
 PtnResiAng[k] + phiOffset0 ≥ azimuthPi ? –2 × azimuthPi :  
 PtnResiAng[k] + phiOffset0 < -azimuthPi ? 2 × azimuthPi : 0) : 0  
 where  
 azimuthPi := 1 << (ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11)

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided, scaledPhiAng is the scaled value of the quantized primary residual.

scaledPhiAng := DivExp2Tz(PtnPhiResidual[PtnIdx] × invR, invRFracBits - azimuthTwoPiLog2)  
 where  
 azimuthTwoPiLog2 := ptree\_ang\_azimuth\_pi\_bits\_minus11 + 12  
 (invR, invRFracBits) := IntRecip(scaledR)  
 where  
 scaledR := (PtnPred[0] + residAng[0]) > 0 ? (PtnPred[0] + residAng[0]) × 8 : 1

#### Points represented by a node

The reconstructed STV and, if applicable, RPI point coordinates are appended to the output point lists PointPos and PointAng for each point represented by the node.

for (i = 0; i ≤ ptn\_dup\_point\_cnt[PtnIdx]; i++, PointCnt++)  
 for (k = 0; k < 3; k++) {  
 PointPos[PointCnt][k] = PtnPos[k]  
 if (geom\_angular\_enabled)  
 PointAng[PointCnt][k] = PtnAng[k]  
 }

The decoded point position is output for each point represented by the current node.

for (i = 0; i ≤ ptn\_dup\_point\_cnt[curNodeIdx]; i++, PointCnt++)  
 for (k = 0; k < 3; k++) {  
 PointPos[PointCnt][k] = nodePos[k]  
 if (geom\_angular\_enabled)  
 PointAng[PointCnt][k] = nodeAng[k]  
 }

#### Predicted STV coordinates for angular coded geometry

When angular geometry coding is enabled, the node's RPI coordinates are converted to Cartesian STV coordinates, as specified by PtnAngStv[ 𝑘 ], for prediction of the coded position.

PtnAngStv[k] := AngularOrigin[k] + (  
 k == 0 ? DivExp2Fz(ρ × IntCos(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24) :  
 k == 1 ? DivExp2Fz(ρ × IntSin(φ, ptree\_ang\_azimuth\_pi\_bits\_minus11 + 11), 24) :  
 k == 2 ? DivExp2Fz(DivExp2Fz(BeamElev[i] × ρ, 15) − BeamOffsetV[i], 3) : na)  
 where  
 ρ := PtnAng[0] << ptree\_ang\_radius\_scale\_log2  
 φ := PtnAng[1]  
 i := PtnAng[2]

#### Prediction from ancestor nodes

Node coordinates can be predicted from up to three ancestor nodes. Depending upon whether angular geometry is enabled or disabled, the prediction is for either RPI or STV coordinates. The coordinates of the parent, grandparent and great-grandparent nodes are specified by ptnP[ 𝑘 ], ptnG[ 𝑘 ] and ptnU[ 𝑘 ], respectively.

ptnP[k] := PtnRef[1][k]  
ptnG[k] := PtnRef[2][k]  
ptnU[k] := PtnRef[3][k]

PtnRef[ancestor][k] := geom\_angular\_enabled ? PointAng[ptIdx][k] : PointPos[ptIdx][k]  
 where  
 ptIdx := PtnStack[PtnDepth − ancestor]

When the adaptive quantization step size of the predictive geometry azimuth angle residuals is not provided, the predicted coordinates, specified by PtnPred[ 𝑘 ], are:

* when the prediction mode is 0 and if angular geometry is:
  + disabled, the origin ( 0, 0, 0 );
  + enabled, a coordinate specified by predAngMode0[ 𝑘 ] whose 𝜌-component is ptn\_radius\_min, and whose 𝜑- and 𝑖-components are the same as the parent node, or zero if the predicted node is the root node of a predictive tree.
* when the prediction mode is 1, the coordinates of the parent node;
* when the prediction mode is 2, the coordinates of the parent node, translated by the vector from the second to first ancestor;
* when the prediction mode is 3, the coordinates of the parent node, translated by the vector from the third to the second ancestor.

When adaptive quantization step size of the predictive geometry azimuth angle residuals is provided, the prediction mode is always 1 and the predicted coordinates, specified by PtnPred[ 𝑘 ], are:

* the coordinates specified by predAngMode0[ 𝑘 ] if the predicted node is the root node of a predictive tree;
* derived according to the prediction list for angular coordinates, for the two first predicted coordinates PtnPred[0 ] and PtnPred[1]; and the third coordinate of the parent node for the third predicted coordinate PtnPred[2 ].

PtnPred[k] :=  
 interFlag ? (interPredDir ? PtnSecondInterPredList[interModeIdx] :   
 PtnInterPredList[interModeIdx]) :  
 predMode == 0 && geom\_angular\_enabled ? predAngMode0[k] :  
 predMode == 0 ? 0 :  
 predMode == 1 && PtnIdx == 0 ? predAngMode0[k] :  
 predMode == 1 && usePredList ? PtnPredList[ptnPredIdx][k] + PtnPredAdjust :  
 predMode == 1 ? ptnP[k] :  
 predMode == 2 ? ptnP[k] + ptnP[k] − ptnG[k] :  
 predMode == 3 ? ptnP[k] + ptnG[k] − ptnU[k] : na  
 where  
 interFlag := ptn\_inter\_flag[PtnIdx]  
 interPredDir := ptn\_pred\_direction[PtnIdx]  
 interModeIdx:= ptn\_inter\_pred\_mode[PtnIdx]  
 predMode := ptn\_pred\_mode[PtnIdx]  
 ptnPredIdx := ptn\_pred\_idx[nodeIdx]  
 usePredList := ptree\_ang\_azimuth\_scaling\_enabled && k < 2 && ptnPredIdx > 0  
 PtnPredAdjust :=  
 k == 1 && abs(deltaPhi) >= phiStep ? phiMul \* deltaPhi : 0  
 where  
 deltaPhi := ptnP[1] - PtnPredList[ptnPredIdx][1]  
 phiStep := ptree\_ang\_azimuth\_step\_minus1 + 1  
 phiMul := Div(deltaPhi, phiStep, 0)

predAngMode0[k] :=  
 k == 0 ? ptn\_radius\_min :  
 PtnDepth > 0 ? ptnP[k] : 0

[Ed. (JT): (PtnIdx == 0) could be appended with && ptree\_ang\_azimuth\_scaling\_enabled for clarity, but not required.]

## TriSoup

### General

Subclause 9.4 specifies the parsing and the reconstruction of point positions from the TriSoup process. It applies when geom\_tree\_type is 0 and when trisoup\_enable\_flag is 1.

At first, the occupancy tree process described in subclause 9.2 is applied to obtain an occupancy tree at depth occtreeMaxDepthMinus1. The log2 node dimensions at depth occtreeMaxDepthMinus1 is equal to the value trisoup\_node\_size\_log2\_minus2 obtained from the geometry data unit header.

Then, the set of all occupied leaf nodes, of the occupancy tree, at depth occtreeMaxDepthMinus1 constitutes the set of TriSoup nodes. The slice geometry is represented by TriSoup edge vertices (9.4.3.1) located on the edges of the cuboid volumes associated with the occupied leaf nodes and, optionally, by residual values of centroid vertices determined from TriSoup edge vertices, and, optionally, by face vertices created from two centroid vertices in the cuboid volumes and in the adjacent nodes.

The point positions are reconstructed by generating (9.4.3.2) a set of TriSoup triangles from the TriSoup edge vertices, TriSoup face vertices and centroid vertices, and by voxelization (9.4.3.3) of the TriSoup triangles into points by a ray tracing process.

#### Definition and ordering of TriSoup nodes

TriSoup nodes are the occupied leaf nodes, at maximum coded tree level occtreeMaxDepthMinus1, obtained from the decoding process of the occupancy tree as described in clause 9.2. The TriSoup log2 node dimension is OccLvlNodeSizeLog2[ occtreeMaxDepthMinus1+1 ][ 𝑘 ] along the 𝑘-th component. The occupancy tree is coded such that the three TriSoup log2 node dimensions are all equal to trisoup\_node\_size\_log2\_minus2 + 2. Consequently, the volumes associated with TriSoup nodes are cubes having edge length equal to

TriSoupCubeSize := Exp2( trisoup\_node\_size\_log2\_minus2 + 2 )

The edge length TriSoupNodeSize[ nodeIdx ][ k ] of the nodeIdx-th TriSoup node is set as follows:

TriSoupNodeSize[nodeIdx][k] := TriSoupCubeSize

The TriSoup node ordering is inherited from the occupancy tree node ordering, as described in clause 9.2.2.2, applied to the leaf nodes of the occupancy tree. This order is obtained from a breadth-first traversal of the occupancy tree and, at each level, child nodes of a node are ordered following a Morton order. This results in a traversal in the ascending Morton order of node location.

The number of TriSoup nodes is numberTriSoupNodes. In the following of clause 9.4 nodeIdx is a node index between 0 and numberTriSoupNodes – 1 that refers to the nodeIdx-th TriSoup node.

The location TriSoupNodeLoc[ nodeIdx ][ k ] of the nodeIdx-th TriSoup node lower corner in the slice coordinate system is obtained as described in sub-clause 9.2.2.1 by

TriSoupNodeLoc[nodeIdx][k] = nodeLoc[k] × TriSoupCubeSize

where nodeLoc[ 𝑘 ] is obtained recursively on the occupancy tree levels as described in clause 9.2.2.3.

When one or both of trisoup\_non\_cubic\_node\_start\_edge\_presence\_flag and trisoup\_non\_cubic\_node\_end\_edge\_presence\_flag are equal to 1, TriSoupNodeLoc[ *nodeIdx* ][ 𝑘 ] and TriSoupNodeSize[ *nodeIdx* ][ 𝑘 ] are modified as follows:

if(trisoup\_slice\_bb\_pos\_bits > 0 && TriSoupNodeLoc[nodeIdx][k] < TriSoupSliceBbPos[k]){  
 TriSoupNodeSize[nodeIdx][k] :=   
 TriSoupNodeLoc[nodeIdx][k] + TriSoupNodeSize[nodeIdx][k] − TriSoupSliceBbPos[k]  
 TriSoupNodeLoc[nodeIdx][k] := TriSoupSliceBbPos[k]  
}

if(trisoup\_slice\_bb\_width\_bits > 0 &&   
 TriSoupSliceBbBoundary[k]) < TriSoupNodeLoc[nodeIdx][k] + TriSoupNodeSize[nodeIdx][k])  
 TriSoupNodeSize[nodeIdx][k] :=   
 TriSoupSliceBbBoundary[k] − TriSoupNodeLoc[nodeIdx][k] + 1

#### Definition and ordering of TriSoup edges

The cuboid volume associated with the nodeIdx-th TriSoup node is depicted on Figure 1. Twelve edges are obtained from the boundaries of the volume. Each edge is defined by its start position edgePosStart and its end position edgePosEnd.

