# Slice attributes

## General

Clause 10 specifies the reconstruction of a single slice attribute for the coded slice geometry. The reconstructed attribute values are stored in the array PointAttr.

## Point coordinates

### General

Attribute coding can use either the slice's reconstructed STV point positions or the points' scaled angular coordinates.

The expression AttrPos[ ptIdx ][ 𝑘 ] specifies the coordinates of each point for attribute coding:

* When attr\_coord\_conv\_enabled is 0 and attr\_inter\_prediction\_enabled is 0, AttrPos is equivalent to PointPos, which is the slice geometry in the slice’s coordinate system.
* When attr\_coord\_conv\_enabled is 0 and attr\_inter\_prediction\_enabled is 1, AttrPos is equivalent to the slice geometry translated to the coding coordinate system by the addition of the slice origin, SliceOrigin.
* Otherwise, AttrPos[ ptIdx ][ 𝑘 ] are angular point coordinates as specified by 10.2.2.

AttrPos[ptIdx][k] := attr\_coord\_conv\_enabled  
 ? AttrPosAng[ptIdx][k]  
 : (attr\_inter\_prediction\_enabled   
 ? PointPos[ptIdx][k] + SliceOrigin[k]  
 : PointPos[ptIdx][k])

### Conversion to scaled angular coordinates

The conversion is specified by the expression AttrPosAng[ ptIdx ][ 𝑘 ].

When geom\_tree\_type is equal to 1 and  slice\_attr\_inter\_prediction is equal to 1, the point’s angular coordinates shall be offset by the minimum value between the minimum angular coordinates of the current slice and previously applied offset value. The minimum angular coordinates of the current slice are specified by the expression *MinCurAng*[*k*]. The previously applied offset value is specified by the expression *MinRefAng*[*k*]. The minimum value between *MinRefAng* and *MinCurAng* is specified by the expression *MinAng*[*k*].

MinCurAng[k] := geom\_tree\_type == 1 && k == 1   
 ? −Exp2(ptree\_ang\_azimuth\_pi\_bits\_minus11 + 10) : 0

MinAng[k] := geom\_tree\_type == 1 && slice\_attr\_inter\_prediction == 1   
 ? Min(MinCurAng[k], MinRefAng[k]) : MinCurAng[k]

Otherwise, the point's angular coordinates shall be offset by the minimum angular coordinates.

The offset coordinates shall be scaled. Any negative coordinate after conversion shall be clipped to 0.

AttrPosAng[ptIdx][k] := DivExp2Up(relPos × attr\_coord\_conv\_scale[k], 8)  
 where  
 relPos := Max(0, PointAng[ptIdx][k] – MinAng[k])

It is a requirement of bitstream conformance that attr\_coord\_conv\_scale shall not cause any converted coordinate, AttrPosAng[ ptIdx ][ 𝑘 ], to be greater than Exp2( MaxSliceDimLog2 ) − 1.

When geom\_tree\_type is equal to 1, after the coordinate’s conversion of a slice, *MinRefAng* shall be set equal to *MinAng.*

## Syntax element semantics

#### Attribute data unit coefficients

The array AttrCoeff, with elements AttrCoeff[ coeffIdx ][ 𝑐 ], contains transform coefficient values. Elements of the array shall be initialized to zero.

zero\_run\_length\_prefix, zero\_run\_length\_minus3\_div2, zero\_run\_length\_minus3\_mod2 and zero\_run\_length\_minus11 together specify, in accordance with the expression ZeroRunLength, the number of consecutive transform coefficient tuples with all components equal to zero. Any of zero\_run\_length\_minus3\_div2, zero\_run\_length\_minus3\_mod2 and zero\_run\_length\_minus11 that are not present shall be inferred to be 0.

ZeroRunLength := zero\_run\_length\_prefix  
 + 2 × zero\_run\_length\_minus3\_div2 + zero\_run\_length\_minus3\_mod2  
 + zero\_run\_length\_minus11

#### Attribute coefficient tuples

Attribute coefficient values are signalled for a coeffIdx-th coefficient tuple when at least one component is not equal to zero.

coeff\_abs[ 𝑐 ] and coeff\_sign[ 𝑐 ] together specify the 𝑐-th transform coefficient component AttrCoeff[ coeffIdx ][ 𝑐 ]. coeff\_sign[ 𝑐 ] specifies whether (when 0) the coefficient's sign is positive or (when 1) negative. If coeff\_sign[ 𝑐 ] is not present, it shall be inferred to be 0.

The coefficients of the coeffIdx-th tuple are specified by the derivation of AttrCoeff:

for (c = 0; c < AttrDim; c++){  
 absVal = coeff\_abs[c]  
  
 if (c == AttrDim − 1)  
 if (AttrDim == 1  
 || AttrDim == 2 && coeff\_abs[0] == 0  
 || AttrDim == 3 && coeff\_abs[0] == 0 && coeff\_abs[1] == 0)  
 absVal++  
  
 AttrCoeff[coeffIdx][(c + 1) % AttrDim] = (1 − 2 × coeff\_sign[c]) × absVal  
}

When a point is eligible for direct prediction, the LSBs of coeff\_abs encode the direct predictor mode.

#### Raw attribute values

raw\_attr\_component\_length, when present, specifies the length in bytes of each syntax element raw\_attr\_value.

raw\_attr\_value[ ptIdx ][ 𝑐 ] specifies the attribute value for the 𝑐-th component of the ptIdx-th point in canonical decoding order. The length in bits of each syntax element is specified by the expression RawAttrValueBits.

RawAttrValueBits := raw\_attr\_width\_present  
 ? 8 × raw\_attr\_component\_length  
 : AttrBitDepth

## Raw attribute decoding

This subclause applies when attr\_coding\_type is 3.

Attribute values shall be set equal to the corresponding raw\_attr\_value syntax elements.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[ptIdx][c] = raw\_attr\_value[ptIdx][c]

## Attribute decoding using the region-adaptive hierarchical transform

### General

The region-adaptive hierarchical transform specified by 10.5 is a recursive two-point transform. It applies when attr\_coding\_type is 0.

The transform constructs a spatial tree of 3D transform blocks using the slice geometry (10.5.2). Basis vectors are calculated for each application of the transform, weighted in proportion to the significance of each coefficient.

When lossless\_coding\_enabled is 1, a transform domain prediction process predicts AC coefficients from the DC coefficients of certain adjoining blocks. When slice\_attr\_inter\_prediction is 1, a spatial tree of reference 3D transform blocks is constructed using the reference slice geometry (10.5.2.6). The transform domain inter prediction process predicts the DC coefficient of the transform block of the root node in the spatial tree from the DC coefficient of certain reference block. For layers other than the root, the transform domain inter prediction process modifies the prediction of AC coefficients from the AC coefficients of certain reference block.

When lossless\_coding\_enabled is 0, a sample domain prediction process predicts DC coefficients of the children nodes from the DC coefficients of certain adjoining blocks. When slice\_attr\_inter\_prediction is 1, a spatial tree of reference 3D transform blocks is constructed using the reference slice geometry (10.5.2.6). The sample domain inter prediction process predicts the DC coefficient of the root node in the spatial tree from the DC coefficient of certain reference block. For layers other than the root, the sample domain inter prediction predicts the DC coefficients of the children nodes.

The reconstruction process is specified for a single attribute component Cidx ∈ 0 .. AttrDim − 1. It is skipped in transform level Lvl when:

* Lvl is greater than 0 and the number of blocks in this level is equal to the number of blocks in the previous transform level Lvl + 1, or
* Lvl is equal to 0 and the number of blocks in this transform level is equal to the number of coded points.

The reconstruction process starts by:

* mapping coded coefficients to the transform tree (10.5.3) and
* scaling the coded coefficients (10.5.4).

Then in turn for each level, starting from the root of the transform tree (Lvl = RahtRootLvl) and proceeding down the tree until completing level 0, :

* performing transform domain prediction (10.5.5) when lossless\_coding\_enabled is 1 or performing sample domain prediction (10.5.6) when lossless\_coding\_enabled is 0,
* applying the inverse transform (10.5.7) and
* adding sample domain prediction (10.5.7) when lossless\_coding\_enabled is 0

The reconstructed attribute values are specified by 10.5.8.

### Transform tree

#### General

The tree of transform bocks is defined recursively:

* In tree level 0, each block groups together points with identical attribute coordinates.
* Each subsequent tree level 𝑙 shrinks the preceding level 𝑙 − 1 by a factor of two in each dimension; each 2×2×2 block groups together up to eight blocks from the preceding level.

An example tree is illustrated in Figure 18. The points a to f are grouped into blocks according to their attribute coordinates: C groups together c, d and e. The weight of each block is the number of points spanned by the block (10.5.2.3): C has a weight of 3; B with a weight of 5 groups together C, b and f; A with a weight of 6 groups together B and a.

图示

描述已自动生成

Key

|  |  |
| --- | --- |
| a to f | Points |
| A to C | Transform blocks |
| × | Transform coefficients at input to inverse transform for labelled block |
| ○ | Inverse transformed coefficient |
| ● | Inherited DC coefficient (See RahtDcCoeff) |
| 1 to 6 | Coefficient weights (See RahtCoeffWeightM) |

Figure 18 — Example RAHT tree, block weights and transform structure

#### State variables

The RAHT tree is specified in terms of the following state variables:

* The sparse array RahtCoeff of transform block coefficients; RahtCoeff[ lvl ][ bs ][ bt ][ bv ][ 𝑖 ] is the 𝑖-th coefficient for the block located at ( bs, bt, bv ) in transform level lvl. Unset elements shall be inferred to be 0.
* The array RahtBlkLoc of transform block locations; RahtBlkLoc[ lvl ][ nIdx ][ 𝑘 ] is the location of the nIdx-th coded block in transform level lvl.
* The array RahtBlkCnt of node counts per tree level; RahtBlkCnt[ lvl ] is the number of blocks in transform level lvl.
* The variable RahtLvlCnt, the number of transform levels.

#### Transform block weight

The weight of the DC transform coefficient for a block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkWeight[ lvl ][ bs ][ bt ][ bv ]. It is equal to the number of points that the coefficient applies to.

* 1. The sum of all block weights in any transform level is equal to the number of coded points (PointCnt).
  2. A block's weight is equal to the sum of its child block weights.

RahtBlkWeight[lvl][bs][bt][bv] :=  
 RahtBlkWeight = 0  
 for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 RahtBlkWeight += isPointInSubtree[ptIdx]  
 where  
 isPointInSubtree[ptIdx] :=  
 bs == AttrPos[ptIdx][0] >> lvl  
 && bt == AttrPos[ptIdx][1] >> lvl  
 && bv == AttrPos[ptIdx][2] >> lvl

#### Number of transform levels and per-level block order

The root node of the transform tree is the lowest block in the tree with a DC coefficient that spans the entire geometry; i.e. it has a weight equal to the number of coded points, PointCnt.

The tree level containing the root node is RahtRootLvl:

RahtRootLvl := RahtLvlCnt − 1

Within a transform level, blocks are ordered for coefficient coding by ascending Morton-coded block location, as specified by the derivation of RahtBlkLoc. Empty blocks are ignored.

for (RahtLvlCnt = 0; !done; RahtLvlCnt++)  
 for (mIdx = 0, nIdx = 0, wSum = 0; wSum < PointCnt; mIdx++) {  
 (bs, bt, bv) = FromMorton(mIdx)  
  
 wSum += RahtBlkWeight[RahtLvlCnt][bs][bt][bv]  
 if (RahtBlkWeight[RahtLvlCnt][bs][bt][bv] == 0)  
 continue  
  
 RahtBlkCnt[RahtLvlCnt]++  
 RahtBlkLoc[RahtLvlCnt][nIdx][0] = bs  
 RahtBlkLoc[RahtLvlCnt][nIdx][1] = bt  
 RahtBlkLoc[RahtLvlCnt][nIdx][2] = bv  
 nIdx++  
  
 done = RahtBlkWeight[RahtLvlCnt][bs][bt][bv] == PointCnt  
 }

#### 2×2×2 transform block coefficient weights

Transform coefficient weights are specified for each directional stage of the two-point transform and inverse transform for 2×2×2 transform blocks by the expression RahtCoeffWeightM[ lvl ][ stage ][ bs ][ bt ][ bv ][ 𝑚 ]; the parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl, lvl > 0;
* 𝑚 specifies the transform coefficient index in forward transform stage or inverse transform stage stage.

RahtCoeffWeightM[lvl][stage][bs][bt][bv][m] := RahtCoeffWeight[lvl][stage][s][t][v]  
 where  
 s := 2 × bs + FromMorton[m][0]  
 t := 2 × bt + FromMorton[m][1]  
 v := 2 × bv + FromMorton[m][2]

Within a block, coefficient weights are determined iteratively starting from its child block weights (stage 0) to the transform block coefficient weights of stage 3. At each stage, for each pair of inverse-transformed values 𝑎 and 𝑏, the weight for the DC (wL) and AC (wH) coefficient is the sum of the weights for a and b. If the weight for either 𝑎 or 𝑏 is 0, the AC coefficient weight is 0.

The expression RahtCoeffWeight[ lvl ][ stage ][ 𝑠 ][ 𝑡 ][ 𝑣 ] specifies the derivation of a weight in transform stage stage for the coefficient corresponding to the block located at ( 𝑠, 𝑡, 𝑣 ) in tree level lvl − 1.

RahtCoeffWeight[lvl][stage][s][t][v] :=  
 stage == 0 ? RahtBlkWeight[lvl − 1][s][t][v] :  
 stage == 1 ? v % 2 == 0 ? wL[0][0][1] : wH[0][0][−1] :  
 stage == 2 ? t % 2 == 0 ? wL[0][1][0] : wH[0][−1][0] :  
 stage == 3 ? s % 2 == 0 ? wL[1][0][0] : wH[−1][0][0] : na  
 where  
 wL[ds][dt][dv] := wSum[ds][dt][dv]  
 wH[ds][dt][dv] := wSum[ds][dt][dv] × wHnz[ds][dt][dv]  
 wSum[ds][dt][dv] := wP[s][t][v] + wP[s + ds][t + dt][v + dv]  
 wHnz[ds][dt][dv] := wP[s][t][v] × wP[s + ds][t + dt][v + dv] > 0  
 wP[s][t][v] := RahtCoeffWeight[lvl][stage − 1][s][t][v]

RahtBlkWeight[ lvl ][ 𝑠 ][ 𝑡 ][ 𝑣 ] ≡ RahtCoeffWeight[ lvl ][ 3 ][ 2 × 𝑠 ][ 2 × 𝑡 ][ 2 × 𝑣 ]; lvl > 0.

In the example of Figure 18, block B has stage 0 coefficient weights of 1, 3 and 1; stage 1 and 2 weights of 1, 4 and 1; and stage 3 weights of 5, 4 and 5. RahtCoeffWeight[ 1 ][ 1 ][ 3 ][ 0 ][ 2 ] would be 4.

#### Reference transform tree

The tree of reference transform blocks is defined recursively, applying the same process as the tree of transform blocks and using the reference slice geometry RefAttrPos:

* In tree level 0, each block groups together points with identical attribute coordinates.
* Each subsequent tree level 𝑙 shrinks the preceding level 𝑙 − 1 by a factor of two in each dimension; each 2×2×2 block groups together up to eight blocks from the preceding level.

The weight of the DC transform coefficient for a reference block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkWeightRef[ lvl ][ bs ][ bt ][ bv ]. It is equal to the number of points that the coefficient applies to, which is derived from the RefAttrPos in a similar way as RahtBlkWeight in 10.5.2.3.

The sum of attributes of points within each reference block is recorded when defining the tree of reference transform blocks. The sum of attributes of points within a reference block located at ( bs, bt, bv ) in transform level lvl is specified by the expression RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ], which is derived from *RefPointAttr*.

Transform coefficient weights are specified for each directional stage of the two-point transform for reference 2×2×2 transform blocks by the expression RahtCoeffWeightMRef[ lvl ][ stage ][ bs ][ bt ][ bv ][ 𝑚 ], which is derived from RahtBlkWeightRef in a similar way as RahtCoeffWeightM in 10.5.2.5; the parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl, lvl > 0;
* 𝑚 specifies the transform coefficient index in forward transform stage stage.

### Coefficient order

#### General

Subclause 10.5.3 specifies the correspondence between coded transform coefficients and the transform tree.

Starting from the root of the transform tree and proceeding in breadth-first order, coefficients are coded for each transform block; all transform blocks within one tree level are coded before those of the next level. Within a tree level, blocks shall be traversed in ascending Morton order of block location.

The order of coefficients within a transform block is specified by 10.5.3.2 for 2×2×2 blocks (tree levels greater than 0) and 10.5.3.3 for blocks of co-located points (tree level 0).

The mapping from the coded order to the transform tree is specified in terms of the following variables:

* Lvl, the index of the mapped transform level.
* CoeffIdx, the index into the decoded coefficient array AttrCoeff for the next mapped coefficient.

CoeffIdx = 0  
for (Lvl = RahtLvlCnt; Lvl ≥ 0; Lvl−−) {  
 if (Lvl > 0)  
 if (RahtBlkCnt[Lvl] != RahtBlkCnt[Lvl-1]) {  
 … /\* See 10.5.3.2 \*/  
 }   
 else   
 if (RahtBlkCnt[Lvl]!= PointCnt) {  
 … /\* See 10.5.3.3 \*/  
 }  
}

#### Mapping for a tree level of 2×2×2 transform blocks

This subclause applies to tree levels greater than 0.

For each 2×2×2 transform block, up to 7 AC coefficients are mapped from the bitstream to coefficient indexes within the block. In the case of the root transform block, the DC coefficient is additionally mapped.

Table 26 specifies the order in which transform block coefficients are coded; RahtCoeffOrder[ 𝑖 ] is the block index of the 𝑖-th coded coefficient.

Only coefficients with a non-zero transform coefficient weight are coded.

Table 38 — 2×2×2 RAHT coefficient coding order

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑖 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| RahtCoeffOrder[ 𝑖 ] | 0 | 4 | 2 | 1 | 6 | 5 | 3 | 7 |

for (nIdx = 0; nIdx < RahtBlkCnt[Lvl]; nIdx++) {  
 bs = RahtBlkLoc[Lvl][nIdx][0]  
 bt = RahtBlkLoc[Lvl][nIdx][1]  
 bv = RahtBlkLoc[Lvl][nIdx][2]  
  
 for (i = 0; i < 8; i++) {  
 /\* skip the DC coefficient that will be inherited \*/  
 if (i == 0 && Lvl < RahtRootLvl)  
 continue  
  
 if (RahtCoeffWeightM[Lvl][3][bs][bt][bv][RahtCoeffOrder[i]] > 0)  
 RahtCoeff[Lvl][bs][bt][bv][RahtCoeffOrder[i]] = AttrCoeff[CoeffIdx++][Cidx]  
 }  
}

#### Mapping for co-located points

This subclause applies to the final tree level (Lvl == 0), after all other tree levels have been mapped.

