

Figure 7-38 — Temporal Prediction Structure

7.6.8 Vector decoding process of non-scalable progressive B-VOPs

In B-VOPs there are three kinds of vectors, namely, 16x16 forward vector, 16x16 backward vector and the delta vector for the direct mode. The vectors are decoded with respect to the corresponding vector predictors. The basic decoding process of a differential vector is the exactly same as defined in P-VOPs except that for the delta vector of the direct mode the f_code is always one. The vector is then reconstructed by adding the decoded differential vector to the corresponding vector predictor. The vector predictor for the delta vector is always set to zero, while the forward and backward vectors have their own vector predictors, which are reset to zero only at the beginning of each macroblock row. The vector predictors are updated in the following three cases:

- after decoding a macroblock of forward mode only the forward predictor is set to the decoded forward vector
- after decoding a macroblock of backward mode only the backward predictor is set to the decoded backward vector
- after decoding a macroblock of bi-directional mode both the forward and backward predictors are updated separately with the decoded vectors of the same type (forward/backward).

7.6.9 Motion compensation in non-scalable progressive B-VOPs

In B-VOPs the overlapped motion compensation (OBMC) is not employed. The motion-compensated prediction of B-macroblock is generated by using the decoded vectors and taking reference to the padded forward/backward reference VOPs as defined below. Arbitrarily shaped reference VOPs shall be padded accordingly.

7.6.9.1 Basic motion compensation procedure

All of the ISO/IEC 14496-2 motion compensation techniques are based on the formation of a prediction block, pred[i][j] of dimension (width, height), from a reference image, ref[x][y]. The coordinates of the current block (or macroblock) in the reference VOP is (x,y), the motion half-pel resolution motion vector is (dx_halfpel, dy_halfpel). The pseudo-code for this procedure is given below. For quarter pel mode the formation of the prediction block is carried out as described in subclause 7.6.2.2.

The component_width() and component_height() function give the coded VOP dimensions for the current component. For luminance, component_width() is video_object_layer_width for a rectangular VOP or vop_width otherwise rounded up to the next multiple of 16. The luminance component_height() is defined similarly. The chrominance dimensions are one half of the corresponding luminance dimension.

© ISO/IEC 2004 – All rights reserved
Google
Exhibit 1042

```
clip_ref(ref, x, y)
{
    return(ref[MIN(MAX(x, 0), component_width(ref) - 1)]
              [MIN(MAX(y, 0), component_height(ref) - 1)]);
}
mc(pred,                        /* prediction block */
   ref,                         /* reference component */
   x, y,                        /* ref block coords for MV=(0, 0) */
   width, height,               /* reference block dimensions */
   dx_halfpel, dy_halfpel,      /* half-pel resolution motion vector */
   rounding,                    /* rounding control (0 or 1 ) */
   pred_y0,                     /* field offset in pred blk (0 or 1) */
   ref_y0,                      /* field offset in ref blk (0 or 1) */
   y_incr)                      /* vertical increment (1 or 2) */
{
    dx = dx_halfpel >> 1;
    dy = y_incr * (dy_halfpel >> y_incr);
    if (dy_halfpel & y_incr) {
        if (dx_halfpel & 1) {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[ix][iy + pred_y0] =
                        (clip_ref(ref, x_ref + 0, y_ref + 0) +
                         clip_ref(ref, x_ref + 1, y_ref + 0) +
                         clip_ref(ref, x_ref + 0, y_ref + y_incr) +
                         clip_ref(ref, x_ref + 1, y_ref + y_incr) +
                         2 - rounding) >> 2;
                }
            }
        } else {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[ix][iy + pred_y0] =
                        (clip_ref(ref, x_ref, y_ref + 0) +
                         clip_ref(ref, x_ref, y_ref + y_incr) +
                         1 - rounding) >> 1;
                }
            }
        }
    } else {
        if (dx_halfpel & 1) {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[ix][iy + pred_y0] =
                        (clip_ref(ref, x_ref + 0, y_ref) +
                         clip_ref(ref, x_ref + 1, y_ref) +
                         1 - rounding) >> 1;
                }
            }
        } else {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[ix][iy + pred_y0] =
                        clip_ref(ref, x_ref, y_ref);
                }
            }
        }
    }
}
```

© ISO/IEC 2004 – All rights reserved

7.6.9.2 Forward mode

Only the forward vector (MVFx,MVFy) is applied in this mode. The prediction blocks Pf_Y, Pf_U, and Pf_V are generated from the forward reference VOP, ref_Y_for for luminance component and ref_U_for and ref_V_for for chrominance components, as follows:

```
mc(Pf_Y, ref_Y_for, x, y, 16, 16, MVFx, MVFy, 0, 0, 0, 1);

mc(Pf_U, ref_U_for, x/2, y/2, 8, 8,  MVFx_chro, MVFy_chro, 0, 0, 0,1);

mc(Pf_V, ref_V_for, x/2, y/2, 8, 8,  MVFx_chro, MVFy_chro, 0, 0, 0,1);
```

where (MVFx_chro, MVFy_chro) is motion vector derived from the luminance motion vector by dividing each component by 2 then rounding on a basis of Table 7-13. Here (and hereafter) the function MC is defined in subclause 7.6.9.

7.6.9.3 Backward mode

Only the backward vector (MVBx,MVBy) is applied in this mode. The prediction blocks Pb_Y, Pb_U, and Pb_V are generated from the backward reference VOP, ref_Y_back for luminance component and ref_U_back and ref_V_back for chrominance components, as follows:

```
mc(Pb_Y, ref_Y_back, x, y, 16, 16, MVBx, MVBy, 0, 0, 0, 1);

mc(Pb_U, ref_U_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);

mc(Pb_V, ref_V_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);
```

where (MVBx_chro, MVBy_chro) is motion vector derived from the luminance motion vector by dividing each component by 2 then rounding on a basis of Table 7-13.

7.6.9.4 Bi-directional mode

Both the forward vector (MVFx,MVFy) and the backward vector (MVBx,MVBy) are applied in this mode. The prediction blocks Pi_Y, Pi_U, and Pi_V are generated from the forward and backward reference VOPs by doing the forward prediction, the backward prediction and then averaging both predictions pixel by pixel as follows.

```
mc(Pf_Y, ref_Y_for, x, y, 16, 16, MVFx, MVFy, 0, 0, 0, 1);

mc(Pf_U, ref_U_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);

mc(Pf_V, ref_V_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);

mc(Pb_Y, ref_Y_back, x, y, 16, 16, MVBx, MVBy, 0, 0, 0, 1);

mc(Pb_U, ref_U_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);

mc(Pb_V, ref_V_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);

Pi_Y[i][j] = (Pf_Y[i][j] + Pb_Y[i][j] + 1)>>1;          i,j=0,1,2…15;

Pi_U[i][j] = (Pf_U[i][j] + Pb_U[i][j] + 1)>>1;      i,j=0,1,2…7;

Pi_V[i][j] = (Pf_V[i][j] + Pb_V[i][j] + 1)>>1;          i,j=0,1,2…7;
```

where (MVFx_chro, MVFy_chro) and (MVBx_chro, MVBy_chro) are motion vectors derived from the forward and backward luminance motion vectors by dividing each component by 2 then rounding on a basis of Table 7-13, respectively.

© ISO/IEC 2004 – All rights reserved

7.6.9.5 Direct mode

This mode uses direct bi-directional motion compensation derived by employing I-, P-, or S(GMC)-VOP macroblock motion vectors and scaling them to derive forward and backward motion vectors for macroblocks in B-VOP. This is the only mode which makes it possible to use motion vectors on 8x8 blocks in B-VOPs. Only one delta motion vector is allowed per macroblock.

7.6.9.5.1 Formation of motion vectors for the direct mode

The direct mode utilises the motion vectors (MVs) of the co-located macroblock in the most recently I-, P-, or S(GMC)- VOP. The co-located macroblock is defined as the macroblock which has the same horizontal and vertical index with the current macroblock in the B-VOP. The averaged motion vector defined in subclause 7.8.7.3 is used as the MV of the co-located macroblock when this co-located macroblock is included in an S(GMC)-VOP and mcsel == '1' (i.e. when the co-located macroblock is a GMC macroblock). The MV vectors are the block vectors of the co-located macroblock after applying the vector padding defined in subclause 7.6.1.6. If the co-located macroblock is transparent and thus the MVs are not available, the direct mode is still enabled by setting MV vectors to zero vectors.

7.6.9.5.2 Calculation of vectors

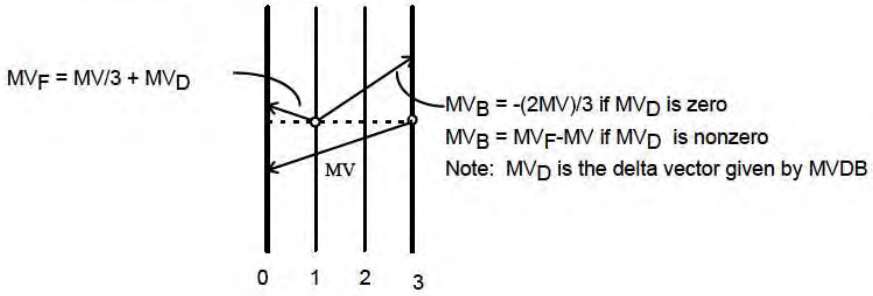


Figure 7-39 — Direct Bi-directional Prediction

Figure 7-39 shows scaling of motion vectors. The calculation of forward and backward motion vectors involves linear scaling of the collocated block in temporally next I-, P-, or S(GMC)-VOP, followed by correction by a delta vector (MVDx,MVDy). The forward and the backward motion vectors are {(MVFx[i],MVFy[i]), (MVBx[i],MVBy[i]), i = 0,1,2,3} and are given in half or quarter sample units as follows.

MVFx[i] = (TRB x MVx[i]) / TRD + MVDx

MVBx[i] = (MVDx==0)? ((TRB - TRD) x MVx[i]) / TRD : MVFx[i] - MVx[i]

MVFy[i] = (TRB x MVy[i]) / TRD + MVDy

MVBy[i] = (MVDy==0)? ((TRB - TRD) x MVy[i]) / TRD : MVFy[i] - MVy[i]

i = 0,1,2,3.

where {(MVx[i],MVy[i]), i = 0,1,2,3} are the MV vectors of the co-located macroblock, TRD is the difference in temporal reference of the B-VOP and the previous reference VOP. TRD is the difference in temporal reference of the temporally next reference VOP with temporally previous reference VOP, assuming B-VOPs or skipped VOPs in between. In case that the MV components of the co-located macroblock are given in quarter sample units and the components MVDx and MVDy of the delta vector are given in half sample units, the components of the co-located macroblock {(MVx[i],MVy[i]), i = 0,1,2,3} are converted to half sample units before the calculation of the forward and the backward motion vectors {(MVFx[i], MVFy[i]), (MVBx[i], MVBy[i]), i = 0,1,2,3}. For this conversion, each component {(MVx[i],MVy[i]), i = 0,1,2,3} is first divided by 2 and then rounded on the basis of Table 7-13.

© ISO/IEC 2004 – All rights reserved

7.6.9.5.3 Generation of prediction blocks

Motion compensation for luminance is performed individually on 8x8 blocks to generate a macroblock. The process of generating a prediction block consists of using computed forward and backward motion vectors {(MVFx[i],MVFy[i]), (MVBx[i],MVBy[i]), i = 0,1,2,3} to obtain appropriate blocks from reference VOPs and averaging these blocks, same as the case of bi-directional mode except that motion compensation is performed on 8x8 blocks. Note that in the case of quarter sample mode, the motion compensation of four 8x8 blocks, using the same motion vector for each block, does not lead to the same result as the motion compensation of a single 16x16 block using the same vector. This is due to the block boundary mirroring described in subclause 7.6.2.2.

For the motion compensation of both chrominance blocks, the forward motion vector (MVFx_chro, MVFy_chro) is calculated by the sum of K forward luminance motion vectors dividing by 2K and then rounding toward the nearest half sample position as defined in Table 7-10 to Table 7-13. In quarter sample mode the vectors are divided by 2 before summation. The backward motion vector (MVBx_chro, MVBy_chro) is derived in the same way. The rest process is the same as the chrominance motion compensation of the bi-directional mode described in subclause 7.6.9.4.

7.6.9.6 Motion compensation in skipped macroblocks

If the co-located macroblock in the most recently decoded I- or P-VOP is skipped, the current B-macroblock is treated as the forward mode with the zero motion vector (MVFx,MVFy). If the modb equals to '1' the current B-macroblock is reconstructed by using the direct mode with zero delta vector. If the co-located macroblock in the most recently decoded S(GMC)-VOP is skipped, this co-located macroblock is treated as a non-skipped macroblock with the averaged motion vector defined in subclause 7.8.7.3 for the current B-macroblock.

7.6.10 Motion Compensation Decoding of Reduced Resolution VOP

This subclause describes motion compensation decoding process for VOP with vop_reduced_resolution equal to "1". In this case, motion compensation is done with Macroblock size of 32 x 32 pixels or Block size of 16 x 16 pixels instead of Macroblock size of 16 x 16 pixels or Block size of 8 x 8 pixels for the case of normal resolution VOP(vop_reduced_resolution is set to "0" or vop_redcuced_resolution flag is not present). Figure 7-40 shows block diagram of motion compensation decoding process of Reduced Resolution VOP.

© ISO/IEC 2004 – All rights reserved

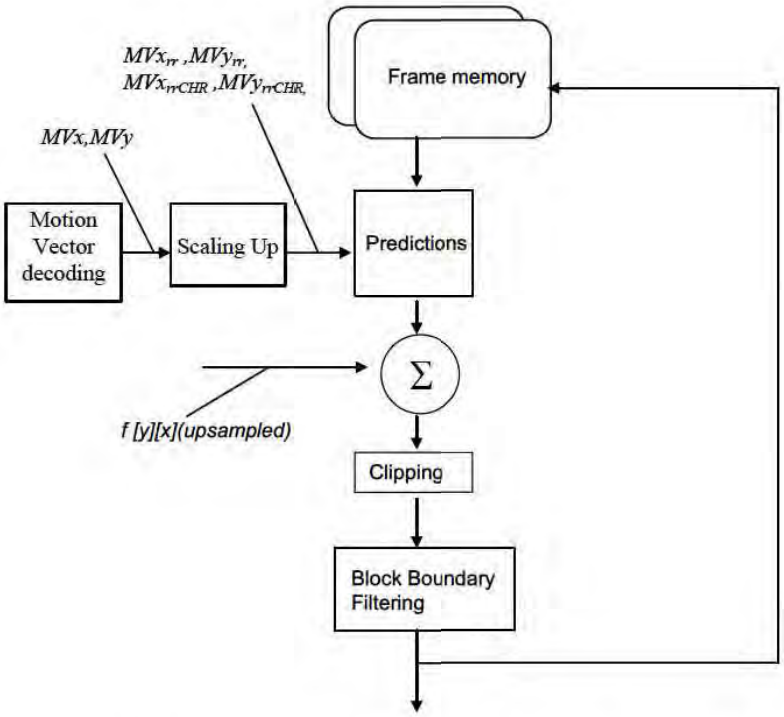


Figure 7-40 — Motion Compensation Decoding of Reduced Resolution VOP

7.6.10.1 Decoding Process of Reduced Resolution P-VOP

7.6.10.1.1 Motion Vector Decoding

Motion vector for luminance component *MVx, MVy* is decoded as in the same way as described in subclause 7.6.3. Four vector case, motion vector for each block is also decoded as described in 7.6.5. These decoded vectors for luminance are used for the candidate of the motion vector predictor in the decoding process of the following Macroblocks.

7.6.10.1.2 Scaling up of Motion Vectors

Decoded luminance motion vector *MVx, MVy* shall be scaled up to MVx_{rr}, MVy_{rr} by the following formula:

$$MVx_{rr} = 0 \qquad \text{if } MVx = 0$$

$$MVx_{rr} = \text{Sign}(MVx) * (2.0 * \text{Abs}(MVx) - 0.5) \qquad \text{if } MVx \neq 0$$

$$MVy_{rr} = 0 \qquad \text{if } MVy = 0$$

$$MVy_{rr} = \text{Sign}(MVy) * (2.0 * \text{Abs}(MVy) - 0.5) \qquad \text{if } MVy \neq 0$$

As a result, each vector component is restricted to have a half-integer or zero value, and the range of each motion vector component MVx_{rr}, MVy_{rr} is enlarged to approximately twice the range of decoded motion vector component *MVx, MVy*.

Motion vector for chrominace blocks MVx_{rrCHR}, MVy_{rrCHR} is derived by calculating the sum of the K luminance vectors MVx_{rr}, MVy_{rr}, that corresponds to K 16x16 blocks that do not lie outside the VOP shape and dividing this sum by 2*K; the component values of the resulting sixteenth/twelfth/eighth/fourth sample resolution vectors are modified towards the nearest half sample position as indicated in Table 7-10 to Table 7-13.

© ISO/IEC 2004 – All rights reserved

7.6.10.1.3 Prediction

By using the scaled up luminance motion vector MVx_{rr}, MVy_{rr} and derived chrominance vector MVx_{rrCHR}, MVy_{rrCHR} predicton data for each 32x32 Macroblock or 16x16 block is generated from the refrence VOP data stored in frame memory. Half sample interpolation shall be done as described in subclause 7.6.2.

If Overlapped Motion Compensation is also used for the Reduced Resolution VOP, enlarged matrices of weighting values are used to perform the overlapped motion compensation. Except that the size of each block and matrices of weighting values is 16×16, the procedure of the creation of each prediction block is identical to the description of subclause. 7.6.6.

The enlarged matrices of weighting values for the 16×16 luminance prediction are given in Figure 7-40, Figure 7-41, and Figure 7-42.

4	4	5	5	5	5	5	5	5	5	5	5	5	5	4	4
4	4	5	5	5	5	5	5	5	5	5	5	5	5	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	6	6	6	6	6	6	6	6	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
4	4	5	5	5	5	5	5	5	5	5	5	5	5	4	4
4	4	5	5	5	5	5	5	5	5	5	5	5	5	4	4

Figure 7-41 — Weighting values, H_0, for prediction with motion vector of current 16×16 luminance block

© ISO/IEC 2004 – All rights reserved

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
1	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Figure 7-42 — Weighting values, H_1, for prediction with motion vector of 16×16 luminance blocks on top or bottom of current 16×16 luminance block

© ISO/IEC 2004 – All rights reserved

2	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2
2	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2
2	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2
2	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2

Figure 7-43 — Weighting values, H_2, for prediction with motion vector of 16×16 luminance blocks to the left or right of current 16×16 luminance block

7.6.10.1.4 Addition and Clipping

The 16 x 16 reconstructed prediction error block data derived by upsampling process described in subclause 7.4.6 is added to the 16 x 16 prediction block data generated by prediction process defined in subclause 7.6.10.1.3. All the resulted 16 x 16 reconstructed block data shall be clipped to the range of [0: $2^{bits\_per\_pixel} - 1$].

7.6.10.1.5 Block Boundary Filtering

The filter operation described in this clause shall be performed along the edges of the 16×16 reconstructed blocks data.

The filtering is performed on the complete reconstructed VOP data before storing the data in the frame memory for future prediction. No filtering is performed across VOP edges and NEWPRED segment (defined in subclause 7.14.1) edges. Chrominance as well as luminance data is filtered.

If A and B are two pixel values on a line — horizontal or vertical — of the reconstructed VOP, and A belongs to one 16 × 16 block called block1 whereas B belongs to a neighboring 16 × 16 block called block2 which is to the right or below of block1. Figure 7-44 shows examples for the position of these pixels.

© ISO/IEC 2004 – All rights reserved

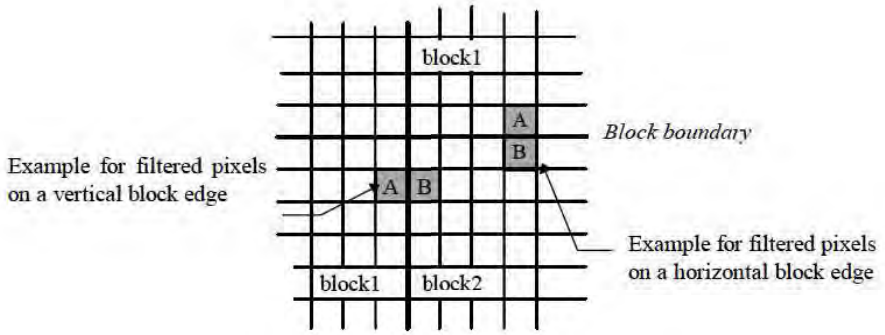


Figure 7-44 — Block boundary filter

One of the following conditions must be fulfilled in order to turn the filter on for a particular edge:

block1 belongs to a coded macroblock (not_coded == 0 || Macroblock type == INTRA) or

block2 belongs to a coded macroblock (not_coded == 0 || Macroblock type == INTRA)

A shall be replaced by A1 and B shall be replaced by B1.

A1 = (3*A + B + 2) / 4

B1 = (A + 3*B + 2) / 4

Filtering across horizontal edges: Basically this process is assumed to take place first. More precisely, the pixels $\begin{pmatrix} A \\ B \end{pmatrix}$ that are used in filtering across a horizontal edge shall not have been influenced by previous filtering across a vertical edge.

Filtering across vertical edges:

Before filtering across a vertical edge using pixels (A,B), all modifications of pixels (A,B) resulting from filtering across a horizontal edge shall have taken place.

7.6.10.2 Decoding Process of Reduced Resolution I-VOP

For I-VOP with the vop_reduced_resolution equal to "1", Addition and Clipping process described in subclause 7.6.10.1.4 shall be performed with the fixed prediction data value of "0". Block Boundary Filtering operation described in subclause 7.6.10.1.5 also shall be applied. The reconstructed I-VOP data is stored in the frame memory for future prediction.

7.7 Interlaced video decoding

This subclause specifies the additional decoding process that a decoder shall perform to recover VOP data from the coded bitstream when the interlaced flag in the VOL header is set to "1". Interlaced information (subclause 6.3.6.3) specifies the method to decode bitstream of interlaced VOP.

7.7.1 Field DCT and DC and AC Prediction

When dct_type flag is set to '1' (field DCT coding), DCT coefficients of luminance data are formed such that each 8x8 block consists of data from one field as being shown in Figure 6-12. DC and optional AC (see "ac_pred_flag") prediction will be performed for a intra-coded macroblock. For the intra macroblocks which have dct_type flag being set to "1", DC/AC prediction are performed to field blocks shown in Figure 7-45. After taking inverse DCT, all luminance blocks will be inverse permuted back to (frame) macroblock. Chrominance (block) data are not effected by dct_type flag.

© ISO/IEC 2004 – All rights reserved

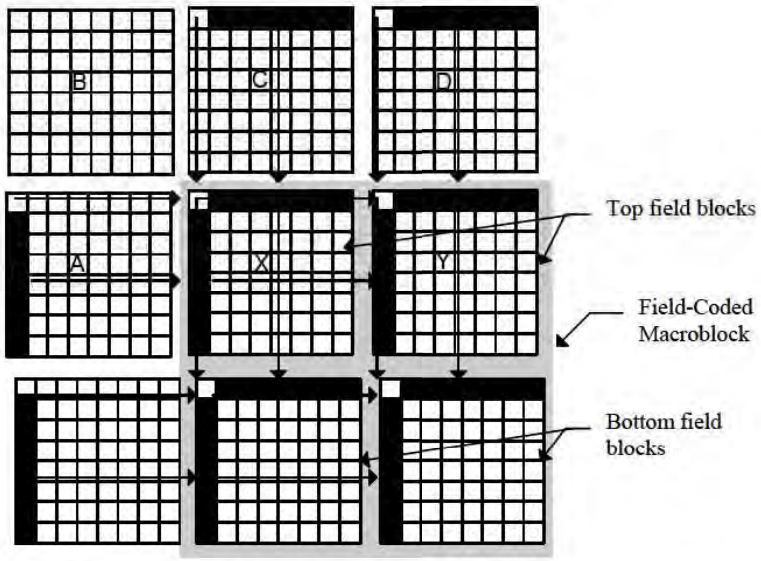


Figure 7-45 — Previous neighboring blocks used in DC/AC prediction for interlaced intra blocks.

7.7.2 Motion compensation

For non-intra macroblocks in P- and B-, and S(GMC)-VOPs, motion vectors are extracted syntactically following subclause 6.2.6 "Macroblock". The motion vector decoding is performed separately on the horizontal and vertical components.

7.7.2.1 Motion vector decoding in P- and S(GMC)-VOP

For each component of motion vector in P- and S(GMC)-VOPs, the median value of the candidate predictor vectors for the same component is computed and add to corresponding component of the motion vector difference obtained from the bitstream. To decode the motion vectors in a P- and S(GMC)-VOP, the decoder shall first extract the differential motion vectors ($(MVDx_{f1}, MVDy_{f1})$ and $(MVDx_{f2}, MVDy_{f2})$ for top and bottom fields of a field predicted macroblock, respectively) by a use of variable length decoding and then determine the predictor vector from three candidate vectors. These candidate predictor vectors are generated from the three motion vectors of three spatial neighborhood decoded macroblocks or blocks as follows. Note that the averaged motion vector as defined in subclause 7.8.7.3 is used as the candidate predictor vector for macroblocks with mcsel == '1' when the current VOP is an S(GMC)-VOP. Such macroblock is referred to as a frame predicted macroblock in the following.

CASE 1 :

If the current macroblock is a field predicted macroblock and none of the coded spatial neighborhood macroblocks is a field predicted macroblock, then candidate predictor vectors MV1, MV2, and MV3 are defined by Figure 7-46. If the candidate block *i* is not in four MV motion (8x8) mode, MV*i* represents the motion vector for the macroblock. If the candidate block *i* is in four MV motion (8x8) mode, the 8x8 block motion vector closest to the upper left block of the current MB is used. The predictors for the horizontal and vertical components are then computed by

$$P_x = Median(MV1x, MV2x, MV3x)$$
$$P_y = Median(MV1y, MV2y, MV3y).$$

© ISO/IEC 2004 – All rights reserved

For differential motion vectors both fields use the same predictor and motion vectors are recovered by

$$MVx_{f1} = MVDx_{f1} + P_x$$
$$MVy_{f1} = 2 * (MVDy_{f1} + (P_y / 2))$$
$$MVx_{f2} = MVDx_{f2} + P_x$$
$$MVy_{f2} = 2 * (MVDy_{f2} + (P_y / 2))$$

where "/" is integer division with truncation toward 0. Note that all motion vectors described above are specified as integers with one LSB representing a half- or quarter pel displacement. The vertical component of field motion vectors always even (in half- or quarter pel frame coordinates). Vertical half- or quarter pel interpolation between adjacent lines of the same field is denoted by MVy_{fi} be an odd multiple of 2 (e.g. -2,2,6,..). No vertical interpolation is needed when MVy_{fi} is an multiple of 4 for half or 8 for quarter pel (it is a full pel value).

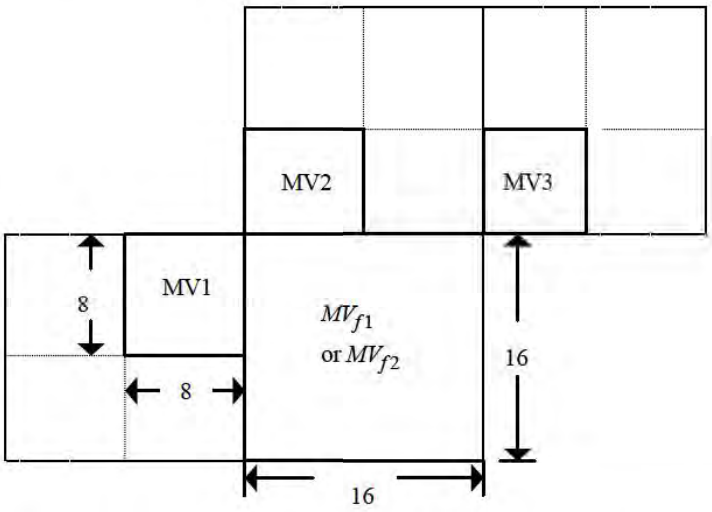


Figure 7-46 — Example of motion vector prediction for field predicted macroblocks (Case1)

CASE 2 :

If the current macroblock or block is frame predicted macroblock or block and if at least one of the coded spatial neighborhood macroblocks is a field predicted macroblock, then the candidate predictor vector for each field predicted macroblock will be generated by averaging two field motion vectors such that all fractional pel offsets are mapped into the half or quarter -pel displacement. Each component (P_x or P_y) of the final predictor vector is the median value of the candidate predictor vectors for the same component. The motion vector is recovered by

$$MVx = MVDx + P_x$$
$$MVy = MVDy + P_y.$$

where

$$P_x = Median\left(MV1x, Div2Round(MVx_{f1} + MVx_{f2}), MV3x\right),$$
$$P_y = Median\left(MV1y, Div2Round(MVy_{f1} + MVy_{f2}), MV3y\right),$$

© ISO/IEC 2004 – All rights reserved

Div2Round(*x*) is defined as follows: *Div2Round*(*x*) = (*x* >> 1) | (*x* & 1).

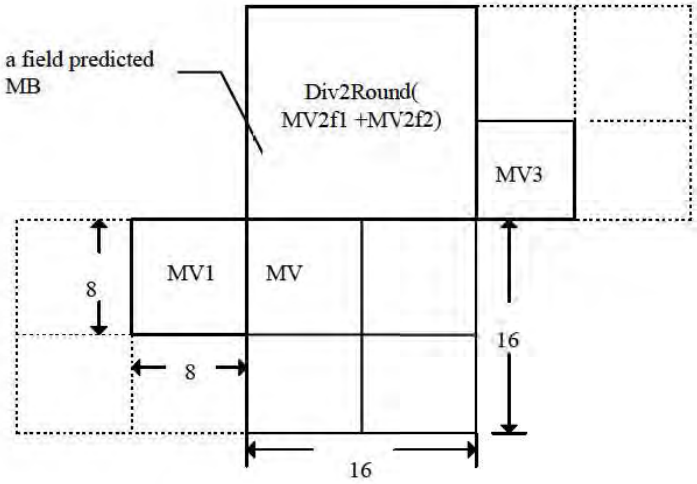


Figure 7-47 — Example of motion vector prediction for field predicted macroblocks (Case 2)

CASE 3 :

This case applies if the current macroblock is a field predicted macroblock and at least one of the coded spatial neighborhood macroblocks is a field predicted macroblock. If the candidate block *i* is field predicted, the candidate predictor vector MV*i* will be generated by averaging two field motion vectors such that all fractional pel offsets are mapped into the half- or quarter pel displacement as discribed in CASE 2. If the candidate block *i* is neither in four MV motion (8x8) mode nor in field prediction mode, MV*i* represents the frame motion vector for the macroblock. If the candidate block *i* is in four MV motion (8x8) mode, the 8x8 block motion vector closest to the upper left block of the current MB is used. The predictors for the horizontal and vertical components are then computed by

$$P_x = Median(MV1x, MV2x, MV3x)$$
$$P_y = Median(MV1y, MV2y, MV3y)$$

where

$$MVi\ x = Div2Round(MVx_{f1} + MVx_{f2}),$$
$$MVi\ y = Div2Round(MVy_{f1} + MVy_{f2}),$$

for some *i* in {1,2,3}.

For differential motion vectors both fields use the same predictor and motion vectors are recovered by (see both Figure 7-46 and Figure 7-47)

$$MVx_{f1} = MVDx_{f1} + P_x$$
$$MVy_{f1} = 2 * (MVDy_{f1} + (P_y / 2))$$
$$MVx_{f2} = MVDx_{f2} + P_x$$
$$MVy_{f2} = 2 * (MVDy_{f2} + (P_y / 2))$$

© ISO/IEC 2004 – All rights reserved

The motion compensated prediction macroblock is calculated calling the "field_compensate_one_reference" using the motion vectors calculated above. In quarter_sample mode the macroblock is calculated as described in subclause 7.6.2.2, accordingly. The top_field_ref, bottom_field_ref, and rounding type come directly from the syntax as forward_top_field_reference, forward_bottom_field_reference and vop_rounding_type respectively. The reference VOP is defined such the even lines (0, 2, 4, ...) are the top field and the odd lines (1, 3, 5, ...) are the bottom field.

```
field_motion_compensate_one_reference(
    luma_pred, cb_pred, cr_pred, /* Prediction component pel array */
    luma_ref, cb_ref, cr_ref,    /* Reference VOP pel arrays */
    mv_top_x, mv_top_y,           /* top field motion vector */
    mv_bot_x, mv_bot_y,          /* bottom field motion vector */
    top_field_ref,               /* top field reference */
    bottom_field_ref,            /* bottom field reference */
    x, y,                        /* current luma macroblock coords */
    rounding_type)               /* rounding type */
{
    mc(luma_pred, luma_ref, x, y, 16, 16, mv_top_x, mv_top_y,
        rounding_type, 0, top_field_ref, 2);
    mc(luma_pred, luma_ref, x, y, 16, 16, mv_bot_x, mv_bot_y,
        rounding_type, 1, bottom_field_ref, 2);
    mc(cb_pred, cb_ref, x/2, y/2, 8, 8,
        Div2Round(mv_top_x), Div2Round(mv_top_y),
        rounding_type, 0, top_field_ref, 2);
    mc(cr_pred, cr_ref, x/2, y/2, 8, 8,
        Div2Round(mv_top_x), Div2Round(mv_top_y),
        rounding_type, 0, top_field_ref, 2);
    mc(cb_pred, cb_ref, x/2, y/2, 8, 8,
        Div2Round(mv_bot_x), Div2Round(mv_bot_y),
        rounding_type, 1, bottom_field_ref, 2);
    mc(cr_pred, cr_ref, x/2, y/2, 8, 8,
        Div2Round(mv_bot_x), Div2Round(mv_bot_y),
        rounding_type, 1, bottom_field_ref, 2);
}
```

In the case that obmc_disable is "0", the OBMC is not applied if the current MB is field-predicted. If the current MB is frame-predicted (including 8x8 mode) and some adjacent MBs are field-predicted, the motion vectors of those field-predicted MBs for OBMC are computed in the same manner as the candidate predictor vectors for field-predicted MBs are.

7.7.2.2 Motion vector decoding in B-VOP

For interlaced B-VOPs, a macroblock can be coded using (1) direct coding, (2) 16x16 motion compensation (includes forward, backward & bidirectional modes), or (3) field motion compensation (includes forward, backward & bidirectional modes). Motion vector in half or quarter sample accuracy will be employed for a 16x16 macroblock being coded. Chrominance vectors are derived by scaling of luminance vectors using the rounding tables described in Table 7-13 (i.e. by applying *Div2Round* to the luminance motion vectors, divided by 2 in case of quarter_sample mode). These coding modes except direct coding mode allow switching of quantiser scale from the one previously in use. Specification of dquant, a differential quantiser scale involves a 2-bit overhead. In direct coding mode, the quantiser scale value for previous coded macroblock is used.

For interlaced B-VOP motion vector predictors, four prediction motion vectors (PMVs) are used:

© ISO/IEC 2004 – All rights reserved

Table 7-14 — Prediction motion vector allocation for interlaced P- and S(GMC)-VOPs

Function	PMV
Top field forward	0
Bottom field forward	1
Top field backward	2
Bottom field backward	3

These PMVs are used as follows for the different macroblock prediction modes:

Table 7-15 — Prediction motion vectors for interlaced B-VOP decoding

Macroblock mode	PMVs used	PMVs updated
Direct	none	none
Frame forward	0	0,1
Frame backward	2	2,3
Frame bidirectional	0,2	0,1,2,3
Field forward	0,1	0,1
Field backward	2,3	2,3
Field bidirectional	0,1,2,3	0,1,2,3

The PMVs used by a macroblock are set to the value of current macroblock motion vectors after being used.

When a frame macroblock is decoded, the two field PMVs (top and bottom field) for each prediction direction are set to the same frame value. The PMVs are reset to zero at the beginning of each row of macroblocks. The predictors are not zeroed by skipped macroblocks or direct mode macroblocks.

The frame based motion compensation modes are described in subclause 7.6. The field motion compensation modes are calculated using the "field_motion_compensate_one_reference()" pseudo code function described above. The field forward mode is denoted by mb_type == "0001" and field_prediction == "1". The PMV update and calculation of the motion compensated prediction is shown below. The luma_fwd_ref_VOP[][], cb_fwd_ref_VOP[][], cr_fwd_ref_VOP[][] denote the entire forward (past) reference VOP pixel arrays. The coordinates of the upper left corner of the luminance macroblock is given by (x, y) and MVD[].x and MVD[].y denote an array of the motion vector differences in the order they occur in the bitstream for the current macroblock.

```
PMV[0].x = PMV[0].x + MVD[0].x;
PMV[0].y = 2 * (PMV[0].y / 2 + MVD[0].y);
PMV[1].x = PMV[1].x + MVD[1].x;
PMV[1].y = 2 * (PMV[1].y / 2 + MVD[1].y);
field_motion_compensate_one_reference(
    luma_pred, cb_pred, cr_pred,
    luma_fwd_ref_vop, cb_fwd_ref_vop, cr_fwd_ref_vop,
    PMV[0].x, PMV[0].y, PMV[1].x, PMV[1].y,
    forward_top_field_reference,
    forward_bottom_field_reference,
    x, y, 0);
```

The field backward mode is denoted by mb_type == "001" and field_prediction == "1". The PMV update and prediction calculation is outlined the following pseudo code. The luma_bak_ref_vop[][], cb_bak_ref_vop[][], cr_bak_ref_vop[][] denote the entire backward (future) reference VOP pixel arrays.

© ISO/IEC 2004 – All rights reserved

```
    PMV[2].x = PMV[2].x + MVD[0].x;
    PMV[2].y = 2 * (PMV[2].y / 2 + MVD[0].y);
    PMV[3].x = PMV[1].x + MVD[1].x;
    PMV[3].y = 2 * (PMV[3].y / 2 + MVD[1].y);
    field_motion_compensate_one_reference(
        luma_pred, cb_pred, cr_pred,
        luma_bak_ref_vop, cb_bak_ref_vop, cr_bak_ref_vop,
        PMV[2].x, PMV[2].y, PMV[3].x, PMV[3].y,
        backward_top_field_reference,
        backward_bottom_field_reference,
        x, y, 0);
```

The bidirectional field prediction is used when mb_type == "01" and field_prediction == "1". The prediction macroblock (in luma_pred[][], cb_pred[][], and cr_pred[][]) is calculated by:

```
for (mv = 0; mv < 4; mv++) {
        PMV[mv].x = PMV[mv].x + MVD[mv].x;
        PMV[mv].y = 2 * (PMV[mv].y / 2 + MVD[mv].y);
    }
    field_motion_compensate_one_reference(
        luma_pred_fwd, cb_pred_fwd, cr_pred_fwd,
        luma_fwd_ref_vop, cb_fwd_ref_vop, cr_fwd_ref_vop,
        PMV[0].x, PMV[0].y, PMV[1].x, PMV[1].y,
        forward_top_field_reference,
        forward_bottom_field_reference,
        x, y, 0);
    field_motion_compensate_one_reference(
        luma_pred_bak, cb_pred_bak, cr_pred_bak,
        luma_bak_ref_vop, cb_bak_ref_vop, cr_bak_ref_vop,
        PMV[2].x, PMV[2].y, PMV[3].x, PMV[3].y,
        backward_top_field_reference,
        backward_bottom_field_reference,
        x, y, 0);
    for (iy = 0; iy < 16; iy++) {
        for (ix = 0; ix < 16; ix++) {
            luma_pred[ix][iy] = (luma_pred_fwd[ix][iy] +
                                 luma_pred_bak[ix][iy] + 1) >> 1;
        }
    }
    for (iy = 0; iy < 8; iy++) {
        for (ix = 0; ix < 8; ix++) {
            cb_pred[ix][iy] = (cb_pred_fwd[ix][iy] +
                               cb_pred_bak[ix][iy] + 1) >> 1;
            cr_pred[ix][iy] = (cr_pred_fwd[ix][iy] +
                               cr_pred_bak[ix][iy] + 1) >> 1;
        }
    }
```

The direct mode prediction can be either progressive (see subclause 7.6.9.5) or interlaced as described below. Interlaced direct mode is used when ever the co-located macroblock (macroblock with the same coordinates) of the future reference VOP has field_predition flag is "1 Note that if the future macroblock is a skipped macroblock in a

© ISO/IEC 2004 – All rights reserved

P-VOP, a GMC macroblock (i.e. mcsel == '1') in an S(GMC)-VOP, or an intra macroblock, the direct mode prediction is progressive. Otherwise, interlaced direct mode prediction is used.

Interlaced direct coding mode is an extension of progressive direct coding mode. Four derived field motion vectors are calculated from the forward field motion vectors of the co-located future reference VOP, a single differential motion vector and the temporal position of the B-VOP fields with respect to the fields of the past and future reference VOPs. The four derived field motion vectors are denoted mvf[0] (top field forward) mvf[1], (bottom field forward), mvb[0] (top field backward), and mvb[1] (bottom field backward). MV[i] is the future reference picture motion vector for the top (i == 0) and bottom (i == 1) fields. Only one delta motion vector (used for both field), MVD[0], occurs in the bitstream for the field direct mode predicted macroblock. MVD[0] is decoded assuming f_code == 1 regardless of the number in VOP header. The interlaced direct mode prediction (in luma_pred[][], cb_pred[][] and cr_pred[][]) is calculated as shown below.

```
for (i = 0; i < 2; i++) {
    mvf[i].x = (TRB[i] * MV[i].x) / TRD[i] + MVD[0].x;
    mvf[i].y = (TRB[i] * MV[i].y) / TRD[i] + MVD[0].y;
    mvb[i].x = (MVD[i].x == 0) ?
        (((TRB[i] - TRD[i]) * MV[i].x) / TRD[i]) :
        mvf[i].x - MV[i].x);
    mvb[i].y = (MVD[i].y == 0) ?
        (((TRB[i] - TRD[i]) * MV[i].y) / TRD[i]) :
        mvf[i].y - MV[i].y);
field_motion_compensate_one_reference(
    luma_pred_fwd, cb_pred_fwd, cr_pred_fwd,
    luma_fwd_ref_vop, cb_fwd_ref_vop, cr_fwd_ref_vop,
    mvf[0].x, mvf[0].y, mvf[1].x, mvf[1].y,
    colocated_future_mb_top_field_reference,
    colocated_future_mb_bottom_field_reference,
    x, y, 0);
field_motion_compensate_one_reference(
    luma_pred_bak, cb_pred_bak, cr_pred_bak,
    luma_bak_ref_vop, cb_bak_ref_vop, cr_bak_ref_vop,
    mvb[1].x, mvb[1].y, mvb[1].x, mvb[1].y,
    0, 1, x, y, 0);
for (iy = 0; iy < 16; iy++) {
    for (ix = 0; ix < 16; ix++) {
        luma_pred[ix][iy] = (luma_pred_fwd[ix][iy] +
                             luma_pred_bak[ix][iy] + 1) >> 1;
    }
}
for (iy = 0; iy < 8; iy++) {
    for (ix = 0; ix < 8; ix++) {
        cb_pred[ix][iy] = (cb_pred_fwd[ix][iy] +
                           cb_pred_bak[ix][iy] + 1) >> 1;
        cr_pred[ix][iy] = (cr_pred_fwd[ix][iy] +
                           cr_pred_bak[ix][iy] + 1) >> 1;
    }
}
```

The temporal references (TRB[i] and TRD[i]) are distances in time expressed in field periods. Figure 7-48 shows how they are defined for the case where i is 0 (top field of the B-VOP). The bottom field is analogously.

© ISO/IEC 2004 – All rights reserved

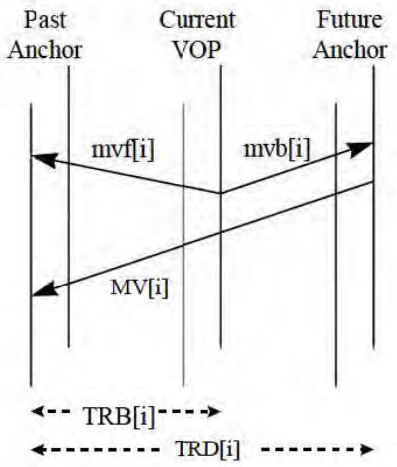


Figure 7-48 — Interlaced direct mode

The calculation of TRD[i] and TRB[i] depends not only on the current field, reference field, and frame temporal references, but also on whether the current video is top field first or bottom field first.

$$TRD[i] = 2*(T(future)//Tframe - T(past)//Tframe) + \delta[i]$$

$$TRB[i] = 2*(T(current)//Tframe - T(past)//Tframe) + \delta[i]$$

where T(future), T(current) and T(past) are the cumulative VOP times calculated from modulo_time_base and vop_time_increment of the future, current and past VOPs in display order. Tframe is the frame period determined by

$$Tframe = T(first\_B\_VOP) - T(past\_reference\_of\_first\ B\_VOP)$$

where first_B_VOP denotes the first B-VOP following the Video Object Layer syntax. The important thing about Tframe is that the period of time between consecutive fields which constitute an interlaced frame is assuemed to be 0.5 * Tframe for purposes of scaling the motion vectors.

The value of δ is determined from Table 7-16; it is a function of the current field parity (top or bottom), the reference field of the co-located macroblock (macroblock at the same coordinates in the furture reference VOP), and the value of top_field_first in the B-VOP's video object plane syntax.

Table 7-16 — Selection of the parameter δ

future reference VOP reference fields of the co-located macroblock		top_field_first == 0		top_field_first == 1	
Top field reference	Bottom field reference	Top field, $\delta[0]$	Bottom field, $\delta[1]$	Top field, $\delta[0]$	Bottom field, $\delta[1]$
0	0	0	-1	0	1
0	1	0	0	0	0
1	0	1	-1	-1	1
1	1	1	0	-1	0

© ISO/IEC 2004 – All rights reserved

The top field prediction is based on the top field motion vector of the P-VOP macroblock of the future reference VOP. The past reference field is the reference field selected by the co-located macroblock of the future reference VOP for the top field. Analogously, the bottom field predictor is the average of pixels obtained from the future reference VOP's bottom field and the past reference field referenced by the bottom field motion vector of the corresponding macroblock of the future reference VOP. When interlaced direct mode is used, vop_time_increment_resolution must be the smallest integer greater than or equal to the number of frames per second. In each VOP, vop_time_increment counts individual frames within a second.

7.8 Sprite decoding

The subclause specifies the additional decoding process for a sprite video object. The sprite decoding can operate in two modes: basic sprite decoding and low-latency sprite decoding. Figure 7-49 is a diagram of the sprite decoding process. It is simplified for clarity.

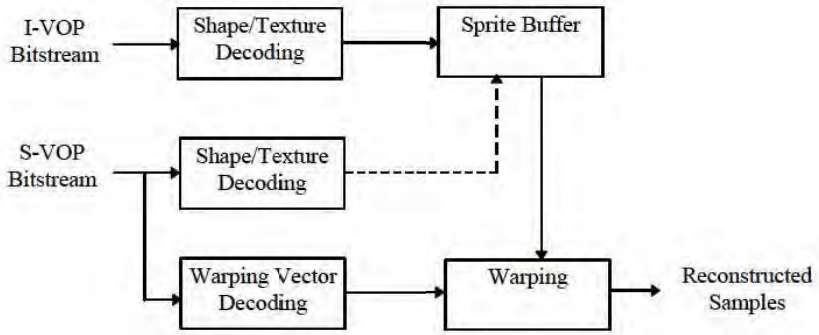


Figure 7-49 — The sprite decoding process

7.8.1 Higher syntactic structures

The various parameters in the VOL and VOP bitstreams shall be interpreted as described in Clause 6. When sprite_enable == 'static', vop_coding_type shall be "I" only for the initial VOP in a VOL for basic sprites (i.e. low_latency_sprite_enable == '0'), and all the other VOPs shall be S-VOPs (i.e. vop_coding_type == "S"). The reconstructed I-VOP in a VOL for basic sprites is not displayed but stored in a sprite memory, and will be used by all the remaining S-VOPs with sprite_enable == 'static' in the same VOL. An S-VOP with sprite_enable == 'static' is reconstructed by applying warping to the VOP stored in the sprite memory, using the warping parameters (i.e. a set of motion vectors) embedded in the VOP bitstream. Alternatively, in a VOL for low-latency sprites (i.e. low_latency_sprite_enable == '1'), these S-VOPs with sprite_enable == 'static' can update the information stored in the sprite memory before applying warping.

