Octree Colour Quantization Algorithm

mgr inż. Paweł Aszklar
pawel.aszklar@pw.edu.pl

Warsaw, April 3, 2022

1 Introduction

The goal of colour quantization is to select a palette of limited number of colours that can be used to approximate the original image as close as possible. The algorithm presented here employs an octree data structure where each node can have up to eight children. The algorithm aims to minimize memory usage and therefore ensures that the number of leaves in the tree doesn't exceed the intended palette size at all times throughout the algorithm execution.

2 Data Structures

Before describing the algorithm let's introduce the data structures that it will use. We will assume that each pixel of the processed image stores a triplet of intensities for each channel: red (R), green (G), and blue (B). Those intensities are stored as a one-byte integer value without sign. Pseudo-code with a data structure capable of storing such representation of colour is presented in listing 1.

Listing 1: RGB colour structure

```
struct COLOR {
   r : UINT8
   g : UINT8
   b : UINT8
}
```

2.1 Tree Nodes

Any node in the octree can either be a leaf or an internal node. An internal node should be able to store references to up to 8 children, although not all nodes might exist, see. section 3.1.2. Leaf nodes on the other hand

will store total of per-channel intensities of pixels assigned to them as well as their count, see. section 3.1.3. Listing 2 presents a data structure capable of representing both internal and leaf nodes.

Listing 2: Octree node structure

```
struct NODE {
    isLeaf   : BOOL
    sumR     : UINT
    sumG     : UINT
    sumB     : UINT
    count    : UINT
    children : NODE*[8]
```

2.2 Octree

Besides the reference to the root node, the octree can store additional data that will simplify the implementation of the algorithm: maximum palette size, current leaf count, and array of lists of internal (non-leaf) nodes at each depth. The array of lists will allow us to easily find internal nodes to reduce whenever the leaf count exceeds the maximum palette size, see. section 3.2. The array contains eight lists, since that is the maximum depth of any internal node, see. section 3.1.2. Listing 3 presents the pseudo-code for octree data structure.

Listing 3: Octree structure

```
struct OCTREE {
    root       : NODE*
    leafCount  : UNIT8
    maxLeaves  : UINT8
    innerNodes : LIST<NODE*>[8]
```

3 Octree Creation

In the first stage the algorithm constructs the octree. The colour of each pixel in the image is sequentially added to the octree structure. After each pixel is added, we must ensure the number of leaves has not exceeded the maximum palette size — if it did, the tree must be reduced. Single iteration of tree reduction procedure does not guarantee that the number of leaves decreases — it removes 1 to 8 leaves but always adds 1. Therefore, the procedure might need to be repeated multiple times until the leaf count is brought down to or below the maximum. Only after the reduction the

algorithm can continue, inserting the colour of the next pixel from the image into the tree. Listing 4 presents the algorithm's main loop in pseudo-code.

Listing 4: Octree building

```
image  := COLOR[W, H]
octree := OCTREE {
        root := NUL,
        leafCount := 0,
        maxLeaves := N
    }
FOR y := 0 TO H-1
    FOR x := 0 TO W-1
        add(octree, image[x,y])
        WHILE octree.leafCount > N
            reduce(octree)
```

3.1 Inserting Pixel Colour

Each octree node corresponds to a cube-shaped (axis-aligned) subset of RGB color space. Root node represents the entire space, and for each parent node the subsets for all children can be determined by subdividing the cube subset of the parent in half perpendicularly to each axis. Figure 1 illustrates such subdivision by three orthogonal planes creating 8 equally-sized parts. This ensures that: the subsets for children of a given node are disjointed, union of child subsets equals the subset of the parent, and that subset of a descendant node is contained within the subset of any of its ancestors.

The colour of a given pixel should be added to a leaf node the subset of which contains that colour. We must, therefore, descend the tree, selecting the child node corresponding to a subset that contains the colour, until either a leaf node is found.

