Time-Efficient Dynamic Scene Management Using Octrees

Anand Gupta
Division of COE
Netaji Subhas Institute of Technology
New Delhi, India
anand@coe.nsit.ac.in

S. Vaishnavi, Saurav Malviya
Division of ICE
Netaji Subhas Institute of Technology
New Delhi, India
vaishnavi.89@gmail.com, malviya.saurav@gmail.com

Abstract - **In this paper, we present a method of management of a dynamic scene using octrees. The use of octrees in image rendering in 3D space is suitable as the octree is essentially a tree data structure in three dimensions. Most such methods resort to modification - namely, resizing and rebuilding - of the nodes of the tree used in order to accomplish the desired results. The main concern in such an approach is to minimize, or preferably, avoid resizing of nodes during runtime, as it takes a great toll on system resources. Here we present an algorithm that completely avoids resizing of nodes, hence achieving greater efficiency. This aspect of the algorithm is also borne out by the experimental conclusions we have obtained.**

Keywords- Image rendering; octree; time-efficiency

I. INTRODUCTION

Management of a dynamic scene requires keeping track of the positions of an object both before and after changes in the scene. The type of data structure used for storage of the image data has a major influence on the efficiency of the algorithm employed. Usage of a tree data structure makes the handling of the scene data much easier, as shown in [11]. Tree data structures include binary trees, B-trees, kd-trees, quadtrees and octrees. In a case such as ours, octrees are preferable to the other kinds of trees. The octree is the tree data structure that most closely mimics the 3D space which we are working in, hence making the quantization of the image data into the tree nodes much easier, as was examined in [1] and [4]. Also, since the octree is a 3D structure, there exists a one-to-one correspondence between a point in space and a node of the octree, hence eliminating all need for establishing a mechanism for uniquely identifying a node corresponding to a particular point in space. All in all, our requirement for an efficient and effective data structure for use in a 3D space environment leads us to choose the octree as the best possible option.

Related Work: Octree implementation in the field of dynamic scene management is a relatively new development, the earliest work in which dates back to the early 1980s. The major papers include:

1. Quadtree and Related Hierarchical Data Structures – H Samet, 1984 [2].

2. The Design and Analysis of Spatial Data Structures - H Samet, 1989 [6].

978-1-4244-2824-3/08/$25.00 ©2008 IEEE

In papers [2] and [6], Samet outlines various tree data structures in 2D and 3D space, and also explains various operations that can be conducted on the data.

3. An Algorithm for Perspective Viewing of Objects Represented by Octrees - Walid G. Aref, H Samet, 1995 [8].

The paper [8] provides an algorithm for handling objects, the information of which has been quantized into the nodes of an octree.

4. Efficient Neighbor Finding Algorithms in Quadtree and Octree - P Bhattacharya, 2001 [9].

Paper [9] explains various neighbor finding algorithms for both 2D and 3D tree data structures. We have drawn extensively from the research done on octree neighbor-finding in this paper.

5. Dynamic Irregular Octrees - Joshua Shagam, Joseph Pfeiffer Jr., 2003 [10].

The paper [10] talks about irregular octrees, a subject which has not been touched upon here, but which was, nonetheless, helpful as far as understanding the dynamic aspect of octrees was concerned

Our Work: The earlier papers which have used octrees to effect dynamic scene management have made use of the concept of resizing of nodes at runtime. Cases in point would be [12] and [13]. This takes a great toll on the processing efficiency of the program. Our method avoids this element of programming by making the leaf node large enough to house the largest object, hence leading to better values of time and space complexity for the algorithm. At the time of initialization of the tree structure, a check is placed to stop the creation of further child nodes once the child node is not large enough to house the entire object. Such an approach eliminates the need for resizing and rebuilding of the octree at runtime. We input the endpoints of a ray. Starting from one endpoint, we traverse the octree through our neighbor finding algorithm till the other is reached, as outlined in [5], [3] and [14]. This helps us identify the nodes falling in the path of the ray, thereby allowing us to manage the movement of the object. This process is illustrated by the flowchart in Fig. 1. The organization of the paper is as follows: section 2 introduces the method of encoding the octree; section 3 discusses the dynamic management of the scene using octrees; section 4 presents the

efficiency of our algorithm and builds a comparison with regard to the existing method, which is given in [13].