图片包含 应用程序

描述已自动生成

Figure 18 — Volume and edges associated with a TriSoup node

The start and end positions of the twelve edges are obtained from Table 25 by

for (edgeTag = 0; edgeTag < 12; edgeTag ++) {  
 for (k = 0; k < 3; k++) {  
 edgePosStart[nodeIdx][edgeTag][k] = TriSoupNodeLoc[nodeIdx][k]  
 + edgePosStartShift[edgeTag][k] × TriSoupNodeSize[nodeIdx][k]

edgePosEnd[nodeIdx][edgeTag][k] = edgePosStart[nodeIdx][edgeTag][k]  
 + edgePosEndShift[edgeTag][k] × TriSoupNodeSize[nodeIdx][k]

}  
}

By generating the twelve edges of the numberTriSoupNodes TriSoup nodes, a collection of 12×numberTriSoupNodes of edges is obtained. An edge is uniquely defined by its start and end positions. Consequently, an edge may present multiple times in the collection of 12×numberTriSoupNodes edges because adjacent TriSoup nodes can share a common edge. By removing duplicated edges, a set of unique numberTriSoupEdges TriSoup edges is obtained.

Table 25 — Twelve edges' positions relative to the TriSoup node location

| edge number (edgeTag) | edgePosStartShift[edgeTag][k] | | | edgePosEndShift[edgeTag][k] | | |
| --- | --- | --- | --- | --- | --- | --- |
|  | k =0 | k =1 | k =2 | k =0 | k =1 | k =2 |
| **0** | 0 | 0 | 0 | 1 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 1 | 0 |
| **2** | 0 | 1 | 0 | 1 | 0 | 0 |
| **3** | 1 | 0 | 0 | 0 | 1 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 1 |
| **5** | 0 | 1 | 0 | 0 | 0 | 1 |
| **6** | 0 | 1 | 1 | 0 | 0 | 1 |
| **7** | 1 | 0 | 0 | 0 | 0 | 1 |
| **8** | 0 | 0 | 1 | 1 | 0 | 0 |
| **9** | 0 | 0 | 1 | 0 | 1 | 0 |
| **10** | 0 | 1 | 1 | 1 | 0 | 0 |
| **11** | 1 | 0 | 1 | 0 | 1 | 0 |

The set of unique TriSoup edges is ordered lexicographically, first in the three-component start position and second in the three-component end position of the edges. In the following of clause 9.4 edgeIdx is an edge index between 0 and numberTriSoupEdges – 1 that refers to the edgeIdx -th TriSoup edge in the set of unique edges. uniqueEdgePosStart[ *edgeIdx* ] and uniqueEdgePosEnd[ *edgeIdx* ] specify start and end position of *edgeIdx*-th unique edge.

### Syntax element semantics

#### Semantics of TriSoup syntax elements present in the geometry data unit header

trisoup\_node\_size\_log2\_minus2 plus 2 specifies the size of TriSoup node.

trisoup\_sampling\_value\_minus1 plus 1 specifies the sampling distance of each rayOrigin used by ray tracing process to generate decoded points.

trisoup\_num\_unique\_segments\_bits\_minus1 plus 1 specifies the length in bits of trisoup\_num\_unique\_segments\_minus1 syntax element.

trisoup\_num\_unique\_segments\_minus1 plus 1 specifies the number of TriSoup edges.

trisoup\_vertex\_number\_bits specifies the number of bits for a decoded TriSoup vertex. The values VertexPrecisionLog2 and bitDropped specify the precision and the quantization scale of decoded TriSoup vertex position.

VertexPrecisionLog2 := trisoup\_vertex\_number\_bits > 0 ? trisoup\_vertex\_number\_bits : trisoup\_node\_size\_log2\_minus2 + 2

bitDropped := max(0,  trisoup\_node\_size\_log2\_minus2 + 2  - VertexPrecisionLog2)

trisoup\_centroid\_vertex\_residual\_flag specifies whether (when 1) or not (when 0) residuals of centroid positions are present in TriSoup syntax structure.

trisoup\_face\_vertex\_flag is equal to 1 specifies that the eligibility of face vertex is judged for each TriSoup node as specified in 9.4.2.4 and face vertex may be present on the face of TriSoup nodes. When trisoup\_face\_vertex\_flag is equal to 0 specifies that the eligibility of face vertex is not judged for any TriSoup node and face vertex is not present on the face of any TriSoup nodes. When trisoup\_face\_vertex\_flag is not present, it shall be inferred to be 0.

trisoup\_halo\_flag specifies whether (when 1) or not (when 0) halo is applied in the ray tracing process.

trisoup\_adaptive\_halo\_flag specifies whether (when 1) or not (when 0) halo is applied in the ray tracing process. When trisoup\_adaptive\_halo\_flag is not present, it shall be inferred to be 0.

trisoup\_vertex\_merge specifies whether (when 1) or not (when 0) the vertex merge process that replace multiple vertices near the corner of TriSoup nodes by a single vertex is applied.

trisoup\_slice\_bb\_pos\_bits specifies the length in bits of each trisoup\_slice\_bb\_pos\_xyz syntax element. When trisoup\_slice\_bb\_pos\_bits is not present, it shall be inferred to be 0.

trisoup\_slice\_bb\_pos\_xyz[ k] and trisoup\_slice\_bb\_pos\_log2\_scaletogether specify the lower corner k-th XYZ coordinates of the slice bounding box in the slice coordinate system. The lower corner of the slice bounding box in STV coordinates is specified by the expression TriSoupSliceBbPos[ k]. When trisoup\_slice\_bb\_pos\_bits is 0, trisoup\_slice\_bb\_pos\_xyz[k ] and trisoup\_slice\_bb\_pos\_log2\_scale shall be inferred to be 0.

TriSoupSliceBbPos[k] :=   
triSoup\_slice\_bb\_pos\_xyz[StvToXyz[k]] << trisoup\_slice\_bb\_pos\_log2\_scale

trisoup\_slice\_bb\_width\_bitsspecifies the length in bits of each trisoup\_slice\_bb\_width\_xyz syntax element. When trisoup\_slice\_bb\_width\_bits is not present, it shall be inferred to be 0.

trisoup\_slice\_bb\_width\_xyz[ k ] and trisoup\_slice\_bb\_width\_log2\_scale together specify the k-th XYZ component of the slice bounding box dimensions in the slice coordinate system. When trisoup\_slice\_bb\_width\_bits is not 0, the upper boundary of the slice bounding box in STV coordinates is specified by the expression TriSoupSliceBbBoundary[k ].

TriSoupSliceBbBoundary[k] := TriSoupSliceBbPos[k] +   
trisoup\_slice\_bb\_width\_xyz[StvToXyz[k]] << trisoup\_slice\_bb\_width\_log2\_scale

#### Semantics of TriSoup syntax elements associated with TriSoup edges

vertex\_flag[edgeIdx] specifies whether (when 1) or not (when 0) a TriSoup vertex exists on edgeIdx-th TriSoup edge.

has\_vertex[nodeIdx][edgeTag] specifies whether (when 1) or not (when 0) a TriSoup vertex exists on edgeTag-th TriSoup edge of nodeIdx-th TriSoup node. has\_vertex[nodeIdx][edgeTag] is set by corresponding vertex\_flag[edgeIdx] as defined in 9.4.1.1.

vertex\_position[edgeIdx] specifies a quantized one-dimensional vertex position according to the direction of each TriSoup edge with a vertex. When vertex\_position[edgeIdx] is not present (when vertex\_flag[edgeIdx] is equal to 0), vertex\_position[edgeIdx] shall be inferred to be -1.

edgeVertex[nodeIdx][edgeTag] specifies a quantized one-dimensional vertex position according to the direction of edgeTag-th TriSoup edge of nodeIdx-th TriSoup node. edgeVertex[nodeIdx][edgeTag] is set by corresponding vertex\_position[edgeIdx] as defined in9.4.1.1.

When trisoup\_vertex\_merge is equal to 1, the value vertexNumOfCorner[ns][nt][nv] specifies the number of vertices around a node corner at ( ns, nt, nv ) and the threshold value thVertexMerge derived as follows.

thVertexMerge := min(2048, TriSoupCubeSize << 6)  
for (edgeIdx = 0; edgeIdx < numberTriSoupEdges; edgeIdx++){  
 if (vertex\_flag[edgeIdx]) {  
 ns = uniqueEdgePosStart[edgeIdx][0]  
 nt = uniqueEdgePosStart[edgeIdx][1]  
 nv = uniqueEdgePosStart[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv] = 0  
   
 ns = uniqueEdgePosEnd[edgeIdx][0]  
 nt = uniqueEdgePosEnd[edgeIdx][1]  
 nv = uniqueEdgePosEnd[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv] = 0  
 }  
}  
for (edgeIdx = 0; edgeIdx < numberTriSoupEdges; edgeIdx++){  
 if (vertex\_flag[edgeIdx]) {  
 relativePos = (vertex\_position[edgeIdx] << (8 + bitDropped)) + (128 << bitDropped)  
 if (relativePos < thVertexMerge) {  
 ns = uniqueEdgePosStart[edgeIdx][0]  
 nt = uniqueEdgePosStart[edgeIdx][1]  
 nv = uniqueEdgePosStart[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv]++  
 }  
 if ((TriSoupCubeSize << 8) - relativePos < thVertexMerge) {  
 ns = uniqueEdgePosEnd[edgeIdx][0]  
 nt = uniqueEdgePosEnd[edgeIdx][1]  
 nv = uniqueEdgePosEnd[edgeIdx][2]  
 vertexNumOfCorner[ns][nt][nv]++  
 }  
 }  
}

#### Semantics of TriSoup syntax elements associated with TriSoup nodes

centroid\_residual\_is\_zero[nodeIdx] specifies whether (when 1) or not (when 0) a centroid residual of nodeIdx-th node is 0.

centroid\_residual\_is\_zero[nodeIdx], centroid\_residual\_magnitude[nodeIdx], and centroid\_residual\_sign[nodeIdx] together specify a quantized residual of each centroid. When centroid\_residual\_is\_zero[nodeIdx] is not present, it shall be inferred to be 1. When centroid\_residual\_magnitude[nodeIdx] and centroid\_residual\_sign[nodeIdx] are not present shall be inferred to be 0.