7.8.2 Sprite Reconstruction

The luminance, chrominance and grayscale alpha data of a sprite are stored in two-dimensional arrays. The width and height of the luminance array are specified by sprite_width and sprite_height respectively. The samples in the sprite luminance, chrominance and grayscale alpha arrays are addressed by two-dimensional integer pairs (i', j') and (i_c', j_c') as defined in the following:

- Top left luminance and grayscale alpha sample
  (i', j') = (sprite_left_coordinate, sprite_top_coordinate)
- Bottom right luminance and grayscale alpha sample
  (i', j') = (sprite_left_coordinate + sprite_width – 1,
  sprite_top_coordinate + sprite_height – 1)
- Top left chrominance sample
  (i_c', j_c') = (sprite_left_coordinate / 2, sprite_top_coordinate / 2)

© ISO/IEC 2004 – All rights reserved

- Bottom right chrominance sample
  (i_c', j_c') = (sprite_left_coordinate / 2 + sprite_width// 2 – 1,
  sprite_top_coordinate / 2 + sprite_height// 2 – 1).

Likewise, the addresses of the luminance, chrominance and grayscale alpha samples of the VOP currently being decoded are defined in the following:

- Top left sample of luminance and grayscale alpha
  (i, j) = (0, 0) for rectangular VOPs, and
  (i, j) = (vop_horizontal_mc_spatial_ref, vop_vertical_mc_spatial_ref) for non-rectangular VOPs

- Bottom right sample of luminance and grayscale alpha
  (i, j) = (video_object_layer_width - 1, video_object_layer_height - 1) for rectangular VOPs, and
  (i, j) = (vop_horizontal_mc_spatial_ref + vop_width - 1,
  vop_vertical_mc_spatial_ref + vop_height - 1) for non-rectangular VOPs

- Top left sample of chrominance
  (i_c, j_c) = (0, 0) for rectangular VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2, vop_vertical_mc_spatial_ref / 2) for non-rectangular VOPs

- Bottom right sample of chrominance
  (i_c, j_c) = (video_object_layer_width / 2 - 1, video_object_layer_height / 2 - 1) for rectangular VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2 + vop_width// 2 - 1,
  vop_vertical_mc_spatial_ref / 2 + vop_height// 2 - 1) for non-rectangular VOPs

7.8.3 Low-latency sprite reconstruction

This subclause allows a large static sprite to be reconstructed at the decoder by properly incorporating its corresponding pieces. There are two types of pieces recognized by the decoder—object and update. The decoded sprite object-piece (i.e., embedded in a S-VOP with low_latence_sprite_enable==1 and sprite_transmit_mode=="piece") is a highly quantised version of the original sprite piece while the sprite update-piece (i.e., sprite_transmit_mode=="update") is a residual designed to improve upon the quality of decoded object-piece. Sprite pieces are rectangular pieces of texture (and shape for the object-piece) and can contain "holes," corresponding to macroblocks, that do not need to be decoded. Five parameters are required by the decoder to properly incorporate the pieces: piece_quant, piece_width, piece_height, piece_xoffset, and piece_yoffset.

Macroblocks raster scanning is employed to decode each piece. However, whenever the scan encounters a macroblock which has been part of some previously sent sprite piece, then the macroblock is not decoded and its corresponding macroblock layer is empty. In that case, the decoder treats the macroblock as a hole in the current sprite piece. Since a macroblock can be refined as long as there is some available bandwidth, more than one update may be decoded per macroblock and the holes for a given refinement step have no relationship to the holes of later refinement steps. Therefore, the decoding process of a hole for an update piece is different than that for the object-piece. For the object-piece, no information is decoded at all and the decoder must "manage" where "holes" lie. (see subclause 7.8.3.1). For the update-piece, the not_coded bit is decoded to indicate whether or not one more refinement should be decoded for this given macroblock. (see subclause 7.8.3.2). Note that a hole could be non-transparent and have had shape information decoded previously. Multiple intermingled object-pieces and update-pieces may be decoded at the same current VOP. Part of a sequence could consist for example of rapidly showing a zooming out effect, a panning to the right, a zooming in, and finally a panning to the left. In this case, the first decoded object-piece covers regions on all four sides of the previous VOP transmitted piece, which is now treated as a hole and not decoded again. The second decoded object-piece relates to the right panning, and the third object-piece is a smaller left-panning piece due to the zooming-in effect. Finally, the last piece is different; instead of an object, it contains the update for some previous object-piece of zooming-in (thus, the need to update to refine for higher quality). All four pieces will be decoded within the same VOP. When sprite_transmit_mode = ="pause," the decoder recognizes that all sprite object-pieces and update-pieces for the current VOP session have been sent. However, when sprite_transmit_mode = "stop," the decoder understands that all object and update-pieces have been sent for the entire video object layer, not just for the current VOP. session. In addition, once all object-pieces or update-pieces have been decoded during a VOP session (i.e., signaled by sprite_transmit_mode == "pause" or sprite_transmit_mode == "stop"), the static sprite is padded (as defined in subclause 7.6.1), then the portion to be displayed is warped, to complete the current VOP session.

© ISO/IEC 2004 – All rights reserved

For the S-VOPs (i.e., vop_coding_type == "S") with sprite_enable == 'static', the macroblock layer syntax of object-pieces is the same as those of I-VOP. Therefore, shape and texture are decoded using the macroblock layer structure in I-VOPs with the quantisation of intra macroblocks. The syntax of the update-pieces is similar to the P-VOP inter-macroblock syntax with the quantisation of non-intra macroblocks); however, the differences are indicated in Table B.1, specifically that there are no motion vectors and shape information included in this decoder syntax structure. In summary, this decoding process supports the construction of any large sprite image progressively, both spatially and in terms of quality.

7.8.3.1 Decoding of holes in sprite object-piece

Implementation of macroblock scanning must account for the possibility that a macroblock uses prediction based on some macroblock sent in a previous piece. When an object-piece with holes is decoded, the decoder in the process of reconstruction acts as if the whole original piece were decoded, but actually only the bitstream corresponding to the "new macroblock" is received. Whenever macroblocks raster scanning encounters a hole, the decoder needs to manage the retrieval of relevant information (e.g. DCT quantiser scales, AC and DC prediction parameters, and BAB bordering values) from the corresponding macroblock decoded earlier.

7.8.3.2 Decoding of holes in sprite update-pieces

In contrast to the send_mb() used by the object-pieces, the update-pieces use the not_coded bit. When not_coded = 1 in the P-VOP syntax, the decoder recognizes that the corresponding macroblock is not refined by the current sprite update-piece. When not_coded = 0 in the P-VOP syntax, the decoder recognizes that this macroblock is refined. The prediction for the update piece is obtained by extracting the "area" of the static sprite defined by (piece_width, piece_height, piece_xoffset, piece_yoffset). This area is then padded and serves as prediction for the update pieces. Since there is no shape information included in an update-piece, the result of its transparent_mb() is retrieved from the corresponding macroblock in the object-piece decoded earlier. In addition, an update macroblock cannot be transmitted before its corresponding object macroblock. As a result, the very first sprite piece transmitted in the low-latency mode shall be an object-piece.

7.8.4 Sprite reference point decoding

The syntactic elements in sprite_trajectory () and below shall be interpreted as specified in Clause 6. du[i] and dv[i] ($0 \le i <$ no_of_sprite_warping_points) specifies the mapping between indexes of some reference points in the VOP and the corresponding reference points in the sprite or reference VOP. These points are referred to as VOP reference points and sprite reference points, respectively, in the rest of the specification.

The index values for the VOP reference points are defined as:

$(i_0, j_0) = (0, 0)$ when video_object_layer_shape == 'rectangular', and
(vop_horizontal_mc_spatial_ref, vop_vertical_mc_spatial_ref) otherwise,
$(i_1, j_1) = (i_0+W,\ j_0)$,
$(i_2, j_2) = (i_0,\ j_0 + H)$,
$(i_3, j_3) = (i_0+W,\ j_0+H)$

where W = video_object_layer_width and H = video_object_layer_height when video_object_layer_shape == 'rectangular' or W = vop_width and H = vop_height otherwise. Only the index values with subscripts less than no_of_sprite_warping_points shall be used for the rest of the decoding process.

The index values for the sprite reference points shall be calculated as follows:

$$(i_0', j_0') = (s / 2)\ (2\ i_0 + du[0],\ 2\ j_0 + dv[0])$$
$$(i_1', j_1') = (s / 2)\ (2\ i_1 + du[1] + du[0],\ 2\ j_1 + dv[1] + dv[0])$$
$$(i_2', j_2') = (s / 2)\ (2\ i_2 + du[2] + du[0],\ 2\ j_2 + dv[2] + dv[0])$$
$$(i_3', j_3') = (s / 2)\ (2\ i_3 + du[3] + du[2] + du[1] + du[0],\ 2\ j_3 + dv[3] + dv[2] + dv[1] + dv[0])$$

where i_0', j_0', etc are integers in $\frac{1}{s}$ pel accuracy, where s is specified by sprite_warping_accuracy. Only the index values with subscripts less than no_of_sprite_warping_points need to be calculated.

© ISO/IEC 2004 – All rights reserved

When no_of_sprite_warping_points == 2 or 3, the index values for the *virtual sprite points* are additionally calculated as follows:

$$(i_1'', j_1'') = (16\,(i_0 + W') + ((W - W')\,(r\,i_0' - 16\,i_0) + W'\,(r\,i_1' - 16\,i_1))\ //\ W,\\ 16\,j_0 + ((W - W')\,(r\,j_0' - 16\,j_0) + W'\,(r\,j_1' - 16\,j_1))\ //\ W)$$
$$(i_2'', j_2'') = (16\,i_0 + ((H - H')\,(r\,i_0' - 16\,i_0) + H'\,(r\,i_2' - 16\,i_2))\ //\ H,\\ 16\,(j_0 + H') + ((H - H')\,(r\,j_0' - 16\,j_0) + H'\,(r\,j_2' - 16\,j_2))\ //\ H)$$

where i_1'', j_1'', i_2'', and j_2'' are integers in $\frac{1}{16}$ pel accuracy, and $r = 16/s$. W' and H' are defined as the smallest integers that satisfy the following condition:

$W' = 2^{\alpha}$, $H' = 2^{\beta}$, $W' \geq W$, $H' \geq H$, $\alpha > 0$, $\beta > 0$, both α and β are integers.

The calculation of i_2'', and j_2'' is not necessary when no_of_sprite_warping_points == 2.

7.8.5 Warping

For any pixel (i, j) inside the VOP boundary, $(F(i, j), G(i, j))$ and $(F_c(i_c, j_c), G_c(i_c, j_c))$ are computed as described in the following. These quantities are then used for sample reconstruction as specified in subclause 7.8.6. The following notations are used to simplify the description:

$$I = i - i_0,$$
$$J = j - j_0,$$
$$I_c = 4\,i_c - 2\,i_0 + 1,$$
$$J_c = 4\,j_c - 2\,j_0 + 1,$$

When no_of_sprite_warping_point == 0,

$$(F(i, j), G(i, j)) = (s\,i,\ s\,j),$$
$$(F_c(i_c, j_c), G_c(i_c, j_c)) = (s\,i_c,\ s\,j_c).$$

When no_of_sprite_warping_point == 1 and sprite_enable == 'static,

$$(F(i, j), G(i, j)) = (i_0' + sI,\ j_0' + s\,J),$$
$$(F_c(i_c, j_c), G_c(i_c, j_c)) = i_0'\ ///\ 2 + s\,(i_c - i_0 / 2),\ \ j_0'\ ///\ 2 + s\,(j_c - j_0 / 2)).$$

When no_of_sprite_warping_point == 1 and sprite_enable == 'GMC',

$$(F(i, j), G(i, j)) = (i_0' + sI,\ j_0' + s\,J),$$
$$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((i_0' >> 1)\ |\ (i_0'\ \&\ 1)) + s\,(i_c - i_0 / 2),\ ((j_0' >> 1)\ |\ (j_0'\ \&\ 1)) + s\,(j_c - j_0 / 2))$$

When no_of_sprite_warping_points == 2,

$$(F(i, j), G(i, j)) = (\,i_0' + ((-r\,i_0' + i_1'')\,I + (r\,j_0' - j_1'')\,J)\ ///\ (W'\,r),\\ j_0' + ((-r\,j_0' + j_1'')\,I + (-r\,i_0' + i_1'')\,J)\ ///\ (W'\,r)),$$
$$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2\,W'\,r\,i_0' - 16W')\ ///\ (4\,W'\,r),\\ ((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c + 2\,W'\,r\,j_0' - 16W')\ ///\ (4\,W'\,r)).$$

According to the definition of W' and H' (i.e. $W' = 2^{\alpha}$ and $H' = 2^{\beta}$), the divisions by "///" in these functions can be replaced by binary shift operations. By this replacement, the above equations can be rewritten as:

$$(F(i, j), G(i, j)) = (\,i_0' + (((-r\,i_0' + i_1'')\,I + (r\,j_0' - j_1'')\,J + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),\\ j_0' + (((-r\,j_0' + j_1'')\,I + (-r\,i_0' + i_1'')\,J + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),$$
$$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2\,W'\,r\,i_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2),\\ ((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c + 2\,W'\,r\,j_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2)),$$

where $2^{\rho}=r$.

© ISO/IEC 2004 – All rights reserved

When no_of_sprite_warping_points == 3,

$$
\begin{aligned}
(F(i, j), G(i, j)) &= (i_0' + ((-r\, i_0' + i_1'')\, H'\, I + (-r\, i_0' + i_2'')W'\, J) \;///\; (W'H'r),\\
&\quad\; j_0' + ((-r\, j_0' + j_1'')\, H'\, I + (-r\, j_0' + j_2'')W'\, J) \;///\; (W'H'r)),\\
(F_c(i_c, j_c), G_c(i_c, j_c)) &= (((-r\, i_0' + i_1'')\, H'\, I_c + (-r\, i_0' + i_2'')W'\, J_c + 2\, W'H'r\, i_0' - 16W'H') \;///\; (4W'H'r),\\
&\quad\; ((-r\, j_0' + j_1'')\, H'\, I_c + (-r\, j_0' + j_2'')W'\, J_c + 2\, W'H'r\, j_0' - 16W'H') \;///\; (4W'H'r)).
\end{aligned}
$$

According to the definition of W' and H', the computation of these functions can be simplified by dividing the denominator and numerator of division beforehand by W' (when $W' < H'$) or H' (when $W' \geq H'$). As in the case of no_of_sprite_warping_points == 2, the divisions by "///" in these functions can be replaced by binary shift operations. For example, when $W' \geq H'$ (i.e. $\alpha \geq \beta$) the above equations can be rewritten as:

$$
\begin{aligned}
(F(i, j), G(i, j)) &= (i_0' + (((-r\, i_0' + i_1'')\, I + (-r\, i_0' + i_2'')\, 2^{\alpha-\beta}\, J + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),\\
&\quad\; j_0' + (((-r\, j_0' + j_1'')\, I + (-r\, j_0' + j_2'')\, 2^{\alpha-\beta}\, J + 2^{\alpha+\rho-1}) >> (\alpha+\rho))),\\
(F_c(i_c, j_c), G_c(i_c, j_c)) &= (((-r\, i_0' + i_1'')\, I_c + (-r\, i_0' + i_2'')\, 2^{\alpha-\beta}\, J_c + 2W'r\, i_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2),\\
&\quad\; ((-r\, j_0' + j_1'')\, I_c + (-r\, j_0' + j_2'')\, 2^{\alpha-\beta}\, J_c + 2W'r\, j_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2)).
\end{aligned}
$$

When no_of_sprite_warping_point == 4,

$$
\begin{aligned}
(F(i, j), G(i, j)) &= ((a\, I + b\, J + c) \;///\; (g\, I + h\, J + D\, W H),\\
&\quad\; (d\, I + e\, J + f) \;///\; (g\, I + h\, J + D\, W H)),\\
(F_c(i_c, j_c), G_c(i_c, j_c)) &= ((2\, a\, I_c + 2\, b\, J_c + 4\, c - (g\, I_c + h\, J_c + 2\, D\, W H)\, s) \;///\; (4g I_c + 4\, h J_c + 8D\, W H),\\
&\quad\; (2\, d\, I_c + 2\, e\, J_c + 4\, f - (g\, I_c + h\, J_c + 2\, D\, W H)\, s) \;///\; (4\, g\, I_c + 4\, h J_c + 8D\, W H))
\end{aligned}
$$

where

$$
\begin{aligned}
g &= ((i_0' - i_1' - i_2' + i_3')\,(j_2' - j_3') - (i_2' - i_3')\,(j_0' - j_1' - j_2' + j_3'))\, H,\\
h &= ((i_1' - i_3')\,(j_0' - j_1' - j_2' + j_3') - (i_0' - i_1' - i_2' + i_3')\,(j_1' - j_3'))\, W,\\
D &= (i_1' - i_3')\,(j_2' - j_3') - (i_2' - i_3')\,(j_1' - j_3'),\\
a &= D\,(i_1' - i_0')\, H + g\, i_1',\\
b &= D\,(i_2' - i_0')\, W + h\, i_2',\\
c &= D\, i_0'\, W\, H,\\
d &= D\,(j_1' - j_0')\, H + g\, j_1',\\
e &= D\,(j_2' - j_0')\, W + h\, j_2',\\
f &= D\, j_0'\, W\, H.
\end{aligned}
$$

A set of parameters that causes the denominator of any of the above equations to be zero for any pixel in an opaque or boundary macroblock is disallowed. The implementor should be aware that a 32 bit register may not be sufficient for representing the denominator or the numerator in the above transform functions for affine and perspective transform. The usage of a 64 bit floating point representation should be sufficient in such case.

7.8.6 Sample reconstruction

The reconstructed value Y of the luminance sample (i, j) in the currently decoded VOP shall be defined as

$$Y = ((s - r_j)((s - r_i)\, Y_{00} + r_i\, Y_{01}) + r_j\,((s - r_i)\, Y_{10} + r_i\, Y_{11})) \;//\; s^2,$$

when sprite_enable == 'static', and

$$Y = ((s - r_j)((s - r_i)\, Y_{00} + r_i\, Y_{01}) + r_j\,((s - r_i)\, Y_{10} + r_i\, Y_{11}) + s^2/2 - \text{rounding\_control}) \;/\; s^2,$$

when sprite_enable == 'GMC', where Y_{00}, Y_{01}, Y_{10}, Y_{11} represent the luminance sample in the sprite or reference VOP at $(F(i, j)////s, G(i, j)////s)$, $(F(i, j)////s + 1, G(i, j)////s)$, $(F(i, j)////s, G(i, j)////s + 1)$, and $(F(i, j)////s + 1, G(i, j)////s + 1)$ respectively, and $r_i = F(i, j) - (F(i, j)////s)s$ and $r_j = G(i, j) - (G(i, j)////s)s$. See sublause 7.6.2 for the definition of rounding_control. Figure 7-50 illustrates this process.

© ISO/IEC 2004 – All rights reserved

In case any of Y_{00}, Y_{01}, Y_{10} and Y_{11} lies outside the luminance binary mask of the sprite or reference VOP, it shall be obtained by the padding process as defined in subclause 7.6.1. Additionally, in the case when the current VOP is a rectangular S(GMC)-VOP and any of Y_{00}, Y_{01}, Y_{10} and Y_{11} lies outside the luminance bounding rectangle of the reference VOP, the reconstructed sample value is obtained as described in subclause 7.6.4.

When brightness_change_in_sprite == 1, the final reconstructed luminance sample (i, j) is further computed as Y = $(Y$ * (brightness_change_factor + 100)) // 100, clipped to the range of [0, $2^{\text{bits\_per\_pixel}}$-1].

Similarly, the reconstructed value C of the chrominance sample (i_c, j_c) in the currently decoded VOP shall be defined as

$$C = ((s - r_j)((s - r_i) C_{00} + r_i C_{01}) + r_j ((s - r_i) C_{10} + r_i C_{11})) // s^2,$$

when sprite_enable == 'static', and

$$C = ((s - r_j)((s - r_i) C_{00} + r_i C_{01}) + r_j ((s - r_i) C_{10} + r_i C_{11}) + s^2/2 - \text{rounding\_control}) / s^2,$$

when sprite_enable == 'GMC', where C_{00}, C_{01}, C_{10}, C_{11} represent the chrominance sample in the sprite or reference VOP at ($F_c(i_c, j_c)////s$, $G_c(i_c, j_c)////s$), ($F_c(i_c, j_c)////s + 1$, $G_c(i_c, j_c)////s$), ($F_c(i_c, j_c)////s$, $G_c(i_c, j_c)////s + 1$), and ($F_c(i_c, j_c)////s + 1$, $G_c(i_c, j_c)////s + 1$) respectively, and $r_i = F_c(i_c, j_c) - (F_c(i_c, j_c))////s)s$ and $r_j = G_c(i_c, j_c) - (G_c(i_c, j_c)////s)s$. In case any of C_{00}, C_{01}, C_{10} and C_{11} lies outside the chrominance binary mask of the sprite or reference VOP, it shall be obtained by the padding process as defined in subclause 7.6.1. Additionally, in the case when the current VOP is a rectangular S(GMC)-VOP and any of C_{00}, C_{01}, C_{10} and C_{11} lies outside the chrominance bounding box of the reference VOP, the reconstructed sample value is obtained as described in subclause 7.6.4.

The same method is used for the reconstruction of grayscale alpha and luminance samples. The reconstructed value A of the grayscale alpha sample (i, j) in the currently decoded VOP shall be defined as

$$A = ((s - r_j)((s - r_i) A_{00} + r_i A_{01}) + r_j ((s - r_i) A_{10} + r_i A_{11})) // s^2,$$

when sprite_enable == 'static', and

$$A = ((s - r_j)((s - r_i) A_{00} + r_i A_{01}) + r_j ((s - r_i) A_{10} + r_i A_{11}) + s^2/2 - \text{rounding\_control}) / s^2,$$

when sprite_enable == 'GMC', where A_{00}, A_{01}, A_{10}, A_{11} represent the grayscale alpha samples in the sprite or reference VOP at ($F(i, j)////s$, $G(i, j)////s$), ($F(i, j)////s + 1$,$G(i, j)////s$), ($F(i, j)////s$, $G(i, j)////s + 1$), and ($F(i, j)////s + 1$,$G(i, j)////s + 1$) respectively, and $r_i = F(i, j) - (F(i, j)////s)s$ and $r_j = G(i, j) - (G(i, j)////s)s$. In case any of A_{00}, A_{01}, A_{10} and A_{11} lies outside the luminance binary mask of the sprite or reference VOP, it shall be obtained by the padding process as defined in subclause 7.6.1. Additionally, in the case when the current VOP is a rectangular S(GMC)-VOP and any of A_{00}, A_{01}, A_{10} and A_{11} lies outside the grayscale alpha bounding box of the reference VOP, the reconstructed sample value is obtained as described in subclause 7.6.4.

When sprite_enable == "static", the reconstructed value of luminance binary mask sample $BY(i,j)$ shall be computed following the identical process for the luminance sample. However, corresponding binary mask sample values shall be used in place of luminance samples Y_{00}, Y_{01}, Y_{10}, Y_{11}. Assume the binary mask sample opaque is equal to 255 and the binary mask sample transparent is equal to 0. If the computed value is bigger or equal to 128, $BY(i, j)$ is defined as opaque. Otherwise, $BY(i, j)$ is defined as transparent. The chrominance binary mask samples shall be reconstructed by decimating of the corresponding 2 x 2 adjacent luminance binary mask samples as specified in subclause 7.6.1.4.

© ISO/IEC 2004 – All rights reserved

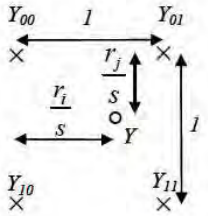


Figure 7-50 — Pixel value interpolation (it is assumed that sprite samples are located on an integer grid)

7.8.7 GMC decoding

7.8.7.1 GMC prediction

For S(GMC)-VOPs, the pixels in a macroblock (including pixels in the grayscale alpha MB) are predicted using GMC prediction when mcsel == '1'. The predicted macroblock is obtained by applying warping to the reference VOP. The prediction error is decoded by the same method as in usual interframe motion compensation. The prediction based on warping is performed using sprite reference point decoding described in subclause 7.8.4, sub-pel warping described in subclause 7.8.5, and sample reconstruction described in subclause 7.8.6.

7.8.7.2 Texture and shape decoding in S(GMC)-VOP

Decoding of texture and shape in S(GMC)-VOPs follows the same rule as those in P-VOPs. Global warping parameters are not used for shape decoding in S(GMC)-VOPs. Macroblocks with mcsel == '1' are regarded as intra-coded macroblocks in shape motion vector decoding since they do not have their own motion vector.

Overlapped block MC is disabled over the boundary between macroblocks with different values for mcsel.

When interlaced == '1', macroblocks with mcsel == "1" are predicted using frame prediction (i.e. the warping method is identical with the case when interlaced == '0').

7.8.7.3 Motion vector decoding

Motion vectors of macroblocks with mcsel == '0' and not_coded == '0' in S-VOPs are decoded using the same rules as P-VOPs. Although macroblocks with mcsel == '1' do not have their own *block* motion vectors, they have *pel-wise* motion vectors for sprite warping obtained from global motion parameters. The candidate motion vector predictor from the reference macroblock with mcsel == '1' is obtained as the *averaged value* of the pel-wise motion vectors in the macroblock. Note that any reference macroblock outside video packet is treated as transparent. In case the frame-based direct mode in B-VOPs, the motion vector of the co-located macroblock with mcsel == '1' is also obtained as the averaged value of the pel-wise motion vectors in the macroblock when the most recently decoded VOP is S(GMC)-VOP.

Note : Averaged value of pel-wise vectors is calculated using the all luminance pixels in the macroblock, that is 256 pixels are used (Nb is always 256 into the following expression).

© ISO/IEC 2004 – All rights reserved

$$AMV_x = \left(\sum_y \sum_x MV_x(x,y) \right) / N_b$$

$$AMV_y = \left(\sum_y \sum_x MV_y(x,y) \right) / N_b$$

MVx(x,y): Horizontal motion vector at *(x,y)*

MVy(x,y): Vertical motion vector at *(x,y)*

N_b: Number of pixels in the reference block

AMVx: Averaged value of horizontal motion vectors in the block

AMVy: Averaged value of vertical motion vectors in the block

Here,

Since the *AMVs* have fractional values, they are quantised to half integer values when quarter_sample == '0', and quarter integer values when quarter_sample == '1', using the "//" operator. For example, values within [0.0, 0.25) are rounded to 0, [0.25, 0.75) are rounded to 0.5, and [0.75, 1.0] are rounded to 1.0 when quarter_sample == '0'. Values within [0.0, 0.125) are rounded to 0, [0.125, 0.375) are rounded to 0.25, [0.375, 0.625) are rounded to 0.5, [0.625, 0.875) are rounded to 0.75, and [0.875, 1.0] are rounded to 1.0 when quarter_sample == '1'.

If the quantised AMV is outside the motion vector range specified by f_code, it is clipped in the range.

The operation above is performed independently for horizontal and vertical components.

For example, if the left block is coded in GMC (Figure 7-51), the candidate predictor is obtained as the averaged value of the pel-wise motion vectors in the left block.

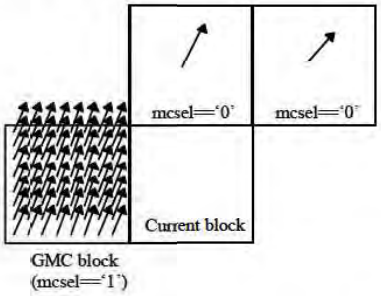


Figure 7-51 — Pel-wise motion vectors of GMC blocks

7.9 Generalized scalable decoding

This subclause specifies the additional decoding process required for decoding scalable coded video.

The scalability framework is referred to as generalized scalability which includes the spatial and the temporal scalabilities. The temporal scalability offers scalability of the temporal resolution, and the spatial scalability offers scalability of the spatial resolution. Each type of scalability involves more than one layer. In the case of two layers, consisting of a lower layer and a higher layer; the lower layer is referred to as the reference layer and the higher layer is called the enhancement layer.

© ISO/IEC 2004 – All rights reserved

In the case of temporal scalability, both rectangular VOPs as well as arbitrary shaped VOPs are supported. In the case of spatial scalability, only rectangular VOPs are supported. Figure 7-52 shows a high level decoder structure for generalized scalability.

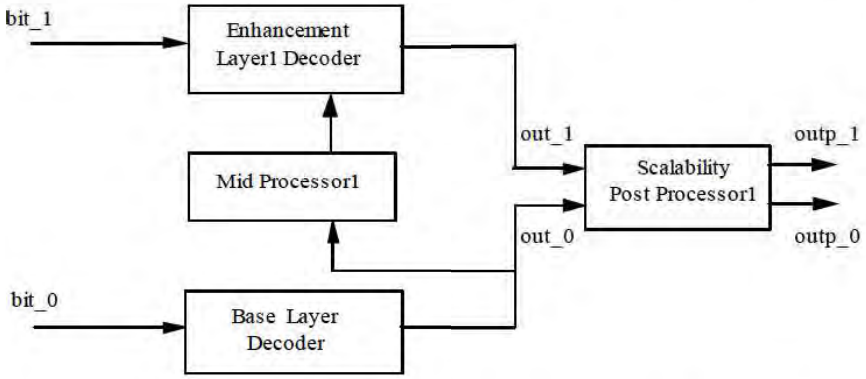


Figure 7-52 — High level decoder structure for generalized scalability

The base layer and enhancement layer bitstreams are input for decoding by the corresponding base layer decoder and enhancement layer decoder.

When spatial scalability is to be performed, mid processor 1 performs spatial up or down sampling of input. The scalability post processor performs any necessary operations such as spatial up or down sampling of the decoded reference layer for display resulting at outp_0 while the enhancement layer without resolution conversion may be output as outp_1.

When temporal scalability is to be performed, the decoding of base and enhancement layer bitstreams occurs in the corresponding base and enhancement layer decoders as shown. In this case, mid processor 1 does not perform any spatial resolution conversion. The post processor outputs the base layer VOPs without any conversion, but temporally multiplexes the base and enhancement layer VOPs to produce higher temporal resolution enhancement layer.

The reference VOPs for prediction are selected by ref_select_code as specified in Table 7-17 and Table 7-18. In coding of P-VOPs belonging to an enhancement layer, the forward reference is one of the following four: the most recently decoded VOP of enhancement layer, the most recent VOP of the reference layer in display order, the next VOP of the reference layer in display order, or the temporally coincident VOP in the reference layer.

In B-VOPs, the forward reference is one of the following two: the most recently decoded enhancement VOP or the most recent reference layer VOP in display order. The backward reference is one of the following three: the temporally coincident VOP in the reference layer, the most recent reference layer VOP in display order, or the next reference layer VOP in display order.

In case of hierarchy_type == 1(temporal scalability), the default value of ref_select_code is “00”. The value of vop_coded should be “1” for the first VOP in an enhancement layer if the value of video_object_layer_shape is ‘00’.

In case of hierarchy_type == 0(spatial scalability), the default value of ref_select_code is “11” for P-VOP and "00" for B-VOP. The value of vop_coded should be “1” for the first VOP in an enhancement layer if the value of video_object_layer_shape is ‘00’.

© ISO/IEC 2004 – All rights reserved

Table 7-17 — Prediction reference choices in enhancement layer P-VOPs for scalability

ref_select_code	forward prediction reference
00	Most recently decoded enhancement VOP belonging to the same layer for which "vop_coded == 1".
01	Most recently VOP in display order belonging to the reference layer for which "vop_coded == 1".
10	Next VOP in display order belonging to the reference layer.
11	Temporally coincident VOP in the reference layer (no motion vectors)

NOTE 1 — The value "10" is prohibited if "vop_coded = 0" for the next VOP in display order belonging to the reference layer.

NOTE 2 — The value "11" is prohibited if "vop_coded = 0" for the temporally coincident VOP in the reference layer.

Table 7-18 — Prediction reference choices in enhancement layer B-VOPs for scalability

ref_select_code	forward temporal reference	backward temporal reference
00	Most recently decoded enhancement VOP of the same layer for which "vop_coded == 1".	Temporally coincident VOP in the reference layer (no motion vectors)
01	Most recently decoded enhancement VOP of the same layer for which "vop_coded == 1".	Most recent VOP in display order belonging to the reference layer.
10	Most recently decoded enhancement VOP of the same layer for which "vop_coded == 1".	Next VOP in display order belonging to the reference layer.
11	Most recent VOP in display order belonging to the reference layer for which "vop_coded == 1".	Next VOP in display order belonging to the reference layer.

The following combinations of video_object_layer_shape in the base and enhancement layer and enhancement_type in the enhancement_layer are supported.

	Base Layer	Enhancement Layer	Enhancement _type	status
hierarchy_type=1	rectangular	rectangular	0	supported
	rectangular	binary shape	1	supported
	binary Shape	binary shape	0/1	supported
	binary Shape	rectangular	-	forbidden
	binary only	binary only	-	forbidden
hierarchy_type=0 && video_object_layer_ver_id = 0001	rectangular	rectangular	0	supported
	rectangular	binary shape	-	forbidden
	binary shape	binary shape	-	forbidden
	binary shape	rectangular	-	forbidden
	binary only	binary only	-	forbidden
hierarchy_type=0 && video_object_layer_ver_id = 0010	rectangular	rectangular	0	supported
	rectangular	binary shape	1	supported
	binary shape	binary shape	0/1	supported
	binary shape	rectangular	-	forbidden
	binary only	binary only	-	supported

© ISO/IEC 2004 – All rights reserved

7.9.1 Temporal scalability

Temporal scalability involves two layers, a lower layer and an enhancement layer. Both the lower and the enhancement layers process the same spatial resolution. The enhancement layer enhances the temporal resolution of the lower layer and if temporally remultiplexed with the lower layer provides full temporal rate.

7.9.1.1 Base layer and enhancement layer

In the case of temporal scalability, the decoded VOPs of the enhancement layer are used to increase the frame rate of the base layer. Figure 7-53 shows a simplified diagram of the motion compensation process for the enhancement layer using temporal scalability.

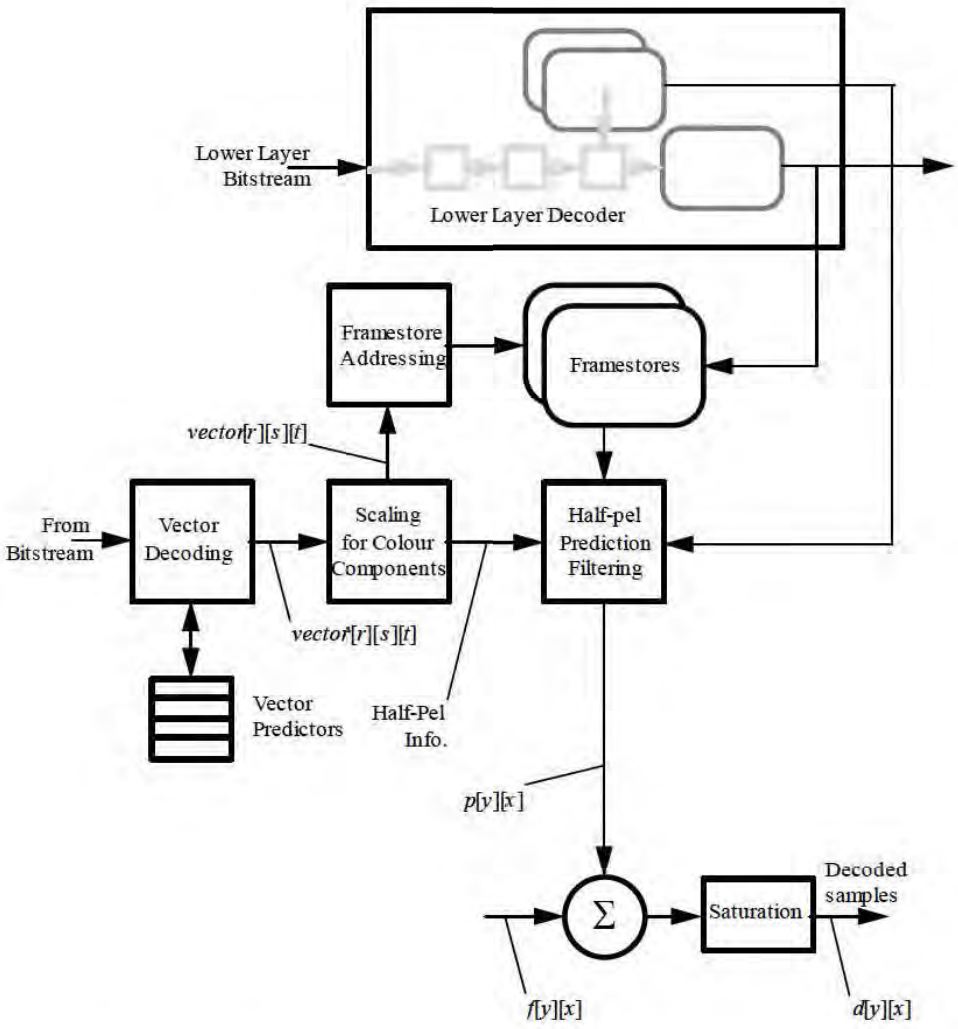


Figure 7-53 — Simplified motion compensation process for temporal scalability

Predicted samples p[y][x] are formed either from frame stores of the reference layer or from frame stores of enhancement layer. The difference data samples f[y][x] are added to p[y][x] to form the decoded samples d[y][x].

© ISO/IEC 2004 – All rights reserved

There are two types of enhancement structures indicated by the "enhancement_type" flag. When the value of enhancement_type is "1", the enhancement layer increases the temporal resolution of a partial region of the reference layer. When the value of enhancement_type is "0", the enhancement layer increases the temporal resolution of an entire region of the reference layer.

The value of video_object_layer_width in base layer and enhancement layer shall be identical if both layers are rectangular. The value of video_object_layer_height in base layer and enhancement layer shall be identical if both layers are rectangular.

7.9.1.2 Base layer

The decoding process of the base layer is the same as non-scalable decoding process.

7.9.1.3 Enhancement layer

The VOP of the enhancement layer is decoded as either I-VOP, P-VOP or B-VOP. The shape of the VOP is either rectangular (video_object_layer_shape is "00") or arbitrary (video_object_layer_shape is "01"). B-VOP in base layer shall not be used as a reference for enhancement layer VOP although B-VOP in enhancement layer can be a reference for enhancement layer VOP.

7.9.1.3.1 Decoding of I-VOPs

The decoding process of the I-VOPs in the enhancement layer is the same as the non-scalable decoding process. ref_layer_id, ref_layer_sampling_direc, hor_sampling_factor_n, horsampling_factor_m, vertical_sampling_factor_n, vertical_sampling_factor_m and ref_select_code are ignored in the temporal scalability I-VOPs.

7.9.1.3.2 Decoding of P-VOPs

The reference layer is indicated by ref_layer_id in Video Object Layer class. Other decoding process is the same as non-scalable P-VOPs except the process specified in subclauses 7.9.1.3.4 and 7.9.1.3.5.

For P-VOPs, the ref_select_code is either "00", "01" or "10".

When the value of ref_select_code is "00", the prediction reference is set by the most recently decoded VOP belonging to the same layer.

When the value of ref_select_code is "01", the prediction reference is set by the previous VOP in display order belonging to the reference layer.

When the value of ref_select_code is "10", the prediction reference is set by the next VOP in display order belonging to the reference layer.

In the case of hierarchy_type=1(temporal scalability), hor_sampling_factor_n, hor_sampling_factor_m, vert_sampling_factor_n and vert_sampling_factor_m are ignored.

7.9.1.3.3 Decoding of B-VOPs

The reference layer is indicated by ref_layer_id in Video Object Layer class. Other decoding process is the same as non-scalable B-VOPs except the process specified in subclauses 7.9.1.3.4 and 7.9.1.3.5.

For B-VOPs, the ref_select_code is either "01", "10" or "11".

When the value of ref_select_code is "01", the forward prediction reference is set by the most recently decoded VOP belonging to the same layer and the backward prediction reference is set by the previous VOP in display order belonging to the reference layer.

© ISO/IEC 2004 – All rights reserved

When the value of ref_select_code is "10", the forward prediction reference is set by the most recently decoded VOP belonging to the same layer, and the backward prediction reference is set by the next VOP in display order belonging to the reference layer.

When the value of ref_select_code is "11", the forward prediction reference is set by the previous VOP in display order belonging to the reference layer and the backward prediction reference is set by the next VOP in display order belonging to the reference layer. The picture type of the reference VOP shall be either I or P (vop_coding_type = "00" or "01").

When the value of ref_select_code is "01" or "10", direct mode is not allowed. modb shall always exist in each macroblock, i.e. the macroblock is not skipped even if the co-located macroblock is skipped.

In the case of hierarchy_type=1(temporal scalability), hor_sampling_factor_n, hor_sampling_factor_m, vert_sampling_factor_n and vert_sampling_factor_m are ignored.

7.9.1.3.4 Decoding of arbitrary shaped VOPs

The vop_shape_coding_type for enhancement layer is '0' when vop_coding_type is '00'. Otherwise, the vop_shape_coding_type for enhancement layer is '1'. Prediction for arbitrary shape in P-VOPs or in B-VOP is same as the one in the base layer (see subclause 7.5.2.1.2).

For arbitrary shaped VOPs with the value of enhancement_type being "1", the shape of the reference VOP is defined as an all opaque rectangle whose size is the same as the reference layer when the shape of the reference layer is rectangular (video_object_layer_shape = "00").

When the value of ref_select_code is "11" and the value of enhancement_type is "1", modb shall always exist in each macroblock, i.e. the macroblock is not skipped even if the co-located macroblock is skipped.

7.9.1.3.5 Decoding of backward and forward shape

Backward shape and forward shape are used in the background composition process specified in subclause 8.1. The backward shape is the shape of the enhanced object at the next VOP in display order belonging to the reference layer. The forward shape is the shape of the enhanced object at the previous VOP in display order belonging to the reference layer.

For the VOPs with the value of enhancement_type being "1", backward shape is decoded when the load_backward_shape is "1" and forward shape is decoded when load_forward_shape is "1".

When the value of load_backward_shape is "1" and the value of load_forward_shape is "0", the backward shape of the previous VOP is copied to the forward shape for the current VOP. When the value of load_backward_shape is "0", the backward shape of the previous VOP is copied to the backward shape for the current VOP and the forward shape of the previous VOP is copied to the forward shape for the current VOP.

The decoding process of backward and forward shape is the same as the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = "10").

7.9.2 Spatial scalability

7.9.2.1 Binary shape spatial scalability

Refer to the subclause 7.5.4.

© ISO/IEC 2004 – All rights reserved

7.9.2.2 Texture Spatial Scalability

7.9.2.3 Base Layer and Enhancement Layer

In the case of spatial scalability, the enhancement bitstream is used to increase the resolution of the image. When the output with the lowest resolution is required, only the base layer is decoded. When the output with higher resolution is required, both the base layer and the enhancement layer are decoded.

Figure 7-54 is a diagram of the video decoding process with spatial scalability.

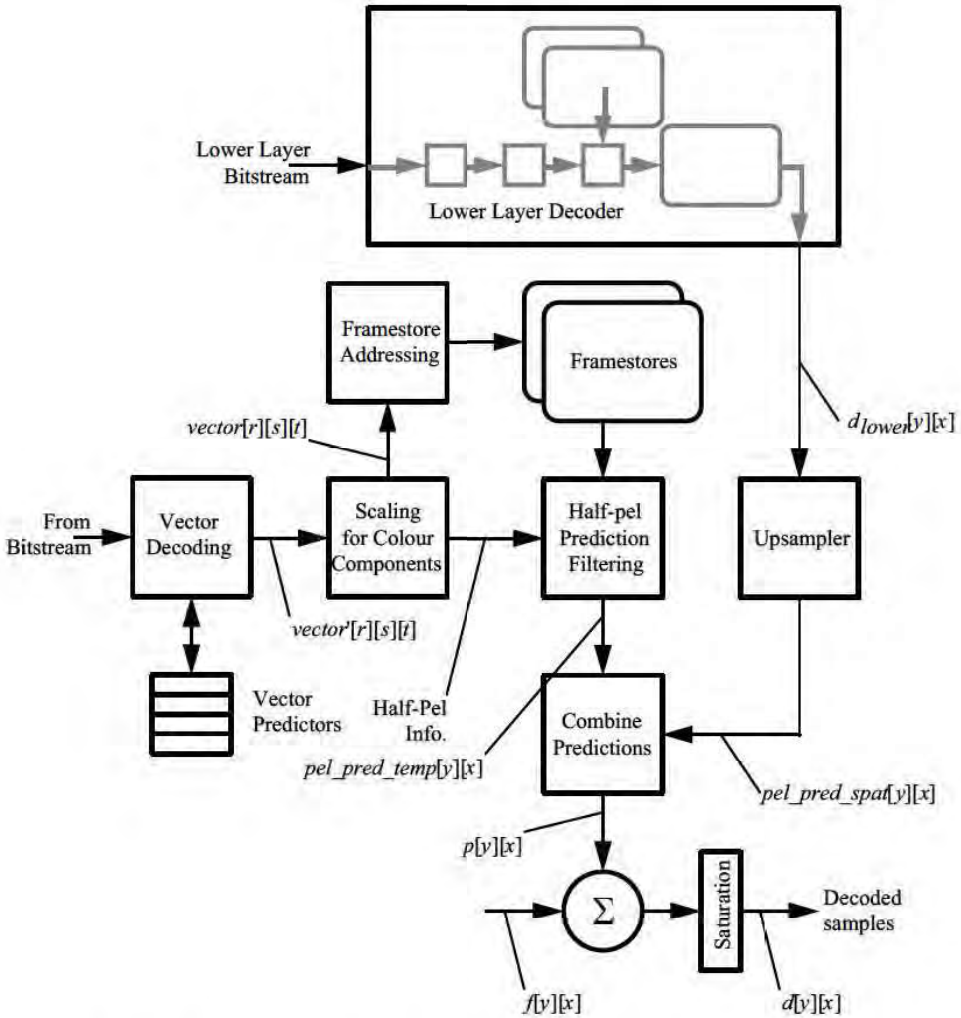


Figure 7-54 — Simplified motion compensation process for spatial scalability

7.9.2.4 Decoding of Base Layer

The decoding process of the base layer is the same as nonscalable decoding process.

© ISO/IEC 2004 – All rights reserved

7.9.2.5 Prediction in the enhancement layer

A motion compensated temporal prediction is made from reference VOPs in the enhancement layer. In addition, a spatial prediction is formed from the lower layer decoded frame ($d_{lower}[y][x]$). These predictions are selected individually or combined to form the actual prediction.

In the enhancement layer, the forward prediction in P-VOP and the backward prediction in B-VOP are used as the spatial prediction. The reference VOP is set to the temporally coincident VOP in the reference layer. The forward prediction in B-VOP is used as the temporal prediction from the enhancement layer VOP. The reference VOP is set to the most recently decoded VOP of the enhancement layer. The interpolated prediction in B-VOP is the combination of these predictions. If the reference VOP has arbitrary shape, padding process shall be done before resampling.

In the case that a macroblock is not coded, either because the entire macroblock is skipped or the specific macroblock is not coded there is no coefficient data. In this case f[y][x] is zero, and the decoded samples are the prediction, p[y][x].

7.9.2.6 Formation of spatial prediction

Forming the spatial prediction requires definition of the spatial resampling process. The formation is performed at the mid-processor. The resampling process is defined for a whole VOP, however, for decoding of a macroblock, only the 16x16 region in the upsampled VOP, which corresponds to the position of this macroblock, is needed.

