A GENERAL ALGORITHM FOR GEOMETRICAL TRANSFORMATIONS ON QUADTREES

Oscar E. Ruiz S. (Research Assistant)
Placid M. Ferreira (Assistant Professor)

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

TECHNICAL REPORT
University of Illinois at Urbana – Champaign
1989

A General Algorithm for Geometrical Transformations on Quadtrees

O. Ruiz, Graduate Student
P. M. Ferreira, Assistant Professor
Department of Mechanical and Industrial Engineering
University of Illinois at Urbana-Champaign

Abstract

A Geometrical transformations on a quadtrees is more of a representation problem than one of transformation. Thus, for any desired geometrical transformation, the most expensive operation on quadtrees (and octrees) is that of rebuilding the converted data structure to match the original space grid. This paper develops a general algorithm which uses a representation strategy for geometrical transformations. It is implemented on a pointer-based quadtree without using additional auxiliary data structures and has a worst case complexity of O(M), where M is the number of nodes in the output quadtree. The algorithm, although developed and tested for quadtrees, extents naturally to octrees.

1. INTRODUCTION.

Every CAD/CAM software system is built on the basis of a suitable representation scheme for the domain of objects it seeks to model. B-rep (Boundary Representation), CSG (Constructive Solid Geometry), Spatial Enumeration, and many other schemes [12] , [13] have been proposed and developed. Each representation scheme may be characterized by its domain, representation properties (completeness, uniqueness, validity enforcement, etc) and the efficiency with which object models can be stored and manipulated in it. As different engineering problems, encountered in the analysis and subsequent manufacture of a part, require different information and manipulation of the part model, it has been conjectured that the ability to switch(translate) between different representation schemes (each being most efficient for a particular problem) is necessary in CAD/CAM systems [14] , [15]. Sometimes such a translation is not possible because the representation scheme chosen is ambiguous or simply because the object is out of its domain. An example of the first situation is the translation of a model from a wire-

frame scheme to a B-rep scheme; it is well-known that the wire-frame scheme is ambiguous and, consequently, it is possible that B-reps of several objects could match a particular wire-frame model. The representation of a sphere in a polyhedral modeler is an example of the second situation. No exact representation of this object is possible in such a scheme.

Among the different representation schemes, spatial enumeration schemes are found to be particularly useful for performing set operations (unions, intersections and differences) and for calculating integral properties such as surface area, weight, volume, centroid location ,moments of inertia, etc., of a solid. The drawback of such schemes, besides the fact that they are approximate representations, is the large quantities of data required for representing even relatively simple objects. For example, in spatial enumeration, the accuracy of the representation increases as the cell size decreases. However, the size of the representation increases exponentially as well. A trade-off is achieved by trying to express the largest possible portion of the solid with one single cube, and recursively applying the same idea on the remaining portions of the part, using smaller cubes. The result is space savings, but the algorithmic aspects of developing, maintaining, transforming and using such schemes turn out to be very involved. The algorithmic aspects of such recursive schemes are improved by fixing the locations at which decompositions can take place in the 'modeling space' (or fixing the decomposition grid). The effects of this constraint are far-reaching. First, because the same grid is used, for a given resolution of representation, the number of potential decompositions is finite, making the representation generation problem less complex. The second effect makes set operations among the objects in a given modeling space easier, because all of them are built using the same set of cubes in the same locations in the space. This concept forms the basis of octree (or quadtree) representations.

In an octree, the modeling space is initially divided into 8 octants each one of which is labeled FULL, VOID or MIXED, depending whether the solid being represented completely fills it, is completely out of it, or is partially occupying it. For the case in which the octant is labeled MIXED, that octant is further divided into its 8 sub-octants and, hopefully, some of them will be labeled VOID or FULL. The remaining sub-octants will be recursively subdivided in a similar manner. Of course,

it is possible that this process be repeated indefinitely with the size of the octants decreasing with each recursion. However, when the cell size is small enough for it to be irrelevant (in the context of the application) whether a MIXED cell is labeled VOID or FULL, this process of recursive sub-division is stopped by an arbitrary decision.

Several data structures have been used for the internal representation of octrees, the most popular ones being: pointer-based structures and the linear representation. In the pointer-based structure each octant in the the space is represented by a node labeled FULL, VOID or MIXED; if FULL or VOID, it is a leaf in the tree; if MIXED, it must have 8 children (4 for quadtrees) for which the same idea applies. In the linear representation, each octant (quadrant) is marked with a number, and only the FULL nodes are represented. Each FULL node is represented by a string of module 8 (module 4 for quadtrees) digits which designate the quadrant in which that node lies within its parent's quadrant. A NorthWest quadrant is 0, a NorthEast one is 1, a SouthWest one is 2 and a SouthEast one is 3. Some authors [5] name the collection of strings representing FULL nodes, the "leafcode" of the tree; see figure 2. For easier manipulation and searching the entire set of strings may be ordered. The ordering criterion is given by the mentioned digits being compared from left to right.

The basic idea of recursive subdivision has many variations. By changing the shape of the sub-division volume (or area for quadtrees) from a block (or rectangle) many different variations such as triangular (ternary or quaternary) decompositions are obtained. When working with co-ordinate spaces other than Cartesian, such as a polar spaces, a *cone tree* may be used. While these are interesting extensions to the basic idea of octree decomposition, they are not immediately relevant to the scope of this paper. The reader is referred to Samet [10] and [11] for an in-depth discussion on variations on the basic concept of quadtrees and octrees. Chapter 5 of [10], in particular, discusses several alternative primitives for the leaves of the tree.

As was stated earlier, integral property calculations and set operations are easily performed on octrees. However, automatic representation generation (i.e., the representation of some arbitrary object by an octree), and geometrical

transformations such as translation, scaling and rotation are complicated and expensive. The reason for the latter stems from the fact that the octants/cubes representing an object are resized, positioned and oriented when the object undergoes a geometrical transformation. They might, therefore, correspond to unacceptable sizes, positions or orientations compared to the specified and fixed ones for the modeling space. Each of these octants would, therefore, have to be represented in terms of the original decomposition grid. Thus, a geometrical transformation involves a representation task. The purpose of this paper is to develop a representation strategy in a general form such that it can be used for translation, scaling, rotation or any combination of the three.

This paper is organized as follows: section 2 is a survey of the related literature; section 3 states the problem and explains the algorithm used; section 4 presents the complexity analysis and experimental results and section 5 draws the conclusions of this work. Some details of the computer implementation are presented in the appendix.