Each transform block with a node weight 𝑤 greater than 1 codes 𝑤 − 1 AC coefficients with the same attribute coordinates.

for (nIdx = 0; nIdx < RahtBlkCnt[0]; nIdx++) {  
 ns = RahtBlkLoc[0][nIdx][0]  
 nt = RahtBlkLoc[0][nIdx][1]  
 nv = RahtBlkLoc[0][nIdx][2]  
  
 for (i = 1; i < RahtBlkWeight[0][ns][nt][nv]; i++)  
 RahtCoeff[0][ns][nt][nv][i] = AttrCoeff[CoeffIdx++][Cidx]  
}

### Coefficient scaling

#### General

Subclause 10.5.4 specifies the scaling of coded coefficients for a block located at ( Bs, Bt, Bv ) in tree level Lvl. It shall be applied to every block in every tree level in any order.

If a regional QP offset is present (i.e. attr\_region\_cnt > 0), a tree (10.5.4.4) is specified that blends QP offsets along region boundaries according to the structure of the RAHT tree.

#### For a transform block

Within a transform block, coded coefficients shall be scaled according to a per-coefficient QP. The DC coefficient is not scaled except when coded in the root node of the transform tree.

mCnt := Lvl > 0 ? 8 : RahtBlkWeight[0][Bs][Bt][Bv]  
for (m = 0; m < mCnt; m++) {  
 /\* skip the DC coefficient that will be inherited \*/  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] = RahtCoeffScaled[Lvl][Bs][Bt][Bv][m]  
}

The scaling of the 𝑚-th coded coefficient of a transform block located at ( bs, bt, bv ) in tree level lvl is specified by the expression RahtCoeffScaled[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ]: it is scaled by the fixed-point step size AttrQstep[ qp ] (10.7.4) and represented as a 15 fractional-bit, fixed-point coefficient value.

RahtCoeffScaled[lvl][bs][bt][bv][m] := coeff × AttrQstep[qp] << 7  
 where  
 coeff := RahtCoeff[lvl][bs][bt][bv][m]  
 qp := lossless\_coding\_enabled == 0 ? RahtCoeffQp[lvl][bs][bt][bv][m] : 4

[Ed. (YZ): RahtCoeffQp shall be derived as 4 when perform lossless compression, so the constraint that limit qp to 4 here may be removed.]

#### Per coefficient QP

The expression RahtCoeffQp[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ] specifies the QP for the 𝑚-th coefficient of the transform block located at ( bs, bt, bv ) in tree level lvl for the Cidx-th attribute component:

* rgnOffset[ qc ] is the per-coefficient offset from the region-dependent QP offset tree.
* dpth is the depth of the transform block in the RAHT tree.
* ACqpoffset is the per-coefficient AC coefficient QP offset.

RahtCoeffQp[lvl][bs][bt][bv][m] := AttrQp[dpth][rgnOffset][Cidx > 0] + ACqpoffset  
 where  
 dpth := RahtRootLvl − lvl  
 isACoffset = dpth <= attr\_AC\_qp\_layer\_cnt\_minus1 && m > 0  
 ACqpoffsetP = isACoffset ? attr\_AC\_qp\_offset[ dpth ][0][m – 1] : 0  
 ACqpoffsetS = isACoffset ? attr\_AC\_qp\_offset[ dpth ][1][m – 1] : 0  
 Acqpoffset := Cidx > 0 ? ACqpoffsetS : ACqpoffsetP  
 rgnOffset[qc] := RahtTreeQpOffsetM[lvl][bs][bt][bv][m][qc]

#### Region-dependent QP offset tree

The integer, averaged region-dependent QP offset for each coefficient of a 2×2×2 transform block is specified by the expression RahtTreeQpOffsetM[ lvl ][ bs ][ bt ][ bv ][ 𝑚 ]. The parameter(s):

* bs, bt and bv specify a transform block location in tree level lvl;
* 𝑚 specifies the transform coefficient index from the final forward transform stage.

RahtTreeQpOffsetM[lvl][bs][bt][bv][m] :=  
 lvl == 0 ? RahtTreeQpOffset[ 0][3][bs][bt][bv][qc] >> 4  
 : RahtTreeQpOffset[lvl][3][ms][mt][mv][qc] >> 4  
 where  
 ms := 2 × bs + FromMorton[m][0]  
 mt := 2 × bt + FromMorton[m][1]  
 mv := 2 × bv + FromMorton[m][2]

The fixed-point, region-dependent QP offset tree is structurally identical to the transform tree. It is specified recursively for a QP component qc by RahtTreeQpOffset[ lvl ][ 𝑠 ][ 𝑡 ][ 𝑣 ][ qc ]:

* For tree level 0, the offset is the regional QP offset for a point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ).
* For each subsequent tree level 𝑙, a 2×2×2 block of QP offsets is averaged for each transform stage in turn. Each QP offset in a block at stage 0 is that of the DC transform block coefficient for a child block in the preceding level 𝑙 − 1.

Within a block, each subsequent transform stage averages, along the transformed axis, adjacent pairs of QP offsets from the preceding stage that have a non-zero transform coefficient weight. For a pair of QPs 𝑎 and 𝑏 with respective weights wa and wb:

* The weight for the corresponding DC coefficient is 𝑎 + 𝑏 divided by 2 if wa and wb are non-zero, or 1 otherwise.
* The weight for the corresponding AC coefficient is 𝑏 if wa and wb are non-zero, or 0 otherwise.

Averages shall be calculated using four fractional bits.

RahtTreeQpOffset[lvl][stage][s][t][v][qc] :=  
 lvl == 0 ? RahtRegionQpOffset[qs][qt][qv][qc] << 4 :  
 stage == 0 ? RahtTreeQpOffset[lvl − 1][3][2 × s][2 × t][2 × v][qc] :  
 stage == 1 ? v % 2 == 0 ? qpL[0][0][1] : qpH[0][0][−1] :  
 stage == 2 ? t % 2 == 0 ? qpL[0][1][0] : qpH[0][−1][0] :  
 stage == 3 ? s % 2 == 0 ? qpL[1][0][0] : qpH[−1][0][0] : na  
 where  
 qpL[ds][dt][dv] := qpSum[ds][dt][dv] >> qpHnz[ds][dt][dv]  
 qpH[ds][dt][dv] := qpC[s][t][v] × qpHnz[ds][dt][dv]  
 qpSum[ds][dt][dv] := qpC[s][t][v] + qpC[s + ds][t + dt][v + dv]  
 qpHnz[ds][dt][dv] := wC[s][t][v] × wC[s + ds][t + dt][v + dv] > 0  
 qpC[s][t][v] := RahtTreeQpOffset[lvl][stage − 1][s][t][v][qc]  
 wC[s][t][v] := RahtCoeffWeight[lvl][stage − 1][s][t][v]

An example tree is illustrated in Figure 19 for the transform tree of Figure 18. The hatched area has a regional QP offset of +6; co-located points c, d and e have an offset of +6; points b and f, 0. In block A, the stage 3 QPs are used to scale the transform coefficients. For the coefficient at 𝑚 = 0, 3¾, the QP is the mean of the stage 2 QPs 1½ and 6; for the coefficient at 𝑚 = 4, it is 1½. Scaling of coefficient uses the integer part of the fractional QP.

图示, 工程绘图

描述已自动生成

Key

|  |  |
| --- | --- |
| b to f | Points (See Figure 18) |
| A to C | Transform blocks (See Figure 18) |
| × | Transform coefficients at input to inverse transform for labelled block |
| ○ | Inverse transformed coefficient |
| ● | Inherited DC coefficient (See RahtDcCoeff) |
| 0 to 6 | QP values (See RahtTreeQpOffset) |

Figure 19 — Example region-dependent QP offset tree

### Transform domain prediction

#### General

Subclauses 10.5.5.2 - 10.5.5.6 apply when raht\_prediction\_enabled\_flag is 1 and lossless\_coding\_enabled is 1, subclauses 10.5.5.7 - 10.5.5.9 apply when slice\_attr\_inter\_prediction is 1 and lossless\_coding\_enabled is 1. These specify the transform domain prediction for a block located at ( Bs, Bt, Bv ) in tree level Lvl. These shall be performed for every eligible block for intra prediction (10.5.5.2) or inter prediction (10.5.5.7) in the tree level, in any order, by:

* generating a prediction block (10.5.5.3) and/or;
* generating a inter prediction block and resampling the inter prediction block (10.5.5.8);
* applying the forward transform for the intra prediction block (10.5.5.5) and/or;
* applying the forward transform for the inter prediction block (10.5.5.9);
* modifying the prediction block from the inter prediction block;
* adding the resulting AC transform coefficients to the coefficient residuals in the coefficient tree.

The prediction block and its transform are specified in terms of the eight-element array RahtPredBlk; RahtPredBlk[ 𝑚 ] is the prediction block value for the Morton-coded location 𝑚. The inter prediction block and its transform are specified in terms of the eight-element array RahtInterPredBlk; RahtInterPredBlk[ 𝑚 ] is the inter prediction block value for the Morton-coded location 𝑚.

if (RahtPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtPredBlk[m] = (RahtPredW[m] >> 15} << 15  
  
 … /\* in−place, forward transform of RahtPredBlk (10.5.5.5) \*/  
  
}   
if (RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtInterPredBlk[m] = RahtInterPredW[m]  
  
 … /\* in−place, forward transform of RahtInterPredBlk （10.5.5.9） \*/  
  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtPredBlk[m] = RahtInterPredBlk[m]  
 }  
}  
  
if (RahtPredEligible[Lvl][Bs][Bt][Bv] || RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] += RahtPredBlk[m]  
 }  
}

#### Eligibility

When enabled, transform domain prediction shall be performed for 2×2×2 transform blocks unless the block:

* is the root of the transform tree; or
* has only one non-empty child block; or
* is adjoined (10.5.5.3) by fewer than raht\_prediction\_samples\_min non-empty blocks; or
* has an ancestor, except the root block, that is adjoined by fewer than raht\_prediction\_subtree\_min non-empty blocks.

The expression RahtBlkChildCnt[ lvl ][ bs ][ bt ][ bv ] is the number of non-empty child blocks of the block located at ( bs, bt, bv ) in tree level lvl .

RahtBlkChildCnt[lvl][bs][bt][bv] :=  
 RahtBlkChildCnt = 0  
 for(m = 0; m ≤ 8; m++){  
 if(RahtCoeffWeightM[lvl][0][bs][bt][bv][m] > 0)  
 RahtBlkChildCnt ++   
 }

The expression RahtPredEligible[ lvl ][ bs ][ bt ][ bv ] specifies whether the transform block located at ( bs, bt, bv ) in tree level lvl is eligible.

RahtPredEligible[lvl][bs][bt][bv] := raht\_prediction\_enabled\_flag  
 && lvl > 0  
 && lvl < RahtRootLvl  
 && !DisabledBySendingMode  
 && RahtBlkChildCnt[lvl][bs][bt][bv] > 1  
 && RahtNeighCnt[lvl][bs][bt][bv] ≥ raht\_prediction\_samples\_min  
 && RahtNeighCntMinAncestor[lvl][bs][bt][bv] ≥ raht\_prediction\_subtree\_min  
 where  
 DisabledBySendingMode := (raht\_intra\_layer\_code\_enabled && depth < layer\_code\_depth && depth ≥ 0) ? !slice\_raht\_intra\_layer\_code\_mode[depth] : 0  
 depth := RahtRootLevel – lvl – 1

#### Generation of prediction block from adjoining blocks

A prediction block is generated from up to 19 transform blocks that contain a DC coefficient: the co-located block and those that adjoin the predicted block by a face or an edge.

The expression RahtNeighCnt[ lvl ][ bs ][ bt ][ bv ] is the number of non-empty blocks that can be used to predict the block located at ( bs, bt, bv ), which is defined by (RahtBlkLoc[ lvl ][ nIdx ][ 0 ], RahtBlkLoc[ lvl ][ nIdx ][ 1 ], RahtBlkLoc[ lvl ][ nIdx ][ 2]), is the location of the nIdx-th coded block in tree level lvl. When the block located at ( bs, bt, bv ) in tree level lvl has only one non-empty child block*, the* number of non-empty blocks that can be used to predict that block is set as 19*.*

RahtNeighCnt[lvl][bs][bt][bv] :=   
 RahtBlkChildCnt[lvl][bs][bt][bv] == 1 ? 19 : SumN19[neighWeightGt0]  
 where  
 neighWeightGt0[ds][dt][dv] := RahtBlkWeight[lvl][bs + ds][bt + dt][bv + dv] > 0 &&  
 ¬isNeighOutsideSearchRange  
 isNeighOutsideSearchRange :=  
 Morton(bs + ds, bt + dt, bv + dv) < Morton(bs, bt, bv) ?  
 Morton(bs + ds, bt + dt, bv + dv) < Morton(LE(0), LE(1), LE(2)) :  
 Morton(bs + ds, bt + dt, bv + dv) > Morton(BE(0), BE(1), BE(2))  
 sr := raht\_prediction\_search\_range  
 LE(k) := RahtBlkLoc[lvl][Max(nIdx - sr, 0)][k]  
 BE(k) := RahtBlkLoc[lvl][Min(nIdx + sr, RahtBlkCnt[lvl] - 1)][k]

The expression RahtNeighCntMinAncestor[ lvl ][ bs ][ bt ][ bv ] is lowest value of RahtNeighCnt for any ancestor of the block located at ( bs, bt, bv ) in tree level lvl. In determining eligibility, the root node shall be considered to have 19 adjoining blocks.

RahtNeighCntMinAncestor[lvl][bs][bt][bv] :=  
 lvl >= RahtRootLvl − 1 ? 19 : Min(neighCntP, minAncestorCnt)  
 where  
 neighCntP := RahtNeighCnt[lvl + 1][bs / 2][bt / 2][bv / 2]  
 minAncestorCnt := RahtNeighCntMinAncestor[lvl + 1][bs / 2][bt / 2][bv / 2]

The expression SumN19[ expr ] sums the result of applying expr to the relative tree location of each of the 19 possible adjacent blocks.

SumN19[expr] :=  
 SumN19 = 0  
 for (ds = −1; ds ≤ 1; ds++)  
 for (dt = −1; dt ≤ 1; dt++)  
 for (dv = −1; dv ≤ 1; dv++)  
 if (Abs(ds) + Abs(dt) + Abs(dv) < 3)  
 SumN19 += expr[ds][dt][dv]

#### Upsampling

##### Normalized DC values

The samples used to generate an upsampled prediction block are transform-block DC coefficients (10.5.7.2) normalized by their weight as specified by RahtDcNorm, that is a 15 fractional-bit, fixed-point value; RahtDcNorm[ lvl ][ bs ][ bt ][ bv ] is the sample value for the block located at ( bs, bt, bv ) in tree level lvl.

RahtDcNorm[lvl][bs][bt][bv] := raht\_buffer\_extension\_flag == 0 ?   
(lossless\_coding\_enabled == 0 ? DivExp2Fz((coeff >> wShift) × (IntRecipSqrt(w) >> 25 − wShift), 30) : DivExp2Fz(coeff, 15)) :   
(lossless\_coding\_enabled == 0 ? DivExp2Fz((coeff >> wShift) × (IntRecipSqrt(w) >> 25 − wShift), 15) : DivExp2Fz(coeff, 0))  
 where  
 w := RahtBlkWeight[lvl][bs][bt][bv]  
 coeff := RahtDcCoeff[lvl][bs][bt][bv]  
 wShift := w > 1024 ? IntLog2(w − 1) >> 1 : 0

##### Exclusion of adjoining blocks

Adjoining blocks shall be excluded from the upsampling process if either their weight is zero or the normalized DC value for their primary attribute component is:

* less than or equal to 0,2 times that of co-located block; or
* greater than or equal to 2,5 times that of the co-located block.

The expression RahtPredExcluded[ ds ][ dt ][ dv ] specifies whether the block with relative location ( ds, dt, dv ) is excluded from contributing to the upsampled prediction of the block ( Bs, Bt, Bv ).

RahtPredExcluded[ds][dt][dv] :=  
 Cidx == 0 ? empty || sample ≤ limitMin || sample ≥ limitMax  
 : … /\* Value of RahtPredExcluded[ds][dt][dv] for Cidx == 0 \*/  
 where  
 empty := RahtBlkWeight[Lvl][Bs][Bt][Bv] == 0  
 sample := 10 × RahtSample[Lvl][Bs + ds][Bt + dt][Bv + dv]  
 limitMin := 2 × RahtSample[Lvl][Bs][Bt][Bv]  
 limitMax := 25 × RahtSample[Lvl][Bs][Bt][Bv]

##### Eligibility of replacing adjoining blocks with child blocks

Adjoining block shall be replaced by its one child block to calculate the sample value of the prediction block for the Morton-coded sample location *m* in the block ( Bs, Bt, Bv ) when the eligibility conditions are meet. The child block to replace the adjoining block is the child block with relative location ( dsc, dtc, dvc ) to the 𝑚-th child block in tree level lvl - 1. The replacement shall be performed when the following eligibility conditions are true:

* raht\_subnode\_prediction\_enabled is 1;
* The Morton code of the Adjoining block is less than the Morton code of the block;
* The Adjoining block adjoin the block by a face and it has a non-empty child block that adjoin the 𝑚-th child block of the block by a face; or the Adjoining block adjoin the block by an edge and it has a non-empty child block that adjoin the 𝑚-th child block of the block by an edge.

The expression *IsReplaced*[ ds ][ dt ][dv][ 𝑚 ] specifies whether the adjoining block with relative location ( ds, dt, dv ) is replaced by its child block to calculate the sample of the prediction block for the 𝑚-th child block of the block ( Bs, Bt, Bv ).

IsReplaced[ds][dt][dv][m] :=   
 isAdj && isEarly && (RahtBlkWeight[lvl-1][bsc + dsc][btc + dtc][bvc + dvc]>0?)  
 where  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)  
 isAdj := (m & adjMask) == adjLoc  
 isEarly := Morton(Bs + ds, Bt + dt, Bv + dv) < Morton(Bs, Bt, Bv)  
 bsc := 2 × Bs + FromMorton[m][0]  
 btc := 2 × Bt + FromMorton[m][1]  
 bvc := 2 × Bv + FromMorton[m][2]  
 dsc := ds  
 dtc := dt  
 dvc := dv

##### Upsampled prediction block

The samples of the 2x2x2 prediction block are the weighted averages of the adjoining blocks' normalized DC values when raht\_subnode\_prediction\_enabled is 0. For each sample, the weight for the adjoining block depends upon the relative positions of the block and the sample location.