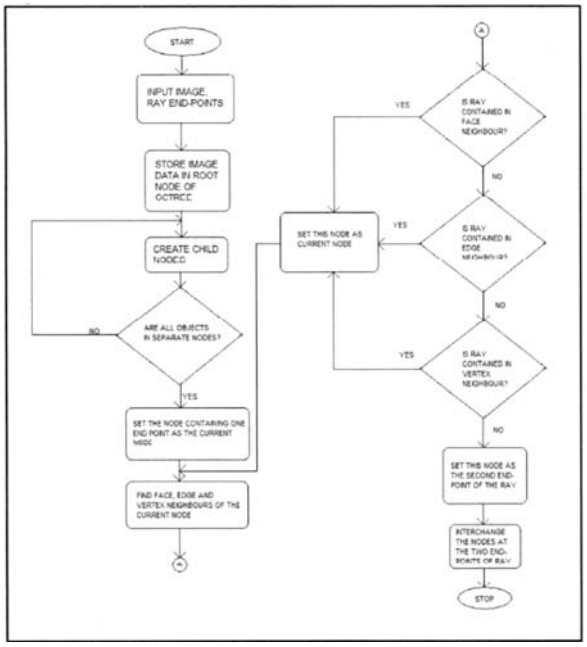

Figure 1. Flowchart illustrating the algorithm.

II. METHOD OF ENCODING

3D space can be divided into 8 octants. Each octant is given an indexing number which can be used to identify it uniquely within that space. Each octant is further divided into 8 octants. This process is continued recursively till an octant has optimum space for the containment of one complete object. The six possible directions are assigned six unique identifier strings. Each octant has twelve edges and eight vertices. The edges are assigned the codes X00, X01, X10, X11, 0X0, 0X1, 1X0, 1X1, 00X, 01X, 10X and 11X. This type of encoding proves helpful in the neighbor finding part of the experiment.

The six directions are namely, left, right, front, back, top and bottom, and their respective codes are: XX0, XX1, 0XX, 1XX, X0X and X1X. The eight child nodes of an octant are assigned codes corresponding to their position and the code of their parent node. The coding is done as follows:

FRONT FACE:-
Top-left: 0
Top-right: 1
Bottom-left: 2
Bottom-right: 3

BACK FACE:-
Top-left: 4
Top-right: 5
Bottom-left: 6
Bottom-right: 7

The code thus formulated is appended to the code of the parent node. Ergo, if the code of a node in the octree is $a_0a_1a_2.....a_n$, then the code of its child node in, say, the front bottom-right direction is $a_0a_1a_2.....a_n3$. It can be seen that the number of digits in the code helps to identify the level of the particular node in the tree structure. The root node does not have any code and its child nodes have single-digit codes.

III. DYNAMIC SCENE MANAGEMENT USING OCTREES

The nodes containing the end points of the ray are identified. Neighbor finding is initiated from one of the end points, proceeding towards the other, as given in [7].

A. Neighbor Finding

In an octree, which is essentially a 3D tree data structure, a node of the octree can have three neighbors - face neighbor, edge neighbor and vertex neighbor. Each 3D octant has 6 faces, and therefore has at most six neighboring octants which share a face with it.

1) Face Neighbor Finding: In an octant, there exist eight nodes. Out of the six possible face neighbors of a node, three of them exist in the same octant as the node. The remaining three are present in each of the three face neighbors of the parent node.

