FUNCTION-TREES: AN EFFICIENT DATA STRUCTURE FOR COMPACT BINARY PICTURES

Giovanni Gallo, Daria Spampinato
gallo@dipmat.unict.it
http://aci.dipmat.unict/gallo
Dipartimento di Matematica, Università di Catania
viale A.Doria 6, 95100 Catania, Italy

ABSTRACT

In the representation of binary images or of binary three-dimensional scenes two principal paradigms have been introduced: boundary representation and hierarchical decomposition. The data-structure introduced in this paper, called function-tree or function-octree for a 3D scene, is an elementary generalization of the standard quadtree and octree. It combines the compactness of a boundary representation of spatial data with the nice recursive properties of a tree representation.This is achieved by decomposing regularly an image into rectangular sub-images whose black regions are *trapezoidal regions*. A trapezoidal region (defined in detail in the paper) is a region below the graph of a function. Trapezoidal regions can be, in turn, efficiently encoded with a boundary representation. Efficient algorithms to construct the function-tree are described in the paper. The time complexities of such algorithms are $O(n^2)$ for a $n \times n$ binary picture and $O(n^3)$ for a $n \times n \times n$ binary scene. The algorithms have been implemented. A final quick review of the possible applications of the proposed structures concludes the paper.

INTRODUCTION

Boundary representation describes the borders of the black regions in the image and could give an extremely compact representation, particularly for images or scenes that are not too fragmented. Boundary representation, moreover, is the natural choice for applications that require to minimize plotting time ([And83]) and in 3D for special ray-tracing techniques ([YCK92]). On the other hand boundary representation, generally, does not permit an easy access to geometric local information like membership in a given region, area computation, etc. ([DRS80], [HSt79], [Sam80]). These operations can be performed in a more natural way by coding an image or a scene in a hierarchical data-structure based on a recursive decomposition.The best known among hierarchical structures are certainly the quadtree and its three-dimensional generalization, the octree ([Sam84], [Sam85], [Sam90a], [Sam90b]). These structures have been proved ubiquitous and extremely useful in graphics and image processing ([SWe88a], [SWe88b]) but they could be at the same time quite redundant.

Several generalizations of traditional quadtrees have been proposed to overcome some of the problem that their use in graphics and solid modeling could present (see for example [Sam84]). All of them are motivated by particular applications and try to take advantage of special properties of the data. Particularly simple, beautiful and powerful is the generalization proposed in [BNa90]. Altought the structures proposed in this paper incorporates some of Brunet's generalized quadtrees principles, they do not have the same flexibility and wide range of applications of such structures. We believe, anyway, that the elementary character and simplicity of implementation of the function-tree and function-octree make the use of these structures an attractive choice, at least for special applications.

The data-structure, called function-tree, introduced in this paper, combines the compactness of a boundary representation of spatial data with the nice recursive properties of a tree representation. This is achieved by decomposing regularly an image into rectangular sub-images whose black regions are *trapezoidal regions*. A trapezoidal region is a region below the graph of a function. Trapezoidal regions can be, in turn, efficiently encoded with a boundary representation. The proposed data structure assumes that the image or the binary 3D scene has been already digitalized in an array of 0's and 1's of the proper dimension and size.

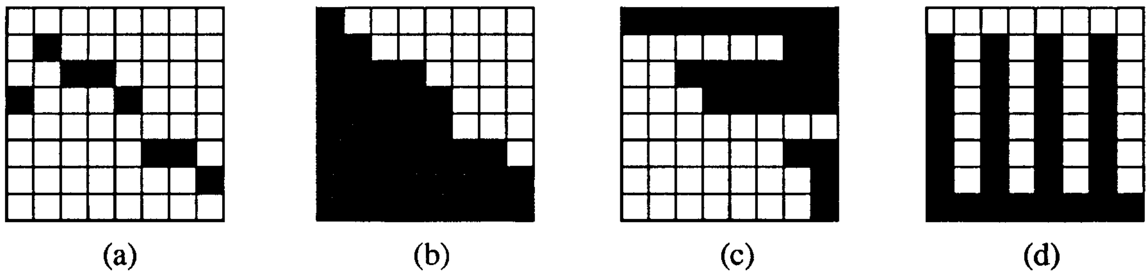


Figure 1. (a) Graph of the discrete function (f(1)=5, f(2)=7, ... f(8)=2); (b) Example of a trapezoidal region both in the E and N directions; (c) Example of a trapezoidal region in the W direction; (d) The "comb".

