

ADVANCED SCREEN CONTENT CODING SOLUTION

TECHNICAL FIELD

[0001] The present disclosure is generally directed to screen
5 content coding.

BACKGROUND

[0002] Screen content coding imposes new challenges for video
compression technology because of its distinct signal
characteristics compared with conventional natural videos. There
10 appear to be a few promising techniques for the advanced screen
content coding, e.g., pseudo string match, color palette coding,
and intra motion compensation or intra block copy.

[0003] Among these techniques, pseudo string match shows the
highest gain for lossless coding, but with significant complexity
15 overhead and difficulties on lossy coding mode. The color palette
coding is developed for screen content under the assumption that
non-camera captured content typically contains a limited few
distinct colors, rather than the continuous color tone in natural
videos. Even though the pseudo string match and color palette
20 coding methods showed great potential, intra motion compensation
or intra block copy was adopted into the working draft (WD)
version 4 and reference software of on-going HEVC range extension
(HEVC RExt) for screen content coding. This is mainly due to the
fact that the motion estimation and compensation approach has
25 been studied extensively over decades, as well as its idea and
practical implementation is fairly easy (especially for
hardware).

[0004] However, the coding performance of intra block copy is
bounded because of its fixed block structure partitions. On the
30 other hand, performing block matching, something similar to
motion estimation in intra picture, also brings up the encoder
complexity significantly on both computing and memory access.

SUMMARY

[0005] This disclosure is directed to an advanced screen content coding solution.

[0006] In one example embodiment, a method for coding screen
5 content into a bitstream selects a color palette table for a coding unit (CU) of screen content. The color palette table created for the CU and a color palette table is created for a neighboring CU. A color index map is created having indices for the coding unit (CU) of the screen content using the selected
10 color palette table. The selected color palette table and the color index map are encoded/compressed for each of a plurality of CUs into a bitstream.

BRIEF DESCRIPTION OF THE FIGURES

[0007] For a more complete understanding of the present disclosure, and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, wherein like numbers designate like objects, and in which:

[0008] FIGURE 1 illustrates a screen content encoding solution using color palette table and index map mode or palette mode according to one example embodiment of this disclosure;

[0009] FIGURE 2 illustrates a screen content decoding solution for color palette table and index map mode or palette mode;

[0010] FIGURE 3 illustrates a process or workflow of the screen content solution for this color palette table and index map mode or palette mode of a CU;

[0011] FIGURE 4 illustrates a conventional G, B, R in planar mode (left) to Packed mode (right);

[0012] FIGURE 5 illustrates color palette table re-generation using neighboring reconstructed blocks;

[0013] FIGURE 6 illustrates an index map is parsed from a real word screen content;

[0014] FIGURE 7 illustrates a piece of a segment for a 1-D search after horizontal scanning;

[0015] FIGURE 8 illustrates a U_PIXEL module;

[0016] FIGURE 9 illustrates a U_ROW module;

[0017] FIGURE 10 illustrates a U_CMP module;

[0018] FIGURE 11 illustrates a U_COL module;

[0019] FIGURE 12 illustrates a U_2D_BLOCK module;

[0020] FIGURE 13 is an illustration of horizontal and vertical scan for index map processing of a exemplified CU;

[0021] FIGURE 14A is an illustration of a 4:2:0 chroma sampling format;

[0022] FIGURE 14B is an illustration of a 4:4:4 chroma sampling format;

[0023] FIGURE 15 illustrates an interpolation between 4:2:0 and 4:4:4;

[0024] FIGURE 16 illustrates Index Map processing with Upper/Left line buffer;

5 [0025] FIGURE 17 illustrates the apparatus and methods/flows incorporated into the current HEVC;

[0026] FIGURE 18 illustrates one example of a communication system; and

[0027] FIGURE 19A and FIGURE 19B illustrate example devices
10 that may implement the methods and teachings according to this disclosure.

DETAILED DESCRIPTION

[0028] In this disclosure, an advanced screen content coding solution is described that outperforms a High-Efficiency Video Coding (HEVC) range extension (such as HEVC Version 2 or HEVC RExt). This new solution includes several algorithms that are designed specifically for coding screen content. These algorithms include pixel representation using a color palette or a color table, referred to herein as a color palette table, color palette compression, color index map compression, string search, and residual compression. This technology is developed, harmonized, and can be integrated with the HEVC range extension (RExt) and future HEVC extensions to support efficient screen content coding. However, this technology could be implemented with any existing video standards. For simplicity, HEVC RExt is used as an example in the description below, and HEVC RExt software is used to describe and demonstrate the compression efficiency. This solution is integrated as an additional mode by using a color palette table and index map, defined herein as a color palette mode, in HEVC to demonstrate the performance.

[0029] The concept and description of this disclosure is illustrated in the Figures. FIGURE 1 shows an encoder 10 having a processor 12 including memory, and FIGURE 2 shows a decoder 14 having a processor 16 and memory, together illustrating an example embodiment of an encoding and decoding solution for the color palette mode, respectively, in accordance with this disclosure. As shown, the encoder 10 and decoder 14 each comprise a processor and memory and form a codec solution. The codec solution includes the processor 12 of encoder 10 executing new algorithms or methods including Process 1 creating a Color Palette Table, Process 2 classifying colors or pixel values using a previously derived color palette table corresponding color indices, Process 3 encoding the Color Palette Table, Process 4 encoding the color index map, Process 5 encoding the residuals,

and Process 6 writing new syntax elements into the compressed bitstream. Processor 16 of decoder 14 executes new algorithms or methods including the reverse steps. FIGURE 3 provides a process or workflow of the screen content solution according to this disclosure.

[0030] Basically, a high-efficiency color palette compression method (CPC) is performed on each coding unit (CU). A coding unit is a basic operating unit in HEVC and HEVC RExt, which is a squared block of pixels consisting of three components (i.e., RGB, or YUV, or XYZ).

[0031] At each CU level, the CPC method includes two major steps. First, the processor 12 derives or generates a color palette table in the first step. This table is ordered according to a histogram (i.e., occurrence frequency of each color value), or its actual color intensity, or any arbitrary method in order to increase the efficiency of the following encoding process. Based on the derived color palette table, each pixel in the original CU is converted to its color index within the color palette table. A contribution of this disclosure is technology to efficiently encode, such as using compression, the color palette table and the color index map of each CU into the stream. At the receiver side, the processor 16 parses the compressed bitstream to reconstruct, for each CU, the complete color palette table and the color index map, and then further derive the pixel value at each position by combining the color index and color palette table.