The expression RahtPred[ 𝑚 ], that is a 15 fractional-bit, fixed-point value, specifies the value for the Morton-coded sample location m; where:

* SumN19[ 𝑤 ] is sum of the weight of each adjoining block;
* SumN19[ wNeigh ] is the 15 fractional-bit, fixed-point the sum of the weighted normalized DC values for each adjoining block;
* the 15 fractional-bit, fixed-point reciprocal of the sum of weights, RahtPredRecipW[ 𝑥 ] is specified by Table 27.

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

RahtPred[m] := (raht\_buffer\_extension\_flag == 0 ? SumN19[wNeigh] << 15 : SumN19[wNeigh]) × RahtPredRecipW[SumN19[w]]  
 where  
 w[ds][dt][dv] := RahtPredExcluded[ds][dt][dv] && Abs(Morton(bs + ds, bt + dt, bv +   
 dv)- Morton(bs, bt, bv)) <=raht\_prediction\_search\_range ? 0 :  
 RahtPredWeight[ds][dt][dv][m]  
 wNeigh[ds][dt][dv] := w[ds][dt][dv] × RahtDcNorm[Lvl][Bs + ds][Bs + dt][Bs + dv]

The expression RahtPredWeight[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be 4 for the co-located block, 2 for blocks that adjoin by a face and 1 for blocks that adjoin by only an edge.

RahtPredWeight[ds][dt][dv][m] := (m & adjMask) == adjLoc ? weight : 0  
 where  
 weight := 4 >> (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

Table 39 — Values of RahtPredRecipW[ 𝑥 ]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑥 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| RahtPredRecipW[ 𝑥 ] | 8 192 | 6 554 | 5 461 | 4 681 | 4 096 | 3 641 | 3 277 | 2 979 | 2 731 | 2 521 |

##### Replacement of adjoining blocks with child blocks

Replacement of adjoining blocks with their child blocks shall be performed when raht\_subnode\_prediction\_enabled is 1. The child blocks’ normalized DC values and weights are to replace the adjoining blocks’ normalized DC values and weights when the eligibility conditions are meet.

RahtPred[m] := (raht\_buffer\_extension\_flag == 0 ? SumN19[wNeigh] << 15 : SumN19[wNeigh]) × RahtPredRecipWC[SumN19[w]-1]  
 where  
 s := 2 × Bs + FromMorton[m][0]  
 t := 2 × Bt + FromMorton[m][1]  
 v := 2 × Bv + FromMorton[m][2]  
 RahtPredRealWeight[ds][dt][dv][m] :=   
 IsReplaced [ds][dt][dv][m] ? RahtPredWeightC[ds][dt][dv][m]   
 : RahtPredWeightB[ds][dt][dv][m]  
 w[ds][dt][dv] := RahtPredExcluded[ds][dt][dv] ? 0 : RahtPredRealWeight[ds][dt][dv][m]  
 RahtDcN := IsReplaced [ds][dt][dv][m] ? RahtDcNorm[lvl-1][s + ds][t + dt][v + dv]   
 : RahtDcNorm[Lvl][Bs + ds][Bt + dt][Bv + dv]  
 wNeigh[ds][dt][dv] := w[ds][dt][dv] × RahtDcN

The expression *RahtPredWeightB*[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the not replaced block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be specified by raht\_prediction\_weights[0] for the co-located block, raht\_prediction\_weights[1] for blocks that adjoin by a face and raht\_prediction\_weights[2] for blocks that adjoin by only an edge.

RahtPredWeightB[ds][dt][dv][m] := (m & adjMask) == adjLoc ? weight : 0  
 where  
 idx := (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 weight := raht\_prediction\_weights[idx]  
 adjMask := Morton(ds ≠ 0, dt ≠ 0, dv ≠ 0)  
 adjLoc := Morton(ds > 0, dt > 0, dv > 0)

The expression *RahtPredWeightC*[ ds ][ dt ][ dv ][ 𝑚 ] is the weight to be applied to the normalized DC value of the child block of the replaced block with relative tree location ( ds, dt, dv ) for the Morton-coded prediction block sample location 𝑚. The weight shall be specified by raht\_prediction\_weights[3] for child blocks that adjoin by a face and raht\_prediction\_weights[4] for child blocks that adjoin by only an edge.

RahtPredWeightC[ds][dt][dv][m] := IsReplaced[ds][dt][dv][m] ? weight : 0  
 where  
 idx := (ds ≠ 0) + (dt ≠ 0) + (dv ≠ 0)  
 weight := raht\_prediction\_weights[2+idx]

The 15 fractional-bit, fixed-point reciprocal of the sum of weights, RahtPredRecipWC[ 𝑥 ] is specified by Table 28.

Table 40 — Values of RahtPredRecipWC[ 𝑥 ]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑥 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| RahtPredRecipWC[ 𝑥 ] | 32768 | 16384 | 10923 | 8192 | 6554 | 5461 | 4681 | 4096 | 3641 | 3277 |
| 𝑥 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| RahtPredRecipWC[ 𝑥 ] | 2979 | 2731 | 2521 | 2341 | 2185 | 2048 | 1928 | 1820 | 1725 | 1638 |
| 𝑥 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| RahtPredRecipWC[ 𝑥 ] | 1560 | 1489 | 1425 | 1365 | 1311 | 1260 | 1214 | 1170 | 1130 | 1092 |
| 𝑥 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| RahtPredRecipWC[ 𝑥 ] | 1057 | 1024 | 993 | 964 | 936 | 910 | 886 | 862 | 840 | 819 |
| 𝑥 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| RahtPredRecipWC[ 𝑥 ] | 799 | 780 | 762 | 745 | 728 | 712 | 697 | 683 | 669 | 655 |
| 𝑥 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| RahtPredRecipWC[ 𝑥 ] | 643 | 630 | 618 | 607 | 596 | 585 | 575 | 565 | 555 | 546 |
| 𝑥 | 60 | 61 | 62 | 63 |  |  |  |  |  |  |
| RahtPredRecipWC[ 𝑥 ] | 537 | 529 | 520 | 512 |  |  |  |  |  |  |

#### Forward transform for a 2×2×2 block prediction block

The forward transform for a 2×2×2 prediction block comprises transforming pairs of coefficients along each axis.

First, along the V axis:

rahtFwd1D[2][0][1]  
rahtFwd1D[2][2][3]  
rahtFwd1D[2][4][5]  
rahtFwd1D[2][6][7]

Second, along the T axis:

rahtFwd1D[1][0][2]  
rahtFwd1D[1][1][3]  
rahtFwd1D[1][4][6]  
rahtFwd1D[1][5][7]

Third, along the S axis:

rahtFwd1D[0][0][4]  
rahtFwd1D[0][1][5]  
rahtFwd1D[0][2][6]  
rahtFwd1D[0][3][7]

The expression rahtFwd1D[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place, forward, two-point transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis.

rahtFwd1D[k][aIdx][bIdx] := RahtFwd(aCoeff, bCoeff, wa, wb)  
 where  
 aCoeff := RahtPredBlk[aIdx]  
 bCoeff := RahtPredBlk[bIdx]  
 wa := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wb := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

#### Forward two-point transform

This subclause specifies the in-place, forward, two-point transform RahtFwd( aCoeff, bCoeff, wa, wb ). Its parameters are:

* the expressions aCoeff and bCoeff that identify the coefficients to be transformed in-place;
* the weights wa and wb that are the coefficient weights for aCoeff and bCoeff, respectively.

The specification of the forward two-point transform applies only to prediction blocks for the reconstruction of point attributes.

The transform basis vectors 15-fractional-bit, fixed-point coefficients 𝑎 and 𝑏:

a := IntSqrt(wa << 30) × IntRecipSqrt(wa + wb) >> 40  
b := IntSqrt(wb << 30) × IntRecipSqrt(wa + wb) >> 40

If both wa or wb are 0, the transform result is:

if (wa == 0 && wb == 0)  
 yl = yh = 0

If either wa or wb is 0, the transform result is:

if (wa == 0 || wb == 0) {  
 yl = wa ≠ 0 ? aCoeff : bCoeff  
 yh = 0  
}

If either wa or wb is zero, the respective opposing coefficient 𝑏 or 𝑎 is not necessarily .

Otherwise (both wa and wb are greater than 0), the transform result is:

if (wa ≠ 0 && wb ≠ 0) {  
 yh = bCoeff − aCoeff  
 yl = aCoeff + ((yh >> 16) << 15)  
}

The transform result replaces the input coefficients:

aCoeff = yl  
bCoeff = yh

#### Eligibility for inter prediction

When inter prediction is enabled, inter prediction shall be performed for 2×2×2 transform blocks unless:

* inter prediction is disabled for current level; or
* the block has no reference block which share the same location in the reference transform tree.

The expression RahtInterPredEligible[ lvl ][ bs ][ bt ][ bv ] specifies whether the transform block located at ( bs, bt, bv ) in tree level lvl is eligible for inter prediction.

RahtPredEligible[lvl][bs][bt][bv] := slice\_attr\_inter\_prediction  
 && lvl > 0  
 && lvl > (RahtRootLvl – raht\_inter\_layer\_depth\_minus1 - 1)  
 && !DisabledBySendingMode  
 && RahtBlkWeightRef[lvl][bs][bt][bv] > 0  
 where  
 DisabledBySendingMode := (raht\_inter\_layer\_code\_enabled && depth < layer\_code\_depth && depth ≥ 0) ? !slice\_raht\_inter\_layer\_code\_mode[depth] : 0  
 depth := RahtRootLevel – lvl – 1

#### Generation of inter prediction block

The samples used to generate an inter prediction block are the sum of point attributes within each child block of the reference block as specified by RahtInterPred; RahtInterPred[ lvl  - 1][ bsc ][ btc ][ bvc ] is the sample value for the child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

RahtInterPred[lvl][bs][bt][bv] := RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ]

Each sample in the prediction block is specified by RahtInterPredW[ m ] for the m-th child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

* If one child block is with zero weight, the sample used to generate the inter prediction block shall be set to 0;
* Else, if the corresponding child block of the reference block is with zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the reference block;
* Else, the corresponding child block of the reference block is with non-zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the corresponding child block of the reference block.

RahtInterPredW[ m ] := RahtBlkWeight[lvl - 1][bsc][btc][bvc] ?   
(RahtBlkWeightRef[lvl - 1][bsc][btc][bvc] ? RahtInterPred[lvl – 1][bsc][btc][bvc] : RahtInterPred[lvl][bs][bt][bv]) : 0

#### Forward transform for a 2×2×2 block inter prediction block

The forward transform for a reference 2×2×2 prediction block comprises transforming pairs of coefficients along each axis.

First, along the V axis:

rahtFwd1D[2][0][1]  
rahtFwd1D[2][2][3]  
rahtFwd1D[2][4][5]  
rahtFwd1D[2][6][7]

Second, along the T axis:

rahtFwd1D[1][0][2]  
rahtFwd1D[1][1][3]  
rahtFwd1D[1][4][6]  
rahtFwd1D[1][5][7]

Third, along the S axis:

rahtFwd1D[0][0][4]  
rahtFwd1D[0][1][5]  
rahtFwd1D[0][2][6]  
rahtFwd1D[0][3][7]

The expression rahtFwd1Dref[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place, forward, two-point transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis.

rahtFwd1Dref[k][aIdx][bIdx] := RahtFwd(aCoeffRef, bCoeffRef, waRef, wbRef)  
 where  
 aCoeffRef := RahtInterPredBlk[aIdx]  
 bCoeffRef := RahtInterPredBlk[bIdx]  
 waRef := RahtCoeffWeightMRef[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wbRef := RahtCoeffWeightMRef[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

### Sample domain prediction and last component prediction

#### General

Subclauses 10.5.5.2 - 10.5.5.4, subclause 10.5.6.2 and apply when raht\_prediction\_enabled\_flag is 1 and lossless\_coding\_enabled is 0, subclause 10.5.5.7 and subclauses 10.5.6.3 - 10.5.6.4 apply when slice\_attr\_inter\_prediction is 1 and lossless\_coding\_enabled is 0, subclause 10.5.6.5 applies when last\_comp\_pred\_enabled is 1 and lossless\_coding\_enabled is 0. These specify the sample domain prediction and last component prediction in transform domain for a block located at ( Bs, Bt, Bv ) in tree level Lvl. These shall be performed for every eligible block for intra prediction (10.5.5.2) or inter prediction (10.5.5.7) in the tree level, in any order, by:

* generating a prediction block (10.5.5.3) and/or;
* generating a inter prediction block (10.5.6.4);
* weighting the values of the prediction block (10.5.6.2);
* resampling and weighting the inter prediction block (10.5.6.3);
* if the DC values of the current block and the reference block are similar, applying temporal filtering (10.5.6.4) for the inter prediction block and modifying the prediction block from the inter prediction block;
* applying the last component prediction in transform domain(10.5.6.5);

The prediction block is specified in terms of the eight-element array RahtPredBlk; RahtPredBlk[ 𝑚 ] is the prediction block value for the Morton-coded location 𝑚. The inter prediction block is specified in terms of the eight-element array RahtInterPredBlk; RahtInterPredBlk[ 𝑚 ] is the inter prediction block value for the Morton-coded location 𝑚.

if (RahtPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtPredBlk[m] = RahtPredW[m]  
  
}   
if (RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++)  
 RahtInterPredBlk[m] = RahtInterPredW[m]  
  
 if (RahtDcCoeffRef[lvl][Bs][Bt][Bv] < RahtDcCoeff[lvl][Bs][Bt][Bv] \* 0.5 || RahtDcCoeffRef[lvl][Bs][Bt][Bv] > RahtDcCoeff[lvl][Bs][Bt][Bv] \* 2)  
 continue (not inter-eligible)  
  
 … /\* in-place, temporal filtering of RahtInterPredBlk （10.5.6.4） \*/  
  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtPredBlk[m] = RahtInterPredBlk[m]  
 }  
}  
  
/\* in-place, last component prediction for RahtCoeff（10.5.6.5） \*/

#### Weighted prediction block

The upsampled prediction block shall be weighted with each sample weighted by the square root of the corresponding transform coefficient weight. The weighted sample value is specified by RahtPredW[ 𝑚 ] for the block located at ( Bs, Bt, Bv ) in tree level Lvl.

RahtPredW[m] := lossless\_coding\_enabled == 0 ? DivExp2Fz(RahtPred[m] × w, 15) : (RahtPred[m] >> 15) << 15  
 where  
 w := IntSqrt(RahtCoeffWeightM[Lvl][0][Bs][Bt][Bv][m] << 30)

#### Resampling and Weighted inter prediction block

The samples used to generate an inter prediction block are the sum of point attributes within each child block of the reference block as specified by RahtInterPred; RahtInterPred[ lvl  - 1][ bsc ][ btc ][ bvc ] is the sample value for the child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

RahtInterPred[lvl][bs][bt][bv] := RahtBlkSumAttrRef[ lvl ][ bs ][ bt ][ bv ]

The weighted sample value is specified by RahtInterPredW[ m ] for the m-th child block located at ( bsc, btc, bvc ) in tree level lvl  - 1.

* If one child block is with zero weight, the sample used to generate the inter prediction block shall be set to 0;
* Else, if the corresponding child block of the reference block is with zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the reference block and be weighted by the transform coefficient weight of the reference block;
* Else, the corresponding child block of the reference block is with non-zero weight, the sample used to generate the inter prediction block shall be the sum of point attributes within the corresponding child block of the reference block and be weighted by its corresponding transform coefficient weight.

RahtInterPredW[ m ] := RahtBlkWeight[lvl - 1][bsc][btc][bvc] ?   
(RahtBlkWeightRef[lvl - 1][bsc][btc][bvc] ? DivExp2Fz((RahtInterPred[lvl – 1]  
[bsc][btc][bvc] >> wShift ) \* rsqrtW, 15) : DivExp2Fz((RahtInterPred[lvl]  
[bs][bt][bv] >> wShiftP ) \* rsqrtWP, 15)) : 0   
where  
 w := RahtBlkWeightRef[lvl - 1][bsc][btc][bvc]  
 wShift := w > 1024 ? IntLog2(w - 1) >> 1 : 0  
 rsqrtW := IntRecipSqrt(w) >> (25 – wShift)  
 w := RahtBlkWeightRef[lvl][bs][bt][bv]  
 wShiftP := wP > 1024 ? IntLog2(wP - 1) >> 1 : 0  
 rsqrtWP := IntRecipSqrt(wP) >> (25 – wShiftP)

#### RAHT temporal filtering

When the tree level lvl  of the current block is smaller than or equal to RahtRootLvl – raht\_inter\_skip\_layers, the temporal filtering is enabled for the transformed inter prediction block. The transformed inter-predicted block is filtered by the derived filter as specified by RahtInterFilter.