2. SURVEY OF ALGORITHMS FOR ROTATION,TRANSLATION AND SCALING OF OCTREES.

In [1], Jackins & Tanimoto address the problem of translation, and rotation by 90^{o}. The algorithm translates each node of the source tree by the DELTA(translation) vector in the space, and then represents the resulting object in the original space grid. To do this, it has to use the list of eight (eight nodes for octrees; four for quadtrees) nodes which overlap the current target node. "A node at level i in the target tree receives a list of nodes for comparison that reside at level i in the source tree; the only exception to this rule is that there may occur on the list source nodes labeled FULL which reside at levels above i". Clearly, this approach does not work for scaling operations, because after such a transformation, corresponding levels in the two trees do not represent the space size. The neighborhood information is explicitly stored in the list of overlapping source nodes. The order of the algorithm is O(8n) (n= log of diameter of the object); this is equivalent to saying that it is of the order of the number of nodes in the target tree.

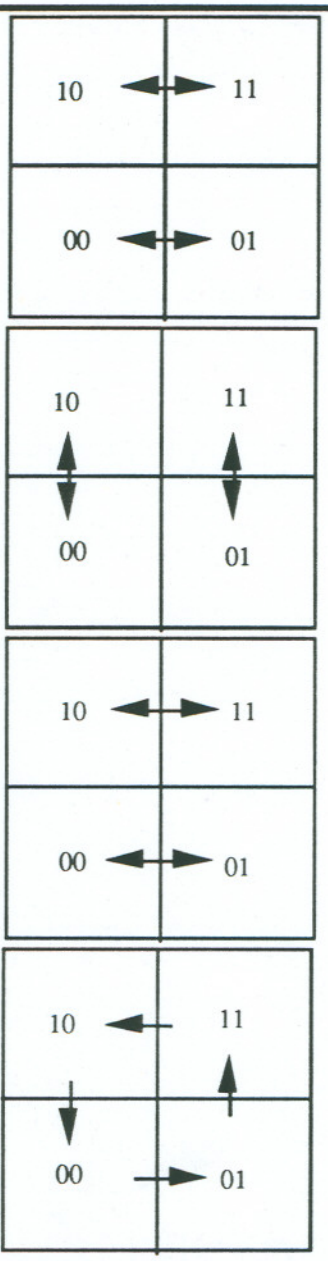
In [2], Meagher develops the same idea of using an auxiliary data structure for keeping track of overlaying nodes which completely cover the space being considered for labeling in the tree and applies this to arbitrary translation, scaling and rotation. In his terms this overlaying space is a set of *obels*. He also proposes a method for improving the arithmetic operations by expressing each square as four line equations, $Ax+By+C=0$ with $A^2 + B^2 =1$. This makes the derivation of the line equations for the children of that node easier. Further, it allows one to perform tests of intersection using the distance point-to-line criteria to decide if some square overlaps with another. This lowers the number of floating point operations needed. This method is applicable in any algorithm which tests for inclusion and intersection of rectangular regions, allowing a reduction in the time devoted to floating point operations; however it will not, by itself, modify the complexity of any algorithm which uses it. In [2] it turns out that in spite of incurring into the overhead of maintaining and using an auxiliary data structure, the worst-case complexity does not get better than O(M), where M is the number of nodes in the output (target) octree.

Kawaguchi and Endo [16], and Kawaguchi, Endo and Matsunaga [17], (also referred to in [11]) present a very appealing similarity between clustering of raw pixel data from a black and white picture and syntactic evaluation of the picture by a grammar whose production rules are:

Starting_Symbol -> <Subframe>
<Subframe> -> <Subframe> <Subframe> <Subframe> <Subframe>
<Subframe> -> **black**
<Subframe> -> **white**

where <Subframe> is the only non-terminal symbol and **black** and **white** are the terminal symbols. There is obviously a mapping between the sequence of production rules used to "compile" a picture and the construction of the quadtree which represents it. Some portion of the picture must be evaluated to decide which production rule must be fired; this represents the construction of the quadtree in a Depth First (DF code) traversal. Conversely, the DF code is the binary version of the Depth First traversal of the quadtree. Since this is a string of 0's and 1's, separators "(" help to encode-decode the node information. The process of building the string has a complexity with a lower bound of the order of the size of the tree represented. When the DF code is available for a given picture, certain operations which represent transformations such as translation by an integer multiple of the pixel size, rotation by an angle which is multiple of 90°, mirroring and scaling can be performed on it. The positional code to describe the subquadrants is shown in FIGURE 1, together with the transformations they undergo to represent the geometrical operation. Of course, after the DF code transformation, a new process of compaction and "(" re-arrangement is necessary to ensure the validity of the DF code as a quadtree object.

In Gargantini [4], an algorithm for linear quadtree translation and 90/180-degree rotation is proposed. It is based on performing arithmetic operations on the position codes of the black nodes in the linear quadtree. For translating a region of N_P pixels, represented by a linear octree, the octree is decomposed into its constitutive pixels, and each pixel positional code is modified to express the translated one.



Right/Left Translation:
01 <-> 00
10 <-> 11

Up/Down Translation:
00 <-> 10
01 <-> 11

Mirror Transformation:
Vertical Axis
01 <-> 00
10 <-> 11

90°rotation
00 -> 01
01 -> 11
11 -> 10
10 -> 00

FIGURE 1: DF code transformations representing geometric operations.

Once the pixel-wise transformation is done, sorting is performed on the resulting list of transformed pixels and then compaction is necessary to have the new linear quadtree representing the transformed object. Pixel collection translation is done in $O(N_P)$ time; sorting is done in $O(N_P \log N_P)$; compaction of N_P pixels is claimed to be performed in $O(N_P)$ time. Unfortunately, details of the procedure are not discussed. This algorithm works in $O(N_P \log N_P)$. The paper does not address the problem of arbitrary rotation. With arbitrary rotations problems arise because, for each pixel

in the image, the mapping applied to its positional code is different, depending on its position relative to that of the rotation axis.

In [3] Ahuja & Nash show that translation can be accomplished by module-8 arithmetic manipulations on the location codes of FULL nodes in the source tree. To do this, the binary representation of the translation distance must be obtained; in this way each one of the binary digits will affect only the levels on the tree which represent a bigger size than that represented by the binary digit. Every time a black node is found, its new "address" in the target tree is computed and that "destination location is then accessed, starting from the root of the tree and moving down... This may require new branches and nodes when the path ... does not already exist..". Because this algorithm traverses the source tree and each time starts the target search in the root of the target tree, its complexity is O(n.M) (n=levels in the output tree; M=nodes in the input tree). However, the module-8 arithmetic idea is not used to directly manipulate the branches in the tree. Weng & Ahuja in [6] propose an algorithm for rotation and translation in which each FULL node in the source tree is transformed and the result of this transformation is completely represented (up to a level allowed by the representation resolution) in the space grid. The schemes proposed ([6], [3]) are very interesting and intuitively appealing. It is possible that neighborhood information can be exploited to localize work performed each time in determining the position of leaf inside the target tree. Currently, for each FULL node found in the source tree, the search for its representation in the target tree must start from the root. The complexity of this algorithm is bounded by O(K*n) where K is the number of nodes in the source tree and n is the logarithm of the side length of the modeling space.