The value driftQ[nodeIdx] specifies a quantized centroid residual.

driftQ[nodeIdx] := (2 × centroid\_residual\_sign[nodeIdx] - 1) × (centroid\_residual\_is\_zero[nodeIdx] + centroid\_residual\_magnitude[nodeIdx])

has\_face\_vertex[nodeIdx][fvIdx] is equal to 1 specifies a face vertex is added on the nodeIdx*-*th TriSoup node’s face which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin. has\_face\_vertex[nodeIdx][fvIdx] is equal to 0 specifies a face vertex is not added on the nodeIdx*-*th TriSoup node’s face which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin. When has\_face\_vertex[nodeIdx][fvIdx] is not present, it shall be inferred to 0.

#### Presence of has\_face\_vertex

When trisoup\_face\_vertex\_flag is equal to 1, the presence of has\_face\_vertex[nodeIdx][fvIdx] is specified by the expression FaceEligible[nodeIdx][fvIdx]. It shall be present in the TriSoup syntax when FaceEligible[nodeIdx][fvIdx] is equal to 1.

Centroid[nodeIdx] is the result of the process specified in 9.4.3.2.4. FaceVertex[nodeIdx][fvIdx] is the result of the process specified in 9.4.3.2.5. Let adjNode[nodeIdx][fvIdx] be the node index of the adjacent TriSoup node in the direction of fvIdx-th component from the nodeIdx-th TriSoup node.

When all the conditions listed below are true, FaceEligible[nodeIdx][fvIdx] is set to 1.

* centroid\_residual\_is\_zero[nodeIdx] is equal to 0, Centroid[nodeIdx][k] is greater than or equal to 0 and Centroid[nodeIdx][k] is less than or equal to TriSoupNodeSize[nodeIdx][k] for each k (k = 0, 1, 2), and
* adjNode[nodeIdx][fvIdx]-th TriSoup node is available, and
* centroid\_residual\_is\_zero[adjNode[nodeIdx][fvIdx]] is equal to 0, Centroid[adjNode[nodeIdx][fvIdx]][k] is greater than or equal to 0 and Centroid[adjNode[nodeIdx][fvIdx]][k] is less than or equal to TriSoupNodeSize[adjNode[nodeIdx][fvIdx]][k] for each k (k = 0, 1, 2), and
* ((fvIdx < 2) ? (has\_vertex[nodeIdx][3 − fvIdx] + has\_vertex[nodeIdx][6 − fvIdx] + has\_vertex[nodeIdx][7 − fvIdx] + has\_vertex[nodeIdx][11 − fvIdx]) : (has\_vertex[nodeIdx][8] + has\_vertex[nodeIdx][9] + has\_vertex[nodeIdx][10] + has\_vertex[nodeIdx][11])) is equal to either 2 or 3, and
* Both dp0 and dp1, which are derived as follows, are greater than 0.

evCnt = numTriSoupVertices[nodeIdx]  
distMin = 0x7fffffff  
for(evIdx = 0; evIdx < ((evCnt == 3) ? 1 : evCnt); evIdx++){  
 ev0 = evIdx  
 ev1 = evIdx + 1  
 if( ev1 >= evCnt )  
 ev1 −= evCnt  
 evCoord0 = sortedVertices[nodeIdx][ev0] + 128  
 evCoord1 = sortedVertices[nodeIdx][ev1] + 128  
 if( ( TriSoupNodeSize[nodeIdx][fvIdx] ≠ evCoord0[fvIdx] ) ||   
 ( TriSoupNodeSize[nodeIdx][fvIdx] ≠ evCoord1[fvIdx] ) )  
 continue;  
 middlePoint = ( evCoord0 + evCoord1 ) / 2  
 distVec = (middlePoint – FaceVertex[nodeIdx][fvIdx]) >> 8  
 dist = distVec[0] × distVec[0] + distVec[1] × distVec[1] +   
 distVec[2] × distVec[2]  
 if( distMin > dist ) {  
 evIdxMin[nodeIdx][fvIdx][0] = ev0  
 evIdxMin[nodeIdx][fvIdx][1] = ev1  
 distMin = dist  
 }  
}

eeVec = sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][1]] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]]  
eeVecNorm = IntSqrt(eeVec[0] × eeVec[0] + eeVec[1] × eeVec[1] +   
 eeVec[2] × eeVec[2])  
eUnitVec = eeVecNorm ? ( ( eeVec << 8 ) / eeVecNorm ) : 0  
enVec = (faceVertex[nodeIdx][fvIdx] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]]) ×   
 eUnitVec >> 8  
nfVec = faceVertex[nodeIdx][fvIdx] −   
 sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]] −   
 ((enVec × eUnitVec) >> 8);

dp0 = DriftVec[nodeIdx] × nfVec  
dp1 = DriftVec[adjNode[nodeIdx][fvIdx]] × nfVec

Otherwise, when all the conditions listed below are true, FaceEligible[nodeIdx][fvIdx] is set to 1.

* centroid\_residual\_is\_zero[nodeIdx] is equal to 0, Centroid[nodeIdx][k] is greater than or equal to 0 and Centroid[nodeIdx][k] is less than or equal to TriSoupNodeSize[nodeIdx][k] for each k (k = 0, 1, 2), and
* adjNode[nodeIdx][fvIdx]-th TriSoup node is available, and
* centroid\_residual\_is\_zero[adjNode[nodeIdx][fvIdx]] is equal to 0, Centroid[adjNode[nodeIdx][fvIdx]][k] is greater than or equal to 0 and Centroid[adjNode[nodeIdx][fvIdx]][k] is less than or equal to TrSoupNodeSize[adjNode[nodeIdx][fvIdx]][k] for each k (k = 0, 1, 2), and
* numTriSoupVertices[nodeIdx] is equal to 2 and numTriSoupVertices[adjNode[nodeIdx][fvIdx]] is equal to 2.

Otherwise, FaceEligible[nodeIdx][fvIdx] is set to 0.

### TriSoup decoding process

TriSoup decoding process consists on the following three processes.

* Decoding of TriSoup edge vertices located on TriSoup edges (9.4.3.1)
* Derivation of TriSoup vertices in TriSoup nodes (9.4.3.2)
* Determination of decoded points by the voxelization of TriSoup triangles (9.4.3.3)

#### Decoding of TriSoup edge vertices located on TriSoup edges

Vertex flag (segment indicator) and vertex position of each TriSoup edge are decoded as follows.

* determination of neighbouring information (9.4.3.1.1)
* determination of contexts and decoding of vertex flag and vertex position using dynamic OBUF (9.4.3.1.2)
* initialization and decoding TriSoup vertices bits using OBUF (9.4.3.1.3)

##### Determination of Neighbouring Information

Mask information with 16 bits length neighbourMask[edgeIdx] is set for all unique TriSoup Edges (edgeIdx = 0, 1,…, numberTriSoupEdges) as follows.

* Assume edgeStartPos of edgeIdx-th unique TriSoup edge is (k0, k1, k2).
* A variable TCS is set to TriSoupCubeSize.

TCS := TriSoupCubeSize

* edgeDirection of edgeIdx-th unique TriSoup Edge is derived as follows,
  + If edgeEndPos of i-th unique TriSoup Edge is (k0+TCS, k1, k2), edgeDirection is set to 0.
  + Otherwise, if edgeEndPos of edgeIdx-th unique TriSoup Edge is (k0, k1+TCS, k2), edgeDirection is set to 1.
  + Otherwise, edgeDirection is set to 2.
* Construct neighbourMask[edgeIdx] as defined in 9.4.3.1.4.

Then edgePattern[edgeIdx] is derived for all unique TriSoup edges (edgeIdx = 0, 1,…, numberTriSoupEdges). edgePattern[edgeIdx] consists of unique edge indexes of nine neighbouring edges a through edgeIdx in the Table 25. When the corresponding edge of variable j (j  = 0, 1,…, 8) exists, edgePattern[edgeIdx][j] is set to the unique edge index of the corresponding edge. Otherwise, when the corresponding edge of variable j does not exist, edgePattern[edgeIdx][j] is set to -1.

Table 25 — edgeStartPos and edgeEndPos of neighbour edges

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | edgeStartPos, edgeEndPos | | |
| j | edge | edgeDirection = 0 | edgeDirection = 1 | edgeDirection = 2 |
| - | E | (k0, k1, k2), (k0 + TCS, k1, k2) | (k0, k1, k2),  (k0, k1 + TCS, k2) | (k0, k1, k2),  (k0, k1, k2 + TCS) |
| 0 | a | (k0 - TCS, k1, k2),  (k0, k1, k2) | (k0, k1 - TCS, k2),  (k0, k1, k2) | (k0, k1, k2 - TCS),  (k0, k1, k2) |
| 1 | b | (k0, k1, k2 - TCS),  (k0, k1, k2) | (k0, k1, k2 - TCS),  (k0, k1, k2) | (k0, k1 - TCS, k2),  (k0, k1, k2) |
| 2 | c | (k0, k1 - TCS, k2),  (k0, k1, k2) | (k0 - TCS, k1, k2),  (k0, k1, k2) | (k0 - TCS, k1, k2),  (k0, k1, k2) |
| 3 | d | (k0, k1, k2),  (k0, k1, k2 + TCS) | (k0, k1, k2),  (k0, k1, k2 + TCS) | (k0 - TCS, k1, k2 + TCS),  (k0, k1, k2 + TCS) |
| 4 | e | (k0, k1, k2),  (k0, k1 + TCS, k2) | (k0 - TCS, k1 + TCS, k2),  (k0, k1 + TCS, k2) | (k0, k1 - TCS, k2 + TCS),  (k0, k1, k2 + TCS) |
| 5 | f | (k0, k1 - TCS, k2),  (k0 + TCS, k1 - TCS, k2) | (k0 - TCS, k1, k2),  (k0 - TCS, k1 + TCS, k2), | (k0 - TCS, k1, k2),  (k0 - TCS, k1, k2 + TCS) |
| 6 | g | (k0, k1, k2 - TCS),  (k0 + TCS, k1, k2 - TCS) | (k0, k1, k2 - TCS),  (k0, k1 + TCS, k2 - TCS) | (k0, k1 - TCS, k2),  (k0, k1 - TCS, k2 + TCS) |
| 7 | h | (k0, k1 - TCS, k2),  (k0, k1 - TCS, k2 + TCS) | (k0, k1, k2 - TCS),  (k0 + TCS, k1, k2 - TCS), | (k0, k1 - TCS, k2),  (k0 + TCS, k1 - TCS, k2), |
| 8 | i | (k0, k1, k2 - TCS),  (k0, k1 + TCS, k2 - TCS) | (k0 - TCS, k1 + TCS, k2 + TCS),  (k0, k1 + TCS, k2 + TCS) | (k0 - TCS, k1 + TCS, k2 + TCS),  (k0, k1 + TCS, k2 + TCS) |

##### Determination of Contexts and Decoding of Vertex Flag and Vertex Position

The following process is applied to all unique edges.

