Geometric Transformations in Octrees using Shears

Carlos Saona*, Isabel Navazo and Àlvar Vinacua

10th December 1997

Abstract

Existent algorithms to perform geometric transformations on octrees can be classified in two families: inverse transformation and address computation ones. Those in the inverse transformation family essentially resample the target octree from the source one, and are able to cope with all the affine transformations. Those in the address computation family only deal with translations, but are commonly accepted as faster than the former ones for they do no intersection tests, but directly calculate the transformed address of each black node in the source tree. This work introduces a new translation algorithm that shows to perform better than previous one when very small displacements are involved. This property is particularly useful in applications such as simulation, robotics or computer animation.

1 Introduction

Octrees are one of the most common hierarchical data structures in three dimensional space. They have been successfully used in many different, areas, such as solid modeling, robotics and collision detection, to name a few. Though, they have some drawbacks, mainly its size and poor performance in geometric transformations, specially when compared to b-rep models.

As stated in [1], existent geometric algorithms can be classified in two categories: those that do intersection tests (also called inverse transformation ones) and those that do address computation. Algorithms of the first family ([2], [3]) work with two octrees resident in memory, and, essentially, resample the source octree. Those of the second family ([4],[5],[6],[7] and [8]) traverse the octree computing the destination address or path of each black node and moving it accordingly, so they do not need to maintain two octrees in memory, but one. As they do no intersection tests, they are widely accepted as the faster ones. However, resampling algorithms are more flexible, as they are able to translate, rotate and scale; on the contrary, path computing ones only deal with translations.

*This work has been partially supported by an FPI research grant of the Ministerio de Educacion y Ciencia and the CICYT project TIC 95-630-C05

This work focuses on the translation of octrees. Section 2 studies the path calculation algorithm in more depth. Section 3 introduces a new path computation algorithm whose performance when small translations or rotations are involved is shown to be better than previous ones. We think that this behaviour would make it a better choice in applications such as computer simulation, walkthrough or robotics, where objects do not change their positions in capricious ways, but move continuously from one place to another. Section 4 studies the theoretical cost of our approach, and provides some empirical tests. Finally, in appendix A we show how to transform a rotation into a series of shears, a common technique in other areas, but that had never been used to rotate octrees. As an algorithm to perform shear operations in octrees by means of address computation is also presented, we effectively demonstrate that path computation techniques are not limited to translation.

2 Address-computation translation algorithms

There are many translation algorithms that do path computation: the simplest one was presented in [4]. First, it decomposes each black node into nodes of width one, and then computes their destination addresses. As a final step, the tree is condensed. Later, [5] took profit of the fact that when the translation distance is a multiple of the length of the node it is not necessary to fragment the node down to unit length; when this condition does not hold, however, the algorithm in [5] decomposes the node until it holds, so the asymptotic cost does not change. There is a later improvement of [5] in [6]. There, the authors prove that there is no need at all to fragment black nodes prior to translation - as this method is the basis of our incremental algorithm it is explained in more depth below. All these algorithms deal with translations parallel to one of the coordinate axes, so three runs are needed for arbitrary translations. The algorithm in [8], though, does not require axis parallelism, so arbitrary translations can be performed in just one step. Besides, this algorithm generalizes the one in [7]. A good survey of all the works cited above can be found in [ll]. An explanation of the work in [5] and its sequel in [6] follows.

The key of the algorithm is the computation of the destination address of a node. Consider the octant labeling of figure 1, and a unit length node n whose address or path from the root node is $n_l \ldots n_1$. Also consider table 1(SYM), that represents octant symmetries along coordinate axes (for instance, octants 1 and are symmetric along Y axis), and table 2(EXT), that shows whether a node and its neighbour in some direction have different parents (i.e., they are not brothers).