In [5], Van Lierop uses the reciprocal of Ahuja's approach to perform arbitrary rotation and/or translation; for each node in the target tree, the algorithm looks for information in the source tree; if sufficient conditions for labelling are found, it is labelled (FULL or VOID); if not, recursion on its children is performed, using a subset of the input quadtree which was used their parent node. The quadtree is stored in leafcode (linear quadtree). In this form, the search space can be represented as two markers which delimit the particular space which is to be searched. The parent node receives a space delimited by two markers, L1,U1, and it passes other markers, L2,U2, to its children with $L1 < L2 < U2 < U1$. In this way the

interval within the leafcode is shortened in each step. The complexity of the algorithm is O(M * (log N + n)) (M=nodes in the output quadtree; n = resolution and N = number of input leaves). One problem which arises when linear quadtrees are used is that the savings in storage space are paid for by sacrificing neighborhood information. This shows up in the time complexity of algorithms using leafcodes. Referring to FIGURE 2,the nodes 13X and 21X are code-neighbors in the leafcode; however, they are not spatial neighbors. If a search was being conducted for some subspace of 20X, the last step in the search involves 13X and 21X, while with pointer representation the node 13X would be discarded earlier. This is because in the point-

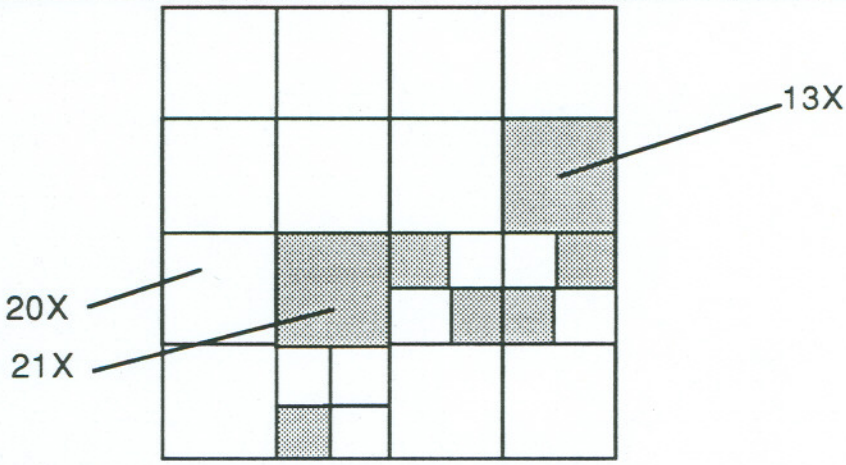


FIGURE 2: Leafcode representation of the object shown.

er-based representation, the branching factor is higher, and only spatial neighbors are considered. The price paid for this advantage is the overhead in storage space.

In [8] and [9], Samet *et al.* attack the problem of representing an array of pixels as a linear quadtree. The linear quadtree is held on disk, and ordering is achieved by maintaining an index, in this case a B-tree. In this way, an inserting or deleting job is done by modifying the B-tree. An advantage of the B-tree is that it remains almost perfectly balanced, and its branching factor is high, which leads to a small traverses in order to get its leaves (which are FULL nodes of the octree). The algorithm calls an "active node" a node with at least one pixel within it already considered, and with at least one pixel within it not considered yet. This strategy "...assumes the existence of a data structure which keeps track of the active quadtree nodes. For each pixel in the raster scan traversal, do the following. If the pixel is the same color as the appropriate active node, do nothing. Otherwise, insert the largest possible node for which this is the first pixel and add it to the active nodes set.

Remove any active nodes for which it is the last pixel." If the metric for measuring the performance of the algorithm is the number of INSERT operations (which implies disk access) the complexity of it would be O(I) (I=Insert operations). However, the Insert operation is not an elementary operation; it being a B-tree management operator which implies rearrangement of the B-tree topology, some times requires operations which involve each level in the structure; therefore it does not seem adequate for measuring algorithm performance.

With the exception of the approach proposed by Meagher, all the approaches incur an extra logM effort (M being the number of nodes in the target tree) because they do not exploit the neighborhood information which is encoded in the octree. Repeated traversal through the tree causes the extra effort. Meagher's approach does so, but incurs the expense of an auxiliary structure (which is a list of obels). The algorithm presented in this paper extends current capabilities in that it presents a unified approach to all geometric transformations. In addition, in terms of computational effort, it matches Meagher's approach (both are linear in the number of target tree nodes) and does so without using any auxiliary structures. It accomplishes this by simultaneously traversing the source tree and building the target tree, profiting from the neigborhood information implicit in the structure.

3. PROBLEM STATEMENT AND ALGORITHM OUTLINE.

In this paper a general transformation algorithm which relies on a representation procedure for pointer-based quadtrees is developed. The algorithm is independent of the transformation(rotation, scaling, translation or any combination) performed on the original tree, and does not require auxiliary data structures to encode neighborhood data.

The basic approach used by the algorithm is as follows: when a transformation is to be performed on an object in a modeling space, an imaginary modeling space is also considered to be transformed with it. The quadtree representing the original object in the real modeling space and that representing the transformed object in the imaginary space are structurally the same. In fact, the latter, which we refer to as $T_{illegal}$ is the image of the former, which we call T_{legal}, after translation, rotation or scaling operations have been performed on each of its nodes, without regards to the validity of the final object as an octree (or quadtree) representation. In other words the transformation:

$$T_{legal} \xrightarrow{R\_T\_S} T_{illegal}$$

(where R_T_S stands for: rotation, translation or scaling, possibly using homogeneous co-ordinate transformations) is performed. The problem can now be considered as one where we attempt to represent the illegal tree as a legal quadtree in the original space. That is,

$$T_{illegal} \xrightarrow{REPRESENT} New\_T_{legal}$$

$T_{illegal}$ has some properties which should be mentioned.

1.- The working space or modeling space, S, of the original octree T_{legal} is not the same as that of $T_{illegal}$.

2.- $T_{illegal}$ is an valid representation of the object in the transformed or imaginary modeling space. The leaves of the tree do not have any spatial overlap.

3.- The resolution level of $T_{illegal}$ could be different from that of T_{legal}. The leaves in $T_{illegal}$ will have whatever dimensions and orientations that

result from the geometric transformation. However, nodes at any level of the tree continue to representing half the dimensions represented by their parent nodes.