This paper is organized as follows: the first section introduces the function-tree structure and compares it with other data-structures. The second section describes a bottom-up algorithm to build a function-tree and its complexity. Applications are quickly surveyed in the final section.

1 THE FUNCTION-TREE

1.1 Motivations

For definitions and generalities about quadtrees and octrees the reader is referred to [Sam84], [Sam90a], [Sam90b]. In the same references a comprehensive guide to the variations that have been proposed in the literature can be found.
The average number of nodes in a quadtree for a simple polygon is roughly proportional to the perimeter of the black region in a picture (more precisely, the number of nodes is O(p+r), where p is the length of the perimeter and r is the resolution of the decomposition ([Sam84], [Hun78]). Hence, for binary images, traditional quadtrees take advantage of the properties of black regions with a small perimeter/area ratio. Visually this kind of regions appears not too fragmented and with a consistent number of internal pixels.
We found experimentally that the space requirements for coding these regions could be reduced using the new special data-structure introduced in this paper, *trapezoidal regions*, defined below, are the basic elements of such data structure.

Definition 1. *A trapezoidal region is the region below the function of a discrete function* $f: \{1,...,n\} \to \{0,...,n\}$*, where the domain and the range are conveniently oriented.*

The graph of a discrete function is shown in Figure 1.a. Examples of trapezoidal regions are shown in Figure 1.b and 1.c. Observe that the region in Figure 1.b is trapezoidal both in E and N directions: this is not at all a rare occurrence.

Observe that a discrete function $f: \{1,...,n\} \to \{0,...,n\}$ requires only $n \log(n+1)$ bits to be coded using binary digits.
An elementary way to code the boundary of a region is to store the coordinates of the pixels of the boundary in an array of convenient size (*array representation*) or in a list. Some improvements on space occupancy could be obtained, especially in presence of a very regular boundary, if a *chain code representation* is used. For details about chain-code the reader is referred to [Fre74].

Different data structures and representations for binary images of $n \times n$ pixels are compared in Table 1. Here it is assumed that each image contains only one trapezoidal region. In particular, Table 1 describes the worst case space occupancy, the worst case time complexity to build the data structures starting from a 2D array of binary pixels and the worst case time complexities to perform boolean and membership operations. The quadtree attains the space bound of $O(n^2)$ for the trapezoidal region called "comb" (see Figure 1.d). The space occupancy of the chain

Data structure	Space Occupancy	Building Time	$\cup$	$\cap$	$\in$
Chain code	O(n log n)	$O(n^2)$	O(n)	O(n)	O(n)
Array	O(n log n)	$O(n^2)$	O(n)	O(n)	O(1)
Quadtree	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(log n)

Table 1. Comparisons of data structures for a trapezoidal region.

code and array representation requires, in the worst case, for a n × n image, to store O(n) integers in the range {0, ... ,n}. This, as pointed above, can be done in O(n log (n+1)) space. To perform the set-theoretic operation $\cup$ and $\cap$ in the case of chain code and array representation, for a trapezoidal region, one needs only to scan two linear data structures, each one of size O(n). Such operation can, hence be done in time O(n). Finally, the set-theoretic membership requires, when an array representation is used, only an integer comparison, hence it can be performed in constant time.

Table 1 immediately gives the motivations for the data structure proposed in this paper: since the array representation is quite efficient in dealing with trapezoidal regions, a good way to code a binary image could be to decompose it in sub-images each one containing a single trapezoidal region. Each sub-image could be, in turn, coded with an array.