If we translate n one unit in direction e it is easy to see that, if $\neg\mathrm{EXT}[e][n_1]$, the destination node address will only differ from the original one in the last digit, that is to say, will be $n_l \ldots n'_1$. Moreover, it is quite obvious that $n'_1 = \mathrm{SYM}[e][n_1]$. Otherwise, if $\mathrm{EXT}[e][n_1]$ holds, the destination address will be $n_l \ldots n'_2 n'_1$, where $n'_2 = \mathrm{SYM}[e][n_2]$ and $n'_1 = \mathrm{SYM}[e][n_1]$. Note that, in this example, $\mathrm{EXT}[e][n_1]$ has had a "carry" effect on the second digit.

It can be proved that the destination address of node $n_l \ldots n_1$ when translated in direction e by t expressed in binary form as $t_l \ldots t_1$, can be computed

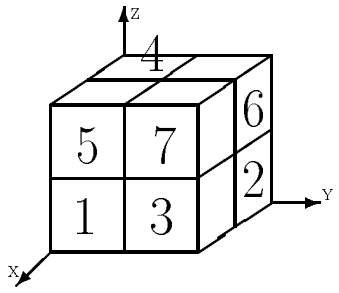


Figure 1: Labeling of octants (octant 0 is at the origin).

as an "additive" operation whose behaviour is completely defined by tables 1 and 2. Table SYM gives the result of "adding one", and table EXT rules whether to "carry" (figure 2 shows an example). More precisely, we can compute the destination address $n'_l \ldots n'_1$ as

$$
\begin{aligned}
n'_i &= (n_i \oplus t_i) \oplus c_{i-1} \\
c_0 &= 0 \\
c_{i+1} &= \begin{cases} 1 & \text{, if } (\mathrm{EXT}[e][n_{i+1}] \wedge t_{i+1} + c_i = 1) \vee (t_{i+1} + c_i = 2) \\ 0 & \text{, otherwise} \end{cases} \\
x \oplus y &= \begin{cases} \mathrm{SYM}[e][x] & \text{, if } y = 1 \\ x & \text{, if } y = 0 \end{cases}
\end{aligned}
$$

Furthermore, it is not strictly necessary for n to have length one in order to calculate its destination address. As a matter of fact, it will always be possible as long as t is a multiple of the length of n. Say $n_l \ldots n_k$ is the address of n, so its length is 2^{k-1}. As $t \equiv 0 \pmod{2^{k-1}}$, we can state that the binary form of t will be $t_l \ldots t_k 0 \ldots 0$. To compute $n'_l \ldots n'_k$, just discard the first $k-1$ least significant bits of t and proceed as if n had unit length.

Unfortunately, when t is not a multiple of n's length, it is not so easy to compute its destination, for it is not unique. In fact, not being a multiple of n's length implies that the node will break up when translated. One solution to this problem, as given in [5], is to recursively split n until t is a multiple of its descendants length. Luckily enough, as [6] shows, there is a better solution.

Consider two nodes n_i and n_s whose length divides t and that are located at two opposite corners of node n. You can compute the address of $n_i(n_s)$ by adding $0(7)$ to the right of n's address till is a multiple of its length . To move n, first translate n_i and n_s and then fill all the nodes between them.

Observe also that checking whether t is multiple of the length of a node can be done in $0(1)$ time. Due to the octree structure, every node $n = n_l \ldots n_k (1 \leq k \leq l)$ has length 2^{k-1}; therefore, $t \equiv 0 \pmod{2^{k-1}}$ holds iff $t = t_l \ldots t_k 0 \ldots 0$. So, prior to translation, compute the index h of the first bit of t's binary representation that is non nil. Then, t will be multiple of the length of a m level node n iff $m \leq h$ (say root is at level 0).

Finally, to translate the octree, traverse the tree in a front to back fashion

	0	1	2	3	4	5	6	7
X	1	0	3	2	5	4	7	6
Y	2	3	0	1	6	7	4	5
Z	4	5	6	7	0	1	2	3

Table 1: SYM. Octant axial symmetries along each axis.

	0	1	2	3	4	5	6	7
+X	F	T	F	T	F	T	F	T
+Y	F	F	T	T	F	F	T	T
+Z	F	F	F	F	T	T	T	T

Table 2: EXT. An octant and his neighbour in direction $+e$ have different parents. To get the $-e$ table, negate.

moving the black nodes as noted above. Once all the nodes are translated, compress the tree.