Algorithm Outline

In the previous paragraph it was argued that the problem of geometric transformations is essentially a representation problem. The algorithm starts with an object, represented by a tree T on the transformed or imaginary space, and the objective is to obtain the representation of that object in a space S (the original modeling space), which is shifted either rotationally or translationally relative to T, and possibly scaled to some other size.

The first action is based on looking for an easy answer, namely:

If the S and T spaces are completely disconnected, the representation of the object in the space S is VOID. FIGURE 3 shows this situation.

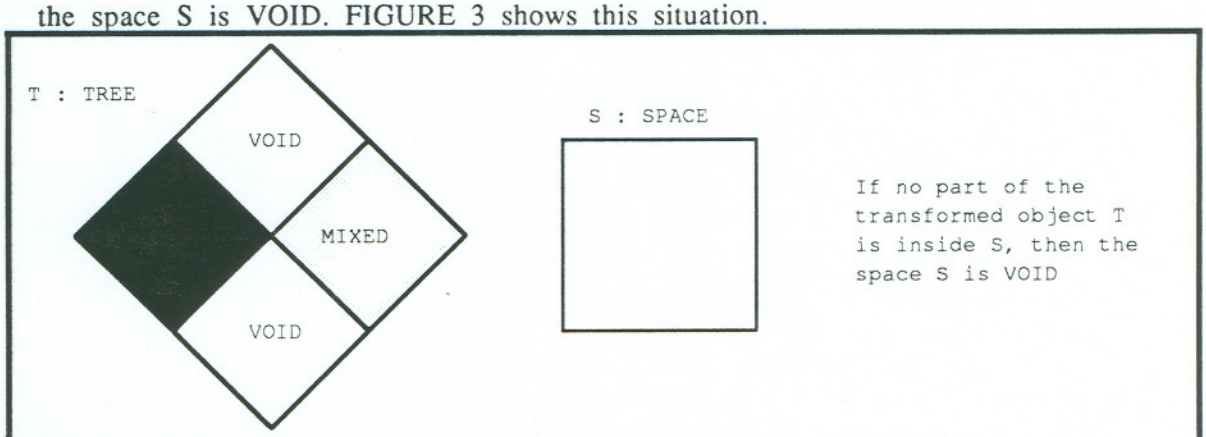


FIGURE 3: Decision of the algorithm REPRESENT() when object and space are disjoint.

If the situation is not so easy, we can reduce the problem size in two ways:

First: It might turn out that all information contained in T is not required for building the quadtree in S. Thus, we look for the minimum part of T which contains all the information to reach a conclusion about its representation in S. This information is in T itself or in some quadrant (or branch) of T. Let's suppose this information is in some tree Tb (Tb will always be a sub-tree in

$T_{illegal}$), which represents some space **Sb** which is a descendent of T. The function MINIMUM_SPACE(*Tree_1*,*Square*):*Tree_2* looks for the smallest space represented within *Tree_1* which contains all the information to fill *Square*, and returns that space expressed in the *Tree_2*. FIGURE 4 shows the motivation for looking in a reduced space.

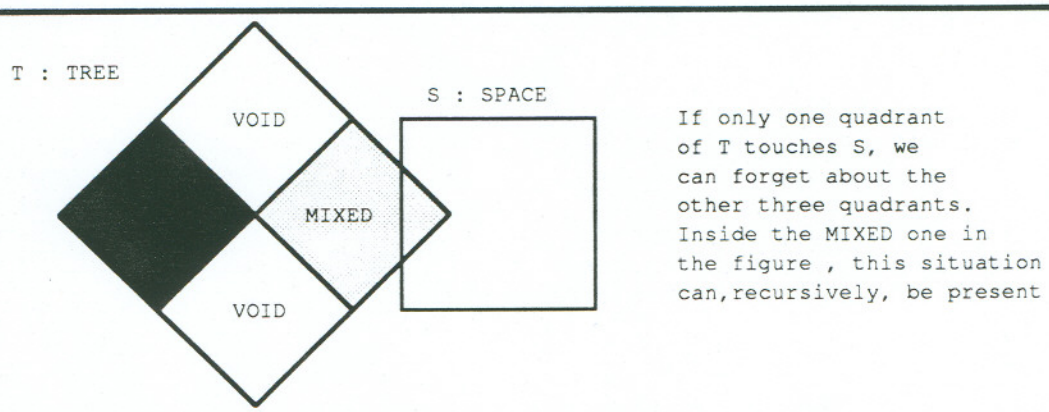


FIGURE 4: Identification of the part of T which has the information to label S

If the S space is completely contained in the **Sb** space and **Sb** is VOID or FULL, then, the S space will take the same attribute. (See FIGURE 5)

If **Sb** has an L-shaped region with uniform label which completely contains S, then S will take that attribute. (See FIGURE 5)

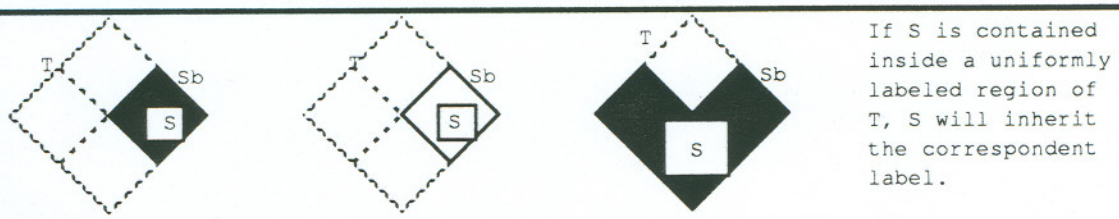


FIGURE 5: Cases in which immediate decision about S can be made.

Second: Even though Sb might be informationally complete to fill S, it is possible that either Sb is not uniformly labelled, or, it does not completely enclose S. In these cases no decision can be reached about S and it is necessary to divide it into four quadrants and try to represent them using the information in Sb. In this way the algorithm recursively proceeds to determine the labels for the children of S while simultaneously reducing the size of the space where

this information is searched for to only the minimum space that is informationally complete to make each decision. (See FIGURE 6)

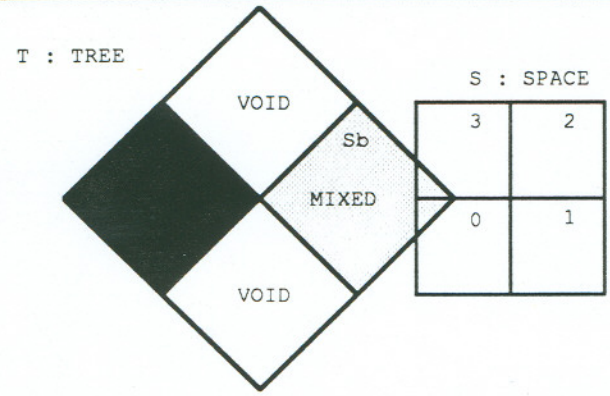


FIGURE 6: Case in which recursion is necessary because either the intersection of space Sb and S does not have uniform label and/or it does not entirely cover S.

