A Simple Method for Color Quantization: Octree Quantization

M. Gervautz and W. Purgathofer (Austria)

Abstract

A new method for filling a color table is presented that produces pictures of similar quality as existing methods, but requires less memory and execution time. All colors of an image are inserted in an octree, and this octree is reduced from the leaves to the root in such a way that every pixel has a well defined maximum error. The algorithm is described in PASCAL notation.

Keywords: color quantization, image display, color table, raster graphics, octree.

Introduction

The human eye is able to distinguish about 200 intensity levels in each of the three primaries red, green, and blue. All in all, up to 10 million different colors can be distinguished. The RGB-cube with 256 subdivisions on each of the red, green, and blue axes, as it is very often used, represents about 16.77 million colors and suffices for the eye. It enables display of color shaded scenes without visible color edges, and is therefore well suited for computer graphics (Fig. 1).

Color devices (mainly frame buffers) that allow for the projection of those 16 million colors at the same time are complicated and therefore expensive. On the other hand, even good dithering techniques produce relatively poor quality pictures on cheap devices //Jar76//. Therefore devices with color tables are produced that allow the use of a small contingent K (e.g. K=256) of colors out of a larger palette (e.g. 16 million colors).

When displaying images that contain more than K colors on such devices, the problem arises of which K colors out of the possible colors shall be selected and how the original colors are mapped onto the representatives to produce a satisfying picture. Such a selection is also needed for some other algorithms, such as the CCC-method for image encoding //Cam86//. The question is how much expense can or shall be invested in

N. Magnenat-Thalmann et al. (eds.), *New Trends in Computer Graphics*

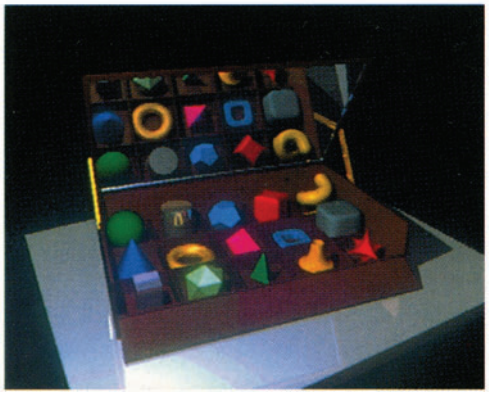

Figure 1: Computer generated image displayed with 16 million colors

this job. This paper first describes existing methods for the solution of this problem, and then presents a new algorithm we called "octree quantization" in detail.

Existing Solutions

The simplest way to handle the problem is to divide the RGB-cube into equal slices in each dimension and use the cross product of these (few) color levels of every primary for the color table. This "uniform quantization" could, e.g., devide the red axis and the green axis into 8 levels each, and the blue axis (our eye is less sensitive to blue) into 4 levels, so that 8.8.4=256 colors are available. The mapping of an image value into this selection is simply done by rounding each of the components (Fig. 2).

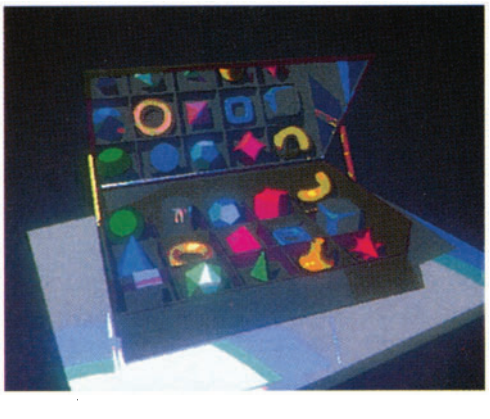

Figure 2: The same image as Fig. 1 displayed with 64 colors obtained from uniform quantization

The "popularity algorithm" chooses the K most frequently occurring colors for the color table. Therefore, in a first pass the whole image is explored, and all its colors are stored in a color histogram with their frequencies. The required memory for this histogram is quite large. Then the K colors with the highest frequencies are extracted. After the color table entries have been selected, the problem of mapping the original colors onto the available representatives remains. For this //Hec82// presents a method for finding the nearest color table neighbour for every point within the RGB-cube (Fig. 3). Although this "locally sorted search" lies significantly below the primitive solution to the problem (comparison with all color table entries) in terms of execution time, still a relatively high effort remains.