* Derive common information for vertex flag and position as defined in 9.4.3.1.5.
* Derive context ctxMap1 and ctxMap2 for vertex flag.

ctxMap1 = min(nclosestPattern, 2) × 15 × 2 + (neighbEdge-1) × 2 + ((ctx1 == 4));  
ctxMap2 = neighbEnd << 11;  
ctxMap2 |= (patternClose & (0b00000110)) << 9 - 1  
ctxMap2 |= direction << 7  
ctxMap2 |= (patternClose & (0b00011000))<< 5 - 3   
ctxMap2 |= (patternClose & (0b00000001))<< 4   
orderedPclosePar =   
 (((pattern >> 5) & 3) << 2) + (!!(pattern & 128) << 1) + !!(pattern & 256)  
ctxMap2 |= orderedPclosePar

* Decode vertex flag vertex\_flag[edgeIdx] as shown in 9.4.3.1.3.
* If vertex\_flag[edgeIdx] is equal to 1, the following processes are applied to decode vertex\_position[edgeIdx].
* Derive context ctxMap1 and ctxMap2 for the first bit of position.

ctxFullNbounds = (4 × (ctx0 <= 1 ? 0 : (ctx0 >= 3 ? 2 : 1)) + (max(1, ctx1) - 1)) × 2 + (ctxE == 3)  
ctxMap1 = ctxFullNbounds × 2 + (nclosestStart > 0)  
ctxMap2 = missedCloseStart << 8  
ctxMap2 |= (patternClosest & 1) << 7  
ctxMap2 |= direction << 5  
ctxMap2 |= patternClose & (0b00011111)

* Decode the first bit of position vertex\_position[edgeIdx][0] as shown in 9.4.3.1.3 and decoded bit is set as v.
* When trisoup\_vertex\_number\_bits is greater than 1, the following processes are applied.
  + derive context for the second bit of position

ctxMap1 = ctxFullNbounds × 2 + (nclosestStart > 0)  
ctxMap1 = (ctxMap1 << 1) + v  
ctxMap2 = missedCloseStart << 8  
ctxMap2 |= (patternClose & 1) << 7  
ctxMap2 |= (patternClosest & 1) << 6  
ctxMap2 |= direction << 4  
ctxMap2 |= (patternClose & (0b00011111)) >> 1  
orderedPclosePar = (((patternClose >> 5) & 3) << 2) + (!!(patternClose & 128) << 1) + !!(patternClose & 256)  
ctxMap2 = (ctxMap2 << 4) + orderedPclosePar

* + Decode the second bit of position vertex\_position[edgeIdx][1] as shown in 9.4.3.1.3 and variable v is updated.

v = (v << 1) + vertex\_position[edgeIdx][1]

* + When trisoup\_vertex\_number\_bits is greater than 2, the following processes are applied.
    - derive context for the third bit of position

ctxBits3 = (6 × (ctx0 >> 1) + missedCloseStart) × 2 + (ctxE == 3)  
ctxBits3 = 4 × ctxBits3 + v

* + - Decode the third bit of position vertex\_position[edgeIdx][2] as shown in 9.4.3.1.3 and variable v is updated.

v = (v << 1) + vertex\_position[edgeIdx][2]

* + - When trisoup\_vertex\_number\_bits is greater than 3, decode remaining bits of vertex\_position[edgeIdx][bits] (bits=3,…,trisoup\_vertex\_number\_bits-1) by bypass decoding and variable v is updated.

##### Initialization and Decoding TriSoup Vertices Bits Using OBUF

The following processes are applied to initialize OBUF instances.

* To decode vertex position bits of TriSoup nodes, a first buffer (TriSoup buffer) is created for all OBUF instances to be later used by TriSoup coding according to 12.3, and the OBUF buffer size obufBufferSize and the buffer depth obufLeafDepth of fully deployed trees are set as follows.

obufBufferSize := 20000  
obufLeafDepth := 4

* The context arrays OBUF ACPMs are created according to 12.2.2.
* All OBUF instances used by TriSoup coding are created and initialized according to 12.2. The size nBit1 and nBit2 are set as according to Table 29.
  + When i is equal to 0, the OBUF instance TOI[ i ] is used for coding vertex presence flag for an edge.
  + When i is equal to 1, the OBUF instance TOI[ i ] is used for coding the first bit of vertex position.
  + When i is equal to 2, the OBUF instance TOI[ i ] is used for coding the second bit of vertex position.
* The array ctxIdxMap[ j][i ] is initialized according to Table 30, Table 31 and Table 32.

The following processes are applied to decode TrisSoup vertices bits.

* Each bit of vertex position of coded TriSoup nodes is decoded according to 12.4 by using two contextual information ctxMap1 and ctxMap2, which are set as the first contextual informationinfo1and the second contextual informationinfo2,and calling the OBUF instances of TriSoup coding to get the decoded bin.
* During updating of dynamic OBUF trees, the primary part remains unchanged (is never reduced) and the dynamic of secondary part may be reduced by applying a OBUF tree updating process according to the 12.4.1.2.

Table 22 — the size values (nBit1,nBit2) of TriSoup OBUF instance TOI[ i ]

|  |  |  |  |
| --- | --- | --- | --- |
|  | *i* | | |
| **0** | **1** | **2** |
| **(**nBit1**,**nBit2**)** | (7,15) | (6,11) | (7,15) |

Table 23 — Initial values of CtxIdxMap[ j][0] for coding vertex presence flag of an edge

| j | CtxIdxMap[ j][0] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 8 .. 15 | 15 | 15 | 42 | 96 | 71 | 37 | 15 | 15 |
| 16 .. 23 | 22 | 51 | 15 | 15 | 30 | 27 | 15 | 15 |
| 24 .. 31 | 64 | 15 | 48 | 15 | 224 | 171 | 127 | 24 |
| 32 .. 39 | 127 | 34 | 80 | 46 | 141 | 44 | 66 | 49 |
| 40 .. 47 | 127 | 116 | 140 | 116 | 105 | 39 | 127 | 116 |
| 48 ..55 | 114 | 46 | 172 | 109 | 60 | 73 | 181 | 161 |
| 56 ..63 | 112 | 65 | 240 | 159 | 127 | 127 | 127 | 87 |
| 64 .. 71 | 183 | 127 | 116 | 116 | 195 | 88 | 152 | 141 |
| 72 .. 79 | 228 | 141 | 127 | 80 | 127 | 127 | 160 | 92 |
| 80 .. 87 | 224 | 167 | 129 | 135 | 240 | 183 | 240 | 184 |
| 88 .. 95 | 240 | 240 | 127 | 127 | 127 | 127 | 127 | 127 |
| 96 ..103 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 104 ..111 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 112 ..119 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 120 ..127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

Table 24 — Initial values of CtxIdxMap[ j][1] for coding the first bit of vertex position

| j | CtxIdxMap[ j][1] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 116 | 127 | 118 | 15 | 104 | 56 | 97 | 15 |
| 8 .. 15 | 96 | 15 | 29 | 15 | 95 | 15 | 46 | 15 |
| 16 .. 23 | 196 | 116 | 182 | 53 | 210 | 104 | 163 | 69 |
| 24 .. 31 | 169 | 15 | 114 | 15 | 121 | 15 | 167 | 63 |
| 32 .. 39 | 240 | 127 | 184 | 92 | 240 | 163 | 197 | 77 |
| 40 .. 47 | 239 | 73 | 179 | 59 | 213 | 48 | 185 | 108 |
| 48 ..55 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 56 ..63 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

Table 25 — Initial values of CtxIdxMap[ j][2] for coding the second bit of vertex position

| j | CtxIdxMap[ j][2] | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 .. 7 | 141 | 127 | 127 | 127 | 189 | 81 | 36 | 127 |
| 8 .. 15 | 143 | 105 | 103 | 116 | 201 | 60 | 38 | 116 |
| 16 .. 23 | 116 | 127 | 15 | 127 | 153 | 59 | 15 | 116 |
| 24 .. 31 | 69 | 105 | 15 | 127 | 158 | 93 | 36 | 79 |
| 32 .. 39 | 141 | 161 | 116 | 127 | 197 | 102 | 53 | 127 |
| 40 .. 47 | 177 | 125 | 88 | 79 | 209 | 75 | 102 | 28 |
| 48 ..55 | 95 | 74 | 72 | 56 | 189 | 62 | 78 | 18 |
| 56 ..63 | 88 | 116 | 28 | 45 | 237 | 100 | 152 | 35 |
| 64 .. 71 | 141 | 240 | 127 | 127 | 208 | 133 | 101 | 141 |
| 72 .. 79 | 186 | 210 | 168 | 98 | 201 | 124 | 138 | 15 |
| 80 .. 87 | 195 | 194 | 103 | 94 | 229 | 82 | 167 | 23 |
| 88 .. 95 | 92 | 197 | 112 | 59 | 185 | 87 | 156 | 79 |
| 96 ..103 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 104 ..111 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 112 ..119 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 120 ..127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |

##### Construction of neighbour mask

Construct neighbourMask[edgeIdx] for each TriSoup edge corresponding to edgeIdx as follows.