The spatial prediction is made by resampling the reconstructed VOP in the reference layer to the same sampling grid as the enhancement layer. In the first step, the reconstructed VOP is subject to vertical resampling. Then, the vertically resampled image is subject to horizontal resampling. If the reference VOP in the reference lower layer has rectangular shape, the reference VOP is re-sampled without padding process. If the reference VOP in the reference lower layer has arbitrary shape, padding process shall be done before the re-sampling process. The padding process in the formation of spatial prediction is identical to the padding process in the base layer, which is described in subclause 7.5.

In the case of arbitrary shape VOPs, its size and locations change from time to time. In order to ensure the formation of the spatial prediction, it is necessary to identify the location of the resampled VOPs in the absolute coordinate system. If the reference layer is rectangular in shape, the spatial reference of the resampled VOP is set to (0,0).

In case of arbitrary shaped VOP, the VOP location of each layer should conserve the absolute location as described in Figure 7-55.

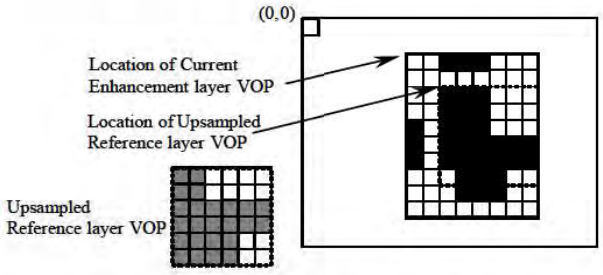


Figure 7-55 — VOP location of each layer

7.9.2.6.2 Vertical resampling

The image subject to vertical resampling, $d_{lower}[y][x]$, is resampled to the enhancement layer vertical sampling grid using linear interpolation between the sample sites according to the following formula, where vert_pic is the resulting image:

© ISO/IEC 2004 – All rights reserved

vert_pic[y_h][x] = (16 - phase) * d_{lower} [y1][x] + phase * d_{lower} [y2][x]

where

y_h = output sample coordinate in vert_pic

y1 = (y_h * vertical_sampling_factor_m) / vertical_sampling_factor_n

y2 = y1 + 1 if y1 < video_object_layer_height - 1
y1 otherwise

phase = (16 * ((y_h * vertical_sampling_factor_m) %
vertical_sampling_factor_n)) // vertical_sampling_factor_n

where video_object_layer_height is the height of the reference VOL.

If the reference layer VOP is arbitrary shaped VOP, the following definition should be used for y1 and y2.

y1 = MIN((MAX((y_h * vertical_sampling_factor_m) / vertical_sampling_factor_n,
VOP_vertical_mc_spatial_ref)), VOP_vertical_mc_spatial_ref + vop_height-1)

y2 = MIN((MAX((y_h * vertical_sampling_factor_m) / vertical_sampling_factor_n +1,
VOP_vertical_mc_spatial_ref)), VOP_vertical_mc_spatial_ref + vop_height-1)

where vop_height is the height of the reference VOP.

Samples which lie outside the vertically upsampled reconstructed frame which are required for upsampling are obtained by border extension of the vertically upsampled reconstructed frame.

NOTE In case of rectangular shape, the calculation of phase assumes that the sample position in the enhancement layer at yh = 0 is spatially coincident with the first sample position of the lower layer. It is recognised that this is an approximation for the chrominance component if the chroma_format == 4:2:0.

7.9.2.6.3 Horizontal resampling

The image subject to horizontal resampling, $vert\_pict[y][x]$, is resampled to the enhancement layer horizontal sampling grid using linear interpolation between the sample sites according to the following formula, where hor_pic is the resulting image:

hor_pic[y][xh] = ((16 - phase) * vert_pic[y][x1] + phase * vert_pic[y][x2]) // 256

where

xh = output sample coordinate in hor_pic

x1 = (xh * horizontal_sampling_factor_m) / horizontal_sampling_factor_n

x2 = x1 + 1 if x1 < video_object_layer_width - 1
x1 otherwise

phase = (16 * ((xh * horizontal_sampling_factor_m) %
horizontal_sampling_factor_n)) // horizontal_sampling_factor_n

where video_object_layer_width is the width of the reference VOL.

If the reference layer VOP is arbitrary shaped VOP, the following definition should be used for x1 and x2.

x1 = MIN((MAX((x_h * horizontal_sampling_factor_m) / horizontal_sampling_factor_n,
vop_horizontal_mc_spatial_ref)), vop_horizontal_mc_spatial_ref + vop_width-1)

x2 = MIN((MAX((x_h * horizontal_sampling_factor_m) / horizontal_sampling_factor_n +1,
vop_horizontal_mc_spatial_ref)), vop_horizontal_mc_spatial_ref + vop_width-1)

where vop_width is the width of the reference VOP.

Samples which lie outside the lower layer reconstructed frame which are required for upsampling are obtained by border extension of the lower layer reconstructed frame.

© ISO/IEC 2004 – All rights reserved

7.9.2.7 Selection and combination of spatial and temporal predictions

The spatial and temporal predictions can be selected or combined to form the actual prediction in B-VOP. The spatial prediction is referred to as "backward prediction", while the temporal prediction is referred to as "forward prediction". The combination of these predictions can be used as "interpolate prediction". In the case of P-VOP, only the spatial prediction (prediction from the reference layer) can be used as the forward prediction. The prediction in the enhancement layer is defined in the following formulae.

```
pel_pred[y][x] = pel_pred_temp[y][x]                          (forward in B-VOP)
pel_pred[y][x] = pel_pred_spat[y][x] = hor_pict[y][x]         (forward in P-VOP and backward in B-VOP)
pel_pred[y][x] = (pel_pred_temp[y][x] + pel_pred_spat[y][x])//2   (Interpolate in B-VOP)
```

pel_pred_temp[y][x] is used to denote the temporal prediction (formed within the enhancement layer). pel_pred_spat[y][x] is used to denote the prediction formed from the lower layer. pel_pred[y][x] is denoted the resulting prediction.

7.9.2.8 Decoding process of enhancement layer

The VOP in the enhancement layer is decoded as either I-VOP, P-VOP or B-VOP.

7.9.2.8.1 Decoding of I-VOPs

The decoding process of the I-VOPs in the enhancement layer is the same as the non_scalable decoding process. ref_layer_id, ref_layer_sampling_direc, hor_sampling_factor_n, horsampling_factor_m, vertical_sampling_factor_n, vertical_sampling_factor_m and ref_select_code are ignored in the spatial scalability I-VOPs.

7.9.2.8.2 Decoding of P-VOPs

In P-VOP, the ref_select_code shall be "11", i.e., the prediction reference is set to the temporally coincident VOP in the reference layer. The reference layer is indicated by ref_layer_id in VideoObjectLayer class. In the case of spatial prediction, the motion vector shall be set to 0 at the decoding process and is not encoded in the bitstream.

A variable length codeword giving information about the macroblock type and the coded block pattern for chrominance is mcbpc. The codewords for mcbpc in the enhancement layer are the same as the base layer and shown in Table B.7. mcbpc shall be included in coded macroblocks.

The macroblock type gives information about the macroblock and which data elements are present. Macroblock types and included elements in the enhancement layer bitstream are listed in subclause B.1.1.

In the case of the enhancement layer of spatial scalability, INTER4V shall not be used. The macroblock of INTER or INTER+Q is encoded using the spatial prediction.

7.9.2.8.3 Decoding of B-VOPs

In B-VOP, the ref_select_code shall be "00", i.e., the backward prediction reference is set to the temporally coincident VOP in the reference layer, and the forward prediction reference is set to the most recently decoded VOP in the enhancement layer. In the case of spatial prediction, the motion vector shall be set to 0 at the decoding process and is not encoded in the bitstream.

modb shall be present in coded macroblocks belonging to B-VOPs. The codeword is the same as the base layer and is shown in Table B.3. In case mb_type does not exist the default shall be set to "Forward MC" (prediction from the last decoded VOP in the same reference layer). modb shall be encoded in all macroblocks. If its value is equal to '1', further information is not transmitted for this macroblock. The decoder treats the prediction of this macroblock as forward MC with motion vector equal to zero.

mb_type is present only in coded macroblocks belonging to B-VOPs. The mb_type gives information about the macroblock and which data elements are present. mb_type and included elements in the enhancement layer bitstream are listed in Table B.5.

© ISO/IEC 2004 – All rights reserved

In the case of the enhancement layer of spatial scalability, direct mode shall not be used. The decoding process of the forward motion vectors are the same as the base layer.

7.9.2.9 Decoding of binary shape

The decoding process of the binary shape is the same as binary shape spatial scalable decoding process described in sublcause 7.5.4.

7.10 Still texture object decoding

The block diagram of the decoder is shown in Figure 7-56.

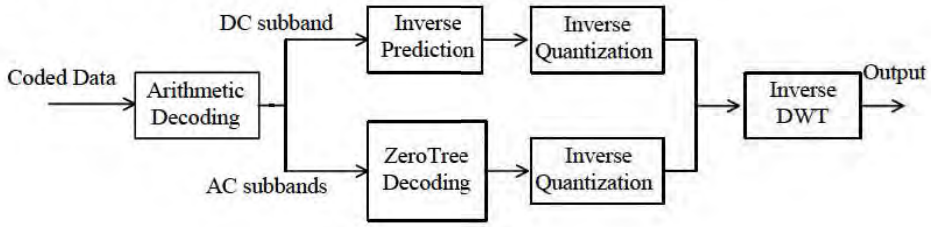


Figure 7-56 — Block diagram of the decoder

The basic modules of a zero-tree wavelet based decoding scheme are as follows:

Arithmetic decoding of the DC subband using a predictive scheme.
Arithmetic decoding of the bitstream into quantised wavelet coefficients and the significance map for AC subbands.
Zero-tree decoding of the higher subband wavelet coefficients.
Inverse quantisation of the wavelet coefficients.
Composition of the texture using inverse discrete wavelet transform (IDWT).

7.10.1 Decoding of the DC subband

The wavelet coefficients of DC subband are decoded independently from the other bands. First the quantisation step size decoded, then the offset value of the quantisation indices "band_offset" and the maximum value of the differential quantisation indices "band_max_value" are decoded from bitstream. The parameter "band_offset" is negative or zero integer and the parameter "band_max_value" is a positive integer, so only the magnitude of these parameters are read from the bitstream.

The arithmetic model is initialized with a uniform distribution. Then, the differential quantisation indices are decoded using the arithmetic decoder in a bitplane-by-bitplane fashion, from MSB to LSB. In each bitplane, the indices are decoded in a raster scan order, starting from the upper left index and ending with the lowest right one. Separate probability model for each bitplane and color component is used. The model is updated with the decoding of each bits of the predicted wavelet quantisation index to adopt the probability model to the statistics of DC band.

The "band_offset" is added to all the decoded quantisation indices, and an inverse predictive scheme is applied. Each of the current indices w_X is predicted from three quantisation indices in its neighborhood, i.e. w_A, w_B, and w_C (see Figure 7-57), and the predicted value is added to the current decoded coefficient. That is,

if $(|w_A - w_B|) < |w_B - w_C|)$
$\hat{w}x = wC$
else
$\hat{w}x = wA$
$w_x = w_x + \hat{w}x$

© ISO/IEC 2004 – All rights reserved

If any of nodes A, B or C is not in the image, its value is set to zero for the purpose of the inverse prediction. Finally, the inverse quantisation scheme is applied to all decoded values to obtain the wavelet coefficients of DC band.

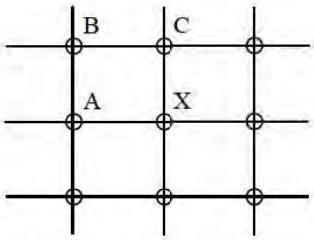


Figure 7-57 — DPCM decoding of DC band coefficients

7.10.2 ZeroTree Decoding of the Higher Bands

The zero-tree algorithm is based on the strong correlation between the amplitudes of the wavelet coefficients across scales, and on the idea of partial ordering of the coefficients. The coefficient at the coarse scale is called the *parent*, and all coefficients at the same spatial location, and of similar orientation, at the next finer scale are that parent's children. Figure 7-58 shows a wavelet tree where the parents and the children are indicated by squares and connected by lines. Since the DC subband (shown at the upper left in Figure 7-58) is coded separately using a DPCM scheme, the wavelet trees start from the adjacent higher bands.

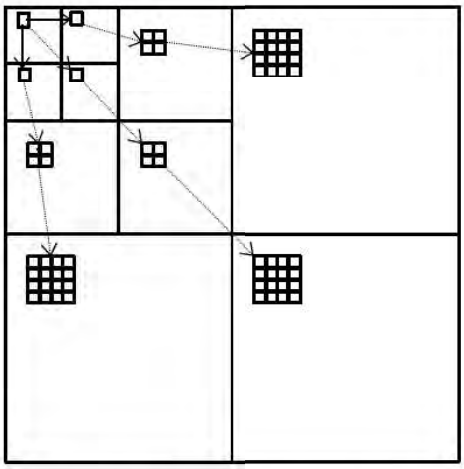


Figure 7-58 — Parent-child relationship of wavelet coefficients

In transform-based coding, it is typically true that a large percentage of the transform coefficients are quantised to zero. A substantial number of bits must be spent either encoding these zero-valued quantised coefficients, or else encoding the location of the non-zero-valued quantised coefficients. ZeroTree Coding uses a data structure called a *zerotree*, built on the parent-child relationships described above, and used for encoding the location of non-zero quantised wavelet coefficients. The zerotree structure takes advantage of the principle that if a wavelet coefficient at a coarse scale is "insignificant" (quantised to zero), then all wavelet coefficients of the same orientation at the same spatial location at finer wavelet scales are also likely to be "insignificant". Zerotrees exist at any tree node where the coefficient is zero and all its descendents are also zero.

The wavelet trees are efficiently represented and coded by scanning each tree from the root in the 3 lowest AC bands through the children, and assigning one of four symbols to each node encountered: *zerotree root (ZTR)*, *value zerotree root (VZTR)*, *isolated zero (IZ)* or *value (VAL)*. A *ZTR* denotes a coefficient that is the root of a

© ISO/IEC 2004 – All rights reserved

zerotree. Zerotrees do not need to be scanned further because it is known that all coefficients in such a tree have amplitude zero. A *VZTR* is a node where the coefficient has a nonzero amplitude, and all four children are zerotree roots. The scan of this tree can stop at this symbol. An *IZ* identifies a coefficient with amplitude zero, but also with some nonzero descendant. A *VAL* symbol identifies a coefficient with amplitude nonzero, and with some nonzero descendant. The symbols and quantised coefficients are losslessly encoded using an adaptive arithmetic coder. Table 7-19 shows the mapping of indices of the arithmetic decoding model into the zerotree symbols:

Table 7-19 — The indexing of zerotree symbols

index	Symbol
0	IZ
1	VAL
2	ZTR
3	VZTR

In order to achieve a wide range of scalability levels efficiently as needed by different applications, three different zerotree scanning and associated inverse quantisation methods are employed. The encoding mode is speficied in bitstream with quantisation_type field as one of 1) single_quant, 2) multi_quant or 3) bilevel_quant:

Table 7-20 — The quantisation types

code	quantisation_type
01	single_quant
10	multi _quant
11	bilevel_quant

In single_quant mode, the bitstream contains only one zero-tree map for the wavelet coefficients. After arithmetic decoding, the inverse quantisation is applied to obtain the reconstructed wavelet coefficients and at the end, the inverse wavelet transform is applied to those coefficients.

In multi_quant mode, a multiscale zerotree decoding scheme is employed. Figure 7-59 shows the concept of this technique.

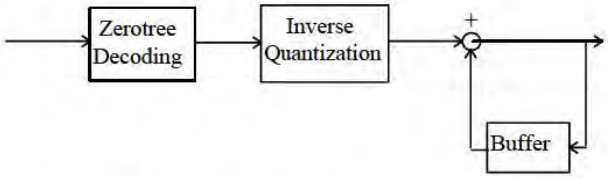


Figure 7-59 — Multiscale zerotree decoding

The wavelet coefficients of the first spatial (and/or SNR) layer are read from the bitstream and decoded using the arithmetic decoder. Zerotree scanning is used for decoding the significant maps and quantised coefficients and locating them in their corresponding positions in trees. These values are saved in the buffer to be used for quantisation refinement at the next scalability layer. Then, an inverse quantisation is applied to these indices to obtain the quantised wavelet coefficients. An inverse wavelet transform can also be applied to these coefficients to obtain the first decoded image. The above procedure is applied for the next spatial/SNR layers.

The bilevel_quant mode enables fine granular SNR scalability by encoding the wavelet coefficients in a bitplane-by-bitplane fashion. This mode uses the same zerotree symbols as the multi_quant mode. In this mode, a zero-tree map is decoded for each bitplane, indicating which wavelet coefficients are nonzero relative to that bitplane. The inverse quantisation is also performed bitplane by bitplane. After the zero-tree map, additional bits are decoded to refine the accuracy of the previously decoded coefficients.

© ISO/IEC 2004 – All rights reserved

7.10.2.1 Zerotree Scanning

In all the three quantisation modes, the wavelet coefficients are scanned either in the tree-depth fashion or in the band-by-band fashion. In the tree-depth scanning order all coefficients of each tree are decoded before starting decoding of the next tree. In the band-by-band scanning order, all coefficients are decoded from the lowest to the highest frequency subbands.

Figure 7-60 shows the scanning order for a 16x16 image, with 3 levels of decomposition. In this figure, the indices 0,1,2,3 represent the DC band coefficients which are decoded separately. The remaining coefficients are decoded in the order shown in this figure. As an example, indices 4,5,..., 24 represent one tree. At first, coefficients in this tree are decoded starting from index 4 and ending at index 24. Then, the coefficients in the second tree are decoded starting from index 25 and ending at 45. The third tree is decoded starting from index 46 and ending at index 66 and so on.

0	1	4	67	5	6	68	69	9	10	13	14	72	73	76	77
2	3	130	193	7	8	70	71	11	12	15	16	74	75	78	79
25	88	46	109	131	132	194	195	17	18	21	22	80	81	84	85
151	214	172	235	133	134	196	197	19	20	23	24	82	83	86	87
26	27	89	90	47	48	110	111	135	136	139	140	198	199	202	203
28	29	91	92	49	50	112	113	137	138	141	142	200	201	204	205
152	153	215	216	173	174	236	237	143	144	147	148	206	207	210	211
154	155	217	218	175	176	238	239	145	146	149	150	208	209	212	213
30	31	34	35	93	94	97	98	51	52	55	56	114	115	118	119
32	33	36	37	95	96	99	100	53	54	57	58	116	117	120	121
38	39	42	43	101	102	105	106	59	60	63	64	122	123	126	127
40	41	44	45	103	104	107	108	61	62	65	66	124	125	128	129
156	157	160	161	219	220	223	224	177	178	181	182	240	241	244	245
158	159	162	163	221	222	225	226	179	180	183	184	242	243	246	247
164	165	168	169	227	228	231	232	185	186	189	190	248	249	252	253
166	167	170	171	229	230	233	234	187	188	191	192	250	251	254	255

Figure 7-60 — Tree depth scanning order of a wavelet block in the all three modes

Figure 7-61 shows that the wavelet coefficients are scanned in the subband by subband fashion, from the lowest to the highest frequency subbands. Figure 7-62 shows an example of decoding order for a 16x16 image with 3 levels of decomposition for the subband by subband scanning. The DC band is located at upper left corner (with indices 0, 1,2, 3) and is decoded separately as described in DC band decoding. The remaining coefficients are decoded in the order which is shown in the figure, starting from index 4 and ending at index 255. In multi_quant mode, at first scalability layer, the zerotree symbols and the corresponding values are decoded for the wavelet coefficients of that scalability layer. For the next scalability layers, the zerotree map is updated along with the corresponding value refinements. In each scalability layer, a new zerotree symbol is decoded for a coefficient only if it was decoded as ZTR or IZ in previous scalability layer or it is currently in SKIP mode. A node is said to be in SKIP mode when the number of quantisation refinement levels for the current scalability layer is one. The detailed description of the refinement of quantisation level is found in subclause 7.10.3. If the coefficient was decoded as VAL in previous layer and it is not currently in SKIP mode, a VAL symbol is also assigned to it at the current layer and only its refinement value is decoded from bitstream.

© ISO/IEC 2004 – All rights reserved

0	1	4	7	16	17	28	29	64	65	68	69	112	113	116	117
2	3	10	13	18	19	30	31	66	67	70	71	114	115	118	119
5	8	6	9	40	41	52	53	72	73	76	77	120	121	124	125
11	14	12	15	42	43	54	55	74	75	78	79	122	123	126	127
20	21	32	33	24	25	36	37	160	161	164	165	208	209	212	213
22	23	34	35	26	27	38	39	162	163	166	167	210	211	214	215
44	45	56	57	48	49	60	61	168	169	172	173	216	217	220	221
46	47	58	59	50	51	62	63	170	171	174	175	218	219	222	223
80	81	84	85	128	129	132	133	96	97	100	101	144	145	148	149
82	83	86	87	130	131	134	135	98	99	102	103	146	147	150	151
88	89	92	93	136	137	140	141	104	105	108	109	152	153	156	157
90	91	94	95	138	139	142	143	106	107	110	111	154	155	158	159
176	177	180	181	224	225	228	229	192	193	196	197	240	241	244	245
178	179	182	183	226	227	230	231	194	195	198	199	242	243	246	247
184	185	188	189	232	233	236	237	200	201	204	205	248	249	252	253
186	187	190	191	234	235	238	239	202	203	206	207	250	251	254	255

Figure 7-61 — The band-by-band scanning order for all three modes

In bilevel_quant mode, the band by band scanning is also employed, similar to the multi_quant mode. When bi-level quantisation is applied, the coefficients that are already found significant are replaced with zero symbols for the purpose of zero-tree forming in later scans.

7.10.2.2 Entropy Decoding

The zero-tree (or type) symbols, quantised coefficient values (magnitude and sign), and residual values (for the multi quant mode) are all decoded using an adaptive arithmetic decoder with a given symbol alphabet. The arithmetic decoder adaptively tracks the statistics of the zerotree symbols and decoded values. For both the single quant and multi quant modes the arithmetic decoder is initialized at the beginning of each color loop for band-by-band scanning and at the beginning of the tree-block loop for tree-depth scanning. Therefore, the decoder is initialized three times in each band for band-by-band scanning and the decoder is initialized once before going to the tree block-loop for tree-depth scanning. In order to avoid start code emulation, the arithmetic encoder always starts with stuffing one bit '1' at the beginning of the entropy encoding. It also stuffs one bit '1' immediately after it encodes every 22 successive '0's. It stuffs one bit '1' to the end of bitstream in the case in which the last output bit of arithmetic encoder is '0'. Thus, the arithmetic decoder reads and discards one bit before starts entropy decoding. During the decoding, it also reads and discards one bit after receiving every 22 successive '0's. The arithmetic decoder reads one bit and discards it if the last input bit to the arithmetic decoder is '0'. To reduce the number of bitplanes in encoding the magnitude of quantised coefficient values, the encoder subtracts the magnitude by one first. Thus the decoder adds one back after decoding the magnitude of quantised coefficient values.

The context models used for SQ and MQ are identical to the ones used in BQ mode.

For both scanning orders in the single quant and multi_quant modes separate probability models are kept for each color and wavelet decomposition layer for the type and sign symbols while separate probability models are kept for each color, wavelet decomposition layer, *and* bitplane for the magnitude and residual symbols. All the models are initialized with a uniform probability distribution.

© ISO/IEC 2004 – All rights reserved

The models and symbol sets for the non-zerotree type quantities to be decoded are as follows:

Model	Possible Values
Sign	POSITIVE (0), NEGATIVE (1)
Magnitude	0, 1
Residual	0, 1

The possible values of the magnitudes and residuals are only 0 or 1 because each bitplane is being decoded separately. The non-residual values are decoded in two steps. First, the absolute value is decoded in a bitplane fashion using the *magnitude* probability models, then its sign is decoded.

For the decoding of the type symbols different probability models are kept for the leaf and non-leaf coefficients. For the multi quant mode, context modeling, based on the zerotree type of the coefficient in the previous scalability layer, is used. The different zerotree type models and their possible values are as follows:

Context and Leaf/Non-Leaf	Possible Values
INIT	ZTR (2), IZ (0), VZTR (3), VAL (1)
ZTR	ZTR (2), IZ (0), VZTR (3), VAL (1)
ZTR DESCENDENT	ZTR (2), IZ (0), VZTR (3), VAL (1)
IZ	IZ (0), VAL (1)
LEAF INIT	ZTR (0), VZTR (1)
LEAF ZTR	ZTR (0), VZTR (1)
LEAF ZTR DESCENDENT	ZTR (0), VZTR (1)

For the single quant mode only the INIT and LEAF INIT models are used. For the multi quant mode for the first scalability layer only the INIT and LEAF INIT models are used. Subsequent scalability layers in the multi quant mode use the context associated with the type. If a new spatial layer is added then the contexts of all previous leaf band coefficients are switched to the corresponding non-leaf contexts. The coefficients in the newly added bands use the LEAF INIT context. The residual models are used to decode the coefficient refinements if in the previous layer, a *VZTR or VAL* symbol was assigned. If a node is currently not in SKIP mode (meaning that no refinement is being done for the coefficient – see subclause 7.10.3 on inverse quantisation for details) only the magnitude of the refinements are decoded as these values are always zero or positive integers.

If a node is in SKIP mode, then its new zerotree symbol is decoded from bitstream, but no value is decoded for the node and its value in the current scalability layer is assumed to be zero.

States in Previous Bitplane	Possibilities in current bitplane
ZTR	ZTR, VZTR, IZ, VAL
VZTR	SKIP
IZ	IZ, VAL
VAL	SKIP
DZTR	ZTR, VTRZ, IZ, VAL

© ISO/IEC 2004 – All rights reserved

For the bi-level quantisation mode, the zero-tree map is decoded for each bitplane, indicating which wavelet coefficients are zeros relative to the current quantisation step size. Different probability models for the arithmetic decoder are used and updated according to the local contexts. For decoding the zerotree symbols, five context models are used, which are dependent on the status of the current wavelet coefficients in the zerotree formed in the previous bitplane decoding. Specifically, the five models correspond to the following contexts of the current wavelet coefficient:

· IZ: the previous zerotree symbol is Isolated Zero.

· VAL: the previous zerotree symbol is Value.

· ZTR: the previous zerotree symbol is Zerotree Root.

· VZTR: the previous zerotree symbol is valued zerotree root.

· DZTR: in previous bitplane, the current coefficient is a descendant of a zerotree root

The additional symbol DZTR is used for switching the models only, where DZTR refers to the descendant of a ZTR symbol. The context symbols DZTR can be inferred from the decoding process and are not included in the bitstream. They are used for switching the models only. At the beginning of the decoding the first bitplane, the contexts of the coefficients are initialized to be DZ. For the highest subband, only IZ and VAL are possible (no ZTR and VZTR are possible). Therefore, we initialize the arithmetic model for the last band differently (with zero probablility for ZTR and VZTR symbols).

For decoding the sign information, another context model (the sign model) is used and updated. For decoding the refinement bits, another statistical model (the refinement model) is used

Each decomposition levels have their own separate arithmetic models. Therefore, the above decoding process applies to each decomposition levels. All models are initialized at the beginning of coding each bitplane.

After the zero-tree map, additional bits are received to refine the accuracy of the coefficients that are already marked significant by previously received information at the decoder. For each significant coefficient, the 1-bit bi-level quantised refinement values are entropy coded using the arithmetic coder.

7.10.3 Inverse Quantisation

Different quantisation step sizes (one for each color component) are specified for each level of scalability. The quantiser of the DC band is a uniform mid-step quantiser with a dead zone equal to the quantisation step size. The quantisation index is a signed integer number and the quantisation reconstructed value is obtained using the following equation:

$$V = id * Qdc,$$

where V is the reconstructed value, *id* is the decoded index and Qdc is the quantisation step size.

All the quantisers of the higher bands (in all quantisation modes) are uniform mid-step quantiser with a dead zone 2 times the quantisation step size. For the single quantisation mode, the quantisation index is an signed integer. The reconstructed value is obtained using the following algorithm:

```
if (id == 0)
    V =0;
else if ( id > 0 )
    V = id*Q+Q/2;
else
    V = id*Q-Q/2;
```

where V is the reconstructed value, id is the decoded index and Q is the quantisation step size.

© ISO/IEC 2004 – All rights reserved

In the multi- quantisation mode each SNR layer within each spatial layer has an associated quantisation step-size value (Q value). These different Q Values are used for SNR scalability. A lower Q Value will result in a more accurate reconstruction.

If a coefficient is in a given spatial layer it is also in all higher numbered spatial layers. SNR scalability may be continued on these coefficients in the higher numbered spatial layers in the same way as is done in the spatial layer the coefficient first arises in. Thus, we can think of all the coefficients which first arise in a particular spatial layer as having a corresponding sequence of Q Values (call it a Q Sequence). The Q Sequence is made up of the quantisation values for all SNR layers in the spatial layer the coefficient first arises in plus the quantisation values in all SNR layers in all higher spatial layers. The order is from lower to higher numbered spatial layers and from lower to higher numbered SNR layers within each spatial layer.

EXAMPLE

Let the quantisation value of the *i*-th spatial layer and the *j*-th SNR layer be denoted by *Q(i,j)*. Assume we have the following scenario:

Spatial Layer	SNR Layer 0	1	2	.
0	*Q(0,0)*	*Q(0,1)*	*Q(0,2)*	
1	*Q(1,0)*	*Q(1,1)*	*Q(1,2)*	
2	*Q(2,0)*	*Q(2,1)*	*Q(2,2)*	

The Q Sequence which will be used to quantise all coefficients which first arise in spatial layer 0 is:

<Q(0,0) Q(0,1) Q(0,2) Q(1,0) Q(1,1) Q(1,2) Q(2,0) Q(2,1) Q(2,2)>

while the sequence for all coefficients first arising in spatial layer 1 is:

<Q(1,0) Q(1,1) Q(1,2) Q(2,0) Q(2,1) Q(2,2)>

and the sequence for all coefficients first arising in spatial layer 2:

<Q(2,0) Q(2,1) Q(2,2)>.

As in the single- quantisation case we would like to have a uniform quantiser for all layers. Due to the manner in which the Q Values are used to achieve scalability (described below), in order to have a (approximately) uniform quantiser at each layer, we may have to revise the Q Values extracted from the bitstream before reconstruction. This revision is necessary if the Q Values within each Q Sequence are not integer multiples of one another or if Q Value is greater than a Q Value occurring earlier in the Q Sequence.

EXAMPLES

Q Sequences needing no revision: *<24 8 2>* and *<81 81 27>*.

Q Sequences needing revision: *<31 9 2>* (non-integer multiples) and *<81 162 4>* (increasing Q Value).

If a coefficient's quantisation indices have been zero for all previous scalability layers (spatial and SNR) or if it is the first scalability layer, then the reconstruction is the similar to the single-quantisation mode described above. The difference is in that the refined Q Values may be used instead of the ones extracted from the bitstream. The refinement process is described below in steps 1 and 2. If there has been a non-zero quantisation index in a previous scalability layer then the quantisation index specifies a refinement of the previous quantisation. The indices are then called residuals. For every coefficient and scalability layer we know (1) the quantisation interval where the coefficient occurred in the last scalability layer (both size and location), (2) the spatial layer the coefficient first arose in (and thus, which Q Sequence to use), (3) the current Q Value and the previous Q Value in the Q Sequence, and (4) the refinment (if any) of the previous Q Value.

The reconstruction of the residual is calculated in the following five steps.

© ISO/IEC 2004 – All rights reserved

Step 1: Calculation of the Number of Refinement Intervals

The quantisation interval which was indexed in the previous layer is to be partitioned into disjoint intervals. The number of these "refinement" intervals is calculated based solely on the current Q Value (call it *curQ*) and the previous Q Value (call it *prevQ*). Note that *prevQ* may have been revised as mentioned above. Letting *m* be the number of refinement intervals we calculate

m = ROUND(prevQ÷curQ)
where ROUND(x) = MAX(nearest integer of x, 1)).

If *m = 1*, no refinement is needed and no value will have been sent. If, at a certain scalability layer, a node has *m=1* then it is said to be in SKIP mode. Thus, steps 2, 3, and 4 need not be performed for the coefficient.

Step 2: Calculation of the Maximum Refinement Interval Size

Using the number of refinement intervals, the current Q Value, *curQ*, is revised (if necessary).

curQ = CEIL(prevQ ÷ m)
where CEIL rounds up to the nearest integer.

curQ represents the maximum size of the intervals in the partition. Since *prevQ* is the previous layer's *curQ* (see step 5), we see that *prevQ* represents the maximum size of the intervals in the partition used in the previous scalability layer.

Step 3: Construction of Refinement Partition

Using the values *m* and *curQ* calculated above and the size of quantisation interval where the coefficient occurred in the last scalability layer, we form the refinement partition.

The previous layer's quantisation interval is partitioned into *m* intervals which are of size *curQ* or *curQ-1*. The residual will be one of the values *{0, 1, ..., m-1}* which represent an index into this partition. A lower number index corresponds to an interval in the partition covering lower magnitude values. If the partition is made up of different size intervals (*curQ* and *curQ-1*) then the *curQ* size intervals correspond to the lower indices. Some combination of m *curQ* and *curQ-1* interval sizes are sufficient to cover the previous quantisation interval. From step 2 we know that the previous quantisation interval is of size *prevQ* or *prevQ-1*.

Step 4: Calculation of Reconstructed Value

The interval in the partition indexed by the residual is mapped to the reconstruction value. The reconstruction is just the middle point of the interval in the partition that the residual indexes. That is, if *PartStart* is the start of the interval in the partition which is indexed by the residual, *PartStartSize* is the size of the interval, *sign* is the corresponding sign (known from prior scalability layers), and // is integer division then the reconstructed value is:

PartStart + sign*(PartStartSize-1)//2

Step 5: Assignment of Maximum Size

If there is another scalability layer then *prevQ* is assigned the value of *curQ*.

Note that since steps 1, 2, and 5 depend entirely on the Q Values found in the Q Sequences they only need to be done once in each scalability layer for each Q Sequence being used in the current spatial layer.

FOUR EXAMPLES

In the examples:

1. let the Q Values be Q1 , Q2 , and Q3,
2. let two sample coefficients to be quantised be *C1* and *C2,*
3. let *Cq1* and *Cq2* denote the corresponding quantised coefficients or residuals, and
4. let *iC1* and *iC2* denote the corresponding reconstructed coefficient values.

© ISO/IEC 2004 – All rights reserved

1. Q Values not in need of revision

Q1 = 24 , Q2 = 8, Q3 = 2,
C1 = 16, and C2 = 28.

At first scalability layer we have

Cq1 = C1/Q1 = 0
iCq1 = 0
Cq2 = C2/Q1 = 1
iCq2 = 35

At second scalability layer we have

prevQ = Q1 = 24
curQ = Q2 = 8
m = ROUND(prevQ÷curQ) = ROUND(24÷8) = 3
curQ = CEIL(prevQ÷m) = CEIL(24÷3) = 8
partition sizes = {8, 8, 8}
Cq1 = C1/curQ = 2
iCq1 = 19
Cq2 = 0 (residual)
iCq2 = 27

At third scalability layer we have

prevQ = curQ = 8
curQ = Q3 = 2
m = ROUND(prevQ÷curQ) = ROUND(8÷2) = 4
curQ = CEIL(prevQ÷m) = CEIL(8÷2) = 4
partition sizes = {2, 2, 2, 2}
Cq1 = 0 (residual)
iCq1 = 16
Cq2 = 2 (residual)
iCq2 = 28

2. Q Values not in need of revision

Q1 = 81, Q2 = 81, Q3 = 27,
C1 = 115 , and C2 = 28.

At first scalability layer we have

Cq1 = C1/Q1 = 1
iCq1 = 121
Cq2 = C2/Q1 = 0
iCq2 = 0

At second scalability layer we have

prevQ = Q1 = 81
curQ = Q2 = 81
m = ROUND(prevQ÷curQ) =ROUND(81÷81) = 1
curQ = CEIL(prevQ÷m) = CEIL(81÷1) = 81
partition sizes = {81} (no refinement needed)
Cq1 = 0 (residual)
iCq1 = 121
Cq2 = C2/curQ = 0
iCq2 = 0

© ISO/IEC 2004 – All rights reserved

At third scalability layer we have

prevQ = curQ = 81
curQ = Q3 = 27
m = ROUND(prevQ÷curQ) = ROUND(81÷27) = 3
curQ = CEIL(prevQ÷m) = CEIL(81÷3) = 27
partition sizes = {27, 27, 27}
Cq1 = 1 (residual)
iCq1 = 121
Cq2 = C2/curQ = 1
iCq2 = 40

3. Q Values in need of revision

Q1 = 31 , Q2 = 9, Q3 = 2,
C1 = 115 , and C2 = 5.

At first scalability layer we have

Cq1 = C1/Q1 = 3
iCq1 = 108
Cq2 = C2/Q1 = 0
iCq2 = 0

At second scalability layer we have

prevQ = Q1 = 31
curQ = Q2 = 9
m = ROUND(prevQ÷curQ) = ROUND(31÷9) = 3
curQ = CEIL(prevQ÷m) = CEIL(31÷3) = 11
partition sizes = {11, 10, 10}
Cq1 = 2 (residual)
iCq1 = 118
Cq2 = C2/curQ = 0
iCq2 = 0

At third scalability layer we have

prevQ = curQ = 11
curQ = Q3 = 2
m = ROUND(prevQ÷curQ) = ROUND(11÷2) = 6
curQ = CEIL(prevQ÷m) = CEIL(11÷6) = 2
partition sizes = {2, 2, 2, 2, 2,1} if value occurs in level with size 11
partition sizes = {2, 2, 2, 2, 1, 1} if value occurs in level with size 10
Cq1 = 0 (residual)
iCq1 = 114
Cq2 = C2/curQ = 2
iCq2 = 4

4. Q Values in need of revision

Q1 = 81, Q2 = 162, Q3 = 4,
C1 = 115 , and C2 = 5.

© ISO/IEC 2004 – All rights reserved

At first scalability layer we have

Cq1 = C1/Q1 = 1
iCq1 = 121
Cq2 = C2/Q1 = 0
iCq2 = 0

At second scalability layer we have

prevQ = Q1 = 81
curQ = Q2 = 162
m = ROUND(prevQ÷curQ) = ROUND(81÷162) = 1
curQ = CEIL(prevQ÷m) = CEIL(81÷1) = 81
partition sizes = {81} (no refinement needed)
Cq1 = 0 (not used)
iCq1 = 121
Cq2 = C2/curQ = 0 (not used)
iCq2 = 0

At third scalability layer we have

prevQ = curQ = 81
curQ = Q3 = 4
m = ROUND(prevQ÷curQ) = ROUND(81÷4) = 20
curQ = CEIL(prevQ÷m) = CEIL(81÷20) = 5
partition sizes = {5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}
Cq1 = 8 (residual)
iCq1 = 121
Cq2 = C2/curQ = 1
iCq2 = 6

It is important to note that coefficients which first arose in different spatial layers may use different *prevQ* and *curQ* values. They are being quantised from different lists. That is, they have different corresponding Q Sequences.

7.10.3.1 Shape adaptive zerotree decoding

When the texture_object_layer_shape is not rectangular or **texture_object_layer_width**!= integer*$2^{\text{wavelet\_decomposition\_level}}$ or **texture_object_layer_height**!= integer*$2^{\text{wavelet\_decomposition\_level}}$, the inverse shape adaptve wavelet transform is chosen to reconstructed the arbitrary-shaped image object or rectangular texture object. Decoding shape adaptive wavelet coefficients is the same as decoding regular wavelet coefficients except keep track of the locations of where to put the decoded wavelet coefficients according to the shape information. or a generated mask. The mask is generated with **texture_object_layer_width*texture_object_layer_height** pixels of value 255 at the upper-left corner of a frame of size w*$2^{\text{wavelet\_decomposition\_level}}$ * h*$2^{\text{wavelet\_decomposition\_level}}$ and the rest of the pixels being value 0, where w is the smallest integer that makes w*$2^{\text{wavelet\_decomposition\_level}}$ > **texture_object_layer_width** and h is the smallest integer that makes h*$2^{\text{wavelet\_decomposition\_level}}$ > **texture_object_layer_height**. Similar to decoding of regular wavelet coefficients, the decoded zerotree symbols at a lower subband are used to determine whether decoding is needed at higher subbands. The difference is now that some zerotree nodes correspond to the pixel locations outside the shape boundary and no bits are to be decoded for these out_nodes. Root layer is defined as the lowest three AC subbands, leaf layer is defined as the highest three AC subbands. For decomposition level of one, the overlapped root layer and leaf laver shall be treated as leaf layer.

7.10.3.1.1 DC layer

The DC coefficient decoding is the same as that for rectangular image except the following,

1. Only those DC coefficients inside the shape boundary in the DC layer shall be traversed and decoded and DC coefficients outside the shape boundary may be set to zeros.
2. For the inverse DC prediction in the DC layer, if a reference coefficient (Figure 7-57) in the prediction context is outside the shape boundary, zero shall be used to form the prediction syntax.

© ISO/IEC 2004 – All rights reserved

7.10.3.1.2 Root layer

At the root layer (the lowest 3 AC bands), the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;
- the four children nodes of this node are marked "to_be_decoded" (TBD);

otherwise,

- a zerotree symbol is decoded for this node using an adaptive arithmetic decoder.

If the decoded symbol for the node is either isolated_zero (IZ) or value (VAL),

- the four children nodes of this node are marked TBD;

otherwise,

- the symbol is either zerotree_root (ZTR) or valued_zerotree_root (VZTR) and the four children nodes of this node are marked "no_code" (NC).

If the symbol is VAL or VZTR,

- a non-zero wavelet coefficient is decoded for this node by root model;

otherwise,

- the symbol is either IZ or ZTR and the wavelet coefficient is set to zero for this node.

7.10.3.1.3 Between root and leaf layer

At any layer between the root layer and the leaf layer, the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;
- the four children nodes of this node are marked as either TBD or NC depending on whether this node itself is marked TBD or NC respectively;

otherwise, if it is marked NC,

- no bits are decoded for this node;
- the wavelet coefficient is set to zero for this node;
- the four children nodes are marked NC;

otherwise,

- a zerotree symbol is decoded for this node using an adaptive arithmetic decoder.

If the decoded symbol for the node is either isolated_zero (IZ) or value (VAL),

- the four children nodes of this node are marked TBD;

otherwise,

- the symbol is either zerotree_root (ZTR) or valued_zerotree_root (VZTR) and the four nodes of this node are marked "no_code" (NC).

If the symbol is VAL or VZTR,

- a non-zero wavelet coefficient is decoded for this node by valnz model;

otherwise,

- the symbol is either IZ or ZTR and the wavelet coefficient is set to zero for this node.

© ISO/IEC 2004 – All rights reserved

7.10.3.1.4 Leaf layer

At the leaf layer, the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;

otherwise, if it is marked NC,

- no bits are decoded for this node;
- the wavelet coefficient is set to zero for this node;

otherwise,

- * a wavelet coefficient is decoded for this node by valz adaptive arithmetic model;

7.10.3.2 Shape decomposition

The shape information for both shape adaptive zerotree decoding and the inverse shape adaptive wavelet transform is obtained by decomposing the reconstructed shape from the shape decoder. Assuming binary shape with 0 or 1 indicating a pixel being outside or inside the arbitrarily shaped object, the shape decomposition procedure can be described as follows:

- For each horizontal line, collect all even-indexed shape pixels together as the shape information for the horizontal low-pass band and collect all odd-indexed shape pixels together as the shape information for the horizontal high-pass band, except for the special case where the number of consecutive 1's is one.
- For an isolated 1 in a horizontal line, whether at an even-indexed location or at an odd-indexed location, it is always put together with the shape pixels for the low-pass band and a 0 is put at the corresponding position together with the shape pixels for the high-pass band.
- Perform the above operations for each vertical line after finishing all horizontal lines.
- Use the above operations to decompose the shape pixels for the horizontal and vertical low-pass band further until the number of decomposition levels is reached.

7.10.4 Still Texture Error Resilience

In error resilience mode, start code will always be used. The decoder is expecting packetized bitstream. The first packet is the image header information, then the shape coding information (for shape coding combined with error resilience, each scalable shape layer will be one packet starting with the texture_shape_layer_start_code), followed by the packets for DC and AC bands. Each packet start with texture marker (texture_marker = '00000000000000001') or a startcode (if the packet is the first in the layer), followed by the number of the first and last texture units encoded in the texture packet, named TU_first and TU_last, and by one bit header_extension_code, which is optionally followed by header information.

The TU_first and TU_last provide absolute reference for the texture units in the packet. This parameter specifies the texture unit number within a texture object. These two parameters are decoded by the function get_param(). The labeling of the texture units starts from 0 in DC band and following the same decoding order as in the non-error resilient case. In AC band, the TU number starts from 60 (> maximum number of bitplanes in DC band).

The header_extention_code bit has value 1 to indicate that additional header information is sent in the packet, zero otherwise, depending on error resilience needs. The information following the header_extention_code repeats the header information contained in the TextureObjectLayer().

Within each packet, segment_markers (a ZTR symbol coded by arithmetic coding) are expected when large segment is encountered. A segment is formed at the end of the currently decoded *sub_unit* (a part of a texture unit, to be defined in the following) whenever the total number of bits from the end of the last segment or from the beginning of the packet exceeds a threshold target_segment_length. The segment marker is decoded using the function decode_segment_marker().

© ISO/IEC 2004 – All rights reserved

7.10.4.1 Decoder Constraints

In order to support error resilience, the following constrains are used in the TOL.

To prevent error propagation, DPCM prediction in DC part is disabled.

To assure that the symbol identifying the resynch marker is unique, we impose the following constraint to the arithmetic decoder (AD): when the AD decodes the 000000000000000 (15 zero bits) symbol while decoding the wavelet coefficients, it expect the above bits to be followed by one '1' bit.

The AD has to reset all the models used in the adaptation process when starting a new texture packet. This assures that there are no dependencies across the texture packets. The models will be reset to uniform distribution.

7.10.4.2 Texture packet structure

When the TOL is operating in error resilient mode, the basic unit of a texture packet is the stream generated by encoding one texture unit. In the TOL error resilient case we define four types of texture units. Each texture unit lies entirely within one scalability layer.

7.10.4.2.1 DC band

In DC band, the texture unit is defined as a bitplane of all DC coefficients, starting from the MSB. The sub-unit is defined as one row of pixels in a texture unit.

7.10.4.2.2 Tree-depth scanning

In the tree-depth scanning mode the AC texture unit consists of the stream generated when encoding one tree structure, as specified in Figure 7-62. Black blocks represent possible resynch marker positions. The total number of texture block is a function of the image size and of the number of levels in the wavelet decomposition. The texture blocks are formed following the scanning used to encode the coefficient. This is done to avoid degradation in coding efficiency performance.

The sub_unit are defined as follows:

All pixels in levels 0 and 1 within a tree.

The $2^l \times 2^l$ block within a tree for levels l = 2, 3.

The 16x16 block within a tree for levels l >3.

An example is shown in Figure 7-63.