[0032] In an illustrative example of this disclosure, assume a CU with $N \times N$ pixels ($N=8, 16, 32, 64$ for compatibility with HEVC). The CU typically contains three chrominance (chroma) components (i.e., G, B, R, or Y, Cb, Cr, or X, Y Z) at a different sampling ratio (i.e., 4:4:4, 4:2:2, 4:2:0). For simplicity, sequences of 4:4:4 are illustrated in the disclosure. For sequences of 4:2:2 and 4:2:0 videos, chroma upsampling could be applied to obtain the 4:4:4 sequences, or each color component could be processed

independently. Then, the same procedure described in this disclosure can be applied. For 4:0:0 monochrome videos, this can be treated as an individual plane of 4:4:4 without other two planes. All methods for 4:4:4 can be applied directly.

5 Packed or Planar

[0033] This method is shown for the Block CTU or CU in FIGURE 1. First of all, a flag is defined, called the enable_packed_component_flag, for each CU to indicate whether the current CU is processed using packed fashion or conventional
10 planar mode (i.e., G, B, R or Y, U, V components are processed independently). FIGURE 4 illustrates a conventional G, B, R in planar mode (left) to Packed mode (right). YUV or other color format could be processed in the same fashion as exemplified for RGB content.

15 [0034] Both the packed mode and the planar mode have its own advantage and disadvantage. For instance, the planar mode supports parallel color component processing for G/B/R or Y/U/V. However, it might suffer the low coding efficiency. The packed mode can share the header information (such as the color palette
20 table and index map in this disclosure) for this CU among different color components. However, it might break the parallelism. An easy way to decide whether the current CU should be encoded in the packed fashion is to measure rate distortion (R-D) cost. The enable_packed_component_flag is used to signal
25 the encoding mode to the decoder explicitly.

[0035] In addition, to define the enable_packed_component_flag at the CU level for low-level handling, it can be duplicated in slice header or even sequence level (e.g., Sequence Parameter Set or Picture Parameter Set) to allow slice level or sequence level
30 handling, depending on the specific application requirement.

Color palette table and Index Map Derivation

[0036] As shown in FIGURE 1, for Processes 1 and 3, for each CU, pixel locations are transversed and the color palette table

and index map for the subsequent processing is derived. Each distinct color is ordered in the color palette table, depending on either its histogram (i.e., frequency of occurrence), or its intensity, or any arbitrary method in order to increase the efficiency of the following encoding process. For example, if the encoding process uses a differential pulse-code modulation (DPCM) method to code the difference between adjacent pixels, the optimal coding result can be obtained if the adjacent pixels are assigned with adjacent color index in Color palette table.

[0037] After obtaining the color palette table, each pixel is mapped to the corresponding color index to form the index map of the current CU. The processing of index map is described in the subsequent section.

[0038] For a conventional planar CU, each color or chrominance component can have its individual color palette table, such as colorTable_Y, colorTable_U, colorTable_V or colorTable_R, colorTable_G, colorTable_B, naming a few here as an example. Meanwhile, the color palette table for a major component can be derived, such as Y in YUV or G in GBR, and shared for all components. Normally, by this sharing, other color components, other than Y or G, would have some mismatch relative to its original pixel colors from those shared in color palette table. The residual engine (such as HEVC coefficients coding methods) is then applied to encode those mismatched residuals. On the other hand, for a packed CU, a single color palette table is shared among all components.

[0039] A pseudo code is provided to exemplify the color palette table and index map derivation as follows:

```
deriveColorTableIndexMap()  
{  
    deriveColorTable();  
    deriveIndexMap();  
}
```

```
        deriveColorTable(src, cuWidth, cuHeight, maxColorNum)
    {
        // src - input video source in planar or packed
5      mode
        // cuWidth, cuHeight - width and height of
        current CU
        /* maxColorNum - max num of colors allowed in
        color table*/
10      /*transverse */
        //
        //  memset(colorHist, 0,
        (1<<bitDepth)*sizeof(UINT))
        pos=0;
15      cuSize=cuWidth*cuHeight;
        while (pos<cuSize) {
            colorHist[src[pos++]]++;
        }

20      /*just pick non-zero entry in colorHist[] for
        color intensity ordered table*/
        j=0;
        for(i=0;i<(1<<bitDepth);i++)
        {
25      if(colorHist[i]!=0)
            colorTableIntensity[j++] = colorHist[i];
        }
        colorNum=j;

30      /*quicksort for histogram*/

        colorTableHist = quickSort(colorTableIntensity,
        colorNum);
```

```
        /*if maxColorNum >= colorNum, all colors will be
picked*/

        /*if maxColorNum < colorNum, only maxColorNum
5 colors will be picked for colorTableHist. In this
case, all pixels will find its best matched color and
corresponding index with difference (actual pixel and its
corresponding color) coded by the residual engine.*/

        /*Best number of colors in color table could be
10 determined by iterative R-D cost derivation!*/
    }

    deriveIndexMap()
    {
15         pos=0;
        cuSize=cuWidth*cuHeight;
        while ( pos < cuSize)
        {
            minErr=MAX_UINT;
20             for (i=0;i<colorNum;i++)
            {
                err = abs(src[pos] - colorTable[i]);
                if (err<minErr)
                {
25                     minErr = err;
                    idx = i;
                }
            }
            idxMap[pos] = idx;
30         }
    }
```

Color palette table Processing

[0040] For Process 1 in FIGURE 1, color palette table processing involves the processor 12 encoding of the size of a color palette table (i.e., the total number of distinct colors) and each color itself. A majority of the bits are consumed by the encoding of each color in a color palette table. Hence, focus is placed on the color encoding (or encoding of each entry in color palette table).

[0041] The most straightforward method to encode the colors in a color palette table is using the pulse-code modulation (PCM) style algorithm where each color is coded independently. Alternatively, the nearest prediction for a successive color can be applied, and then the prediction delta can be encoded rather than the default color intensity, which is DPCM (differential PCM) style. Both methods can be later entropy encoded using equal probability model or adaptive context model, depending on the trade-off between complexity costs and coding efficiency.

[0042] Here, another advanced scheme is disclosed, called a Neighboring Color palette table Merge, where a color_table_merge_flag is defined to indicate whether the current CU uses the color palette table from its left or upper CU. If not, the current CU will carry the color palette table signaling explicitly. For the merging process, another color_table_merge_direction indicates the merging direction either from upper or from left CU. Of course, the candidates could be more than current upper or left CU, e.g. upper-left, upper-right and etc. However, the upper and left CU are used in this disclosure to exemplify the idea. For any of which, each pixel is compared with the entries in an existing color palette table and assigned an index yielding the least prediction difference (i.e., pixel subtracts the closest color in color palette table) via deriveIdxMap(). For the case where the prediction difference is non-zero, all these residuals are

encoded using the HEVC RExt residual engine. Note that whether using the merging process or not can be decided by the R-D cost.

[0043] There are several ways to generate the neighboring color palette tables for being used in the merging process in coding the current CU. Depending on its implementation, one of them requires updating at both the encoder and the decoder and the other one is an encoder side process only.