* Initialize neighbourMask[edgeIdx] as 0.
* When node e1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= 1

* When node e2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 1)

* When node e3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 2)

* When node e4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 3)

* When node a1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 4)

* When node a2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 5)

* When node a3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 6)

* When node a4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 7)

* When node a1 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 8)

* When node a2 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 9)

* When node a3 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 10)

* When node a4 exists, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 11)

* When edgeDirection is equal to 1, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 13)

* When edgeDirection is equal to 2, neighbourMask[edgeIdx] is updated as follows.

neighbourMask[edgeIdx] |= (1 << 14)

Table 25 — Locations of neighbour nodes

|  |  |  |  |
| --- | --- | --- | --- |
| Node | TriSoupNodeLoc | | |
| edgeDirection = 0 | edgeDirection = 1 | edgeDirection = 2 |
| e1 | (k0, k1, k2) | (k0, k1, k2) | (k0, k1, k2) |
| e2 | (k0, k1 - TCS, k2) | (k0 - TCS, k1, k2) | (k0, k1 - TCS, k2) |
| e3 | (k0, k1, k2 - TCS) | (k0, k1, k2 - TCS) | (k0 - TCS, k1 - TCS, k2) |
| e4 | (k0, k1 - TCS, k2 - TCS) | (k0 - TCS, k1, k2 - TCS) | (k0 - TCS, k1, k2) |
| a1 | (k0 - TCS, k1, k2) | (k0, k1 - TCS, k2) | (k0, k1, k2 - TCS) |
| a2 | (k0 - TCS, k1 - TCS, k2) | (k0 - TCS, k1 - TCS, k2) | (k0, k1 - TCS, k2 - TCS) |
| a3 | (k0 - TCS, k1, k2 - TCS) | (k0, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 - TCS) |
| a4 | (k0 - TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1, k2 - TCS) |
| b1 | (k0 + TCS, k1, k2) | (k0, k1 + TCS, k2) | (k0, k1, k2 + TCS) |
| b2 | (k0 + TCS, k1 - TCS, k2) | (k0 - TCS, k1 + TCS, k2) | (k0, k1 - TCS, k2 + TCS) |
| b3 | (k0 + TCS, k1, k2 - TCS) | (k0, k1 + TCS, k2 - TCS) | (k0 - TCS, k1 - TCS, k2 + TCS) |
| b4 | (k0 + TCS, k1 - TCS, k2 - TCS) | (k0 - TCS, k1 + TCS, k2 - TCS) | (k0 - TCS, k1, k2 + TCS) |

##### Derive common information for vertex flag and position

Derive common information to decode vertex flag and position as follows.

* Derive ctxE, ctx0, ctx0, and direction as follows.

ctxE = (!!(neighbourMask[edgeIdx] & 1)) + (!!(neighbourMask[edgeIdx] & 2)) + (!!(neighbourMask[edgeIdx] & 4)) + (!!(neighbourMask[edgeIdx] & 8)) - 1

ctx0 = (!!(neighbourMask[edgeIdx] & 16)) + (!!(neighbourMask[edgeIdx] & 32)) + (!!(neighbourMask[edgeIdx] & 64)) + (!!(neighbourMask[edgeIdx] & 128))

ctx1 = (!!(neighbourMask[edgeIdx] & 256)) + (!!(neighbourMask[edgeIdx] & 512)) + (!!(neighbourMask[edgeIdx] & 1024)) + (!!(neighbourMask[edgeIdx] & 2048))

direction = neighbourMask[edgeIdx] >> 13

* Derive pattern, patternClose, patternClosest, and nclosestPattern.

for(v = 0; v < 9; v++){  
 if (segind[edgePattern[v]]){  
 pattern |= 1 << v  
 vertexPos2bits =   
 vertices[correspondanceSegment2V[idxEdge]] >> max(0, nbitsVertices - 2)  
 if (towardOrAway[v18])  
 vertexPos2bits = max2bits - vertexPos2bits; // reverses for away  
 if (vertexPos2bits >= mid2bits)  
 patternClose |= 1 << v;  
 if (vertexPos2bits >= max2bits)  
 patternClosest |= 1 << v;  
 nclosestPattern += vertexPos2bits >= max2bits && v <= 4;  
 }  
}

* Derive missedCloseStart and nclosestStart.

missedCloseStart = !(pattern & 2) + !(pattern & 4)  
nclosestStart =  
 !!(patternClosest & 1) + !!(patternClosest & 2) + !!(patternClosest & 4)

* Derive neighbEdge, neighbEnd, and neighbStart.

neighbEdge = (neighbourMask[edgeIdx] >> 0) & 15  
neighbEnd = (neighbourMask[edgeIdx] >> 4) & 15  
neighbStart = (neighbourMask[edgeIdx] >> 8) & 15  
if (direction == 2) {  
 neighbEdge = ((neighbourMask[edgeIdx] >> 0 + 0) & 1)  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 3) & 1) << 1  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 1) & 1) << 2  
 neighbEdge += ((neighbourMask[edgeIdx] >> 0 + 2) & 1) << 3  
  
 neighbEnd = ((neighbourMask[edgeIdx] >> 4 + 0) & 1)  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 3) & 1) << 1  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 1) & 1) << 2  
 neighbEnd += ((neighbourMask[edgeIdx] >> 4 + 2) & 1) << 3  
  
 neighbStart = ((neighbourMask[edgeIdx] >> 8 + 0) & 1)  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 3) & 1) << 1  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 1) & 1) << 2  
 neighbStart += ((neighbourMask[edgeIdx] >> 8 + 2) & 1) << 3  
}

#### Derive TriSoup vertices corresponding each TriSoup node

TriSoup vertices for each TriSoup node shall be derived as following steps:

* Derive information related to vertex merge (9.4.3.2.1)
* Derive TriSoup vertices corresponding each TriSoup edge (9.4.3.2.2)
* Derive initial position of centroid vertices (9.4.3.2.3),
* Determine the dominant axis and sort vertices (9.4.3.2.4),
* Derive refined centroid position (9.4.3.2.5),
* Derive TriSoup face vertices corresponding each TriSoup node (9.4.3.2.6),
* Insert TriSoup face vertices to corresponding sorted TriSoup vertices (9.4.3.2.7)

##### Derive information related to vertex merge

When trisoup\_vertex\_merge is equal to 1, expressions activeNode[nodeIdx] and activeCorner[nodeIdx][n] for each TriSoup node corresponding to nodeIdx are derived as follows.

The expression activeNode[nodeIdx] indicates whether the TriSoup node corresponding to nodeIdx need to be processed.

numVertex = 0  
for (edgeTag = 0; edgeTag < 12; edgeTag++) {  
 if (edgeVertex[nodeIdx][edgeTag] >= 0) {  
 numVertex++  
 }  
}  
if (numVertex >= 6) {  
 activeNode[nodeIdx] = 1  
} else {  
 activeNode[nodeIdx] = 0  
}

The expression activeCorner[nodeIdx][n] indicates whether the corner corresponding to n (n=0,…,7) of the TriSoup node corresponding to nodeIdx need to be dealt with.

for (n = 0; n < 8; n++) {  
 ns = TriSoupNodeLoc[nodeIdx][0] + TriSoupNodeSize[nodeIdx][0] × Bit(n, 0)  
 nt = TriSoupNodeLoc[nodeIdx][1] + TriSoupNodeSize[nodeIdx][1] × Bit(n, 1)  
 nv = TriSoupNodeLoc[nodeIdx][2] + TriSoupNodeSize[nodeIdx][2] × Bit(n, 2)  
 if (vertexNumOfCorner[ns][nt][nv] >= 4) {  
 activeCorner[nodeIdx][n] = 1  
 } else {  
 activeCorner[nodeIdx][n] = 0  
 }  
}

##### Derive TriSoup vertices corresponding each TriSoup edge

Vertices vertex[nodeIdx][vIdx][k] for each TriSoup node corresponding to nodeIdx are derived as follows.

The variable vIdx is initialized by 0.

For all edges (edgeTag=0,…,11) on a TriSoup node corresponding to nodeIdx, the edge direction and the expression edgeVertexModify[nodeIdx][edgeTag], which indicates whether the edge indexed by edgeTag on the TriSoup node corresponding to nodeIdx need to be modified are derived as follows.

* When edgeVertex[nodeIdx][edgeTag] is greater than or equal to 0, the following processes are applied.
  + Derive direction[k] (k=0, 1, 2):

for (k = 0; k < 3; k++) {  
 direction[k] = edgePosEnd[nodeIdx][edgeTag][k] - edgePosStart[nodeIdx][edgeTag][k]  
 }

* + The expression edgeVertexModify[nodeIdx][edgeTag] is initialized by 0.
  + When trisoup\_vertex\_merge is equal to 1 and activeNode[nodeIdx] is equal to 1, the expression edgeVertexModify[nodeIdx][edgeTag] is modified as follows.

relativePos = (edgeVertex[nodeIdx][edgeTag] << (8 + bitDropped))  
               + (128 << bitDropped)  
n = edgePosStartShift[edgeTag][0]  
    + (edgePosStartShift[edgeTag][1] << 1)  
    + (edgePosStartShift[edgeTag][2] << 2)  
if (relativePos < thVertexMerge && activeCorner[nodeIdx][n]) {  
 edgeVertexModify[nodeIdx][edgeTag] = 1  
}  
  
n = edgePosEndShift[edgeTag][0]  
    + (edgePosEndShift[edgeTag][1] << 1)  
    + (edgePosEndShift[edgeTag][2] << 2)  
if ((TriSoupCubeSize << 8) - relativePos < thVertexMerge   
      && activeCorner[nodeIdx][n]) {  
 edgeVertexModify[nodeIdx][edgeTag] = 1  
}

* + When edgeVertexModify[nodeIdx][edgeTag] is equal to 0, dequantized vertex positions vertex[nodeIdx][vIdx][k] is derived.

for (k = 0; k < 3; k++) {  
  vertex[nodeIdx][vIdx][k] =  
    (edgePosStart[nodeIdx][edgeTag][k] - TriSoupNodeLoc[nodeIdx][k]) << 8  
 vertex[nodeIdx][vIdx][k] -= 128  
 if (direction[k] > 0){  
 vertex[nodeIdx][vIdx][k] +=  
      (edgeVertex[nodeIdx][edgeTag] << (bitDropped + 8)) + (128 << bitDropped)  
 }  
}  
vIdx++

When trisoup\_vertex\_merge is equal to 1 and activeNode[nodeIdx] is equal to 1, vertex[nodeIdx][vIdx][k] is added as follows.

for (n = 0; n S 8; n++){  
 if (activeCorner[nodeIdx][n]){  
 vertex[nodeIdx][vIdx][0] = ((TriSoupNodeSize[nodeIdx][0] × Bit(n, 0)) << 8) – 128  
 vertex[nodeIdx][vIdx][1] = ((TriSoupNodeSize[nodeIdx][1] × Bit(n, 1)) << 8) - 128  
 vertex[nodeIdx][vIdx][2] = ((TriSoupNodeSize[nodeIdx][2] × Bit(n, 2)) << 8) - 128  
 vIdx++  
 }  
}

Number of vertices numTriSoupVertices[nodeIdx] is set by vIdx.

numTriSoupVertices[nodeIdx] = vIdx

##### Derive initial position of centroid vertices

The initial position of centroid vertices Centroid[nodeIdx][k] for each TriSoup node nodeIdx is derived as follows as a weighted sum. Firstly, weights weights[i] associated with edge vertices of the node are initialized to zero.