Based on the above discussion, the pseudo-code for the macro-algorithm would be:

REPRESENT(T : TREE ; S: SPACE) : TREE;

{S represents a grid in the space; T is the quadtree representation of some object in another displaced and resized grid. The procedure returns the representation of T in the grid S.}

```
1        var TEMPO:TREE;
2    IF  (BEYOND_MAXIMUM_RESOLUTION(S))        ->
3        REPRESENT := ARBITRARY_DECISION(T,S);
4        ELSE      ->
5            IF  COMPLETELY_DISCONNECTED ( T, S )      ->
6                    REPRESENT := VOID;
7                ELSE ->
8                    Tb := MINIMUM_SPACE(T, S);
9                    IF POSSIBLE_DECISION(Tb,S) ->
10                       REPRESENT := LABEL(Tb,S)
11                   ELSE       ->
12                       TEMPO := NULL_TREE;
13                       FOR EACH(Si) Subspace of S      ->
14                           ADD(TEMPO, REPRESENT(Tb,Si));
15                       ROF;
16                       REPRESENT := NORMALIZE(TEMPO);
17                   FI
18           FI
19       FI
```

Before the macro-algorithm is discussed, it is important to address the termination conditions for it. For some situations the representation task could never terminate; this is because for some nodes (those on the edge of the object for example) the algorithm may have to choose to follow the recursive option infinitely. In such a

case, an arbitrary decision must be taken when the algorithm realizes it has gone beyond the minimum grid size chosen for the representation.

Some comments to explain the algorithm are also necessary:

-In line 2, the algorithm recognizes its arrival at a point at which it is necessary to apply some arbitrary criteria to label the current node. It is not necessary for this level to be the maximum resolution chosen for the representation; MAXIMUM_RESOLUTION can be defined so that it allows the algorithm to proceed one or two levels further and use the information collected there to make a decision about the labels to be assigned to the nodes at the required maximum resolution. For example, a node at the maximum resolution can be labelled FULL if three of its four children are labelled FULL.

-In line 9, the boolean function POSSIBLE_DECISION(*Tree*, *Space*) returns YES or NO depending on whether the information found in *Tree* is enough to uniformly label *Space*. This may mean:

- *Tree* completely encloses *Space* and *Tree* has a uniform label (FIGURE 7.a).
- *Tree* is an L-shaped, uniform-labeled region which encloses *Space*. By L-shaped, we also imply the case in which two non-diagonal quadrants have a uniform label (FIGURE 7.b, 7.c).
- *Tree* does not completely enclose *Space*, and perhaps *Tree* does not have uniform label, but the part of *Tree* which intersects *Space* is VOID. As the other part of *Space* must be labeled VOID anyway, we can at this point just label *Space* as VOID (FIGURE 7.d).

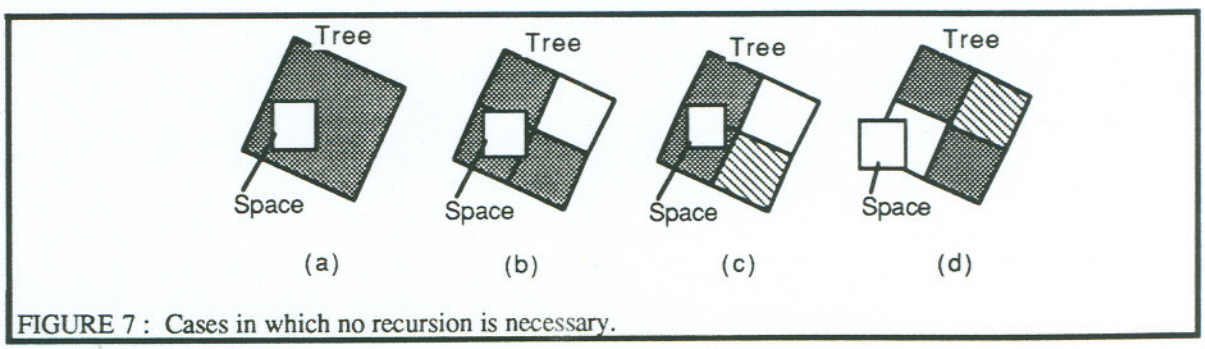
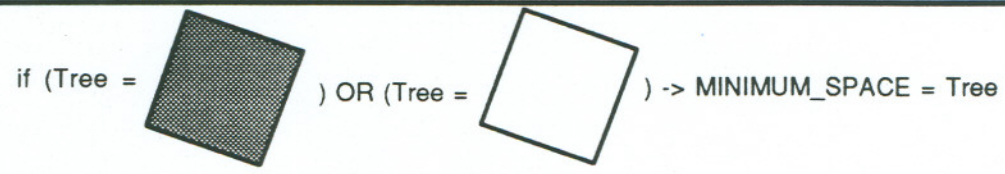
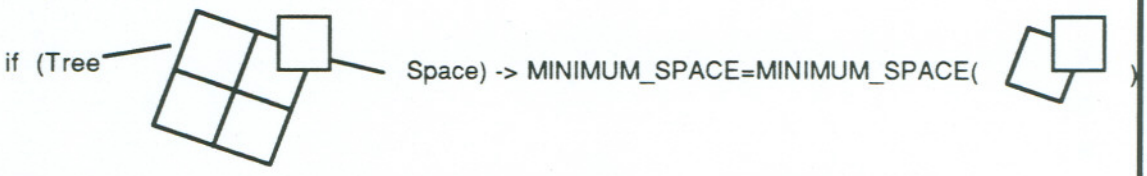


FIGURE 7 : Cases in which no recursion is necessary.

-In line 8, MINIMUM_SPACE(*Tree, Space*) is invoked. The purpose of this function is to find the minimum subspace contained in *Tree*, which is informationally complete for labeling *Space*. See FIGURE 8, in which the different conditions for deciding what the minimum space to be considered should be are shown.



a. Tree is FULL or VOID no smaller space exists.



b. Space has to deal with only one quadrant of Tree. Recursion on such quadrant is necessary.

c. Space has to deal with more than one quadrant of Tree. As an quadtree only can be either decomposed in four children or not to be decomposed at all, no smaller space will be found.

FIGURE 8: Graphical specification of procedure MINIMUM_SPACE().

The procedure is based on making decisions about whether, or not, two rectangles intersect and whether one is included in the other. Since this is a very basic decision that is extensively used by the algorithm, the implementation details are discussed in the appendix.