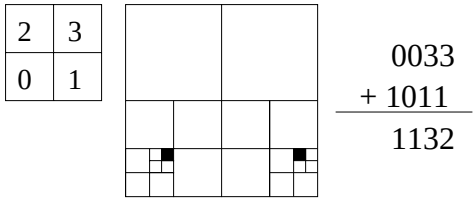


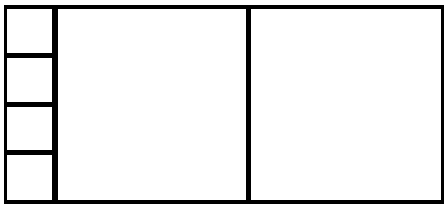
Figure 2: A 2D example of address computation: node 0033 is translated 11 units in $+X$ direction

3 The Incremental Translation Algorithm

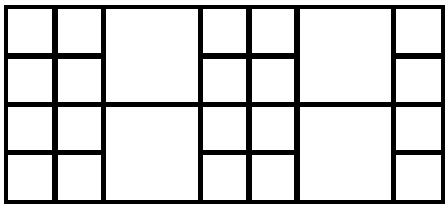
Suppose a situation as the one depicted in figure 3, where we want to move one unit to the right a quadtree formed by two big nodes surrounded to the left by a column of minimum nodes. The classical translation algorithm will first move the big nodes, then the set of smaller ones, and, finally, compress the tree. The displaced quadtree will be identical to the original one, except for the minimum nodes. As a matter of fact, the number of nodes is exactly the same. Unfortunately, the translated tree before compression will be significantly larger, as each one of the big nodes breaks itself into pieces when moved one unit to the right.

Moreover, it is quite obvious that every octree that tries to represent a solid object will have a structure similar to the one in figure 3, the smaller nodes being at the object's boundary, and getting bigger as we approach the interior. Therefore, we can safely state that the classical algorithm will perform badly in terms of memory when tiny displacements are involved. Still worse, every odd displacement will produce an unnecessary growth of the tree.

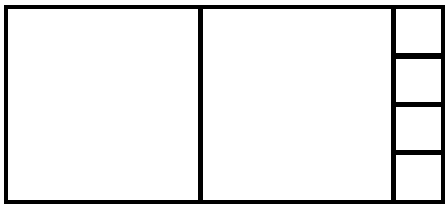
The key of the algorithm that we propose is to try to make the minimal changes to the source octree when translating. If we take another glance at the



(a) Source quadtree



(b) Translated quadtree, before compression



(c) Translated and compressed quadtree

Figure 3: A simple example of the classical translation algorithm. The quadtree in (a) is translated to the right.

figure 3, it is easy to see that there is no need at all to touch the big nodes in the source quadtree; it suffices to translate the smaller ones, The ideal situation is shown in figure 4, where we have many equally sized nodes aligned in the direction of the translation and the magnitude of the displacement, say, t, is a multiple of the length of the nodes, say l. It is quite easy then to perform a memory optimal translation: just make the leftmost t/l nodes jump to the end of the row. Two questions arise now: when it is possible to do that kind of trick and if it is always worth the effort.

A complete case analysis follows. Suppose we want to move t units a node n, whose edge is l units long. The first test we should do is to check the colour of node's neighbour n' in the direction of the translation, whose edge's size is l'.

1. The node's neighbour is black. Due to the octree structure, we can assert that the neighbour's size will be greater or equal to the size of n. Different situations arise depending on the relationship between t and l.