© ISO/IEC 2004 – All rights reserved

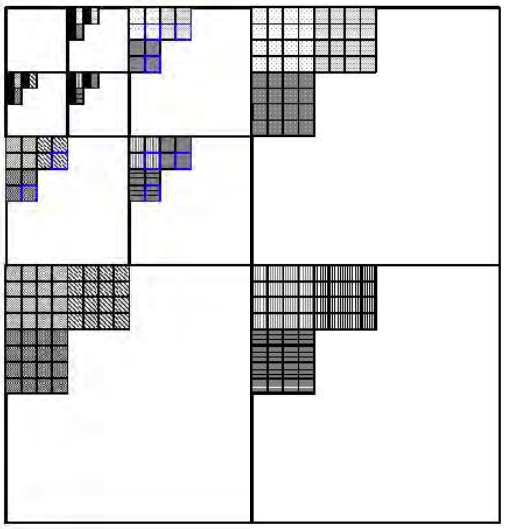


**Figure 7-62 — Texture units for the single_quant mode.
All coefficients with the same shade belong into the same texture unit**

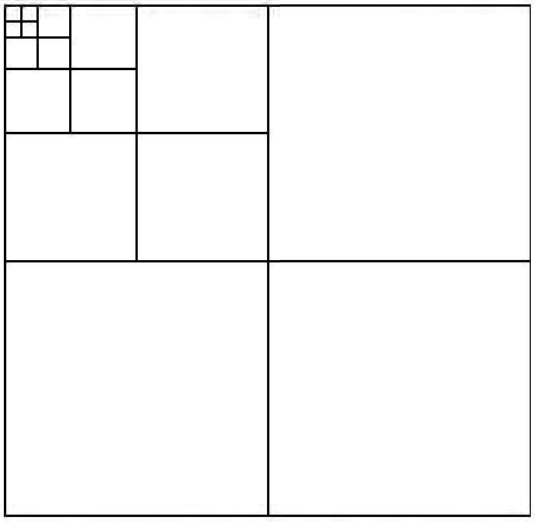


**Figure 7-63 — Sub-unit formation in a wavelet decomposed image.
Each type of shade represents a sub-unit.**

7.10.4.2.3 Subband-by-subband scanning

In the subband-by-subband scanning mode, the AC texture unit consists of the bit stream generated when encoding a slice of subband coefficients from all three subbands in the same wavelet decomposition layer, as specified in Figure 7-64. The subbands are scanned using the same scanning order used to encode the coefficients.

© ISO/IEC 2004 – All rights reserved

The sub_unit is defined in as follows:

1) All pixels at level 0 within a TU

2) The $2^l \times 2^l$ blocks within a TU for levels l = 1, 2, 3.

3) The 16x16 blocks within a TU for levels l >3.

Please note that this approach does not break the scalability, as the order of the bits generated by the encoder is preserved. Also, note that in the subband-by-subband scanning mode there is a strong dependency across subbands. The dependency consists of the having a part of the zero-tree significance map coded together with the subband coefficients. Because of this, losing a part of the subband means to lose part of the coefficients in the lower subbands.

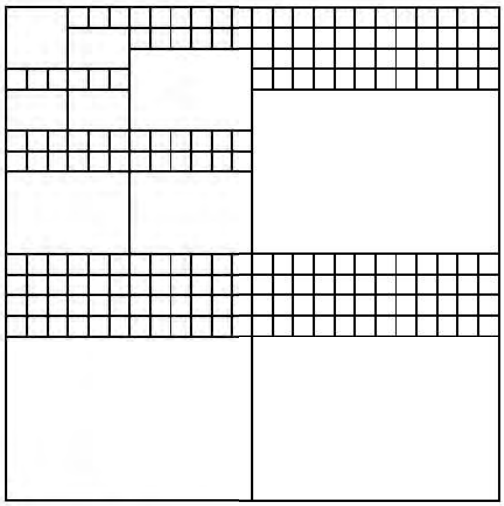


Figure 7-64 — Texture units for the band_by_band mode.
All coefficients with the same shade belong to the same texture unit.

7.10.5 Wavelet Tiling

The block diagram of the decoder supporting the Wavelet Tiling is shown in Figure 7-65.

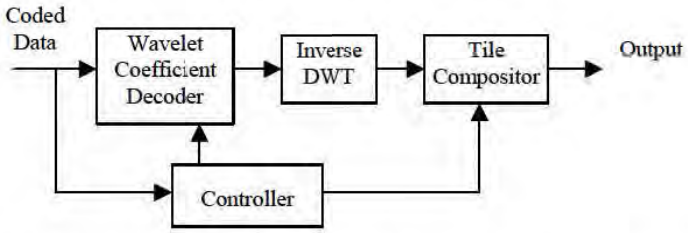


Figure 7-65 — Block diagram of the decoder with tiling option

© ISO/IEC 2004 – All rights reserved

The module 'Controller' extracts the control information from coded data. It signals the 'Wavelet coefficient Decoder' module to decode a target tile based on the information. The target tile is decided by a system layer request. 'Controller' also signals 'Tile Compositor' module in order to specify a position of a decoded tile described by 'tile_id'.

The module of 'Wavelet Coefficient Decoder' is basically equivalent to the all modules without Inverse DWT appeared in Figure 7-56. 'Inverse DWT' is equivalent to the modules appeared in Figure 7-56. 'Tile Compositor' composes the decoded tiles based on the signal from 'Controller' and output single Still Texture Object.

7.10.5.1 Structure of Tile

The structure of Still Texture Object with wavelet tiling is shown in Figure 7-66. This picture describes the case when texture_object_layer_shape=='00'. When texture_object_layer_shape=='01', object_width and object_height are used instead of texture_object_layer_width and texture_object_layer_height.

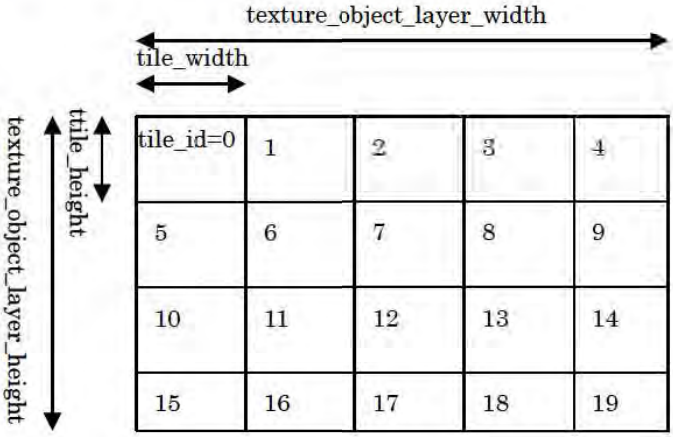


Figure 7-66 — Structure of tile

The relative position of each tile is identified by tile_id.

The number of tiles in row is computed by CEIL(texture_object_layer_width÷tile_width) and the number of tiles in column is computed by CEIL(texture_object_layer_height÷tile_height) when texture_object_layer_shape=='00'. Otherwise, object_width and object_height are used instead of texture_object_layer_width and texture_object_layer_height. Where CEIL() rounds up to the nearest integer.

7.10.5.2 The tile size of last row and last column

With regard to the tile size of last row and last column, texture_object_layer_width % tile_width and texture_object_layer_height % tile_height are used when texture_object_layer_shape=='00' if texture_object_layer_width and texture_object_layer_height are not a multiple of tile_width and tile_height. When texture_object_layer_shape=='01', object_width and object_height are used instead of texture_object_layer_width and texture_object_layer_height.

7.10.5.3 Searching a start position of the tile

The starting position of the target tile in the bitstream is found by using a look-up table before starting to decode. This way is applicable in tiling_jump_table_enable=1. In case of tiling_jump_table_enable = 0, texture_tile_start_code is used to find the start point of bitstream for each tile. For the look-up table generation, a size of each tile is stored as 'tile_size_low' and 'tile_size_high' in a header field. The decoding start position; *position[i]* for the *i*-th tile can be calculated by

© ISO/IEC 2004 – All rights reserved

$$position[i] = \sum_{j=0}^{i-1} tile\_size[j]$$

where the bitstream size of *i*-th tile is expressed by *tile_size[i]*. This mechanism enables to decode the wavelet coefficients of the tile at any place in the entire image.

7.10.6 Scalable binary shape object decoding

At first binary_arithmetic_decode() is called for decoding the base layer shape. For decoding the shape object of each layer in the enhancement layer, enh_binary_arithmetic_decode() is performed in the macroblock decoding manner. For decoding each shape block, SI_bab_type is decoded in first. SI_bab_type denotes the BAB type of SI(Scan Interleaving). And then decoding for BAB pixels based on CAE is performed as the BAB type of SI.

7.10.6.1 Decoding of base layer

It is identical as the video non-scalable shape coding. Refer to the shape decoding for base layer in the subclause 7.5 Shape Coding.

7.10.6.2 Decoding of enhancement layer

7.10.6.2.1 Block size control

The BAB size is controlled before decoding shape object of each enhancement layer. Decision of BAB size depends on the object size of each scalable layer. The resolutions of objects are divided into 3 classes. If the current coding layer is the *L*-th layer in the bitstream. the BAB size in each object resolution is obtained as follows:

```
layer_width = object_width >> (wavelet_decomposition_levels – L – 1);

layer_height = object_height >> (wavelet_decomposition_levels – L – 1);

if( layer_width > = 1024 || layer_height > = 1024)     bab_size = 64;

else if( layer_width > = 512 || layer_height > = 512)   bab_size = 32;

else                                  bab_size = 16;
```

7.10.6.2.2 Border formation

When decoding a BAB, pixels from neighbouring BABs can be used to make up the context. A 2 pixel wide border about the current BAB is used as depicted in Figure 7-67. In the figure, the size of the current BAB can be changed as described in subclause 7.10.6.2.1.

© ISO/IEC 2004 – All rights reserved

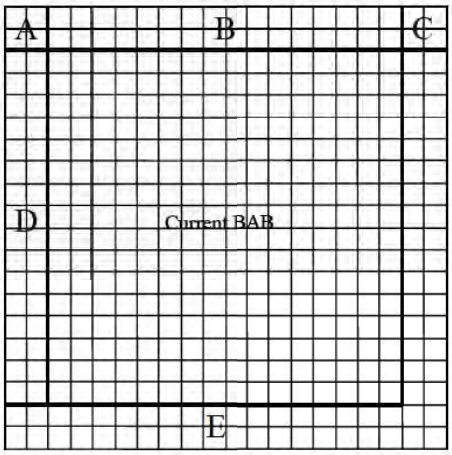


Figure 7-67 — Bordered BAB. A: TOP_LEFT_BORDER. B: TOP_BORDER. C: TOP_RIGHT_BORDER. D: LEFT_BORDER. E: BOTTOM_OR_RIGHT_BORDER

The border of the current BAB is partitioned into 5:

- TOP_LEFT_BORDER, which contains pixels from the BAB located to the upper-left of the current BAB and which consists of 2 lines of 2 pixels
- TOP_BORDER, which contains pixels from the BAB located above the current BAB and which consists of 2 lines of the number of bab_size pixels
- TOP_RIGHT_BORDER, which contains pixels from the BAB located to the upper-right of the current BAB and which consists of 2 lines of 2 pixels
- LEFT_BORDER, which contains pixels from the BAB located to the left of the current BAB and which consists of 2 columns of the number of bab_size pixels
- BOTTOM_OR_RIGHT_BORDER, which contains pixels from the collocated pixels in the lower layer and which consists of 2 lines in bottom and right of the current BAB. The pixel in (i,j) of the lower layer is repeatedly used for this border pixels in (2i,2j), (2i+1,2j), (2i,2j+1), and (2i+1,2j+1) of the current layer.

7.10.6.2.3 Decoding of BAB type of Scan Interleaving

For decoding a BAB of enhancement layer shape object, BAB type of SI is determined in advance. SI_bab_type denotes the two fields of BAB type of SI that are transitional BAB and exceptional BAB. Transitional BAB and exceptional BAB of SI are defined as follow.

transitional BAB: When the input BAB has no ESD and there exists no swapping pixel from wavelet decomposition, the input BAB is determined as transitional BAB. For decoding transitional BAB, only the TSD data of the BAB are decoded by BAC. All_0 and all_255 BAB are also included in this SI_bab_type. Since all_0 or all_255 BAB has no ESD or TSD, the BAB can be reconstructed by decoding only SI_bab_type.

exceptional BAB: When the input BAB has ESD or When there exist some swapping pixels from wavelet decomposition, the input BAB is determined as exceptional BAB. For decoding exceptional BAB, all of the pixels in the BAB are decoded by BAC.

The first decoded symbol of enh_binary_arithmetic_decode() is SI_bab_type. SI_bab_type is set to '0' for transitional BAB and is set to '1' for exceptional BAB. For decoding this flag by BAC, SI_bab_type_prob[] is used for the probability table of odd symmetry filter case, and sto_SI_bab_type_prob_even[] is used for the probability table of even symmetry filter case.

© ISO/IEC 2004 – All rights reserved

7.10.6.2.4 Scan order decision

If SI_bab_type is '0' (transitional BAB) the scan order should be determined before performing BAC. The decision method is similar to the method in the subclause 7.5.4.11 except that it uses right and bottom neighboring pixels instead of left and upper neighboring pixels. The bottom and right borders of the collocated block of the lower layer are extended with its neighboring pixels like Figure 7-68. If an extended border is outside VOP, the pixel values of the extended border are set to zero as in the subclause 7.5.4.11.

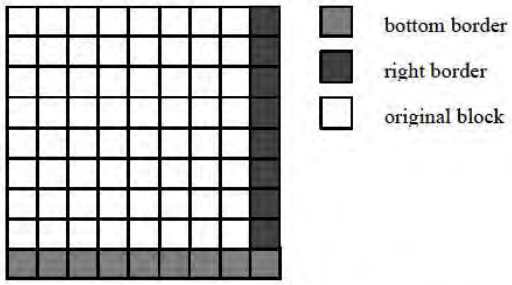


Figure 7-68 — Extension of border pixels

7.10.6.2.5 Decoding of transitional BAB

The decoding of transitional BAB is almost same with the scalable shape decoding of transitional BAB for P-VOP in the subclause 7.5.4 except that the decoding order of pixels inside the transitional BAB for still texture objects is different from the method in the subclause 7.5.4.8.

An example of decoding order of transitional BAB with size 16 x 16 (bab_size==16) is shown in Figure 7-69. In the figure, the "R"s indicate the pixels copied from collocated pixels in the reference layer. Apart from that, the actual decoding is based on the scan interleaving (SI) method as in the subclause 7.5.4.10. The binary arithmetic decoding and the usage of context values for arithmetic decoding are the same as those of the subclauses 7.5.4.13 and 7.5.4.14, respectively.

© ISO/IEC 2004 – All rights reserved

R	0	R	8	R	16	R	24	R	32	R	40	R	48	R	56
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
R	1	R	9	R	17	R	25	R	33	R	41	R	49	R	57
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
R	2	R	10	R	18	R	26	R	34	R	42	R	50	R	58
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
R	3	R	11	R	19	R	27	R	35	R	43	R	51	R	59
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
R	4	R	12	R	20	R	28	R	36	R	44	R	52	R	60
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
R	5	R	13	R	21	R	29	R	37	R	45	R	53	R	61
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
R	6	R	14	R	22	R	30	R	38	R	46	R	54	R	62
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
R	7	R	15	R	23	R	31	R	39	R	47	R	55	R	63
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191

Figure 7-69 — The decoding order of transitional BAB (bab_size==16)

7.10.6.2.6 Decoding of exceptional BAB

The exceptional BAB coding includes two steps. First the collocated $BAB_{1/2}$ (vertically enhanced exceptional BAB with size bab_size/2 x bab_size) in the half-higher resolution layer is coded. Then the exceptional BAB in the current layer is coded based on the $BAB_{1/2}$ at half-higher resolution layer. Figure 7-70 illustrates the decoding steps for exceptional BABs.



Figure 7-70 — Decoding steps for exceptional BABs

When decoding a vertically enhanced BAB, pixels from neighboring BABs in the half resolution layer can be used to make up the context. A 1 pixel wide border about the current $BAB_{1/2}$ is used as depicted in Figure 7-71.

© ISO/IEC 2004 – All rights reserved

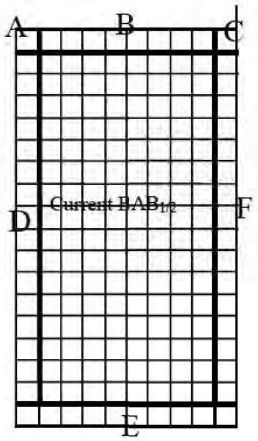


Figure 7-71 — Bordered half-higher resolution $BAB_{1/2}$.
A: TOP_LEFT_BORDER. B: TOP_BORDER. C: TOP_RIGHT_BORDER.
D: LEFT_BORDER. E: BOTTOM, F: RIGHT_BORDER

The border of the current $BAB_{1/2}$ is partitioned into 6 regions:

a) TOP_LEFT_BORDER, which contains pixels from the $BAB_{1/2}$ located to the upper-left of the current $BAB_{1/2}$ and which consists 1 pixel

b) TOP_BORDER, which contains pixels from the $BAB_{1/2}$ located above the current $BAB_{1/2}$ and which consists of 1 lines of the number of bab_size/2 pixels

c) TOP_RIGHT_BORDER, which contains pixels from the $BAB_{1/2}$ located to the upper-right of the current $BAB_{1/2}$ and which consists of 1 pixels

d) LEFT_BORDER, which contains pixels from the $BAB_{1/2}$ located to the left of the current $BAB_{1/2}$ and which consists of 1 columns of the number of bab_size pixels

e) BOTTOM_BORDER, which contains pixels from the collocated pixels in the lower layer and which consists of 1 lines in bottom of (bab_size/2+2) pixels. The pixels at (2i,j) in this border are filled by of the pixels at (i, j) in lower layer.

f) RIGHT_BORDER, which contains pixels from the collocated pixels in the lower layer and which consists of 1 column of the number of bab_size pixels. The pixel at (2i,j) in this border is filled by pixel at (i,j) from lower layer. And the pixel at (2i+1, j) in this border is filled by (i,j) from lower layer if the pixel at (i,j-1) in lower layer is not zero, otherwise it is filled by the logic OR of pixel at (i,j) and pixel at (i+1,j) in the lower layer.

By using 8-bits contexts, a pixel is arithmetic decoded as in CAE. The contexts for exceptional BAB decoding of shape object are described in Figure 7-72 (a), (b) and Figure 7-73 (a), (b). The contexts are calculated as below. And for each context, the probability table, sto_enh_odd_prob0[], sto_enh_even_prob0[], sto_enh_odd_prob1[], and sto_enh_even_prob1[] are used, respectively.

Also note that all the probability tables in the Appendix give the probability of zeros under given contexts. The probability tables are normalized to within range [0, 65536]. There are three special cases,

(1) Probability of zero under certain context is zero. This means that under that context, it is certain that the coded value and can only be one.

(2) Probability of zero under certain context is 65536. This means that under that context, it is certain that the coded value can only be zero.

(3) Probability of zero under certain context is 65537. This means that this specific context can never occur.

$Context_0 = (T_9<<7)|\ (T_8<<6)|\ (T_7<<5)|(T_6<<4)|(T_5<<3)|(T_4<<2)|(T_3<<1)|T_2$

© ISO/IEC 2004 – All rights reserved

$Context_1 = (T_1<<7) | (T_{10}<<6) | (T_7<<5) | (T_6<<4) | (T_5<<3) | (T_4<<2) | (T_3<<1) | T_2$

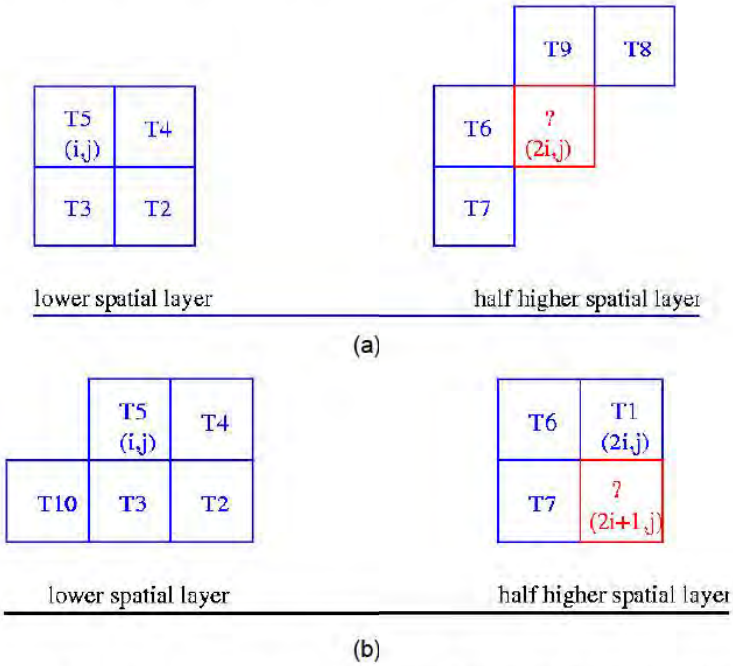


Figure 7-72 — The context used for vertically enhanced exceptional BABs

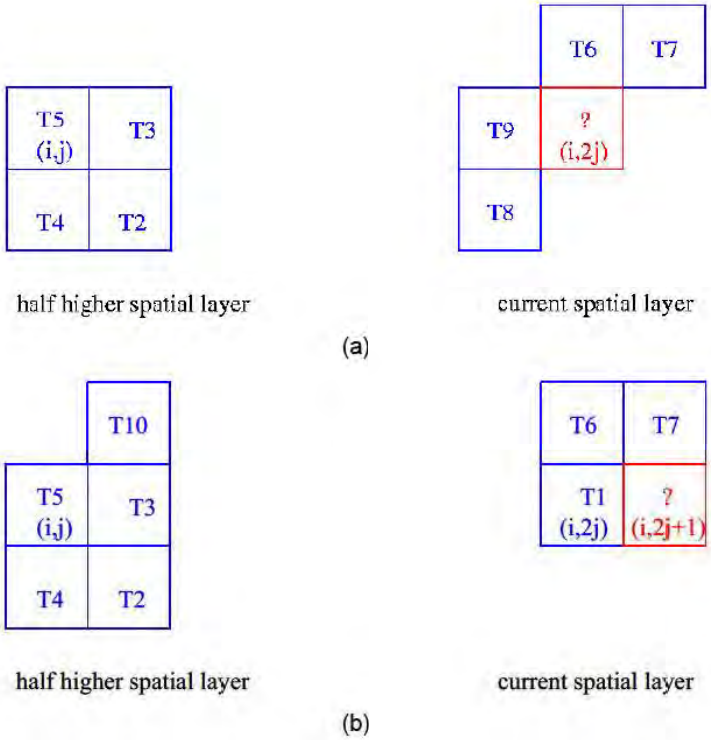


Figure 7-73 — The context used for enhanced exceptional BABs

© ISO/IEC 2004 – All rights reserved

7.11 Mesh object decoding

An overview of the decoding process is show in Figure 7-74.

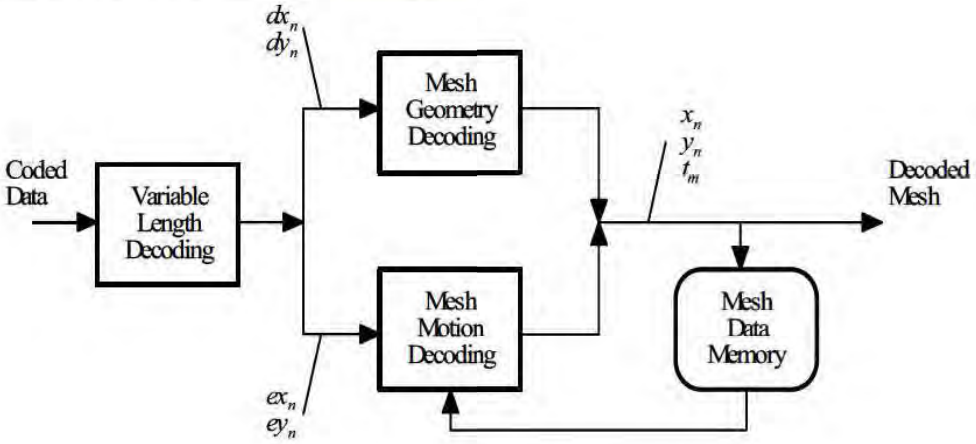


Figure 7-74 — Simplified 2D Mesh Object Decoding Process

Variable length decoding takes the coded data and decodes either node point location data or node point motion data. Node point location data is denoted by dx_n, dy_n and node point motion data is denoted by ex_n, ey_n, where n is the node point index (n = 0, ..., N-1). Next, either mesh geometry decoding or mesh motion decoding is applied. Mesh geometry decoding computes the node point locations from the location data and reconstructs a triangular mesh from the node point locations. Mesh motion decoding computes the node point motion vectors from the motion data and applies these motion vectors to the node points of the previous mesh to reconstruct the current mesh.

The reconstructed mesh is stored in the mesh data memory, so that it may be used by the motion decoding process for the next mesh. Mesh data consists of node point locations (x_n, y_n) and triangles t_m, where m is the triangle index (m = 0, ..., M-1) and each triangle t_m contains a triplet <i, j, k> which stores the indices of the node points that form the three vertices of that triangle.

A mesh object consists of a sequence of mesh object planes. The is_intra flag of the mesh object plane class determines whether the data that follows specifies the initial geometry of a new dynamic mesh, or that it specifies the motion of the previous mesh to the current mesh, in a sequence of meshes. Firstly, the decoding of mesh geometry is described; then, the decoding of mesh motion is described. In this part of ISO/IEC 14496, a pixel-based coordinate system is assumed, where the x-axis points to the right from the origin, and the y-axis points down from the origin.

7.11.1 Mesh geometry decoding

Since the initial 2D triangular mesh is either a uniform mesh or a Delaunay mesh, the mesh triangular structure (i.e. the connections between node points) is not coded explicitly. Only a few parameters are coded for the uniform mesh; only the 2D node point coordinates $\vec{p}_n = (x_n, y_n)$ are coded for the Delaunay mesh. In each case, the coded information defines the triangular structure of the mesh implicitly, such that it can be computed uniquely by the decoder. The mesh_type_code specifies whether the initial mesh is uniform or Delaunay.

7.11.1.1 Uniform mesh

A 2D uniform mesh subdivides a rectangular object plane area into a set of rectangles, where each rectangle in turn is subdivided into two triangles. Adjacent triangles share node points. The node points are spaced equidistant horizontally as well as vertically. An example of a uniform mesh is given in Figure 7-75.

© ISO/IEC 2004 – All rights reserved

Five parameters are used to specify a uniform mesh. The first two parameters, nr_mesh_nodes_hor and nr_mesh_nodes_vert, specify the number of node points of the mesh in the horizontal, resp. vertical direction. In the example of Figure 7-75, nr_mesh_nodes_hor is equal to 5 and nr_mesh_nodes_vert is equal to 4. The next two parameters, mesh_rect_size_hor and mesh_rect_size_vert, specify the horizontal, resp. vertical size of each rectangle in half pixel units. The meaning of these parameters is indicated in Figure 7-75. The last parameter, triangle_split_code, specifies how each rectangle is split to form two triangles. The four methods of splitting that are allowed are indicated in Figure 7-76. The top-left node point of a uniform mesh coincides with the origin of a local coordinate system.

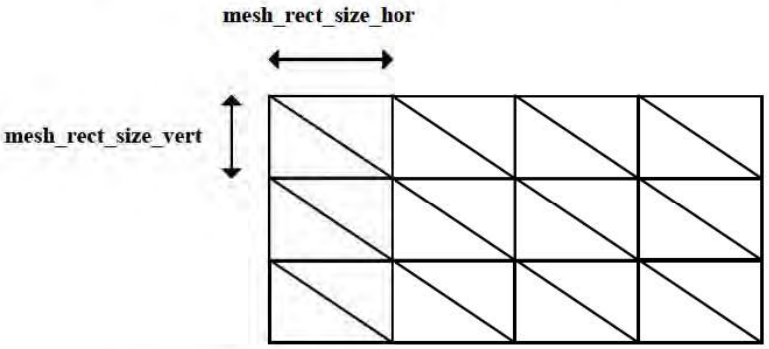


Figure 7-75 — Specification of a uniform 2D mesh

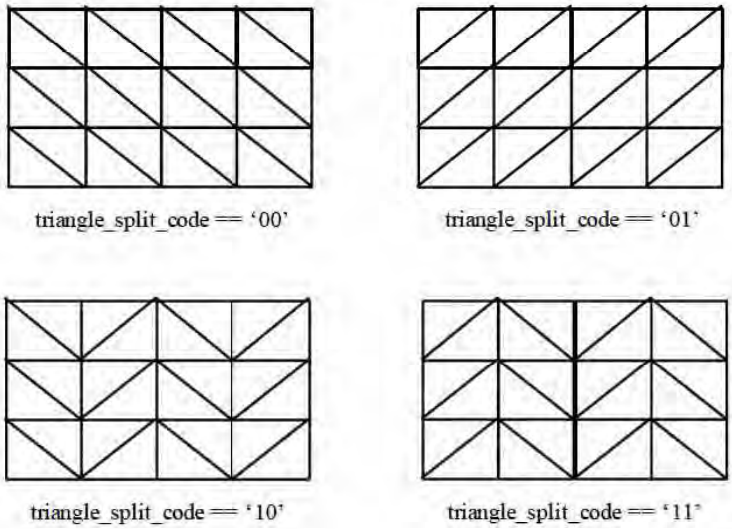


Figure 7-76 — Illustration of the types of uniform meshes defined

7.11.1.2 Delaunay mesh

First, the total number of node points in the mesh N is decoded; then, the number of node points that are on the boundary of the mesh N_b is decoded. Note that N is the sum of the number of nodes in the interior of the mesh, N_i and the number of nodes on the boundary, N_b,

$$N = N_i + N_b.$$

© ISO/IEC 2004 – All rights reserved

Now, the locations of boundary and interior node points are decoded, where we assume the origin of the local coordinate system is at the top left of the bounding rectangle surrounding the initial mesh. The x-, resp. y-coordinate of the first node point, $\vec{p}_0 = (x_0, y_0)$, is decoded directly, where x_0 and y_0 are specified w.r.t. to the origin of the local coordinate system. All the other node point coordinates are computed by adding a dx_n, resp. dy_n value to, resp. the x- and y-coordinate of the previously decoded node point. Thus, the coordinates of the initial node point $\vec{p}_0 = (x_0, y_0)$ is decoded as is, whereas the coordinates of all other node points, $\vec{p}_n = (x_n, y_n)$, n = 1, ..., N - 1, are obtained by adding a decoded value to the previously decoded node point coordinates:

$$x_n = x_{n-1} + dx_n \quad \text{and} \quad y_n = y_{n-1} + dy_n .$$

The ordering in the sequence of decoded locations is such that the first N_b locations correspond to boundary nodes. Thus, after receiving the first N_b locations, the decoder is able to reconstruct the boundary of the mesh by connecting each pair of successive boundary nodes, as well as the first and the last, by straight-line edge segments. The next N - N_b values in the sequence of decoded locations correspond to interior node points. Thus, after receiving N nodes, the locations of both the boundary and interior nodes can be reconstructed, in addition to the polygonal shape of the boundary. This is illustrated with an example in Figure 7-77.

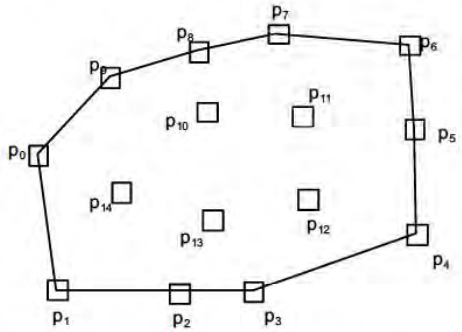


Figure 7-77 — Decoded node points and mesh boundary edge

The mesh is finally obtained by applying constrained Delaunay triangulation to the set of decoded node points, where the polygonal mesh boundary is used as a constraint. A constrained triangulation of a set of node points $\vec{p}_n$ contains the line segments between successive node points on the boundary as edges and contains triangles only in the interior of the region defined by the boundary. Each triangle $t_k = \langle \vec{p}_l, \vec{p}_m, \vec{p}_n \rangle$ of a constrained Delaunay triangulation furthermore satisfies the property that the circumcircle of t_k does not contain in its interior any node point $\vec{p}_r$ visible from all three vertices of t_k. A node point is visible from another node point if a straight line drawn between them falls entirely inside or exactly on the constraining polygonal boundary. The Delaunay triangulation process is defined as any algorithm that is equivalent to the following.

a. Determine any triangulation of the given node points such that all triangles are contained in the interior of the polygonal boundary. The triangulation shall contain 2 N_i + N_b – 2 triangles.

b. Inspect each interior edge, shared by two opposite triangles, of the triangulation and test if the edge is locally Delaunay. If there is an interior edge that is not locally Delaunay, the two opposite triangles <p_a, p_b, p_c> and <p_a, p_c, p_d> sharing this edge are replaced by triangles <p_a, p_b, p_d> and <p_b, p_c, p_d>. Continue until all interior edges of the triangulation are locally Delaunay.

An interior edge, shared by two opposite triangles <p_a, p_b, p_c> and <p_a, p_c, p_d>, is locally Delaunay if point p_d is outside the circumcircle of triangle <p_a, p_b, p_c>. If point p_d is inside the circumcircle of triangle <p_a, p_b, p_c>, then the edge is not locally Delaunay. If point p_d is exactly on the circumcircle of triangle <p_a, p_b, p_c>, then the edge between points p_a and p_c is deemed locally Delaunay only if point p_b or point p_d is the point (among these four points) with the maximum x-coordinate, or, in case there is more than one point with the same maximum x-coordinate, the point with the maximum y-coordinate among these points.An example of a mesh obtained by constrained triangulation of the node points of Figure 7-77 is shown in Figure 7-78.

© ISO/IEC 2004 – All rights reserved

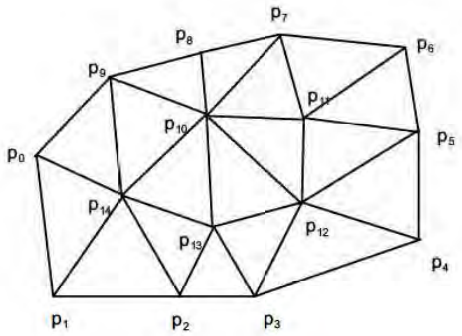


Figure 7-78 — Decoded triangular mesh obtained by constrained Delaunay triangulation

7.11.2 Decoding of mesh motion vectors

Each node point $\vec{p}_n$ of a 2D Mesh Object Plane numbered *k* in the sequence of Mesh Object Planes has a 2D motion vector $\vec{v}_n = (vx_n, vy_n)$, defined from Mesh Object Plane *k* to *k*+1. By decoding these motion vectors, one is able to reconstruct the locations of node points in Mesh Object Plane numbered *k*+1. The triangular topology of the mesh remains the same throughout the sequence. Node point motion vectors are decoded according to a predictive method, i.e., the components of each motion vector are predicted using the components of already decoded motion vectors of other node points.

7.11.2.1 Motion vector prediction

To decode the motion vector of a node point $\vec{p}_n$ that is part of a triangle $t_k = \langle \vec{p}_l, \vec{p}_m, \vec{p}_n \rangle$, where the two motion vectors vectors $\vec{v}_l$ and $\vec{v}_m$ of the nodes $\vec{p}_l$ and $\vec{p}_m$ have already been decoded, one can use the values of $\vec{v}_l$ and $\vec{v}_m$ to predict $\vec{v}_n$ and add the prediction vector to a decoded prediction error vector. Starting from an initial triangle t_k of which all three node motion vectors have been decoded, there must be at least one other, neighboring, triangle t_w that has two nodes in common with t_k. Since the motion vectors of the two nodes that t_k and t_w have in common have already been decoded, one can use these two motion vectors to predict the motion vector of the third node in t_w. The actual prediction vector $\vec{w}_n$ is computed by averaging of the two prediction motion vectors and the components of the prediction vector are rounded to half-pixel accuracy, as follows:

$$\vec{w}_n = 0.5 \bullet \left(\text{floor}(vx_m + vx_l + 0.5), \text{floor}(vy_m + vy_l + 0.5)\right),$$
$$\vec{v}_n = \vec{w}_n + \vec{e}_n$$

Here, $\vec{e}_n = (ex_n, ey_n)$ denotes the prediction error vector, the components of which are decoded from variable length codes. This procedure is repeated while traversing the triangles and nodes of the mesh, as explained below. While visiting all triangles of the mesh, the motion vector data of each node is decoded from the bitstream one by one. Note that no prediction is used to decode the first motion vector,

$$\vec{v}_{n_0} = \vec{e}_{n_0},$$

and that only the first decoded motion vector is used as a predictor to code the second motion vector,

$$\vec{v}_{n_1} = \vec{v}_{n_0} + \vec{e}_{n_1}.$$

Note further that the prediction error vector is specified only for node points with a nonzero motion vector. For all other node points, the motion vector is $\vec{v}_n = (0,0)$.

© ISO/IEC 2004 – All rights reserved

Finally, the horizontal and vertical components of mesh node motion vectors are processed to lie within a certain range, equivalent to the processing of video block motion vectors described in subclause 7.6.3.

7.11.2.2 Mesh traversal

We use a *breadth-first traversal* to order all the triangles and nodes in the mesh numbered *k*, and to decode the motion vectors defined from mesh *k* to *k*+1. The breadth-first traversal is determined uniquely by the topology and geometry of an intra-coded mesh. That is, the ordering of the triangles and nodes shall be computed on an intra-coded Mesh Object Plane and remains constant for the following predictive-coded Mesh Object Planes. The breadth-first traversal of the mesh triangles is defined as follows (see Figure 7-79 for an illustration).

First, define the *initial triangle* as follows. Define the top left mesh node as the node *n* with minimum $x_n + y_n$, assuming the origin of the local coordinate system is at the top left. If there is more than one node with the same value of $x_n + y_n$, then choose the node point among these with minimum *y*. The initial triangle is the triangle that contains the edge between the top-left node of the mesh and the next clockwise node on the boundary. Label the initial triangle with the number 0.

Next, all other triangles are iteratively labeled with numbers 1, 2, ..., *M* - 1, where *M* is the number of triangles in the mesh, as follows.

> Among all labeled triangles that have adjacent triangles which are not yet labeled, find the triangle with the lowest number label. This triangle is referred to in the following as the *current triangle*. Define the *base edge* of this triangle as the edge that connects this triangle to the already labeled neighboring triangle with the lowest number. In the case of the initial triangle, the base edge is defined as the edge between the top-left node and the next clockwise node on the boundary. Define the *right edge* of the current triangle as the next counterclockwise edge of the current triangle with respect to the base edge; and define the *left edge* as the next clockwise edge of the current triangle with respect to the base edge. That is, for a triangle $t_k = \langle \vec{p}_l, \vec{p}_m, \vec{p}_n \rangle$, where the vertices are in clockwise order, if $\langle \vec{p}_l \vec{p}_m \rangle$ is the base edge, then $\langle \vec{p}_l \vec{p}_n \rangle$ is the right edge and $\langle \vec{p}_m \vec{p}_n \rangle$ is the left edge.
>
> Now, check if there is an unlabeled triangle adjacent to the current triangle, sharing the right edge. If there is such a triangle, label it with the next available number. Then check if there is an unlabeled triangle adjacent to the current triangle, sharing the left edge. If there is such a triangle, label it with the next available number.

This process is continued iteratively until all triangles have been labeled with a unique number *m*.

The ordering of the triangles according to their assigned label numbers implicitly defines the order in which the motion vector data of each node point is decoded, as described in the following. Initially, motion vector data for the top-left node of the mesh is retrieved from the bitstream. No prediction is used for the motion vector of this node, hence this data specifies the motion vector itself. Then, motion vector data for the second node, which is the next clockwise node on the boundary w.r.t. the top-left node, is retrieved from the bitstream. This data contains the prediction error for the motion vector of this node, where the motion vector of the top-left node is used as a prediction. Mark these first two nodes (that form the base edge of the initial triangle) with the label 'done'.

Next, process each triangle as determined by the label numbers. For each triangle, the base edge is determined as defined above. The motion vectors of the two nodes of the base edge of a triangle are used to form a prediction for the motion vector of the third node of that triangle. If that third node is not yet labeled 'done', motion vector data is retrieved and used as prediction error values, i.e. the decoded values are added to the prediction to obtain the actual motion vector. Then, that third node is labeled 'done'. If the third note is already labeled 'done', then it is ignored and no data is retrieved. Note that due to the ordering of the triangles as defined above, the two nodes on the base edge of a triangle are guaranteed to be labeled 'done' when that triangle is processed, signifying that their motion vectors have already been decoded and may be used as predictors.

© ISO/IEC 2004 – All rights reserved

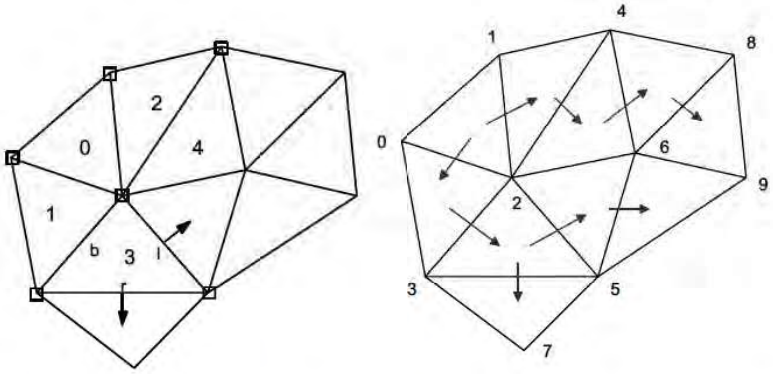


Figure 7-79 — Breadth-first traversal of a 2D triangular example mesh

In Figure 7-79 an example is shown of breadth-first traversal. On the left, the traversal is halfway through the mesh - five triangles have been labeled (with numbers) and the motion vectors of six node points have been decoded (marked with a box symbol). The triangle which has been labeled '3' is the 'current triangle'; the base edge is 'b'; the right and left edge are 'r' and 'l'. The triangles that will be labeled next are the triangles sharing the right, resp. left edge with the current triangle. After those triangles are labeled, the triangle which has been labeled '4' will be the next 'current triangle' and another motion vector will be decoded. On the right, the traversed 2D triangular mesh is shown, illustrating the transitions between triangles and final order of node points according to which respective motion vectors are decoded.

7.12 FBA object decoding

The frame rate, time code and skip frames information is independently associated with face, body or both depending on the fba_object_mask. For example, if the frame rate is set to 30Hz during a frame with fba_object_mask='01' (face) followed by a frame with frame rate set to 15Hz with fba_object_mask='10' (body) then the frame rate for the face remains set to 30Hz. If the fba_object_mask='11' the temporal information is applied to both face and body.

In frame based decoding, at the beginning of each predictive frame, the adaptive arithmetic coder shall be reset. At the end of each frame, it shall be flushed.

Each animated field has its own set of models. At each intra frame, if the stream has been declared in random access mode (see ISO/IEC 14496-1), the models shall be reset to the uniform statistics. If the stream is not in random access mode, the models shall not be reset unless the decoding structures (AnimQP) are modified.

For more details on the arithmetic decoding process, refer to ISO/IEC 14496-1, subclause 9.3.8.

7.12.1 Frame based face object decoding

This subclause specifies the additional decoding process required for face object decoding.

The coded data is decoded by an arithmetic decoding process. The arithmetic decoding process is described in detail in Annex B. Following the arithmetic decoding, the data is de-quantised by an inverse quantisation process. The FAPs are obtained by a predictive decoding scheme as shown in Figure 7-80.

The base quantisation step size QP for each FAP is listed in Table C-1. The quantiser scale fap_quant is applied uniformly to all FAPs. The magnitude of the quantisation scaling factor ranges from 1 to 8. The value of fap_quant == 0 has a special meaning, it is used to indicate lossless coding mode, so no Inverse quantisation is applied. The quantisation step size is obtained as follows:

```
if (fap_quant)
    qstep = QP * fap_quant
else
    qstep = 1
```


© ISO/IEC 2004 – All rights reserved

The inverse quantised FAP'(t) is obtained from the decoded coefficient FAP''(t) as follows:

FAP'(t) = qstep * FAP''(t)

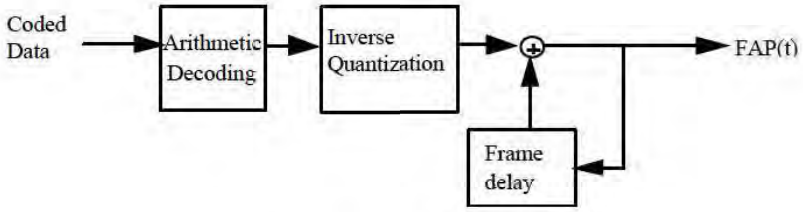


Figure 7-80 — FAP decoding

7.12.1.1 Decoding of faps

For a given frame FAPs in the decoder assume one of three of the following states:

- set by a value transmitted by the encoder
- retain a value previously sent by the encoder
- interpolated by the decoder

FAP values which have been initialized in an intra coded FAP set are assumed to retain those values if subsequently masked out unless a special mask mode is used to indicate interpolation by the decoder. FAP values which have never been initialized must be estimated by the decoder. For example, if only FAP group 2 (inner lip) is used and FAP group 8 (outer lip) is never used, the outer lip points must be estimated by the decoder. In a second example the FAP decoder is also expected to enforce symmetry when only the left or right portion of a symmetric FAP set is received (e.g. if the left eye is moved and the right eye is subject to interpolation, it is to be moved in the same way as the left eye).

7.12.2 DCT based face object decoding

The bitstream is decoded into segments of FAPs, where each segment is composed of a temporal sequence of 16 FAP object planes. The block diagram of the decoder is shown in Figure 7-81.

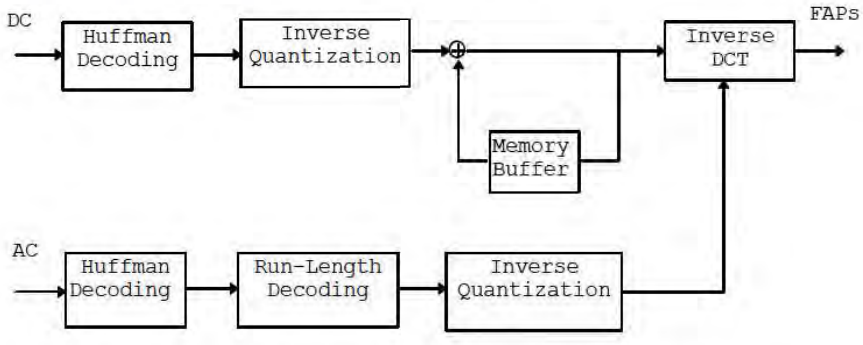


Figure 7-81 — Block diagram of the DCT-based decoding process

© ISO/IEC 2004 – All rights reserved

The DCT-based decoding process consists of the following three basic steps:

- Differential decoding the DC coefficient of a segment.
- Decoding the AC coefficients of the segment.
- Determining the 16 FAP values of the segment using inverse discrete cosine transform (IDCT).

A uniform quantisation step size is used for all AC coefficients. The quantisation step size for AC coefficients is obtained as follows:

```
qstep[i]  = fap_scale[fap_quant_inex] * DCTQP[i]
```

where DCTQP[i] is the base quantisation step size and its value is defined in subclause 6.3.10.10. The quantisation step size of the DC coefficient is one-third of the AC coefficients. Different quantisation step sizes are used for different FAPs.

The DCT-based decoding process is applied to all FAP segments except the viseme (FAP #1) and expression (FAP #2) parameters. The latter two parameters are differential decoded without transform. The decoding of viseme and expression segments are described at the end of this subclause.

For FAP #3 to FAP #68, the DC coefficient of an intra coded segment is stored as a 16-bit signed integer if its value is within the 16-bit range. Otherwise, it is stored as a 31-bit signed integer. For an inter coded segment, the DC coefficient of the previous segment is used as a prediction of the current DC coefficient. The prediction error is decoded using a Huffman table of 512 symbols. An "ESC" symbol, if obtained, indicates that the prediction error is out of the range [-255, 255]. In this case, the next 16 bits extracted from the bitstream are represented as a signed 16-bit integer for the prediction error. If the value of the integer is equal to -256*128, it means that the value of the prediction error is over the 16-bit range. Then the following 32 bits from the bitstream are extracted as a signed 32-bit integer, in twos complement format and the most significant bit first.