3.1.1 Calculating child node index

The selection of subdivision planes for each node does not need to be stored in the node. Let us consider the problem of selecting a child of a root node to descend to for a given input colour. The subdivision plane perpendicular to the R axis is: $R = 0 + \lfloor \frac{255-0}{2} \rfloor = 127$. Comparing the red channel of the input colour to that threshold value can tell us whether it belongs to one of the four children above or one of the four on and below that plane. Fortunately, instead of comparison, we can examine the highest bit of that channel — values up to 127 have that bit unset, values 128 and above have it set. Following the same approach for the other two axes, green

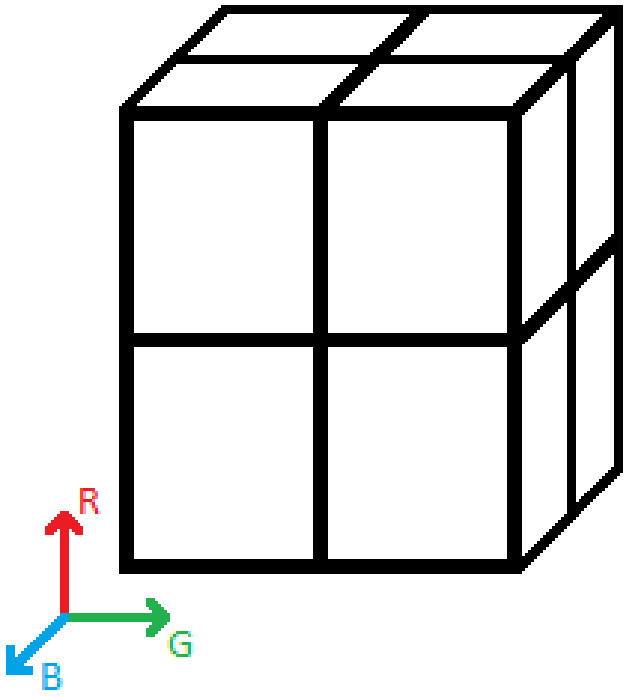

Figure 1: Podział sześcianu w przestrzeni RGB

and blue, we obtain three bit-flags. Those bit-flags can be combined into a three-digit binary number, giving us an index of a child node in range $[0, 7]$.

Descending by one level to a selected child node, the threshold value for subdivision plane of the red channel is either $R = 0 + \lfloor \frac{127-0}{2} \rfloor = 63$ or $R = 128 + \lfloor \frac{255-128}{2} \rfloor = 128 + 63 = 191$ (depending on which child was selected in the previous step), and similarly for green and blue. Since all colours in the corresponding RGB subset will have equal highest bits in each channel, we can again determine on which side of each of the subdivision planes the colour lies this time examining the second-highest bits of each channel, and use a binary number composed of those three bit-flags as a child index.

Extending the same notion to an arbitrary depth d in the tree (where depth 0 corresponds to the tree root), to select the child index for a given input colour, we must examine bits number $7 - d$ (where number 7 is the most significant, 0 is the least significant) of each channel. Listing 5 presents the pseudo-code for such procedure.

3.1.2 New node creation

Octree nodes are only created as needed, so often times the child node

Listing 5: Calculating child index

```
FUNCTION childIndex (
    color : COLOR,
    depth : UINT
) : UINT
    bitR := color.r >> (7 - depth) & 0x1
    bitG := color.g >> (7 - depth) & 0x1
    bitB := color.b >> (7 - depth) & 0x1
    RETURN (bitR << 2) | (bitG << 1) | bitB
```

we need to descend to will not exist. In such case a new node needs to be created and it's reference stored at the selected index in the children array of the parent.

New nodes are marked as leaves if and only if they are added at the maximum depth (i.e. depth 8, since we only have 8 bits in each channel — the RGB subset of such node contains only a single colour). When adding a leaf node, the leaf count of the tree needs to be incremented. On the other hand, a new non-leaf node needs to be appended to the list of interior nodes at the given depth (as a reminder, such lists will be useful for tree reduction — see. section 3.2.) Listing 6 presents a pseudo-code of a helper function for creating new nodes.

Listing 6: Creating new node

```
FUNCTION createNode (
    octree : OCTREE,
    depth  : UINT
) : NODE*
    newNode := allocate empty node
    newNode.isLeaf := (depth == 8)
    IF NOT newNode.isLeaf
        octree.innerNodes[depth].append(newNode)
    ELSE
        octree.leafCount += 1
    RETURN newNode
```

3.1.3 Recursive tree descent

Pixel colour is added to the octree via the following recursive algorithm, starting at depth 0 with the root node:

1. If current node is a leaf (*Note!* Although new nodes are only marked as leaves when added at the maximum depth, the tree reduction can

reduce an inner node to a leaf node. Therefore a leaf can be encountered at any depth in the tree):

   (a) add intensities of each channel of the colour to the totals in the leaf node,

   (b) increment the leaf's pixel count,

   (c) END.

2. Otherwise select child node index based on pixel colour and current depth.

3. If child node with such index does not exist:

   (a) create a new node,

   (b) store its reference in children array under selected index.