[0044] Updating at both the encoder and the decoder: In this method, the color palette table of neighbor CUs are generated upon the available reconstructed pixels, regardless of CU depth, size and etc. For each CU, the reconstructions are retrieved for its neighboring CU at the same size and same depth (assuming the color similarity would be higher in this case). For example, if a current CU is 16x16 with depth = 2, no matter the partition of its neighboring CUs (for example 8x8 with depth = 3 for left CU and 32x32 with depth =1 for upper CU), the pixel offset (=16) will be located from the current CU origin to the left to process the left 16x16 block and to the upper for the upper 16x16 block, as shown in the FIGURE 5. Note that both the encoder and the decoder should maintain this process.

[0045] Constrained Encoder Only Process: for this method, the merging process occurs when a current CU shares the same size and depth as its upper and/or left CU. The color palette tables of the available neighbors are used to derive the color index map of the current CU for subsequent operations. For example, for a current 16x16 CU, if its neighboring CU, i.e., either upper or left placed, are encoded using the color palette table and index method, its color palette table is used for the current CU directly to derive the R-D cost. This merge cost is compared with the case that the current CU derives its color palette table explicitly (as well as other conventional modes existing in the HEVC or HEVC RExt). Whichever produces the less R-D cost is chosen as the final mode to be written into the output bit

stream. As seen, only the encoder is required to experiment/simulate different potential modes. At the decoder side, the color_table_merge_flag and the color_table_merge_direction infer the merge decision and merge direction without requiring additional processing workload.

Color Index Map Processing

[0046] For Process 3 in FIGURE 1, for coding the color index map, a few solutions have been studied, such as RUN mode, RUN and COPY_ABOVE, and adaptive neighbor index prediction. In this disclosure, a 1D string matching approach and its 2D variation is disclosed to encode the index map coding. At each position, it finds its matched point and records the matched distance and length for a 1D string match, or width/height for a 2D string match. For an unmatched position, its index intensity, or delta value between its index intensity and predicted index intensity, is encoded directly.

[0047] Disclosed here is a straightforward 1D search method over the color index map. Referring to FIGURE 6, an index map is parsed from a real word screen content. FIGURE 7 shows a piece of a segment after a 1-D search (i.e., just beginning of this index map).

[0048] On top of this 1-D color index vector, string match is applied. An example of this 1-D string match is given below. For the first position of each index map, such as 14 as shown in FIGURE 7, since there is no buffered reference yet, this very first index is treated as the "unmatched pair", where it is given -1 and 1 to its corresponding distance and length, noted as (dist, len) = (-1, 1). For the 2nd index, again another "14", it is the first index coded as reference, therefore the dist=1. Because there is another "14" at 3rd position, the length is 2, i.e., len=2, (given the every proceed index could be served as the reference immediately for the subsequent index). Moving

forward to the 4th position, encountered is the "17" which has not been seen before. Hence, it is encoded as an unmatched pair again, i.e., (dist, len) = (-1, 1). For the unmatched pair, the flag is encoded (such as the "dist == -1") and followed by the real value of the index (like first appeared "14", "17", "6" and etc). On the other hand, for the matched pairs, the flag is still encoded (such as the "dist != -1"), and followed by the length of the matched string.

[0049] Here is a summary for the encoding procedure using the exemplified index shown in FIGURE 7.

```
dist = -1, len = 1, idx=14 (unmatched)
dist= 1, len = 2 (matched)
dist = -1, len = 1, idx=17 (unmatched)
dist= 1, len = 3 (matched)
dist = -1, len = 1, idx= 6 (unmatched)
dist= 1, len = 25 (matched)
dist= 30, len = 4 (matched) /*for the "17" which appeared
before*/
...
```

[0050] A pseudo code is given for this matched pair derivation, i.e.,

```
Void deriveMatchedPairs ( TComDataCU* pcCU, Pel* pIdx, Pel*
pDist, Pel* pLen, UInt uiWidth, UInt uiHeight)
{
// pIdx is a idx CU bounded within uiWidth*uiHeight

UInt uiTotal = uiWidth*uiHeight;
UInt uiIdx   = 0;
Int j       = 0;
Int len     = 0;
```

15

```
// first pixel coded as itself if there isn't left/upper
buffer
pDist[uiIdx] = -1;
pLen[uiIdx]  =  0;
5  uiIdx++;

while (uiIdx < uiTotal )
{
    len      = 0;
10    dist    = -1;
    for ( j=uiIdx-1; j >= 0; j-- )
    {
        // if finding matched pair, currently exhaustive
search is applied
15        // fast string search could be applied
        if ( pIdx[j] == pIdx[uiIdx] )
        {
            for (len = 0; len < (uiTotal-uiIdx); len++
)
            {
20                if ( pIdx[j+len] != pIdx[len+uiIdx]
)
                {
                    break;
                }
25            }
            if ( len > maxLen ) /*better to change with R-D
decision*/
            {
                maxLen = len;
30                dist    = (uiIdx - j );
            }
        }
    }
}
```

```
        pDist[uiIdx] = dist;
        pLen[uiIdx] = maxLen;
        uiIdx = uiIdx + maxLen;
5         }
    }
```

[0051] The following steps are made when a 2D search variation is used:

- 10 1. Identify the location of current pixel and reference pixel as starting point,
2. Apply a horizontal 1D string match to the right direction of current pixel and reference pixel. Maximum search length is constrained by the end of current horizontal row. Record the maximum search length as
15 right_width
3. Apply a horizontal 1D string match to the left direction of current pixel and reference pixel. Maximum search length is constrained by the beginning of current horizontal row. Record the maximum search length as
20 left_width
4. Perform same 1D string match at next row, using pixels below current pixel and reference pixel as new current pixel and reference pixel
- 25 5. Stop until right_width == left_width == 0.
6. Now for each height[n] = {1, 2, 3...}, there is a corresponding array of width[n] {{left_width[1], right_width[1]}, {left_width[2], right_width[2]}, {left_width[3], right_width[3]}...}
- 30 7. Define a new min_width array {{lwidth[1], rwidth[1]}, {lwidth[2], rwidth[2]}, {lwidth[3], rwidth[3]}...} for each height[n], where lwidth[n] = min(left_width[1:n-1]), rwidth[n] = min(right_width[1:n-1])

8. A size array{size[1], size[2], size[3]...} is also defined, where $\text{size}[n] = \text{height}[n] \times (\text{lwidth}[n] + \text{hwidth}[n])$

9. Assume size[n] holds the maximum value in size array, the width and height of 2D string match is selected using the corresponding {lwidth[n], rwidth[n], height[n]}

[0052] One way to optimize the speed of the 1D or 2D search is to use running hash. A 4-pixel running hash structure is described in this disclosure. A running hash is calculated for every pixel at horizontal direction to generate a horizontal hash array running_hash_h[]. Another running hash is calculated on top of running_hash_h[] to generate a 2D hash array running_hash_hv[]. Each value match in this 2D hash array represents a 4x4 block match. To perform a 2D match, as many as 4x4 block matches are to be found before performing pixel-wised comparison to their neighbors. Since pixel wised comparison is limited to 1-3 pixels, the search speed can be increased dramatically.