The AC coefficients, for both inter and intra coded segments, are decoded using Huffman tables. The run-length code indicates the number of leading zeros before each non-zero AC coefficient. The run-length ranges from 0 to 14 and proceeds the code for the AC coefficient. The symbol 15 in the run length table indicates the end of non-zero symbols in a segment. Therefore, the Huffman table of the run-length codes contains 16 symbols. The values of non-zero AC coefficients are decoded in a way similar to the decoding of DC prediction errors but with a different Huffman table.

The bitstreams corresponding to viseme and expression segments are differential decoded without IDCT. For an intra coded segment, the quantised values of the first viseme_select1, viseme_select2, viseme_blend, expression_select1, expression_select2, expression_intensity1, and expression_intensity2 within the segment are decoded using fixed length code. These first values are used as the prediction for the second viseme_select1, viseme_select2, … etc of the segment and the prediction error are differential decoded using Huffman tables. For an inter coded segment, the last viseme_select1, for example, of the previous decoded segment is used to predict the first viseme_select1 of the current segment. The decoded values (before inverse quantisation) of differential coded viseme and expression parameter fields are obtained.

```
byviseme_segment_select1q[k] = viseme_segment_select1q[k-1] +
    viseme_segment_select1q_diff[k] - 14
viseme_segment_select2q[k]       = viseme_segment_select2q[k-1] +
    viseme_segment_select2q_diff[k] - 14
viseme_segment_blendq[k]     = viseme_segment_blendq[k-1] +
    viseme_segment_blendq_diff[k] - 63
expression_segment_select1q[k]   = expression_segment_select1q[k-1] +
    expression_segment_select1q_diff[k] - 6
expression_segment_select2q[k]   = expression_segment_select2q[k-1] +
    expression_segment_select2q_diff[k] - 6
expression_segment_intensity1q[k] = expression_segment_intensity1q[k-1] +
    expression_segment_intensity1q_diff[k] - 63
expression_segment_intensity2q[k] = expression_segment_intensity2q[k-1] +
    expression_segment_intensity2q_diff[k] - 63
```


© ISO/IEC 2004 – All rights reserved

7.12.3 Decoding of the viseme parameter fap 1

Fourteen visemes have been defined for selection by the Viseme Parameter FAP 1, the definition is given in Annex C. The viseme parameter allows two visemes from a standard set to be blended together. The viseme parameter is composed of a set of values as follows.

Table 7-21 — Viseme parameter range

viseme () {	Range
viseme_select1	0-14
viseme_select2	0-14
viseme_blend	0-63
viseme_def	0-1
}	

Viseme_blend is quantised (step size = 1) and defines the blending of viseme1 and viseme2 in the decoder by the following symbolic expression where viseme1 and 2 are graphical interpretations of the given visemes as suggested in the non-normative annex.

final viseme = (viseme 1) * (viseme_blend / 63) + (viseme 2) * (1 - viseme_blend / 63)

The viseme can only have impact on FAPs that are currently allowed to be interpolated.

If the viseme_def bit is set, the current mouth FAPs can be used by the decoder to define the selected viseme in terms of a table of FAPs. This FAP table can be used when the same viseme is invoked again later for FAPs which must be interpolated.

7.12.4 Decoding of the viseme parameter fap 2

The expression parameter allows two expressions from a standard set to be blended together.The expression parameter is composed of a set of values as follows.

Table 7-22 — Expression parameter range

expression () {	Range
expression_select1	0-6
expression_intensity1	0-63
expression_select2	0-6
expression_intensity2	0-63
init_face	0-1
expression_def	0-1
}	

Expression_intensity1 and expression_intensity2 are quantised (step size = 1) and define excitation of expressions 1 and 2 in the decoder by the following equations where expressions 1 and 2 are graphical interpretations of the given expression as suggested by the non-normative reference:

final expression = expression1 * (expression_intensity1 / 63)+ expression2 * (expression_intensity2 / 63)

The decoder displays the expressions according to the above fomula as a superposition of the 2 expressions.

© ISO/IEC 2004 – All rights reserved

The expression can only have impact on FAPs that are currently allowed to be interpolated. If the init_face bit is set, the neutral face may be modified within the neutral face constraints of mouth closure, eye opening, gaze direction, and head orientation before FAPs 3-68 are applied. If the expression_def bit is set, the current FAPs can be used to define the selected expression in terms of a table of FAPs. This FAP table can then be used when the same expression is invoked again later.

7.12.5 Fap masking

The face is animated by sending a stream of facial animation parameters. FAP masking, as indicated in the bitstream, is used to select FAPs. FAPs are selected by using a two level mask hierarchy. The first level contains two bit code for each group indicating the following options:

- no FAPs are sent in the group
- a mask is sent indicating which FAPs in the group are sent. FAPs not selected by the group mask retain their previous value if any previously set value (not interpolated by decoder if previously set)
- a mask is sent indicating which FAPs in the group are sent. FAPs not selected by the group mask retain must be interpolated by the decoder
- all FAPs in the group are sent.

7.12.6 Frame Based Body Decoding

This clause specifies the additional decoding process required for body model decoding.

The BAPs are quantised and coded by a predictive coding scheme, similar to the FAPs. For each parameter to be coded in the current frame, the decoded value of this parameter in the previous frame is used as the prediction. Then the prediction error, i.e., the difference between the current parameter and its prediction, is computed and coded by arithmetic coding. This predictive coding scheme prevents the coding error from accumulating. The arithmetic decoding process is described in detail in subclause B.2.3.

Similar to FAPs, each BAP has a different precision requirement. Therefore different quantisation step sizes are applied to the BAPs. The base quantisation step size for each BAP is defined in the tables below. The bit rate is controlled by adjusting the quantisation step via the use of a quantisation index parameter called **bap_pred_quant_index** is applied uniformly to all BAPs. The magnitude of the quantisation index parameter ranges from 0 to 31 and is an index to a bap_pred_scale_table as described in subclause 6.3.10.1 to obtain the BAP_QUANT parameter.

The base quantisation step size BQP for each BAP is listed in Annex C. The quantiser scale BAP_QUANT is applied uniformly to all BAPs. The magnitude of the quantisation scaling factor ranges from 1 to 31. The value of BAP_QUANT == 0 has a special meaning, it is used to indicate lossless coding mode, so no inverse quantisation is applied. The quantisation step size is obtained as follows:

```
BAP_QUANT = bap_pred_scale[bap_pred_quant_index]
If (BAP_QUANT)
    qstep = BQP * BAP_QUANT
Else
    qstep = 1
```

The inverse quantised BAP'(t) is obtained from the decoded coefficient BAP''(t) as follows:

BAP'(t) = qstep * BAP''(t)

© ISO/IEC 2004 – All rights reserved

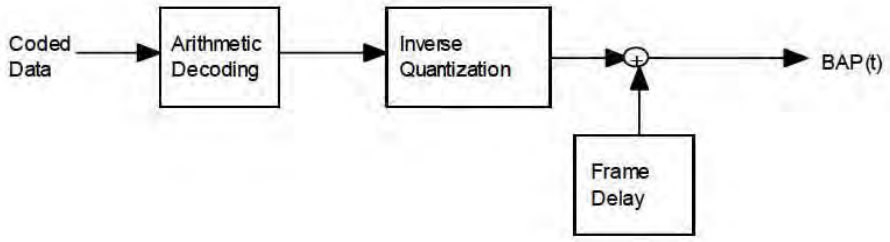


Figure 7-82 — BAP predictive coding

7.12.6.1 Decoding of BAPs

For a given frame, BAPs in the decoder assume one of the three following states:

1. set by a value transmitted by the encoder

2. retain a value previously sent by the encoder

BAP values which have been initialized in an intra coded BAP set are assumed to retain those values if subsequently masked out.

7.12.7 DCT based body object decoding

The bitstream is decoded into segments of BAPs, where each segment is composed of a temporal sequence of 16 BAP object planes. The block diagram of the decoder is shown in Figure 7-83.

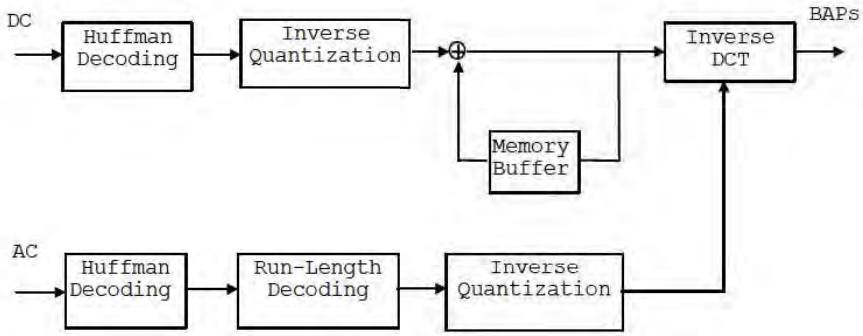


Figure 7-83 — Block diagram of the DCT-based BAPs decoding process

The DCT-based decoding process consists of the following three basic steps:

1. Differential decoding the DC coefficient of a segment.

2. Decoding the AC coefficients of the segment

3. Determining the 16 BAP values of the segment using inverse discrete cosine transform (IDCT).

© ISO/IEC 2004 – All rights reserved

A uniform quantisation step size is used for all AC coefficients. The quantisation step size for AC coefficients is obtained as follows:

qstep[i] = bap_scale[bap_quant_inex] * BQP[i]

where BQP[i] is the base quantisation step size and its value is defined in Subclause 6.3.10.8. The quantisation step size of the DC coefficient is one-third of the AC coefficients.

The DCT-based decoding process is applied to all BAPs. The DC coefficient of an intra coded segment is stored as a 16-bit signed integer if its value is within the 16-bit range. Otherwise, it is stored as a 31-bit signed integer. For an inter-coded segment, the DC coefficient of the previous segment is used as a prediction of the current DC coefficient. The prediction error is decoded using a Huffman table of 512 symbols. An "ESC" symbol, if obtained, indicates that the prediction error is out of the range [-255, 255]. In this case, the next 16 bits extracted from the bitstream are represented as a signed 16-bit integer for the prediction error. If the value of the integer is equal to -256*128, it means that the value of the prediction error is over the 16-bit range. Then the following 32 bits from the bitstream are extracted as a signed 32-bit integer, in twos complement format and the most significant bit first.

The AC coefficients, for both inter and intra coded segments, are decoded using Huffman tables. The run-length code indicates the number of leading zeros before each non-zero AC coefficient. The run-length ranges from 0 to 14 and proceeds the code for the AC coefficient. The symbol 15 in the run length table indicates the end of non-zero symbols in a segment. Therefore, the Huffman table of the run-length codes contains 16 symbols. The values of non-zero AC coefficients are decoded in a way similar to the decoding of DC prediction errors but with a different Huffman table.

7.13 3D Mesh Object Decoding

The Topological Surgery decoder is composed of four main modules, as shown in Figure 7-84, namely:

An arithmetic decoder, which reads a section of the input stream and outputs a bit stream.

A vertex graph decoder, which reads a section of the bit stream and outputs a bounding loop look-up table. However, when has_stitches flag is on, stitches are decoded and a vertex clustering array is built.

A triangle tree decoder, which reads a section of the bit stream and outputs the length of each run and the size of each sub-tree of the triangle tree.

A triangle data decoder, which reads a section of the input bit and outputs a stream of triangle data. This stream of triangle data contains the geometry and the properties associated with each triangle.

When a 3D mesh bitstream is received, the control information including stitches and partition types, as well as the header information including properties are obtained through the demultiplexer. The remaining bitstream is fed into the arithmetic decoder.

© ISO/IEC 2004 – All rights reserved

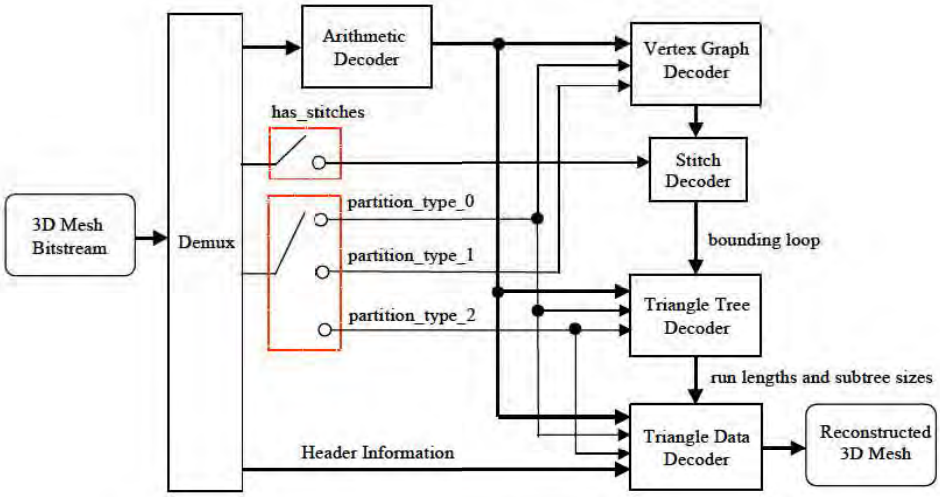


Figure 7-84 — Block diagram of the Topological Surgery decoder

In the hierarchical mode, the Topological Surgery decoder is used to decode the first (coarsest) level of detail mesh (base mesh), and the Forest Split decoder is used to recursively refine each level of detail to produce a new, higher resolution, level of detail. The Forest Split decoder is composed of four blocks, as shown in Figure 7-85, namely:

An arithmetic decoder, which reads a section of the input stream and outputs a bit stream.

A forest decoder, which reads a section of the bit stream and determines which edges of the current level mesh are part of the forest of edges.

A simple polygon decoder, which reads a section of the bit stream and outputs a sequence of simple polygons, including not only connectivity information, but also vertex coordinates, and property values.

A forest split operator, which takes the output data produced by the forest decoder and the simple polygon decoder, and applies the corresponding forest split operation, producing a new level of detail mesh.

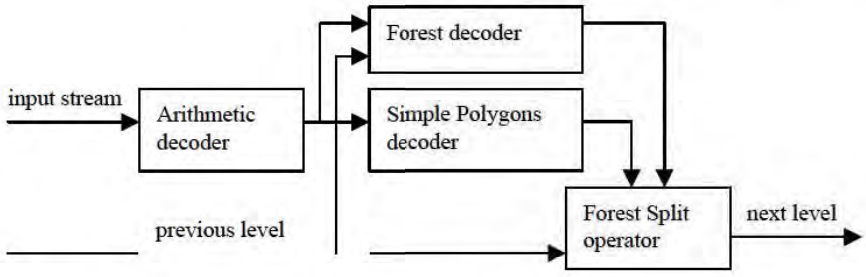


Figure 7-85 — Block diagram of the Forest Split decoder

7.13.1 Start codes and bit stuffing

A start code is a two-byte code of the form '0000 0000 00xx xxxx'. Several such codes are inserted into the bitstream for synchronisation purposes. To prevent any wrongful synchronisation, such codes shall not be emulated by the data. This is guaranteed by the insertion of a stuffing bit '1' after each byte-aligned sequence of eight 0's. The decoder shall skip these stuffing bits when parsing the bit stream. Note that the arithmetic coder is designed such as to never generate a synchronisation code. Therefore the bit skipping rule does not need be applied to the portions of the bit stream that contain arithmetic coded data.

© ISO/IEC 2004 – All rights reserved

7.13.2 The Topological Surgery decoding process.

The connectivity of a 3D mesh is represented as a Simple Polygon (triangulated with a single boundary loop) with zero or more pairs of boundary vertices identified, and with zero or more internal edges labeled as polygon edges. When the pairs of corresponding boundary edges are identified, the edges of the resulting reconstructed mesh corresponding to boundary edges of the Simple Polygon form the Vertex Graph. The correspondence among pairs of Simple Polygon boundary edges is recovered by the decoding process from the structure of Vertex Graph, producing the Vertex Loop look-up table. The Vertex Graph is represented as (i) a rooted spanning tree, (ii) the Vertex Tree, and (iii) zero or more jump edges. The Simple Polygon is represented as (i) a rooted spanning tree, (ii) the Triangle Tree, which defines an order of traversal of the triangles, (iii) a sequence of marching patterns, and (iv) a sequence of polygon_edges. The marching patterns are used to reconstruct the triangles by marching on the left or on the right along the polygon bounding loop, starting from an initial edge called the root edge. The polygon edges are used to join or not to join triangles sharing a marching edge to form the faces of the mesh. As the triangles of the simple polygon are visited in the order of traversal of the Triangle Tree, simple polygon boundary edges are put in correspondence using the information contained in the Vertex Loop look-up table, and so, reconstructing the original connectivity.

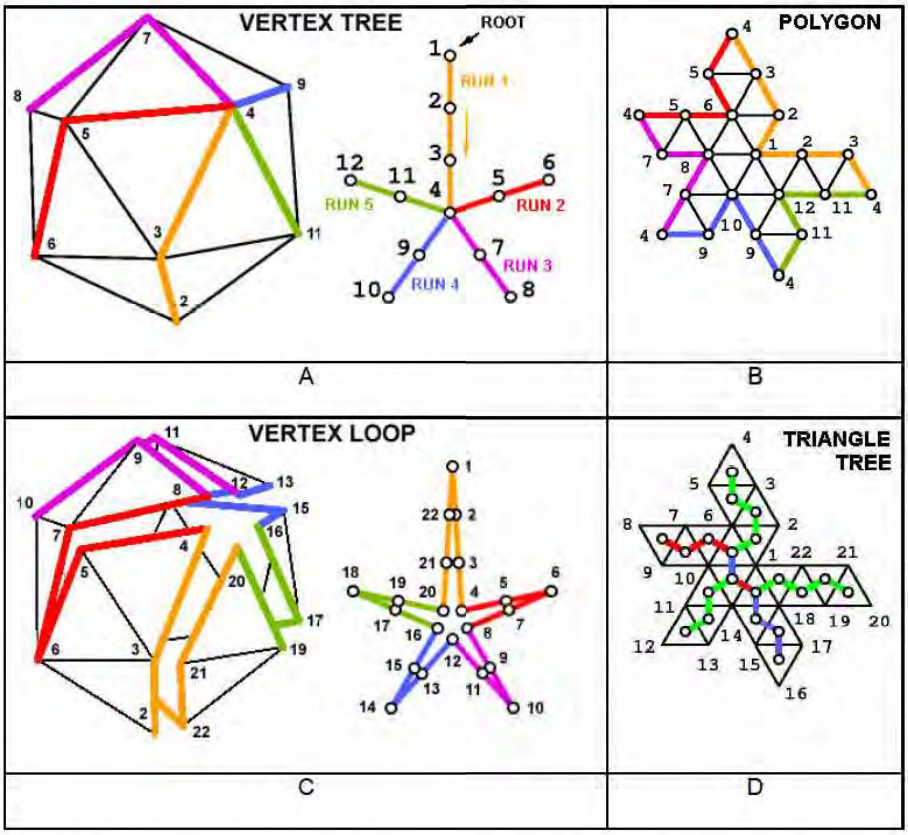


Figure 7-86 — Topological Surgery representation of a simple mesh.
A : for a simple mesh, the vertex graph is a tree.
B : when a simple mesh is cut through the vertex tree, the result is a simple polygon.
C : each edge of the vertex tree corresponds to exactly two boundary edges of the simple polygon.
D : the triangle tree spanning the dual graph of the simple polygon. In the general case, the face forest is first constructed spanning the dual graph of the 3D mesh. The vertex graph is composed of the remaining edges of the 3D mesh. If the 3D mesh has polygonal faces, they are subsequently triangulated by inserting virtual marching edges, converting the face forest into a triangle forest with one triangle tree per connected component.

© ISO/IEC 2004 – All rights reserved

The geometry and property data are quantised and predicted as a function of ancestors in the order of traversal of the triangles. The corresponding prediction errors are transmitted in the same order of traversal, interlaced with the corresponding marching and polygon_edges, so that, as soon as all these triangle data are received, the decoder can output the corresponding triangle.

A non-manifold 3D mesh is represented as a manifold 3D mesh and a Vertex Clustering array. The Vertex Clustering array is represented in the bitstream as a sequence of stitching commands (or Stitches) with one command per vertex. The decoding process decodes the Stitches after the vertex graph and constructs the Vertex Clustering array using the Stitches. Figure 7-87 shows a 3D mesh with a singular vertex and a singular edge, with the polygons incident to the singular vertex highlighted.

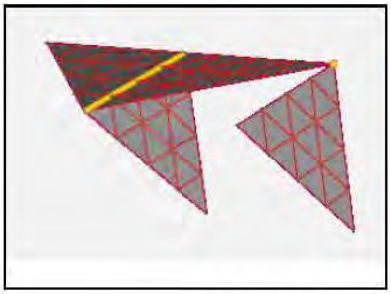


Figure 7-87 — 3D meshes with singular vertices and singular edges (each one with three incident faces) and singular edges (yellow)

At a high level, the Topological Surgery decoder performs the following steps:

```
Decode header information
for each partition {
 if (partition_type is 0) {
    for each connected component {
       decode vertex graph and construct bounding loop table
       if (has_stitches)
          decode stitches and build vertex clustering array
       decode triangle tree and construct table of triangle tree run lengths
       for each triangle in simple polygon {
          decode marching field and polygon field, and reconstruct connectivity
of triangle
          decode and reconstruct vertex coordinates and properties
       }
    }
 }
 else if (partition_type is 1) {
    for each vertex graph {
       decode vertex graph and construct bounding loop table
       if (has_stitches)
          decode stitches and build vertex clustering array
    }
 }
 else if (partition_type is 2) {
    decode partition header information
    decode triangle tree and construct table of triangle tree run lengths
    for each triangle in simple polygon {
       decode marching field, orientation field, and polygon field, and
reconstruct connectivity of triangle
       decode and reconstruct vertex coordinates and properties
    }
 }
}
```

© ISO/IEC 2004 – All rights reserved

7.13.3 The Forest Split decoding process

The hierarchical mode representation is based on the Progressive Forest split (PFS) scheme. In this scheme a 3D mesh is represented as a low resolution base mesh followed by a sequence of forest split operations. A forest split operation is represented by a forest in the graph of vertices and edges of a mesh, a sequence of simple polygons, and a sequence of vertex coordinates and property updates. The mesh is refined by cutting the mesh through the forest, splitting the resulting boundaries apart, filling each of the resulting tree boundary loops with one of the simple polygons, and finally, displacing the new vertices and properties. An optional pre-smoothing step can be used as a global predictor for the vertex coordinates, after the connectivity and properties of the new faces bound per face and per corner are updated. Also, and optional post-smoothing step can be used to reduce the quantisation artifacts after all the updates have been applied.

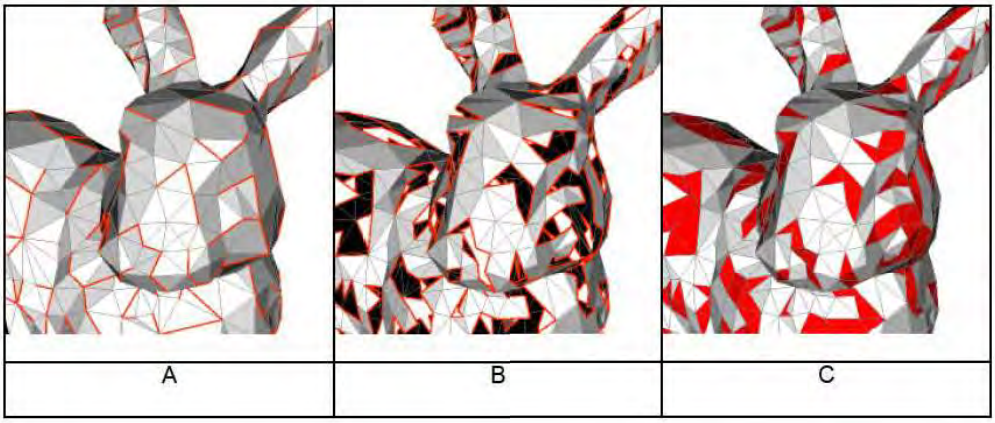


Figure 7-88 — The Forest Split operation. A 3D mesh with a forest of edges marked (A). The tree loops resulting from cutting the 3D mesh through the forest edges (B). Each tree loop is filled by a simple polygon composed of polygonal faces (C).

At a high level, the Forest Split decoder performs the following steps:

```
Decode pre and post-smoothing parameters.
Decode Forest.
For each tree in the forest {
    Cut mesh through the tree edges, and enumerate mesh components.
    Decode Triangle Tree.
    Decode per triangle connectivity data and property updates
        for properties bound per face and per corner.
    Update connectivity by triangulating the tree loop
        according to the decoded simple polygon.
}
Optionally, apply pre-smoothing step to vertex coordinates as a global predictor.
For each tree in the forest {
    Decode property updates for all the tree loop vertices, faces, and corners
}
```


© ISO/IEC 2004 – All rights reserved

Optionally, decode vertex coordinates, and property updates for all the vertices, faces, and corners not incident to any triangle loop.

Optionally, apply post-smoothing step to vertex coordinates.

7.13.4 Header decoder

The header information is divided into two parts. The first part contains information about the high level shading properties of the model, and the second defines the properties that are bound to the mesh.

7.13.4.1 High level shading properties

These are the **ccw**, **solid**, **convex**, **creaseAngle** fields defined in an IndexedFaceSet node. The crease angle is quantised to a 6-bit value and is reconstructed as 2*pi***creaseAngle**/63.

7.13.4.2 Property bindings and quantiser scales

There are four kinds of properties: vertex coordinates (coord), normals (normal), colors (color) and texture coordinates (texCoord). Properties can be bound to the mesh in four different ways: no binding, per vertex, per face and per corner. Not all combinations of bindings are valid. Table 7-23 lists the valid combinations. The binding of each property is obtained from coord_binding, normal_binding, color_binding and texCoord_binding, respectively.

Table 7-23 — List of valid combinations of properties and bindings

	no binding	**per vertex**	**per face**	**per corner**
coord	forbidden	valid	forbidden	forbidden
normal	valid	valid	valid	valid
color	valid	valid	valid	valid
texCoord	valid	valid	forbidden	valid

For each property for which there is a binding, i.e. the binding field does not contain the no binding value, the following information is further decoded: a bounding box, a quantisation step, a prediction mode, a list of coefficients used for linear prediction. The decoding process for the vertex coordinates is further described below. The same decoding process applies to the other properties.

The presence of a bounding box is given by the field **coord_bbox**. The bounding box is a cube represented by the position of its lower left corner and the length of its side. For geometry these parameters are given by **coord_xmin**, **coord_ymin**, **coord_zmin** and **coord_size**. If no bounding box is coded, as default bounding box is assumed. The default bounding box has its lower left corner at the origin, and a unit size.

The next field **coord_quant** indicates the number of bits to which each coordinate is quantised to. The **coord_pred_type** field indicates the prediction mode. It should always be equal to '10'.

The **coord_nlambda** field indicates the number of coefficients used for linear prediction. It can take one of three values, namely '01', '10', and '11'. **coord_nlambda**-1 number of values are then read from the **coord_lambda** field. The last coord_lambda value is computed so that the sum of all the coord_lmabda value is equal to one. This field indicates the weight given to an ancestor for prediction. The floating point value of a coefficient is given by the decoded signed integer divided by 2 to the power **coord_quant**. The last coefficient is never transmitted and is defined to be 1 minus the sum of all other coefficients.

The are some restrictions for the normal property: the **normal_bbox** field is always false, and the **normal_quant** value must always an odd number.

© ISO/IEC 2004 – All rights reserved

7.13.5 partition type

There are three partition types to convey vertext graph(vg), triangle tree(tt), and triangle data(td), The partition type specifies admissible combinations of these three pieces of data in the bitstream. See Figure 7-89.

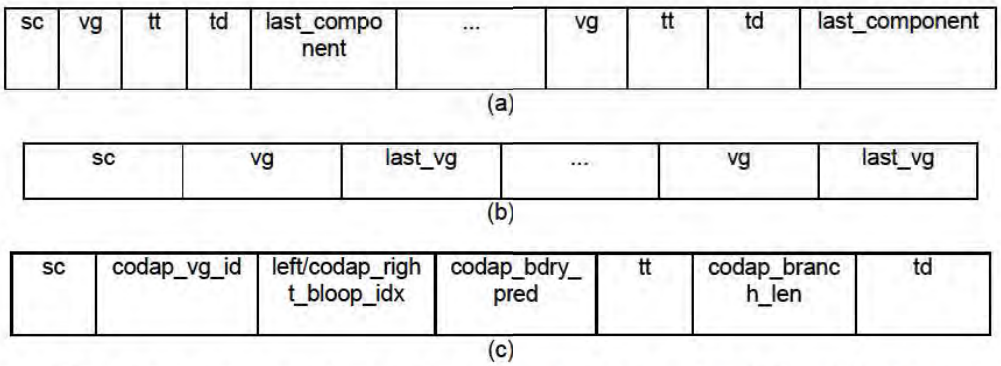


Figure 7-89 — Types of data partition : (a) one or more group of vg, tt and td included in a partition. (b) one or more vgs included in a partition. (c) the partition has one pair of tt and td.

7.13.5.1 connected component partition (partition_type_0)

This partition indicates the bitstream with one or more sequence of connected component consisting of vertex graph, triangle tree, and triangle data. The end of the connected component sequence is determined by the **last_component** field. The following partition may be either type 0 or 1.

7.13.5.2 vertex graph partition (partition_type_1)

This partition indicates the bitstream with one or more sequence of vertex graph. The end of the vertex graph sequence is determined by the **last_vg** field. The partition following the vertex graph partition shall be tt/td pair partition.

7.13.5.3 tt/td pair partition (partition_type_2)

This partition indicates the bitstream with one pair of triangle tree and triangle data. The tt/td pair partition is characterized by the vertex graph id, visiting indices, and boundary prediction mode. These variables are given fields **codap_vg_id, left/codap_right_bloop_idx**, and **codap_bdry_pred. codap_vg_id** field indicates a vertex graph corresponding to the tt/td pair. It is used to get the bounding loop information from the vertex graph. **codap_left_bloop_idx** and **codap_right_bloop_idx** fields indicate the left and right starting points of the bounding loop. They are also used to decide if a vertex in the partition is already visited in the previous partition. **codap_bdry_pred** field indicates if the vertices on the boundary of the partition should be decoded when the vertex is shared with previous partitions. If the partition ends at a branch, **codap_branch_len** field is added to the bitstream. Any type of partition may follow the tt/td pair partition.

7.13.5.3.1 Restricted boundary prediction mode (codap_bdry_pred=0)

The restricted boundary prediction mode does not duplicate vertices between partitions. Since the vertices predicted in the previous partitions may not be available, prediction is done only with the available vertices which are predicted in the current partition. If the previous partitions are lost, the triangles at the boundaries may not be reconstructed due to the fact that the vertices from the previous partitions are not available. However, the vertices predicted in the current partition may be reconstructed and decoding can continue.

The following process is applied for the prediction with a subset of all ancestors. Let the current method for prediction with 3 ancestors (a,b,c) be $d' = f(a,b,c)$. Then the equation is $d = d' + e$ and the value e is encoded. However, when the ancestors are visited in the previous partitions, the prediction method is as follows.

© ISO/IEC 2004 – All rights reserved

if (all ancestors (a, b, and c) are not available) then $d' = 0$

else if (only one ancestor, say t, is available) then $d' = t$

else if (two ancestors, say t_1 and t_2, are available) then

if (both ancestors are edge distance 1 from current vertex) then $d' = (t_1 + t_2)/2$

else if (t_1 is edge distance 1 from current vertex) then $d' = t_1$

else $d' = t_2$

else $d' = f(a,b,c)$.

7.13.5.3.2 Extended boundary prediction mode (codap_bdry_pred=1)

When this mode is selected, the decoder assumes that every vertex is not visited in the previous partition. Thus, after the three vertices of the root triangle are predicted, the vertices in the rest of the triangles in the partition will have all three ancestors for prediction.

7.13.6 Vertex Graph Decoder

The vertex graph decoder reconstructs a sequence of vertex runs. It then constructs the Vertex Loop look-up table.

For meshes that have simple topology and that do not have a boundary, the vertex graph does not contain any loops and is hence a tree. The field **vg_simple,** if set, indicates that the graph contains no loop. Some fields below are skipped when **vg_simple** is set.

Each vertex run is characterized by several variables, namely length, last, forward, loop and loop index. These variables are given by the fields **vg_run_length**, **vg_last**, **vg_forward_run**, **vg_loop** and **loop_index**. **vg_forward_run**, **vg_loop** and **loop_index** contain information relative to loops. When **vg_simple** is true, these three fields are not coded.

The process to reconstruct the vertex loop uses an auxiliary loop queue variable, initialy empty. When **vg_loop** is set the current run is put into the loop queue. When **vg_forward_run** is not set a run is pulled out from the queue. **loop_index** determines the position of the pulled out run in the queue. **vg_forward_run** is not coded when the loop queue is empty. **loop_index** is coded by its unary representation. When the index is equal to the number of elements in the queue minus 1, the trailing bit of its unary representation is not coded. Therefore when there is a single element in the queue, **loop_index** is not coded.

The end of the graph is determined by the **vg_last** and **vg_leaf** fields. An auxiliary depth variable is used for this purpose. It is decremented each time **vg_last** is set and incremented each time **vg_leaf** is not set. The depth is initialised to zero at the beginning of the graph. The end of the graph is reached when the depth variable becomes negative.

© ISO/IEC 2004 – All rights reserved

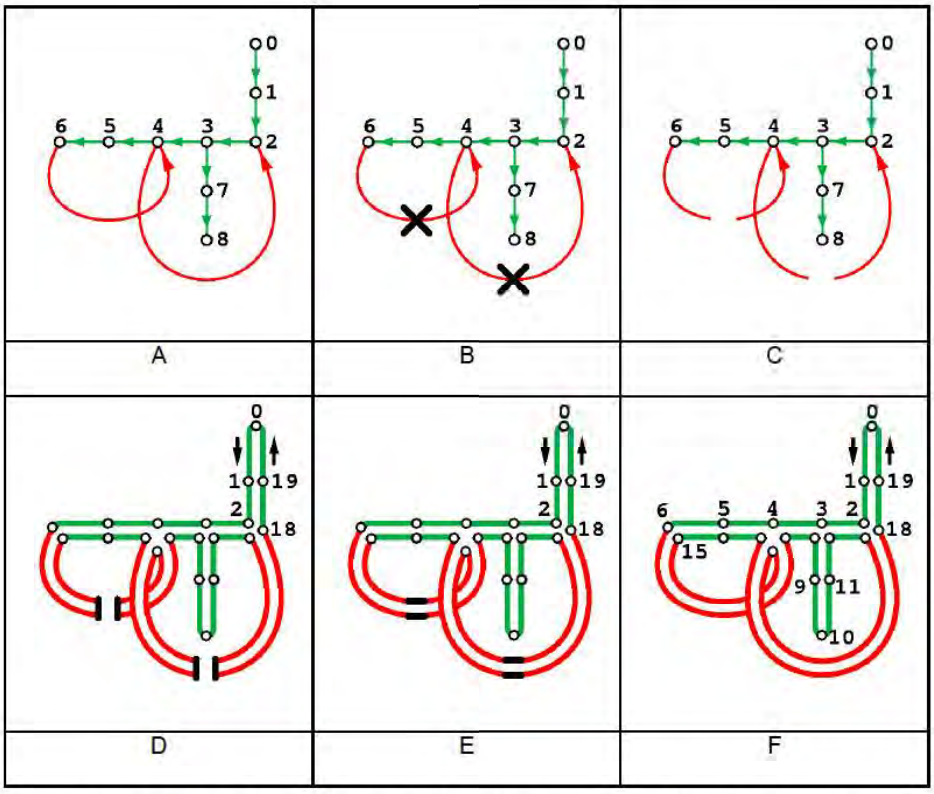


Figure 7-90 — Steps to build the bounding loop from the vertex graph.
A : vertex graph decomposed into vertex tree (green) and jump edges (red).
B : extended vertex tree is created by cutting jump edges in half.
C : the extended vertex tree has two leaves for each jump edge.
D : build the extended vertex tree loop.
E : connect the start and end of each jump edge.
F : resulting boundary loop.

Vertex numbers are assigned according to a depth-first traversal of the graph. The bounding loop look-up table is constructed by going around the graph and for each traversed vertex recording its number.

7.13.6.1 Stitches Decoder

The Stitches decoder reconstructs the Vertex Clustering that should be applied to the vertices in the Vertex Graph to reconstruct the 3D mesh. The following pseudo-code summarizes the operation of the Stitches decoder: if the boolean **has_stitches** flag in the current component is true, then for each vertex of the Vertex Graph in decoder order, the Stitches decoder decodes a stitching command from the bitstream. If the boolean value **stitching_command** is true, then the boolean value **pop_or_get** is decoded; if the boolean value **pop_or_get** is false, an unsigned integer is decoded, and associated to the current vertex i as an anchor (to stitch to). The current vertex i is then pushed to the back of the anchor_stack.

if **pop_or_get** is true, then the boolean value **pop** is decoded, followed with the unsigned integer value **stack_index**.

© ISO/IEC 2004 – All rights reserved

```
decode(anchor_stack) {
 if (has_stitches == true)
  for (i = nV0; i < nV1; i++) {
                // nV0 is the first vertex of the current component,
                // and nV1 –1 is the last vertex
   decode stitch_cmd;
    if (stitch_cmd) {
     decode stitch_pop_or_get;
     if (stitch_pop_or_get) {
       decode stitch_pop;
       decode stitch_stitch_stack_index;
       retrieve stitching_anchor from anchor_stack;
       if (stitch_pop) {
        remove stitching_anchor from anchor_stack;
       } // end if
       decode stitch_incr_length;
       if (stitch_incr_length != 0) {
        decode stitch_incr_length_sign;
       } // end if
       decode stitch_push;
       if (stitch_push)
        push i to the back of anchor_stack
       retrieve stitch_length at anchor;
       total_length = stitch_length + stitch_incr_length;
       if (total_length > 0)
        decode stitch_reverse;
       stitch i to anchor for length of total_length
       and in a reverse fashion if (stitch_reverse);
     } // end if (stitch_pop_or_get)
     decode stitch_length;
     push i to the back of anchor_stack;
     save stitch_length at anchor i;
   } // end if (stitch_cmd)
  } // end for
}
```

© ISO/IEC 2004 – All rights reserved

Using **stack_index**, a stitching_anchor is retrieved from the anchor_stack. This is the anchor that the current vertex i will be stitched to. If the **pop** boolean variable is true, then the stitching_anchor is removed from the anchor_stack. Then, an integer **incremental_length** is decoded as an unsigned integer. If it is different from zero, its sign (boolean **incremental_length_sign**) is decoded, and is used to update the sign of **incremental_length**. A **push_bit** boolean value is decoded. If **push_bit** is true, the current vertex i is pushed to the back of the anchor_stack. An integer **stitch_length** associated with the stitching_anchor is retrieved. A total_length is computed by adding **stitch_length** and **incremental_length**; if total_length is greater than zero, then a **reverse_bit** boolean value is decoded. Then the Vertex Clustering array is updated by stitching the current vertex i to the stitching anchor with a length equal to total_length and potentially using a reverse stitch. The decoder uses a v_father array to perform this operation. The v_father array provides for each vertex in the Vertex Graph the index of its father (predecessor) in the directed Vertex Graph without jumps (potential the vertex itself it it is a root). To stitch the current vertex i to the stitching anchor with a length equal to total_length, starting from both i and the anchor at the same time, we walk along the Vertex Graph as indicated in the v_father entries for both i and the anchor by looking up the v_father entries total_length+1 times, and for each corresponding entries ((i, anchor), (v_father[i],v_father[anchor]), (v_father[v_father[i]], v_father[v_father[anchor]])) record in the Vertex Clustering array that the entry with the largest order of traversal should be the same as the entry with the lowest order of traversal. For decoding the stitches, For instance if (j > k), then Vertex Clustering[j] = Vertex Clustering[k]. The Vertex Clustering array defines a graph that is a forest. Each time an entry in the Vertex Clustering array is changed, we perform path compression on the forest by updating the Vertex Clustering array such that each element refers directly to the root of the tree of the forest it belongs to. The following figure shows both a (forward) stitch and reverse stitch.

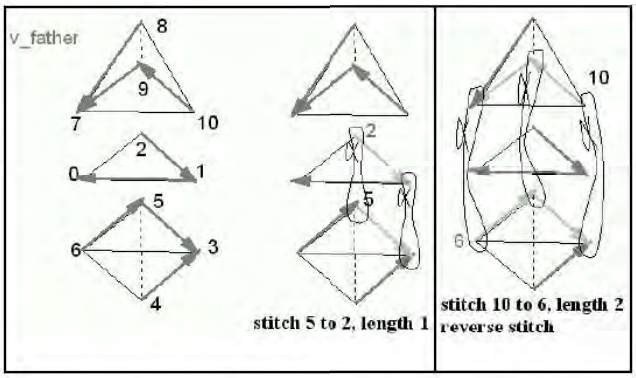


Figure 7-91 — Examples of a (forward) stitch and a reverse stitch

If the stitch is a reverse stitch, then we first follow the v_father entries starting from the anchor for a length equal to total_length (from vertex 6 to vertex 3 in the figure above), recording the intermediate vertices in a temporary array. We then follow the v_father entries starting from the vertex i and for each corresponding entry stored in the temporary array (from the last entry to the first entry), we update the Vertex Clustering array as explained above.

The Vertex Clustering array is used to indicate when a coord or property should be decoded or was already decoded when decoding triangle data as explained in subclause 7.13.8.

7.13.7 Triangle Tree Decoder

The triangle tree decoder reconstructs a sequence of triangle runs. For each run it generates a run length and the size of its sub-tree.

The triangle tree is a sequence of triangle runs. For each run a length **tt_run_length** and a boolean flag **tt_leaf** is given. A branching run is a run for which **tt_leaf** is false and a leaf run a run for which **tt_leaf** is true. The decoder stops decoding the sequence of runs when the number of decoded leaf runs is superior to the number of branching runs. In the syntax a variable depth is used that counts the number of branching runs minus the number of leaf runs. This implies that the total number of runs is always an odd number.

© ISO/IEC 2004 – All rights reserved

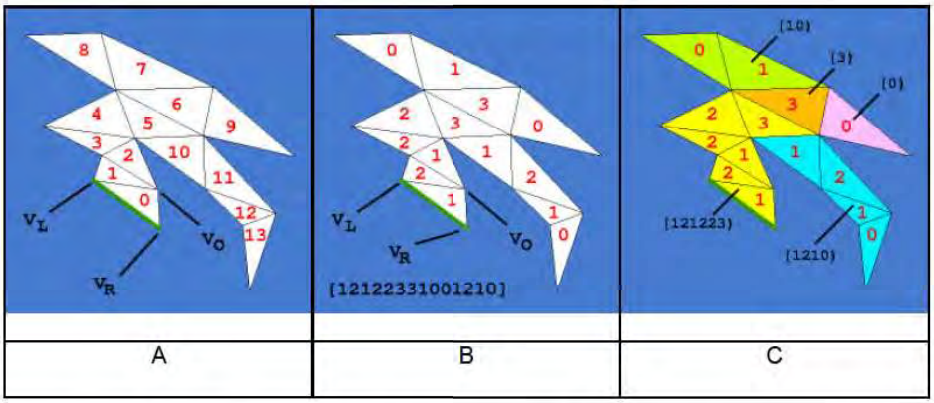


Figure 7-92 — Representing a simple polygon as a table of triangle runs.
A: depth-first order of traversal of the triangle tree starting from the root edge (green).
B: each triangle is classified by a 2-bit code as leaf(0), advance-right (1), advance-left (2), or branching (3).
C : The sequence of 2-bit codes is partitioned into runs ending in branching or leaf triangles. Each run is encoded as a (tt_run_length,tt_leaf) pair, plus a sequence of marching_edge bits.
Optionally, in the case of 3D mshes with polygonal faces, an additional sequence of polygon_edge bits is used to indicate which internal edges of the simple polygon correspond to original internal edges of the mesh, and which ones to virtual internal edges.

The **tt_leaf** flag is also used for the computation of the length of each run.

If the length of each run is correctly decoded and does not have virtual triangles, then the sum of the length of all runs must be equal to the length of the bounding loop constructed by the vertex graph decoder minus 2 (assuming that the bounding loop is correctly reconstructed).

However, if a run is partitioned, the triangle tree bitstream containes one or two virtual leaf triangles. Hence when decoding triangle data, the triangle data corresponding to the virtual leaf triangles shall not be decoded.

There are one or two virtual triangles if the number of triangles in the current partition is less than **codap_right_bloop_idx** - **codap_left_bloop_idx** - 1. Otherwise there are no virtual triangles. If there are virtual triangles and the third-last triangle is a branching triangle, the last two triangles in the partition are the virtual triangles. Other wise only the last triangle is virtual.

7.13.8 Triangle Data Decoder

The triangle data decoder operates in three steps. It first decodes a header, then a root triangle, which is followed by a sequence of triangles. Note that this sequence is empty if there is a single triangle in the current connected component.

The header is a boolean flag **triangulated** that indicates if all faces in the connected component are triangles. If this flag indicates true, the **polygon_edge** field is not coded.

The decoding of a root triangle and of a non-root triangle are the same with the following exception. The **polygon_edge** field is never coded for a root triangle of a connected component, and the number of property samples may differ.

Two fields indicate connectivity information for the current triangle: **marching_edge** and **polygon_edge**. **marching_edge** is not coded for branching and leaf triangles. **polygon_edge** is not coded for the root triangle or if the **triangulated** flag is set true. However in a partition of partition_type_2, the **polygon_edge** is coded for the root triangle if the **triangulated** field is set false

The number of per vertex property samples may differ depending on the boundary prediction mode and visited vertices. If the boundary prediction mode is restricted, then the two starting points are already visited in the

© ISO/IEC 2004 – All rights reserved

previous partition and only one root sample is coded. However, if the remaining vertex is also visited, there are no samples coded. If the boundary prediction mode is extended, the root triangle always has all 3 samples.

Table 7-24 — Number of samples to decode in a root triangle for each binding

Binding	# samples	Detail
None	0	
per vertex	0, 1, or 3	1 root and 2 non-root samples
per face	1	1 root sample
per corner	3	2 root and 1 non-root samples

For non-root triangles, the triangle data decoder decodes zero or one sample for each property bound per vertex. If the opposite vertex of the current triangle has been visited before, the value of the sample is already known and does not need to be decoded again. If the opposite vertex of the current triangle has never been visited before, then a sample is read. The bounding loop look-up table and the sizes of subtrees of the triangle tree are necessary to determine whether the opposite vertex of the current triangle has been visited previously. The boundary prediction mode is necessary to determine whether to consider the visited vertices within the partition or including every vertices visited in the previous partitions.

Table 7-25 — Number of samples to decode in a non-root triangle for each binding. The number of samples may be conditioned.

Binding	# samples	Condition
None	0	
per vertex	0 or 1	vertex never visited
per face	0 or 1	**polygon_edge** is set
per corner	1 or 3	**polygon_edge** is set

The traversal order is fixed in basic topological surgery. However, in order to change traversal order, **td_orientation** field is coded at a branching triangle on the main stem of the triangle tree in the partition. In other words, if the number of previously decoded branching triangles is equal to that of leaf triangles within the current partition, i.e. the depth value is zero, the **td_orientation** field is coded.

7.13.8.1 Decoding a root sample

A root sample is the first decoded sample in a root triangle. If there are several samples in a root triangle only the first sample is a root sample. The remaining samples in the root triangle are non-root samples.

The **coord_quant** bits of each coordinate of a root geometry sample are received starting with the most significant.

7.13.8.2 Decoding a non-root sample

A coordinate of a non-root geometry sample is composed of **coord_quant** bits, plus a sign bit. The coordinate is coded bit by bit starting from the most significant. The leading 0's, if any, and the leftmost 1, if any, are **coord_leading_bit**'s are coded using an adaptive probability estimation model. If not all **coord_leading_bits** are 0's a **coord_sign_bit** is coded. Finally **coord_quant** minus the number of **coord_leading_bits** **coord_trailing_bits** are coded using a fixed probability model.

