SPATIAL TREES: A FAST ACCESS METHOD FOR UNSTRUCTURED GRAPHICAL DATA

Dan R. Olsen Jr.
Computer Science Department
Brigham Young University

Carol N. Cooper
Intel Corporation

Abstract

A spatial tree is a data structure for organizing graphical elements into a tree using their extents so as to optimized geometric queries on the database. Algorithms are presented for insertion and deletion. Queries on 2D spatial trees are modeled as simple rectangles for windowing purposes. Query algorithms for rays and convex polyhedra are presented for 3D spatial trees. It is shown how a least-recently-used swapping algorithm is highly effective when portions of the tree must reside in secondary storage. Simulation statistics as well as worst case analysis are provided for each algorithm.

1.0 Introduction

As larger and larger graphical databases are created, the costs of performing graphical queries on such databases significantly increase. Many such databases are organized in a hierarchic fashion. At each level of the hierarchy an extent can be computed which can be used to reject entire subtrees from consideration in a query. This technique is well known [Newm 79] and very effective in optimizing the display process.

There are, however, a number of applications which have no such natural hierarchic decomposition. An example of such an application is a topographic map. There are masses of small details for which there is little natural structure along which the map can be decomposed into sub-parts. A second example of this problem is a municipal planning system. These are typically quite large but one is usually looking at a small part of it.

Such a system can possibly be decomposed into sewage, streets, water, electrical etc. The problem, however, lies in the fact that all of these subsets have generally the same extents (they all cover the entire city). The natural decomposition then is of no use in extent testing to optimize our queries. What is needed is a data structure which will automatically impose a spatial decomposition on a database rather than relying on an inherent decomposition from the application.

A second problem, which we would like our data structure to address, is that large graphical worlds are frequently too large to fit entirely in fast. local storage. This occurs both in processors with limited main memory which must swap to and from disk and in distributed workstations which must swap to and from a host or file server. Any data structure for large graphical worlds must address this problem and optimize swapping where possible.

In dealing with the swapping problem we have made an assumption about the usage of a graphical database. We call this assumption locality of reference. This is the assumption that any access of such a database will be close to the previous access. This assumption is valid if one considers such interactive tasks as panning and zooming. Each successive window is close to or overlapping the previous one. When one performs a picking query from a locator input it will always be found inside of the last window query. In a ray tracing algorithm, successive rays are adjacent to each other. There are applications where the locality of reference assumption might not hold but for a large number of interactive applications it does.

If we are using a data structure in which graphical elements are stored according to their extents and the locality of reference assumption holds then a least-recently-used swapping algorithm should significantly reduce the swapping overhead.

The remainder of the paper will proceed by describing two dimensional spatial trees followed by the algorithms to manipulate them. We will also discuss the theoretical performance of each of these algorithms followed by the results of simulating their use.

2.0 2D Spatial Trees

A spatial tree is composed of a set of nodes. each of which corresponds to a rectangular area in world coordinates. The area, or extent, of the root of the tree is all of world coordinates. The extent of each node is divided into four quadrants by dividing each axis in half. Each quadrant forms the extent for four subnodes, as shown in Figure 1.

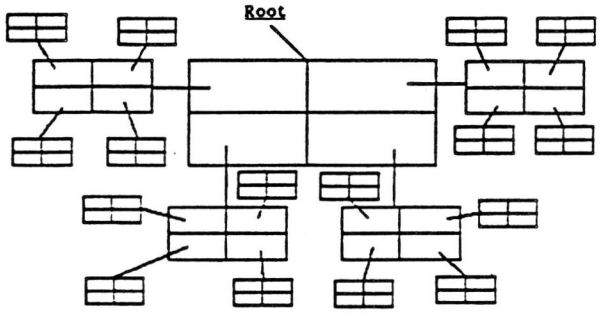

Figure 1.

This is very similar to a quad tree [Fink 74]. The difference lies in the fact that the graphical elements stored in a spatial tree are not pixels but rather any whole graphical element. The definition of a spatial tree is independent of any particular set of graphical elements. The only information known about an element to be inserted into the tree is its extent. This makes subdivision of the elements down to the pixel level impossible. Spatial trees are also related to B-trees [Come 72] and more specifically to digital B-trees [Lome 81], which subdivide their key space in a binary fashion. The difference between a digital B-tree and a spatial tree is that the keys in a spatial tree are not points in the key space but rather ranges, areas or volumes.