Say we select node A in Fig. 2. It is evident that three of its face neighbors are nodes B, E and C in the same octant. The face neighbor in the left direction (XX0) is the node B of the octant to the left of the one under scrutiny. In the top direction (X0X), the face neighbor is present in the octant to the top of this one and is the node C of that octant. And finally, in the front direction (0XX), the node E in the octant to the front of this one is the sixth face neighbor.

If we consider the directions left and right - XX0 and XX1 respectively - it can be seen that the last digits of the codes of A's face neighbors in both these directions are the same (the neighbor to the right is node B of the same octant, while the neighbor to the left is also node B, but of a different octant). Similarly, the last digits of the codes of the face neighbors in the front and back and the top and bottom pairs of directions are the same as well. Hence, it can be shown that the last digits of the face neighbors of any node in any two directions 180 degrees apart remain the same.

Therefore, on specifying a certain direction, the program looks for a face neighbor of the node in that direction first in its octant itself. In case the face neighbor is present, the code of that node along with the preceding code-sequence of the parent node is returned. In the event of the face neighbor not being a part of the same octant, the function looks for the face neighbor of the parent node in the given direction. Then it finds the child node having the same last digit as that of the face neighbor in the opposite direction (which resides in the same octant).

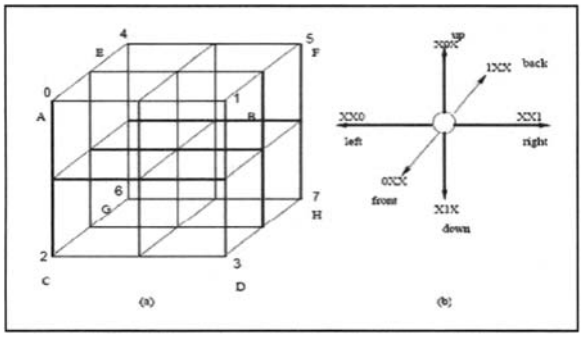

Figure 2. Face neighbor finding of a node.

2) Edge Neighbor Finding: Each node of an octant has twelve possible edge neighbors, three of which exist in the same octant as the node.

For example, we select node A in Fig. 3. The three edge neighbors in the same octant are nodes D, F and G. These nodes correspond to the edges X11, 1X1 and 11X respectively. The edge neighbors along the other nine edges exist in three other octants which are among the face neighbors and edge neighbors of the parent node. The fact that some of the edge neighbors of a node exist in the face neighbors of its parent node enables a more efficient edge neighbor finding algorithm.

The edge neighbors of a given node have the same last digit when they lie at the ends of the diagonal of the face perpendicular to the edge in question. A function is created which takes this fact into account and returns the last digit of the edge neighbor of a node. Suppose, the edge neighbors of node A are to be calculated. The three neighbors in the parent node are F, G and D. The eight vertices of the node A are thus coded as 00, 01, 02, 03, 04, 05, 06 and 07 preceded by the code of A's parent node.

Now, consider an edge neighbor of the node A which lies outside of its parent node. Say, along the edge 0X1, the edge neighbor of A has its last digit as 5 and lies in the node which is the front directional face neighbor of the parent node of A. Then again, along the edge 0X0, the edge neighbor of A has 5 as the last digit and is present in a node which is, again, the edge neighbor along the edge 0X0 of the parent node of A. Hence, we can divide the edge neighbor finding into three distinct cases - one, where the edge neighbor of a node is present in the parent node itself, two, where the edge neighbor exists in one of the face neighbors of the parent node, and three, when the edge neighbor of a node is seen to be in one of the edge neighbors of the parent node.

We distinguish these three cases by assigning certain parameters to be passed back to the edge neighbor finding function. In the first case, the string "P" is passed back to the function.