-In line 16, NORMALIZE(*Tree*) behaves differently, based on whether it is working beyond or within the representation resolution specified for the quadtree. In the former case, the children are deleted and the parent node is given a uniform label irrespective of whether the children have identical labels or not. For example, one criteria that can be used to establish the label of a node is to give it a FULL label if three out of four children are labelled FULL. In the second case, within the

representation resolution, compaction can be performed only if all the children of a node have identical labels. In this case they are deleted and their parent takes their label.

The working of the algorithm involves the simultaneous traversal of the illegal tree and the development of the legal tree. Therefore, neighborhood information is preserved and exploited without the use of any auxiliary data structures. Also, the additional work of traversal/backtracking over the illegal tree is avoided, thus resulting in a linear-time (in the number of nodes of the tree) algorithm, as will be evident in the next section. This algorithm works for transformations of any magnitude which means that no normalization has to be done on the translation distance or the scaling factor. Also, any rotational angle is possible. FIGURE 9 shows a very simple, initial object; FIGURE 10 shows it after performing a translation of -30% of the length of the modeling space in both axis ; FIGURE 11 shows the object after being scaled by a factor 0.5 and rotated (about the center of the space) by 45 degrees. Notice in FIGURE 10, the superposition of the illegal object, partially out of the modelling space, and its representation (legal) in which only the part of the object which stays in the original space or universe is represented by the algorithm.

4. COMPLEXITY AND EXPERIMENTAL RESULTS.

The algorithm having been discussed and results demonstrated, the next problem to be addressed is that of estimating the performance of the algorithm. The worst-case performance in terms of basic operations that the algorithm might have to perform when performing a transformation is estimated. An assumption is made that the number of nodes in source and target trees are comparable at least in order of magnitude.

Let, at any instant, the algorithm be dealing with a target space S; then the computational task can be decomposed into two parts:

- Four calls for representing a S/4 space.
- One call MINIMUM_SPACE. Let's suppose the worst conditions occurs and the call has to travel to the bottom of the tree to find the minimum space.

Therefore, the total effort is given by:

$$T(S) = 4T(S/4) + K1*\text{Log } S$$

if a variable y is used, such that,

$$4^y = S,$$

then

$$4^{y-1} = S/4 \quad \text{and} \quad \log S = K2 * y$$

therefore

$$T(4^y) = 4T(4^{y-1}) + K3 * y$$

If the function $T(4^y) == f(y)$; therefore

$$f(y) - 4*f(y-1) = K3*y$$

Taking the z-transform, we have:

$$F(z) - 4F(z)/z = K3*z/(z-1)^2$$

then

$$F(z)*(1 - 4/z) = K3*z/(z-1)^2$$

solving for F(z):

$$F(z) = K3* z^2/(z-1)^2(z-4)$$

which can be decomposed as

$$F(z) = A/(z-4) + B/(z-1) + C/(z-1)^2$$

or

$$F(z) = Az/(z-4)z + Bz/(z-1)z + Cz/z(z-1)^2$$

using the inverse z-transform, the following is obtained:

$$f(y) = A * 4^{(y-1)} + B + C*(y-1)$$

and substituting for the function f,

$$T(4^y) = A * 4^{(y-1)} + B + C*(y-1)$$

Since 4^y is proportional to the number of nodes in the tree, M:

$$T(M) \cong K * M + B + K \log M$$

Which means

$$T(M) \cong O(M)$$

To experimentally verify the complexity of the algorithm, the pointer follow-up operations performed in the procedures REPRESENT() and MINIMUM_SPACE() were counted. This was done to make the data collected independent of the hardware on which the programs were run (unlike the situation in which execution time is observed). The pointer follow-up operation counts the number of displacements within the source tree (MINIMUM_SPACE()) and the target tree (REPRESENT()). It can also be seen as the number of recursive calls made during the execution of the procedures.

Five different test cases were run. Each had a different number of nodes in the starting or source tree and this varied from 1 node to 17 nodes. Three of these are shown in the FIGURE 12. In FIGURE 12a the internal representation for the space shown is also displayed. EMPTY or VOID nodes are stored as NULL pointers while FULL ones are stored as FULL-labeled spaces with all four children NULL. For each test case, a 13 degree rotation was performed (See FIGURE 13) This was repeated for different levels of resolution (or depths of the quadtree) starting from 3 (with the possibility of having at most 64 leaves) and going up to 7 (16,384 leaves) levels. The graph in FIGURE 14 shows the results of the algorithm. Since the number of levels in the representation is proportional to the log of the number of nodes in the tree, the results have been presented as *Logarithm of Operations Vs. Number of Levels*, however they can be read as *Logarithm of Operations Vs. Logarithm of Number of Nodes*.

An examination of the graph reveals a linear relationship between the number of elementary operations and the nodes, thus verifying the linear relationship that has been predicted by the complexity analysis. Another important observation that can be made from the graph is the dependency of the number of elementary operations on the number of nodes in the source tree for smaller source

trees. This is because the assumption in the complexity analysis about the nodes in the source and target trees being of the same order of magnitude is not satisfied. However, it can be observed that as the number of nodes in the source tree increases this dependency is lost. For example, the lines corresponding to 9, 13 and 17 source tree nodes show no particular order.