   (a) If l is equal to t, then, as argued before, the best alternative is to make n "jump" over successive black nodes: travel to the right till you find a grey node of size l or a white one bigger or equal than

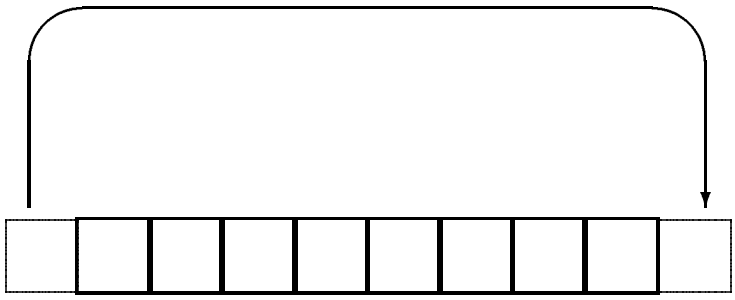


Figure 4: The perfect case. A translation of a row of equally sized nodes of length equal to the magnitude of the displacement.

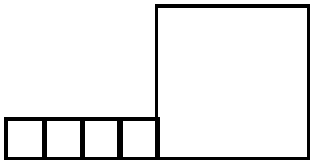


Figure 5: To jump or not to jump.

n. This would be the perfect solution if all the "jumped" nodes were of size l. If this is not the case, we cannot be positive about its optimality, as figure 5 reflects: if the translation is equal to the size of one of the small nodes, the best thing to do is to jump; but, if it is equal to the size of the big node, classical translation is better.

This problem, though, is not too serious, as it does not represent the common situation. Typically, big black nodes (those whose fragmentation should be avoided) are surrounded by the smaller nodes that represent the frontier, so if a black node has a bigger black node as a neighbour, chances are that the situation will be as in figure 3, and not as in figure 5.

(b) When l is strictly greater than t, there are two choices. The first one, to break node n into eight sons and recursively move the leftmost ones; the second one, to follow the classical algorithm. Once again, though, the best course to take is the first one. If we are to move t units two consecutive black nodes, of sizes l and l' respectively, it is perfectly clear that the classical algorithm will split up both of them, as $t < l \leq l'$. But the fact that $t < l$ guarantees that the neighbour node needs not to be touched, that is to say, node n' will remain in the translated tree after compression.

Moreover, it is quite possible that these two nodes are not isolated, but part of a longer row of black nodes, all of which will suffer from unnecessary splitting. As a matter of fact, node $n_{,}$, the leftmost one, is the only node whose fragmentation is unavoidable. So, the best action to take is to recursively move the sons of n.

(c) If t is strictly greater than l, the situation becomes somehow more complex. On the one hand, we could make n jump and recursively

move n' by $t-l$. Again, this would be the best choice if n and n' were in a big row of nodes. On the other hand, it is also possible that t is not only bigger than l, but also than the length of that hypothetical row; for instance, imagine what will happen with two l sized nodes that had to be moved t, where $t \gg l$: they will alternatively jump each other again and again; an obvious waste of time.

More precisely, a row of N nodes each of length l requires N translations to move t units (perhaps involving some unnecessary fragmentation), opposed to t/l "jumps". Thence, if $t < N \cdot l$, it pays off to do the displacement incrementally. Otherwise, it is simpler to do it in the classical fashion. Moreover, to prevent fragmentation in the last case, you could translate $t \operatorname{div} l$ units classically and then $t \bmod l$ incrementally.

2. The node's neighbour is white. As in the previous case, we can assure that the size of the neighbour is greater or equal to the size of n. It is quite obvious, then, that, the node is isolated, and that there is nothing to do but translate it in the classical manner.

3. The node's neighbour is grey. In this case, we do not really know if n is isolated or not; both cases are possible. If it really is, classical translation of n and recursive analysis of n' are preferable. If it is not isolated, we could think of breaking n and recursively analyze its sons, hoping that incremental movement would then be possible. Unfortunately, as testing if n is isolated or not would be too costly, other criteria should be considered.

   It is a fact that, as n' is not black, the movement of n and n' will result in either two grey nodes, if $t \mod l \neq 0$, or in one black node and another grey one, if $t \mod l = 0$. Therefore, as our intention is not to yield unnecessary fragmentation, the optimum solution would be to translate n and n' classically by $t \operatorname{div} l$ and then by $t \bmod l$ incrementally. Sadly, this solution implies the cohabitation of three types of nodes: unmoved nodes, partially moved ones and completely moved ones. The existence of partially moved ones will excessively complicate the tree traversal, so we should think of another solution.