1.2 The data-structure for 2D

In this section we describe in detail the data-structure function-tree relative to binary image of size $2^k \times 2^k$. A *function-tree* is a tree of degree four with four types of nodes.

Each node corresponds to a region of the image. Precisely a node on level t corresponds to a $2^{k-t} \times 2^{k-t}$ square sub-image. Recursively, each one of the four children of a non-leaf node represents, in the proper order, one of the four quadrants of the sub-square represented from the parent. The recursive decomposition is hence similar to the decomposition operated with a standard quadtree.

The four types of nodes in a function-tree are: BLACK, WHITE, FUNCTION (for the leaf nodes) and GRAY for all the internal nodes.

GRAY nodes code regions that are neither trapezoidal neither completely black or white. Each GRAY node has four children.

A BLACK node is assigned to a node corresponding to a completely black region and a WHITE node to a completely white region.

A FUNCTION node is assigned to regions containing only one trapezoidal black region. In the actual implementation of the structure all the nodes have a flag signaling their type, a pointer to the parent and, if there is any child, pointers to them.FUNCTION nodes on the t-th level, moreover, contains a two bits flag to signal a direction (among N, E, W and S) and a 2^{k-t} list of compactly coded integers in the range $\{0, \dots, 2^{k-t}\}$. The information about the direction and these integers uniquely describe, in the obvious way, the trapezoidal region contained in the sub-square coded by the FUNCTION node. Observe that the integers can be chosen in the limited range $\{0, \dots, 2^{k-t}\}$ because they describe the boundary of the trapezoidal region using coordinates relative to the sub-block. BLACK and WHITE nodes, in conclusion, use only a constant number of bits, while FUNCTION nodes, for an image of dimension $2^k \times 2^k$, on the t-th level require $O((k-t)2^{k-t})$ number of bits. Figure 2.b shows the quadtree decomposition of the test binary image in Figure 2.a. Figure 2.c shows the function-tree decomposition of the same image. The compactness properties of the function-tree representation are quite evident.

The function-tree, generally, does not exceed the space requirements of the corresponding quadtree:

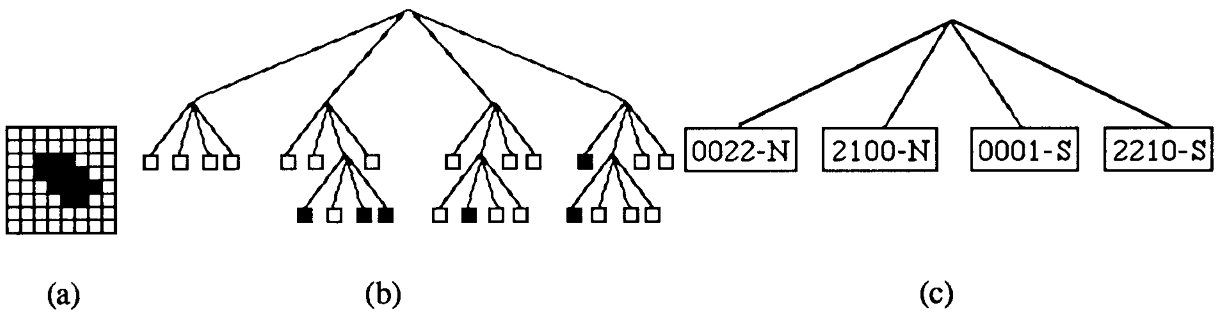


Figure 2. (a) A binary picture; (b) The quadtree decomposition; (c) The function-tree decomposition.

Proposition 1. *Given a binary image of* $n \times n$ *pixels, let* T_g *be the tree relative to the function-tree decomposition of the image and* T_q *the tree relative to the quadtree decomposition of the image. Looking at* T_g *and* T_q *regardless to the kind of information coded in their nodes, the following properties hold:*

1. $\text{leaves}(T_g) \leq \text{leaves}(T_q)$;
2. $\text{nodes}(T_g) \leq \text{nodes}(T_q)$;
3. $\text{av_size}(T_g) \geq \text{av_size}(T_q)$, where av_size($T$) denote the average size of the blocks of the decomposition T.