[0053] From the above description, the matched widths of each row are different from each other, thus each row has to be processed separately. To achieve efficiency and low complexity, a block based algorithm is disclosed, which can be used in both a hardware and software implementation. Much similar to standard motion estimation, this algorithm processes one rectangle block at a time.

[0054] Take a 4x4 block as example. The basic unit in this design is called U_PIXEL, as shown in FIGURE 8. The coded signal is a flag that indicates if the reference pixel has already been encoded from previous string match operation. Optionally, the input signal Cmp[n-1] can be forced to "0", which allows removal of the last "OR" gate from U_PIXEL module.

[0055] The first step is to process each row in parallel. Each pixel in one row of the rectangle is assigned to one U_PIXEL

block; this processing unit is called U_ROW. An example of the processing unit for the first row is shown in FIGURE 9.

[0056] 4 U_ROW units are needed to process this 4x4 block, as shown in FIGURE 10. Its output is an array of cmp[4][4].

5 [0057] The next step is to process each column of the cmp array in parallel. Each cmp in a column of the cmp array is processed by processing unit U_COL, as shown in FIGURE 11.

[0058] 4 U_COL units are needed to process this 4x4 block. Its output is an array of rw[4][4] as shown in FIGURE 12.

10 [0059] The number of zeros in each row of rw[n][0-3] is then counted and the 4 results are recorded to array r_width[n]. It is noted r_width[n] equals to rwidth[n] in step #7. l_width[n] is generated in the same fashion. The min_width array in step #7 can be obtained as {{l_width[1], r_width[1]}, { l_width[2], r_width[2]}, { l_width[3], r_width[3]}}...

[0060] This hardware architecture can be modified to fit in the parallel processing framework of any modern CPU/DSP/GPU. A simplified pseudo-code for fast software implementation is listed below.

```

20      // 1. Generate array C[][]
      For(y = 0; y < height; ++y)
      {
          For(x = 0; x < width; ++x)
          {
25              tmp1 = cur_pixel ^ ref_pixel;
              tmp2 = tmp1[0] | tmp1[1] | tmp1[2] | tmp1[3] |
tmp1[4] | tmp1[5] | tmp1[6] | tmp1[7];
              C[y][x] = tmp2 & (!coded[y][x]);
          }
30      }

      // 2. Generate array CMP[][]
      For(y = 0; y < height; ++y)

```

```
{
    CMP[y][0] = C[y][0];
}
For(x = 1; x < width; ++x)
5   {
    For(y = 0; y < height; ++y)
    {
        CMP[y][x] = C[y][x] | CMP[y][x-1]
    }
10  }

// 3. Generate array RW[][] or LW[][]
For(x = 0; x < width; ++x)
{
15    RW[0][x] = CMP[0][x];
}
For(y = 1; y < height; ++y)
{
    For(x = 0; x < width; ++x)
20    {
        RW[y][x] = CMP[y][x] | RW[y-1][x];
    }
}

25 // 4. Convert RW[][] to R_WIDTH[]
For(y = 0; y < height; ++y)
{
    // count zero, or leading zero detection
    R_WIDTH[y] = LZD(RW[y][0], RW[y][1], RW[y][2],
30  RW[y][3]);
}
```

[0061] There is no data dependence in each loop, so a traditional software parallel processing method, such as loop unrolling, MMX/SSE, can be applied to increase the execution speed.

5 [0062] This method can also apply to a 1D search if the number of rows is limited to one. A simplified pseudo-code for fast software implementation of fix length based 1D search is listed below.

```

// 1. Generate array C[]
10 For(x = 0; x < width; ++x)
{
    tmp1 = cur_pixel ^ ref_pixel;
    tmp2 = tmp1[0] | tmp1[1] | tmp1[2] | tmp1[3] | tmp1[4]
| tmp1[5] | tmp1[6] | tmp1[7];
15 C[x] = tmp2 & (!coded[x]);
}

// 2. Generate array RW[] or LW[]
If (last "OR" operation in U_PIXEL module is removed)
20 Assign RW[] = C[]
Else {
    RW [0] = C[0];
    For(x = 1; x < width; ++x)
    {
25 RW [x] = C[x] | RW [x-1]
    }
}

// 3. Convert RW[][] to R_WIDTH[]
// count zero, or leading zero detection
30 If(last "OR" operation in U_PIXEL module is removed)
    R_WIDTH = LZD(RW[0], RW[1], RW[2], RW[3]);
Else
    R_WIDTH[y] = COUNT_ZERO(RW[0], RW[1], RW[2], RW[3]);

```

[0063] After both 1D match and 2D match is completed, max (1d length, 2d (width x height)) is chosen as the winner. If the lwidth of 2D match is non-zero, the length of the prior 1D match (length = length - lwidth) needs to be adjusted to avoid the
5 overlap between prior 1D match and current 2D match. If the length of the prior 1D match becomes zero after adjustment, it is removed from the match list.

[0064] The next starting location is calculated using current_location + length if the previous match is a 1D match, or
10 current_location + (lwidth+rwidth) if the previous match is a 2D match. When a 1D match is performed, if any to-be-matched-pixel falls into any previous 2D match region where its location has already been covered by a 2D match, the next pixels will be scanned through until a pixel location is found where it has not
15 been coded by previous match.

[0065] After obtaining these matched pairs, an entropy engine is applied to convert these symbols into the binary stream. Exemplified here is the idea of using the equal probability context mode. An advanced adaptive context mode could be applied
20 as well for better compression efficiency.

```
// loop for each CU, uiTotal=uiWidth*uiHeight, uiIdx=0;
while ( uiIdx < uiTotal) {
    // *pDist: store the distance value for each matched
    pair
25    // *pIdx: store the index value for each matched pair
    // *pLen: store the length value for each matched pair
    // encodeEP() and encodeEPs() are reusing HEVC or
    similar by-pass entropy coding.

30    if (pDist[uiIdx] == -1 )
    {
        //encode one-bin with equal-probability model to
        indicate the
```

```
        //whether current pair is matched or not.
        unmatchedPairFlag = TRUE;
        encodeEP(unmatchedPairFlag);
        //uiIndexBits is controlled by the color table
5      size
        // i.e., for 24 different colors, we need 5 bits,
        for 8 colors, 3 bits
        encodeEPs(pIdx[uiIdx], uiIndexBits);
        uiIdx++;
10      }
      else
      {
        unmatchedPairFlag= FALSE;
        encodeEP(unmatchedPairFlag);
15
        /*bound binarization with max possible value*/
        UInt uiDistBits =0;
        // offset is used to add additional references
        from neighboring blocks
20      // here, we first let offset=0;
        while( (1<<uiDistBits)<= (uiIdx+offset))
        {
            uiDistBits++;
        }
25      encodeEPs(pDist[uiIdx], uiDistBits);

        /*bound binarization with max possible value*/
        UInt uiLenBits =0;
        while( (1<<uiLenBits)<= (uiTotal-uiIdx))
30      {
            uiLenBits++;
        }
        encodeEPs(pLen[uiIdx], uiLenBits);
```

```
        uiIdx += pLen[uiIdx];  
    }  
}
```