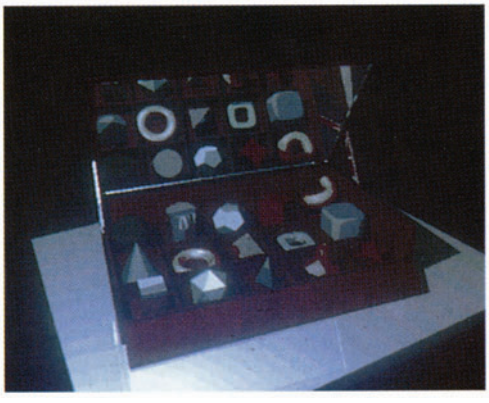

Figure 3: The same image as Fig. 1 displayed with 64 colors obtained from the popularity algorithm

The "median cut algorithm" //Hec82// tries to select K colors in such a way that each of these colors represents approximately the same number of pixels. To achieve this the color cube is subdivided into K rectangular boxes. In each of the K-1 subdivision steps the rectangular box with the most points in it is split into two parts along the longest dimension with about the same numbers of points in each half. The set of color table entries is obtained by calculating the mean value of points in each of the K boxes. For this, again, a color histogramm is needed and additionally radix lists for the subdivision steps. If the boxes are organized as a k-d-tree, this structure can be used very well for the mapping of the actual colors onto the color table (Fig. 4).

Figure 4: The same image as Fig. 1 displayed with 64 colors obtained from the median cut method

The new Method: Octree Quantization

Principle of the Method

The image is read sequentially. The first K different colors are used as initial entries to the color table. If another color is added, which means that the already processed part of the image has K+1 different colors, some very near neighbours are merged into one and substituted by their mean. For every further color this step is repeated, so that at any moment no more than K representatives are left. This, of course, is also true when the image is completely processed.

The Octree

For this method a data structure has to be used that enables quick detection of colors that lie close together in the color space. An octree is well suited for this problem //Jak80, Mea82//. The RGB-cube can easily be administered by an octree.

```
const  MaxDepth = 8;  (* maximum depth of the octree *)

type Color  = record  R,G,B : integer end ;

     Octree = ↑ Node;
     Node   = record
                Level      : integer;
                case  Leaf : boolean of
                  false : (Next : array [0..7] of Octree);
                  true  : (ColorCount : integer;
                           ColorIndex : integer;
                           RGB        : Color);
              end ;
```

It suffices to use an octree of depth 8 (two in the eighth is 256 levels in red, green, blue; eight in the eighth gives 16 million colors) to represent all possible colors. The red, green, and blue components (each between 0 and 255) are the coordinates within the octree:

```
function Branch (RGB : Color; Depth : integer) : integer;

(* evaluates the branch of the octree for the color RGB
   in depth Depth *)

begin
  Branch:= Bit (MaxDepth-Depth, RGB.R) * 4 +
           Bit (MaxDepth-Depth, RGB.G) * 2 +
           Bit (MaxDepth-Depth, RGB.B);
end ;
```

Every exact color is represented by a leaf in depth 8. Intermediate nodes represent subcubes of the RGB space. The greater the depth of such a node, the smaller is the color subcube represented by it, therefore the depth of a node is a measure for the maximum distance of its colors.

The Algorithm

Just as for the median cut algorithm, the octree quantization is done in three phases:

- evaluation of the representatives
- filling the color table
- mapping the original colors onto the representatives

These three steps are now described in detail using the color octree.