The proof is immediate observing that the decomposition in smaller squares of the $n \times n$ pixels image induced by the quadtree is necessarily a refinement of the decomposition induced by a function-tree.Hence, regardless to the information contained in the leaves, a function-tree can be obtained replacing branches of the quadtree with a single FUNCTION node.

The space occupancy for the function-tree data structure strongly depends on its implementation. We found experimentally that for images of small size (less than 64×64 pixel) there is no clear winner between quadtrees and function-trees. But as the size of the image grows the function-tree generally takes considerably less space than the quadtree. The worst case for both the function-tree and the quadtree happens when the binary image to code is a checkerboard. In this case the function-tree is identical to a traditional quadtree, because there are no regions (other than single pixels) that are properly trapezoidal. The number of nodes, for both structures, in this case, is $O(n^2)$, for a checkerboard of size $n \times n$ pixels. Quadtrees could achieve the upper bound of $O(n^2)$ even for trapezoidal regions. This, for example, happens for "comb" regions (see Figure 1.d). Function-tree, instead, behave quite well on this kind of pictures: the space occupancy for "comb" regions is, in fact, still $O(n \lg n)$.

1.3 Function-Octree

The advantages of using the function-tree for 2D-images are enhanced when the same structure is generalized to 3D. In this case the structure, called function-octree, uses as basic element the 3D-equivalent of the trapezoidal 2D-regions.

Definition 2. *A trapezoidal region is the region below the graph of a discrete function* $f: \{1,\dots,n\} \times \{1,\dots,n\} \rightarrow \{0,\dots,n\}$ *, where the domain and the range are conveniently oriented.*

Examples of trapezoidal 3D regions are shown in Figure 3.

Observe that a discrete function $f: \{1,\dots,n\} \times \{1,\dots,n\} \rightarrow \{0,\dots,n\}$ requires only $n^2\log(n+1)$ bits to be coded using binary digits.

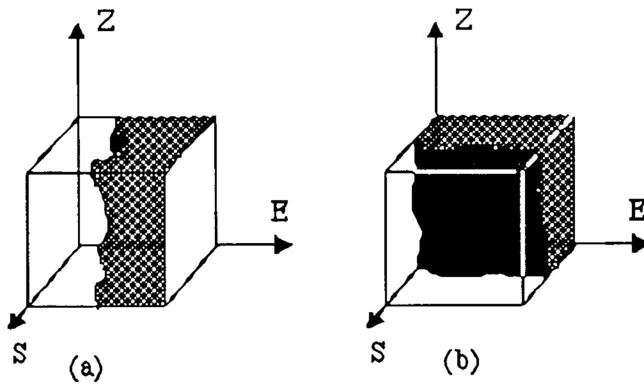


Figure 3. (a) A 3D region that is trapezoidal in direction W; (b) a 3D region that is trapezoidal in the S direction.

A *function-octree* is a tree of degree eight with four types of nodes. As with the octree each node corresponds to a region of the 3D scene. More precisely a node on level t corresponds to a $2^{k-t} \times 2^{k-t} \times 2^{k-t}$ cubic sub-scene. Recursively, each one of the eight children of a non-leaf node represents, in the proper order, one of the eight octants of the cube represented from the parent.

The four types of nodes in the structure are: BLACK, WHITE, FUNCTION for the leaf nodes and GRAY for all the internal nodes.

GRAY nodes code volumes that are not either trapezoidal either completely full or empty. Each GRAY node has eight children.

A BLACK node is assigned to a node corresponding to a completely full volume and a WHITE node to an empty volume.

A FUNCTION node is assigned whenever the volume it represents contains only a trapezoidal full volume.

In the actual implementation of the structure all the nodes have a flag signaling their type, a pointer to the parent and, if there is any child, pointers to them.