RahtInterPredBlk[m] = lvl <= (RahtRootLvl – raht\_skip\_layers) ? (RahtInterFilter \* RahtInterPredBlk[m]) >> 7 : RahtInterPredBlk[m]

RahtInterFilter is determined for each tree level. If raht\_send\_filters is equal to 0, RahtInterFilter is determined by a set of stored filters as specified by FixedRahtFilters, based on the tree level lvl. If raht\_send\_filters is equal to 1, raht\_inter\_filter\_qidx[dpth - raht\_skip\_layers] is inverse quantized and subtracted from 128 to derive RahtInterFilter.

if(raht\_send\_filters)  
 RahtInterFilter = 128 – (raht\_inter\_filter\_qidx[dpth - raht\_skip\_layers] × AttrQstep[qp]>> 7)  
else  
 RahtInterFilter = FixedRahtFilters[LvlIdx]  
 where  
 FixedRahtFileters := [128, 128, 128, 127, 125, 121, 115]  
 LvlIdx := dpth > 6 ? 6 : dpth  
 dpth := RahtRootLevel – lvl

#### Last component prediction for RAHT

When attr\_coding\_type is 0 and raht\_last\_comp\_pred\_enabled is 1, the third attribute coefficient component is, if present, a residual to a prediction by the second scaled coefficient component. The expression *LcpScale*[lvl][nIdx]specifies the scale factor applied at the nIdx -th coded block in transform level lvl to the second coefficient components to predict third coefficient components. The scale factor *LcpScale*[lvl][nIdx]is calculated using the second and third components of the coefficients of its preceding 128 blocks.

if (AttrDim == 3) {  
 bs = RahtBlkLoc[Lvl][nIdx][0]  
 bt = RahtBlkLoc[Lvl][nIdx][1]  
 bv = RahtBlkLoc[Lvl][nIdx][2]  
 LcpScale[lvl][nIdx] = 0;  
 for(n = 0; n < Min(nIdx,128); n++){  
 ns = RahtBlkLoc[Lvl][nIdx - n][0]  
 nt = RahtBlkLoc[Lvl][nIdx - n][1]  
 nv = RahtBlkLoc[Lvl][nIdx - n][2]  
 for(m = 0; m < 8; m++){  
 if(m == 0 && Lvl < RahtRootLvl)  
 continue；  
 sumk1k1 += RahtCoeffLcp[lvl][ns][nt][nv][m][1] \* RahtCoeffLcp[lvl][ns][nt][nv][m][1];  
 sumk1k2 += RahtCoeffLcp[lvl][ns][nt][nv][m][1] \* RahtCoeffLcp[lvl][ns][nt][nv][m][2];  
 }  
 }  
 if (sumk1k2 && sumk1k1) {  
 LcpScale[lvl][nIdx] = Clip3(-16, 16, Div(sumk1k2, sumk1k2,4));  
 }  
 RahtCoeffLcp[lvl][bs][bt][bv][m][2] += DivExp2Floor(LcpScale[lvl][nIdx] ×   
 RahtCoeffLcp[lvl][bs][bt][bv][m][1],4)  
}

when AttrDim is 3, RahtCoeffLcp[ lvl ][ bs ][ bt ][ bv ][ m][ Cidx] specifies the coefficient residuals for the Cidx component of RahtCoeff[ lvl ][ bs ][ bt ][ bv ][ m], where Cidx ∈ 0 .. AttrDim – 1.

### Inverse transform

#### General

Subclause 10.5.7 specifies the inverse transform for a block located at ( Bs, Bt, Bv ) in tree level Lvl. It shall be applied to every block in the tree level in any order.

When sample domain prediction is applied to the transform block, the prediction is added to the inverse transformed block.

if (RahtPredEligible[Lvl][Bs][Bt][Bv] || RahtInterPredEligible[Lvl][Bs][Bt][Bv]) {  
 for (m = 0; m < 8; m++){  
 if (m == 0 && Lvl < RahtRootLvl)  
 continue  
 RahtCoeff[Lvl][Bs][Bt][Bv][m] += RahtPredBlk[m]  
 }  
}

#### DC transform coefficient inheritance

Each block other than the root node of the transform tree shall inherit its DC coefficient from the corresponding inverse-transformed coefficient in its parent block.

When sample domain prediction is not applied to the transform block, the DC coefficient is obtained as follows:

If raht\_buffer\_extension\_flag is equal to 0, the inherited coefficient shall be rounded to retain two fractional bits, with half values rounded away from zero.

if (Lvl < RahtRootLvl)  
 RahtCoeff[Lvl][Bs][Bt][Bv][0] = raht\_buffer\_extension\_flag == 0 ? DivExp2Fz(RahtDcCoeff[Lvl][Bs][Bt][Bv], 13) << 13 : RahtDcCoeff[Lvl][Bs][Bt][Bv]

For a block located at ( bs, bt, bv ) in tree level lvl, the corresponding coefficient in the parent block is specified by RahtDcCoeff[ lvl ][ bs ][ bt ][ bv ].

RahtDcCoeff[lvl][bs][bt][bv] := lvl < RahtRootLvl ? RahtCoeff[lvl + 1][bs / 2][bt / 2][bv / 2][mP] : 0  
 where  
 mP := Morton[bs & 1][bt & 1][bv & 1]

When sample domain prediction is applied to the transform block, the DC coefficient is inferred as the difference between the inherited coefficient and the DC coefficient of the prediction block.

if (Lvl < RahtRootLvl)  
 RahtCoeff[Lvl][Bs][Bt][Bv][0] = RahtDcCoeff[Lvl][Bs][Bt][Bv] – RahtPredDcCoeff[Lvl][Bs][Bt][Bv]

For a block located at ( bs, bt, bv ) in tree level lvl, the corresponding DC coefficient of the prediction block is specified by RahtPredDcCoeff[ lvl ][ bs ][ bt ][ bv ].

[Ed. (YZ): Definition of RahtPredDcCoeff[ lvl ][ bs ][ bt ][ bv ] is TBD.]

#### For a 2×2×2 transform block

The inverse transform for a 2×2×2 block located at ( Bs, Bt, Bv ) in tree level Lvl comprises transforming pairs of coefficients along each axis.

First, along the S axis:

rahtInv1D[0][0][4]  
rahtInv1D[0][1][5]  
rahtInv1D[0][2][6]  
rahtInv1D[0][3][7]

Second, along the T axis:

rahtInv1D[1][0][2]  
rahtInv1D[1][1][3]  
rahtInv1D[1][4][6]  
rahtInv1D[1][5][7]

Third, along the V axis:

rahtInv1D[2][0][1]  
rahtInv1D[2][2][3]  
rahtInv1D[2][4][5]  
rahtInv1D[2][6][7]

The expression rahtInv1D[ 𝑘 ][ aIdx ][ bIdx ] specifies the invocation of the in-place inverse transform for the aIdx-th and bIdx-th coefficients along the 𝑘-th axis of the block.

rahtInv1D[k][aIdx][bIdx] := RahtInv(aCoeff, bCoeff, wa, wb)  
 where  
 aCoeff := RahtCoeff[Lvl][Bs][Bt][Bv][aIdx]  
 bCoeff := RahtCoeff[Lvl][Bs][Bt][Bv][bIdx]  
 wa := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][aIdx]  
 wb := RahtCoeffWeightM[Lvl][stage][Bs][Bt][Bv][bIdx]  
 stage := 2 − k

#### For co-located points

The inverse transform for a block of co-located points located at ( Bs, Bt, Bv ) in tree level 0 comprises iteratively transforming the block's DC coefficient paired with each successive block coefficient.

for (i = 1; i < RahtBlkWeight[0][Bs][Bt][Bv]; i++)  
 rahtInvDup[i]

The expression rahtInvDup[ 𝑖 ] specifies the invocation of the in-place inverse transform for the 𝑖-th coefficient.

rahtInvDup[i] := RahtInv(aCoeff, bCoeff, wa, 1)  
 where  
 aCoeff := RahtCoeff[0][Bs][Bt][Bv][0]  
 bCoeff := RahtCoeff[0][Bs][Bt][Bv][i]  
 wa := RahtBlkWeight[0][Bs][Bt][Bv] − i

#### Inverse two-point transform

This subclause specifies the in-place, inverse, two-point transform RahtInv( aCoeff, bCoeff, wa, wb ). Its parameters are:

* the expressions aCoeff and bCoeff that respectively identify low- and high-frequency transform coefficients;
* the weights wa and wb that are the respective coefficient weights for aCoeff and bCoeff.

The transform basis vectors use 15-fractional-bit, fixed-point coefficients 𝑎 and 𝑏:

a := IntSqrt(wa << 30) × IntRecipSqrt(wa + wb) >> 40  
b := IntSqrt(wb << 30) × IntRecipSqrt(wa + wb) >> 40

If either wa or wb is 0, the transform result is:

if (wa == 0 || wb == 0) {  
 if (lossless\_coding\_enabled == 0) {  
 ya = wa ≠ 0 ? aCoeff : 0  
 yb = wb ≠ 0 ? aCoeff : 0  
 }  
 else {  
 ya = aCoeff – ((bCoeff >> 16) << 15)  
 yb = bCoeff + ya  
 }  
}

If either wa or wb is zero, the respective opposing coefficient 𝑏 or 𝑎 is not necessarily .

Otherwise (both wa and wb are greater than 0), the transform result is:

if (wa ≠ 0 && wb ≠ 0) {  
 ya = DivExp2Fz(aCoeff × a, 15) − DivExp2Fz(bCoeff × b, 15)  
 yb = DivExp2Fz(bCoeff × a, 15) + DivExp2Fz(aCoeff × b, 15)  
}

### Reconstructed attribute values

Reconstructed attribute values are specified by the expression RahtRecon[ 𝑠 ][ 𝑡 ][ 𝑣 ][ 𝑖 ] for an 𝑖-th co-located point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ). They are:

* extracted from inverse transformed blocks in the bottom two tree levels; unique points from tree level 1; duplicate points from tree level 0; then
* rounded to discard the 15 fractional bits of the fixed-point representation, with halve values rounded away from zero; then
* clipped to be within the attribute value range [ 0, AttrMaxVal ].

RahtRecon[s][t][v][i] := Clip3(0, AttrMaxVal, DivExp2Fz(value, 15))  
 where  
 value := RahtBlkWeight[0][s][t][v] == 1  
 ? RahtDcCoeff[0][s][t][v]  
 : RahtCoeff[0][s][t][v][i]

The mapping of the slice geometry to reconstructed attribute values shall map points with identical attribute coordinates to successive elements 𝑖 of RahtRecon[ 𝑠 ][ 𝑡 ][ 𝑣 ][ 𝑖 ] in canonical point order. i.e. the 𝑖-th element shall be the 𝑖-th instance of the attribute coordinates ( 𝑠, 𝑡, 𝑣 ) from the start of AttrPos.

The following is specified in terms of the sparse array dupPtIdx; dupPtIdx[ 𝑠 ][ 𝑡 ][ 𝑣 ] is the cumulative count of points with attribute coordinates ( 𝑠, 𝑡, 𝑣 ). Unset elements of dupPtIdx shall be inferred to be 0.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++) {  
 s = AttrPos[ptIdx][0]  
 t = AttrPos[ptIdx][1]  
 v = AttrPos[ptIdx][2]  
 i = dupPtIdx[s][t][v]  
 dupPtIdx[s][t][v]++  
  
 PointAttr[ptIdx][Cidx] = RahtRecon[s][t][v][i]  
}

## Attribute decoding using levels of detail

### General

The attribute decoding processes specified by 10.6 are distance-based prediction schemes that use a hierarchical level-of-detail representation of the slice geometry. They apply when attr\_coding\_type is either 1 or 2.

Detail levels are defined by an iterative subsampling process (10.6.5). The finest detail level comprises all points in the slice geometry. With each iteration, a coarser detail level is generated from the previous coarsest detail level.

Every detail level comprises a list of points present in the detail level, and is associated with a list of refinement points. A refinement point is a point that is present in a detail level and not present in any coarser detail level; the refinement points for detail level lvl, when combined with the coarser detail level lvl + 1, form detail level lvl.

For each refinement point, a set of neighbouring points is determined (10.6.6) using inter-detail-level, intra-detail-level and inter-frame searches. The neighbouring points form a predictor set that is used to predict attribute/transform coefficient values.

Attribute reconstruction (10.6.7) proceeds from the coarsest to the finest detail level. Transform coefficients are coded in the same order.

A coded transform coefficient is associated with each refinement point. The transform (10.6.12) comprises two operations: an update step that modifies the predicting points and a prediction step that adds the transform coefficient to a predicted attribute/coefficient value.

图片包含 图示

描述已自动生成

Figure 20 — Example of points in three detail levels and their spatial arrangement.

An example level-of-detail hierarchy is illustrated in Figure 20. is the finest detail level and corresponds to all points in the slice. The generation of subsequent detail levels is performed using periodic subsampling with samplingPeriod equal to 4. The number of detail levels is limited to 3. The points in that are not assigned to are in refinement list . The attribute value for the marked point ● in for is predicted from a set of spatially neighbouring points found using an inter-detail-level search of and an intra-detail-level search of points earlier in . Transform coefficient values are associated with each refinement point and coded from to . The refinement list comprises all points in .

### Syntax element semantics

lod\_dist\_log2\_offset specifies an offset to the APS-specified finest detail level block size lod\_initial\_dist\_log2 for LoD generation and predictor searches. When lod\_dist\_log2\_offset is not present, it shall be inferred to be 0.

### Reconstruction process

The reconstruction of point attribute values comprises:

* deriving a set of detail levels from the slice geometry (10.6.5);
* searching for point predictors (10.6.6);
* determining transform coefficient weights (10.6.11) or point quantization weights (10.6.13); and
* reconstructing attribute values from coded coefficients (10.6.7).

The reconstructed values are stored in the array PointAttr.

### State variables

Levels of detail are specified in terms of the following state variables; the index lvl identifies a detail level:

* The variable LodCnt, a count of detail levels generated from the slice geometry.
* The array LodPtCnt, the size of each detail level; LodPtCnt[ lvl ] is the number of points in the identified detail level.
* The array LodPtIdx, identifying points in each detail level by their index in the canonical decoding order; LodPtIdx[ lvl ][ 𝑖 ] is the AttrPos index of the 𝑖-th point in the identified detail level.
* The array LodRfmtPtCnt, the size of each detail level's refinement list; LodRfmtPtCnt[ lvl ] is the number of points in the refinement list for the identified detail level.
* The array LodRfmtPtIdx, identifying points in each refinement list by their index in the canonical decoding order; LodRfmtPtIdx[ lvl ][ 𝑖 ] is the AttrPos index of the 𝑖-th refinement point for the identified detail level.

Point predictors are specified in terms of the following state variables; the index ptIdx identifies a point by its index into AttrPos:

* The array PredCnt; PredCnt[ ptIdx ] is the size of the predictor set for the identified point.
* The array PredPtIdx, identifies point predictors by their index in the canonical decoding order; PredPtIdx[ ptIdx ][ ni ] is the AttrPos index or RefAttrPos index of the ni-th point in the predictor set for the identified point.
* The array PredPtRef of the flags to specify whether the point predictors are searched from the reference slice. When PredPtRef[ ptIdx ][ ni ] is equal to 1, the ni-th point in the predictor set is specified to be searched from the reference slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ]; When PredPtRef[ ptIdx ][ ni ] is equal to 0, the ni-th point in the predictor set is specified to be searched from the current slice for the predictor identified by PredPtIdx[ ptIdx ][ ni ].
* The array PredWeight of point predictor weights; PredWeight[ ptIdx ][ ni ] is the prediction weight for the predictor identified by PredPtIdx[ ptIdx ][ ni ].
* The array CoeffWeight of transform coefficient weights; CoeffWeight[ ptIdx ] is the normalization weight for the transform coefficients associated with the identified point.
* The array QuantWeight of point quantization weights; QuantWeight[ ptIdx ] is the quantization weight for the transform coefficients associated with the identified point.

### Levels of detail

#### General generation process

The effect of this process is to represent the LoD structure in the state variables LodCnt, LodPtCnt, LodPtIdx, LodRfmtPtCnt and LodRfmtPtIdx.

The finest detail level shall contain the entire slice geometry (10.6.5.2). It is identified by the detail level index 0.

Detail levels shall be iteratively subsampled (10.6.5.4), starting from the finest detail level, until either a single point remains or lod\_max\_levels\_minus1 subsampled detail levels have been produced. The variable Lvl identifies the detail level to be subsampled.

Lvl = 0  
for (; Lvl < lod\_max\_levels\_minus1; Lvl++) {  
 if (LodPtCnt[Lvl] == 1)  
 break  
 … /\* subsample LodPtIdx[Lvl] \*/  
}  
LodCnt = Lvl + 1

The coarsest detail level is identified by the detail level index LodCnt − 1. All points in the coarsest detail level shall be assigned to the coarsest level's refinement list (10.6.5.3).

#### The finest detail level

The AttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the slice geometry.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 LodPtIdx[0][ptIdx] = ptIdx  
LodPtCnt[0] = PointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The variable maxPtsPerSort identifies the max group size when sorting by group.

maxPtsPerSort = !attr\_canonical\_order\_enabled && !max\_points\_per\_sort\_log2\_plus1   
? LodPtCnt[0] : 1 << (max\_points\_per\_sort\_log2\_plus1 - 1)

The sorted order shall be identical for the decoding of all attributes in a single slice with identical attribute coordinate arrays (AttrPos).

Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < LodPtCnt[0]; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, LodPtCnt[0]);  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = LodPtIdx[0][i]  
 jPtIdx = LodPtIdx[0][j]  
 iMorton = Morton(AttrPos[iPtIdx][0], AttrPos[iPtIdx][1], AttrPos[iPtIdx][2])  
 jMorton = Morton(AttrPos[jPtIdx][0], AttrPos[jPtIdx][1], AttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(LodPtIdx[0][i], LodPtIdx[0][j])  
 }  
}

#### The coarsest detail level

After generation of the LoD hierarchy, all points in the coarsest detail level shall be assigned to its refinement list.

for (i = 0; i < LodPtCnt[LodCnt − 1]; i++)  
 LodRfmtPtIdx[LodCnt − 1][i] = LodPtIdx[LodCnt − 1][i]

#### Generation of a single detail level

The coarser detail level Lvl + 1 shall be produced by subsampling the points of detail level Lvl.

The following definitions are used in the specification of the subsampling processes:

* The expression InLodPtCnt is an alias for LodPtCnt[ Lvl ], the number of points in the input detail level.
* The expression InLodPtIdx[ 𝑖 ] is an alias for LodPtIdx[ Lvl ][ 𝑖 ], the point indexes of the input detail level.
* The expression OutLodPtCnt is an alias for LodPtCnt[ Lvl + 1 ], the number of points in the output detail level.
* The expression OutLodPtIdx[ 𝑖 ] is an alias for LodPtIdx[ Lvl + 1 ][ 𝑖 ], the point indexes of the output detail level.
* The expression OutRfmtPtIdx[ 𝑖 ] is an alias for LodRfmtPtIdx[ Lvl ][ 𝑖 ], the point indexes of the refinement list for detail level Lvl.

Subsampling partitions points in the input detail level into an output detail level and the refinement list for the input detail level. The partitioning process shall preserve the relative ordering of points in the input detail level.

Subsampling shall proceed according to:

* block-based subsampling (10.6.5.8) if lod\_scalability\_enabled is 1, or lod\_decimation\_mode is 2;
* periodic subsampling (10.6.5.5) if lod\_decimation\_mode is 1; or
* distance-based subsampling (10.6.5.6) otherwise.

#### Periodic subsampling

When aps\_extension\_present is 1, geom\_tree\_type is 1, and attr\_canonical\_order\_enabled is 0, the variable CanonicalLodSubsampling is set to 1. Otherwise, CanonicalLodSubsampling is set to 0.