5 [0066] Shown is the encoding procedure for each matched pair. Correspondingly, the decoding process for the matched pair is as follows.

```
    // loop for each CU, uiTotal=uiWidth*uiHeight, uiIdx=0;  
    while ( uiIdx < uiTotal) {  
10        // *pDist: store the distance value for each matched  
        pair  
        // *pIdx: store the index value for each matched pair  
        // *pLen: store the length value for each matched pair  
        // parseEP() and parseEPs() are reusing HEVC or  
15        similar by-pass entropy coding.  
  
        // parse the unmatched pair flag  
        parseEP(&uiUnmatchedPairFlag);  
  
20        if (uiUnmatchedPairFlag )  
        {  
            parseEPs( uiSymbol, uiIndexBits );  
            pIdx[uiIdx] = uiSymbol;  
            uiIdx++;  
25        }  
        else  
        {  
            /*bound binarization with max possible value*/  
            UInt uiDistBits =0;  
30            // offset is used to add additional references  
            from neighboring blocks  
            // here, we first let offset=0;  
            while( (1<<uiDistBits)<= (uiIdx+offset))
```

```

        uiDistBits++;
        UInt uiLenBits =0;
        while( (1<<uiLenBits)<= (uiTotal-uiIdx))
            uiLenBits++;
5
        parseEPs( uiSymbol, uiDistBits);
        pDist[uiIdx] = uiSymbol;
        parseEPs( uiSymbol, uiLenBits);
        pLen[uiIdx] = uiSymbol;
10
        for(UInt i=0; i< pLen[uiIdx]; i++)
            pIdx[i+uiIdx] = pIdx[i+uiIdx- pDist[uiIdx]];
        uiIdx += pLen[uiIdx];
    }
15    }

```

[0067] Note that only pixels in an unmatched position are encoded into a bit stream. To have a more accurate statistics modal, use only these pixels and their neighbors for Color palette table Derivation, instead of using all pixels in this CU.

[0068] For these index or delta output, they usually contain limited number of unique value under certain encoding mode. This disclosure introduces a second delta palette table to utilize this observation. This delta palette table can be built after all literal data are obtained in this CU, it will be signaled explicitly in the bit stream. Alternatively, it can be built adaptively during the coding process, so that the table does not have to be included in the bit stream. A delta_color_table_adaptive_flag is defined for this choice.

[0069] Another advanced scheme is provided, called Neighboring Delta Color palette table Merge. For adaptive delta palette generation, an encoder can use a delta palette from top or left CU as the initial starting point. For non-adaptive palette

generation, the encoder can also use a delta palette from top or left CU and compare the RD cost among top, left and current CU.

[0070] A delta_color_table_merge_flag is defined to indicate whether a current CU uses the delta color palette table from its left or upper CU. A current CU carries the delta color palette table signaling explicitly only when delta_color_table_adaptive_flag==0 and delta_color_table_merge_flag==0 at the same time.

[0071] For a merging process, if delta_color_table_merge_flag is asserted, another delta_color_table_merge_direction is defined to indicate whether the merge candidate is from either upper or left CU.

[0072] An example of an encoding process for an adaptive delta palette generation is shown as follows. At a decoding side, whenever a decoder receives a literal data, it regenerates a delta palette based on reverse steps.

10. Define palette_table[] and palette_count[]

11. Initialize palette_table(n) = n (n = 0...255), alternatively, it can use palette_table[] from top or left CU as initial value

12. Initialize palette_count(n) = 0 (n = 0...255), alternatively, it can use palette_count[] from top or left CU as initial value

13. For any delta value c':

1) Locate n so that palette_table(n) == delta c'

2) Use n as the new index of delta c'

3) ++palette_count(n)

4) Sort palette_count[] so that it is in descendent order

5) Sort palette_table[] accordingly

14. Go back to step 1 until all delta c' in current LCU are processed

[0073] For any block that includes both text and graphics, a mask flag is used to separate the text section and graphics

section. The text section is compressed by the above described method; the graphics section is compressed by another compression method.

[0074] Note that because the value of any pixel covered by the mask flag has been coded by a text layer losslessly, these pixels in graphics section can be as "don't-care-pixel". When the graphics section is compressed, any arbitrary value can be assigned to a don't-care-pixel in order to obtain optimal compression efficiency.

[0075] Since the lossy part could be handled by the color palette table derivation, the index map has to be compressed losslessly. This allows the efficient processing using a 1D or a 2D string match. For this disclosure, the 1D or the 2D string match is constrained at current LCU, but the search window can extend beyond the current LCU. Also note that the matched distance can be encoded using a pair of motion vector in horizontal and vertical directions, i.e., $(MV_x = \text{matched_distance} / \text{cuWidth}, MV_y = \text{matched_distance} - \text{cuWidth} * MV_x)$.

[0076] Given that image would have a different spatial texture orientation at local regions, the 1D search can be allowed in either horizontal or vertical directions by defining the color_idx_map_pred_direction indicator. The optimal index scanning direction can be made based on the R-D cost. FIGURE 6 shows the scanning directions, starting from the very first position. Further illustrated is the horizontal and vertical scanning pattern in FIGURE 9. Consider an 8x8 CU as an example. The deriveMatchPairs() and associated entropy coding steps are performed twice for both the horizontal and the vertical scanning pattern. Then, the final scanning direction is chosen with the smallest RD cost.

Improved Binarization

[0077] As shown in FIGURE 13, the color palette table and a pair of matched information for the color index map are encoded.

They are encoded using fixed length binarization. Alternatively, variable-length binarization can be used.

[0078] For example, as for the color palette table encoding, the table can have 8 different color values. Therefore, it only contains 8 different indices in the color index map. Instead of using fixed 3 bins to encode every index value equally, just one bit can be used to represent the background pixel, for instance 0. Then, the rest of 7 pixel values use a fixed-length codeword, such as 1000, 1001, 1010, 1011, 1100, 1101, and 1110 to encode the color index. This is based on the fact that the background color may occupy the largest percentile and therefore a special codeword for it saves the total bins. This scenario happens commonly for screen content. Consider a 16x16 CU, for fixed 3-bin binarization, it requires $3 \times 16 \times 16 = 768$ bins. Also, let 0 index be background color, occupying 40%, while other colors are equally distributed. In this case, it only requires $2.8 \times 16 \times 16 < 768$ bins.