FUNCTION nodes on the t-th level, moreover, contains a three bits flag to signal a direction (among N, E, W, S, Z and Na) and a $2^{2(k-t)}$ list of compactly coded integers in the range $\{0,...,2^{k-t}\}$. The information about the direction and these integers uniquely describe, in the obvious way, the trapezoidal region contained in the sub-square coded by the FUNCTION node. Observe that the integers can be chosen in the limited range $\{0,...,2^{k-t}\}$ because they describe the boundary of the trapezoidal region using coordinates relative to the sub-scene. BLACK and WHITE nodes, in conclusion, use only a constant number of bits, while FUNCTION nodes, for an image of dimension $2^k \times 2^k$, on the t-th level require $O((k-t)2^{2(k-t)})$ number of bits. All the other details of the structure are similar to the 2D-case. Observe that the structure can be naturally generalized to higher dimensions.

2 Bottom-up construction of a FUNCTION-TREE

In this Section we present an algorithm for the efficient construction of the function-tree of an image and of the function-octree of a volume. In both cases the algorithm is essentially the same: for this reason we first describe it for the 2D case and subsequently we sketch how to generalize it to higher dimensions.The algorithms described in this Section adopts a bottom-up approach to the construction of a function-tree: it combines small function-trees relative to small areas of the image in a larger structure relative to their union.

2.1 The algorithm

The algorithm tries to merge recursively four adjacent square sub-regions. Each one of these regions contains only a trapezoidal region. The merge operation is possible only if the resulting larger square still contains a unique trapezoidal region. If this is the case the algorithms keeps only a FUNCTION node to code the larger square. All the information contained in the four sub-regions is transferred to the new FUNCTION node and the four original nodes could be pruned off the tree. The rest of this sub-section describes the procedure in detail.

The base of the recursion is realized when the square sub-regions to mergeare made of only one pixel. The algorithms scans according to a Morton matrix order the four pixels ([Mor66]). Since each pixel, be it white or black, is a non-proper trapezoidal region the algorithm creates four nodes of the proper (BLACK or WHITE) color.

Suppose now that all the four $m \times m$ sub-squares of a $2m \times 2m$ square region have been processed by the algorithm, and suppose that each one of them has been coded as a FUNCTION node, or as a BLACK node, or as a WHITE node. In this case the algorithm checks if the union of the four trapezoidal regions contained in each sub-square is still a trapezoidal region in some common direction (or is entirely black or entirely white). In this case a new FUNCTION (or BLACK or WHITE) node is created to substitute the four nodes and to store all of their information. If some of the four $m \times m$ sub-squares or if their union do not contain one trapezoidal region in some direction, a GRAY node is created as a parent of the four nodes coding each sub-square. The recursive process ends when a single node, coding the whole image, is created as the root of the function-tree.

To complete the description of the algorithm one must specify how to verify when four siblings, not GRAY nodes could be substituted by a single FUNCTION parent. To operate in this sense the algorithm uses some additional information stored in each non-GRAY node during the construction of the function-tree. Every time the algorithm creates a FUNCTION node A it also appends to it a string L(A). The string tells in which directions the black region of the sub-square could be considered trapezoidal. Moreover the algorithm creates an array of bits Cod(A) describing the black and white pixels in the perimeter of the sub-region relative to A.