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

If CanonicalLodSubsampling is equal to 0, periodic subsampling generates a subsampled output detail level by:

The sampling period for the current detail level is subsamplingPeriod.

samplingPeriod := 2 + lod\_sampling\_period\_minus2[Lvl]

Input points shall be assigned to either the output detail level or the refinement list according to their index in the input detail level modulo the sampling period:

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 if (i % samplingPeriod)  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 else  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
}

If CanonicalLodSubsampling is equal to 1, canonical periodic subsampling generates a subsampled output detail level by:

The sampling period for the current detail level is subsamplingPeriod.

samplingPeriod := (Lv1==0 ? 1 : canonical\_samplingPeriod[Lvl-1]) \* (2 + lod\_sampling\_period\_minus2[Lvl])

Input points shall be assigned to either the output detail level or the refinement list according to their index in the input detail level modulo the sampling period:

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 if (InLodPtIdx[i] % samplingPeriod)  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 else  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
}  
canonical\_samplingPeriod[Lvl] = samplingPeriod

#### Distance-based subsampling

Distance-based subsampling generates a subsampled output detail level by:

* spatially partitioning the input detail level into a lattice of sized cubic blocks; and
* assigning at most one point from each block to the subsampled detail level.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The subsampling process is specified in terms of the following state variables; the indexes bs, bt and bv identify the block location ( bs, bt, bv ):

* The sparse array MapSub; MapSub[ bs ][ bt ][ bv ] equal to 1 indicates that the identified block contains a single point previously assigned to the subsampled detail level. Unset elements of MapSub are inferred to be 0.
* The sparse array MapPtIdx, identifies points assigned to the subsampled detail level. When MapSub[ bs ][ bt ][ bv ] is 1, MapPtIdx[ bs ][ bt ][ bv ] is the AttrPos index of the point assigned to the subsampled detail level.

The points in the input detail level shall be processed sequentially. For each input point:

* The variable PtIdx is the AttrPos index of the point.
* The block location ( Bs, Bt, Bv ) is determined from the point's attribute coordinates.
* Depending upon the result of a per-point test (10.6.5.7), the point shall be assigned to either the output detail level or the refinement list. The result of the test is the variable IsSubsampledPoint.

OutLodPtCnt = outRfmtPtCnt = 0  
for (i = 0; i < InLodPtCnt; i++) {  
 PtIdx = InLodPtIdx[i]  
 Bs = AttrPos[PtIdx][0] >> BlkSizeLog2  
 Bt = AttrPos[PtIdx][1] >> BlkSizeLog2  
 Bv = AttrPos[PtIdx][2] >> BlkSizeLog2  
  
 … /\* IsSubsampledPoint = result of per−point test (10.6.5.7) \*/  
  
 if (MapSub[Bs][Bt][Bv] || ¬IsSubsampledPoint)  
 OutRfmtPtIdx[outRfmtPtCnt++] = PtIdx  
 else {  
 OutLodPtIdx[OutPtCnt++] = PtIdx  
 MapSub[Bs][Bt][Bv] = 1   
 MapPtIdx[Bs][Bt][Bv] = PtIdx  
 }  
}

#### Per-point decision for distance-based subsampling

The derivation of IsSubsampledPoint specifies whether the point shall be assigned to the output detail level.

A point shall be assigned to the output detail level unless the squared distance between it and any previously assigned point from a set of adjacent blocks within an availability window is less than or equal to sqRadius. Each availability window shall be a 128×128×128 block volume identified by ( Bs >> 7, Bt >> 7, Bv >> 7 ).

sqRadius = 3 << 2 × (BlkSizeLog2 − 1)

图示, 工程绘图

描述已自动生成

Figure 21 — Example decisions using distance-based subsampling.

Example per-point decisions are illustrated in Figure 21. Subsampling generates three detail levels. The points assigned to by subsampling (when Lvl = 0) are not within the shaded radius of any other point in . The block size used to subsample is BlkSizeLog2 = 1. All points are within a single availability window.

The array neighPtIdx is a neighCnt-element list of AttrPos indexes of points present in the adjacent blocks of the output detail level that are within the availability window. Table 29 specifies the relative locations of the adjacent blocks.

neighCnt = 0  
for (i = 0; i < 19; i++) {  
 ns = Bs + adjBlkOffset[i][0]  
 nt = Bt + adjBlkOffset[i][1]  
 nv = Bv + adjBlkOffset[i][2]  
 unavailable = (ns ^ Bs) >> 7 || (nt ^ Bt) >> 7 || (nv ^ Bv) >> 7  
 if (unavailable)  
 continue  
  
 if (MapSub[ns][nt][nv])  
 neighPtIdx[neighCnt++] = MapPtIdx[ns][nt][nv]  
}

Table 41— Adjacent block coordinates, adjBlkOffset[ 𝑖 ][ 𝑘 ], relative to ( Bs, Bt, Bv )

| 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| **0** | −1 | 0 | 0 | **5** | −1 | 1 | 0 | **10** | −1 | 0 | −1 | **15** | 1 | −1 | −1 |
| **1** | 0 | −1 | 0 | **6** | 0 | 1 | −1 | **11** | −1 | −1 | 0 | **16** | −1 | 1 | −1 |
| **2** | 0 | 0 | −1 | **7** | 1 | 0 | −1 | **12** | −1 | 1 | 1 | **17** | −1 | −1 | 1 |
| **3** | 0 | −1 | 1 | **8** | 1 | −1 | 0 | **13** | 1 | −1 | 1 | **18** | −1 | −1 | −1 |
| **4** | −1 | 0 | 1 | **9** | 0 | −1 | −1 | **14** | 1 | 1 | −1 |  | | | |

The point's attribute coordinates shall be compared to those of each point identified by the neighPtIdx array to determine the value of IsSubsampledPoint.

IsSubsampledPoint = 1  
for (i = 0; i < neighCnt; i++) {  
 sqDist = 0  
 for (k = 0; k < 3; k++) {  
 d = AttrPos[neighPtIdx[i]][k] − AttrPos[PtIdx][k]  
 sqDist += d × d  
 }  
  
 if (sqDist ≤ sqRadius)  
 IsSubsampledPoint = 0  
}

#### Block-based subsampling

Block-based subsampling generates a subsampled output detail level by:

* spatially partitioning the input detail level into a lattice of sized cubic blocks;
* grouping together blocks, in Morton order, according to the number of points they contain; and
* assigning one point from each block group to the subsampled detail level.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

Under certain conditions, blocks correspond to nodes of the occupancy tree. For instance, when lod\_scalability\_enabled is 1.

A list of block groups shall be generated by traversing the input detail level in canonical order. Consecutive blocks shall be grouped together until the group spans at least minGrpPts points.

minGrpPts := lod\_scalability\_enabled ? 0 : 2 + lod\_sampling\_period\_minus2[Lod]

The array grpBdry, with elements grpBdry[ grpIdx ], identifies block group boundaries as indexes into the input detail level array InLodPtIdx.

for (i = 1, grpStart = 0; i < InLodPtCnt; i++) {  
 ptIdx = InLodPtIdx[i]  
 ptIdxPrev = InLodPtIdx[i − 1]  
 bdryS = (AttrPos[ptIdx][0] ^ AttrPos[ptIdxPrev][0]) >> BlkSizeLog2  
 bdryT = (AttrPos[ptIdx][1] ^ AttrPos[ptIdxPrev][1]) >> BlkSizeLog2  
 bdryV = (AttrPos[ptIdx][2] ^ AttrPos[ptIdxPrev][2]) >> BlkSizeLog2  
 if (bdryS | bdryT | bdryV)  
 if (i − grpStart ≥ minGrpPts)  
 grpBdry[grpCnt++] = grpStart = i  
}  
grpBdry[grpCnt++] = InLodPtCnt

For each group of blocks, a test (10.6.5.9) shall be performed to determine the index of the point to be assigned to the output detail level. All other points shall be assigned to the refinement list. The variables GrpStart and GrpEnd identify the start and end of a block group. The result of the test is the variable IdxOfSubsampledPoint.

OutLodPtCntSize = outRfmtPtCnt = 0  
for (GrpStart = grpIdx = 0; grpIdx < grpCnt; GrpStart = grpBdry[grpIdx++]) {  
 GrpEnd = grpBdry[grpIdx]  
  
 … /\* IdxOfSubsampledPoint = result of per−point test (10.6.5.9) \*/  
  
 for (i = GrpStart; i < GrpEnd; i++) {  
 if (IdxOfSubsampledPoint == i)  
 OutLodPtIdx[OutLodPtCnt++] = InLodPtIdx[i]  
 else  
 OutRfmtPtIdx[outRfmtPtCnt++] = InLodPtIdx[i]  
 }  
}

#### Per block-group decision for block-based subsampling

The derivation of IdxOfSubsampledPoint specifies the input detail level index of the point in the block group that shall be assigned to the output detail level.

The distance to the block group centroid shall be used to select the point assigned to the output detail level. The block group centroid and point distances shall be calculated using attribute coordinates quantized by Exp2( BlkSizeLog2 − 1 ). The distance metric shall be the Manhattan distance.

图示, 工程绘图

描述已自动生成

Figure 22 — Example decisions using block group subsampling with minGrpPts = 3.

Example per-point decisions are illustrated in Figure 22. Subsampling generates two detail levels. To subsample , the points are grouped into block groups containing a minimum of three points. In this example, the block size for (Lvl = 0) is BlkSizeLog2 = 1. The first block group (solid shading) comprises four points from three blocks each with one, one and two points, respectively. The first point that is the closest to the centroid of the points in the block group is assigned to .

The block group centroid shall be the sum of all quantized attribute coordinates, centroidSum, divided by the number of points in the block group, numPtsInGrp.

numPtsInGrp := GrpEnd − GrpStart  
  
for (k = 0; k < 3; k++)  
 centroidSum[k] = 0  
  
for (i = 0; i < numPtsInGrp; i++) {  
 ptIdx = InLodPtIdx[GrpStart + i]  
 for (k = 0; k < 3; k++)  
 centroidSum[k] += AttrPos[ptIdx][k] >> BlkSizeLog2 − 1  
}

The array ptDist maps the index of each point in the block group to the distance between it and the centroid.

for (i = 0; i < numPtsInGrp; i++) {  
 ptIdx = InLodPtIdx[GrpStart + i]  
 ptDist[i] = 0  
 for (k = 0; k < 3; k++) {  
 posk = AttrPos[ptIdx][k] >> BlkSizeLog2 − 1  
 ptDist[i] += Abs(posk × numPtsInGrp − centroidSum[k])  
 }  
}

The point closest to the block group centroid shall be assigned to the output detail level. In the case that the block group contains multiple closest points, the selected point is the closest point with:

* when lod\_scalability\_enabled is 1 and Lvl is odd: the greatest InLodPtIdx index;
* when lod\_scalability\_enabled is 0 or Lvl is even: the lowest InLodPtIdx index.

last := lod\_scalability\_enabled ? Lvl & 1 : 1

minIdx = 0  
for (i = 1; i < numPtsInGrp; i++)  
 if (last ? dist[i] ≤ dist[minIdx] : dist[i] < dist[minIdx])  
 minIdx = i

IdxOfSubsampledPoint = GrpStart + minIdx

#### The finest detail level of the reference slice

When slice\_attr\_inter\_prediction is equal to 1, the finest detail level of the reference slice is specified in terms of the following state variables:

* The variable refLodPtCnt, the size of the finest detail level.
* The array RefLodPtIdx, identifying points in the finest detail level by their index in the canonical decoding order; RefLodPtIdx [ 𝑖 ] is the RefAttrPos index of the 𝑖-th point in the finest detail level.

The RefAttrPos point indexes of the finest detail level shall have an initial one-to-one correspondence with the canonical decoding order of the reference slice geometry.

for (ptIdx = 0; ptIdx < refPointCnt; ptIdx++)  
 LodPtIdx[ptIdx] = ptIdx  
refLodPtCnt = refPointCnt

The point indexes of the finest detail level shall be sorted by group in ascending order of their respective Morton-coded attribute coordinates. The sorted order shall be identical for the decoding of all attributes in a single slice with identical attribute coordinate arrays (RefAttrPos).

Performing a stable sort for each attribute, or reusing the reordered points would satisfy the requirement for identical orders.

An example (inefficient) sorting process is:

for (benIdx = 0; benIdx < RefLodPtCnt; benIdx += maxPtsPerSort) {  
 endIdx = Min(benIdx + maxPtsPerSort, RefLodPtCnt);  
 for (i = benIdx; i < endIdx; i++)  
 for (j = i + 1; j < endIdx; j++) {  
 iPtIdx = RefLodPtIdx[i]  
 jPtIdx = RefLodPtIdx[j]  
 iMorton = Morton(RefAttrPos[iPtIdx][0], RefAttrPos[iPtIdx][1],   
 RefAttrPos[iPtIdx][2])  
 jMorton = Morton(RefAttrPos[jPtIdx][0], RefAttrPos[jPtIdx][1],   
 RefAttrPos[jPtIdx][2])  
 if (iMorton > jMorton)  
 Swap(RefLodPtIdx[i], RefLodPtIdx[j])  
 }  
}

### Predictor search

#### General process

The points used to predict the refinement points of each detail level shall be determined by a search (10.6.6.3).

The effect of this process is to represent the point predictors in the state variables PredCnt, PredPtIdx, PredPtRef and PredWeight.

When attr\_coding\_type is 2, no searches shall be performed for the refinement points of the coarsest detail level.

maxLvl = LodCnt − (attr\_coding\_type == 2)  
for (Lvl = 0; Lvl < maxLvl; Lvl++)  
 for (RfmtIdx = 0; RfmtIdx < LodRfmtPtCnt[Lvl]; RfmtIdx++) {  
 … /\* find predictors (10.6.6.2) of the current point \*/  
 }

#### Minimum reference detail level for inter-level predictor searches

The variable MinInterRefLvl identifies the finest detail level that shall be used as a reference for inter-detail level prediction. When lod\_scalability\_enabled is 1, it shall be the finest detail level with fewer refinement points than the total number of refinement points associated with all finer detail levels.

MinInterRefLvl = 1  
if (lod\_scalability\_enabled) {  
 for (lvl = 1; lvl < LodCnt − 1; lvl++) {  
 if (LodRfmtPtCnt[lvl] < slice\_num\_points\_minus1 − LodPtCnt[lvl])  
 break  
 MinInterRefLvl++  
 }  
}

#### Predictor search for a single refinement point

For a refinement point with index RfmtIdx in detail level Lvl, a search shall be performed to find the closest neighbouring points from a set of candidate neighbours.

The search process is specified in terms of the following variables:

* The variable PtIdx, the AttrPos index of the refinement point.
* The variable RefLvl, the reference detail level used for inter-level predictor searches.

PtIdx = LodRfmtPtIdx[RfmtIdx]  
RefLvl = Max(Lvl + 1, MinInterRefLvl)

An inter-detail-level search shall be performed prior to any intra-level search. Except for the coarsest detail level, the following inter-level searches shall be performed:

* An initial search (10.6.6.6).
* If fewer than three predictors are found (PredCnt[ PtIdx ] < 3), an extended search (10.6.6.7).

When Lvl is greater than or equal to pred\_intra\_min\_lod, an intra-detail-level search (10.6.6.8) shall be performed.

When slice\_attr\_inter\_prediction is equal to 1, an initial inter-frame search (10.6.6.9) and an extended inter-frame search (10.6.6.10) shall be performed.

After completing the searches, weights shall be calculated for each predictor (10.6.6.11), during which the predictor set is pruned and re-sorted. When pred\_blending\_enabled is 1, predictor weights shall be blended (10.6.6.12).

图示

中度可信度描述已自动生成

Figure 23 — Example of searches performed for a single refinement point.

An example predictor search when slice\_attr\_inter\_prediction is equal to 0 is illustrated in Figure 23. Searches are performed for the refinement point 𝛾 in of ; other points in are denoted 𝛼, 𝛽 and 𝛿. Points in the next coarsest detail level, , are marked a to f. After the initial inter-level search, the predictor set is { c, b }. Then, since fewer than three predictors were found, an extended inter-level search is performed over ±pred\_inter\_lod\_search\_range points in the point list. This search adds predictor d to the predictor set { c, b, d }. Finally, an intra- level search is performed over pred\_intra\_lod\_search\_range previous points in . The final predictor set for is { c, 𝛽, 𝛼 }.

#### Inclusion of a candidate point in the predictor set (InsertPredictor)

This subclause defines the function InsertPredictor( candPtIdx  , *candRef*) that conditionally inserts a candidate point into the predictor set of the current refinement point. Each candidate shall be tested against the refinement point's predictor set to determine if and where it is to be inserted.

The parameter candPtIdx is the AttrPos index or the *RefAttrPos* index of the candidate point. The parameter candRef specifies whether the candidate point is searched from the reference slice.

A candidate shall only be inserted into the point's neighbour set once. If candPresent is 1, the candidate is not inserted into the predictor set.

candPresent = 0  
for (i = 0; i < PredCnt[PtIdx]; i++)  
 candPresent |= PredPtIdx[PtIdx][i] == candPtIdx  
 && (slice\_attr\_inter\_prediction  
 ? PredPtRef[PtIdx][i] == candRef  
 : 1)

Otherwise (the candidate is not already present), the following shall be used to decide the inclusion in the predictor set.

* the spatial distance between the candidate and the refinement point.
* the relative spatial location of the candidate to the current point.

The distance shall be calculated as the biased norm weighted by PredBias.

dist = BiasedNorm1(PtIdx, candPtIdx, 0, candRef)

The point shall be inserted into the predictor set, with elements ordered according to the biased distance to the refinement point. When the size of predictor set is less than 6 and prediction\_with\_distribution\_enabled is 1, the point is inserted only when the distance between the point and the current point is equal to the distance between the third predictor in the predictor set and the current point. Points at the same distance shall be ordered by insertion order, with earlier members being ordered before later members.

for (i = 0; i < PredCnt[PtIdx]; i++)  
 if (dist < BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i], 0, PredPtRef[PtIdx][i]))  
 break  
if(i < 3 || ((dist == BiasedNorm1(PtIdx, PredPtIdx[PtIdx][2])) && PredCnt[PtIdx] < 6)  
 for (j = PredCnt[PtIdx]; j > i; j−−){  
 PredPtIdx[PtIdx][j] = PredPtIdx[PtIdx][j − 1]  
 if (slice\_attr\_inter\_prediction)  
 PredPtRef[PtIdx][j] = PredPtRef[PtIdx][j - 1]  
 }  
 PredPtIdx[PtIdx][i] = candPtIdx  
 PredPtRef[PtIdx][i] = slice\_attr\_inter\_prediction ? candRef : 0

The size of the predictor set shall be limited to three elements (when prediction\_with\_distribution\_enabled is 0) or six elements (when prediction\_with\_distribution\_enabled is 1) by discarding the furthest predictor if necessary.