[0079] For the matched pair encoding, the max value can be used to bound its binarization, given the constrained implementation of this approach within the area of the current CU. Mathematically, the matched distance and length could be as long as $64 \times 64 = 4K$ in each case. However, this wouldn't be happening jointly. For every matched position, the matched distance is bounded by the distance between current position and the very first position in the reference buffer (such as the first position in current CU as an example), for instance L. Therefore, the maximum bins for this distance binarization is $\log_2(L) + 1$ (instead of fixed length), and the maximum bins for the length binarization is $\log_2(\text{cuSize} - L) + 1$ with $\text{cuSize} = \text{cuWidth} \times \text{cuHeight}$.

[0080] In addition to the color palette table and index map, the residual coding could be significantly improved by a different binarization method. As for HEVC RExt and HEVC version, transform coefficient is binarization using the variable

length codes at the assumption that the residual magnitude should be small after prediction, transform and quantization. However, after introducing the transform skip, especially for the transform skip on the screen content with distinct color, there commonly exists residuals with larger and random value (not close to "1", "2", "0" relative smaller value). If the current HEVC coefficients binarization are used, it turns out to yield a very long code word. Alternatively, using the fixed length binarization saves the code length for the residuals produced by the color palette table and index coding mode.

[0081] Adaptive chroma sampling for mixed content

[0082] The foregoing provides various techniques for high-efficiency screen content coding under the framework of the HEVC/HEVC-RExt. In practice, in addition to pure screen content (such as text, graphics) or pure natural video, there is also content containing both screen material and camera-captured natural video -- called mixed content. Currently, mixed content is treated with 4:4:4 chroma sampling. However, for the embedded camera-captured natural video portion in such mixed content, the 4:2:0 chroma sampling may be sufficient to provide perceptual lossless quality. This is due to the fact that the human vision system is less sensitive to the spatial changes in chroma components compared with that from the luma components. Hence, sub-sampling typically is performed on the chroma part (e.g., the popular 4:2:0 video format) to achieve noticeable bit rate reduction while maintaining same reconstructed quality.

[0083] The present disclosure provides a new flag (i.e., enable_chroma_subsampling) that is defined and signaled at the CU level recursively. For each CU, the encoder determines whether it is being coded using 4:2:0 or 4:4:4 according to the rate-distortion cost.

[0084] Shown in FIGURE 14A and FIGURE 14B are the 4:2:0 and 4:4:4 chroma sampling formats.

[0085] At the encoder side, for each CU, assuming the input is 4:4:4 source shown above, the rate-distortion cost is derived directly using the 4:4:4 encoding procedure with enable_chroma_subsampling = 0 or FALSE. Then, the process sub-samples 4:4:4 samples to 4:2:0 to derive its bit consumption. The reconstructed 4:2:0 format is interpolated back to the 4:4:4 format for distortion measurement (using SSE/SAD). Together with the bit consumption, the rate-distortion cost is derived when encoding the CU at 4:2:0 space and comparing it with the cost when encoding the CU at 4:4:4. Whichever encoding gives the less rate-distortion cost will be chosen for the final encoding.

[0086] Illustrated in FIGURE 15 is the interpolation process from 4:4:4 to 4:2:0 and vice versa. Usually this video color sampling format conversion process requires a large number of interpolation filters.

[0087] To reduce the implementation complexity, an HEVC interpolation filter (i.e., DCT-IF) may be utilized. As shown in FIGURE 15, the "squared box" represents the original 4:4:4 samples. From 4:4:4 to 4:2:0, the half-pel pixels ("circle") are interpolated using DCT-IF vertically for the chroma components. Also shown are the quarter-pel positions ("diamond") for illustration purposes. The grey shaded "circles" are picked to form the 4:2:0 samples. For the interpolation from 4:2:0 to 4:4:4, the process starts with the grey "circles" in the chroma components, the half-pel positions are interpolated horizontally to obtain all "circles," and then the "squared box" are interpolated using DCT-IF vertically. All the interpolated "squared box" are chosen to form the reconstructed 4:4:4 source.

[0088] Encoder Control

[0089] As discussed in the previous sections, disclosed are flags to control the low-level processing. For instance, enable_packed_component_flag is used to indicate whether current CU uses its packed format or conventional planar format for

encoding the processing. Whether to enable a packed format could depend on the R-D cost calculated at the encoder. For a practical encoder implementation, a low-complexity solution is achieved by analyzing the histogram of the CU and finding the best threshold
5 for the decision, as shown in FIGURE 3.

[0090] The size of the color palette table has a direct impact on the complexity. maxColorNum is introduced to control the trade-off between complexity and coding efficiency. The most straightforward way is choosing the one yielding the least R-D
10 cost.

[0091] Index map encoding direction could be determined by the R-D optimization, or using the local spatial orientation (such as sobel operator based direction estimation).

[0092] This disclosure limits the processing within every
15 CTU/CU. In practice, this constraint can be relaxed. For example, for a color index map processing, the line buffer from its upper and left CU can be used, as shown in the FIGURE 16. With an upper and a left buffer, the search can be extended to further improve the coding efficiency. Given that upper/left buffers are formed
20 using the reconstructed pixels from neighboring CUs, these pixels (as well as its corresponding indices) are available for reference before processing current CU index map. For instance, after re-ordering, the current CU index map could be 14, 14, 14, ... 1, 2, 1 (as 1D presentation). Without a line buffer
25 reference, the first "14" will be coded as an unmatched pair. However, with a neighboring line buffer, the string match can start at the very first pixel, as shown below (horizontal and vertical scanning patterns are shown as well).

[0093] Decoder Syntax

30 [0094] The following information can be used to describe the decoder shown in FIGURE 2. The syntax of this disclosure is aligned with a committee draft of HEVC RExt.