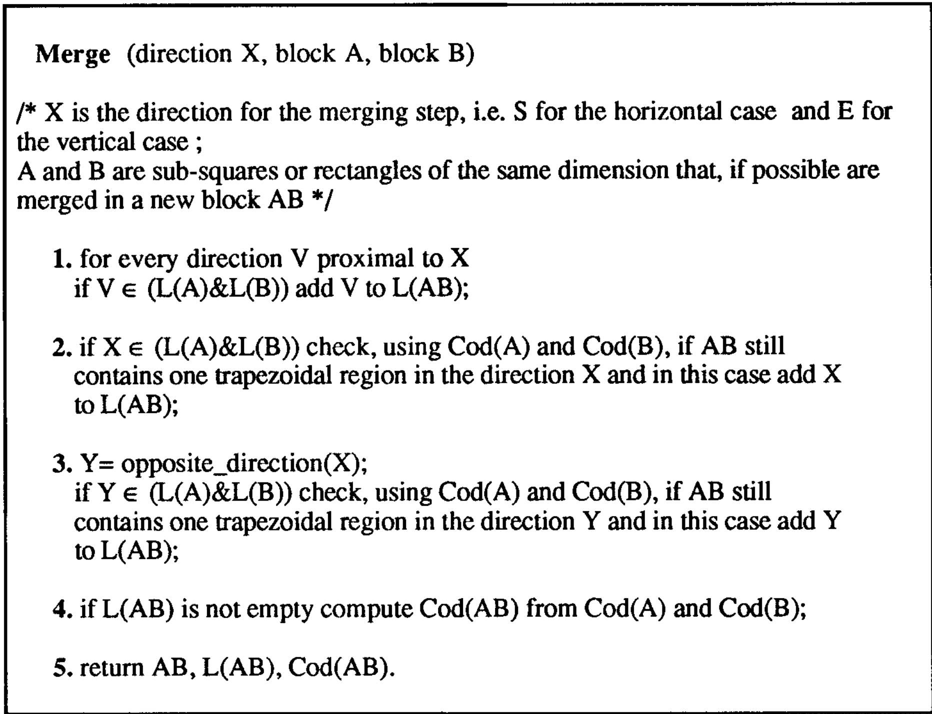
Notice that both the label and the array are temporary structures and they get discarded when the construction of the function-tree is completed. In Figure 5.a the label and the array of bits relative to some regions are reported.

The algorithm operates as follows: first it tries to merge the four $m \times m$ sub-squares in two horizontal $m \times 2m$ rectangles (E-W merge). If this step succeeds the algorithm generates a label and a proper array of bits for each rectangle and it tries to merge the two rectangles together in a $2m \times 2m$ square (N-S merge). The merging process is realized with the Merge function described in Figure 4.

Steps 2, 3 and 4 of the Merge procedure need further explanation. Steps 2 and 3 are similar; to check if the union of two blocks still contains a trapezoidal region in a given direction one needs to compare the two joining edges. For example if two blocks are merged vertically and both contain a trapezoidal region in the direction S one has to check only for the white pixels of the south edge of the superior block if the corresponding pixels on the north edge of the inferior block are white. This can, of course, be done in O(m) time. Step 4 requires the construction of a new array of bits describing the border of the merged region AB. This can be done first erasing the portions of Cod(A) and Cod(B) relative to the joining edges and chaining in the proper way what is left. Again this task can be accomplished in O(m) time. Figure 5 shows the merging procedure for two rectangular blocks along a horizontal edge where Merge is called with direction X=S.

2.2 Algorithm complexity

In this Section we give a worst case bound on the time needed, using the algorithm of the previous section, to build a function-tree, from a binary image.



Merge (direction X, block A, block B)

/* X is the direction for the merging step, i.e. S for the horizontal case and E for the vertical case ;
A and B are sub-squares or rectangles of the same dimension that, if possible are merged in a new block AB */

1. for every direction V proximal to X
   if V ∈ (L(A)&L(B)) add V to L(AB);

2. if X ∈ (L(A)&L(B)) check, using Cod(A) and Cod(B), if AB still contains one trapezoidal region in the direction X and in this case add X to L(AB);

3. Y= opposite_direction(X);
   if Y ∈ (L(A)&L(B)) check, using Cod(A) and Cod(B), if AB still contains one trapezoidal region in the direction Y and in this case add Y to L(AB);

4. if L(AB) is not empty compute Cod(AB) from Cod(A) and Cod(B);

5. return AB, L(AB), Cod(AB).

Figure 4. The Merge procedure.

Lemma 1. *The worst case complexity of the procedure* Merge *called on blocks of dimension* $m \times m$ *or* $2m \times m$ *is* $O(m)$.

Proof. Step 1 and 5 of Merge require constant time and, as observed earlier, steps 2, 3 and 4 require $O(m)$ time.