   Our proposal is to test if $t < l$. If the answer is affirmative, we can be sure that n will have to break, so it is safe to split it and move its leftmost sons. Otherwise, we will incrementally move n' and displace n classically, taking the chance of splitting it unnecessarily. Luckily, this would rarely happen, because in a normal octree the smaller nodes are in the object's frontier, so a black node with a grey right neighbour will probably have a left neighbour whose colour is grey or black. As our algorithm is intended to work with small translations, chances are that, the left, neighbour will jump over n.

Now that we have an incremental movement procedure to move a node, we can think of an appropriate tree traversal. Due to the relevance of the neighbourhood property in our node movement algorithm, it is natural to think of a back to front or a front to back traversal. We will describe the former, but the latter is also possible. In fact, both alternatives are equivalent, neither of them being better than the other.

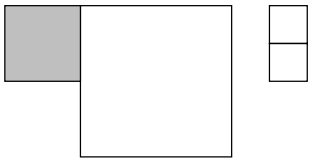


Figure 6: A collision due to the back to front traversal. The shaded node cannot jump to the right.

If we want to move the octree in a back to front fashion, there are two topics we must face:

- As we are traversing in a back to front manner, we can find that, after computing the destination node of a node n, it is not free. The solution, though, is simple: as we have cornputed the destination node, we can recursively move it before effectively translating n, in order to avoid collisions (an example of this situation is depicted in figure 6).

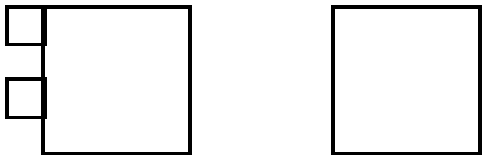


Figure 7: The order problem.

- The other traversal problem is "What should we do after translating a node?", and it is shown in figure 7. After moving one of the smallest nodes (by jumping over the big one), we have two paths opened: either we could go on, and move the other big black node, or go back to move the other small node whose translation was pending. Whatever we do, we should better ensure that the second big black node is not moved twice: once after moving one of the small nodes, and once again after moving the other. The first solution to this problem is tagging. After we have moved the second big node to the right, we tag its previous position as "moved". Of course, we will have to colour all the tagged nodes as white after the entire octree has been translated, but as tree compression is also necessary, we can couple both tasks. The second solution is never to go up in the tree, that is to say, never translate a node of greater length than the one you have already moved, splitting if necessary. Of course, the latter solution increases the fragmentation, and is not adequate if our only concern is to translate the octree, but it shows useful when we want to perform octree shearing.

An algorithm in pseudo-code form is presented in figure 8.

```
Procedure move (VAR o:octree,t:integer) is
 p: stack of node;
 q: stack of <node,int>;
 id,next: node;
 i,L,l: integer;
 next?: boolean;

 p:= push(p,root(o));

 while ~empty(p) do
    id:= top(p);
    pop(p);
    case colour(id)
     is GREY
        for i:=1 to 8 do
         if leftmost(i) then
          p:= push(p,son(o,id,i))
         end if
        end for
     is WHITE
        tag_node(id);
        <next?,next>:= get_next_node(o,id);
        if next? then p.push(next) end if
     is BLACK
        <L,next>:= compute_jump(o,id);
        if ~(colour(next)=MOVED) then
         p:= push(p,next)
        end if;
        q:= push(q,<next,L>)
    end case
 end while
```

```
 while ~empty(q) do
  <id,L>:= top(q);
  if L=0 then
   classical_translation(o,id)
  else
   l:= length(id);
   if l=t then jump(o,id,L)
   else if l>t then
    break_node(o,id);
    for i:=1 to 8 do
     if leftmost(i) then
      q:= push(q,<son(o,id,i),L>)
     end if
    end for
   else if L>t then
     break_node(id);
     for i:=1 to 8 do
      if leftmost(i) then
       p:= push(son(o,id,i))
      end if
     end for
   else
    move_tube_classically(id,L,t)
   end if
  end if
 end while
end Procedure
```

Figure 8: Incremental translation algorithm.