[0095] 7.3.5.8 Coding unit syntax:

coding_unit(x0, y0, log2CbSize) {	Descriptor
if(transquant_bypass_enabled_flag)	
cu_transquant_bypass_flag	ae(v)
if(slice_type != I)	
cu_skip_flag[x0][y0]	ae(v)
nCbS = (1 << log2CbSize)	
if(cu_skip_flag[x0][y0])	
prediction_unit(x0, y0, nCbS, nCbS)	
else {	
if(intra_block_copy_enabled_flag)	
intra_bc_flag[x0][y0]	ae(v)
if(color_table_enabled_flag)	
color_table_flag[x0][y0]	ae(v)
if(delta_color_table_enabled_flag)	
delta_color_table_flag[x0][y0]	ae(v)
if(!intra_bc_flag[x0][y0]) {	
if(slice_type != I)	
pred_mode_flag	ae(v)
if(CuPredMode[x0][y0] != MODE_INTRA	
log2CbSize == MinCbLog2SizeY)	
part_mode	ae(v)
}	
if(CuPredMode[x0][y0] == MODE_INTRA	
) {	
if(PartMode == PART_2Nx2N &&	
pcm_enabled_flag && !intra_bc_flag	
log2CbSize >= Log2MinIpcmCbSizeY &&	
log2CbSize <= Log2MaxIpcmCbSizeY)	
pcm_flag[x0][y0]	ae(v)
if(pcm_flag[x0][y0]) {	
while(!byte_aligned())	

pcm_alignment_zero_bit	f(1)
pcm_sample(x0, y0, log2CbSize)	
} else if(intra_bc_flag[x0][y0]) {	
mvd_coding(x0, y0, 2)	
} else if(color_table_flag[x0][y0] delta_color_table_flag[x0][y0]) {	
enable_packed_component_flag	ae(v)
if(color_table_flag[x0][y0]) {	
color_table_merge_flag	ae(v)
if (color_table_merge_flag){	
color_table_merge_idx	ae(v)
}else{	
color_table_size	ae(v)
for(i=0;i< color_table_size;i++)	
color_table_entry[i]	ae(v)
}	
color_idx_map_pred_direction	ae(v)
}	
if(delta_color_table_flag[x0][y0]) {	
delta_color_table_adaptive_flag	ae(v)
delta_color_table_merge_flag	ae(v)
if (delta_color_table_merge_flag){	
delta_color_table_merge_idx	ae(v)
}else if (!delta_color_table_adaptive_flag){	
delta_color_table_size	ae(v)
for(i=0;i< delta_color_table_size;i++)	
delta_color_table_entry[i]	ae(v)
}	
}	

Pos=0; cuWidth=1<<log2CbSize; cuHeight=1<<log2CbSize; while (Pos<cuWidth*cuHeight){	
matched_flag	ae(v)
if(matched_flag) {	
matched_distance /*MVx, MVy*/	ae(v)
matched_length	ae(v)
}else{	
index_delta	ae(v)
}	
}	
} else {	
....pbOffset = (PartMode == PART_NxN) ? (nCbS / 2) : nCbS	
...	

[0096] FIGURE 17 illustrates the apparatus and methods/flows incorporated into the current HEVC.

[0097] The above identified methods/flows and devices may be incorporated into a wireless or wired, or combination thereof, communications network and implemented in devices, such as that described below, and in the drawings below:

[0098] FIGURE 18 illustrates an example communication system 100 that uses signaling to support advanced wireless receivers according to this disclosure. In general, the system 100 enables multiple wireless users to transmit and receive data and other content. The system 100 may implement one or more channel access methods, such as code division multiple access (CDMA), time division multiple access (TDMA), frequency division multiple access (FDMA), orthogonal FDMA (OFDMA), or single-carrier FDMA (SC-FDMA).

[0099] In this example, the communication system 100 includes user equipment (UE) 110a-110c, radio access networks (RANs) 120a-120b, a core network 130, a public switched telephone network (PSTN) 140, the Internet 150, and other networks 160. While
5 certain numbers of these components or elements are shown in FIGURE 18, any number of these components or elements may be included in the system 100.

[00100] The UEs 110a-110c are configured to operate and/or communicate in the system 100. For example, the UEs 110a-110c are
10 configured to transmit and/or receive wireless signals or wired signals. Each UE 110a-110c represents any suitable end user device and may include such devices (or may be referred to) as a user equipment/device (UE), wireless transmit/receive unit (WTRU), mobile station, fixed or mobile subscriber unit, pager,
15 cellular telephone, personal digital assistant (PDA), smartphone, laptop, computer, touchpad, wireless sensor, or consumer electronics device.

[00101] The RANs 120a-120b here include base stations 170a-170b, respectively. Each base station 170a-170b is configured to
20 wirelessly interface with one or more of the UEs 110a-110c to enable access to the core network 130, the PSTN 140, the Internet 150, and/or the other networks 160. For example, the base stations 170a-170b may include (or be) one or more of several well-known devices, such as a base transceiver station (BTS), a
25 Node-B (NodeB), an evolved NodeB (eNodeB), a Home NodeB, a Home eNodeB, a site controller, an access point (AP), or a wireless router, or a server, router, switch, or other processing entity with a wired or wireless network.

[00102] In the embodiment shown in FIGURE 18, the base station
30 170a forms part of the RAN 120a, which may include other base stations, elements, and/or devices. Also, the base station 170b forms part of the RAN 120b, which may include other base stations, elements, and/or devices. Each base station 170a-170b

operates to transmit and/or receive wireless signals within a particular geographic region or area, sometimes referred to as a "cell." In some embodiments, multiple-input multiple-output (MIMO) technology may be employed having multiple transceivers
5 for each cell.

[00103] The base stations 170a-170b communicate with one or more of the UEs 110a-110c over one or more air interfaces 190 using wireless communication links. The air interfaces 190 may utilize any suitable radio access technology.

10 [00104] It is contemplated that the system 100 may use multiple channel access functionality, including such schemes as described above. In particular embodiments, the base stations and UEs implement LTE, LTE-A, and/or LTE-B. Of course, other multiple access schemes and wireless protocols may be utilized.

15 [00105] The RANs 120a-120b are in communication with the core network 130 to provide the UEs 110a-110c with voice, data, application, Voice over Internet Protocol (VoIP), or other services. Understandably, the RANs 120a-120b and/or the core network 130 may be in direct or indirect communication with one
20 or more other RANs (not shown). The core network 130 may also serve as a gateway access for other networks (such as PSTN 140, Internet 150, and other networks 160). In addition, some or all of the UEs 110a-110c may include functionality for communicating with different wireless networks over different wireless links
25 using different wireless technologies and/or protocols.

[00106] Although FIGURE 18 illustrates one example of a communication system, various changes may be made to FIGURE 18. For example, the communication system 100 could include any number of UEs, base stations, networks, or other components in
30 any suitable configuration, and can further include the EPC illustrated in any of the figures herein.

[00107] FIGURES 19A and 19B illustrate example devices that may implement the methods and teachings according to this disclosure.

In particular, FIGURE 19A illustrates an example UE 110, and FIGURE 19B illustrates an example base station 170. These components could be used in the system 100 or in any other suitable system.

5 [00108] As shown in FIGURE 19A, the UE 110 includes at least one processing unit 200. The processing unit 200 implements various processing operations of the UE 110. For example, the processing unit 200 could perform signal coding, data processing, power control, input/output processing, or any other
10 functionality enabling the UE 110 to operate in the system 100. The processing unit 200 also supports the methods and teachings described in more detail above. Each processing unit 200 includes any suitable processing or computing device configured to perform one or more operations. Each processing unit 200 could, for
15 example, include a microprocessor, microcontroller, digital signal processor, field programmable gate array, or application specific integrated circuit.