Theorem 1. *A function-tree relative to a binary image of* $n \times n$ *pixels can be computed in* $O(n^2)$ *time.*

Proof. Without loss of generality one could assume $n=2^k$.The result is proved by induction on k. In the case of k=1, i.e., a 2×2 image, the function-tree is built in constant time $T(1)=c_1$. In the case k >1, i.e., for a $2^k \times 2^k$ image a function-tree is recursively built merging the function-tree structures relative to four $2^{k-1} \times 2^{k-1}$ sub-squares. Hence $T(k)=4\,T(k-1) + M(k-1)$, where $M(k-1)$, the time required to do the merging, from the Lemma above, is $O(2^{k-1})$.T(k) can be computed solving the recurrence:

$$T(1)=c_1 \qquad\qquad T(k)=4\,T(k-1)+c\,2^{k-1}$$

Using the Master theorem ([CLR90]), or by substitution and telescoping one eventually gets: $T(k)=O(2^{k-1})=O(n^2)$.

2.3 Generalization to 3D

The algorithm in Section 2.1 can be immediately generalized to an algorithm to build the function-octree relative to a binary cubic scene.The recursive construction is pictorially described in Figure 5.b. Observe that the 3D version of the Merge procedure requires $O(n^2)$ time to join two blocks of $O(n) \times O(n) \times O(n)$. Using this observation it is immediate to prove:

Theorem 2. *A function-octree relative to a binary scene of* $n \times n \times n$ *voxels can be computed in* $O(n^3)$ *time.*

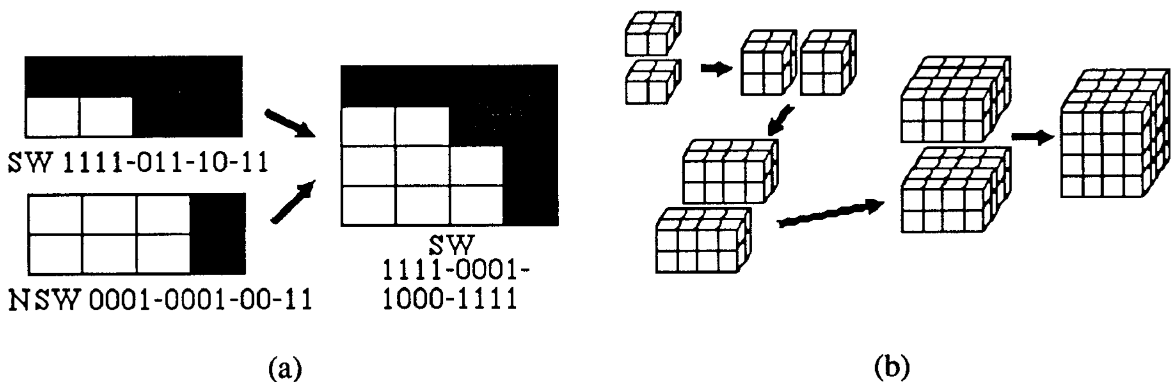


Figure 5. (a) Merging of two rectangular blocks. Labels and codes for the borders are indicated. The edges are reported in the order NSWE; (b) Sequence of merging for the recursive computation of a function-octree from the function-octrees of its octants.

3 EXPERIMENTAL RESULTS

We compared the efficiency of function-trees and quadtrees on a test set of 14 black&white sprites, of 128 x 128 pixels each. Since the actual memory occupancy depends on the impementation we prefer report our results in terms of the average number of nodes in the structures. For a quadtree we found, on the average, 521 nodes, 130 GRAY, 171 BLACK, 220 WHITE. The corresponding function-trees have, on the average, had 132 nodes, 33 GRAY, 16 BLACK, 32 WHITE and 51 FUNCTION. The average length of the array mantained in each FUNCTION node is 8.43. In total FUNCTION nodes mantain 430 "functional values".

4 CONCLUSIONS AND APPLICATIONS

The function-tree and the function-octree structures can be applied usefully in all the situations that involve binary images or scenes that are not too fragmented. A promising application of the data structure introduced in this paper is in the field of fractal compression of images. This compression technique requires a large number of test of similarity between regions of an image. These tests are computationally expensive for generic regions: generally they require $O(n^2)$ time for $n \times n$ pixels regions. If an image can be decomposed in sub-squares, each one containing only one trapezoidal region, the time required for each similarity test is O(n) with a large efficiency gain. We have some encouraging preliminary results in this direction.