4 Results

Say n is the number of nodes of the octree. Then, the number of black nodes is $O(n)$. Let us take a closer look at the incremental translation algorithm two main loops(figure 8). The first one traverses the tree in a way that resembles the back-to-front octree traversal. Anyway, the fact, that visiting a grey node implies visiting its leftmost sons and that getting to a terminal node implies visiting its right brother lets us conclude that every node is visited at least once. As for the `get_next_node` procedure, it can be implemented as a computation of the translation of the node by l units, where l stands for the length of the node. Because the address computation time is a linear function of the length of the address, its asymptotic cost is $O(n)$. Similarly, `compute_jump` has also $O(n)$ time -just imagine a quadtree similar to the one in figure 3 but that has $n/2$ tiny black nodes and $n/2$ big black nodes-. So, the worst case cost of the first loop is $O(n^2)$.

The second loop behaves in a similar manner. The initial number of nodes in `q` is $O(n)$. Although new nodes can be pushed later, it is not difficult to prove that this increment does not change the asymptotic size $O(n)$. Thus, the loop is executed $O(n)$ times. As for the inner procedures, `jump` has cost equal to the depth of the tree, i.e., has $O(n)$ cost. Finally, `move_tube_classically` has also linear cost. This yields a $O(n^2)$ asymptotic cost for the second loop. Thus, the incremental algorithm has asymptotic $O(n^2)$ cost.

However, the average case is not so bad. Although the depth of the tree can be linear, this is quite an abnormal situation; usually, it will be $O(\log n)$. Moreover, [9] and [10] prove that the average number of nodes that have to be

c/i	+(1,1,1)	+(2,2,2)	+(3,3,3)	+(4,4,4)	+(5,5,5)
Sphere	3.7/2.4	2.2/3.1	3.8/3.7	2.2/4.3	4.45/5.1
Cactus	41/25	20/31	40/40	19/40	44/54

Table 3: Time costs in seconds(c/i stands for classical/incremental)

visited when getting a neighbour is $O(1)$. Thus, the average cost is $O(n \cdot \log n)$. Furthermore, due to memory costs, octree's depth is always limited, and it rarely exceeds 16 levels, so we can think of it as a constant. The average cost, then, is $O(n)$.

Finally, tables 3 and 4 depict some empirical results. Tests were done on a Sun SPARCstation 20, and show CPU time as reported by the system and the number of black nodes after translation but before compression. Two octrees, Sphere and Cactus, were moved $+k$ units in the three directions - that is to say, the translation vector was (k, k, k)-. The sphere is a 6-level octree with 1233 black nodes; the cactus is a bigger, non-convex 9-level octree with 8344 black nodes. The number of black nodes of the translated octrees after compression was never greater than 1300 and 8400, respectively.

Table 3 shows that the incremental algorithm performs quite better than the classical one for minimum translations, that is to say, those whose components are ± 1. Otherwise, its time cost is similar (for translations that are both small and odd) or worse (for even translations). The reason why even translations perform so poorly when compared to the classical algorithm could be that they cause less fragmentation than odd ones when moved classically, so the overhead of the incremental algorithm prevails against the benefits of avoiding extra fragmentation. However, table 4 shows that this is not the case. As a matter of fact, the incremental algorithm provoked more fragmentation than the classical one, and this is likely the reason why it runs slower.

5 Future work

Empirical tests seem to prove that the incremental translation algorithm performs better in terms both of memory and time, but only when very small movements (tiny, in fact) are involved. It is reasonable to think that some improvements to the algorithm could be done in order to allow that not so tiny displacements were also performed faster. Moreover, tests would gain value if some real octrees, i.e., bigger ones, were used.

Besides, the proposed method requires the translation to be done in three steps, one for each of the coordinate axes. Thus, it would be possible to speed-up the algorithm if the axial parallelism constraint was removed (as in [8]).