3***coord_quant** probability estimation models are defined to code the **coord_leading_bit** field. The probability model is selected according to the bit position and to the coordinate number.

The decoded value is not the actual value of the sample, but a correction that has to be applied to the prediction to obtain the reconstructed sample. The reconstructed sample is equal to the sum of the prediction and of the decoded value. The computation of the prediction is described below.

© ISO/IEC 2004 – All rights reserved

It may happen that a reconstructed value is outside the bounding box, in which case a masking operation is carried out to insure that the final reconstructed value is indeed inside the bounding box. The masking operation depends on the quantisation step **coord_quant**. The **coord_quant** less significant bits of the reconstructed value are kept as is, and all other (more significant) bits are cleared.

The same structure applies to the other properties.

7.13.8.3 Property Prediction

For increased coding efficiency, the value of a property is coded relative to a prediction. This subclause describes the computation of the prediction for each type of binding. Table 7-26 and Table 7-27 describe all the valid combinations of property binding, property type, and prediction type.

Table 7-26 — Valid property bindings for each property type

binding	meaning	coord	normal	color	texCoord
00	not encoded		X	X	X
01	per vertex	X	X	X	X
10	per face		X	X	
11	per corner		X	X	X

Table 7-27 — Valid combinations of prediction type and property binding

pred_type	nlambda	binding
no_prediction	not coded	bound_per_vertex, bound_per_face, or bound_per_corner
tree_prediction	1	bound_per_face or bound_per_corner
parallelogram_prediction	3	bound_per_vertex

For tree and parallelogram prediction, the prediction of the properties in the current triangle are always based on the values in the previous triangle in the triangle tree. The prediction is weighted by the lambda coefficients decoded by the header decoder.

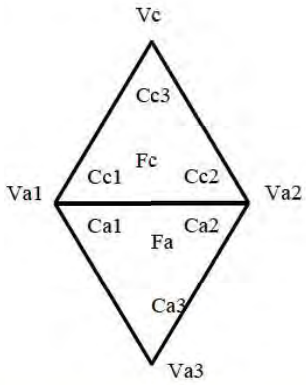


Figure 7-93 — Prediction of property values.
The value at the current vertex Vc is predicted from the values at ancestor vertices Va1, Va2 and Va3.
The value at the current face Fc is predicted from the value at ancestor face Fa.
The value at the current corner Cc1 is predicted from the value at ancestor corner Ca1.
The value at current corner Cc2 is predicted from the value at ancestor corner Ca2.
The value at current corner Cc3 is predicted from the value at current corner Cc1.

© ISO/IEC 2004 – All rights reserved

7.13.8.3.1 Tree prediction

If the property is bound per face, the prediction of Fc is equal to Fa. If the current triangle is the root of the triangle tree, Fc is not predicted.

If the property is bound per corner, the prediction of Cc1 is equal to Ca1, the prediction of Cc2 to Ca2, and the prediction of Cc3 to Cc1. If the current triangle is the root of the triangle tree, Cc1 is not predicted, Cc2 is predicted by Cc1, and Cc3 by Cc1.

7.13.8.3.2 Parallelogram prediction

If the property is bound per vertex, the prediction of Vc is equal to lambda1*Va1 + lambda2*Va2 + lambda3*Va3. If the current triangle is the root of the triangle tree, Va1 and Va2 must be encoded too, Va1 is not predicted, Va2 is predicted by Va1, and Vc by Va1.

7.13.8.4 Inverse quantisation

There are two inverse quantisation procedures. The first one applies to geometry, colors and texture coordinates. The second applies to normals.

Given a quantised geometry sample (qx, qy, qz), the reconstructed geometry sample (x,y,z) is obtained as:

x = **coord_xmin** + ((1<<**coord_quant**)-1)*__coord_size__*xq
y = **coord_ymin** + ((1<<**coord_quant**)-1)*__coord_size__*yq
z = **coord_zmin** + ((1<<**coord_quant**)-1)*__coord_size__*zq

The procedure for colors and texture coordinates is the same.

There is a special inverse quantisation procedure for normals. Normals are always quantised to an odd number of bits **normal_quant**. The 3 most significant bits of **normal_quant** indicate the octant the normal lies in, and the remaining bits define an index within the octant.

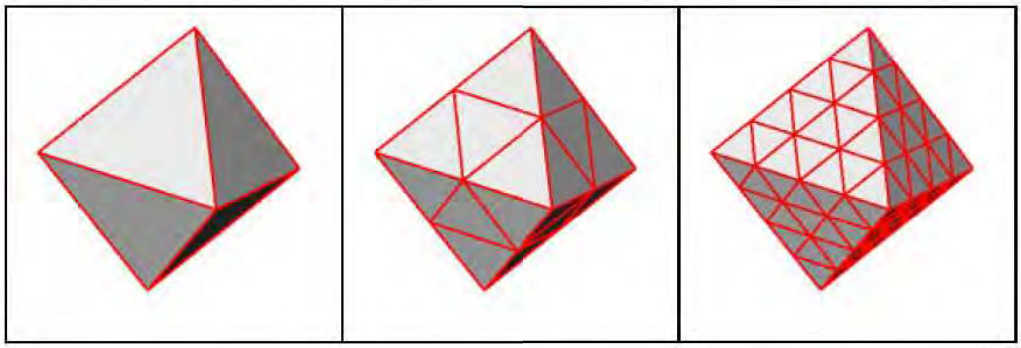


Figure 7-94 — The quantisation of normals is based on the hierarchical subdivision of the unit octahedron. Left: unit octahedron. Middle: unit octahedron after one subdivsion. Right: unit octahedron after two subdivisions.

© ISO/IEC 2004 – All rights reserved

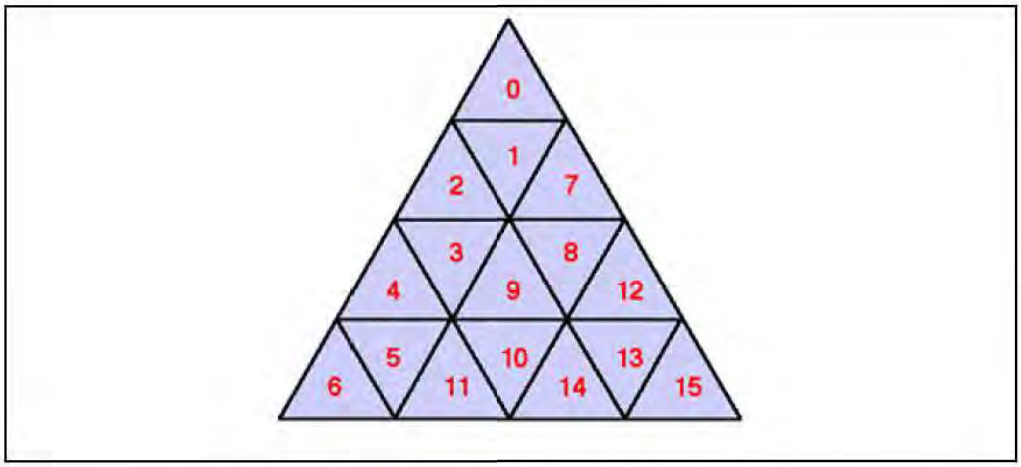


**Figure 7-95 — Numbering of the triangles within an octant.
Note that the top corner corresponds to |z|=1,
the leftmost corner to |x|=1 and the rightmost corner to |y|=1.**

Given an index i, and the 3 sign bits sx, sy and sz, the normal (x,y,z) can be analytically reconstructed as follows:

```
nbins = 1 << ((normal_quant - 3) / 2);

y0 = nbins - ceil(sqrt(nbins*nbins - i));
x0 = i + y0*y0;
skew = (x0 & 1)*2.0/3.0;
x0 = (x0 >> 1) & (nbins - 1);

x = (float)x0 + skew;
y = (float)y0 + skew;
z = (float)nbins - x - y;

n = 1.0/sqrt(x*x + y*y + z*z);
x = (sx) ? -x*n : x*n;
y = (sy) ? -y*n : y*n;
z = (sz) ? -z*n : z*n;
```

7.13.9 Forest Split decoder

In the progressive forest split (PFS) scheme, a 3D model is represented as a low resolution mesh followed by a sequence of forest split operations. A forest split operation is represented by a forest in the graph of vertices and edges of a mesh, a sequence of simple polygons, and a sequence of vertex coordinates and property updates. The mesh is refined by cutting the mesh through the forest, splitting the resulting boundaries apart, filling each of the resulting tree boundary loops with one of the simple polygons, and finally, displacing the new vertices and properties. An optional pre-smoothing step can be used as a global predictor for the vertex coordinates, after the and properties of the new faces bound per face and per corner are updated. Also, and optional post-smoothing step can be used to reduce the quantisation artifacts after all the updates have been applied.

7.13.9.1 Enumeration of Mesh Elements

Given a manifold triangular 3D mesh with *V* vertices, *T* triangles and no boundaries, we assume that the vertices have consecutive *vertex indices* in the range *0, …, V-1*, and the triangles have consecutive *triangle indices* in the range *0,…,T-1*. The edges of the mesh, which are represented by pairs of vertex indices *(i,j)*, with $i < j$, are ordered lexicographically and assigned consecutive *edge indices* in the range *0,…,E-1*. The trees in the forest are ordered according to the minimum vertex index of each tree. The *root vertex* of each tree in the forest is the leaf of the tree with the minimum index. Starting at the root, the boundary loop created by cutting along the tree can be traversed

© ISO/IEC 2004 – All rights reserved

in cyclic fashion in one of the two directions. The *root edge* of the tree is the only edge of the tree that has the root vertex as an endpoint. Of the two triangles incident to the root edge of the tree, the *root triangle* of the tree is the one with minimum triangle index. Of the two edges of the tree boundary loop corresponding to the root edge of the tree, the *root edge* of the tree boundary loop is the one incident to the root triangle.

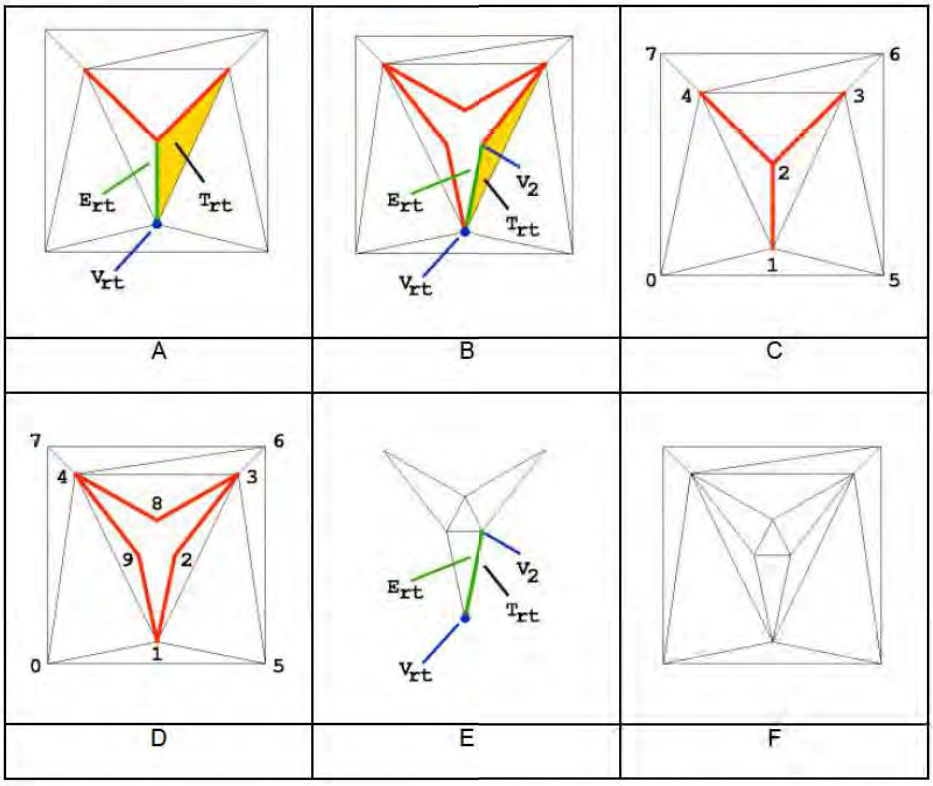


Figure 7-96 — When a mesh is cut through a tree of edges (red and green edges in A), a tree boundary loop (red and green edges in B) is created with each edge of the tree corresponding to two edges of the boundary loop. Some vertex indices are assigned before cutting (in C) to new tree boundary loop vertices, others are assigned subsequent indices (in D). The hole created by the cutting operation is filled by triangulating the boundary loop using a simple polygon (E) resulting in a refined mesh (F) with the same topological type as the initial mesh.

7.13.9.2 Decoding the forest

Figure 7-97 describes the algorithm to decode the forest of edges from the bitstream. Initially, the forest is empty. As edge marks are decoded, the corresponding marked edges are included in the forest. Edges which if marked would create loops are skiped, because they can be predicted as not being marked. The order of the edge bits in the bitstream correspond to the enumeration of edges defined above.

© ISO/IEC 2004 – All rights reserved

```
// forest decoder
// marked edges form a forest in the graph of the mesh

int nV = number of vertices of mesh
Partition p(nV)
// p defines a partition of a set of nV elements
// initially each part contains a single element
// the method find(i) retrieves the number of the part containing element i
// the method join(i,j) joins the parts containing elements i and j, respectively

for ( every edge e={i,j} of mesh ) {
 bool skip_edge = false;
 bool edge_bit  = false;
 if(p.find (i)!=p.find(j)) {
   edge_bit = decode_bit();
 } else {
   // forest => edge_bit==false
   skip_edge = true;
 }
 e.isInForest = edge_bit;
 if(edge_bit==true)
   p.join(I,j);
}
```

Figure 7-97 — forest decoder pseudo-code

7.13.9.3 Cutting Through Forest Edges

Cutting through a forest of edges can be performed sequentially, cutting through one tree at a time. Each cut is typically a local operation, affecting only the triangles incident to vertices and edges of the tree. However, as in the Topological Surgery method, a single cut could involve all the triangles of the mesh. Cutting requires duplicating some tree vertices, assigning additional indices to the new vertices and fixing the specification of the affected triangles. If no tree vertex is a boundary vertex of the mesh, then the tree is completely surrounded by triangles. This is illustrated Figure 7-98 A,B.

© ISO/IEC 2004 – All rights reserved

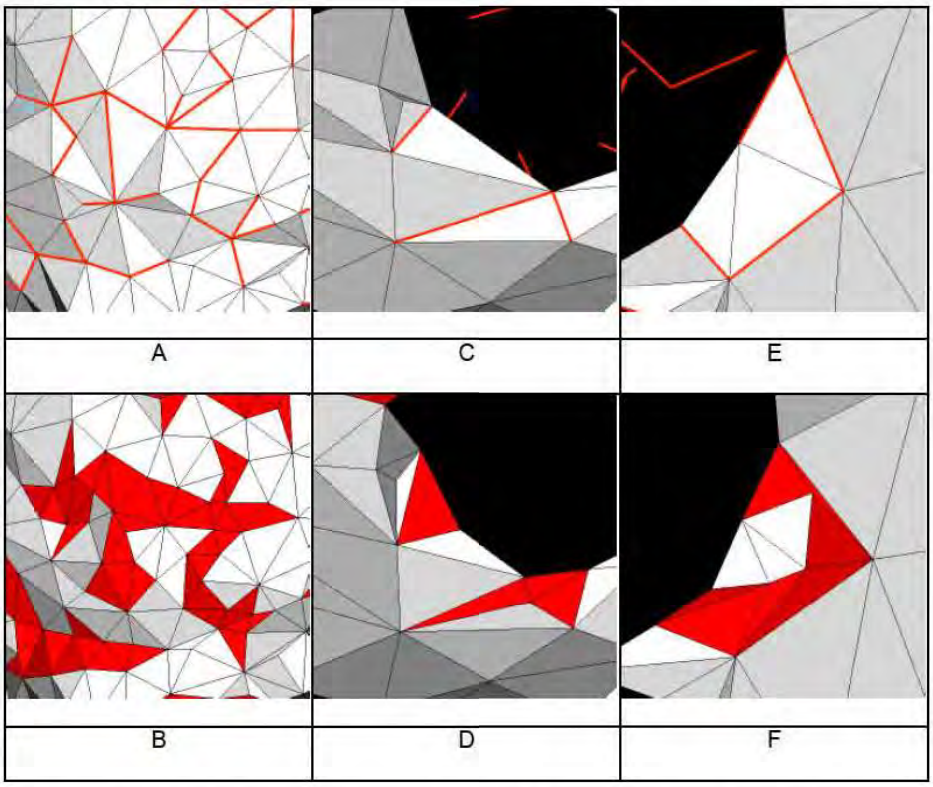


Figure 7-98 — Construction and triangulation of tree boundary loops.
A,B: No tree vertices belong to the mesh boundary.
C,D: A tree vertex isolated in the mesh boundary requires an extra tree loop edge.
E,F: A tree edge on the mesh boundary does not require an extra tree loop edge, but some of the new vertex indices may only be used by new triangles.
Note that the tree may have several contacts with the mesh boundary.

Starting at the root triangle, all the corners of affected triangles can be visited in the order of traversal of the tree boundary loop, by jumping from triangle to neighboring triangle, while always keeping contact with the tree. The direction of traversal is defined by going from the root edge e_{rt} towards the root triangle t_{rt}. This process produces a list of triangle corners, called the *corner loop*, whose values need to be updated with the new vertex indices. While traversing this list, we encounter runs of corners corresponding to the same vertex index before the cut. A new vertex index must be assigned to each one of these runs. To prevent gaps in the list of vertex indices we first need to reuse the vertex indices of the tree vertices, which otherwise would not be corner values of any triangles. The first visited run corresponding to one of these vertices is assigned that vertex index. The next visited run corresponding to the same vertex index is assigned the first vertex index not yet assigned above the number of vertices of the mesh before the cut. This procedure performs the topological cut. For example, in Figure 7-98 C, the vertex index values of the corners in the corner loop list are:

[1233333244444421111]

The list can be decomposed into 6 runs [11111], [2], [33333], [2], [444444] and [2]. As shown in Figure 7-98 D, the vertex indices that are assigned to these runs are 1,2,3,8,4,9. A tree with m edges containing no mesh boundary vertices creates a tree boundary loop of $2m$ edges. When one or more vertices belong to the mesh boundary, the boundary loop has some edges without incident triangles. Cases when vertices belong to the mesh boundary are illustrated in Figure 7-98 C,D, E,F. There are three cases: tree edges belonging to the boundary (Case 1, shown in Figure 7-98 E,F), a tree vertex that is isolated on the mesh boundary (Case 2, shown in Figure 7-98 C,D) or an isolated tree vertex on the boundary that happens to be the root vertex v_{rt} (Case 3, which is a sub-case of Case 2). Case 1 is treated similarly to the general case: we loop around the vertex tree, creating 2 new loop edges for each

© ISO/IEC 2004 – All rights reserved

marked edge. Some of these edges do not belong to any existing triangle. The method that is used for looping around boundary edges must take into account that there are no triangles on the other side. One possibility is to create fake faces on the other side of the boundary and keep the same method for jumping from triangle to triangle. Another possibility, which is exploited in the sample implementation, is to start jumping in the other direction when we hit the boundary. Except from these difference in the algorithm, the same method is used to assign new vertex indices for some of the loop corners. In Case 2, an extra edge should be added to the loop. In Case 3, an extra edge is added as well, but it remains to determine which of the two endpoints of the created edge will be the root vertex of the loop. By convention, it will be the endpoint to which the root triangle is incident. These three cases are illustrated in Figure 7-99. In summary, a tree with *m* edges containing *n* isolated boundary vertices creates a tree boundary loop of *2m+n* edges

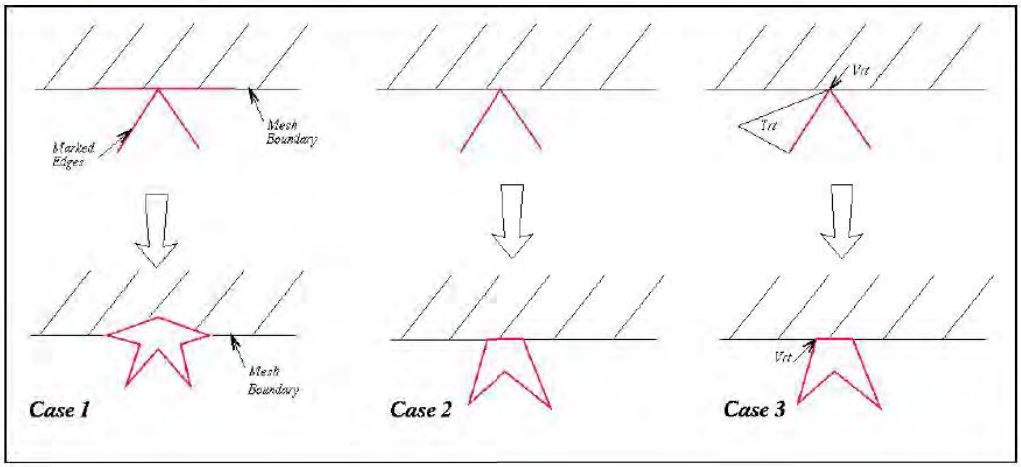


Figure 7-99 — Special cases encountered while constructing the tree boundary loop correrssponding to a tree touching the boundary of the mesh

7.13.9.4 Triangulating Tree Boundary Loops

By replacing each run of corners in the corner loop with the assigned vertex index, we construct a new list representing the tree boundary loop. If the tree boundary loop has m vertices, so does the corresponding polygon boundary loop. Each triangle $t = \{i,j,k\}$ of the simple polygon defines a new triangle of the refined mesh by replacing the polygon boundary loop indices *i,j,k* with their corresponding tree boundary loop indices. Figure 7-98 E,F illustrates this. This is done using the list representing the tree boundary loop as a lookup table. The triangles of the simple polygon are visited in the order of a depth first traversal of its dual tree. The traversal starts with the triangle opposite to the root triangle and always traverses the left bran ch of a branching triangle first.

7.13.9.5 Simple polygon properties bound per face and per corner

The properties bound per face and per corners to the faces of the simple polygons used to fill the tree boundary loops are encoded as in the case of a connected component of a single resolution 3D mesh, predicting per face values along the face tree, and per corner values along the derived corner tree.

7.13.9.6 Tree loop vertex, face, and corner properties

Vertex coordinates and properties bound per vertex corresponding to new tree loop vertices are first assigned the same values as the corresponding tree vertex coordinates and properties before the cut. To prevent the appearance of holes, these vertices are displaced after the boundary loops are triangulated. Rather than encoding the new absolute positions of the tree loop vertices and corresponding per vertex properties, we encode their displacement with respect to the position before the forest split operation, and optionally after pre-smoothing (pre-smoothing is only applied to vertex coordinates). These values are encoded in the order of traversal of the tree loop vertices.

© ISO/IEC 2004 – All rights reserved

Properties bound per face and per corner to tree loop faces and corners are encoded in a similar fashion, except that properties bound per face (a face may be a tree loop face for two or more trees) are encoded with no repetition, skipping values associated with faces previously visited during the current update cycle.

7.13.9.6.1 Pre and post-smoothing

The differences between vertex positions before and after each forest split operation can be made smaller by representing these errors as the sum of a global predictor plus a correction. We use the signal processing smoothing method [4] in Annex O as a global predictor. Figure 7-100 shows a pseudo-code implementation of the scheme used for both pre and post smoothing.

```
// pre and post smoothing

int   nV    // number of vertices of mesh
Vertex v[nV]  // array of vertices of the mesh
Vertex dv[nV] // auxiliary array
int   nv[nV] // auxiliary array

// determine sharp vertices from sharp edges
for(i=0;i<nV;i++)
 v[i].isSharp = false
for ( every edge e={i,j} of mesh ) {
 if(e.isSharp) {
  v[i].isSharp = true
  v[j].isSharp = true
 }
}

// main loop
for(n=0;n<n_smooth;n++) {

// accumulate displacements
 dv = 0;
 nv = 0;
 for ( every edge e={i,j} of mesh ) {
  dve = v[j] - v[i];
  if(v[i].isFixed==false && (e.isSharp==true || v[i].isSharp==false)) {
   dv[i] += dve;
   nv[i] += 1;
  }
  if(v[j].isFixed==false && (e.isSharp==true || v[j].isSharp==false)) {
   dv[j] -= dve;
   nv[j] += 1;
```


© ISO/IEC 2004 – All rights reserved

```
  }
}

// normalize displacements and apply
float a = (n is even)?lambda_smooth:mu_smooth;
for(i=0;i<nV;i++)
 if(nv[i]>0) {
    dv[i] /= nv[i]
    v[i] += a*dv[i];
  }
}
```

Figure 7-100 — Pseudo-code for smoothing operator with discontinuty edges and fixed vertices

An optional pre-smoothing step is applied as a global predictor. After the connectivity refinement step of a forest split operation is applied, the new vertices are positioned where their corresponding vertices in the previous level were positioned and the mesh has many edges of zero length (all the new triangles have zero surface area). The signal processing smoothing method, which tends to equalize the size of neighboring triangles, brings the endpoints of most such edges apart, most often reducing the distance to the desired vertex positions. The corrections, the differences between the vertex positions after the split operation and the result of smoothing the positions before the split operation, are then quantised.

If the 3D mesh approximates a smooth shape, a post-smoothing step can be applied to reduce the visual artifacts produced by the quantisation process. In many cases, even fewer bits per coordinate can be used in the quantisation process without a significant perceptual difference, also reducing the total length of the encoded vertex displacements.

In both the pre and post-smoothing steps optional discontinuity edges and/or fixed vertices can be specified in the bitstream, together with the smoothing parameters. If so, the same discontinuity edges and/or fixed vertices are used for both steps.

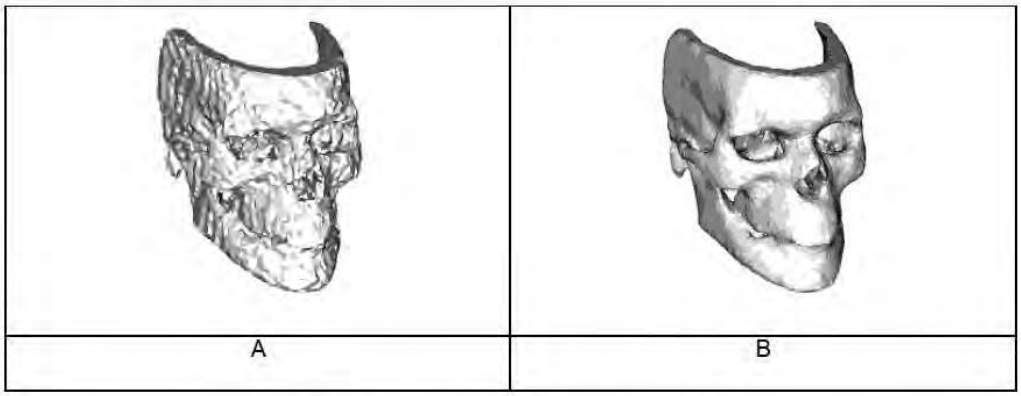


Figure 7-101 — Smoothing a 3D model.
(A) Original 3D model. (B) 3D model after applying the smoothing operator introduced in [4] in Annex O.

© ISO/IEC 2004 – All rights reserved

7.13.10 Arithmetic decoder

This section describes the QF-coder for arithmetic decoding. C++-style routines are provided. The arithmetic decoder relies on a set of variables that are described in the table below.

Register	Description
A	size of current interval
C	current arithmetic code
count	number of renormalisations before reading next byte
symbol	value of decoded symbol
Qe	probability estimate for least probable symbol
MPS	value of most probable symbol

The registers Qe and MPS are dependent upon a context. For each context, there is a state variable including the Qe and MPS registers.

7.13.10.1 Initialisation

The arithmetic coder reads data from the input on a per byte basis. Therefore the stream pointer shall be byte aligned before initialising the arithmetic decoder.

In the initialisation process, all variables are initialised to proper values. 2 bytes are read from the input stream into register C.

```
void qf_start() {

  A = 0x00010000;

  C = 0x00010000;

  count = 0;

  sw_renorm_decode();

}
```

7.13.10.2 Decoding a symbol

When a binary symbol is decoded, the interval is split into two. The half in which C lies determines the value of the bit. The interval is then reduced to the corresponding half. If the size of the new interval is smaller than 0x80000000, then the interval is renormalised and the probability estimator updated.

```
void qf_decode(int *symbol, QState *state) {

  *symbol = state->MPS;

  A -= state->Qe;
```


© ISO/IEC 2004 – All rights reserved

```
  if (C <= A) {
    if (A & 0x80000000)
      return;
    *symbol ^= A < state->Qe;
  }
  else {
    *symbol ^= A >= state->Qe;
    C = C - A;
    A = state->Qe;
  }
  state->update(*symbol);
  renorm_decode();
}
```

7.13.10.3 Renormalisation

In the renormalisation step, the size of the interval is doubled until it reaches at least 0x80000000. Each time the size of the interval is doubled a bit is consumed. When no more bit is available for consumption in the less significant bits of the register C, a new byte is retrieved from the stream.

```
void qf_renorm() {
  do {
    if (count == 0) {
      int b = get_byte();
      C -= 0xff << 8;
      C += b << 8;
       count = (b == 0) ? 7 : 8;
    }
    count--;
    C = C << 1;
    A = A << 1;
  } while (!(A & 0x80000000));
}
```

© ISO/IEC 2004 – All rights reserved

7.13.10.4 Probability estimation

A majority of the fields that are arithmetic coded benefit from adaptive probability estimation. This adaptive estimation is defined by a Markov Model. Each state of the model defines a probability of the LPS, a next state if the currently coded symbol is the MPS, a next state if the currently coded symbol is the LPS, and a flag that indicates if the value of the MPS should be changed if the currently coded symbol is the LPS. The Markov model to be used is defined in Figure 7-84. It is named FA-JPEG for Fast-Attack JPEG, and has been designed such as to minimize the number of states.

A fixed probability of the LPS and a fixed MPS are defined for the fields for which there is no adaptive probability estimation.

Table 7-28 lists the probability estimation used for each of the fields.

The models are global and initialised to index = 0, MPS = 0 before the first connected component.

The context of order 1 models is reset before each component.

The state is updated every time the renormalisation procedure is called.

Table 7-28 — List of fields that are arithmetic coded and their associated probability estimation

Field	probability estimation	context
last_component	FA-JPEG	last_component_context
has_stitches	FA-JPEG	has_stitches_context
codap_last_vg	FA-JPEG	codap_last_vg_context
codap_vg_id	FA-JPEG	zero_context
codap_left_bloop_idx	FA-JPEG	zero_context
codap_right_bloop_idx	FA-JPEG	zero_context
codap_bdry_pred	FA-JPEG	zero_context
vg_simple	FA-JPEG	vg_simple_context
vg_last	FA-JPEG	vg_last_context
vg_forward_run	FA-JPEG	vg_forward_run_context
vg_loop_index	FA-JPEG	vg_loop_index_context
vg_run_length	FA-JPEG	vg_run_length_context
vg_leaf	FA-JPEG	vg_leaf_context
vg_loop	FA-JPEG	vg_loop_context
tt_run_length	FA-JPEG	tt_run_length_context
tt_leaf	FA-JPEG	tt_leaf_context
codap_branch_len	FA-JPEG	codap_branch_len_context
triangulated	FA-JPEG	triangulated_context
marching_edge	FA-JPEG, order 1, initial context = 1	marching_edge_context[0..1]
td_orientation	FA-JPEG	td_orientation_context
polygon_edge	FA-JPEG, order 1, initial context = 1	polygon_edge_context[0..1]
coord_bit	fixed Qe = 0x5601, MPS = 0	zero_context
coord_leading_bit	FA-JPEG, 3*coord_quant contexts	coord_leading_bit_context[0..3*coord_quant-1]
coord_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
coord_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
normal_bit	fixed Qe = 0x5601, MPS = 0	zero_context
normal_leading_bit	FA-JPEG, normal_quant contexts	normal_leading_bit_context[0..normal_quant-1]
normal_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context

© ISO/IEC 2004 – All rights reserved

normal_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_leading_bit	FA-JPEG, 3*color_quant contexts	color_leading_bit_context[0..3*color_quant-1]
color_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_leading_bit	FA-JPEG, 2*texCoord_quant contexts	textCoord_leading_bit_context[0..2*texCoord_quant-1]
texCoord_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
stitch_cmd	FA-JPEG	stitch_cmd_context
stitch_pop_or_get	FA-JPEG	stitch_pop_or_get_context
stitch_pop	FA-JPEG	stitch_pop_context
stitch_stack_index	FA-JPEG	stitch_stack_index_context
stitch_incr_length	FA-JPEG	stitch_incr_length_context
stitch_incr_length_sgn	FA-JPEG	stitch_incr_length_sgn_context
stitch_push	FA-JPEG	stitch_push_context
stitch_reverse	FA-JPEG	stitch_reverse_context
stitch_length	FA-JPEG	stitch_length_context
pfs_forest_edge	FA-JPEG	pfs_forest_edge_context
smooth_with_sharp_edges	fixed Qe = 0x5601, MPS = 0	zero_context
smooth_sharp_edge	FA-JPEG	smooth_sharp_edge_context
smooth_with_fixed_vertices	fixed Qe = 0x5601, MPS = 0	zero_context
smooth_fixed_vertex	FA-JPEG	smooth_fixed_vertex_context
other_update	fixed Qe = 0x5601, MPS = 0	zero_context

Table 7-29 — The Fast Attack JPEG (FA-JPEG) Markov model to estimate probabilities. For each state, we define the probability of the LPS, the next state in the event of an MPS, the next state in the event of an LPS, and a switch MPS flag in the event of an LPS.

State	Probability of LPS	Next state if MPS	Next state if LPS	Switch MPS if LPS
0	0x5601	1	1	yes
1	0x3401	2	6	no
2	0x1801	3	9	no
3	0x0ac1	4	12	no
4	0x0521	5	29	no
5	0x0221	38	33	no
6	0x5601	7	6	yes
7	0x5401	8	14	no
8	0x4801	9	14	no
9	0x3801	10	14	no
10	0x3001	11	17	no
11	0x2401	12	18	no
12	0x1c01	13	20	no
13	0x1601	29	21	no
14	0x5601	15	14	yes

© ISO/IEC 2004 – All rights reserved

15	0x5401	16	14	no
16	0x5101	17	15	no
17	0x4801	18	16	no
18	0x3801	19	17	no
19	0x3401	20	18	no
20	0x3001	21	19	no
21	0x2801	22	19	no
22	0x2401	23	20	no
23	0x2201	24	21	no
24	0x1c01	25	22	no
25	0x1801	26	23	no
26	0x1601	27	24	no
27	0x1401	28	25	no
28	0x1201	29	26	no
29	0x1101	30	27	no
30	0x0ac1	31	28	no
31	0x09c1	32	29	no
32	0x08a1	33	30	no
33	0x0521	34	31	no
34	0x0441	35	32	no
35	0x02a1	36	33	no
36	0x0221	37	34	no
37	0x0141	38	35	no
38	0x0111	39	36	no
39	0x0085	40	37	no
40	0x0049	41	38	no
41	0x0025	42	39	no
42	0x0015	43	40	no
43	0x0009	44	41	no
44	0x0005	45	42	no
45	0x0001	45	43	no

7.14 NEWPRED mode decoding

NEWPRED is a technique in which the reference picture for inter-frame coding is replaced adaptively. This may be according to the upstream messages from the decoder.

7.14.1 Decoder Definition

In NEWPRED mode, decoder may have the additional reference VOP memory to store previously decoded VOPs. When the reference VOP is indicated by vop_id_for_prediction, the decoder select reference VOP from reference VOP memory or additional reference VOP memory to switch the reference picture to the indicated one.

The unit of switching reference VOP, which is NEWPRED segment (NP segment), is indicated by newpred_segment_type. When this flag is "Video Packet", reference VOP shall be switched by Video Packets. In this case, after switching reference VOP, outside of current Video Packet shall be padded in the same way as Unrestricted MC. Each Video Packet boundaries are treated as VOP boundaries, including the treatment of motion vectors which cross those boundaries. The unrestricted motion vector technique is used for Video Packet boundaries. MV prediction and Intra AC/DC prediction are restricted to prediction within the Video Packet. When newpred_segment_type flag is "VOP", reference VOP shall be switched by VOPs.

© ISO/IEC 2004 – All rights reserved

Regardless of newpred_segment_type, the decoder shall not refer to any NP segment which is decoded before the last I_VOP.

7.14.1.1 Position of resynchronization marker

When newpred_segment_type is "Video Packet", it is implied that the positions of Resynchronization Markers are not changed between I_VOPs.

7.14.2 Upstream message

The decoder may return the upstream message to the encoder based on the upstream syntax defined in 6.2.12. Which type of message is required for the encoder is indicated in requested_upstream_message_type in the VOL header of the downstream data. To return the upstream message is not normative procedure.

7.15 Output of the decoding process

This subclause describes the output of the theoretical model of the decoding process that decodes bitstreams conforming to this part of ISO/IEC 14496.

The visual decoding process input is one or more coded visual bitstreams (one for each of the layers). The visual layers are generally multiplexed by the means of a system stream that also contains timing information.

7.15.1 Video data

The output of the video decoding process is a series of VOPs that are normally the input of a display process. he order in which fields or VOPs are output by the decoding process is called the display order, and may be different from the coded order (when B-VOPs are used).

7.15.2 2D Mesh data

The output of the decoding process is a series of one or more mesh object planes. The mesh object planes are normally input to a compositor that maps the texture of a related video object or still texture object onto each mesh. The coded order and the composition order of the mesh object planes are identical.

7.15.3 Face animation parameter data

The output of the decoding process is a sequence of facial animation parameters. They are input to a display process that uses the parameters to animate a face object.

7.16 Video object decoding for the studio profile

This subclause specifies the video object decoding for the studio profile.

7.16.1 Video decoding process

(The identical description to subclause 7.1)

7.16.2 Higher syntactic structures

The various parameters and flags in the bitstream for StudioVideoObjectLayer(), Group_of_StudioVideoObjectPlane(), StudioVideoObjectPlane(), StudioSlice(), StudioMacroblock(), StudioBlock() and StudioDPCMBlock(), as well as other syntactic structures related to them shall be interpreted as discussed earlier. Many of these parameters and flags affect the decoding process. Once all the macroblocks in a given VOP have been processed, the entire VOP will have been reconstructed.

Texture data in a macroblock shall have been encoded by either DCT or DPCM.

© ISO/IEC 2004 – All rights reserved

If compression_mode == 1, texture data in the macroblock shall have been encoded by DCT. The decoding process is indicated in subclause 7.16.4.

If compression_mode == 0, texture data in the macroblock shall have been encoded by DPCM. The decoding process is indicated in subclause 7.16.5.

Reconstructed fields shall be associated together in pairs to form reconstructed frames. (See "vop_structure" in 6.3.13.5.)

The sequence of reconstructed frames shall be reordered as described in 6.3.13.5.

If progressive_sequence == 1 the reconstructed frames shall be output from the decoding process at regular intervals of the frame period as shown in Figure 7-117.

If progressive_sequence == 0 the reconstructed frames shall be broken into a sequence of fields which shall be output from the decoding process at regular intervals of the field period as shown in Figure 7-118. In the case that a frame picture has repeat_first_field == 1 the first field of the frame shall be repeated after the second field. (See "repeat_first_field" in 6.3.13.5.)

7.16.3 VOP reconstruction

The luminance and chrominance values of a VOP from the decoded texture and motion information are reconstructed as indicated in this subclause. Figure 7-102 represents the process.

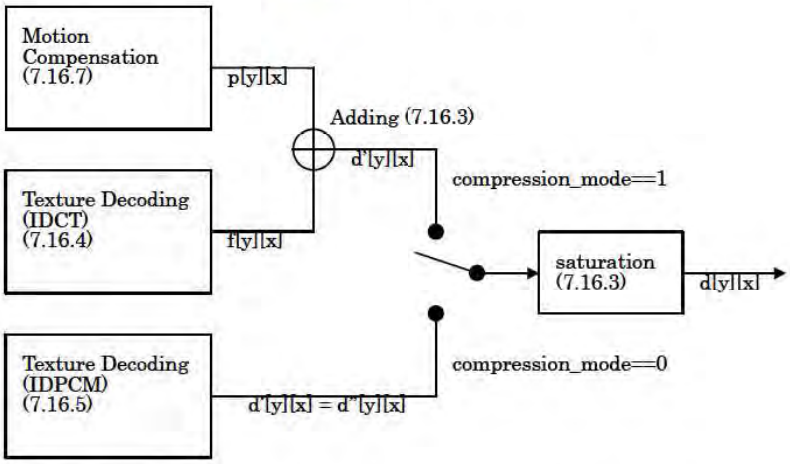


Figure 7-102 — VOP reconstruction from the decoded texutre and motion compensated data

1. In case of INTRA macroblocks of compression_mode==1, the luminance and chrominance values f[y][x] from the decoded texture data form the luminance and chrominance values: d'[y][x] = f[y][x].

2. In case of INTER macroblocks, first the prediction values p[y][x] are calculated using the decoded motion vector information and the texture information of the respective reference VOPs. Then, the decoded texture data f[y][x] is added to the prediction values, resulting in the luminance and chrominance values: d'[y][x] = p[y][x] + f[y][x].

3. In case of INTRA macroblocks of compression_mode==0, the luminance and chrominance values d''[y][x] from the decoded texture data form the luminance and chrominance values: d'[y][x] = d''[y][x].

© ISO/IEC 2004 – All rights reserved

4. Finally, the calculated luminance and chrominance values are saturated so that

$$d[y][x] = \begin{cases} 2^{bits\_per\_pixel} - 1; & d'[y][x] > 2^{bits\_per\_pixel} - 1 \\ d'[y][x]; & 0 \le d'[y][x] \le 2^{bits\_per\_pixel} - 1 \\ 0; & d'[y][x] < 0 \end{cases}$$

NOTE The saturation defined above limits output sample values for Y, Cr and Cb to the range [0:$2^{bits\_per\_pixel}$-1]. Therefore, the values which are assigned as the reserved code words for timing reference in ITU-R BT.601 and BT.709 can occasionally occur.

7.16.4 Texture decoding from DCT coefficients

This subclause describes the process used to decode the texture information of a VOP in case of compression_mode==1. The process of video texture decoding from coded DCT coefficients is given in Figure 7-103.

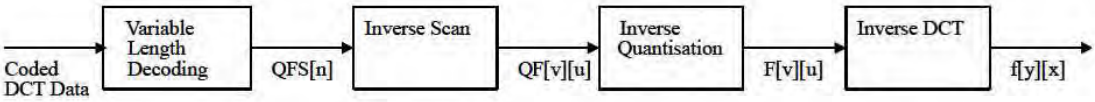


Figure 7-103 — Video Texture Decoding Process from DCT coefficients

7.16.4.1 Variable length decoding

This subclause explains the decoding process. Subclause 7.16.4.1.1 specifies the process used for the DC coefficients (n=0) in an intra coded block. (n is the index of the coefficient in the appropriate zigzag scan order). Subclause 7.16.4.1.2 specifies the decoding process for all other coefficients; AC coefficients ($n \neq 0$) and DC coefficients in non-intra coded blocks.

Let *cc* denote the colour component. It is related to the block number as specified in Table 7-30. In case of rgb_components==0, *cc* is zero for the Y component, one for the C_b component and two for the C_r component. In case of rgb_components==1, zero for the G component, one for the B component and two for the R component.

Table 7-30 — Definition of *cc*, colour component index

	cc		
Block Number	**4:2:0**	**4:2:2**	**4:4:4**
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	1	1	1
5	2	2	2
6		1	1
7		2	2
8			1
9			2
10			1
11			2

© ISO/IEC 2004 – All rights reserved

7.16.4.1.1 DC coefficients decoding in intra blocks

Differential DC coefficients in blocks in intra macroblocks are decoded as a variable length code denoting dct_dc_size as defined in Table B.44 and Table B.45 in Annex B, and a fixed length code dct_dc_differential (Table B.46). The dct_dc_size categorizes the DC coefficients according to their "size". For each category additional bits are appended to the dct_dc_size code to uniquely identify which difference in that category actually occurred (Table B.46). This is done by appending a fixed length code, dct_dc_differential, of dct_dc_size bits.

In case of rgb_components==0, if *cc* is zero then Table B.44 shall be used for dct_dc_size. If *cc* is non-zero then Table B.45 shall be used for dct_dc_size.

In case of rgb_components==1, Table B.44 shall be used for dct_dct_size for all of the components, *cc*.

Three predictors are maintained, one for each of the colour components, *cc*. Each time a DC coefficient in a block in an intra macroblock is decoded the predictor is added to the differential to recover the actual coefficient. Then the predictor shall be set to the value of the coefficient just decoded. At various times, as described below, the predictors shall be reset. The reset value except when a macroblock is encoded as DPCM residuals is derived from the combination of the parameters bits_per_pixel , dct_precision and intra_dc_precision as;

2 ^ (bits_per_pixel+dct_precision+intra_dc_precision-1)

The predictors shall be reset to the above reset value at the following times:

- Whenever a macroblock is decoded if intra_predictors_reset == 1.
- At the start of a slice.
- Whenever a non-intra macroblock is decoded.
- Whenever a macroblock is skipped. i.e. not_coded == 1.
- Whenever a macroblock and a block is transparent.

When a macroblock is encoded as DPCM residuals, the reset value is derived as;

block_mean x (2 ^ (dct_precision+intra_dc_precision))

Because block_mean is encoded for each component independently, the reset value for each component can be different.

The predictors are denoted *dct_dc_pred*[*cc*].

QFS[0] shall be calculated from dct_dc_size and dct_dc_differential by any process equivalent to:

```
if ( dct_dc_size == 0 ) {
    dct_diff = 0;
} else {
    half_range = 2 ^ ( dct_dc_size - 1 );
    if ( dct_dc_differential >= half_range )
        dct_diff = dct_dc_differential;
    else
        dct_diff = (dct_dc_differential + 1) - (2 * half_range);
}
QFS[0] = dct_dc_pred[cc] + dct_diff;
dct_dc_pred[cc] = QFS[0]
```

Annex NA NOTE 1 - The symbol ^ *denotes power (not XOR).*

Annex NB NOTE 2 - *dct_diff* and *half_range* are temporary variables which are not used elsewhere in this specification.

© ISO/IEC 2004 – All rights reserved

It is a requirement of the bitstream that QFS[0] shall lie in the range:

— 0 to ((2^(dct_precision + bits_per_pixel + intra_dc_precision))-1)

7.16.4.1.2 Other coefficients

All coefficients with the exception of the DC intra coefficients shall be decoded using a set of 12 VLC tables, T[0] to T[11] . The default set of the 12 tables is defined in Table B.50 to Table B.61 for intra macroblocks and Table B.62 to Table B.73 for inter macroblocks. The set of tables can be downloaded using the extension syntax of vlc_code_extension().

The decision rule to select one table from the 12 tables for decoding a symbol is defined in this subclause.

In all cases a variable length code shall first be decoded using one of the tables. The decoded value of this code denotes one of three courses of action:

1) End of Block. In this case there are no more coefficients in the block in which case the remainder of the coefficients in the block (those for which no value has yet been decoded) shall be set to zero. This is denoted by "End of block" in the syntax specification of 6.2.13.9.

2) A "normal" coefficient. In which a symbol denoting a value of *run* and/or *level* is decoded followed by a fixed length code of the size indicated by the variable length code according to Table 7-31. The variable length code categorizes symbols according to the value of *run* and/or *level* as indicated in Table 7-31. For each category a fixed length code is appended to the variable length code to uniquely identify which value in that category actually occurred. The additional codes are defined in Table B.47 to Table B.49. For decoding group No.1~6, Table B.47 shall be used. For decoding group No.7~12, Table B.48 shall be used. For decoding group No.13~20, Table B.49 shall be used.

3) An "Escape" coded coefficient. In which a value of *level* is fixed length coded as described in 7.16.4.1.3.