PredCnt[PtIdx] = Min(prediction\_with\_distribution\_enabled ? 6 : 3, PredCnt[PtIdx] + 1)

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

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

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression BiasedNorm1. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance: when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level.

BiasedNorm1(ptIdxA, ptIdxB, pcA, pcB) := dist[0] + dist[1] + dist[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k]) × PredBias[k]   
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

#### Initial inter-level predictor search

The initial inter-level search shall be performed by spatially partitioning the reference detail level into a lattice of sized cubic blocks. Only blocks adjacent to the block containing the refinement point that are within an availability window shall be searched.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The block location ( bs, bt, bv ) identifies the block containing the refinement point.

bs := AttrPos[PtIdx][0] >> BlkSizeLog2  
bt := AttrPos[PtIdx][1] >> BlkSizeLog2  
bv := AttrPos[PtIdx][2] >> BlkSizeLog2

The availability window shall be a 128×128×128 block volume identified by ( bs >> 7, bt >> 7, bv >> 7 ).

The search shall proceed over the search blocks in the order specified by Table 30. Within each search block, points shall be searched in ascending order of index within the reference detail level.

for (si = 0; si < 27; si++) {  
 ss = bs + searchBlkOffsets[si][0]  
 st = bt + searchBlkOffsets[si][1]  
 sv = bv + searchBlkOffsets[si][2]  
 unavailable = (ss ^ bs) >> 7 || (st ^ bt) >> 7 || (sv ^ bv) >> 7  
 if (unavailable)  
 continue  
  
 for (i = 0; i < LodPtCnt[RefLvl]; i++) {  
 candPtIdx = LodPtIdx[RefLvl][i]  
 cs = AttrPos[candPtIdx][0]  
 ct = AttrPos[candPtIdx][1]  
 cv = AttrPos[candPtIdx][2]  
  
 inSblk = cs ≥ (ss << BlkSizeLog2) && cs < (ss + 1 << BlkSizeLog2)  
 inSblk &= ct ≥ (st << BlkSizeLog2) && ct < (st + 1 << BlkSizeLog2)  
 inSblk &= cv ≥ (sv << BlkSizeLog2) && cv < (sv + 1 << BlkSizeLog2)  
  
 if (inSblk)  
 InsertPredictor(candPtIdx, 0)  
 }  
}

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

Table 42 — Search block coordinates, searchBlkOffsets[ 𝑖 ][ 𝑘 ], relative to ( bs, bt, bv )

| 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | | 𝑖 | 𝑘 | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| **0** | 0 | 0 | 0 | **7** | 0 | 1 | 1 | **14** | 1 | 0 | −1 | **21** | 1 | −1 | 1 |
| **1** | −1 | 0 | 0 | **8** | 1 | 0 | 1 | **15** | 1 | −1 | 0 | **22** | 1 | 1 | −1 |
| **2** | 0 | −1 | 0 | **9** | 1 | 1 | 0 | **16** | 0 | −1 | −1 | **23** | 1 | −1 | −1 |
| **3** | 0 | 0 | −1 | **10** | 0 | −1 | 1 | **17** | −1 | 0 | −1 | **24** | −1 | 1 | −1 |
| **4** | 1 | 0 | 0 | **11** | −1 | 0 | 1 | **18** | −1 | −1 | 0 | **25** | −1 | −1 | 1 |
| **5** | 0 | 1 | 0 | **12** | −1 | 1 | 0 | **19** | 1 | 1 | 1 | **26** | −1 | −1 | −1 |
| **6** | 0 | 0 | 1 | **13** | 0 | 1 | −1 | **20** | −1 | 1 | 1 |  | | | |

#### Extended inter-level search

The extended inter-level search evaluates predictor candidates over a span of indexes in the reference detail level.

The span shall be centred around the index centre in the reference detail level. It shall be the index of:

* if at least one predictor has been found for the current point: the first predictor; or

if (PredCnt[PtIdx])  
 for (centre = 0; centre < LodPtCnt[RefLvl] − 1; centre++)  
 if (LodPtIdx[RefLvl][centre] == PredPtIdx[PtIdx][0])  
 break

* otherwise (no predictors have been found): the first point with Morton-coded attribute coordinates greater than those of the current point.

if (PredCnt[PtIdx] == 0) {  
 mortonCurPt = Morton[AttrPos[PtIdx]]  
 for (centre = 0; centre < LodPtCnt[RefLvl] − 1; centre++) {  
 mortonCentre = Morton[AttrPos[LodPtIdx[RefLvl][centre]]]  
 if (mortonCurPt < mortonCentre)  
 break  
 }  
}

The search range for extended inter-level search is specified by the variable *interLodSearchRange*.

if (slice\_attr\_inter\_prediction)  
 interLodSearchRange = attr\_inter\_prediction\_search\_range  
else  
 interLodSearchRange = pred\_inter\_lod\_search\_range

The extended search shall proceed over each index offset 𝑖 of the following, in order: 0, +1, −1, +2, −2, +3 .. *interLodSearchRange*, and −( 3 .. *interLodSearchRange* ).

A predictor candidate shall be evaluated for each valid search index centre + 𝑖 that is within the range specified by *interLodSearchRange* and does not exceed the bounds of the reference detail level.

if (Abs(i) ≤ interLodSearchRange)  
 if (centre + i ≥ 0 && centre + i < LodPtCnt[RefLvl])  
 InsertPredictor(LodPtIdx[RefLvl][centre + i], 0)

#### Intra-level search

The intra-level search evaluates predictor candidates over a span of indexes in the refinement list of the current detail level. Intra-level predictor candidates shall precede the refinement point in the refinement list.

The search range for intra-level search is specified by the variable *intraLodSearchRange*.

if (slice\_attr\_inter\_prediction)  
 intraLodSearchRange = attr\_inter\_prediction\_search\_range  
else  
 intraLodSearchRange = pred\_intra\_lod\_search\_range

A predictor candidate shall be evaluated for each valid search index offset − 𝑖 from the refinement point, for 𝑖 = 1 .. *intraLodSearchRange*, that does not exceed the bounds of the refinement list.

for (i = 1; i ≤ Min(RfmtIdx, intraLodSearchRange); i++)  
 InsertPredictor(LodRfmtPtIdx[Lvl][RfmtIdx − i], 0)

#### Initial inter-frame predictor search

The initial inter-frame search shall be performed by spatially portioning the reference slice into a lattice of sized cubic blocks. Only blocks adjacent to the block containing the refinement point that are within an availability window shall be searched.

BlkSizeLog2 := lod\_initial\_dist\_log2 + lod\_dist\_log2\_offset + Lvl + 1

The block location ( *bs*, *bt*, *bv* ) identifies the block containing the refinement point.

bs := AttrPos[PtIdx][0] >> BlkSizeLog2  
bt := AttrPos[PtIdx][1] >> BlkSizeLog2  
bv := AttrPos[PtIdx][2] >> BlkSizeLog2

The availability window shall be an 8×8×8 block volume identified by ( *bs* >> 3, *bt* >> 3, *bv* >> 3 ).

The search shall proceed over the search blocks in the order specified by Table 30. Within each search block, points shall be searched in ascending order of index within the reference slice.

for (si = 0; si < 27; si++) {  
 ss = bs + searchBlkOffsets[si][0]  
 st = bt + searchBlkOffsets[si][1]  
 sv = bv + searchBlkOffsets[si][2]  
 unavailable = (ss ^ bs) >> 3 || (st ^ bt) >> 3 || (sv ^ bv) >> 3  
 if (unavailable)  
 continue  
  
 for (i = 0; i < RefLodPtCnt; i++) {  
 candPtIdx = RefLodPtIdx[i]  
 cs = RefAttrPos [candPtIdx][0]  
 ct = RefAttrPos [candPtIdx][1]  
 cv = RefAttrPos [candPtIdx][2]  
  
 inSblk = cs ≥ (ss << BlkSizeLog2) && cs < (ss + 1 << BlkSizeLog2)  
 inSblk &= ct ≥ (st << BlkSizeLog2) && ct < (st + 1 << BlkSizeLog2)  
 inSblk &= cv ≥ (sv << BlkSizeLog2) && cv < (sv + 1 << BlkSizeLog2)  
  
 if (inSblk)  
 InsertPredictor(candPtIdx, 1)  
 }  
}

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

#### Extended inter-frame search

The inter-frame search evaluates predictor candidates over a span of indexes in the finest detail level of the reference slice.

The span shall be centred around the index centreRef in the finest detail level of the reference slice. It shall be the index of the first point with Morton-coded attribute coordinates greater than or equal to those of the current point.

mortonCurPt = Morton[AttrPos[PtIdx]]  
for (centreRef = 0; centreRef < RefLodPtCnt − 1; centreRef++) {  
 mortonCentre = Morton[RefAttrPos[RefLodPtIdx[centreRef]]]  
 if (mortonCurPt <= mortonCentre)  
 break  
}

The search range for intra-level search is specified by the variable *interFrameSearchRange*.

interFrameSearchRange := (slice\_biprediction && slice\_attr\_inter\_prediction && slice\_attr\_inter\_prediction2) ? attr\_inter\_prediction\_search\_range >> 1:  
attr\_inter\_prediction\_search\_range

The inter-frame search shall proceed over each index offset 𝑖 of the following, in order: 0, +1, −1, +2, −2, +3 .. *interFrameSearchRange*, and −( 3 .. *interFrameSearchRange* ).

A predictor candidate shall be evaluated for each valid search index centreRef + 𝑖 that is within the range specified by *interFrameSearchRange* and does not exceed the bounds of the finest detail level of the reference slice.

if (Abs(i) ≤ interFrameSearchRange)  
 if (centreRef + i ≥ 0 && centreRef + i < RefLodPtCnt)  
 InsertPredictor(RefLodPtIdx[centreRef + i], 1)

#### Predictor set pruning and generation of prediction weights

After the predictor search for a refinement point is complete, its predictor set shall be pruned, weights computed for each qualifying predictor and the predictors ordered according to weight.

The size of the predictor set shall be limited to pred\_set\_size\_minus1 + 1 elements by discarding the furthest predictors if necessary.

PredCnt[PtIdx] = Min(pred\_set\_size\_minus1 + 1, PredCnt[PtIdx])

When cross\_attr\_prediction\_enabled\_this\_type is 0, predictor weights shall be calculated using the biased squared distance between each predictor and the current point.

for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 dist[ni] = BiasedNorm2(PtIdx, PredPtIdx[PtIdx][ni], 0, PredPtRef[PtIdx][ni])  
 + PredPtRef[PtIdx][ni] ? 1 : 0

If the first predictor is spatially coincident with the current point, all other predictors shall be discarded.

if (dist[0] == 0)  
 PredCnt[PtIdx] = 1

When lod\_scalability\_enabled is 1, predictors with an unbiased squared distance greater than a threshold shall be discarded [Ed. (YX): The inter prediction case shall be defined.]:.

if (lod\_scalability\_enabled) {  
 threshold = 3 × (pred\_max\_range\_minus1 + 1) << 2 × Lvl  
 for (ni = 1; ni < PredCnt[PtIdx]; ni++)  
 if (Norm2(ptIdx, PredPtIdx[PtIdx][ni], 0, 0) > threshold) {  
 PredCnt[PtIdx] = ni  
 break  
 }  
}

When cross\_attr\_prediction\_enabled\_this\_type is 1, predictor weights shall be calculated using the overall distance between each predictor and the current point. [Ed. (YX): The inter prediction case shall be defined.]

for (ni = 0; ni < PredCnt[PtIdx]; ni++  
 dist[ni] = overAllDist[ni]

The overall distance is a weighted combination of the geometry distance and the attribute distance and shall be stored in array overAllDist[ni], ni = 0 .. PredCnt[ PtIdx ] − 1. The geometry distance is defined as the spatial distance which shall be calculated as the biased norm weighted by PredBias. The attribute distance is defined as the sum of the absolute difference in attribute value for each component.

overAllDist[ni] = geomDis + attrWeight \* attrDis  
 where  
 geomDis = BiasedNorm1(PtIdx, PredPtIdx[PtIdx][ni], 0, 0)  
 attrDis = 0  
 for(i = 0; i < attr\_components\_minus1[refAttrIdx] + 1; i++)  
 attrDis += Abs(RecCloudAttr[ ptIdx ][ refAttrIdx][i] -   
 RecCloudAttr[PredPtIdx[PtIdx][ni]][ refAttrIdx][i])

The attribute weight *attrWeight* which is defined by the weight of attribute distance shall be determined as:

attrWeight = lambda \* maxGeom / maxAttr  
 where  
 lambda = attr\_label[refAttrIdx] ? (4011 – 67 \* (attr\_primary\_qp\_minus4 + 4)) >> 15 :   
 (1939 – 33 \* (attr\_primary\_qp\_minus4 + 4)) >> 12  
 for (k = 0; k < 3; j++)  
 maxGeom += Bbox.max[k] – Bbox.min[k]  
 maxAttr = (attr\_components\_minus1[refAttrIdx] + 1) \*   
 1 << (attr\_bitdepth\_minus1[refAttrIdx] + 1)

where *maxGeom* represents the summation of the length, width and height of the slice bounding box and *maxAttr* represents the possible maximum value of the encoded attribute that is used for decoding the current attribute.

If the first predictor is same as the current point, all other predictors shall be discarded.

if (dist[0] == 0)  
 PredCnt[PtIdx] = 1

When cross\_attr\_prediction\_enabled\_this\_type is 1, the predictors shall be reordered in ascending order according to their overall distance to the current point; when cross\_attr\_prediction\_enabled\_this\_type is 0, the predictors shall be reordered according to their biased squared distance to the current point:

* An array order shall have elements such that dist[ order[ 𝑖 ] ], for 𝑖 = 0 .. PredCnt[ PtIdx ] − 1, is an ascending stable sorting of the array dist.
* The members of the predictor set and the dist array shall be permuted according to the elements of the array order.

The predictor distances shall be normalized by the smallest distance to produce initial weights.

n = Max(0, IntLog2(dist[0] − 8))  
for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 weight[ni] = DivExp2Up(dist[ni], n)

Any predictors with a weight 256 times greater than or equal to the smallest weight shall be discarded.

if (PredCnt[PtIdx] == 3 && weight[2] ≥ 256 × weight[0])  
 PredCnt[PtIdx] = 2

if (PredCnt[PtIdx] == 2 && weight[1] ≥ 256 × weight[0])  
 PredCnt[PtIdx] = 1

The final weights shall be derived as:

if (PredCnt[PtIdx] == 1)  
 PredWeight[PtIdx][0] = 256

if (PredCnt[PtIdx] == 2) {  
 PredWeight[PtIdx][1] = Div(weight[0], weight[0] + weight[1], 8)  
 PredWeight[PtIdx][0] = 256 − PredWeight[PtIdx][1]  
}

if (PredCnt[PtIdx] == 3) {  
 d1d2 = weight[1] × weight[2]  
 d0d2 = weight[0] × weight[2]  
 d0d1 = weight[0] × weight[1]  
 sum = d1d2 + d0d2 + d0d1  
 PredWeight[PtIdx][2] = Div(d0d1, sum, 8)  
 PredWeight[PtIdx][1] = Div(d0d2, sum, 8)  
 PredWeight[PtIdx][0] = 256 − PredWeight[PtIdx][1] − PredWeight[PtIdx][2]  
}

#### Blending of predictor weights

When a point has three predictors in its predictor set and pred\_blending\_enabled is 1, the predictor weights shall be blended according to the distance between the predicting points.

The squared distance between each of the three predictors shall be determined:

distA := Norm2(PredPtIdx[PtIdx][0], PredPtIdx[PtIdx][1], PredPtRef[PtIdx][0],   
PredPtRef[PtIdx][1])  
distB := Norm2(PredPtIdx[PtIdx][0], PredPtIdx[PtIdx][2], PredPtRef[PtIdx][0],   
PredPtRef[PtIdx][2])  
distC := Norm2(PredPtIdx[PtIdx][1], PredPtIdx[PtIdx][2], PredPtRef[PtIdx][1],   
PredPtRef[PtIdx][2])

Blending weights shall be selected according to distance:

b1 := distA ≤ distB ? 1 : 5  
b2 := distA ≤ distC ? 5 : 1  
b3 := distB ≤ distC ? 1 : 5

The predictor weights shall be blended and updated:

if (PredCnt[PtIdx] == 3 && pred\_blending\_enabled) {  
 w0 = PredWeight[PtIdx][0]  
 w1 = PredWeight[PtIdx][1]  
 w2 = PredWeight[PtIdx][2]  
  
 w0p = w0 × 10 + w1 × (6 − b2) + w2 × b3 >> 4  
 w1p = w0 × b1 + w2 × (6 − b3) + w1 × 10 >> 4  
  
 PredWeight[PtIdx][0] = w0p  
 PredWeight[PtIdx][1] = w1p  
 PredWeight[PtIdx][2] = 256 − w0p − w1p  
}

#### Distance computation using the biased L2 norm (BiasedNorm2)

This subclause defines the function BiasedNorm2( ptIdxA, ptIdxB  , *pcA*, *pcB*) that is the weighted squared distance between two points.

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression BiasedNorm2. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance: when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level. [Ed. (YX): The inter prediction case shall be defined.]

BiasedNorm2(ptIdxA, ptIdxB, pcA, pcB) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k]) × PredBias[k]  
 dist2[k] := dist[k] × dist[k]  
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

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

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

When *pcA* is equal to 0, the parameter, the parameters ptIdxA is an AttrPos index. When *pcA* is equal to 1, the parameter, the parameters ptIdxA is a RefAttrPos index.

When *pcB* is equal to 0, the parameter, the parameters ptIdxB is an AttrPos index. When *pcB* is equal to 1, the parameter, the parameters ptIdxB is a RefAttrPos index.

The result of this function is specified by the expression Norm2. The expression posA[ ptIdx ][ 𝑘 ] and posB[ ptIdx ][ 𝑘 ] represent the attribute coordinates used to calculate the distance. when lod\_scalability\_enabled is 1, coordinates shall be quantized according to the detail level.[Ed. (YX): The inter prediction case shall be defined.]

Norm2(ptIdxA, ptIdxB, pcA, pcB) := dist2[0] + dist2[1] + dist2[2]  
 where  
 dist[k] := Abs(posA[ptIdxA][k] − posB[ptIdxB][k])  
 dist2[k] := dist[k] × dist[k]  
 posA[ptIdx][k] := pcA ?  
 RefAttrPos[ptIdxA][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxA][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxA][k])  
 posB[ptIdx][k] := pcB ?  
 RefAttrPos[ptIdxB][k]  
 : (lod\_scalability\_enabled  
 ? (AttrPos[ptIdxB][k] >> Lvl) << Lvl  
 : AttrPos[ptIdxB][k])

#### Reduction of predictor set

[Ed. (YX): The inter prediction case shall be defined.]