Finally, we have not compared our rotation algorithm (see Addendum, below) with existent ones. Such a study would also be of interest.

c/i	+1	+2	+3	+4	+5
Sphere	3543/1450	1443/1772	3543/1996	1240/2297	3543/2500
Cactus	27734/10297	10304/12341	27734/14462	8428/14462	27734/18375

(a) Movements along X axis

c/i	+1	+2	+3	+4	+5
Sphere	3459/1429	1471/1772	3459/1961	1240/2297	3459/2437
Cactus	27118/9597	10066/11305	27454/12565	8358/12565	27398/15253

(b) Movements along Y axis

c/i	+1	+2	+3	+4	+5
Sphere	3739/1408	1471/1709	3739/1856	1240/2164	3599/2290
Cactus	27622/10087	10206/11907	27650/13664	8358/13664	28182/17199

(c) Movements along Z axis

Table 4: Number of black nodes before compression (c/i stands for classical/incremental)

A Addendum: Rotation by means of shear decomposition

In the area of image processing, decomposition of 2D rotations into three shears along the coordinate axes is a well-known issue (see [11]). Since arbitrary 3D rotations can be expressed as a product of three rotations around orthogonal axes, it is possible to perform a 3D rotation with 9 shears. Recently, [12] has proved that it is also possible to decompose 3D rotations into just three shears. However, we will use the longer decomposition, because the shears in [12] differ from the identity matrix in two terms, while the ones in [11], due to its two dimensional origin, differ from the identity in just one term. They are simpler to use because only one coordinate is affected at each step.

Without loss of generality, consider a θ-rotation around the X axis, expressed in matrix form as

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\ \theta & -sin\ \theta \\ 0 & sin\ \theta & cos\ \theta \end{pmatrix}$$

This matrix, as [11] proves, can be decomposed into a product of three shears of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -tan\ \frac{\theta}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & sin\ \theta & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -tan\ \frac{\theta}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

The shear algorithm is quite straightforward. Just traverse the octree in a front to back fashion, and, for each black node encountered, calculate its translation vector and move the node using the translation algorithm. In order to diminish the number of floating point operations, it is possible to calculate translations of grey nodes also, so parent node calculations can be reused for their sons. It is also feasible to reduce floating point operations to a minimum. As the shear divides the octree in slices of nodes sharing the same translation vector, we can further reduce the number of floating point computations by traversing the octree on a slice by slice basis. Moreover, we can completely eliminate floating point arithmetics if we use an algorithm *à la* Bresenham to calculate translation vectors from one slice to the next using integer additions only.

References

[1] Homer H. Chen and Thomas S. Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, 43(3):409–431, September 1988.

[2] Chris L. Jackins and Steven L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, November 1980.

[3] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[4] I. Gargantini. Translation, rotation, and superposition of linear quadtrees. *International Journal of Man-Machine Studies*, 18(3):253–263, March 1983.

[5] Narendra Ahuja and Charles Nash. Octree representations of moving objects. *Computer Vision, Graphics, and Image Processing*, 26(2):207–216, May 1984.

[6] W.M. Osse and N. Ahuja. Efficient octree representations of moving objects. In *Seventh International Conference on Pattern Recognition (Montreal, Canada, July 30-August 2, 1984)*, pages 821–823, 1984.

[7] K. Yamaguchi, T.L. Kunii, K. Fujimura, and H. Toriya. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, January 1984.

[8] T.R. Walsh. Efficient axis-translation of binary digital pictures by blocks in linear quadtree representation. *Computer Vision, Graphics, and Image Processing*, 41(3):282–292, March 1988.

[9] H. Samet and C. A. Shaffer. A model for the analysis of neighbor finding in pointer-based quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(6):717–720, 1985.

[10] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[11] Alan W. Paeth. A fast algorithm for general faster rotation. In Andrew S. Glassner, editor, *Graphics Gems*. Academic Press, New York, 1990.

[12] Tommaso Toffoli and Jason Quick. Three-dimensional rotations by three shears. *Graphical Models and Image Processing*, 59(2):89–95, March 1997.