The VLC table which is used to decode a variable length code for the current symbol shall be decided from the 12 tables according to the immediately previous symbol which has already been decoded in a zigzag scanning order. After decoding the current symbol, it is used to decide the VLC table for decoding the next symbol from the 12 tables.

The relation of the value of the current symbol and the VLC table used for decoding a variable length code for the next symbol is indicated in Table 7-31.

The relation of the value of the previous symbol and the VLC table for decoding the current symbol is shown in Table 7-31. A variable length code for the current symbol is decoded using one of the 12 tables, and the variable length code indicates the group (0~21) that the value of the current symbol belongs to. Depending on the group, the size of a fixed length code used to identify the value of the current symbol and the VLC table for decoding the next symbol is decided. For the first non DC coefficient in a block the VLC table T[0] shall be used as the initial table.

© ISO/IEC 2004 – All rights reserved

Table 7-31 — Categorizing AC symbol into groups

group no.	symbols of DCT AC coefficients (zigzag scanning order)	the size of an additional code	VLC table for the next symbol
0	EOB	0	---
1	0*1	0	T[1]
2	0*2 to 0*3	1	T[1]
3	0*4 to 0*7	2	T[1]
4	0*8 to 0*15	3	T[1]
5	0*16 to 0*31	4	T[1]
6	0*32 to 0*63	5	T[1]
7	0*1[1,-1]	1	T[2]
8	0*2[1,-1] to 0*3[1,-1]	2	T[2]
9	0*4[1,-1] to 0*7[1,-1]	3	T[2]
10	0*8[1,-1] to 0*15[1,-1]	4	T[2]
11	0*16[1,-1] to 0*31[1,-1]	5	T[2]
12	0*32[1,-1] to 0*63[1,-1]	6	T[2]
13	-1, or 1	1	T[3]
14	-3 to –2, or 2 to 3	2	T[4]
15	-7 to –4, or 4 to 7	3	T[5]
16	-15 to –8, or 8 to 15	4	T[6]
17	-31 to –16, or 16 to 31	5	T[7]
18	-63 to –32, or 32 to 63	6	T[8]
19	-127 to –64, or 64 to 127	7	T[9]
20	-255 to –128, or 128 to 255	8	T[10]
21	$-(2^{bits\_per\_pixel+dct\_precision+3}-1)$ to –256, or 256 to $(2^{bits\_per\_pixel+dct\_precision+3}-1)$ (escape code)	bits_per_pixel + dct_precision + 4	T[11]

Note : Table T[0] shall be used for the first non DC coefficient in a block.

The VLC code for group '0' of T[0] for inter macroblocks shall not be assigned because there shall be at least one non zero DCT coefficient in an inter block which is indicated as 'coded' by pattern_code[].

The VLC code for group '0'~'13' of T[1] shall not be assigned because there shall be at least one *level* value , the absolute value of which is more than '1', just after group '1'~'6' occurred.

© ISO/IEC 2004 – All rights reserved

7.16.4.1.3 Escape coding

Level values, the absolute value of that is more then 255, are encoded by using the escape code. The escape code that is assigned to group No.21 is followed by a fixed length code of the length 'bits_per_pixel+dct_precision+4'. The following equation defines the relation among the fixed length code, the length and *level* to be decoded.

if flc>>(flclen-1) is 1

level = -1 * ((flc^((1<<flclen)-1)) + 1)

else *level* = flc

Where, flc and flclen denote the fixed length code and its length. The symbol ^ denotes XOR in this subclause.

The symbols that can be coded by using a non escape codeword shall not be coded by using the escape codeword. The codeword with the MSB '1' followed by only 0 binary sequence '100...000' is reserved.

7.16.4.2 Inverse scan

This subclause specifies the way in which the one dimensional data, QFS[n] is converted into a two-dimensional array of coefficients denoted by QF[v][u] where u and v both lie in the range of 0 to 7. Let the data at the output of the variable length decoder be denoted by QFS[n] where n is in the range of 0 to 63.

Two scan patterns are defined. The scan that shall be used shall be determined by alternate_scan which is encoded in StudioVideoObjectPlane().

Figure 7-104 defines *scan*[alternate_scan][*v*][*u*] for the case that alternate_scan is zero. Figure 7-105 defines *scan*[alternate_scan][*v*][*u*] for the case that alternate_scan is one.

		u							
		0	1	2	3	4	5	6	7
	0	0	1	5	6	14	15	27	28
	1	2	4	7	13	16	26	29	42
	2	3	8	12	17	25	30	41	43
	3	9	11	18	24	31	40	44	53
	4	10	19	23	32	39	45	52	54
	5	20	22	33	38	46	51	55	60
	6	21	34	37	47	50	56	59	61
v	7	35	36	48	49	57	58	62	63

Figure 7-104 — Definition of *scan*[0][*v*][*u*]

© ISO/IEC 2004 – All rights reserved

		u						
		0	1	2	3	4	5	6	7
v	0	0	4	6	20	22	36	38	52
	1	1	5	7	21	23	37	39	53
	2	2	8	19	24	34	40	50	54
	3	3	9	18	25	35	41	51	55
	4	10	17	26	30	42	46	56	60
	5	11	16	27	31	43	47	57	61
	6	12	15	28	32	44	48	58	62
	7	13	14	29	33	45	49	59	63

Figure 7-105 — Definition of *scan*[1][*v*][*u*]

The inverse scan shall be any process equivalent to the following:

```
for (v=0; v<8; v++)
    for (u=0; u<8; u++)
        QF[v][u] = QFS[scan[alternate_scan][v][u]]
```

NOTE - The scan patterns defined here are often referred to as "zigzag scanning order".

7.16.4.2.1 Inverse scan for matrix download

When the quantisation matrices are downloaded they are encoded in the bitstream in a scan order that is converted into the two-dimensional matrix used in the inverse quantisation process in an identical manner to that used for coefficients.

For matrix download the scan defined by Figure 7-104 (i.e. *scan*[0][*v*][*u*]) shall always be used.

Let *W*[*w*][*v*][*u*] denote the quantisation matrix in the inverse quantisation process (see 7.16.4.3.2.1), and *W'*[*w*][*n*] denote the matrix as it is encoded in the bitstream. The matrix download shall then be equivalent to the following:

```
for (v=0; v<8; v++)
    for (u=0; u<8; u++)
        W[w][v][u] = W'[w][scan[0][v][u]]
```

7.16.4.3 Inverse quantisation

The two-dimensional array of coefficients, *QF*[*v*][*u*], is inverse quantised to produce the reconstructed DCT coefficients. This process is a multiplication by the base quantiser scale and the quantisation step size. The quantisation step size is modified by two mechanisms; a quantisation matrix is used to modify the step size within a block and a quantiser scale is used in order that the step size can be modified at the cost of only a few bits (as compared to encoding an entire new quantisation matrix). In case of mpeg2_stream==0, the base quantiser scale is used to adjust the dynamic range of the output of the inverse quantisation to the range $[-2^{bits\_per\_pixel + 6}, 2^{bits\_per\_pixel + 6} - 1]$. In case of mpeg2_stream==1, the base quantiser scale shall not affect the decoding process below.

© ISO/IEC 2004 – All rights reserved

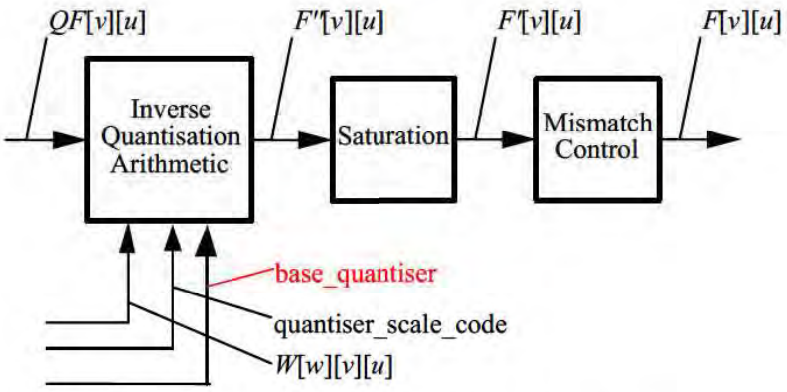


Figure 7-106 — Inverse quantisation process

Figure 7-106 illustrates the overall inverse quantisation process. After the appropriate inverse quantisation arithmetic the resulting coefficients, *F''*[*v*][*u*], are saturated to yield *F'*[*v*][*u*] and then a mismatch control operation is performed to give the final reconstructed DCT coefficients, *F*[*v*][*u*].

NOTE Attention is drawn to the fact that the method of achieving mismatch control in this part of ISO/IEC 14496 is identical to that employed by ISO/IEC 13818-2:2000.

7.16.4.3.1 Intra DC coefficient

The DC coefficients of intra coded blocks shall be inverse quantised in a different manner to all other coefficients.

In intra blocks *F'*[0][0] shall be obtained by multiplying *QF*[0][0] by a constant multiplier, *intra_dc_mult*, (constant in the sense that it is not modified by either the quantisation matrix or the quantiser scale). The multiplier is related to the parameters base_quantiser scale and intra_dc_precision that are encoded in StudioVideoObjectPlane(). Table 7-32 and Table 7-33 specify the relation between intra_dc_precision, base_quantiser scale and *intra_dc_mult*. This relation changes according to the value of mpeg2_stream. The precision of the intra dc coefficient is defined as 'bits_per_pixel+dct_precision+intra_dc_precision'.

Table 7-32 — In case of mpeg2_stream==0:
Reletion among intra_dc_precision, base_quantiser scaleand *intra_dc_mult*

intra_dc_precision	*intra_dc_mult*
0	8 x base_quantiser scale x 8
1	4 x base_quantiser scale x 8
2	2 x base_quantiser scale x 8
3	1 x base_quantiser scale x 8

Table 7-33 — In case of mpeg2_stream==1:
Reletion between intra_dc_precision, *intra_dc_mult*

intra_dc_precision	*intra_dc_mult*
0	8
1	4
2	2
3	1

© ISO/IEC 2004 – All rights reserved

The reconstructed DC values are computed as follows.

F''[0][0] = intra_dc_mult* QF[0][0]

7.16.4.3.2 Other coefficients

All coefficients other than the DC coefficient of an intra block shall be inverse quantised as specified in this subclause.

7.16.4.3.2.1 Quantisation matrices

In the 4:2:0 format, two quantisation matrices are used. One shall be used for intra macroblocks and the other for non-intra macroblocks. In the 4:2:2 or 4:4:4 format, four matrices are used allowing different matrices to be used for luminance and chrominance data. Each matrix has a default set of values which may be overwritten by down-loading a user defined matrix as explained in 6.2.13.2.4.

Let the quantisation matrices be denoted by *W*[*w*][*v*][*u*] where *w* takes the values 0 to 3 indicating which of the matrices is being used. Table 7-34 summarises the rules governing the selection of *w*.

Table 7-34 — Selection of *w*

	4:2:0		4:2:2 and 4:4:4	
	luminance (cc = 0)	**chrominance (cc ≠ 0)**	**luminance (cc = 0)**	**chrominance (cc ≠ 0)**
intra blocks (macroblock_intra = 1)	0	0	0	2
non-intra blocks (macroblock_intra = 0)	1	1	1	3

7.16.4.3.2.2 Quantiser scale

The quantiser scale is decoded as a 5 bit fixed length code, quantiser_scale_code. This indicates the appropriate *quantiser_scale* to apply in the inverse quantisation arithmetic.

q_scale_type (encoded in StudioVideoObjectPlane()) indicates which of two mappings between quantiser_scale_code and *quantiser_scale* shall apply. Table 7-35 shows the two mappings between quantiser_scale_code and *quantiser_scale*.

© ISO/IEC 2004 – All rights reserved

Table 7-35 — Relation between *quantiser_scale* and quantiser_scale_code

quantiser_scale_code	quantiser_scale[q_scale_type]	
	q_scale_type = 0	q_scale_type = 1
0	(forbidden)	
1	2	1
2	4	2
3	6	3
4	8	4
5	10	5
6	12	6
7	14	7
8	16	8
9	18	10
10	20	12
11	22	14
12	24	16
13	26	18
14	28	20
15	30	22
16	32	24
17	34	28
18	36	32
19	38	36
20	40	40
21	42	44
22	44	48
23	46	52
24	48	56
25	50	64
26	52	72
27	54	80
28	56	88
29	58	96
30	60	104
31	62	112

7.16.4.3.2.3 Reconstruction formulae

The following equation specifies the arithmetic to reconstruct *F''*[*v*][*u*] from *QF*[*v*][*u*] (for all coefficients except intra DC coefficients) in the case of mpeg2_stream==0.

$$F''[v][u] = ((2 \times QF[v][u] + k) \times W[w][v][u] \times quantiser\_scale \times base\_quantiser \times 8) / 32$$

where:

$$k = \begin{cases} 0 & \text{intra blocks / non-intra blocks if dead\_zone\_disable==1} \\ Sign(QF[v][u]) & \text{non-intra blocks if dead\_zone\_disable==0} \end{cases}$$

NOTE The above equation uses the "/" operator as defined in 4.1.

© ISO/IEC 2004 – All rights reserved

This equation changes in the case of mpeg2_stream==1 as following:

$$F''[v][u] = ((2 \times QF[v][u] + k) \times W[w][v][u] \times quantiser\_scale) / 32$$

where:

$$k = \begin{cases} 0 & \text{intra blocks} \\ Sign(QF[v][u]) & \text{non-intra blocks} \end{cases}$$

NOTE The above equation uses the "/" operator as defined in 4.1.

7.16.4.3.3 Saturation

The coefficients resulting from the Inverse Quantisation Arithmetic are saturated to lie in the range $[-2^{bits\_per\_pixel + 6}, 2^{bits\_per\_pixel + 6} - 1]$ in the case of mpeg2_stream==0. Thus:

$$F'[v][u] = \begin{cases} 2^{bits\_per\_pixel+6} - 1 & F''[v][u] > 2^{bits\_per\_pixel+6} - 1 \\ F''[v][u] & -2^{bits\_per\_pixel+6} \le F''[v][u] \le 2^{bits\_per\_pixel+6} - 1 \\ -2^{bits\_per\_pixel+6} & F''[v][u] < -2^{bits\_per\_pixel+6} \end{cases}$$

The coefficients resulting from the Inverse Quantisation Arithmetic are saturated to lie in the range $[-2^{bits\_per\_pixel + 3}, 2^{bits\_per\_pixel + 3} - 1]$ in the case of mpeg2_stream==1. Thus:

$$F'[v][u] = \begin{cases} 2^{bits\_per\_pixel+3} - 1 & F''[v][u] > 2^{bits\_per\_pixel+3} - 1 \\ F''[v][u] & -2^{bits\_per\_pixel+3} \le F''[v][u] \le 2^{bits\_per\_pixel+3} - 1 \\ -2^{bits\_per\_pixel+3} & F''[v][u] < -2^{bits\_per\_pixel+3} \end{cases}$$

7.16.4.3.4 Mismatch control

Mismatch control shall be performed by any process equivalent to the following. Firstly all of the reconstructed, saturated coefficients, *F'[v][u]* in the block shall be summed. This value is then tested to determine whether it is odd or even. If the sum is even then a correction shall be made to just one coefficient; *F*[7][7]. Thus:

$$sum = \sum_{v=0}^{v<8} \sum_{u=0}^{u<8} F'[v][u]$$

$$F[v][u] = F'[v][u] \text{ for all } u, v \text{ except } u = v = 7$$

$$F[7][7] = \begin{cases} F'[7][7] & \text{if } sum \text{ is odd} \\ \begin{cases} F'[7][7] - 1 & \text{if } F'[7][7] \text{ is odd} \\ F'[7][7] + 1 & \text{if } F'[7][7] \text{ is even} \end{cases} & \text{if } sum \text{ is even} \end{cases}$$

NOTE 1 It may be useful to note that the above correction for *F*[7][7] may simply be implemented by toggling the least significant bit of the twos complement representation of the coefficient. Also since only the "oddness" or "evenness" of the *sum* is of interest an exclusive OR (of just the least significant bit) may be used to calculate "*sum*".

NOTE 2 Warning. Small non-zero inputs to the IDCT may result in zero output for compliant IDCTs. If this occurs in an encoder, mismatch may occur in some pictures in a decoder that uses a different compliant IDCT. An encoder should avoid this problem and may do so by checking the output of its own IDCT. It should ensure that it never inserts any non-zero coefficients into the bitstream when the block in question reconstructs to zero through its own IDCT function. If this action is not taken by the encoder, situations can arise where large and very visible mismatches between the state of the encoder and decoder occur.

© ISO/IEC 2004 – All rights reserved

7.16.4.3.5 Summary of inverse quantisation process

In summary, the inverse quantisation process is any process numerically equivalent to:

```
for (v=0; v<8;v++) {
    for (u=0; u<8;u++) {
        if ( (u==0) && (v==0) && (macroblock_intra) ) {
            F'[v][u] = intra_dc_mult * QF[v][u];
        } else {
            if ( macroblock_intra ) {
                F'[v][u] = ( QF[v][u] * W[0][v][u] * quantiser_scale * base_quantiser scale * 8 * 2 ) / 32;
            } else {
                F'[v][u] = ( ( ( QF[v][u] *  2 ) + Sign(QF[v][u]) ) * W[1][v][u]
                                                    * quantiser_scale * base_quantiser scale * 8 ) / 32;
            }
        }
    }
}
sum = 0;
for (v=0; v<8;v++) {
    for (u=0; u<8;u++) {
        if ( F'[v][u] > 2^(bits_per_pixel + 6) - 1 ) {
            F[v][u] = 2^(bits_per_pixel + 6) - 1;
        } else {
            if ( F'[v][u] < -2^(bits_per_pixel + 6) ) {
                F[v][u] = -2^(bits_per_pixel + 6);
            } else {
                F[v][u] = F'[v][u];
            }
        }
    sum = sum + F[v][u];
    F[v][u] = F[v][u];
    }
}
if ((sum & 1) == 0) {
    if ((F[7][7] & 1) != 0) {
        F[7][7] = F[7][7] - 1;
    } else {
        F[7][7] = F[7][7] + 1;
    }
}
```

7.16.4.4 Inverse DCT

Once the DCT coefficients, F[u][v] are reconstructed, an IDCT transform that conforms to the specifications of Annex A shall be applied to obtain the inverse transformed values, $f[y][x]$. In the case of mpeg2_stream==0, the decimal point of F[u][v] is shifted 3bits to the left in the binary scale in order to adjust the decimal point of the IDCT input. In the case of mpeg2_stream==1, the reconstructed coefficients are directly input to an IDCT function without the shift process. The inverse transformed values shall be saturated so that: $-2^{bits\_per\_pixel} \le f[y][x] \le 2^{bits\_per\_pixel} - 1$, for all x, y.

7.16.4.4.1 Non-coded blocks and skipped macroblocks

In a macroblock that is not skipped, if pattern_code[i] is one for a given block in the macroblock, then coefficient data is included in the bitstream for that block. This is decoded as specified in the preceding clauses.

However, if pattern_code[i] is zero, or if the macroblock is skipped (not_coded==1), then that block contains no coefficient data. The sample domain coefficients f[y][x] for such a block shall all take the value zero.

© ISO/IEC 2004 – All rights reserved

7.16.5 Texture decoding from DPCM redisuals

If the DPCM mode is selected each non-transparent block is decoded as follows. First the **rice_parameter** field is decoded. It indicates the length of each subsequent **rice_suffix_code** field. The block is scanned line by line, row by row, and the following process is applied to each pixel. An unsigned residual is retrieved by a combination of the **rice_prefix_code**, **rice_suffix_code** and **dpcm_residual** fields. If **rice_prefix_code** indicates the escape mode, the unsigned residual is given by **dpcm_residual**. Otherwise the unsigned residual is given by (**rice_prefix_code**<<**rice_parameter**)+**rice_suffix_code**. The residuals that can be coded by using a combination of rice_prefix_code and rice_suffix_code shall not be coded by using dpcm_residual.

Then the unsigned residual is mapped to a signed residual according to the following transformation:

$x \rightarrow x>>1$ if x is even

$x \rightarrow -x>>1$ if x is odd

A prediction is computed based on the pixels that lie directly to the left 'a', directly above 'b', and directly left above 'c'. If any of these pixels lies outside of the block boundary it is given the default value of $2^{bits\_per\_pixel-1}$. The prediction 'p' is computed as:

p = a+b-c

if (p < min(a,b)) p = min(a,b)

if (p > max(a,b)) p = max(a,b)

A second prediction p2 is computed as:

p2 = (min(a,b,c)+max(a,b,c)) / 2

If p2 is equal to p, p2 is assigned the value of **block_mean**.

If p2 is larger than p, the sign of the residual is inverted. Otherwise no operation is applied to the residual.

Finally the reconstructed pixel value, d''[y][x], is obtained by adding the signed residual and the prediction p, modulo $2^{bits\_per\_pixel}$.

7.16.6 Shape decoding

Binary shape decoding uses a block-based representation. The primary data structure used is denoted as the binary alpha block (bab). The bab is a square block of binary valued pixels representing the opacity/transparency for the pixels in a specified block-shaped spatial region of size 16x16 pels. In fact, each bab is co-located with each texture macroblock.

7.16.6.1 Higher syntactic structures

7.16.6.1.1 VOL decoding

If video_object_layer_shape is equal to '00' then no binary shape decoding is required. Otherwise, binary shape decoding is carried out.

7.16.6.1.2 VOP decoding

If video_object_layer_shape is not equal to '00' then, for each subsequent VOP, the dimensions of the bounding rectangle of the reconstructed VOP are obtained from:

- vop_width
- vop_height

© ISO/IEC 2004 – All rights reserved

If these decoded dimensions are not multiples of 16, then the values of vop_width and vop_height are rounded up to the nearest integer, which is a multiple of 16. If vop_structure is decoded as field structure, both values are definded in the absolute field coordinates.

Additionally, in order to facilitate motion compensation, the horizontal and spatial position of the VOP are obtained from:

- vop_horizontal_mc_spatial_ref
- vop_vertical_mc_spatial_ref

These spatial references may be different for each VOP. The absolute frame coordinate system must be used for all frame VOPs, while the absolute field coordinate system must be used for all field VOPs. Additionally, the decoded spatial references must have an even value.

Once the above elements have been decoded, the binary shape decoder may be applied to decode the shape of each macroblock within the bounding rectangle.

7.16.6.2 Macroblock decoding

The shape information for each macroblock residing within the bounding rectangle of the VOP is decoded into the form of a 16x16 bab.

7.16.6.2.1 Mode decoding

Each bab belongs to one of five types listed in Table 7-36. The type information is given by the bab_type field which influences decoding of further shape information.

Table 7-36 — List of bab types

bab_type	Semantic	Used in
0	MVs==0 && No Update	P-VOPs
1	MVs!=0 && No Update	P-VOPs
2	Transparent	All VOP types
3	Opaque	All VOP types
4	HHC	All VOP types

7.16.6.2.1.1 I-VOPs

In this specification (Studio Profile), only five bab_types are adopted independently of vop_type.

Suppose that f(x,y) is the bab_type of the bab located at (x,y), where x is the BAB column number and y is the BAB row number. The code word for the bab_type at the position (i,j) is determined as follows. A context C is computed from a previously decoded bab_type.

C = f(i-1,j)-2

If `f(x,y)` references a bab outside the current VOP, bab_type is assumed to be transparent for that bab (i.e. `f(x,y)=2`). bab_type of the bab outside the current slice is also assumed to be transparent. The VLC used to decode bab_type for the current bab is switched according to the value of the context C. This context-switched VLC table is given in Table B.75. The context C shall be set to zero for all macroblocks in a VOP in case of intra_predictors_reset==1 or for a macroblock located at the start of a slice.

If the type of the bab is transparent, then the current bab is filled with zero (transparent) values. A similar procedure is carried out if the type is opaque, where the reconstructed bab is filled with values of 255 (opaque). For both transparent and opaque types, no further decoding of shape-related data is required for the current bab. Otherwise further decoding steps are necessary, which is called HHC. Decoding for HHC is described in subclause 7.16.6.2.5.

© ISO/IEC 2004 – All rights reserved

7.16.6.2.1.2 P-VOPs

The decoding of the current bab_type is dependent on the bab_type of the co-located bab in the reference VOP. The reference VOP is a forward reference VOP. The forward reference VOP is defined as the most recent non-empty (i.e. vop_coded != 0) I- or P-VOP in the past. If the current VOP is a P-VOP, the forward reference VOP is selected as the reference VOP.

If the current VOP is a field VOP of the interlaced sequence, the forward reference VOP is defined as the most recent non empty I- or P- field VOP of the same position (top or bottom field) in the past. When the second field of the first frame in GOV is coded as P-VOP, the reference VOP is the first field coded as I-VOP.

If the sizes of the current and reference VOPs are different, some babs in the current VOP may not have a co-located equivalent in the reference VOP. Therefore the bab_type matrix of the reference VOP is manipulated to match the size of the current VOP. Two rules are defined for that purpose, namely a cut rule and a copy rule:

- ***cut rule***. If the number of lines (respectively columns) is smaller in the current VOP than in the reference VOP, the bottom lines (respectively rightmost columns) are eliminated from the reference VOP such that both VOP sizes match.
- ***copy rule***. If the number of lines (respectively columns) is larger in the current VOP than in the reference VOP, the bottom line (respectively rightmost column) is replicated as many times as needed in the reference VOP such that both VOP sizes match.

An example is shown in Figure 7-107 where both rules are applied.

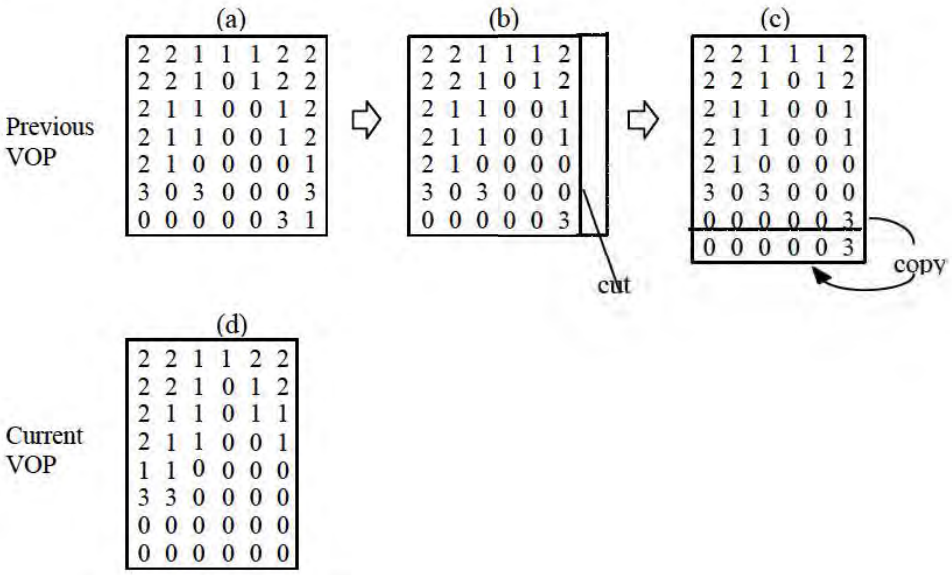


Figure 7-107 — Example of size fitting between current VOP and reference VOP. The numbers represent the type of each bab

The VLC to decode the current bab_type is switched according to the bab_type value of the co-located bab in the reference VOP. These context-switched VLC tables for P-VOPs are given in Table B.76. If the type of the bab is transparent, then the current bab is filled with zero (transparent) values. A similar procedure is carried out if the type is opaque, where the reconstructed bab is filled with values of 255 (opaque). For both transparent and opaque types, no further decoding of shape-related data is required for the current bab. Otherwise further decoding steps are necessary, as listed in Table 7-37. Decoding for motion compensation is described in subclause 7.16.6.2.4, and HHC decoding in subclause 7.16.6.2.5.

© ISO/IEC 2004 – All rights reserved

Table 7-37 — Decoder components applied for each type of bab

bab_type	Motion compensation	HHC
0	yes	no
1	yes	no
2	no	no
3	no	no
4	no	yes

7.16.6.2.2 Binary alpha block motion compensation

Motion Vector of a shape (MVs) is used for motion compensation (MC) of the shape. The value of MVs is reconstructed as described in subclause 7.16.6.2.3. Integer pixel motion compensation is carried out on a 16x16 block basis according to subclause 7.16.6.2.4.

If bab_type is MVs==0 && No Update or MVs!=0 && No Update then the motion compensated bab is taken to be the decoded bab, and no further decoding of the bab is necessary. Otherwise, HHC decoding is required.

7.16.6.2.3 Motion vector decoding

If bab_type indicates that MVs!=0, then mvs_x and mvs_y are VLC decoded. For decoding mvs_x, the VLC given in Table B.77 is used. The same table is used for decoding mvs_y, unless the decoded value of mvs_x is zero. If mvs_x == 0, the VLC given in Table B.78 is used for decoding mvs_y. If bab_type indicates that MVs==0, then both mvs_x and mvs_y are set to zero.

7.16.6.2.4 Motion compensation

For inter mode babs (bab_type = 0 or 1), motion compensation is carried out by simple MV displacement according to the MVs.

Specifically, when bab_type is equal to 0 or 1 i.e. for the no-update modes, a displaced block of 16x16 pixels is copied from the binary alpha map of the previously decoded I or P VOP for which vop_coded is not equal to '0'. If the displaced position is outside the bounding rectangle, then these pixels are assumed to be "transparent".

If the macroblock is in the field VOP, a displayed block is copied from previously decoded field I-VOP of the same position (top or bottom field). When the second field of the first frame in GOV is coded as P-VOP, a displayed block is copied from the first field coded as I-VOP.

7.16.6.2.5 HHC decoding

If an inferior_symbol_macroblock is equal to '0' then the inferior symbol means transparent and the superior symbol means opaque. Otherwise, the inferior symbol means opaque and the superior symbol means transparent. After the inferior symbol is decided, the pattern of inferior symbol in the block is decoded as the following hierarchical process.

7.16.6.2.5.1 Coded binary block pattern

The index of cbbp is obtained by decoding the variable length code to produce coded_binary_block_pattern[i] (i=0,1,2,3). If coded_binary_block_pattern[i] is one for a given block in the macroblock then the data of the block layer is included in the bitstream for that block. However, if coded_binary_block_pattern[i] is zero, then that block contains no inferior symbol for the macroblock. The binary alpha coefficients for such shape blocks shall all take the value equal to the superior symbol for the macroblock.

© ISO/IEC 2004 – All rights reserved

7.16.6.2.5.2 Binary block decoding

For the block whose coded_binary_block_pattern[I] is one, the data of the block layer is decoded. The inferior and the superior symbol for the block is indicated by inferior_symbol_block. If scan_direction is equal to 0, the block is divided into horizontal rows. If scan_direction is equal to 1, the block is divided into vertical columns. If backward_read_flag is set to '0', lbp corresponds to the pixel pattern from left to right pixel in scan_direction='0' or from upper to lower pixel in scan_direction='1'. If backward_read_flag is set to '1', lbp corresponds to the pixel pattern from right to left pixel in scan_type='0' or from lower to upper pixel in scan_type='1'.

The index of clp is obtained by decoding the variable length code to produce coded_line_pattern[i] (I=0,1,2, ,7). If coded_line_pattern[i] is one for a given line in the block then lbp information is included in the bitstream for that line. However, if coded_line_pattern[i] is zero, then that line contains no inferior symbol for the block. The binary alpha coefficient for such shape lines shall all take the value equal to the superior symbol for the block.

The index of lbp is obtained by decoding the variable length code to produce line_bit_pattern[i] (I=0,1,2, ,7). If backward_read_flag is set to '1', line_bit_pattern[i] is set to line_bit_pattern[7-i]. Note that line_bit_pattern[i] equal to '1' means the pixel is the inferior symbol for the block. Otherwise, line_bit_pattern[i] is equal to '1' means the pixel is the superior symbol for the block.

After all pixels are decoded, the inferior symbol and superior symbol are set to either opaque or transparent symbol, respectively.

7.16.6.3 Grayscale Shape Decoding

Grayscale alpha plane decoding is achieved by the separate decoding of a support region and the values of the alpha channel. The support region is transmitted by using the binary shape as described above. The alpha values are transmitted as texture data with arbitrary shape, using almost the same coding method as is used for the luminance texture channel.

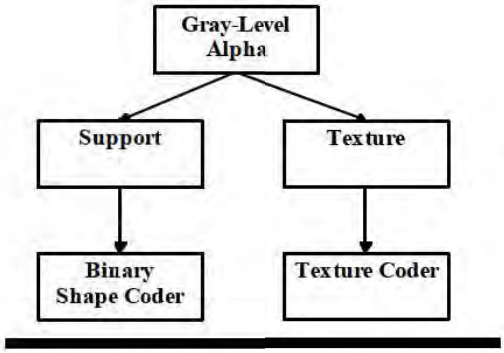


Figure 7-108 — Grayscale shape coding

All samples which are indicated to be transparent by the binary shape data must be set to the value of minimum_alpha_level[i](transparent) in the decoded grayscale alpha plane i. Within the VOP, alpha samples have the values produced by the grayscale alpha decoding process. Decoding of binary shape information is not dependent on the decoding of grayscale alpha. The alpha values are decoded into 16x16 macroblocks in the same way as the luminance channel (see subclause 7.16.4, 7.16.5 and 7.16.7). The 16x16 blocks of alpha values are referred to as alpha macroblocks hereafter. The data for each alpha macroblock is present in the bitstream immediately following the texture data for the corresponding texture macroblock. Any aspect of alpha decoding that is not covered in this document should be assumed to be the same as for the decoding of luminance.

© ISO/IEC 2004 – All rights reserved

7.16.6.3.1 Grayscale Alpha COD Modes

When decoding grayscale alpha macroblocks, CODA is first encountered and indicates the coding status for alpha. It is important to understand that the macroblock syntax elements for alpha are still present in the bitstream for inter macroblocks even if the texture syntax elements indicate “not-coded” (not_coded=’1’). In this respect, the decoding of the alpha and texture data are independent.

For macroblocks which are completely transparent (indicated by the binary shape coding), no alpha syntax elements are present and the grayscale alpha samples must all be set to the value of minimum_alpha_level[i] (transparent). If CODA=”all opaque” (intra, inter macroclocks) or CODA = "residual all zero" (inter macroblocks) then no more alpha data is present. Otherwise, other alpha syntax elements follow, i.e. alpha texture data encoded as either DCT coefficients which are coded and non-transparent or DPCM residuals, as is the case for regular luminance macroblock texture data.

When CODA=”all opaque”, the corresponding decoded alpha macroblock is filled with the value of maximum_alpha_level[i]. This value will be called AlphaOpaqueValue[i].

7.16.6.3.2 Intra Macroblocks coded as DPCM residuals

In case of alpha_compression_mode[i]==1(DPCM mode), the grayscale alpha data shall be decoded as DPCM residuals. DPCM mode can be selected independently of intra/inter mode of the texture data.

7.16.6.3.3 Intra Macroblocks coded as DCT coefficients

When the texture is encoded as DPCM residuals (compression_mode==1) or intra DCT coefficients (compression_mode==0 && macroblock_intra==1), and if alpha_compression_mode[i]==0, the grayscale alpha data is encoded as intra DCT residual.

The intra DC value is decoded in the same way as for luminance.

The DC predictor is used in the same way as for luminance. However, when coda_i indicates that a macroblock is all opaque, the predictor shall be reset so that the next intra block to be decoded is correctly decoded. The reset value is defined as;

AlphaOpaqueValue[i] * (2^(alpah_dct_precision[i]+alpha_intra_dc_precision[i]))

AlphaOpaqueValue[i] is described in subclause 7.16.6.3.1.

7.16.6.3.4 Inter Macroblocks and Motion Compensation

When the texture data in a P-VOP is encoded as inter macroblock (compression_mode==0 && macroblock_intra==0), and if alpha_compression_mode[i]==0, the alpha macroblock shall be decoded as inter macroblock.

Motion compensation is carried out for inter macroblocks, using the 16x16 or 16x8 luminance motion vectors, in the same way as for luminance data. Where the luminance motion vectors are not present because the texture macroblock is skipped, the exact same style of non-coded motion compensation used for luminance is applied to the alpha data. Note that this does not imply that the alpha macroblock is skipped, because an error signal to update the resulting motion compensated alpha macroblock may still be present if indicated by coda_pb.

cbpa is defined in Table B.83 and alpha_pattern_code is derived from cbps.

Alpha inter DCT coefficients are decoded in the same way as the luminance coefficients. The only exception is that the inverse quantisation process in 7.16.4.2.3 shall perfom as if dead_zone_disable is set to zero independently of the value of dead_zone_disable.

© ISO/IEC 2004 – All rights reserved

7.16.6.3.5 Method to be used when blending with greyscale alpha signal

The following explains the blending method to be applied to the video object in the compositor, which is controlled by the composition_method flag and the linear_composition flag. The linear_composition flag is informative only, and the decoder may ignore it and proceed as if it had the value 0. However, it is normative that the composition_method flag be acted upon.

The descriptions below show the processing taking place in YUV space; note that the processing can of course be implemented in RGB space to obtain equivalent results.

saturation

The alpha signals are saturated to lie in the range [Tr, Op]. Thus:

```
        { Op      if alpha > Op
alpha = { alpha         if  Tr <= alpha <= Op
        { Tr      if alpha < Tr
```

composition_method=0 (cross-fading)

If layer N, with an n-bit alpha signal, is overlaid over layer M to generate a new layer P, the composited Y, U, V and alpha values are:

```
Pyuv   =      ( (R - (Nalpha-Tr)) * Myuv + ((Nalpha-Tr) * Nyuv ) ) / R
Palpha =      Op
```

composition_method=1 (Additive mixing)

If layer N, with an n-bit alpha signal, is overlaid over layer M to generate a new layer P, the composited Y, U, V and alpha values are:

```
     { Myuv                                  ..... Nalpha = Tr
Pyuv  = {
     { (Myuv - BLACK)  -  ( (Myuv - BLACK) * (Nalpha-Tr) ) / R+ Nyuv      ..... Nalpha > Tr
```

(this is equivalent to Pyuv = Myuv*(1-alpha) + Nyuv, taking account of black level and the fact that the video decoder does not produce an output in areas where alpha=0)

```
Palpha =      Nalpha + Malpha -Tr - ((Nalpha-Tr)*(Malpha-Tr)) / R
```

where

BLACK is the common black value of foreground and background objects.

Tr: level for complete transparency

Op: level for complete opacity

R: = Op - Tr (dynamic range of alpha signal)

AnnexNC NOTE The compositor must convert foreground and background objects to the same black value and signal range before composition. The black level of each video object is specified by the video_range bit in the video_signal_type field, or by the default value if the field is not present. (The RGB values of synthetic objects are specified in a range from 0 to 1, as described in ISO/IEC 14496-1).

linear_composition = 0: The compositing process is carried out using the video signal in the format from which it is produced by the video decoder, that is, without converting to linear signals. Note that because video signals are usually non-linear ("gamma-corrected"), the composition will be approximate.

© ISO/IEC 2004 – All rights reserved

linear_composition = 1: The compositing process is carried out using linear signals, so the output of the video decoder is converted to linear if it was originally in a non-linear form, as specified by the video_signal_type field. Note that the alpha signal is always linear, and therefore requires no conversion.

7.16.6.4 Multiple Auxiliary Component Decoding

Auxiliary components are defined for the VOP on a pixel-by-pixel basis, and contain data related to the video object, like disparity, depth, additional texture. Up to 3 auxiliary components (including the grayscale shape) are possible. The number and type of these components is indicated by the video_object_layer_shape_extension given in Table 6-84. For example, a value '0000' indicates the grayscale (alpha) shape. The same support region as described in 7.16.6.2 is used for all auxiliary components, and the decoding procedure is the same as described in 7.16.6.3.

7.16.7 Motion compensation decoding

The motion compensation process forms predictions from previously decoded VOPs which are combined with the coefficient data (from the output of the IDCT) in order to recover the final decoded samples.

Up to two separate predictions are formed for each block which are combined together to form the final prediction block p[y][x].

In the case of intra coded macroblocks no prediction is formed so that p[y][x] will be zero. The saturation shown in Figure 7-102 is still required in order to remove prohibited values from f[y][x].

In the case where a block is not coded, either because the entire macroblock is skipped or the specific block is not coded there is no coefficient data. In this case f[y][x] is zero and the decoded samples are the prediction, p[y][x].

All the processes except for resetting motion vector predictors in 7.16.7.4.4 are never executed when a macroblock is completely transparent.

© ISO/IEC 2004 – All rights reserved

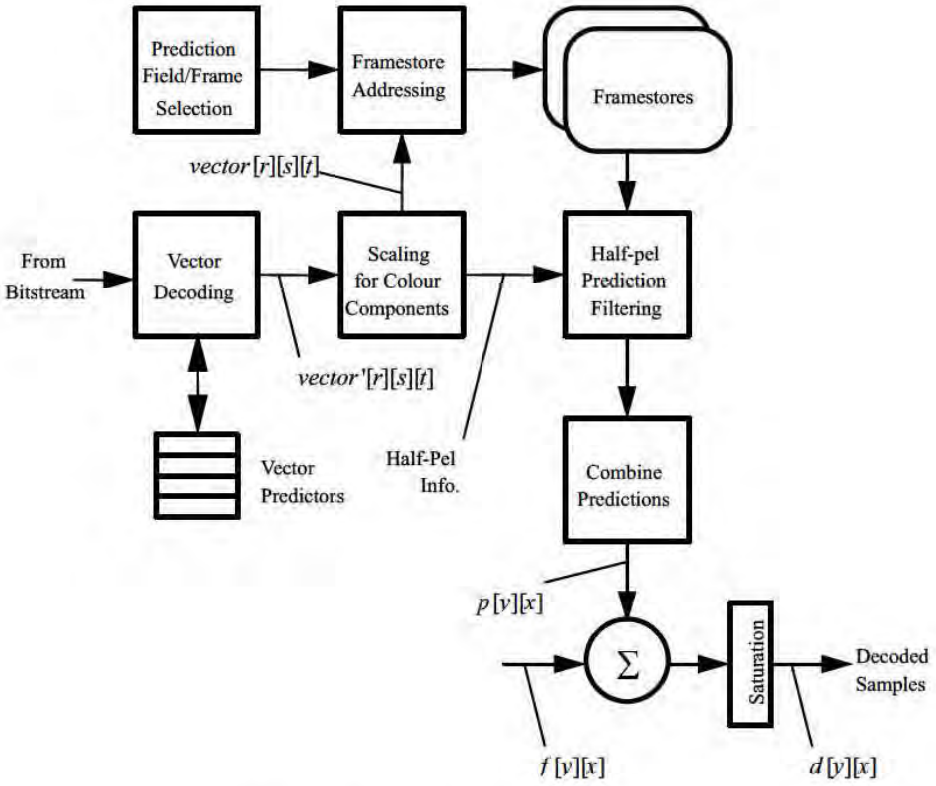


Figure 7-109 — Simplified motion compensation process

7.16.7.1 Motion compensation decoding of arbitrary shaped VOP

In order to perform motion compensated prediction on a per VOP basis, a special padding technique, i.e. the macroblock-based repetitive padding, is applied for the reference VOP. The details of these techniques are described in the following subclauses.

Since a VOP may have arbitrary shape, and this shape can change from one instance to another, conventions are necessary to ensure the consistency of the motion compensation process.

The absolute (frame or field) coordinate system is used for referencing every VOP. At every given instance, a bounding rectangle that includes the shape of that VOP, as described in subclause 7.16.4, is defined. The left and top corner, in the absolute coordinates, of the bounding rectangle is decoded from VOP spatial reference. Thus, the motion vector for a particular feature inside a VOP, e.g. a macroblock, refers to the displacement of the feature in absolute coordinates. No alignment of VOP bounding rectangles at different time instances is performed.

7.16.7.1.1 Padding process

The padding process defines the values of luminance and chrominance samples outside the VOP for prediction of arbitrarily shaped objects. Figure 7-110 shows a simplified diagram of this process.

© ISO/IEC 2004 – All rights reserved

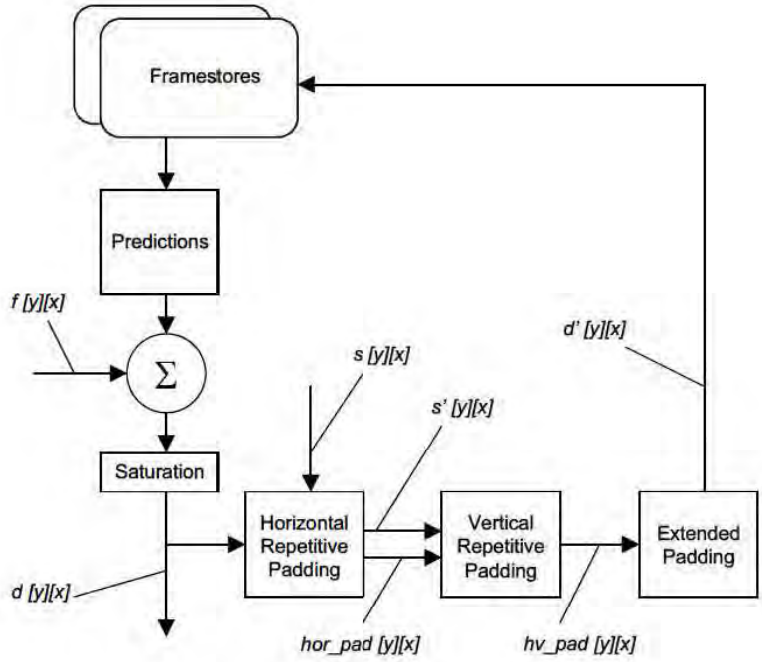


Figure 7-110 — Simplified padding process

A decoded macroblock *d[y][x]* is padded by referring to the corresponding decoded shape block *s[y][x]*. The luminance component is padded per 16 x 16 samples, while the chrominance components are padded per 8 x 8 samples for 4:2:0 format, 16 x 8 samples for 4:2:2 format, 16 x 16 samples for 4:4:4 format. A macroblock that lies on the VOP boundary (hereafter referred to as a boundary macroblock) is padded by replicating the boundary samples of the VOP towards the exterior. This process is divided into horizontal repetitive padding and vertical repetitive padding. The remaining macroblocks that are completely outside the VOP (hereafter referred to as exterior macroblocks) are filled by extended padding.

NOTE The padding process is applied to all macroblocks inside the bounding rectangle of a VOP. The bounding rectangle of the luminance component is defined by vop_width and vop_height extended to a multiple of 16, while that of the chrominance components is defined by (vop_width>>1) and (vop_height>>1) extended to multiple of 8.

7.16.7.1.1.1 Horizontal repetitive padding

Each sample at the boundary of a VOP is replicated horizontally to the left and/or right direction in order to fill the transparent region outside the VOP of a boundary macroblock. If there are two boundary sample values for filling a sample outside of a VOP, the two boundary samples are averaged (//2).

hor_pad[y][x] is generated by any process equivalent to the following example. For every line with at least one shape sample *s[y][x]* == 1(inside the VOP) :

© ISO/IEC 2004 – All rights reserved

```
for (x=0; x<N; x++) {
    if (s[y][x] == 1) { hor_pad[y][x] = d[y][x]; s'[y][x] = 1; }
    else {
        if ( s[y][x'] == 1 && s[y][x"] == 1)  {
            hor_pad[y][x] = (d[y][x']+ d[y][x"])//2;
            s'[y][x] = 1;
        } else if ( s[y][x'] == 1 ) {
            hor_pad[y][x] = d[y][x']; s'[y][x] = 1;
        } else if ( s[y][x"] == 1 ) {
            hor_pad[y][x] = d[y][x"]; s'[y][x] = 1;
        }
    }
}
```

where *x'* is the location of the nearest valid sample (*s[y][x']* == 1) at the VOP boundary to the left of the current location *x*, *x"* is the location of the nearest boundary sample to the right, and *N* is the number of samples of a line in a macroblock. *s'[y][x]* is initialized to 0.