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 weights[i] = 0  
}

A total weight wTotal is also initialized to zero.

wTotal = 0

Then, weights weights[i] and total weight wTotal are derived as follows.

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 w = abs(vertex[nodeIdx][i][0] - vertex[nodeIdx][i2][0])  
 w += abs(vertex[nodeIdx][i][1] - vertex[nodeIdx][i2][1])  
 w += abs(vertex[nodeIdx][i][2] - vertex[nodeIdx][i2][2])  
  
 weights[i] += w  
 weights[i2] += w  
 wTotal += 2\*w  
}

The position Centroid[nodeIdx][] of the centroid is obtained as a weighted sum of edge vertices.

Centroid[nodeIdx][0] = 0  
Centroid[nodeIdx][1] = 0  
Centroid[nodeIdx][2] = 0  
for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 Centroid[nodeIdx][0] += weights[i] \* vertex[nodeIdx][i][0]  
 Centroid[nodeIdx][1] += weights[i] \* vertex[nodeIdx][i][1]  
 Centroid[nodeIdx][2] += weights[i] \* vertex[nodeIdx][i][2]  
}  
Centroid[nodeIdx][0] = Centroid[nodeIdx][0] / wTotal  
Centroid[nodeIdx][1] = Centroid[nodeIdx][1] / wTotal  
Centroid[nodeIdx][2] = Centroid[nodeIdx][2] / wTotal

##### Determine the dominant axis and sort TriSoup vertices

When numTriSoupVertices[nodeIdx] is greater than 3, the dominant axis and sorted TriSoup vertices are derived as follows. Otherwise, the dominant axis is set to 0.

* The sum of triangle areas for each axis k (0, 1, 2) is derived as follows.
  + Derive two axes to calculate triangle areas based on

Table 25 — Corresponding axes S1 and S2 with k

|  |  |  |
| --- | --- | --- |
| **Axis k** | **S1** | **S2** |
| 0 | 2 | 1 |
| 1 | 2 | 0 |
| 2 | 1 | 0 |

* + The score to sort each vertex is calculated in anti-clockwise order according to the edges of the projected square as follows:

for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 if(vertex[nodeIdx][i][S1] >= TriSoupNodeSize[nodeIdx][S1]){  
 Score[i] = vertex[nodeIdx][i][S2]  
 } else if (vertex[nodeIdx][i][S2] >= TriSoupNodeSize[nodeIdx][S2]){  
 Score[i] = TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] -   
 vertex[nodeIdx][i][S1]  
 } else if (vertex[nodeIdx][i][S1] <= 0) {  
 Score[i] = 2 × TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] -   
 vertex[nodeIdx][i][S2]  
 } else {  
 Score[i] = 2 × TriSoupNodeSize[nodeIdx][S2] + TriSoupNodeSize[nodeIdx][S1] +   
 vertex[nodeIdx][i][S1]  
 }  
}

* + Sort vertices by ascending order of scores.
  + Calculate the sum of triangle areas as follows

for (Axis = 0; Axis < 3; Axis++){  
 Area = 0  
 for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 for (k = 0; k < 3; k++){  
 v1[k] = sortedVertices[nodeIdx][i][k] – Centroid[nodeIdx][k]  
 v2[k] = sortedVertices[nodeIdx][i2][k] – Centroid[nodeIdx][k]  
 }  
 cP = CrossProduct(v1,v2)  
 Area += abs(cP[Axis])   
 }  
}

* The axis with the largest area is determined as the dominant axis, and sorted vertices associated with the dominant axis is retained.

##### Derive refined centroid position

When all the following conditions are true, centroid position Centroid[nodeIdx][k] for each TriSoup node shall be refined.

* trisoup\_centroid\_vertex\_residual\_flag is equal to 1,
* At least one of the following conditions are true,
  + numTriSoupVertices[nodeIdx] is greater than 3,
  + All the conditions listed in 9.4.3.2.8 are true.

Otherwise, output of this process is the same as the output of 9.4.3.2.2.

Refinement process of centroid position is as follows.

* Derive a normal vector by retained dominant axis and sorted vertices derived in 9.4.3.2.3.

accumNormal = 0  
if (numTriSoupVertices[nodeIdx] ≠ 2){  
 for (i = 0; i < numTriSoupVertices[nodeIdx]; i++){  
 i2 = i + 1  
 if (i2 >= numTrisoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 accumNormal = CrossProduct (sortedVertices[nodeIdx][i] – Centroid[nodeIdx],  
 sortedVertices[nodeIdx][i2] – Centroid[nodeIdx])  
 }  
} else {  
 accumNormal = Centroid[nodeIdx] - TriSoupNodeSize[ adjNode[nodeIdx][fvIdx] ] -  
 Centroid[adjNode[nodeIdx][fvIdx]]  
}  
absNormal = isqrt(accumNormal[0] × accumNormal[0] +   
 accumNormal[1] × accumNormal[1] +  
 accumNormal[2] × accumNormal[2])  
for (k = 0; k < 3; k++){  
 normal[k] = (accumNormal[k] << 8) / absNormal

}

* Derive context indexes ctxMinMax for ctxDrift0 (centroid\_residual\_is\_zero[nodeIdx]) and context indexes ctxIdxDritSign, lowS and highS for ctxDriftSign (centroid\_residual\_sign[nodeIdx]) as defined in 9.4.3.2.9.
* Derive dequantized centroid position residual driftDQ from decoded centroid position residual driftQ.

driftDQ = 0  
if (driftQ){  
 driftDQ = abs(driftQ) << bitDropped + 6  
 half = 1 << 5 + bitDropped  
 driftDQ += 2 × half / 3 – half  
 if (driftQ < 0)  
 driftDQ = -driftDQ  
}

* Derive drift vector DriftVec[nodeIdx][].

for (k = 0; k < 3; k++) {  
 DriftVec[nodeIdx][k] = (driftDQ × normal[k]) >> 6  
}

* Refine centroid position.

for (k = 0; k < 3; k++) {  
 Centroid[nodeIdx][k] += DriftVec[nodeIdx] [k]  
}

* Bound centroid position.

for (k = 0; k < 3; k++) {  
 Centroid[nodeIdx][k] = max(-128, Centroid[nodeIdx][k])  
 Centroid[nodeIdx][k] =   
 min(((TriSoupCubeSize - 1) << 8) + 127, Centroid[nodeIdx][k])  
}

##### Derive TriSoup face vertices corresponding each TriSoup node

When trisoup\_face\_vertex\_flag is equal to 1, for each TriSoup node, TriSoup face vertex faceVertex[nodeIdx][fvIdx] is set to the intersection of the face, which is orthogonal to the direction of fvIdx-th component and located far side from the slice origin, and the line segment between the centroid vertex of nodeIdx-th TriSoup node and that of adjNode[nodeIdx][fvIdx]-th TriSoup node for each fvIdx (fvIdx = 0, 1, 2) as follows:

c0facePos = (TriSoupNodeSize[nodeIdx][fvIdx] << 8 ) − 128  
c0 = Centroid[nodeIdx]  
c1 = Centroid[adjNode[nodeIdx][fvIdx]]  
c1[fvIdx] += TriSoupNodeSize[nodeIdx][fvIdx] << 8  
denom = c1[fvIdx] − c0[fvIdx]  
t = denom ? (((c0facePos − c0[fvIdx]) << 8) / denom) : 0  
faceVertex[nodeIdx][fvIdx] = c0 + ((t × (c1 − c0) + 128) >> 8)  
faceVertex[nodeIdx][fvIdx][fvIdx] = c0facePos;

##### Insert TriSoup face vertices to corresponding sorted TriSoup vertices

For each TriSoup node, when has\_face\_vertex[nodeIdx][fvIdx] is equal to 1 and numTriSoupVertices[nodeIdx] is not equal to 2, faceVertex[nodeIdx][fvIdx] is added to sortedVertices[nodeIdx] and sortedVertices[adjNode[nodeIdx][fvIdx]] by the following steps for each fvIdx (fvIdx = 0, 1, 2):

* faceVertex[nodeIdx][fvIdx] is inserted between sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][0]] and sortedVertices[nodeIdx][evIdxMin[nodeIdx][fvIdx][1]].
* Face vertex adjFaceVertex to be added to sortedVertices[adjNode[nodeIdx][fvIdx]] is derived as follows:

adjFaceVertex = faceVertex[nodeIdx][fvIdx]  
adjFaceVertex[fvIdx] = −128

* Indices of sortedVertices[adjNode[nodeIdx][fvIdx]] adjEvIdxMin0 and adjEvIdxMin1 are derived as follows:

evCnt = numTriSoupVertices[adjNode[nodeIdx][fvIdx]]  
distMin = 0x7fffffff  
for(evIdx = 0; evIdx < ((evCnt == 3) ? 1 : evCnt); evIdx++){  
 ev0 = evIdx  
 ev1 = evIdx + 1  
 if( ev1 >= evCnt )  
 ev1 −= evCnt  
 evCoord0 = sortedVertices[adjNode[nodeIdx][fvIdx]][ev0] + 128  
 evCoord1 = sortedVertices[adjNode[nodeIdx][fvIdx]][ev1] + 128  
 if( (evCoord0[fvIdx] ≠ 0) || (evCoord1[fvIdx] ≠ 0) )  
 continue;  
 middlePoint = ( evCoord0 + evCoord1 ) / 2  
 distVec = (middlePoint – adjFaceVertex) >> 8  
 dist = distVec[0] × distVec[0] + distVec[1] × distVec[1] +   
 distVec[2] × distVec[2]  
 if( distMin > dist ) {  
 adjEvIdxMin0 = ev0  
 adjEvIdxMin1 = ev1  
 distMin = dist  
 }  
}

* adjFaceVertex is inserted between sortedVertices[adjNode[nodeIdx][fvIdx]][adjEvIdxMin0] and sortedVertices[adjNode[nodeIdx][fvIdx]][adjEvIdxMin1].