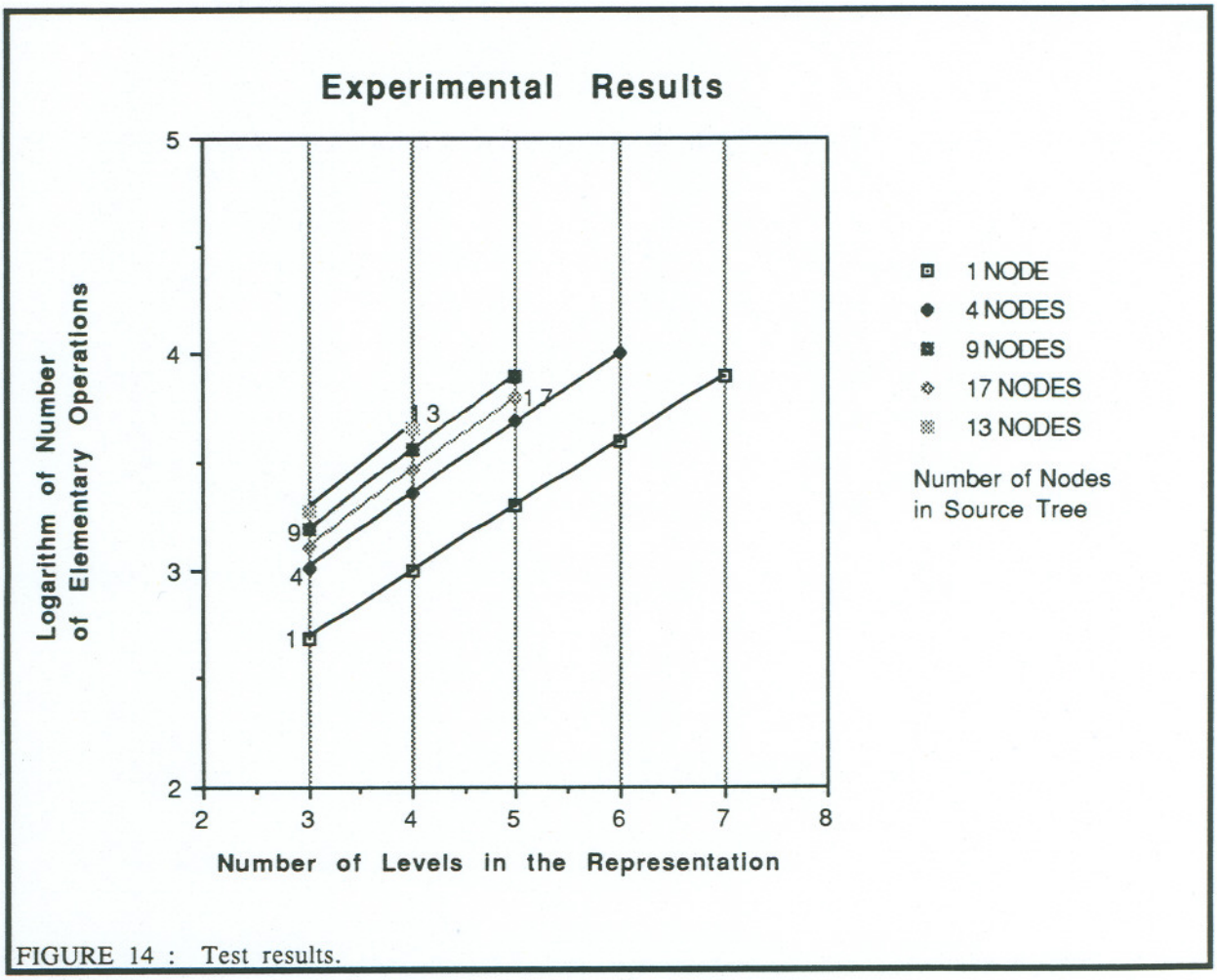


FIGURE 14 : Test results.

5. CONCLUSIONS.

In this paper a general algorithm has been presented for geometric (rotation, translation and scaling) transformations of quadtrees. The algorithm uses a representational approach in that it seeks to represent the known quadtree of an object in one modeling space in another which is displaced or scaled relative to it. The algorithm is efficient in the sense that the computational effort is of the order of the number of nodes in the target quadtree (octree) which can, in fact, be considered the lower bound on the work required to perform a geometrical transformation. This efficiency is obtained by simultaneously traversing the source tree and building the target tree. This simultaneous traversal and building saves the LogM (M being the number of nodes on the tree) effort of searching one of the trees. Though the algorithm has been implemented for quadtrees, the work extends naturally to octrees. The only major change that would be required would be the MINIMUM_SPACE procedure which would now have to use cubes instead of squares. Finally, since the algorithm avoids backtracking and since at each stage either a decision is made or four (eight for octrees) independent problems spawned, the algorithm should lend itself well to parallel processing, an aspect we are currently investigating.

Appendix

In an section 3, it was stated that procedures *Possible_Decision* and *Minimum_Space* are considerably employed and hence their efficiencies are fundamental to the efficiency of the entire algorithm. The basic task performed by these procedures is testing the intersection of two rectangles and the possible inclusion of one in the other. Since the construction of the intersection is not of interest, the traditional calculation of line intersections does not make much sense here. Also, even if the points of intersection were known, this does not consider the possibility of one rectangle being included in the other. Checking for all the possibilities becomes inefficient and therefore a faster approach is developed to suit the specific conditions of the problem on hand.

The basic idea behind the functions used is that a line (in 2D) divides the space into two disjoint regions. Given that, one can easily test whether two points are in the same half space, in opposite half paces, or one or both lie on the half-space boundary, it is possible to test whether a line segment is completely or partially confined to one half space. The next step simply makes use of the fact that a rectangle is the convex intersection of four half spaces, and therefore inclusion in it is based on inclusion in the half spaces.

To test for boolean relations between two rectangular regions in a 2D space, two abstract data types where implemented. Those two data types, SQUARE and STRIP, have meanings and properties which are intuitively obvious.

The object STRIP is the 2D space between two lines; we can define the following operations on it:

INTERS_SQUARE : STRIP x SQUARE --> BOOLEAN

{This operation takes an strip and a square; and returns YES or NO depending on whether the strip intersects the square. See FIGURE 15}

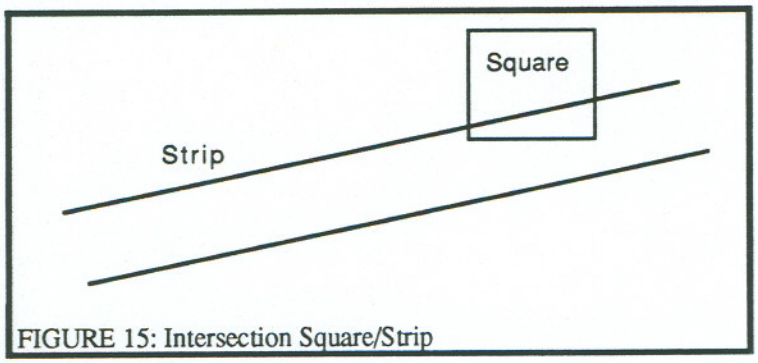


FIGURE 15: Intersection Square/Strip

On the object Square, formed by the intersection of two strips in a 2D space, define the following operations:

STRIP1 : SQUARE --> strip

STRIP2 : SQUARE --> strip

{the operations strip1 and strip2 return the two strips whose intersection make the square See FIGURE 16 }

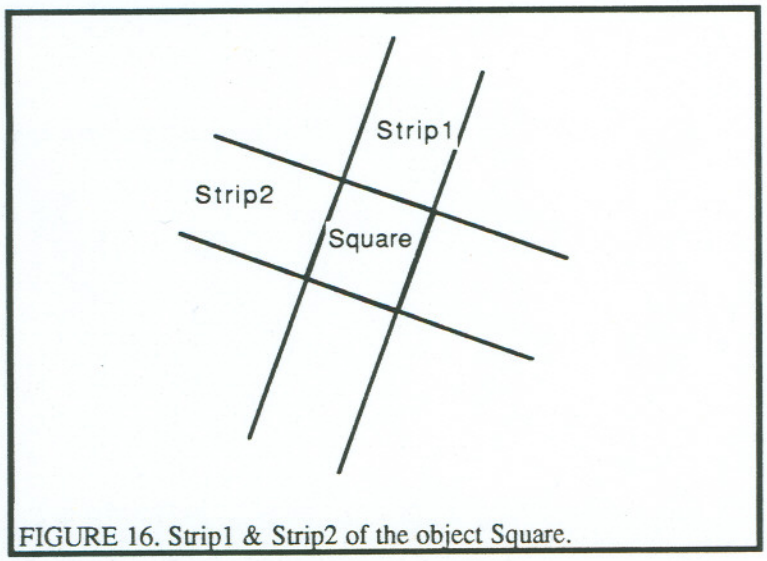


FIGURE 16. Strip1 & Strip2 of the object Square.