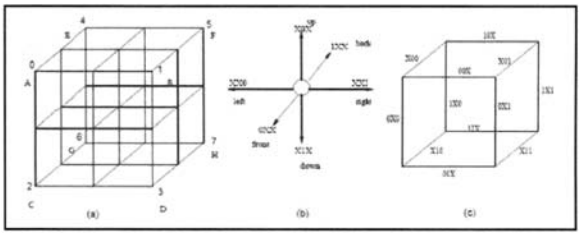

Figure 3. Edge neighbor finding of a node.

When the edge neighbor is present in an edge neighbor of the parent node, the string "EN" is passed back. And lastly, by elimination, the function assumes the edge neighbor to exist in a face neighbor of the parent node if neither of these two strings is passed back (In that case, the direction in which the face neighbor is to be located is returned in string form).

On accepting the parameter "P", the function, after calling the function that returns the last digit of the edge neighbor, appends the last digit to the code of the parent node itself. In case the passed parameter is the string "EN", the function recursively calls itself to check for the edge neighbor of the parent node and eventually appends the last digit to the appropriate code. And in the third case, the function locates the face neighbor of the parent node in the required direction, and suffixes its code with the last digit of the edge neighbor which was previously calculated.

3) Vertex Neighbor Finding: Of the eight nodes of an octant, each has eight possible vertex neighbors. Out of these eight, one exists in the same octant as the node, one in the face neighbor of the parent, one in the vertex neighbor of the parent and the rest in the edge neighbors of the parent node. The last bit of the vertex neighbor present in the parent node is given by subtracting the last bit of the considered node from 7, while the last bit of the vertex neighbor in the vertex neighbor of the parent node is also given by the same value.

For example, we select node A (last bit 0) in Fig. 4, the vertex neighbor in the parent node is H, having the last bit 7. Similarly, the last bit of the vertex neighbor in the vertex neighbor of the parent node sharing vertex 00 with it, is 7. In case of the vertex neighbor residing in the edge neighbor or the face neighbor of the parent node, its last bit will be the same as that of the vertex neighbor in the parent octant of the considered node. The edge neighbor and face neighbor of the parent node are calculated using the above mentioned algorithms.

B. Scene Management

The input to the program is comprised of the scene image and the end-points of the ray. The neighbor finding algorithm detailed above is used to recognize the nodes through which the ray passes. Once the nodes containing the ray are identified, the data in the nodes containing the end points of the ray is processed, corresponding to the change required in the scene.

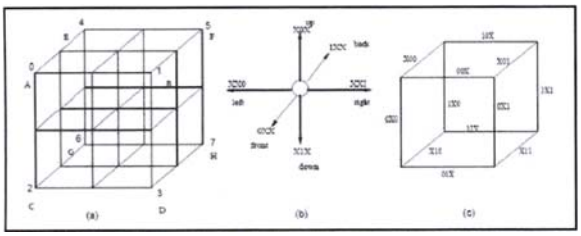

Figure 4. Vertex neighbor finding of a node.

IV. EXPERIMENTAL RESULTS AND CONCLUSIONS

The efficiency of an algorithm is measured on various different parameters. The two most important ones are time complexity and space complexity. These two together are sufficient to describe most computer models to a reasonable degree.

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm. To understand this intuitively, consider the example of an instance that is n bits long that can be solved in n^2 steps. In this example we say the problem has a time complexity of n^2. Of course, the exact number of steps will depend on exactly what machine or language is being used. To avoid that problem, the Big O notation is generally used (sometimes described as the "order" of the calculation, as in "of the order of"). If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers, so this notation allows us to generalize away from the details of a particular computer. For example, mowing grass has linear time complexity because it takes double the time to mow double the area. However, looking up something in a dictionary has only logarithmic time complexity because a double sized dictionary only has to be opened one time more (i.e. exactly in the middle, then the problem size is reduced by half). The space complexity of a problem is a related concept that measures the amount of space, or memory, required by the algorithm. An informal analogy would be the amount of scratch paper needed while working out a problem with pen and paper. Space complexity is also measured with Big O notation.