For each TriSoup node, when has\_face\_vertex[nodeIdx][fvIdx] is equal to 1 and numTriSoupVertices[nodeIdx] is equal to 2 (there are only 2 vertices in sortedVertices), faceVertex[nodeIdx][fvIdx] is added to sortedVertices[nodeIdx] by the following steps for each fvIdx (fvIdx = 0, 1, 2):

* If face\_vertex[nodeIdx][fvIdx] shares the same face as the first TriSoup edge vertex in sortedVertices[nodeIdx], insert it ahead of the first element.
* If face\_vertex[nodeIdx][fvIdx] shares the same face as the second TriSoup edge vertex in sortedVertices[nodeIdx], insert it after the last element.

##### Applicable conditions of non-closed surface reconstruction

The following conditions are applicable conditions of non-closed surface reconstruction for a TriSoup node corresponding to nodeIdx.

* trisoup\_face\_vertex\_flag is equal to 1.
* numTrisoupVertices[nodeIdx] is equal to 2.
* onSameFace, which indicates that edge vertices vertex[nodeIdx][0] and vertex[nodeIdx][1] are positioned on the same node face, is true.

onSameFace = false  
for (k = 0; k < 3; k++) {  
 if (vertex[nodeIdx][0][k] == vertex[nodeIdx][1][k]) {  
 onSameFace = true  
 }  
}

* notOnSameEdge, which indicates that edge vertices vertex[nodeIdx][0] and vertex[nodeIdx][1] are not positioned on the same TriSoup edge, is true.

notOnSameEdge = true  
counter = 0  
for (k = 0; k < 3; k++) {  
 if (vertex[nodeIdx][0][k] == vertex[nodeIdx][1][k] &&  
 (vertex[nodeIdx][0][k] < 0 ||  
 vertex[nodeIdx][0][k] >= (TriSoupNodeSize[nodeIdx] << 8) – 128)) {  
 counter++  
 }  
}  
if (counter >= 2){  
 notOnSameEdge = false  
}

##### Derive context indexes for refined centroid position

Derive context indexes for refined centroid position as follows.

* Derive context indexes ctxMinMax for ctxDrift0 (centroid\_residual\_is\_zero[nodeIdx]).

minPos = sortedVertices[nodeIdx][0][dominantAxis]  
maxPos = sortedVertices[nodeIdx][0][dominantAxis]  
for (i = 1; i < numTriSoupVertices[nodeIdx]; i++){  
 if (sortedVertices[nodeIdx][i][dominantAxis] < minPos)  
 minPos = sortedVertices[nodeIdx][i][dominantAxis]  
 if (sortedVertices[nodeIdx][i][dominantAxis] > maxPos)  
 maxPos = sortedVertices[nodeIdx][i][dominantAxis]  
}  
ctxIdxDrift0 = min(8, (maxPos - minPos) >> (8 + bitDropped))

* Derive context indexes ctxIdxDritSign, lowS and highS for ctxDriftSign (centroid\_residual\_sign[nodeIdx]).

halfDropped = bitDropped == 0 ? 0 : 1 << (bitDropped - 1)  
bound = (TriSoupNodeSize[nodeIdx][dominantAxis] – 1) << 8  
for (m = 1; m < TriSoupNodeSize[nodeIdx][dominantAxis]; m++){  
 temp = Centroid[nodeIdx] + m × normal  
 if (temp[0] < 0 || temp[0] > bound   
 || temp[1] < 0 || temp[1] > bound  
 || temp[2] < 0 || temp[2] > bound)  
 break  
}  
highBound = (m – 1) + halfDropped >> bitDropped  
  
for (m = 1; m < TriSoupNodeSize[nodeIdx][dominantAxis]; m++){  
 temp = Centroid[nodeIdx] - m × normal  
 if (temp[0] < 0 || temp[0] > bound   
 || temp[1] < 0 || temp[1] > bound  
 || temp[2] < 0 || temp[2] > bound)  
 break  
}  
lowBound = (m – 1) + halfDropped >> bitDropped  
  
lowBoundSurface = ((Centroid[nodeIdx][dominantAxis] – minPos) + 128 >> 8) + halfDropped >> bitDropped  
highBOundSurface = ((maxPos – Centroid[nodeIdx][dominantAxis]) + 128 >> 8) + halfDropped >> bitDropped  
  
lowS = min(7, lowBoundSurface)  
highS = min(7, highBoundSurface)  
ctxIdxDriftSign = lowBound == highBound ? 0: 1 + (lowBound < highBound)

#### Determination of decoded points by the voxelization of TriSoup triangles

In this process, decoded points by the voxelization of TriSoup triangles are derived and added to PointPos as follows.

* Set PointCnt is equal to 0.
* Apply the following processes for each TriSoup nodes with nodeIdx
  + Initialize numDecPointsInNode as 0.
  + Point positions corresponding to TriSoup edge vertices are derived and added to decPoints[nodeIdx][] as defined in 9.4.3.3.1.
  + When numTriSoupVertices[nodeIdx] is greater than or equal to 3, or all the conditions listed in 9.4.3.2.8 are true, following processes are applied.
    - When numTriSoupVertices[nodeIdx] is greater than 3, add centroid to decoded points as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented.
    - Construct TriSoup triangles as defined in 9.4.3.3.2.
    - Derive point positions corresponding to each triangle and add the points to decode points as defined in 9.4.3.3.3.
  + Eliminate duplicate points in decPoints[nodeIdx][]. numDecPointsInNode is updated to the number of decoded points in the node nodeIdx after elimination of duplicated points. Because list of points decPoints[nodeIdx][] are disjoint by construction, elimination of duplicated points can be performed locally for each node nodeIdx individually independently on other nodes.
  + Add decoded points to pointPos.

for (i = 0; i < numDecPointsInNode; PointCnt++, i++){  
 for (k = 0; k < 3; k++){  
 pointPos[PointCnt][k] = decPoints[nodeIdx][numDecPointsInNode][k]  
 }  
}

##### Adding edge vertices to decoded points

The following processes applied for all vIdx (vIdx = 0,…,numTriSoupVertices[nodeIdx] - 1).

* Point position d\_point[k] is derived:

for (k = 0; k < 3; k++) {  
 d\_point[k] = vertex[nodeIdx][vIdx][k] + 128 >> 8  
}

* When all the following conditions are true, d\_point is added as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented.
* d\_point belongs to the node nodeIdx. The condition for the d\_point belongs to the node nodeIdx is, for all k (k=0, 1, 2), d\_point[k] is greater than or equal to TriSoupNodeLoc[nodeIdx][k] and lower than or equal to TriSoupNodeLoc[nodeIdx] [k] + TriSoupCubeSize – 1.
* bitDropped is greater than 0 or samplingValue is greater than 1.

If (pointWithinNode(d\_point) && (bitDropped > 0 || samplingValue > 1)){  
 for (k = 0; k < 3; k++) {  
 decPoints[nodeIdx][numDecPointsInNode][k] = d\_point[k]  
 }  
 numDecPointsInNode++  
}  
where  
 pointWithinNode(point) :=  
 point[0] >= nodeLoc[0]  
 && point[0] <= nodeLoc[0] + TrisoupCubeSize – 1  
 && point[1] >= nodeLoc[1]  
 && point[1] <= nodeLoc[1] + TrisoupCubeSize – 1  
 && point[2] >= nodeLoc[2]  
 && point[2] <= nodeLoc[2] + TrisoupCubeSize – 1  
 where  
 nodeLoc = TriSoupNodeLoc[nodeIdx]

##### Construct TriSoup Triangles

Construct TriSoup Triangles for each TriSoup node corresponding to nodeIdx as follows.

* The number of sorted vertices numSortedVertices is set as the number of vertices in sortedVertices[nodeIdx].
* Set numTriangles as follows.
  + numTriangles = 1 (if numSortedVertices is equal to 3),
  + numTriangles = numSortedVertices (otherwise).
* triangles[i][j][k] are derived as follows.

for (i = 0; i < numTriangles; i++){  
 for (k = 0; k < 3; k++){  
 triangles[i][0][k] = sortedVertices[nodeIdx][i][k]  
 }  
 i2 = i + 1  
 if (numTriSoupVertices[nodeIdx]==2 && i2 >= numTriSoupVertices[nodeIdx])  
 break  
 else if (i2 >= numTriSoupVertices[nodeIdx])  
 i2 -= numTriSoupVertices[nodeIdx]  
 for (k = 0; k < 3; k++){  
 triangles[i][1][k] = sortedVertices[nodeIdx][i2][k]  
 triangles[i][2][k] = numTriangles == 1 ?   
 sortedVertices[nodeIdx][i + 2][k] : Centroid[nodeIdx][k]  
 }  
}

##### Derive point positions corresponding to each triangle and add to decoded points

The following processes operated for each triangle with index i (=0,…,numTriangles).

In this process, *edge1*[], *edge2*[], *rayVector*[], *v0*[], preC[4], *h* and *a* shall be computed into 64 bits signed integer registers if letting the possibly of computing them into 64 bits signed integer registers.

* Derive edge vectors of a triangle.

edge1 = triangles[i][1] - triangles[i][0]  
edge2 = triangles[i][2] - triangles[i][0]

* Determine the two directions for ray tracing.

minVal = 1 << 28  
directionExcluded = 0  
h = CrossProduct(edge1, edge2) >> 8  
for (k = 0; k < 3; k++){  
 rayVector = {0, 0, 0}  
 rayVector[k] = 256  
 a = (rayVector × h) >> 8  
 if (abs(a) < minVal){  
 minVal = abs(a)  
 directionExcluded = k  
}

* minRange and maxRange are derived as follows.

for (k = 0; k < 3; k++){  
 minPos = min(triangles[i][0][k], min(triangles[i][1][k], triangles[i][2][k]))  
 minRange[k] = max(0, minPos + 128 >> 8)  
 maxPos = max(triangles[i][0][k], max(triangles[i][1][k], triangles[i][2][k]))  
 maxRange[k] = min(TriSoupNodeSize[nodeIdx][k], maxPos + 128 >> 8)  
}

* precompute preC[4] as follows.

v0 = triangles[i][0]  
preC[0] = edge1[1] × edge2[2] - edge1[2] × edge2[1]  
preC[1] = edge1[2] × edge2[0] - edge1[0] × edge2[2]  
preC[2] = edge1[0] × edge2[1] - edge1[1] × edge2[0]  
tmp = CrossProduct(v0, edge1) >> 8  
preC[3] = tmp[0] × edge2[0] + tmp[1] × edge2[1] + tmp[2] × edge2[2]

* ray tracing is applied to each direction except for the direction corresponding to directionExcluded. decPoints[nodeIdx][] and numDecPointsInNode are updated as described in 9.4.3.3.4.