2.1 Element Insertion

For an element to be placed in a particular node of the tree the element's extent must fit entirely within the extent of the node. Elements are then inserted into the root node (which by definition they will always fit into) and then recursively inserted into the appropriate subnode until it can go down the tree no farther. An element stops when its extent does not fit entirely within the extent of one of the subnodes. This occurs when then element lies across one of the dividing boundaries. Figure 2a shows elements in world coordinates with the dividing boundaries for the nodes of the trees. Figure 2b shows how these elements are placed in the tree

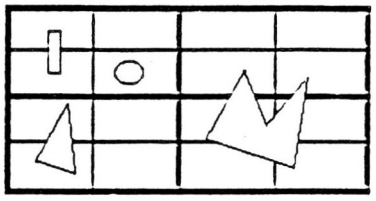

Figure 2a.

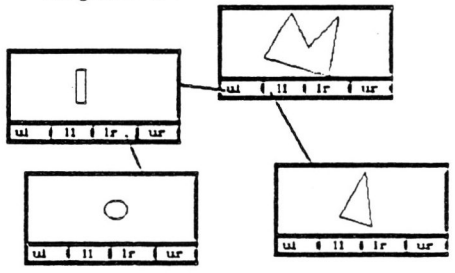

Figure 2b.

Each node of the tree is defined to hold a fixed number of elements which is determined by the implementation. When the number of elements that lie in a node's extent exceeds the number of nodes that can fit in the node, an overflow node is created which has the same extent as the original node. The only subnode which an overflow node can have is an additional overflow node.

The insertion algorithm can then be stated as follows

```
Insert( N:Node; E:Element );
  Begin
    If N is not full then
       place E in N
    Else
      Begin
       If subnodes of N are not present
              then
         Begin
           Take all elements currently in
             N and place them in their
             appropriate subnodes,
             creating subnodes as needed
             and retaining in N all
             elements which cannot fit in
             a subnode.
         End;
       SelectASubNode(N,E,SubNode);
       If SubNode = None Then
         place E in an overflow node
       Else
         Begin
           If SubNode does not exist then
             Create SubNode;
           Insert(SubNode,E);
         End;
      End;
  End;
```

Note that subnodes of N are only created when N is full. When N finally fills then the elements are tested to see if they will fit into subnodes. This prevents the creation of very deep trees with very few elements.

A key factor in the performance of insertion is the probability that a particular element will cross one of the node's dividing boundaries. If such an element crosses a boundary it will remain in the node and not proceed down further. If a lot of this behavior occurs then the tree becomes short and bushy and loses its value as a search structure. In computing these probabilities the ratio of the world coordinate size to the element size is the key consideration. Obviously the larger the element is relative to the size of the world, the more likely it is to hang in the higher nodes of the tree. Our studies show that given 10,000 elements whose size is 1/1000th of the world size which are uniformly distributed across the world, and given a tree which holds 5 elements per node, over 92% of the elements will be stored in the leaves of the tree. This is an excellent result in terms optimizing search times. When simulating uniform insertion using the actual implementation of spatial trees, similar results were achieved.

In order to test the swapping characteristics of the algorithm we saved the uniformly distributed insertions from the previous test. We then grouped these insertions by their centers into areas or buckets. We then sorted the buckets on X and then Y. This caused the inserted elements to be somewhat randomly distributed but to be clumped into neighborhoods and to be ordered into a sort of panning sequence. This was to simulate to some degree the behavior of someone working through the graphical world rather than just randomly placing elements here and there which is not the normal usage. This test is more consistent with our locality of reference assumption than a uniform distribution. In all of our tests the swapping was reduced using the sorted order by 17-68% depending on how full the tree actually was. The best swapping performance was achieved on very large trees with lots of little elements.

2.2 Element Deletion

As elements are deleted from the tree we obviously would like to collapse the tree structure and reduce the space consumed. The strategy is simply one of checking to see if the total number of elements in the current node plus those in the subnodes is less than 1/2 of the node size. If so then collapse the subnodes into the current node and apply the check recursively to the current node's parent. The reason for the 1/2 stipulation is to prevent collapsing a node and then immediately reinserting a new element into it causing its subnodes to be recreated.

2.3 2D Query

Only rectangular queries which are parallel to the axes have been explored using 2D spatial trees. There are two types. A query for all elements lying entirely within the query rectangle and those elements which in any way intersect the query rectangle. The first form of query is for interactive techniques which surround elements to be selected with a rectangle. The second form is for 2D viewing operations and for locator-based picking using a very small rectangle to simulate a gravity field. The query algorithm is rather simple and is shown

below for the case of intersecting elements.

```
Query ( N:Node, R:Rectangle );
  Begin
    For each element E in N or any
              overflow of N Do
       If E intersects R then
          Return E
       Else
          Ignore E;
       For each subnode SN of N Do
         If extent(SN) intersects R then
            Query(SN,R);
  End;
```

The above algorithm is stated recursively for expository reasons. In practice a stack algorithm is required so that the search can halt and return each element found and then be restarted to locate additional elements which match the query.