The process of creation and maintenance of a regular octree has the time complexity of $O(n^2)$. In other algorithms, if the terminal nodes of a ray are not at the same level, resizing is done in order to exchange the data of the two nodes. The worst case scenario is one in which one node is of the level n, and the other is of the level 1. As resizing is done by descending one level in the octree at a time, the time complexity will become $O(n^2+n-1)$. In the method proposed here, the time complexity even in the worst case scenario will remain $O(n^2)$, thereby reducing the time required to effect the changes in the scene. This is illustrated in Fig. 5, where the blue graph (the top curve) represents the time complexity of the algorithm given in paper [13], while the green curve (the lower one) is the time complexity of our algorithm.

It was seen that for a case having a lower number of levels n less than 10, the difference in the time complexities of the algorithms was appreciable. The bar chart for the comparison is shown in Fig. 6. However, for cases where the number of levels of the created octree increased to beyond 40, n^2 became increasingly greater in comparison to n. Hence, the value of n^2+n-1 tends to n^2, and any difference in the two time complexities is negligible, as can be seen by Fig. 7. Therefore, the gain percentage of our algorithm is maximum at fewer levels and decreases for a higher number of levels. This is explained using the graph in Fig. 8.

The space complexity of our method is slightly better than that of other methods because of the absence of the resizing process during runtime. But the size of the octree itself is so large that this improvement upon the usage of memory is negligible.

V. PROSPECTS FOR FUTURE WORK

In the future, this method needs to be tested for scenes involving numerous discrete objects. As far as real-world applications are concerned, it can be used for any situation involving changes. Possible applications for which our method can be tried out are face isolation and face following.

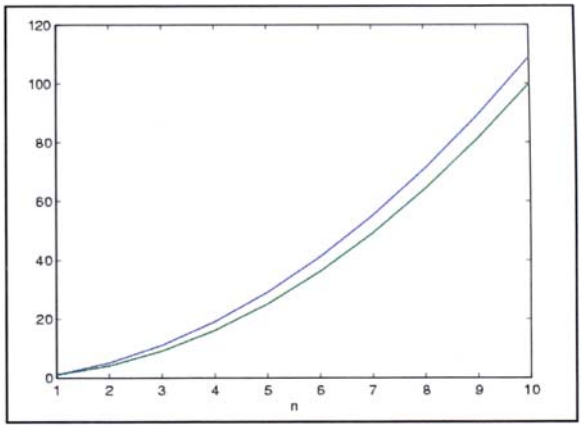

Figure 5. Graph of the time complexity of our algorithm with respect to the time complexity of algorithm given in paper [13].

TABLE I. Comparison Table

Our Method	Method Given in Paper [13]
The resizing of nodes is done only once, that is, at the time of initialization.	Resizing is done at least once at runtime.
Greater efficiency for most situations.	Lesser efficiency for most situations.
Better time and space complexities.	Average time and space complexities.
Superior utilization of system resources.	Greater toll taken on system resources.

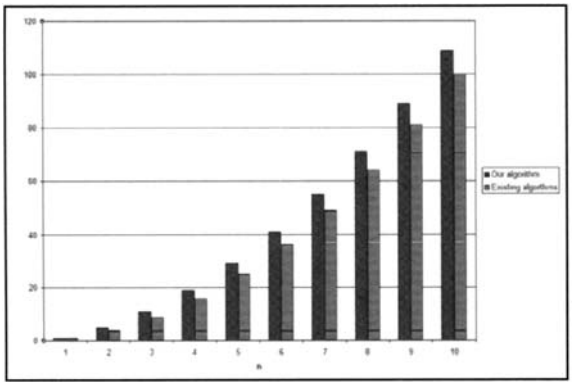

Figure 6. Bar chart of the relative complexities of our algorithm and algorithm given in [13] for the number of levels going from 1 to 10.