Evaluation of the Representatives

The octree is only constructed in those parts, that are necessary for the image of interest. At the beginning, the octree is empty. Every color that occurs in the image is now inserted by generating a leaf in depth 8, thereby the color is represented exactly.

```
var Size        : integer; (* number of leaves *)
    OctreeDepth : integer; (* depth of the octree *)

procedure InsertTree ( var Tree : Octree; RGB : Color;
                       Depth : integer);

(* inserts the color RGB into the subtree Tree
   in depth Depth *)

  procedure NewAndInit ( var Tree : Octree; Depth : integer);

  (* produces and initializes a new octree node *)

  var i : integer;

  begin (* NewAndInit *)
    new (Tree);
    with Tree↑ do
      begin
        Level:= Depth;
        Leaf:= Depth = OctreeDepth;
        if Leaf
        then
          begin
            Size:= Size+1;
            ColorCount:= 0;
            RGB:= (0,0,0);
          end
        else for i:=0 to 7 do Next[i]:= nil ;
  end ;

begin (* InsertTree *)
  if Tree = nil
  then  NewAndInit (Tree,Depth);
  with Tree↑ do
    if Leaf
    then
      begin
        ColorCount:= ColorCount + 1;
        AddColors (Tree↑.RGB, RGB)
      end
    else InsertTree (Next[Branch(RGB,Depth)], RGB, Depth+1);
end ;
```

In this way an incomplete octree is created, in which many branches are missing. Actually, this octree does not have to be filled with all the colors because every time the number of colors reaches K+1, similar colors are merged into one, so that there are never more than K colors left. We will call this action a reduction of the octree.

```
procedure ReduceTree;
(* combines the successors of an intermediate node
   to one leaf *)

var Tree         : Octree;
    Children, i  : integer;
    Sum          : Color;

begin (* ReduceTree *)
  GetReducible (Tree); (* finds a reducible node *)
  Sum:= (0,0,0);
  with Tree↑ do
    begin
      for i:=0 to 7 do
        if  Next[i] <> nil
        then
          begin
            Children:= Children+1;
            AddColors (Sum, Next[i]↑.RGB)
          end ;
      Leaf:= true;
      RGB:= Sum;
    end ;
  Size:= Size-Children+1;
end ;
```

Every time the number of leaves (that is the number of representatives found up to the moment) exceeds K, the octree is reduced. The reduction begins at the bottom of the octree by always substituting some leaves by their predecessor.

Reducing the octree, the following criteria are relevant:

- From all reducible nodes, those that have the largest depths within the octree shall be chosen first, for they represent colors that lie closest together.

- If there is more than one node in the largest depth, additional criteria could be used for an optimal selection (for simplicity, none of them was considered in the following program).

  e.g.: Reduce the node that represents the fewest pixels up to now. In this way the error sum will be kept small.

  Reduce the node that represents the most pixels up to now. In this case large areas will be uniformly filled in a slightly wrong color, and detailled shadings (like antialiasing) will remain.

To construct the color octree, the whole image has to be read once.

```
var K : integer;

procedure GenerateOctree ( var Tree : Octree);
(* constructs an incomplete octree Tree
   from all colors of the image in RGBfile *)

var RGB : Color;

begin (* GenerateOctree *)
  Size:= 0;
  Tree:= nil ;
  RGBread (RGBfile,RGB);
  while not RGBeof (RGBfile) do
    begin
      InsertTree (Tree, RGB, 1);
      while Size > K do ReduceTree;
      RGBread (RGBfile,RGB)
    end ;
end ;
```

Filling the Color Table

At the end the K leaves of the octree contain the colors for the color table. They can be written into the color table by recursively examining the octree:

```
procedure InitColorTable (Tree : Octree;
                          var Index : integer);
(* fills the color table with the means of the colors
   represented by the octree leaves *)

var i : integer;

begin (* InitColorTable *)
  if Tree <> nil
  then
    with Tree↑ do
      if Leaf
      then
        begin
          ColorTable[Index]:= Mean (RGB,ColorCount);
          ColorIndex:= Index; (* the color index is also
                                 written into the octree leaf *)
          Index:= Index+1;
        end
      else
        for i:=0 to 7 do InitColorTable (Next[i], Index);
end ;
```