When analyzing the performance of the query algorithm we are most interested in the elements which fall into the "Ignore E" statement in the algorithm. We are not concerned with elements which are returned because, by definition of the query, we wanted to look at them. We are not concerned with elements that are in nodes that are never checked because they involve no work. Those elements which we do check and then ignore constitute the overhead of the algorithm which we want to minimize as much as possible. The overhead of a linear search algorithm for example is all the elements in the database which are not matched by the query.

Any overhead element will be stored in a node whose extent crosses one of the rectangle's boundaries. Any other node will either be completely inside or completely outside of the query. A simplistic analysis can be performed under the assumption that the spatial tree is complete down to some level N. If the extents of the nodes at level N are 1 by 1 and the dimensions of the rectangle are Rx by Ry then the total number of nodes that can be crossed by the rectangle at level N is 2*(Rx+Ry-2) (the length of the rectangle's perimeter). At level N-1 the number crossed is 1/4 that at level N because there are 1/4 as many nodes at that level. Using the same tree parameters discussed for insertion and a query window that is 1/64th the size of the world, approximately 2% of the elements looked at will be ignored. Our simulations show that the overhead is closer to 9.2%. The reason for this is that the tree produced by the random insertions is not actually complete. This means that there are actually many fewer nodes in the tree than the complete tree upon which the probabilistic analysis was based and thus elements are higher in the tree and more likely to be tested by the query.

3.0 3D Spatial Trees

The concept of spatial trees is very easily extended to three dimensions. The extents are now three dimensional and each node has three dividing planes rather than two dividing lines. Because each node divides into octants rather than quadrants the tree tends to be shorter and bushier. Because there is an additional boundary which an element may cross the probability that an element may "hang" in a higher node increases slightly. Both our simulations and our analysis show that this is not a serious problem. The extension of the insertion and deletion algorithms is trivial. The queries however must change because a rectangular window is not useful concept in three dimensions.

We have implemented two forms of query. The first is a ray query which returns all elements whose extent is pierced by a given ray. The ray query is obviously useful for ray tracing algorithms and is also used for 3D picking techniques. The second form is a plane query which returns all elements which are "inside" of all of a set of planes. The notion of "inside" is defined by where the positive normal vector of a plane points. By retrieving elements which are inside of all the given planes we can query on any convex polyhedron including the usual perspective viewing pyramid.

3.1 Ray Queries

Performing a ray query is a matter of determining for each node, which of the subnode extents the ray pierces and then recursively traversing them. We do this by determining which octant the ray starts in and which dividing planes it intersects. We then sort the intersection points along the axis of greatest excursion for the ray. That is the axis which has

the largest component in the ray's direction vector. The traversal of subnodes then begins with the original octant and then crosses boundaries into new octants in the sorted order.

This process not only simply determines which octants to traverse but it also supplies a traversal order which proceeds from the base of the ray to its head. In most ray tracing algorithms we want to find the intersecting object which is closest to the origin of the ray. This traversal order will encounter the closest one first. Upon finding an intersecting element we can terminate the traversal early. It is important to note that this is only an ordering on the nodes not on the elements in them. We get close to the optimum order but not all the way there. One should also note that the query algorithm itself cannot determine object intersection. It can only determine extent intersection which is not quite the same thing.

Note that the algorithm only enters a new node by crossing a bounding plane. Since a given node only has three dividing planes, the ray can only intersect the original octant plus three others. This means that in the worst case only half of the octants will actually be traversed. If we consider a tree which is complete down to level N then we have a cube consisting of $2^N x 2^N x 2^N$ (or 2^{3N}) nodes. In the worst case the ray will be on one of the diagonals of this cube and will cross all of the bounding planes. This will mean that $2^N+2^N+2^N+1$ (or $3*2^N+1$) nodes will be intersected. This means that at level 5 in the tree only 0.3% of the nodes will be traversed in the worst case. This is a substantial savings. Similar savings have already been demonstrated by [Glas 84] using a similar data structure.

In our simulations of ray queries using uniformly distributed rays we found that a single query accessed only 8.7% of the nodes in the tree and actually checked only 14.6% of the elements in the tree. Out of those elements checked against the ray, the overhead (or those elements not actually pierced by the ray) was 69%.

3.2 Plane Queries

The plane query algorithm is simply a process of determining which subnodes of a node might possibly lie inside of all of the query planes which is simply an intersection of those lying inside of each individual plane. The simplistic approach is to this algorithm is to pass the corner points of each suboctant through the plane equation to determine if it returns a positive or negative value. This value is, of course, proportional to the distance of the point from the plane. Points with positive proportional distances are inside and negative is outside. The relationship between these points and the plane is shown in Figure 3.