Based on the data types and operations shown, a very easy test for intersection of rectangular regions which considers all possible cases of intersections can be developed, based on the following theorem:

THEOREM:

Let Sq1 and Sq2 be two objects of the type SQUARE; then Sq1 intersect Sq2 iff

INTERS_SQUARE(STRIP1(Sq1), Sq2) and

INTERS_SQUARE(STRIP2(Sq1), Sq2) and
INTERS_SQUARE(STRIP1(Sq2), Sq1) and
INTERS_SQUARE(STRIP2(Sq2), Sq1).

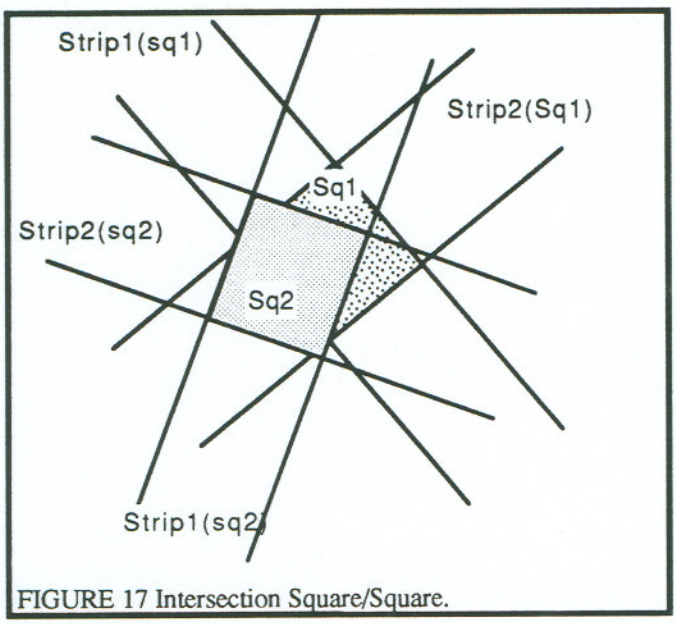


FIGURE 17 Intersection Square/Square.

PROOF:

Notation:

Str11 = STRIP1(Sq1);
Str12 = STRIP2(Sq1);
Str21 = STRIP1(Sq2);
Str22 = STRIP2(Sq2);

Sq1 INTERSECT Sq2 iff
$Sq1 \cap Sq2 \neq \Phi$ iff
$(Sq1 \cap Sq2) \cap (Sq1 \cap Sq2) \neq \Phi$ iff
$(Sq1 \cap Sq2) \cap ((Str11 \cap Str12) \cap (Str21 \cap Str22)) \neq \Phi$ iff
$(Sq1 \cap (Str21 \cap Str22)) \cap (Sq2 \cap (Str11 \cap Str12)) \neq \Phi$ iff
$(Sq1 \cap Str21) \cap (Sq1 \cap Str22) \cap (Sq2 \cap Str11) \cap (Sq2 \cap Str12) \neq \Phi$

Figure 17 shows the intuitive idea behind the theorem. It was used to test intersection of squares in the algorithm. Floating point operations were used, expressing the lines as vectors. No intersections were explicitly calculated and tests

for intersection strip/square were based in cross and/or dot product. If further speedups are required, this theorem can easily be implemented using the idea proposed by Meagher [2].

References:

1. Jackins, C. and Tanimoto, S., "Octrees and their Use in Representing Three Dimensional Objects," Computer Graphics and Image Processing. Vol 14, 1980, pp 249-270.
2. Meagher, D., "Geometric Modeling Using Octree Encoding," Computer Graphics and Image Processing. Vol 19, 1982, pp 129-147.
3. Ahuja, N. and Nash, C., "Octree Representations of Moving Objects," Computer Vision, Graphics and Image Processing. vol 26, 1984, pp 207-216.
4. Gargantini I., "Translation, Rotation and Superposition of Linear Quadtrees," Int J. Man-Machine Studies. Vol 18, 1983, pp 253-263.
5. Van Lierop, M., "Geometrical Transformations on Pictures Represented by Leafcodes," Computer Vision, Graphics and Image Processing. Vol 33, 1986, pp 81-98.
6. Weng, J. and Ahuja, N., "Octrees of Objects in Arbitrary Motion: Representation and Efficiency," Computer Vision, Graphics and Image Processing. Vol 39, 1987, pp 167-185.
7. Shaffer, C. and Samet, H., " Optimal Quadtree Construction Algorithms," Computer Vision, Graphics and Image Processing. Vol 37, 1987, pp 402-419.
8. Samet, C., Shaffer, C., Nelson, R., Huang, Y., Fujimura, K. and Rosenfeld, A., "Recent Developments in Linear Quadtree-based Geographic Information Systems," Image and Vision Computing. vol 5, 1987, pp 187-197.
9. Samet, H., "Hierarchical Data Structures and Algorithms for Computer Graphics," IEEE Computer Graphics & Applications. May 1988.
10. Samet, H., The Design and Analysis of Spatial Data Structures. Addison Wesley, 1989.
11 Samet, H. Applications of Spatial Data Structures. Addison Wesley, 1989.
12. Requicha A., "Representation of Rigid Solids: Theory, Methods and Systems," ACM Computing Surveys. Vol. 12, #4, December 1980, pp 437-464.
13. Requicha, A., "Mathematical Model of Rigid Solid Objects, " Production Automation Project Technical Memo # 28. University of Rochester. Nov. 1977. NY 14627.
14. Requicha, A. and Voelcker, H., "An Introduction to Geometric Modeling and its Applications in Mechanical Design and Production," Advances in Information Systems Science. Vol 8. Editor Julius Tou, Plenum Publishing Corporation. 1981. pp 293-321.

15. Brown, H., Requicha, A. and Voelcker, H., "Geometric Modeling Systems for Mechanical Design and Manufacturing," ACM Conference 1978. pp 770-778.
16. Kawaguchi E., Endo T., "On a Method of Binary-Picture Representation and its Application to Data Compresion," IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol PAMI-2, No 1, January 1980, pp 27-35.
17. Kawaguchi W., Endo T. Matsunaga J, "Depth First Picture Expression Viewed from Digital Picture Processing," IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol PAMI-5, No 4, July 1983, pp 373-384.