[00109] The UE 110 also includes at least one transceiver 202. The transceiver 202 is configured to modulate data or other
20 content for transmission by at least one antenna 204. The transceiver 202 is also configured to demodulate data or other content received by the at least one antenna 204. Each transceiver 202 includes any suitable structure for generating signals for wireless transmission and/or processing signals
25 received wirelessly. Each antenna 204 includes any suitable structure for transmitting and/or receiving wireless signals. One or multiple transceivers 202 could be used in the UE 110, and one or multiple antennas 204 could be used in the UE 110. Although shown as a single functional unit, a transceiver 202 could also
30 be implemented using at least one transmitter and at least one separate receiver.

[00110] The UE 110 further includes one or more input/output devices 206. The input/output devices 206 facilitate interaction

with a user. Each input/output device 206 includes any suitable structure for providing information to or receiving information from a user, such as a speaker, microphone, keypad, keyboard, display, or touch screen.

5 [00111] In addition, the UE 110 includes at least one memory 208. The memory 208 stores instructions and data used, generated, or collected by the UE 110. For example, the memory 208 could store software or firmware instructions executed by the processing unit(s) 200 and data used to reduce or eliminate
10 interference in incoming signals. Each memory 208 includes any suitable volatile and/or non-volatile storage and retrieval device(s). Any suitable type of memory may be used, such as random access memory (RAM), read only memory (ROM), hard disk, optical disc, subscriber identity module (SIM) card, memory
15 stick, secure digital (SD) memory card, and the like.

[00112] As shown in FIGURE 19B, the base station 170 includes at least one processing unit 250, at least one transmitter 252, at least one receiver 254, one or more antennas 256, and at least one memory 258. The processing unit 250 implements various
20 processing operations of the base station 170, such as signal coding, data processing, power control, input/output processing, or any other functionality. The processing unit 250 can also support the methods and teachings described in more detail above. Each processing unit 250 includes any suitable processing or
25 computing device configured to perform one or more operations. Each processing unit 250 could, for example, include a microprocessor, microcontroller, digital signal processor, field programmable gate array, or application specific integrated circuit.

30 [00113] Each transmitter 252 includes any suitable structure for generating signals for wireless transmission to one or more UEs or other devices. Each receiver 254 includes any suitable structure for processing signals received wirelessly from one or

more UEs or other devices. Although shown as separate components, at least one transmitter 252 and at least one receiver 254 could be combined into a transceiver. Each antenna 256 includes any suitable structure for transmitting and/or receiving wireless signals. While a common antenna 256 is shown here as being coupled to both the transmitter 252 and the receiver 254, one or more antennas 256 could be coupled to the transmitter(s) 252, and one or more separate antennas 256 could be coupled to the receiver(s) 254. Each memory 258 includes any suitable volatile and/or non-volatile storage and retrieval device(s).

[00114] Additional details regarding UEs 110 and base stations 170 are known to those of skill in the art. As such, these details are omitted here for clarity.

[00115] It may be advantageous to set forth definitions of certain words and phrases used throughout this patent document. The terms "include" and "comprise," as well as derivatives thereof, mean inclusion without limitation. The term "or" is inclusive, meaning and/or. The phrases "associated with" and "associated therewith," as well as derivatives thereof, mean to include, be included within, interconnect with, contain, be contained within, connect to or with, couple to or with, be communicable with, cooperate with, interleave, juxtapose, be proximate to, be bound to or with, have, have a property of, or the like.

[00116] While this disclosure has described certain embodiments and generally associated methods, alterations and permutations of these embodiments and methods will be apparent to those skilled in the art. Accordingly, the above description of example embodiments does not define or constrain this disclosure. Other changes, substitutions, and alterations are also possible without departing from the spirit and scope of this disclosure, as defined by the following claims.

Claims

1. A method for coding screen content into a bitstream, the method comprising:
 - selecting a color palette table for a coding unit (CU) of screen content;
 - creating a color index map having indices for the CU using the selected color palette table; and
 - encoding the selected color palette table and the color index map for the CU into a bitstream, and
 - wherein the color palette table is derived from a neighboring CU using a reconstructed CU in a pixel domain.
2. The method as specified in Claim 1 wherein the method is processed using a planar color format or an interleaved color format.
3. The method as specified in Claim 2 wherein the method is processed at a level selected from the group of: a CU level, a slice level, a picture level or a sequence level.
4. The method as specified in Claim 1 further comprising generating a color palette table of the neighboring CU based on available reconstructed pixels, regardless of the CU depth and size.
5. The method as specified in Claim 1 further comprising generating a color palette table of the neighboring CU, wherein the neighboring CU is encoded using a color palette mode.
6. The method as specified in Claim 1 wherein the neighboring CU color palette table is encoded using a color palette mode.
7. The method as specified in Claim 1 wherein the color palette table is derived in a pixel domain at a decoder, wherein the encoded bitstream is parsed to reconstruct, for the CU, the color palette table and the color index map.
8. The method as specified in Claim 4 wherein a pixel value is derived at each position in the CU by combining the color index, and the color palette table.
9. The method as specified in Claim 1 further comprising classifying colors or pixel values of the CU based on a previous derived color palette table for corresponding indices.
10. The method as specified in Claim 1 wherein the color palette table is generated and ordered according to a histogram.
11. The method as specified in Claim 1 wherein each pixel of the CU is converted into a color index within the color palette table.
12. The method as specified in Claim 2 wherein a flag is defined for the CU to indicate whether the CU is processed using a packed fashion or a planar mode.

13. The method as specified in Claim 1 wherein the color palette table is processed by encoding a size of the color palette table and each color in the color palette table.

14. The method as specified in Claim 1 further comprising generating a flag indicating the CU uses the color palette table from its left or upper CU.

15. The method as specified in Claim 1 wherein the color index map is encoded using a string match selected from the group comprising: one dimensional (1-D) string match, a hybrid 1-D string match, and a two dimensional (2-D) string match, wherein the string match is signaled using matched pairs.

16. The method as specified in Claim 15 wherein the string match is performed using a running hash method.

17. The method as specified in Claim 1 wherein a two dimensional (2-D) string match is performed over the color index map by identifying a location of a current pixel and a location of reference pixel in the CU as a starting point.

18. The method as specified in Claim 1 wherein the CU has a 4:4:4 format processed using a down-sampled 4:2:0 sampling format.

19. The method as specified in Claim 18 wherein the down-sampled format is processed at a level selected from the group of: a CU level, a slice level, a picture level or a sequence level.

20. A processor for coding screen content into a bitstream, the processor configured to:

select a color palette table for a coding unit (CU) of screen content;

create a color index map having indices for the CU using the selected color palette table; and

encode the selected color palette table and the color index map for the CU into a bitstream, and

wherein the color palette table is derived from a neighboring CU using a reconstructed CU in a pixel domain.