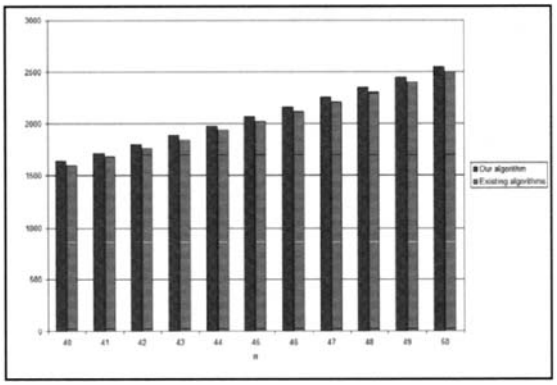

Figure 7. Bar chart of the relative complexities of our algorithm and algorithm given in [13] for the number of levels going from 40 to 50.

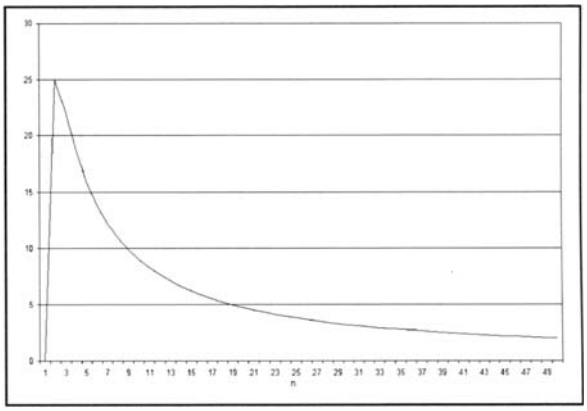

Figure 8. Graph of the percentage gain of the time complexity of our algorithm over that of the algorithm given in [13].

REFERENCES

[1] H. Samet, "Neighbor finding techniques for image represented by quadtrees," Computer Graphics and Image Processing 18, 1982, pp. 37-57.

[2] H. Samet, "Quadtree and related hierarchical structures," Computer Surveys 16(2), 1984, pp.187-260.

[3] A. S. Glassner, "Space subdivision for fast ray tracing," IEEE Computer Graphics and Applications Vol. 4(10), October 1984, pp. 15-22.

[4] H. Samet, "Implementing ray tracing with octrees and neighbour finding," Computer and Graphics Vol. 13(4), 1989, pp. 445-460.

[5] Andrew Glassner, "An introduction to ray tracing," Morgan Kauffman, 1989.

[6] H. Samet, "The design and analysis of spatial data structures," Computer Graphics, Image Processing and GIS, Addison-Wesley, 1990.

[7] I. Gargantini and H. H. Atkinson, "Ray tracing an octree: Numerical evaluation of the first intersection," Computer Graphics Forum Vol. 12(4), 1993, pp. 199-210.

[8] Walid G. Aref and H. Samet, "An algorithm for perspective viewing of objects represented by octrees," Computer Graphics Forum 14(1), 1995, pp. 5966.

[9] Parthajit Bhattacharya, "Efficient neighbour finding algorithms in quadtree and octree," November 2001.

[10] Joshua Shagam and Joseph Pfeiffer Jr., "Dynamic irregular octrees," New Mexico State University, Technical Reports, 2003.

[11] Herve Bronnimann and Marc Glisse, "Cost-optimal trees for ray shooting," Proceedings of the Latin American Symposium on Theoretical Informatics, 2004.

[12] Aaron Knoll, Ingo Wald, Steven G Parker and Charles D Hansen, "Interactive isosurface ray tracing of large octree volumes," Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 115-124.

[13] Huaiqing He, Yu Zhang and Haohan Liu, "A method used to dynamic scene management based on octree," Civil Aviation University of China, Computer Graphics, Imaging and Visualization, 2007. CGIV '07, 14-17 Aug 2007, pp. 16-21.

[14] Alexander Reshetov, Alexei Soupikov and Jim Hurley, "Multi-level ray tracing algorithm," ACM Transaction of Graphics 24(3), 2005, pp. 1176-1185.