Mapping onto the Representatives

The mapping of the original colors onto their representatives can now be managed easily with the octree, too. Trying to find any original color in the reduced octree will end at a leaf in some depth. This node contains a color very similar to the one in search, and is therefore its representative. Since the index of the color table is stored there too, no further search has to be carried out.
If the original image used less than K colors, no reduction will have taken place, and the found color table index will contain exactly the correct color. Otherwise, only the path to the leaf in depth 8 was shortened by the reduction, so that the color will be displayed less exactly by the mean of all the colors that had their pathes over this node. Since the octree contains only K leaves, all original colors are mapped onto valid color table entries. For this the image has to be read a second time.

```
procedure ImageOutput (Tree : Octree);
(* displays the whole image; every original color is
   mapped onto a color table index *)

  procedure Quant (Tree : Octree; Orig : Color) : integer;
  (* for the original color Orig its representative
     is searched for in the octree, and the index of
     its color table entry is returned *)

  begin (* Quant *)
    with Tree↑ do
      if Leaf
      then Quant:= ColorIndex
      else Quant:= Quant (Next[Branch (Orig, Level)], Orig)
  end ;

begin (* ImageOutput *)
  RGBread (RGBfile,RGB);
  while not RGBeof (RGBfile) do
    begin
      PixelOutput (Quant (Tree, RGB));
      RGBread (RGBfile, RGB);
    end ;
end ;
```

The visual result using this octree quantization is of similar quality as the result using the median cut method (Fig. 5).

Improvements

A significant portion of the execution time is spent with the search for an optimal reducible node every time a reduction of the octree has to take place. These nodes can be collected during the construction of the tree easily in an appropriate structure. They have to be sorted by depth to ensure quick

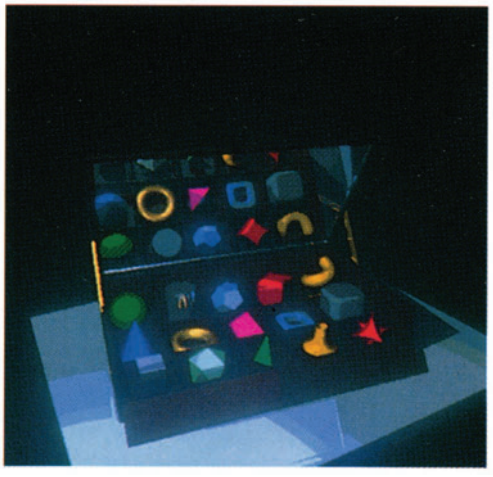

Figure 5: The same image as Fig. 1 displayed with 64 colors obtained from octree quantization

access. An appropriate structure for this purpose has proved to be 8 linear lists (one for every depth level) containing all reducible nodes. All nodes of one depth level are elements of the same list. The node with the largest depth can then be found quickly for reduction. For this, the declaration of the node of the octree has to be expanded:

```
type Node = record
              ...
              NextNode : Octree;
              (* next node in the same depth level *)
              ...
            end ;

var ReduceList : array [0..MaxDepth] of Octree;
     (* one list for every depth level in the octree *)
```

The procedure MakeReducible is activated when a new intermediate node (Leaf = false) is created in NewAndInit.

```
procedure MakeReducible (Level : integer; Node : Octree);

(* inserts the node Node with depth Level into the
   right list *)

begin (* MakeReducible *)
  Node↑.NextNode:= ReduceList[Level];
  ReduceList[Level]:= Node;
end ;
```

```
procedure GetReducible ( var Node : Octree);
(* finds the best reducible node of the octree *)

begin (* GetReducible *)
  while ReduceList[OctreeDepth-1] = nil do
    OctreeDepth:= OctreeDepth-1;
  Node:= ReduceList[OctreeDepth-1];
  ReduceList[OctreeDepth-1]:=
                  ReduceList[OctreeDepth-1]↑.NextNode;
end ;
```