Subclause 10.6.6.15 applies when prediction\_with\_distribution\_enabledis 1.

When the size of the predictor set is larger than 3, the third point in the predictor set may be replaced by one of the following points in the predictor set according to their relative spatial locations.

The array PtDirection represents the relative direction of a point in the predictor set with respect to the current point.

The space around the current point is separated into eight octants indexed from 0 to 7 along the three axes using the current point as the origin. The relative direction of a point in the predictor set with respect to the current point is defined as the index of the octant in which the point is located.

for (j = 0; j < PredCnt[PtIdx];j++)  
 for (k = 0; k < 3; ++k) {  
 dist = ((AttrPos[PtIdx][k] - AttrPos[PredPtIdx[PtIdx][j]][k])  
 PtDirection[j] |= (dist >= 0) << k;  
}

Strict opposite octants and loose opposite octants are defined as strictOpposite and looseOpposite, respectively, where strictOpposite[ 𝑖 ][ 𝑘 ] (or looseOpposite[ 𝑖 ][ 𝑘 ]) denote whether the k-th octant is a strict (or loose) opposite octant of the i-th octant.

Table 43— Strictly opposite octants, strictOpposite[ 𝑖 ][ 𝑘 ]

| 𝑖 | 𝑘 | | | | | | | | 𝑖 | 𝑘 | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **4** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | **5** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **6** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **7** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 44— Loose opposite octants, looseOpposite[ 𝑖 ][ 𝑘 ]

| 𝑖 | 𝑘 | | | | | | | | 𝑖 | 𝑘 | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **0** | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | **4** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| **1** | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | **5** | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **2** | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | **6** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| **3** | 1 | 0 | 0 | 0 |  | 1 | 1 | 0 | **7** | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

The first three points in the predictor set are referred to as initial predictor points.

The variables equal01, equal02 and equal12 indicate whether the pairs of initial predictor points are in the same octant.

equal01 = PtDirection[0] == PtDirection[1]  
equal02 = PtDirection[0] == PtDirection[2]  
equal12 = PtDirection[1] == PtDirection[2]

The predictor set needs no replacement if any two points of the initial predictor points are in strict opposite octants of each other.

Otherwise, the third point in the predictor set may be replaced by a candidate predictor point. A distance threshold dist1 is derived below that is used in the determination of replacement of the third point.

dist1 = (BiasedNorm1(PtIdx, PredPtIdx[PtIdx][2]) × (pred\_replace\_weight\_minus32 + 32)) >> 5

The points in the predictor set that have index value 3 or more, and whose distance to the current point is less than dist1 are referred as candidate predictor points of the predictor set.If a candidate predictor point is at the strict opposite direction of the first or the second initial predictor point, that point is used to replace the third point in the predictor set.

for (i = 3; i < PredCnt[PtIdx]; i++){  
 if (dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i])   
 && (strictOpposite[i][0] || strictOpposite[i][1])){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
}

If the third point in the predictor set is not replaced by the above process, and if the all the initial predictor points are in the same octant p, the third point in the predictor set is be replaced by a candidate point that is in loose opposite direction of octant p as below.

if(equal01 && (equal02 || equal12))  
 for (i = 3; i < PredCnt[PtIdx]; i++){  
 if (dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i]) && looseOpposite[i][0]){  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
 }

If the third point of the predictor set is not replaced by the above process, the replacement process continues. If the first and second initial predictors are not in the same octant and not loose opposite octants, and if the third initial predictor is in the same octant as the first or the second predictor, a candidate predictor point that is not in the same octant as the first or the second initial predictor, that candidate predictor point is used to replace the third point in the predictor set.

if(¬equal01 && (equal02 || equal12) && ¬looseOpposite[0][1])  
 for (i = 3; i < PredCnt[PtIdx]; i++){  
 if(dist1 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i])   
 && PtDirection[i] ≠ PtDirection[0]  
 && PtDirection[i] ≠ PtDirection[1])  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
}

If the third point in the predictor set is not replaced by the above process, the replacement process continues. If the first and second initial predictors are in the same octant and the third initial predictor is not in loose opposite octant as the first predictor, a candidate predictor point that is at the loose opposition direction to the first initial predictor is used to replace the third point in the predictor set.

if(equal01 && ¬looseOpposite[0][2])  
 for (i = 3; i < PredCnt[PtIdx] ; i++){  
 if(dist2 > BiasedNorm1(PtIdx, PredPtIdx[PtIdx][i] && looseOpposite[i][0])  
 PredPtIdx[PtIdx][2] = PredPtIdx[PtIdx][i]  
 break  
 }  
}

After the above processes, the size of the predictor set shall be reduced to three elements by discarding the furthest predictor.

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

### Reconstruction of attribute values

#### General process

Each detail level shall be processed in turn, proceeding from the coarsest to the finest level, according to attr\_coding\_type (10.6.7.3, 10.6.7.4). The variable Lvl is the index of the current detail level.

for (Lvl = LodCnt − 1; Lvl ≥ 0; Lvl−−)  
 … /\* process a detail level \*/

#### Coefficient processing order within a detail level

Within a detail level, processing proceeds in coded coefficient order. The variable PtIdx is the AttrPos index of the current coefficient. The variable CoeffIdx is the AttrCoeff array index of the current coefficient.

for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[Lvl]; rfmtIdx++) {  
 PtIdx = LodRfmtPtIdx[Lvl][rfmtIdx]  
 CoeffIdx = LodPtCnt[Lvl] − rfmtIdx  
 … /\* process current coefficient \*/  
}

#### Processing a detail level (attr\_coding\_type = 1)

When attr\_coding\_type is 1, the following operations shall be performed in turn for each coefficient in the coefficient processing order of the current detail level:

* Prediction mode information is decoded from an encoded coefficient tuple (10.6.8.1). The result is the variable PredMode.
* The unencoded coefficient tuple is scaled (10.6.9.1) to produce transform coefficients.
* Transform coefficient components are divided by 256 with half-values rounded up.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = DivExp2Up(PointAttr[PtIdx][c], 8)

* Transform coefficient are scaled by the point quantization weights (10.6.13.3).
* Transform coefficient components are predicted using inter-component prediction (10.6.10.2) to form prediction residuals.
* The attribute value is predicted and combined with the prediction residual (10.6.12).
* The reconstructed attribute value is clipped.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = Clip3(0, AttrMaxVal, PointAttr[PtIdx][c])

#### Processing a detail level (attr\_coding\_type = 2)

When attr\_coding\_type is 2, the following operations shall be performed in turn, each over all the coefficients in the detail level:

* Coefficient tuples are scaled (10.6.9.1) to produce transform coefficients.
* Transform coefficient components are predicted using last-component prediction (10.6.10.3).
* Transform coefficients are weighted by transform coefficient weights (10.6.11.4).

If Lvl is less than LodCnt − 1, the transform shall be applied (10.6.12):

* Attribute values predicted from the coarser detail level, Lvl + 1, are modified by the transform update operator (10.6.12.1).
* Attribute values corresponding to coefficients in the current detail level are predicted and combined with the scaled transform coefficient to produce the detail level output (10.6.12.1).

When Lvl is 0, the reconstructed attributes values shall be divided by 256 with half-values rounded away from zero and clipped to the maximum attribute value:

if (Lvl == 0)  
 for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[ptIdx][c] = Clip3(0, AttrMaxVal, DivExp2Fz(PointAttr[ptIdx][c], 8))

### Prediction mode coding

#### General

Subclause 10.6.8 specifies the conditional coding of the prediction mode PredMode in coefficient tuples. It applies when pred\_direct\_max\_idx\_plus1 is greater than zero; when pred\_direct\_max\_idx\_plus1 is 0, PredMode shall be 0.

A per-transform-coefficient test (10.6.8.2) shall be performed to determine whether the coefficient tuple encodes a prediction mode. The result of the test is the variable PredModePresent.

If PredModePresent is:

* false, PredMode shall be 0;
* true, the coded prediction mode (PredModeCoded) shall be decoded according to the number of attribute components (10.6.8.3, 10.6.8.4) and the maximum codable prediction mode PredModeMax. As a side-effect of decoding the prediction mode, the coefficient tuple (in AttrCoeff) is updated.

PredModeMax := pred\_direct\_max\_idx\_plus1 + ¬pred\_direct\_avg\_disabled

The prediction mode shall be derived from the coded prediction mode:

PredMode = PredModePresent ? PredModeCoded + pred\_direct\_avg\_disabled : 0

#### Presence of an encoded direct prediction mode

The derivation of PredModePresent specifies the presence of an encoded direct prediction mode:

* A direct prediction mode shall not be coded when disabled, or for refinement points with fewer than two predictors.

if (PredCnt[PtIdx] < 2 || pred\_direct\_max\_idx\_plus1 == 0)  
 PredModePresent = 0

* Otherwise, a prediction mode shall be coded for a refinement point if, for any component, the absolute difference in attribute value between any of its predictors exceeds a bit-depth adjusted threshold.

for (ni = 0; ni < PredCnt[PtIdx]; ni++) {  
 ptIdx = PredPtIdx[PtIdx][ni]  
 for (c = 0; c < AttrDim; c++) {  
 minVal[c] = ni ? Min(minVal[c], PointAttr[ptIdx][c]) : PointAttr[ptIdx][c]  
 maxVal[c] = ni ? Max(maxVal[c], PointAttr[ptIdx][c]) : PointAttr[ptIdx][c]  
 }  
}

maxDiff = 0  
for (c = 0; c < AttrDim; c++)  
 maxDiff = Max(maxDiff, maxVal[c] − minVal[c])

threshold = pred\_direct\_threshold << Max(0, AttrBitDepth − 8)  
PredModePresent = maxDiff ≥ threshold

#### Decoding process for single component attributes

For single component attributes (AttrDim == 1), the prediction mode PredModeCoded is encoded by the LSBs of the coefficient magnitude:

PredModeCoded = 0  
absCoeff = Abs(AttrCoeff[CoeffIdx][0])

if (PredModeMax == 4){  
 PredModeCoded = absCoeff & 3  
 absCoeff >>= 2  
}

if (PredModeMax == 3) {  
 PredModeCoded = absCoeff & 1  
 absCoeff >>= 1  
 if (PredModeCoded){  
 PredModeCoded += absCoeff & 1  
 absCoeff >>= 1  
 }  
}

if (PredModeMax == 2){  
 PredModeCoded = absCoeff & 1  
 absCoeff >>= 1  
}

After decoding the prediction mode, the coefficients shall be updated.

AttrCoeff[CoeffIdx][0] = Sign(AttrCoeff[CoeffIdx][0]) × absCoeff

#### Decoding process for multi-component attributes

For multi-component attributes (AttrDim > 1), the prediction mode PredModeCoded is encoded by the LSB of the last two component's coefficient magnitude:

PredModeCoded = 0  
absCoeffA = Abs(AttrCoeff[CoeffIdx][AttrDim − 2])  
absCoeffB = Abs(AttrCoeff[CoeffIdx][AttrDim − 1])

if (PredModeMax == 4) {  
 PredModeCoded = ((absCoeffA & 1) << 1) + (absCoeffB & 1)  
 absCoeffA >>= 1  
 absCoeffB >>= 1  
}

if (PredModeMax == 3) {  
 PredModeCoded = absCoeffA & 1  
 absCoeffA >>= 1  
 if (PredModeCoded) {  
 PredModeCoded += absCoeffB & 1  
 absCoeffB >>= 1  
 }

if (PredModeMax == 2) {  
 PredModeCoded = absCoeffA & 1  
 absCoeffA >>= 1  
}

After decoding the prediction mode, the coefficients shall be updated.

sgnCoeffA = Sign(AttrCoeff[CoeffIdx][AttrDim − 2])  
sgnCoeffB = Sign(AttrCoeff[CoeffIdx][AttrDim − 1])

AttrCoeff[CoeffIdx][AttrDim − 2] = sgnCoeffA × absCoeffA  
AttrCoeff[CoeffIdx][AttrDim − 1] = sgnCoeffB × absCoeffB

### Scaling

#### Derivation of per-point QP

The QP for a point depends upon the detail level and its attribute coordinates as specified by the expression LodCoeffQp[ qc ] for a QP component qc:

* rgnOffset[ qc ] is the per-coefficient offset from the region-dependent QP offset tree.
* dpth is the depth of detail level in the LoD hierarchy.

LodCoeffQp[qc] := AttrQp[dpth][rgnOffset][Cidx > 0]  
 where  
 s := AttrPos[PtIdx][0]  
 t := AttrPos[PtIdx][1]  
 v := AttrPos[PtIdx][2]  
 dpth := LodCnt − 1 − Lvl  
 rgnOffset[qc] := AttrRegionQpOffset[s][t][v][qc]

#### Scaling by quantization step size

The coefficient tuple shall be scaled by the quantization step size (10.7.4) for the primary and secondary attribute components.

for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = AttrCoeff[CoeffIdx][c] × AttrQstep[LodCoeffQp[c > 0]]

### Coefficient prediction

#### Syntax element semantics

last\_comp\_pred\_coeff\_diff[ dpth ] specifies in accordance with LastCompPredCoeff[ dpth ] the two-fractional-bit, fixed-point scale factor applied at depth dpth of the LoD hierarchy to second coefficient components to predict third coefficient components. The syntax element codes the scale factor relative to LastCompPredCoeffPrev[ dpth ].

LastCompPredCoeff[dpth] := last\_comp\_pred\_enabled  
 ? LastCompPredCoeffPrev[dpth] + last\_comp\_pred\_coeff\_diff[dpth]  
 : 0

LastCompPredCoeffPrev[dpth] := dpth == 0 ? 4 : LastCompPredCoeff[dpth − 1]

It is a requirement of bitstream conformance that LastCompPredCoeff[ dpth ] shall be in the range −128 .. 127 for dpth ∈ 0 .. lod\_max\_levels\_minus1.

inter\_comp\_pred\_coeff\_diff[ dpth ][ 𝑐 ] specifies in accordance with InterCompPredCoeff[ dpth ][ 𝑐 ] the two-fractional-bit, fixed-point scale factor applied at depth dpth of the LoD hierarchy to first coefficient components to predict 𝑐-th coefficient components. The syntax element codes the scale factor relative to InterCompPredCoeffPrev[ dpth ][ 𝑐 ].

InterCompPredCoeff[dpth][c] := inter\_comp\_pred\_enabled  
 ? predCoeff + inter\_comp\_pred\_coeff\_diff[dpth][c]  
 : 0

InterCompPredCoeffPrev[dpth][c] := dpth == 0 ? 4 : InterCompPredCoeff[dpth − 1][c]

It is a requirement of bitstream conformance that InterCompPredCoeff[ dpth ][ 𝑐 ] shall be in the range −128 .. 127 for dpth ∈ 0 .. lod\_max\_levels\_minus1.

#### Inter-component prediction

When attr\_coding\_type is 1 and inter\_comp\_pred\_enabled is 1, secondary attribute coefficient components are residuals to a prediction by the first scaled coefficient component. The predicted value shall round the two fractional bits from the scale factor, with half-values rounded up.

for (c = 1; c < AttrDim; c++) {  
 icpCoeff = InterCompPredCoeff[LodCnt − 1 − Lvl][c]  
 PointAttr[PtIdx][c] += DivExp2Up(icpCoeff × PointAttr[PtIdx][0], 2)  
}

#### Last component prediction

When attr\_coding\_type is 2 and last\_comp\_pred\_enabled is 1, the third attribute coefficient component is, if present, a residual to a prediction by the second scaled coefficient component. The predicted value shall round the two fractional bits from the scale factor towards negative infinity.

if (AttrDim == 3) {  
 lcpCoeff = LastCompPredCoeff[LodCnt − 1 − Lvl]  
 PointAttr[PtIdx][2] += DivExp2Floor(lcpCoeff × PointAttr[PtIdx][1], 2)  
}

### Transform coefficient weights

#### General

Coefficient weights represent the relative significance of a coefficient. Coefficients with larger weights have a greater influence on the decoded attribute values.

The array CoeffWeight is initialized by setting all elements to 256.

for (i = 0; i < PointCnt; i++)  
 CoeffWeight[i] = 256

The derivation of coefficient weights depends upon whether LoD scalability is enabled.

#### Non-scalable case

When lod\_scalability\_enabled is 0, coefficient weights are calculated accumulatively, proceeding from the finest to the coarsest detail level.

The accumulated coefficient weight of each refinement point in a detail level shall be distributed to the points in its predictor set. The distribution is proportional to the respective predictor weights:

for (lvl = 0; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = CoeffWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 predW = PredWeight[ptIdx][ni]  
 CoeffWeight[PredPtIdx[ptIdx][ni]] += DivExp2Up(coeffW × predW, 8)  
 }  
 }  
 }

#### Scalable case

When lod\_scalability\_enabled is 1, a single weight shall be assigned to all refinement points within a detail level:

for (lvl = 1; lvl < LodCnt − 1; lvl++) {  
 weight = (slice\_num\_points\_minus1 + 1) / LodPtCnt[lvl]  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++)  
 CoeffWeight[LodRfmtPtIdx[lvl][rfmtIdx]] = weight × 256  
}

#### Application to coefficient scaling

Transform coefficients shall be scaled by the integer reciprocal square root of their coefficient weight and divided by with half-values rounded away from zero.

weight = IntRecipSqrt(CoeffWeight[PtIdx])  
for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] = DivExp2Fz(PointAttr[PtIdx][c] × weight, 36)

### Transform

#### Update operation

When attr\_coding\_type is 2, the transform update operator shall redistribute coefficient values to predicting points in the coarser detail level.

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 updateN[ptIdx] = updateD[ptIdx] = 0

for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[Lvl]; rfmtIdx++) {  
 rfmtPtIdx = LodRfmtPtIdx[Lvl][rfmtIdx]  
 coeffW = CoeffWeight[rfmtPtIdx]  
 for (ni = 0; ni < PredCnt[rfmtPtIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 nPtIdx = PredPtIdx[rfmtPtIdx][ni]  
 nWeight = DivExp2Up(PredWeight[rfmtPtIdx][ni] × coeffW, 8)  
 updateD[nPtIdx] += nWeight  
 for (c = 0; c < AttrDim; c++)  
 updateN[nPtIdx][c] += nWeight × PointAttr[rfmtPtIdx][c]  
 }  
 }  
}

for (ptIdx = 0; ptIdx < PointCnt; ptIdx++)  
 if (updateD[ptIdx])  
 PointAttr[ptIdx] −= Div(updateN[ptIdx], updateD[ptIdx], 0)

#### Direct prediction

When attr\_coding\_type is 1 and PredMode for a refinement point is greater than zero, its value shall be predicted to be the same as the point with predictor set index PredMode − 1. If the indicated predictor is invalid, prediction shall not be performed.