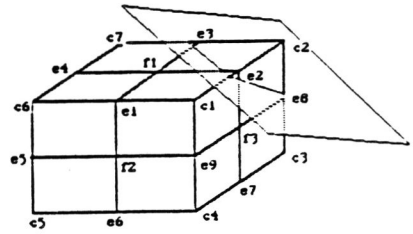

Figure 3.

The cost of multiplying points through plane equations would make the overhead of the spatial trees too high to be effective. Since the dividing planes actually split the extent in half each of the edge points can have their proportional distances computed as the average of two corner points, the face points are the averages of two edge points and the center point is the average of two face points. Averages can be computed with an add a division by 2 (shift right 1). Once the proportional distances for the corners of world coordinates were computed from the plane equation the rest of the computations proceed much more cheaply.

We further optimized the testing of all of the points against the plane by noting two geometric principles about the non-corner points. All of the non-corner points lie at the center of one or more line segments. If both endpoints of a line segment is known to be inside of a given plane then any interior point on that segment is also inside. The same statement can be made about the endpoints being outside of the plane. This case is shown in Figure 4a. A second case occurs when one endpoint is on the opposite side of the plane from the center point. In this case the other (unknown) endpoint must

be on the same side of the plane as the center point. This case is shown in Figure 4b.

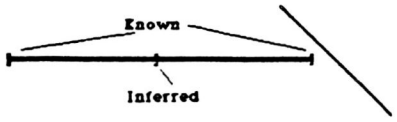

Figure 4a

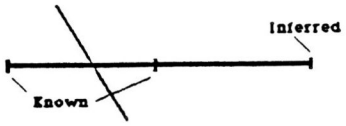

Figure 4b

These principles can be applied to the corner points of suboctants to determine who is inside and outside without even looking at many of the points. Given the plane example in Figure 3 and having already checked c1. c2. c3. c4. c5 and c6 to determine if they are inside or outside we can infer the status of e1, e4, e5, e6, e7 and e9 as well as f1, f2 and f3. As is shown in Figure 5.

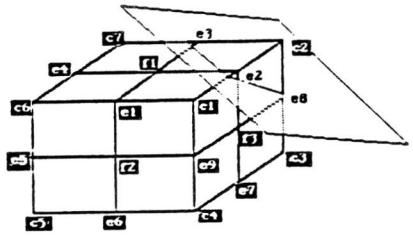

Figure 5

Only e2, e3 and e8 still remain to be checked. In this example only 11 points were checked instead of a possible 27.

These inferences cannot be done at runtime and instead are performed once and coded into a large nested IF. Since the total number of cases required is larger than can be accurately coded by hand we wrote a program to generate the case testing by automatically generating the inferences from points already tested for inside and outside. The generated code for case testing was several hundred lines of nested IF statements.

Our simulations used four planes arranged as a viewing pyramid with uniformly distributed viewing directions and angles. On the average only 32% of the nodes in the tree were traversed. Of those elements which were actually examined by the query only 17.6% were overhead elements.

4.0 Conclusion

The probabilistic analysis that we have performed on spatial trees as well as the results of our simulations lead us to believe that this is an effective data structure for large graphical databases. In addition to the efficiency of query operations on the data a least-recently-used swapping strategy on the nodes is very effective and shows minimal swapping requirements when the locality of reference assumption holds.

Spatial trees also have the advantage that they work on any set of graphical elements for which extents can be computed unlike other data structures which require pixels or polygons as their primitive basis. We have not yet researched the integration of hierarchic models with this structure and how best the two might be integrated in a single system.

References

Comer. D. "The Ubiquitous B-tree." Computing Surveys, Vol 11, 2 (June 1972).

Cooper, C. "Fast Retrieval of Graphical Information." M.S. Thesis, Arizona State University, (Dec. 1984).

Donelson, W. "Spatial Management of Information." Computer Graphics, Vol 12, 3 (Aug 1978).

Finkel, R. A. and Bentley, J. L. "Quad trees: A Structure for Retrieval on Composite Keys." Acta Informatica Vol 4.1-9 (1974).

Glassner. A. S. "Space Subdivision for Fast Ray Tracing." IEEE Computer Graphics and Applications, Vol 4, 10 (Oct. 1984).

Lomet. D. "Digital B-trees." IEEE Proceedings of the 7th International Conference on Very Large Data Bases (Sept. 1981).

Newman, W. and Sproull, R. Principles of Interactive Computer Graphics. McGraw-Hill 1979.