4. Recursively execute the algorithm for the selected child node incrementing the current depth by 1.

Listing 7 illustrates the start of the recursion and the algorithm itself is shown in listing 8.

Listing 7: Inserting pixel colour into the octree

```
FUNCTION add (octree : OCTREE, color : COLOR)
    IF octree.root == NUL
        octree.root = createNode(octree, 0)
    addRecursive(octree, octree.root, color, 0)
```

Calculating the total of per-channel intensities of added pixels might not be important for leaves at the maximum depth (as such nodes correspond to a single colour, so only pixel count would be sufficient). However, since tree reduction can produce leaves at any depth, for which the corresponding subset of RGB space is larger, we need to be able to determine the average of colours assigned to any leaf, regardless of its depth.

3.2 Tree Reduction

Thanks to the auxiliary array of interior node lists for all depths in the tree, tree reduction procedure is rather simple. We select an interior node at the lowest depth. That guarantees that all of its child nodes are leaves and we don't need to examine the entire subtree.

Selecting such node is as simple as finding the lowest depth with a non-empty list of interior nodes. We then remove one node from that list and reduce it to a leaf. The choice of a node from a list can be arbitrary: node with highest or lowest total pixel count in children; node with highest or

Listing 8: Recursive colour insertion into a subtree

```
FUNCTION addRecursive (
    octree : OCTREE,
    parent : NODE*,
    color  : COLOR,
    depth  : UINT
)
    IF parent.isLeaf
        parent.sumR += color.R
        parent.sumG += color.G
        parent.sumB += color.B
        parent.count += 1
    ELSE
        i := childIndex(color, depth)
        IF parent.children[i] == NUL
            parent.children[i] :=
                createNode(octree, depth + 1)
        addRecursive(octree, parent.children[i],
            color, depth + 1)
```

lowest number of children; first, last or random node in the list. Since we reduce interior nodes starting at the lowest depth, in general the approach chosen here doesn't have a significant impact on the quality of the algorithm.

To reduce an parent node to a leaf calculate the sum of per-channel totals and the sum of pixel counts from its children and store it in the parent. Then remove the children references from the array in the parent (*Note!* Remember to free the memory!) and mark the parent as a leaf. Finally the leaf count in the octree needs to be decreased by the number of children removed (but only those that actually existed) and then incremented by one (to account for the parent becoming a leaf.) Listing 9 presents the entire procedure in pseudo-code.

4 Pixel Colour Replacement

Once the octree has been built it will contain the number of leaves less than or equal to the given maximum palette size. Each leaf also contains the total of per-channel intensities and a pixel count — it's worth noting that each pixel of the image has been assigned to exactly one leaf in the final octree. Colour averages from all leaves form the output colour palette. The second stage of the algorithm involves finding a palette colour for each pixel in the image. That colour then replaces the original colour of the pixel. Listing 10 presents this procedure.

Listing 9: Octree node reduction

```
FUNCTION reduce (octree : OCTREE)
    FOR i := 7 TO 0
        IF NOT octree.innerNodes[i].empty()
            BREAK;
    node := select and remove node from octree.innerNodes[i]
    removed := 0
    FOR k := 0 TO 7
        IF NOT node.children[k] == NUL
            node.sumR += node.children[k].sumR
            node.sumG += node.children[k].sumG
            node.sumB += node.children[k].sumB
            node.count += node.children[k].count
            free node.children[k]
            node.children[k] := NUL
            removed += 1
    node.isLeaf = TRUE
    octree.leafCount += 1 - removed
```

Listing 10: Pixel colour replacement

```
FOR x := 0 TO W-1
    FOR y := 0 TO H-1
        image[x,y] = find(octree, image[x,y])
```

Finding a palette entry for a given input pixel colour is also rather simple, since we can use the existing octree structure. Since each pixel has already added to the octree, we can descend it again (at each level selecting the child index using appropriate bits from the intensities of each channel of the pixel colour), until a leaf node is found. The leaf we encounter might not be the same node the pixel was originally added to, but one of it's ancestors (as the tree might have been reduced since.) Nevertheless, all nodes on the path, up to and including that leaf are guaranteed to exist this time. Once the leaf is found we can calculate the average colour and return it as a chose palette entry. Listing 11 illustrates this approach.

Listing 11: Palette colour selection

```
FUNCTION find (octree : OCTREE, color : COLOR)
    node := octree.root
    WHILE NOT node.isLeaf
        i := childIndex(color)
        node := node.children[i]
    RETURN COLOR {
         r := node.sumR / node.count
         g := node.sumG / node.count
         b := node.sumB / node.count
     }
```