##### Ray tracing

[Ed. (YZ) The added parameter shall be further checked to confirm whether need to be computed into 64 bits.]

In this process, decoded points are generated from a TriSoup triangle by ray tracing as follows.

In the processes defined in 9.4.3.3.4, 9.4.3.3.5, 9.4.3.3.6, and 9.4.3.3.7, rayOrigin[], c[4], a1, a2, a3, deltaT1, deltaT2, V0[2], V1[2], V2[2], S and P[2]shall be computed into 64 bits signed integer registers if letting the possibly of computing them into 64 bits signed integer registers.

* Initialize variables rayVector[3], h, a, startposG1, startposG2, endposG1, endposG2, and rayOrigin[3] as defined in 9.4.3.3.5.
* haloTriangle, haloThickness and haloTriangle2D are set as defined in 9.4.3.3.6.
* isVisible[129][129] is initialized as follows.

for (i = startposG1; i <= endposG1; i += samplingValue)  
 for (j = startposG2; j <= endposG2; j += samplingValue)  
 isVisible[i][j] = false

* A variable g1 is initialized as startposG1.
* When g1 is less than or equal to endposG1, the following processes applied.
  + rayOrigin is updated.

rayOrigin[g1pos[direction]] = g1 << 8

* + A variable g2 is initialized as startposG2.
  + When g2 is less than or equal to endposG2, the following processes applied.
    - rayOrigin is updated.

rayOrigin[g2pos[direction]] = g2 << 8

* + - P[2] is initialized.

P = {g1 << 8, g2 << 8};

* + - isVisible[g1][ g2] is updated.

isVisible[g1][g2] = isPointInTriangle(P, V0, V1, V2, haloTriangle2D) ? true : false  
where  
 isPointInTriangle(P, V0, V1, V2, haloTriangle2D) :=  
 ((P[0] - V0[0]) × (V1[1] - V0[1]) - (P[1] - V0[1]) × (V1[0] - V0[0])  
 >= haloTriangle2D)  
 && ((P[0] - V1[0]) × (V2[1] - V1[1]) - (P[1] - V1[1]) × (V2[0] - V1[0])  
 >= haloTriangle2D)  
 && ((P[0] - V2[0]) × (V0[1] - V2[1]) - (P[1] - V2[1]) × (V0[0] - V2[0])  
 >= haloTriangle2D)

* + - intersection[3] is initialized.

intersection[0] = rayOrigin[0]  
intersection[1] = rayOrigin[1]  
intersection[2] = rayOrigin[2]

* + - intersection is updated.

if ((g2 - samplingValue) >= startposG2 && isVisible[g1][g2 - samplingValue])  
 tBuffer[g1][g2] = tBuffer[g1][g2 - samplingValue] + deltaT2  
else if ((g1 - samplingValue) >= startposG1 && isVisible[g1 - samplingValue][g2])  
 tBuffer[g1][g2] = tBuffer[g1 - samplingValue][g2] + deltaT1  
else  
 tBuffer[g1][g2] = ((P[0] \* a1) >> kTriSoupFpBits)+ ((P[1] \* a2) >>   
 kTriSoupFpBits) + a3

intersection[direction] = intersection[direction] + tBuffer[g1][g2]

* + - When isVisible[g1][ g2] is true, decPoints[nodeIdx][] and numDecPointsInNode are updated as defined in 9.4.3.3.7.
    - g2 is updated to search the next point.

g2 = g2 + samplingValue

* + g1 is updated to search the next point.

g1 = g1 + samplingValue

##### Initialize ray tracing parameters

Parameters related to the ray tracing process are initialized as follows.

* Initialize rayVector[3].

rayVector[0] = 0  
rayVector[1] = 0  
rayVector[2] = 0  
rayVector[direction] = 1 << 8

* Calculate a cross product of rayVector and edge2 as h.

h = CrossProduct(rayVector, edge2) >> 8

* Calculate an inner product of edge1 and h as a.

a = InnerProduct(edge1, h) >> 8

* If a is greater than 256, the following processes are applied. Otherwise, the ray tracing process is completed for the input triangle and the direction.
* Starting positions startposG1 and startposG2, and ending positions endposG1 and endposG2 are derived.

g1pos[3] = { 1, 0, 0 }  
g2pos[3] = { 2, 2, 1 }  
startposG1 = minRange[g1pos[direction]]  
startposG2 = minRange[g2pos[direction]]  
endposG1 = maxRange[g1pos[direction]]  
endposG2 = maxRange[g2pos[direction]]

* rayOrigin[3] is initialized.

rayOrigin[0] = minRange[direction][0] << 8  
rayOrigin[1] = minRange[direction][1] << 8  
rayOrigin[2] = minRange[direction][2] << 8

##### Set halo parameters

Parameters related to halo are set as follows.

* haloTriangle is set.
  + If trisoup\_halo\_flag is equal to 0, haloTriangle is set as 0.
  + Otherwise, if samplingValue is strictly greater than 1, the value of haloTriangle is obtained depending on trisoup\_adaptive\_halo\_flag and samplingValue by

haloTriangle = trisoup\_adaptive\_halo\_flag? 50 \* samplingValue : 50  
haloTriangle = haloTriangle > 100 ? 100 : haloTriangle

* + Otherwise, if samplingValue is equal to 1, the value of haloTriangle is obtained depending on bitDropped by

haloBit = (((1 << bitDropped) - 1) << 8) / TriSoupCubeSize  
haloBit = (haloBit \* 24) / 32  
haloTriangle = haloBit > 40 ? 40 : haloBit

* haloThickness is set depending on samplingValue.

haloThickness = samplingValue > 1 ? 16 : 32

* c[4], a1, a2, a3, deltaT1 and deltaT2 are calculated as follows.

c[0] = preC[0] / a  
c[1] = preC[1] / a  
c[2] = preC[2] / a  
c[3] = preC[3] / a

a1 = c[g1pos[direction]]  
a2 = c[g2pos[direction]]  
a3 =((c[direction] × rayOrigin[direction]) >> 8) - c[3]

deltaT1 = a1 × samplingValue  
deltaT2 = a2 × samplingValue

* V0[2], V1[2], V2[2] are set.

V0 = {triangles[i][0][g1pos[direction]], triangles[i][0][g2pos[direction]]}  
V1 = {triangles[i][1][g1pos[direction]], triangles[i][1][g2pos[direction]]}  
V2 = {triangles[i][2][g1pos[direction]], triangles[i][2][g2pos[direction]]}

* A variable S is calculated as follows.

S = (V0[0] - V1[0]) × (V2[1] - V1[1]) - (V0[1] - V1[1]) × (V2[0] - V1[0])

* Determine whether to swap V0 and V2 according to the value of S as follows.

if (S < 0)   
swap(V0,V2)

* 2D halo parameter haloTriangle2D is calculated as follows.

haloTriangle2D = -(((abs(S) + 128) >> 8) × haloTriangle)

##### Point position reconstruction

Point positions based on triangles are reconstructed and added to the decoded point cloud as follows.

* Voxel position foundvoxel[3] is calculated as follows.

for (k = 0; k < 3; k++)  
 foundvoxel[k] = (TriSoupNodeLoc[nodeIdx][k] + intersection[k] + 128) >> 8

* Voxel position foundvoxelUp[3] is calculated as follows.

for (k = 0; k < 3; k++){  
 intersectionUp[k] = intersection[k]  
}  
intersectionUp[direction] += haloThickness  
for (k = 0; k < 3; k++){  
 foundvoxelUp[k] =   
 (TriSoupNodeLoc[nodeIdx][k] + intersectionUp[k] + 128) >> 8  
}

* Voxel position foundvoxelDown[3] is calculated as follows.

for (k = 0; k < 3; k++){  
 intersectionDown[k] = intersection[k]  
}  
intersectionDown[direction] -= haloThickness  
for (k = 0; k < 3; k++){  
 foundvoxelDown[k] =   
 (TriSoupNodeLoc[nodeIdx][k] + intersectionDown[k] + 128) >> 8  
}

* foundvoxel, foundvoxelUp and foundvoxelDown are added as decPoints[nodeIdx][numDecPointsInNode] and numDecPointsInNode is incremented respectively, under the condition they belong the TriSoup node nodeIdx. The condition for a point voxel belongs to the node nodeIdx is, for all k (k=0, 1, 2), voxel[k] is greater than or equal to TriSoupNodeLoc[nodeIdx] [k] and lower than or equal to TriSoupNodeLoc[nodeIdx][k] + TriSoupCubeSize – 1.

if(pointWithinNode(foundvoxel)){  
 for (k = 0; k < 3; k++){  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxel[k]  
 }  
 numDecPointsInNode++  
}  
if(pointWithinNode(foundvoxelUp)){  
 for (k = 0; k < 3; k++){  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxelUp[k]  
 }  
 numDecPointsInNode++  
}  
if(pointWithinNode(foundvoxelDown)){  
 for (k = 0; k < 3; k++){  
 decPoints[nodeIdx][numDecPointsInNode][k] = foundvoxelDown[k]  
 }  
 numDecPointsInNode++  
}  
where  
 pointWithintNode(point) :=  
 point[0] >= nodeLoc[0]  
 && point[0] <= nodeLoc[0] + TrisoupCubeSize – 1  
 && point[1] >= nodeLoc[1]  
 && point[1] <= nodeLoc[1] + TrisoupCubeSize – 1  
 && point[2] >= nodeLoc[2]  
 && point[2] <= nodeLoc[2] + TrisoupCubeSize – 1  
 where  
 nodeLoc = TriSoupNodeLoc[nodeIdx]