At any given moment one level of the octree will be the depth in which the reductions take place. This depth is the level of the deepest intermediate nodes. At the beginning, this is level 7 and it moves towards the root during the octree construction. This "reduction level" states what the minimal distance between two representatives will already have to be. This minimal distance can never again decrease by adding even more colors to the octree. Therefore, nothing beneath this level + 1 will ever again be relevant, so that the insertion of colors can also stop at that depth. The depth of the octree is not constant, but decreases with lifetime (see NewAndInit).

Memory and Computational Expense

Let N be the number of pixels of the original image. If the image is run-length encoded, N can also be the number of runs of the image. The algorithm has to be modified slightly by using runs instead of pixels in the octree.

Let K be the number of representatives, that is the size of the color table.

Let D be the number of different colors in the original image.

In general the following equations hold:

$$N > D > K \quad \text{and} \quad N \gg K.$$

An upper bound for the memory used by the octree is 2*K-1 nodes, because there are K leaves and at the most (in the case of a bintree) K-1 intermediate nodes. The algorithm needs very little memory! It is also independent of N and D, that is, of the image. Only the color table size is relevant.

Upper bounds for the number of steps for the insertions, for the generation of the color table, and for the quantization are:

Insertion : N * MaxDepth

N insertions take place, each of them not deeper than MaxDepth.

Color table generation : 2 * K

To fill the color table the incomplete octree has to be examined once, for every node there is exactly one call to the procedure InitColorTable.

Mapping : N * MaxDepth

For every pixel the color index of its representative is found not deeper than in the maximum tree depth.

Thus the octree quantization algorithm is of O(N), the larger part of the execution time is spent by I/O-operations.

Comparison with the other Methods

The following Table 1 gives a short comparision with the other mentioned methods.

	memory	search for representatives	mapping	picture quality
Uniform Quant.	0	O(K)	O(N)	bad
Popularity algorithm	O(D)	O(K*N)	at least O(N)	depends on data
Median Cut	O(D)	O(N*ld(K))	O(N*ld(K))	good
Octree Quant.	O(K)	O(N)	O(N)	good

Conclusion

A new method was presented to find a color table selection for displaying an image on a screen. The picture quality of this "octree quantization" is as good as that for existing methods. The expense in terms of memory and execution time, however, lies significantly below the expense of those algorithms, especially the memory occupied is independent of the image complexity. The method is therefore well suited for microcomputers, too. The implementation is described completely so that it is easy to adapt it.

Acknowledgements

This project was sponsored by Digital Equipment Inc. and the Foschungsförderungsfonds der gewerblichen Wirtschaft and developed on a minicomputer VAX 11/730. We want to thank our colleagues for valuable discussions, especially Mr. Eduard Gröller and Mr. Michael Zeiller.

References

//Cam86// G.Campbell, T.A.De Fanti, et.al.: Two Bit/Pixel full Color Encoding. In Computer Graphics, ACM-SIGGRAPH, Vol.20, No.4, 1986, pp.215-223.

//Hec82// P.Heckbert: Color Image Quantization for Frame Buffer Display. In Computer Graphics, ACM-SIGGRAPH, Vol.16, No.3, July 1982, pp.297-307.

//Jak80// C.L.Jakson, S.L.Tanimoto: Octrees and Their Use in Representing Three-Dimensional Objects. In Computer Graphics and Image Processing, Vol.14, No.3, 1980, pp.249-270.

//Mea82// D.Meagher: Geometric Modelling Using Octree Encoding. In Computer Graphics and Image Processing, Vol.19, No.2, 1982, pp.129-147.

//Jar76// J.F.Jarvis, N.Judice, N.H.Nike: A Survey of Techniques for the Display of continous tone Pictures on bilevel Displays. In Computer Graphics and Image Processing, Vol.5, No.1, 1976, pp.13-40.