7.16.7.1.1.2 Vertical repetitive padding

The remaining unfilled transparent horizontal samples (where *s'[y][x]* == 0) from subclause 7.16.7.1.1.1 are padded by a similar process as the horizontal repetitive padding but in the *vertical* direction. The samples already filled in subclause 7.16.7.1.1.1 are treated as if they were inside the VOP for the purpose of this vertical pass.

hv_pad[y][x] is generated by any process equivalent to the following example. For every column of *hor_pad[y][x]* :

```
for (y=0; y<M; y++) {
    if (s'[y][x] == 1)
        hv_pad[y][x] =hor_pad[y][x];
    else {
        if ( s'[y'][x] == 1 && s'[y"][x] == 1 )
            hv_pad[y][x] = (hor_pad[y'][x] + hor_pad[y"][x])//2;
        else if ( s'[y'][x] == 1 )
            hv_pad[y][x] = hor_pad[y'][x];
        else if (s'[y"][x] == 1 )
            hv_pad[y][x] = hor_pad[y"][x];
    }
}
```


© ISO/IEC 2004 – All rights reserved

where *y'* is the location of the nearest valid sample (*s'[y'][x]* == 1) above the current location *y* at the boundary of *hv_pad*, *y"* is the location of the nearest boundary sample below *y*, and *M* is the number of samples of a column in a macroblock.

7.16.7.1.1.3 Extended padding

Exterior macroblocks immediately next to boundary macroblocks are filled by replicating the samples at the border of the boundary macroblocks. Note that the boundary macroblocks have been completely padded in subclause 7.16.7.1.1.1 and subclause 7.16.7.1.1.2. If an exterior macroblock is next to more than one boundary macroblock, one of the macroblocks is chosen, according to the following convention, for reference.

The boundary macroblocks surrounding an exterior macroblock are numbered in priority according to Figure 7-111. The exterior macroblock is then padded by replicating upwards, downwards, leftwards, or rightwards the row of samples from the horizontal or vertical border of the boundary macroblock having the largest priority number.

The remaining exterior macroblocks (not located next to any boundary macroblocks) are filled with $2^{bits\_per\_pixel-1}$. For 8-bit luminance component and associated chrominance this implies filling with 128.

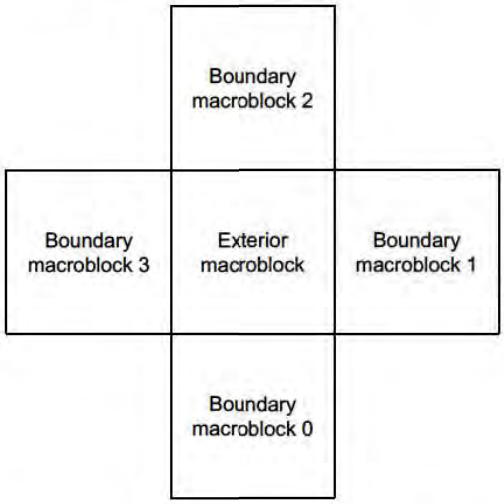


Figure 7-111 — Priority of boundary macroblocks surrounding an exterior macroblock

7.16.7.1.1.4 Padding for chrominance components

Chrominance components are padded according to subclauses 7.16.7.1.1.1 through 7.16.7.1.1.3. The padding is performed by referring to a shape block generated by decimating the shape block of the corresponding luminance component. This decimating of the shape block is performed by the subsampling process described in subclause 6.1.3.6.

7.16.7.1.1.5 Padding of interlaced macroblocks

Macroblocks of an interlaced VOP (progressive_sequence = 0) are padded according to 7.16.7.1.1.1 through 7.16.7.1.1.3. The vertical padding of the luminance component, however, is performed for each field independently. A sample outside of a VOP is therefore filled with the value of the nearest boundary sample of the same field. Completely transparent blocks are padded with $2^{bits\_per\_pixel-1}$. Chrominance components of interlaced VOP are padded according to subclause 7.16.7.1.1.4, however, based on fields. The padding method described in this subclause is not used outside the bounding rectangle of the VOP.

© ISO/IEC 2004 – All rights reserved

7.16.7.2 Prediction modes

There are two major classifications of the prediction mode:

— field prediction; and

— frame prediction.

In field prediction, predictions are made independently for each field by using data from one or more previously decoded fields. Frame prediction forms a prediction for the frame from one or more previously decoded frames. It must be understood that the fields and frames from which predictions are made may themselves have been decoded as either field VOPs or frame VOPs.

Within a field VOP all predictions are field predictions. However in a frame VOP either field predictions or frame predictions may be used (selected on a macroblock-by macroblock basis).

In addition to the major classification of field or frame prediction a special prediction mode is used:

- *16x8 motion compensation* - In which two motion vectors are used for each macroblock. The first motion vector is used for the upper 16x8 region, the second for the lower 16x8 region. In this specification 16x8 motion compensation shall only be used with field VOPs.

7.16.7.3 Prediction field and frame selection

The selection of which fields and frames shall be used to form predictions shall be made as detailed in this clause.

7.16.7.3.2 Field prediction

In P-VOPs, the two reference fields from which predictions shall made are the most recently decoded reference top field and the most recently decoded reference bottom field. The simplest case illustrated in Figure 7-112 shall be used when predicting the first VOP of a coded frame or when using field prediction within a frame-VOP. In these cases the two reference fields are part of the same reconstructed frame.

NOTES -

1 The reference fields may themselves have been reconstructed from two field-VOPs or a single frame-VOP.

2 In the case of predicting a field VOP, the field being predicted may be either the top field or the bottom field.

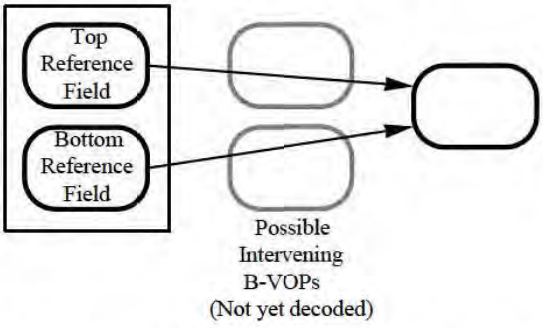


Figure 7-112 — Prediction of the first field or field prediction in a frame-VOP

The case when predicting the second field VOP of a coded frame is more complicated because the two most recently decoded reference fields shall be used, and in this case, the most recent reference field was obtained from decoding the first field VOP of the coded frame. Figure 7-113 illustrates the situation when this second VOP is the bottom field. Figure 7-114 illustrates the situation when this second VOP is the top field.

NOTE The earlier reference field may itself have been reconstructed by decoding a field VOP or a frame VOP.

© ISO/IEC 2004 – All rights reserved

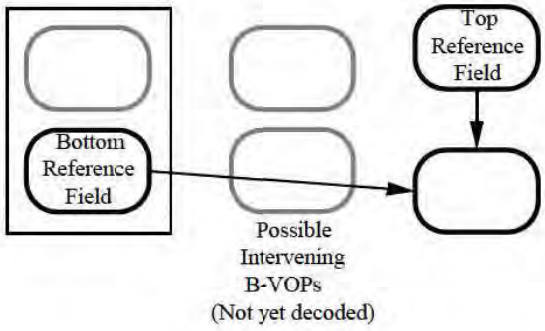


Figure 7-113 — Prediction of the second field-VOP when it is the bottom field

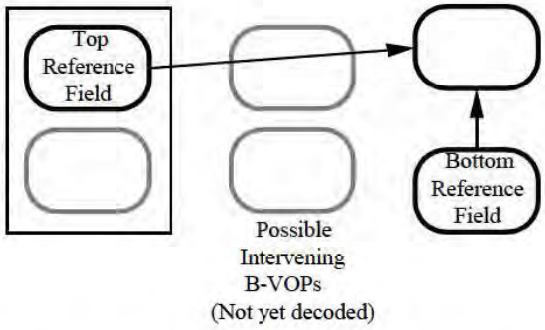


Figure 7-114 — Prediction of the second field-VOP when it is the top field

7.16.7.3.3 Frame prediction

In P-VOPs prediction shall be made from the most recently reconstructed reference frame. This is illustrated in Figure 7-115.

NOTE The reference frame may itself have been reconstructed from two field VOPs or a single frame VOP.

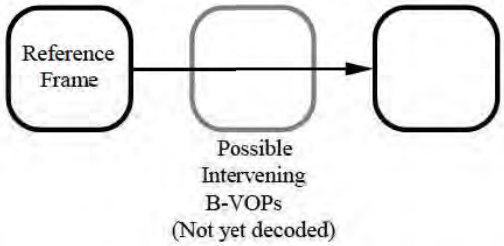


Figure 7-115 — Frame-prediction for P-VOPs

© ISO/IEC 2004 – All rights reserved

7.16.7.4 Motion vectors

Motion vectors are coded differentially with respect to previously decoded motion vectors in order to reduce the number of bits required to represent them. In order to decode the motion vectors the decoder shall maintain two motion vector predictors (each with a horizontal and vertical component) denoted *PMV*[*r*][*s*][*t*]. For each prediction, a motion vector, *vector'*[*r*][*s*][*t*] is first derived. This is then scaled depending on the sampling structure (4:2:0, 4:2:2 or 4:4:4) to give a motion vector, *vector*[*r*][*s*][*t*], for each colour component. The meanings associated with the dimensions in this array are defined in Table 7-38.

Table 7-38 — Meaning of indices in *PMV*[*r*][*s*][*t*], *vector*[*r*][*s*][*t*] and *vector'*[*r*][*s*][*t*]

	0	1
r	First motion vector in Macroblock	Second motion vector in Macroblock
s	Forward motion Vector	unused
t	Horizontal Component	Vertical Component

7.16.7.4.1 Decoding the motion vectors

Each motion vector component, *vector'*[*r*][*s*][*t*], shall be calculated by any process that is equivalent to the following one. Note that the motion vector predictors shall also be updated by this process.

```
r_size = vop_fcode[s][t] - 1
f = 1 << r_size
high = ( 16 * f ) -  1;
low = ( (-16) * f );
range = ( 32 * f );

if ( (f  == 1) || (motion_code[r][s][t] == 0) )
      delta = motion_code[r][s][t] ;
else {
      delta = ( ( Abs(motion_code[r][s][t]) - 1 ) * f ) + motion_residual[r][s][t] + 1;
      if (motion_code[r][s][t] < 0)
          delta = - delta;
}

prediction = PMV[r][s][t];
if ( (mv_format == "field") && (t==1) && (VOP_structure == "Frame VOP") )
      prediction = PMV[r][s][t] //// 2;

vector'[r][s][t]= prediction + delta;
if (vector'[r][s][t] < low )
      vector'[r][s][t] = vector'[r][s][t] + range;
if (vector'[r][s][t] > high)
      vector'[r][s][t] = vector'[r][s][t] - range;

if ( (mv_format == "field") && (t==1) && (VOP_structure == "Frame VOP") )
      PMV[r][s][t] = vector'[r][s][t] * 2;
else
      PMV[r][s][t] = vector'[r][s][t];
```

The parameters in the bitstream shall be such that the reconstructed differential motion vector, *delta*, shall lie in the range [*low*:*high*]. In addition the reconstructed motion vector, *vector'*[*r*][*s*][*t*], and the updated value of the motion vector predictor *PMV*[*r*][*s*][*t*], shall also lie in the range [*low* : *high*]. The allowed range [*low* : *high*] for the motion vectors depends on the parameter vop_fcode[s][t].

r_size, *f*, *delta*, *high*, *low* and *range* are temporary variables that are not used in the remainder of this specification.

© ISO/IEC 2004 – All rights reserved

motion_code[r][s][t] and motion_residual[r][s][t] are fields recovered from the bitstream. mv_format is recovered from the bitstream using Table 6-102 and Table 6-102.

r, *s* and *t* specify the particular motion vector component being processed as identified in Table 7-38.

vector[*r*][*s*][*t*] is the final reconstructed motion vector for the luminance component of the macroblock.

7.16.7.4.2 Motion vector restrictions

In frame VOPs, the vertical component of the field motion vectors shall be restricted so that they only cover half the range that is supported by the vop_fcode that relates to those motion vectors. This restriction ensures that the motion vector predictors will always have values that are appropriate for decoding subsequent frame motion vectors. Table 7-39 summarises the size of motion vectors that may be coded as a function of vop_fcode.

Table 7-39 — Allowable motion vector range as a function of *vop_fcode*[*s*][*t*]

vop_fcode[*s*][*t*]	Vertical components (*t*==1) of field vectors in frame VOPs	All other cases
0	(forbidden)	
1	[-4: +3,5]	[-8: +7,5]
2	[-8: +7,5]	[-16: +15,5]
3	[-16: +15,5]	[-32: +31,5]
4	[-32: +31,5]	[-64: +63,5]
5	[-64: +63,5]	[-128: +127,5]
6	[-128: +127,5]	[-256: +255,5]
7	[-256: +255,5]	[-512: +511,5]
8	[-512: +511,5]	[-1024: +1023,5]
9	[-1024: +1023,5]	[-2048: +2047,5]
10-14	(reserved)	
15	(used when a particular *vop_fcode*[*s*][*t*] will not be used)	

7.16.7.4.3 Updating motion vector predictors

Once all of the motion vectors present in the macroblock have been decoded using the process defined in the previous clause it is sometimes necessary to update other motion vector predictors. This is because in some prediction modes fewer than the maximum possible number of motion vectors are used. The remainder of the predictors that might be used in the VOP must retain "sensible" values in case they are subsequently used.

The motion vector predictors shall be updated as specified in Table 7-40 and Table 7-41. The rules for updating motion vector predictors in the case of skipped macroblocks are specified in 7.16.5.7.

NOTE It is possible for an implementation to optimise the updating (and resetting) of motion vector predictors depending on the VOP type. For example in a P-VOP the predictors for backwards motion vectors are unused and need not be maintained.

© ISO/IEC 2004 – All rights reserved

Table 7-40 — Updating of motion vector predictors in frame VOPs

frame_motion_-type	macroblock_motion_-forward	macroblock_-intra	Predictors to Update
Frame-based‡	-	1	*PMV*[1][0][1:0] = *PMV*[0][0][1:0]
Frame-based	1	0	*PMV*[1][0][1:0] = *PMV*[0][0][1:0]
Frame-based‡	0	0	*PMV*[*r*][*s*][*t*] = 0 §
Field-based	1	0	(none)

NOTE - *PMV*[*r*][*s*][1:0] = *PMV*[*u*][*v*][1:0] means that;
PMV[*r*][*s*][1] = *PMV*[*u*][*v*][1] and *PMV*[*r*][*s*][0] = *PMV*[*u*][*v*][0]

‡ **frame_motion_type** is not present in the bitstream but is assumed to be Frame-based

§ (Only occurs in P-VOP) *PMV*[*r*][*s*][*t*] is set to zero (for all *r*, *s* and *t*). See 7.16.7.4.4.

Table 7-41 — Updating of motion vector predictors in field VOPs

field_motion_-type	macroblock_motion_-forward	macroblock_-intra	Predictors to Update
Field-based‡	-	1	*PMV*[1][0][1:0] = *PMV*[0][0][1:0]
Field-based	1	0	*PMV*[1][0][1:0] = *PMV*[0][0][1:0]
Field-based‡	0	0	*PMV*[*r*][*s*][*t*] = 0 §
16x8 MC	1	0	(none)

NOTE - *PMV*[*r*][*s*][1:0] = *PMV*[*u*][*v*][1:0] means that;
PMV[*r*][*s*][1] = *PMV*[*u*][*v*][1] and *PMV*[*r*][*s*][0] = *PMV*[*u*][*v*][0]

‡ **field_motion_type** is not present in the bitstream but is assumed to be Field-based

§ (Only occurs in P-VOP) *PMV*[*r*][*s*][*t*] is set to zero (for all *r*, *s* and *t*). See 7.16.7.4.4.

7.16.7.4.4 Resetting motion vector predictors

All motion vector predictors shall be reset to zero in the following cases:

- At the start of each slice.
- Whenever a macroblock is completely transparent.
- Whenever an intra macroblock is decoded.
- Whenever a macroblock is encoded as DPCM residuals.
- In a P-VOP when a macroblock is skipped (not coded = 1).

7.16.7.4.5 Prediction in P-VOP

In the case that a P field VOP is used as the second field of a frame in which the first field is an I field VOP a series of semantic restrictions apply. These ensure that prediction is only made from the I field VOP. These restrictions are;

- Field prediction in which **motion_vertical_field_select** indicates the same parity as the field being predicted shall not be used.
- There shall be no skipped macroblocks (not_coded = 0).

7.16.7.4.6 Motion vectors for chrominance components

The motion vectors calculated in the previous clauses refer to the luminance component where;

vector[*r*][*s*][*t*] = *vector*'[*r*][*s*][*t*] (for all r, s and t)

© ISO/IEC 2004 – All rights reserved

For each of the two chrominance components the motion vectors shall be scaled as follows:

4:2:0 Both the horizontal and vertical components of the motion vector are scaled by dividing by two:

```
vector[r][s][0] = vector'[r][s][0] / 2;
vector[r][s][1] = vector'[r][s][1] / 2;
```

4:2:2 The horizontal component of the motion vector is scaled by dividing by two, the vertical component is not altered:

```
vector[r][s][0] = vector'[r][s][0] / 2;
vector[r][s][1] = vector'[r][s][1];
```

4:4:4 The motion vector is unmodified:

```
vector[r][s][0] = vector'[r][s][0];
vector[r][s][1] = vector'[r][s][1];
```

7.16.7.4.7 Semantic restrictions concerning predictions

It is a restriction on the bitstream that reconstructed motion vectors shall not refer to samples outside the decoded area of a reference VOP. For an arbitrary shape VOP, the decoded area refers to the area within the bounding rectangle, padded as described in subclause 7.16.7.1.1. A bounding rectangle is defined by vop_width and vop_height extended to multiple of 16.

7.16.7.5 Forming predictions

Predictions are formed by reading prediction samples from the reference fields or frames. A given sample is predicted by reading the corresponding sample in the reference field or frame offset by the motion vector in the absolute coordinate system.

A positive value of the horizontal component of a motion vector indicates that the prediction is made from samples (in the reference field/frame) that lie to the right of the samples being predicted in the absolute coordinate system.

A positive value of the vertical component of a motion vector indicates that the prediction is made from samples (in the reference field/frame) that lie the below the samples being predicted in the absolute coordinate system.

All motion vectors are specified to an accuracy of one half sample. Thus if a component of the motion vector is odd, the samples will be read from mid-way between the actual samples in the reference field/frame. These half-samples are calculated by simple linear interpolation from the actual samples.

In the case of field-based predictions it is necessary to determine which of the two available fields to use to form the prediction. In the case of field-based prediction and 16x8 MC an additional bit, motion_vertical_field_select, is encoded to indicate which field to use.

If motion_vertical_field_select is zero then the prediction is taken from the top reference field.

If motion_vertical_field_select is one then the prediction is taken from the bottom reference field.

For each prediction block the integer sample motion vectors int_vec[t] and the half sample flags half_flag[t] shall be formed as follows;

© ISO/IEC 2004 – All rights reserved

```
for (t=0 ; t<2 ; t++) {
    int_vec[t] = vector[r][s][t] //// 2 ;
    if ((vector[r][s][t] – (2 * int_vec[t]) != 0)
        half_flag[t] = 1 ;
    else
        half_flag[t] = 0 ;
}
```

Then for each sample in the prediction block the samples are read and the half sample prediction applied as follows;

```
if ( ( ! half_flag[0] )&& ( ! half_flag[1]) )
    pel_pred[y][x] = pel_ref[y + int_vec[1]][x + int_vec[0]] ;

if ( ( ! half_flag[0] )&& half_flag[1] )
    pel_pred[y][x] = ( pel_ref[y + int_vec[1]][x + int_vec[0]] +
                       pel_ref[y + int_vec[1]+1][x + int_vec[0]] ) // 2 ;

if ( half_flag[0]&& ( ! half_flag[1]) )
    pel_pred[y][x] = ( pel_ref[y + int_vec[1]][x + int_vec[0]] +
                       pel_ref[y + int_vec[1]][x + int_vec[0]+1] ) // 2 ;

if ( half_flag[0]&& half_flag[1] )
    pel_pred[y][x] = ( pel_ref[y + int_vec[1]][x + int_vec[0]] +
                       pel_ref[y + int_vec[1]][x + int_vec[0]+1] +
                           pel_ref[y + int_vec[1]+1][x + int_vec[0]] +
                               pel_ref[y + int_vec[1]+1][x + int_vec[0]+1] ) // 4 ;
```

where *pel_pred*[*y*][*x*] is the prediction sample being formed and *pel_ref*[*y*][*x*] are samples in the reference field or frame.

7.16.7.6 Motion vector selection

Table 7-42 shows the prediction modes used in field VOPs and Table 7-43 shows the predictions used in frame VOPs. In each table the motion vectors that are present in the bitstream are listed in the order in which they appear in the bitstream.

Table 7-42 — Predictions and motion vectors in field VOPs

field_ motion_ type	macroblock_motion_- forward	macro- block_- intra	Motion vector	Prediction formed for
Field-based‡	-	1	*vector*[0][0][1:0]	None
Field-based	1	0	*vector*[0][0][1:0]	whole field, forward
Field-based‡	0	0	*vector*[0][0][1:0]*§	whole field, forward
16x8 MC	1	0	*vector*[0][0][1:0] *vector*[1][0][1:0]	upper 16x8 field, forward lower 16x8 field, forward

NOTE - Motion vectors are listed in the order they appear in the bitstream

‡ **field_motion_type** is not present in the bitstream but is assumed to be Field-based

* These motion vectors are not present in the bitstream

§ The motion vector is taken to be (0; 0) as explained in 7.16.7.4.5.

© ISO/IEC 2004 – All rights reserved

Table 7-43 — Predictions and motion vectors in frame VOPs

frame_-motion_-type	macroblock_motion_-forward	macro-block_-intra	Motion vector	Prediction formed for
Frame-based‡	-	1	*vector*[0][0][1:0]	None
Frame-based	1	0	*vector*[0][0][1:0]	frame, forward
Frame-based‡	0	0	*vector*[0][0][1:0]*§	frame, forward
Field-based	1	0	*vector*[0][0][1:0] *vector*[1][0][1:0]	top field, forward bottom field, forward

NOTE - Motion vectors are listed in the order they appear in the bitstream
‡ **frame_motion_type** is not present in the bitstream but is assumed to be Frame-based
* These motion vectors are not present in the bitstream
§ The motion vector is taken to be (0; 0) as explained in 7.16.7.4.5.

7.16.7.7 Skipped macroblocks

A skipped macroblock is a macroblock for which no data is encoded, that is part of a coded slice. With the exception of the first non-transparent macroblock in a slice, if not_coded flag in a macroblock is '1', the macroblock is to be skipped. The decoder shall form a prediction for skipped macroblocks which shall then be used as the final decoded sample values.

The process differs between field VOPs and frame VOPs.

There shall be no skipped macroblocks in I-VOPs.

7.16.7.7.1 P field VOP

- The prediction shall be made as if field_motion_type is "Field-based"
- The prediction shall be made from the field of the same parity as the field being predicted.
- Motion vector predictors shall be reset to zero.
- The motion vector shall be zero.

7.16.7.7.2 P frame VOP

- The prediction shall be made as if frame_motion_type is "Frame-based"
- Motion vector predictors shall be reset to zero.
- The motion vector shall be zero.

7.16.7.8 Combining predictions

The final stage is to combine the various predictions together in order to form the final prediction blocks.

It is also necessary to organise the data into blocks that are either field organised or frame organised in order to be added directly to the decoded coefficients.

The transform data is either field organised or frame organised as specified by *dct_type*.

© ISO/IEC 2004 – All rights reserved

7.16.7.8.1 Simple frame predictions

In the case of simple frame predictions no further processing is required.

The predictions for chrominance components of 4:2:0, 4:2:2 and 4:4:4 formats shall be of size 8 samples by 8 lines, 8 samples by 16 lines and 16 samples by 16 lines respectively.

7.16.7.8.2 Simple field predictions

In the case of simple field predictions (i.e. not 16×8) no further processing is required

In the case of simple field prediction in a frame picture the predictions for chrominance components of 4:2:0, 4:2:2 and 4:4:4 formats for each field shall be of size 8 samples by 4 lines, 8 samples by 8 lines and 16 samples by 8 lines respectively.

7.16.7.8.3 16x8 Motion compensation

In this prediction mode separate predictions are formed for the upper 16x8 region of the macroblock and the lower 16x8 region of the macroblock.

The predictions for chrominance components, for each 16x8 region, of 4:2:0, 4:2:2 and 4:4:4 formats shall be of size 8 samples by 4 lines, 8 samples by 8 lines and 16 samples by 8 lines respectively.

7.16.8 Output of the decoding process

This section describes the output of the theoretical model of the decoding process that decodes bitstreams conforming to this specification.

The decoding process input is one or more coded video bitstreams (one for each of the layers). The video layers are generally multiplexed by the means of a system stream that also contains timing information.

The output of the decoding process is a series of fields or frames that are normally the input of a display process. The order in which fields or frames are output by the decoding process is called the display order, and may be different from the coded order (when B-pictures are used). The display process is responsible for the action of displaying the decoded fields or frames on a display device. If the display device cannot display at the frame rate indicated in the bitstream, the display process may perform frame rate conversion. This specification does not describe a theoretical model of the display process nor the operation of the display process.

Since some of the syntax elements, such as progressive_frame, may be needed by the display process, in this theoretical model of the decoding process, all the syntactic elements that are decoded by the decoding process are output by the decoding process and may be accessed by the display process.

When a progressive sequence is decoded (progressive_sequence is equal to 1), the luminance and chrominance samples of the reconstructed frames are output by decoding process in the form of progressive frames and the output rate is the frame rate. Figure 7-116 illustrates this in the case of chroma_format equals to 4:2:0.

© ISO/IEC 2004 – All rights reserved

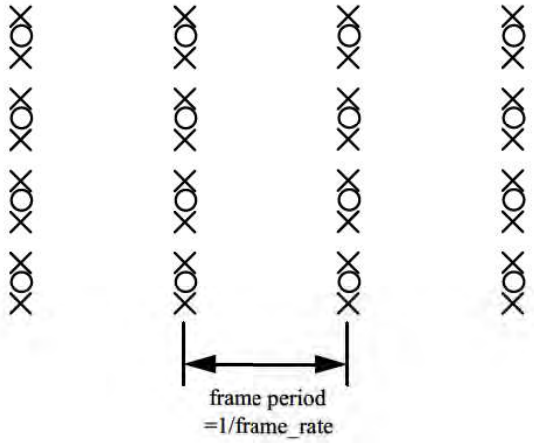


Figure 7-116 — progressive_sequence == 1

The same reconstructed frame is output one time if repeat_first_field is equal to 0, and two or three consecutive times if repeat_first_field is equal to 1, depending on the value of top_field_first. Figure 7-117 illustrates this in the case of chroma_format equals to 4:2:0 and repeat_first_field equals 1.

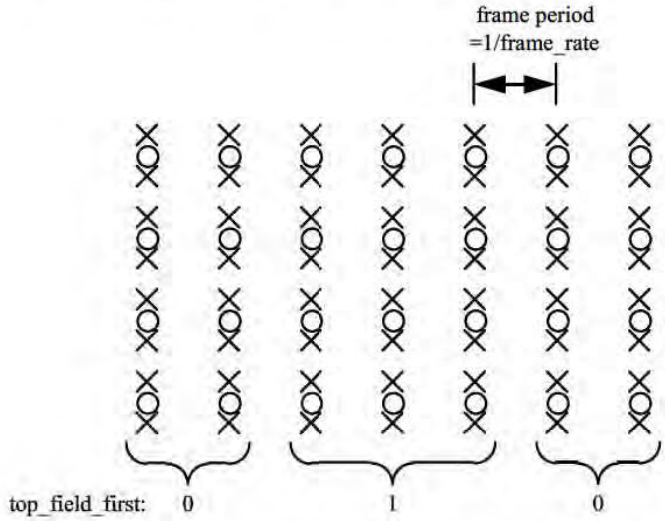


Figure 7-117 — progressive_sequence == 1, repeat_first_field = 1

When decoding an interlaced sequence (progressive_sequence is equal to 0), the luminance samples of the reconstructed frames are output by the decoding process in the form of interlaced fields at a rate that is twice the frame rate. Figure 7-118 illustrates this.

© ISO/IEC 2004 – All rights reserved

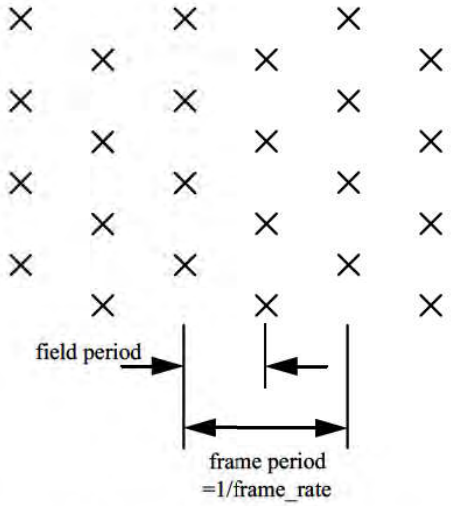


Figure 7-118 — progressive_sequence == 0

It is a requirement on the bitstream that the fields at the output of the decoding process shall always be alternately top and bottom (note that the very first field of a sequence may be either top or bottom).

If the reconstructed frame is interlaced (progressive_frame is equal to 0), the luminance samples and chrominance samples are output by the decoding process in the form of two consecutive fields. The first field output by the decoding process is the top field or the bottom field of the reconstructed frame, depending on the value of top_field_first.

Although all the samples of progressive frames represent the same instant in time, all the samples are not output at the same time by the decoding process when the sequence is interlaced.

If the reconstructed frame is progressive (progressive_frame is equal to 1), the luminance samples are output by the decoding process in the form of two or three consecutive fields, depending on the value of repeat_first_field.

NOTE The information that these fields originate from the same progressive frame in the bitstream is conveyed to the display process.

All of the chrominance samples of the reconstructed progressive frame are output by the decoding process at the same time as the first field of luminance samples. This is illustrated in Figure 7-119 and Figure 7-120.

© ISO/IEC 2004 – All rights reserved

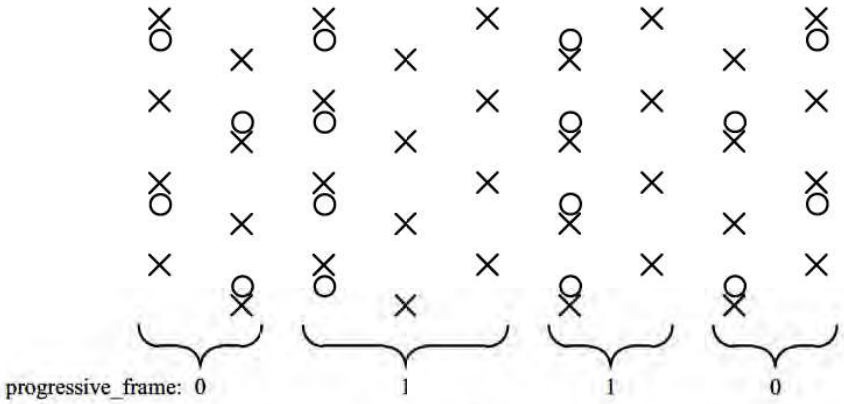


Figure 7-119 — progressive_sequence == 0 with 4:2:0 chrominance.

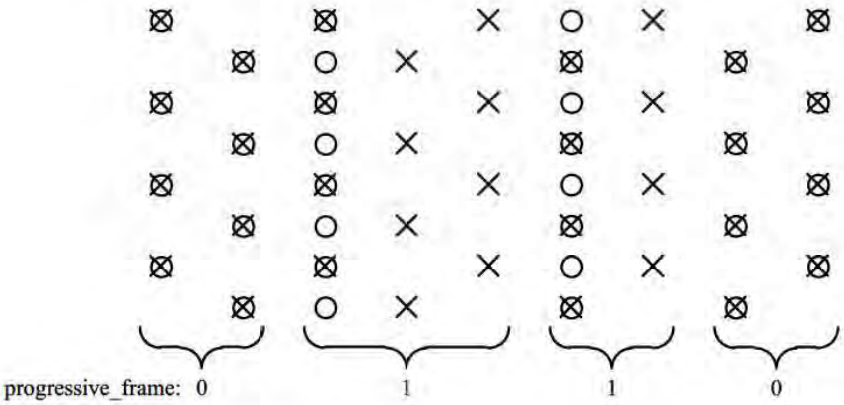


Figure 7-120 — progressive_sequence == 0 with 4:2:2 or 4:4:4 chrominance.

7.16.9 Sprite decoding

The subclause specifies the additional decoding process for a sprite video object. The sprite decoding can operate in two modes: basic sprite decoding and low-latency sprite decoding. Figure 7-121 is a diagram of the sprite decoding process. It is simplified for clarity.

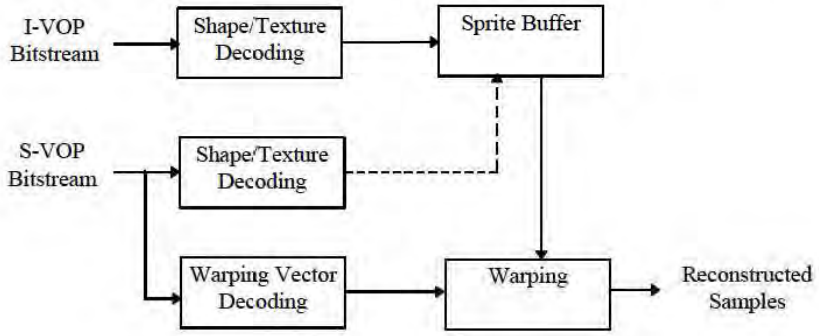


Figure 7-121 — The sprite decoding process

© ISO/IEC 2004 – All rights reserved

7.16.9.1 Higher syntactic structures

The various parameters in the VOL and VOP bitstreams shall be interpreted as described in Clause 6. When sprite_enable == '1', vop_coding_type shall be "I" only for the initial VOP in a VOL for basic sprites (i.e. low_latency_sprite_enable == '0'), and all the other VOPs shall be S-VOPs (i.e. vop_coding_type == "S"). The reconstructed I-VOP in a VOL for basic sprites is not displayed but stored in a sprite memory, and will be used by all the remaining S-VOPs in the same VOL. An S-VOP is reconstructed by applying warping to the VOP stored in the sprite memory, using the warping parameters (i.e. a set of motion vectors) embedded in the VOP bitstream. Alternatively, in a VOL for low-latency sprites (i.e. low_latency_sprite_enable == '1'), these S-VOPs can update the information stored in the sprite memory before applying warping.

7.16.9.2 Sprite Reconstruction

The luminance, chrominance and grayscale alpha data of a sprite are stored in two-dimensional arrays. The width and height of the luminance array are specified by sprite_width and sprite_height respectively. The samples in the sprite luminance, chrominance and grayscale alpha arrays are addressed by two-dimensional integer pairs (i', j') and (i_c', j_c') as defined in the following:

- Top left luminance and grayscale alpha sample
  (i', j') = (sprite_left_coordinate, sprite_top_coordinate)
- Bottom right luminance and grayscale alpha sample
  (i', j') = (sprite_left_coordinate + sprite_width – 1,
  sprite_top_coordinate + sprite_height – 1)
- Top left chrominance sample
  For 4:2:0 VOPs,
  (i_c', j_c') = (sprite_left_coordinate / 2, sprite_top_coordinate / 2)
  For 4:2:2 VOPs,
  (i_c', j_c') = (sprite_left_coordinate / 2, sprite_top_coordinate)
  For 4:4:4 VOPs,
  (i_c', j_c') = (sprite_left_coordinate, sprite_top_coordinate)
- Bottom right chrominance sample
  For 4:2:0 VOPs,
  (i_c', j_c') = (sprite_left_coordinate / 2 + sprite_width// 2 – 1,
  sprite_top_coordinate / 2 + sprite_height// 2 – 1).
  For 4:2:2 VOPs,
  (i_c', j_c') = (sprite_left_coordinate / 2 + sprite_width// 2 – 1,
  sprite_top_coordinate + sprite_height – 1).
  For 4:4:4 VOPs,
  (i_c', j_c') = (sprite_left_coordinate + sprite_width – 1,
  sprite_top_coordinate + sprite_height – 1).
- Likewise, the addresses of the luminance, chrominance and grayscale alpha samples of the VOP currently being decoded are defined in the following:
- Top left sample of luminance and grayscale alpha
  (i, j) = (0, 0) for rectangular VOPs, and
  (i, j) = (vop_horizontal_mc_spatial_ref, vop_vertical_mc_spatial_ref) for non-rectangular VOPs
- Bottom right sample of luminance and grayscale alpha
  (i, j) = (video_object_layer_width - 1, video_object_layer_height - 1) for rectangular VOPs, and
  (i, j) = (vop_horizontal_mc_spatial_ref + vop_width - 1,
  vop_vertical_mc_spatial_ref + vop_height - 1) for non-rectangular VOPs
- Top left sample of chrominance
  (i_c, j_c) = (0, 0) for rectangular VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2, vop_vertica_mc_spatial_ref / 2) for non-rectangular 4:2:0 VOPs

© ISO/IEC 2004 – All rights reserved

(i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2, vop_vertical_mc_spatial_ref) for non-rectangular 4:2:2 VOPs
(i_c, j_c) = (vop_horizontal_mc_spatial_ref, vop_vertical_mc_spatial_ref) for non-rectangular 4:4:4 VOPs

- Bottom right sample of chrominance
  (i_c, j_c) = (video_object_layer_width / 2 - 1, video_object_layer_height / 2 - 1) for rectangular 4:2:0 VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2 + vop_width// 2 - 1,
  vop_vertical_mc_spatial_ref / 2 + vop_height// 2 - 1) for non-rectangular 4:2:0 VOPs
  (i_c, j_c) = (video_object_layer_width / 2 - 1, video_object_layer_height - 1) for rectangular 4:2:2 VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref / 2 + vop_width// 2 - 1,
  vop_vertical_mc_spatial_ref + vop_height - 1) for non-rectangular 4:2:2 VOPs
  (i_c, j_c) = (video_object_layer_width - 1, video_object_layer_height - 1) for rectangular 4:4:4 VOPs, and
  (i_c, j_c) = (vop_horizontal_mc_spatial_ref + vop_width- 1,
  vop_vertical_mc_spatial_ref + vop_height - 1) for non-rectangular 4:4:4 VOPs

7.16.9.3 Sprite reference point decoding

The syntactic elements in sprite_trajectory () and below shall be interpreted as specified in Clause 6. du[i] and dv[i] ($0 \le i <$ no_sprite_point) specifies the mapping between indexes of some reference points in the VOP and the corresponding reference points in the sprite. These points are referred to as VOP reference points and sprite reference points, respectively, in the rest of the specification.

The index values for the VOP reference points are defined as:

(i_0, j_0) = (0, 0) when video_object_layer_shape == 'rectanglar', and
(vop_horizontal_mc_spatial_ref, vop_vertical_mc_spatial_ref) otherwise,
$(i_1, j_1) = (i_0+W,\ j_0)$,
$(i_2, j_2) = (i_0,\ j_0 + H)$,
$(i_3, j_3) = (i_0+W,\ j_0+H)$

where W = video_object_layer_width and H = video_object_layer_height when video_object_layer_shape == 'rectanglar' or W = vop_width and H = vop_height otherwise. Only the index values with subscripts less than no_sprite_point shall be used for the rest of the decoding process.

The index values for the sprite reference points shall be calculated as follows:

$(i_0', j_0') = (s / 2)\ (2\ i_0 + du[0],\ 2\ j_0 + dv[0])$
$(i_1', j_1') = (s / 2)\ (2\ i_1 + du[1] + du[0],\ 2\ j_1 + dv[1] + dv[0])$
$(i_2', j_2') = (s / 2)\ (2\ i_2 + du[2] + du[0],\ 2\ j_2 + dv[2] + dv[0])$
$(i_3', j_3') = (s / 2)\ (2\ i_3 + du[3] + du[2] + du[1] + du[0],\ 2\ j_3 + dv[3] + dv[2] + dv[1] + dv[0])$

where i_0', j_0', etc are integers in $\frac{1}{s}$ pel accuracy, where s is specified by sprite_warping_accuracy. Only the index values with substcripts less than no_sprite_point need to be calculated.

When no_of_sprite_warping_points == 2 or 3, the index values for the *virtual sprite points* are additionally calculated as follows:

$$(i_1'', j_1'') = (16\ (i_0 + W') + ((W - W')\ (r\ i_0' - 16\ i_0) + W'\ (r\ i_1' - 16\ i_1))\ //\ W,$$
$$16\ j_0 + ((W - W')\ (r\ j_0' - 16\ j_0) + W'\ (r\ j_1' - 16\ j_1))\ //\ W)$$
$$(i_2'', j_2'') = (16\ i_0 + ((H - H')\ (r\ i_0' - 16\ i_0) + H'\ (r\ i_2' - 16\ i_2))\ //\ H,$$
$$16\ (j_0 + H') + ((H - H')\ (r\ j_0' - 16\ j_0) + H'\ (r\ j_2' - 16\ j_2))\ //\ H)$$

where i_1'', j_1'', i_2'', and j_2'' are integers in $\frac{1}{16}$ pel accuracy, and $r = 16/s$. W' and H' are defined as the smallest integers that satisfy the following condition:

$W' = 2^{\alpha}$, $H' = 2^{\beta}$, $W' \ge W$, $H' \ge H$, $\alpha > 0$, $\beta > 0$, both α and β are integers.

The calculation of i_2'', and j_2'' is not necessary when no_of_sprite_warping_points == 2.

© ISO/IEC 2004 – All rights reserved

7.16.9.4 Warping

For any pixel (i, j) inside the VOP boundary, $(F(i, j), G(i, j))$ and $(F_c(i_c, j_c), G_c(i_c, j_c))$ are computed as described as follows. These quantities are then used for sample reconstruction as specified in subclause 7.16.9.5. The following notations are used to simplify the description:

$I = i - i_0$,
$J = j - j_0$,

For 4:2:0 VOPs,

$I_c = 4\, i_c - 2\, i_0 + 1$,

$J_c = 4\, j_c - 2\, j_0 + 1$,

For 4:2:2 VOPs,

$I_c = 4\, i_c - 2\, i_0 + 1$, for calculating $F_c(i_c, j_c)$,

$I_c = 2\, i_c - i_0$, for calculating $G_c(i_c, j_c)$,

$J_c = 2\, j_c - 2\, j_0 + 1$, for calculating $F_c(i_c, j_c)$,

$J_c = j_c - j_0$, for calculating $G_c(i_c, j_c)$,

For 4:4:4 VOPs,

$I_c = i_c - i_0$,

$J_c = j_c - j_0$,

When no_of_sprite_warping_point == 0,

$(F(i, j), G(i, j)) = (si, sj)$,
$(F_c(i_c, j_c), G_c(i_c, j_c)) = (si_c, sj_c)$.

When no_of_sprite_warping_point == 1,

$(F(i, j), G(i, j)) = (i_0' + sI, j_0' + sJ)$,
$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' /// 2 + s\,(i_c - i_0 / 2),\ j_0' /// 2 + s\,(j_c - j_0 / 2))$, for 4:2:0 VOPs.

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' /// 2 + s\,(i_c - i_0 / 2),\ j_0' + sJ_c)$, for 4:2:2 VOPs.

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' + sI_c,\ j_0' + sJ_c)$, for 4:4:4 VOPs.

When no_of_sprite_warping_points == 2,

$$(F(i, j), G(i, j)) = \begin{aligned} &(i_0' + ((-r\, i_0' + i_1'')\, I + (r\, j_0' - j_1'')\, J) /// (W'\, r),\\ &\ j_0' + ((-r\, j_0' + j_1'')\, I + (-r\, i_0' + i_1'')\, J) /// (W'\, r)), \end{aligned}$$

$$(F_c(i_c, j_c), G_c(i_c, j_c)) = \begin{aligned} &(((-r\, i_0' + i_1'')\, I_c + (r\, j_0' - j_1'')\, J_c + 2\, W'\, r\, i_0' - 16W') /// (4\, W'\, r),\\ &\ ((-r\, j_0' + j_1'')\, I_c + (-r\, i_0' + i_1'')\, J_c + 2\, W'\, r\, j_0' - 16W') /// (4\, W'\, r)), \end{aligned}$$

for 4:2:0 VOPs,

© ISO/IEC 2004 – All rights reserved

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2\,W'\,r\,i_0' - 16W') /// (4\,W'\,r),$
$j_0' + ((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c) /// (W'\,r)),$

for 4:2:2 VOPs,

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' + ((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c) /// (W'\,r),$
$j_0' + ((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c) /// (W'\,r)),$

for 4:4:4 VOPs.

According to the definition of W' and H' (i.e. $W' = 2^{\alpha}$ and $H' = 2^{\beta}$), the divisions by "///" in these functions can be replaced by binary shift operations. By this replacement, the above equations can be rewritten as:

$(F(i, j), G(i, j)) = (i_0' + (((-r\,i_0' + i_1'')\,I + (r\,j_0' - j_1'')\,J + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),$
$j_0' + (((-r\,j_0' + j_1'')\,I + (-r\,i_0' + i_1'')\,J + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),$
$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2\,W'\,r\,i_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2),$
$((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c + 2\,W'\,r\,j_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2)),$

for 4:2:0 VOPs,

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2\,W'\,r\,i_0' - 16W' + 2^{\alpha+\rho+1}) >> (\alpha+\rho+2),$
$j_0' + (((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c + 2^{\alpha+\rho-1})),$

for 4:2:2 VOPs,

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' + (((-r\,i_0' + i_1'')\,I_c + (r\,j_0' - j_1'')\,J_c + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),$
$j_0' + (((-r\,j_0' + j_1'')\,I_c + (-r\,i_0' + i_1'')\,J_c + 2^{\alpha+\rho-1}) >> (\alpha+\rho)),$

for 4:4:4 VOPs.

where $2^{\rho}=r$.

When no_of_sprite_warping_points == 3,

$(F(i, j), G(i, j)) = (i_0' + ((-r\,i_0' + i_1'')\,H'\,I + (-r\,i_0' + i_2'')W'\,J) /// (W'H'r),$
$j_0' + ((-r\,j_0' + j_1'')\,H'\,I + (-r\,j_0' + j_2'')W'\,J) /// (W'H'r)),$
$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,H'\,I_c + (-r\,i_0' + i_2'')W'\,J_c + 2\,W'H'r\,i_0' - 16W'H') /// (4W'H'r),$
$((-r\,j_0' + j_1'')\,H'\,I_c + (-r\,j_0' + j_2'')W'\,J_c + 2\,W'H'r\,j_0' - 16W'H') /// (4W'H'r)),$

for 4:2:0 VOPs,

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (((-r\,i_0' + i_1'')\,H'\,I_c + (-r\,i_0' + i_2'')W'\,J_c + 2\,W'H'r\,i_0' - 16W'H') /// (4W'H'r),$
$j_0' + ((-r\,j_0' + j_1'')\,H'\,I_c + (-r\,j_0' + j_2'')W'\,J_c) /// (W'H'r))$

for 4:2:2 VOPs,

$(F_c(i_c, j_c), G_c(i_c, j_c)) = (i_0' + ((-r\,i_0' + i_1'')\,H'\,I_c + (-r\,i_0' + i_2'')W'\,J_c) /// (W'H'r),$
$j_0' + ((-r\,j_0' + j_1'')\,H'\,I_c + (-r\,j_0' + j_2'')W'\,J_c) /// (W'H'r)),$

for 4:4:4 VOPs.

According to the definition of W' and H', the computation of these functions can be simplified by dividing the denominator and numerator of the division beforehand by W' (when $W' < H'$) or H' (when $W' \geq H'$). As in the case of no_of_sprite_warping_points == 2, the divisions by "///" in these functions can be replaced by binary shift operations. For example, when $W' \geq H'$ (i.e. $\alpha \geq \beta$) the above equations can be rewritten as:

© ISO/IEC 2004 – All rights reserved