Another application of function-octree can be found in the field of visualization of 3D-data. It is well known, in fact, that octree representation of solids helps the removal of hidden surfaces ([VAh88]). A limit of this techniques is that an object comes out as the union of many small cubic cells. On the other side there are many efficient techniques to smoothly draw the graph of a 2-dimensional function in 3D. These two tools can be combined, by means of function-octree, to produce a rendering procedure.

Particularly promising seems to use a function-tree description of a 3D scene together with discrete ray tracing techniques ([CHw95],[YCK92]).

AKNOWLEDGMENTS

The authors wish to thank prof.H.Samet and prof.B.Mishra for the remarks and suggestions given on a earlier version of this paper.

REFERENCES

[And83] D.P.Anderson, *Techniques for Reducing Pen Plotting Time*. ACM Trans. on Graphics, Vol.2, no.3, 197-212, 1983.

[BNa90] P.Brunet, I.Navazo, *Solid Representation and Operation Using Extended Octrees.* ACM Trans. on Graphics, Vol.9, no.2, 170-197, 1990.

[CHw95] J.H.Chuang, W.J.Hwang, *A new space subdivision for ray tracing CSG solids.* IEEE Comp. Graphics and Appl. (Nov.1995) 56-62.

[CLR90] T.H.Cormen, C.E.Leiserson, R.L.Rivest, *Introduction to algorithms.* McGraw-Hill Book Company, New York, 1990.

[DRS80] C.R.Dyer, A.Rosenfeld, H.Samet, *Region representation: Boundary codes from quadtrees.* Commun. ACM 23, 3 (Mar. 1980), 171-179.

[Fre74] H.Freeman *Computer processing of line-drawing images.* ACM Comput. Surv. 6, 1 (Mar. 1974), 57-97.

[Hun78] G.M.Hunter, *Efficient computation and data structures for graphics.* Ph.D Dissertation Dept. of Electrical Eng. and Computer Science, Princeton University,Princeton, NJ, 1978.

[HSt79] G.M.Hunter, K.Steigliz, *Operation on images using quadtrees.* IEEE Trans. Pattern Anal. Mach. Intell. 1, 2 (Apr. 1979), 145-153.

[Mor66] G.M.Morton *A computer oriented geodetic data base and a new technique in file sequencing.* 1966.

[Sam80] H.Samet, *Region representation: Quadtrees from boundary codes.* Comm. ACM 23, 3 (Mar. 1980), 163-170.

[Sam84] H.Samet, *The quadtree and related hierarchical data structures.* ACM Comput. Surv. 16, 2 (June 1984), 187-260.

[Sam85] H.Samet, *Data structures for quadtree approximation and compression.* Commun. ACM 28, 9 (Sept. 1985), 973-993.

[SW88a] H.Samet, R.E.Webber, *Hierarchical Data Structures and Algorithms for Computer Graphics. Part 1: Fundamentals.* IEEE Comp. Graph and Appl. (May 1988) 48-68.

[SW88b] H.Samet, R.E.Webber, *Hierarchical Data Structures and Algorithms for Computer Graphics. Part 2: Applications,* IEEE Comp. Graphics and Appl. (July 1988) 59-75.

[Sam90a] H.Samet, *Applications of spatial data-structures: computer graphics, image processing and GIS.* Addison-Wesley, Reading MA 1990.

[Sam90b] H.Samet, *The design and analysis of spatial data structures.* Addison-Wesley Reading, MA 1990.

[VAh88] J.Veenstra, N.Ahuja, *Line drawings of octree-represented objects.* ACM Trans. Gr. 7, 1 (Jan. 1988), 61-75.

[YCK92] R.Yagel, D.Cohen, A.Kaufman, *Discrete Ray Tracing.* IEEE Comp. Graphics and Appl. (Sept.1992) 19-28.