It is a requirement of bitstream conformance that PredMode shall be less than or equal to PredCnt[ CoeffIdx ].

if (PredMode && PredMode ≤ PredCnt[PtIdx])  
 for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] += PredPtRef[PtIdx][PredMode - 1] ?   
 RefPointAttr[PredPtIdx[PtIdx][PredMode − 1]][c] :   
 PointAttr[PredPtIdx[PtIdx][PredMode − 1]][c]

#### Average prediction

When attr\_coding\_type is 2 or PredMode for a refinement point is 0, the weighted average of the predictor set shall predict the value of the refinement point:

if (PredMode == 0)  
 for (c = 0; c < AttrDim; c++) {  
 sum = 0  
 for (ni = 0; ni < PredCnt[PtIdx]; ni++)  
 sum += PredWeight[PtIdx][ni] × PredPtRef[PtIdx][ni] ?   
 RefPointAttr[PredPtIdx[PtIdx][ni]][c] :   
 PointAttr[PredPtIdx[PtIdx][ni]][c]  
 PointAttr[PtIdx][c] += DivExp2Fz(sum, 8)  
 }

### Point quantization weights

#### General

Point quantization weights represent the predictive relationships between points. Points with larger quantization weights have a greater influence on the prediction process.

The array QuantWeight is initialized by setting all elements to 256.

for (i = 0; i < PointCnt; i++)  
 QuantWeight[i] = 256

#### Quantization weights derivation

Quantization weights are calculated accumulatively, proceeding from the finest to the coarsest detail level. The quantization weight of each point is updated by the points in its predictor set. The weights of the points in the predictor set to update the quantization weights are specified by quant\_neigh\_weight:

for (lvl = 0; lvl < LodCnt − 1; lvl++)  
 for (rfmtIdx = 0; rfmtIdx < LodRfmtPtCnt[lvl]; rfmtIdx++) {  
 ptIdx = LodRfmtPtIdx[lvl][rfmtIdx]  
 coeffW = QuantWeight[ptIdx]  
 for (ni = 0; ni < PredCnt[ptIdx]; ni++) {  
 if (!PredPtRef[ptIdx][ni]){  
 QuantWeight[PredPtIdx[ptIdx][ni]] +=  
 DivExp2Inf(coeffW × quant\_neigh\_weight[ni], 8)  
 }  
 }  
 }

#### Application to coefficient scaling

Transform coefficients shall be scaled by quantization step size (10.7.4) or quantization weights.

weight = Min(QuantWeight[PtIdx], AttrQstep[qp]) >> 8  
for (c = 0; c < AttrDim; c++)  
 PointAttr[PtIdx][c] /= weight

## Attribute quantization parameters

### Syntax element semantics

attr\_qp\_offset[ qc ] specifies per-slice offsets used to derive QPs for the primary (qc = 0) and any secondary (qc = 1) attribute components. When attr\_qp\_offset[ qc ] is not present, it shall be inferred to be 0.

attr\_qp\_layers\_present specifies whether (when 1) or not (when 0) per-transform-layer QP offsets are present in the ADU.

attr\_qp\_layer\_cnt\_minus1 plus 1 specifies, when present, the number of levels in the LoD hierarchy or RAHT tree for which QP offsets are signalled.

attr\_qp\_layer\_offset[ dpth ][ qc ] specifies QP offsets used for the primary (qc = 0) and any secondary (qc = 1) attribute components. Each offset applies to transform coefficients at depth dpth of the LoD hierarchy or RAHT tree. If the LoD hierarchy or RAHT tree has a greater number of levels than attr\_qp\_layer\_cnt\_minus1 + 1, attr\_qp\_layer\_offset[ attr\_qp\_layer\_cnt\_minus1 ][ qc ] also specifies the QP offsets for transform coefficients at a depth greater than attr\_qp\_layer\_cnt\_minus1.

The expression AttrQpLayerOffset[ dpth ][ qc ] specifies the per layer QP offsets at depth dpth of the LoD hierarchy or RAHT tree.

AttrQpLayerOffset[dpth][qc] := attr\_qp\_layers\_present > 0  
 ? attr\_qp\_layer\_offset[Min(attr\_qp\_layer\_cnt\_minus1, dpth)][qc]  
 : 0

attr\_qp\_region\_cnt specifies the number of spatial regions within the slice that have a region QP offset signalled.

In profiles specified in this version of this document, all but the first region are ignored.

attr\_qp\_region\_bits\_minus1 plus 1 specifies the length in bits of each syntax element attr\_qp\_region\_origin\_xyz, attr\_qp\_region\_size\_minus1\_xyz, attr\_qp\_region\_origin\_rpi and attr\_qp\_region\_size\_minus1\_rpi.

attr\_qp\_region\_origin\_xyz[ 𝑖 ][ 𝑘 ] and attr\_qp\_region\_size\_minus1\_xyz[ 𝑖 ][ 𝑘 ] specify, when present, the 𝑖-th spatial region in the slice where attr\_qp\_region\_offset[ 𝑖 ][ qc ] applies. The region is a bounding box in the slice coordinate system with lower corner XYZ coordinates attr\_qp\_region\_origin\_xyz[ 𝑖 ][ 𝑘 ] and dimensions attr\_qp\_region\_size\_minus1\_xyz[ 𝑖 ][ 𝑘 ] + 1.

attr\_qp\_region\_origin\_rpi[ 𝑖 ][ 𝑘 ] and attr\_qp\_region\_size\_minus1\_rpi[ 𝑖 ][ 𝑘 ] specify, when present, the 𝑖-th spatial region in the slice where attr\_qp\_region\_offset[ 𝑖 ][ qc ] applies. The region is a bounding box in the scaled angular coordinate system (10.2.2) used for attribute coding with lower corner RPI coordinates attr\_qp\_region\_origin\_rpi[ 𝑖 ][ 𝑘 ] and dimensions attr\_qp\_region\_size\_minus1\_rpi[ 𝑖 ][ 𝑘 ] + 1.

The expressions AttrRegionQpOrigin[ 𝑖 ][ 𝑘 ] and AttrRegionQpSize[ 𝑖 ][ 𝑘 ] specify the 𝑘-th component of the bounding box origin and size for the 𝑖-th QP region in attribute coordinates.

AttrRegionQpOrigin[i][k] = attr\_coord\_conv\_enabled  
 ? attr\_qp\_region\_origin\_rpi[i][k]  
 : attr\_qp\_region\_origin\_xyz[i][StvToXyz[k]]

AttrRegionQpSize[i][k] = attr\_coord\_conv\_enabled  
 ? attr\_qp\_region\_size\_minus1\_rpi[i][k] + 1  
 : attr\_qp\_region\_size\_minus1\_xyz[i][StvToXyz[k]] + 1

A constraint on the bounds of a region is specified by expression AttrRegionSizeConstraint[ 𝑖 ][ 𝑘 ]. It is a requirement of bitstream conformance that AttrRegionSizeConstraint[ 𝑖 ][ 𝑘 ] shall be true for every component 𝑘 of each region 𝑖.

AttrRegionSizeConstraint[i][k] :=  
 AttrRegionQpOrigin[i][k] + AttrRegionQpSize[i][k] < Exp2(MaxSliceDimLog2)

attr\_qp\_region\_offset[ 𝑖 ][ qc ] specifies offsets used to derive the QPs for the primary (qc = 0) and any secondary (qc = 1) attribute components of points positioned within the region defined by AttrRegionQpOrigin[ 𝑖 ] and AttrRegionQpSize[ 𝑖 ]. When attr\_qp\_region\_offset[ 𝑖 ][ qc ] is not present, it shall be inferred to be 0.

attr\_AC\_qp\_offset\_present specifies whether (when 1) or not (when 0) per-RAHT-layer AC transform coefficient QP offsets are present in the ADU. When attr\_AC\_qp\_offset\_present is not present, it shall be inferred to be 0.

attr\_AC\_qp\_layer\_cnt\_minus1 plus 1 specifies, when present, the number of levels in the RAHT tree for which AC QP offsets are present in the ADU. When attr\_AC\_qp\_layer\_cnt\_minus1 is not present, it shall be inferred to be -1.

attr\_AC\_qp\_offset[ dpth ][ qc][ACcompidx] specifies the per AC coefficient (ACcompidx = 0,1,..6) QP offsets for the primary (qc = 0) and any secondary (qc = 1) attribute components at depth dpth of the RAHT tree.

### Per-point regional QP offset

The region-dependent QP offset for a point with attribute coordinates ( 𝑠, 𝑡, 𝑣 ) is specified by the expression AttrRegionQpOffset[ 𝑠 ][ 𝑡 ][ 𝑣 ][ qc ].

AttrRegionQpOffset[s][t][v][qc] :=  
 isPointInRegion[0][s][t][v] ? attr\_qp\_region\_offset[0][qc] : 0

The expression isPointInRegion[ rgnIdx ][ 𝑠 ][ 𝑡 ][ 𝑣 ] specifies whether the coordinates ( 𝑠, 𝑡, 𝑣 ) are within the rgnIdx-th region.

isPointInRegion[rgnIdx][s][t][v] :=  
 rIdx < attr\_qp\_region\_cnt  
 && s ≥ AttrRegionQpOrigin[rIdx][0] && s < regionEnd[0]  
 && t ≥ AttrRegionQpOrigin[rIdx][1] && t < regionEnd[1]  
 && v ≥ AttrRegionQpOrigin[rIdx][2] && v < regionEnd[2]  
 where  
 regionEnd[k] := AttrRegionQpOrigin[rIdx][k] + AttrRegionQpSize[rIdx][k]

### Attribute coefficient QP

An attribute coefficient QP is specified by the expression AttrQp[ dpth ][ rgnOffset ][ qc ], parameterized by:

* dpth, the depth of a level in the LoD hierarchy or RAHT tree;
* rgnOffset, an expression that when applied to an argument qc is a region-dependent QP offset; and
* qc, indicating a primary or secondary QP component.

The expressions qpP and qpS are the QPs for the primary and secondary attribute components. Attribute QPs shall be clipped to the bit-depth dependent range [ 4, qpMax ].

AttrQp[dpth][rgnOffset][qc] := qc == 0 ? qpP : qpS  
 where  
 qpMax := 51 + 6 × (AttrBitDepth − 8)  
 qpP := Clip3(4, qpMax, attr\_primary\_qp\_minus4 + 4 + qpOffset[qc])  
 qpS := Clip3(4, qpMax, qpP + attr\_secondary\_qp\_offset + qpOffset[qc])  
 qpOffset[qc] := attr\_qp\_offset[qc] + AttrQpLayerOffset[dpth][qc] + rgnOffset[qc]

### Definition of AttrQstep

This subclause specifies the expression AttrQstep[ qp ] that is the attribute scale factor for the attribute quantization parameter qp.

AttrQstep[qp] := AttrLevelScale[qp % 6] << (qp / 6)

Values of AttrLevelScale are specified by Table 31.

Table 45 — Values of AttrLevelScale[ 𝑖 ]

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

## Reference slice generation for attribute

### General

Subclause 10.8 applies when attr\_inter\_prediction\_enabled is equal to 1.

After the decoding of a frame, subclause 10.8.3 applies to generate the reference frame for the attribute coding of the next point cloud frame to be decoded.

After the generation of point coordinates of the current slice as specified by subclause 10.2, subclause 10.8.4 applies to generate the reference slice.

The output of this process are the reference slice attribute and the reference slice geometry for attribute coding. The expression *RefPointAttr*[*ptIdx*][*c*] specifies the attribute values of each point in the reference slice. The expression *RefAttrPos*[*ptIdx*][*k*] specifies the coordinates of each point for attribute coding in the reference slice.

### Reconstructed frame generation

The coordinates for attribute coding of the previously reconstructed point cloud frame are stored in the array *CloudAttrPos*[*ptIdx*][*k*]. *AttrPos*[*ptIdx*][*k*] is an alias into the geometry coordinates array for attribute coding for the points in the slice.

AttrPos[ptIdx][k] :=   
CloudAttrPos[RecCloudPointCnt + ptIdx][k]

The coordinates of points for attribute coding shall be generated as specified by Subclause 10.2.

[Ed. (AR): Explain the process of generation of reference frame used for attributes, including the case of predictive geometry coding+inter.]

### Reference frame generation

The geometry coordinates for attribute coding of the reference frame are stored in the array *RefCloudAttrPos*[*ptIdx*][*k*]. The attributes of the reference frame are stored in the array *RefCloudAttr*[*ptIdx*][*k*]. The number of points in the reference frame is specified by the variable *refCloudPointCnt*.

* When biprediction\_enabled is 0, subclause 10.8.3.1 applies to generate the reference frame for uni-prediction.
* Otherwise, subclause 10.8.3.2 applies to generate the reference frame for bi-prediction.

#### Reference frame for uni-prediction

When geom\_tree\_type is 0, global\_motion\_enabled is equal to 0 or attr\_coord\_conv\_enabled is equal to 1, *RefCloudAttrPos* is set equal to *CloudAttrPos*, and *RefCloudAttr* is set equal to *RecCloudAttr*. When geom\_tree\_type is 0, global\_motion\_enabled is equal to 1 and attr\_coord\_conv\_enabled is equal to 0, *RefCloudAttrPos* is set equal to *GMPointCloud* (*GMPointCloud* is derived by invoking subclause 9.2.15.2) and *RefCloudAttr* is set equal to *RecCloudAttr*.

When geom\_tree\_type is 1 and attr\_coord\_conv\_enabled is equal to 1, *RefCloudAttrPos* and *RecCloudAttr* are derived as follows:

ptIdx = 0  
for(beamId = 0; beamId <= num\_beams\_minus1; beamId++)  
 for(qAzim = MinQAzim; qAzim <= MaxQAzim; qAzim++)  
 for(j = 0; j <= maxPointsPerEntryMinus1; j++)  
 if (PtnCurrFramePos[beamId][qAzim][j][0] ¬= -1) {  
 for(k = 0; k < 3; k ++)  
 RefCloudAttrPos[ptIdx][k] = PtnCurrFramePos[beamId][qAzim][j][k]  
 RefCloudAttr[ptIdx][0] = PtnCurrFramePos[beamId][qAzim][j][0]  
 ptIdx++  
 }  
refCloudPointCnt = ptIdx

#### Reference frame for bi-prediction

When biprediction\_enabled is 1 or 2, the geometry coordinates for attribute coding *CloudAttrPos* and the reconstructed attributes *RecCloudAttr* are indicated by the notional frame counter to be used as the reference frame for subsequent point cloud frames.

The geometry coordinates for attribute coding of the first original reference frame are stored in the array *RefCloudAttrPosFirst*[*ptIdx*][*k*]. The attributes of the first original reference original frame are stored in the array *RefCloudAttrFirst*[*ptIdx*][*k*]. The number of points in the first original reference frame is specifies by the variable *refCloudPointCntFirst*. The geometry coordinates for attribute coding of the second original reference frame are stored in the array *RefCloudAttrPosSecond*[*ptIdx*][*k*]. The attributes of the second original reference frame are stored in the array *RefCloudAttrSecond*[*ptIdx*][*k*]. The number of points in the second original reference frame is specifies by the variable *refCloudPointCntSecond*.

The two original reference frames are determined as follows:

* when slice\_biprediction is 0, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame;
* when biprediction\_enabled is 1 and slice\_biprediction is 1, *RefCloudAttrPosSecond* and *RefCloudAttrSecond* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame.
  + If the previously reconstructed point cloud frame is a B-frame, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed B-frame;
  + otherwise, *RefCloudAttrPosFirst* and *RefCloudAttrFirst* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the previously reconstructed I-frame or P-frame.
* when biprediction\_enabled is 2 and slice\_biprediction is 1, *RefCloudAttrPosFirst*, *RefCloudAttrFirst*, *RefCloudAttrPosSecond* and *RefCloudAttrSecond* are set to the geometry coordinates for attribute coding and the reconstructed attributes of the reference frames determined by invoking subclause 9.2.15.3.

*RefCloudAttrPos* and *RefCloudAttr* are determined as follows:

* when slice\_biprediction is 1, slice\_attr\_inter\_prediction2 is 1 and slice\_attr\_inter\_prediction is 0, *RefCloudAttrPos* is set equal to *RefCloudAttrPosSecond*, and *RefCloudAttr* is set equal to *RefCloudAttrSecond*;
* when slice\_biprediction is 1, slice\_attr\_inter\_prediction2 is 1 and slice\_attr\_inter\_prediction is 1, the reference frame is set equal to the fusion of the two original reference frames:

refCloudPointCnt = refCloudPointCntFirst + refCloudPointCntSecond  
for (i = 0; i < refCloudPointCntFirst; i++){  
 for (c = 0; c < AttrDim; c++)  
 RefCloudAttr[i][c] = RefCloudAttrFirst[i][c]  
 for (k = 0; k < 3; k++)  
 RefCloudAttrPos[i][k] = RefCloudAttrPosFirst[i][k]  
}  
for (i = 0; i < refCloudPointCntSecond; i++){  
 for (c = 0; c < AttrDim; c++)  
 RefCloudAttr[i + refCloudPointCntFirst][c] = RefCloudAttrSecond[i][c]  
 for (k = 0; k < 3; k++)  
 RefCloudAttrPos[i + refCloudPointCntFirst][k] = RefCloudAttrPosSecond[i][k]  
}

* otherwise, *RefCloudAttrPos* is set equal to *RefCloudAttrPosFirst*, and *RefCloudAttr* is set equal to *RefCloudAttrFirst*.

### Reference slice generation

The attribute values and coordinates of the reference slice are derived based on the bounding box of the coordinates of the current slice. The number of points in the reference slice is specified by the variable *refPointCnt*.

refPointCnt = 0  
for (i = 0; i < refCloudPointCnt; i++)  
 if (ptInBox[i]){  
 for (c = 0; c < AttrDim; c++)  
 RefPointAttr[refPointCnt][c] = RefCloudAttr[i][c]  
 for (k = 0; k < 3; k++)  
 RefAttrPos[refPointCnt][k] = RefCloudAttrPos[i][k]  
 refPointCnt++  
 }  
where,  
 ptInBox[i] = pos[0] < bMax[0] && pos[0] > bMin[0] && pos[1] < bMax[1]   
 && pos[1] > bMin[1] && pos[2] < bMax[2] && pos[2] > bMin[2]  
 where,  
 pos = RefCloudAttrPos[i]  
 bMax = Bbox.max  
 bMin = Bbox.min