Space Subdivision Algorithms for Ray Tracing

by

David MacDonald

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1988

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature David MacDonald

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature David MacDonald

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Ray tracing provides computer rendering of synthetic images with many realistic qualities, but is computationally expensive. Ray tracing requires testing of rays against a scene to see which objects, if any, are intersected. The high number of such tests required by typical ray tracers contributes significantly to the computational expense of ray tracing.

An efficient method of reducing the computation involved in the intersection tests is to organize the objects composing the scene into one of several types of hierarchical data structures.

This thesis describes algorithms for the construction, storage, and traversal of the space subdivision hierarchy. Methods are suggested for decreasing computational requirements of the data structure with respect to these three areas.

One suggested strategy for improving performance in all three areas (construction, storage, and traversal) is implemented for the bintree structure. The performance of these simulations is compared with implementations of contemporary methods and some efficiency gains are shown.

Further work is suggested, including adaptation of some of the ideas presented within this thesis to more general types of hierarchical structures.

Acknowledgements

I would like to thank Kellogg Booth for supervising me during the creation of this thesis. I am very grateful for his suggestions and encouragements throughout my stay at the University of Waterloo.

I am indebted to my two faculty readers, Charles Colbourn and Derick Wood, and to my student reader, Stewart Kingdon. Their comments and criticisms added greatly to the quality of this thesis.

I sincerely appreciate the financial assistance provided thoughout my graduate studies by the Natural Sciences and Engineering Research Council.

This thesis is dedicated to my friends.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

One popular algorithm for computer rendering of synthetic images is ray tracing. The main reason that the use of ray tracing is so widespread is the simplicity of coding and comparative ease with which ray tracing renders many realistic effects, including shadows, penumbrae, reflection, refraction (transparency), and motion blur.

The principal drawback of ray tracing is its comparatively high computational cost, which is due primarily to the high occurrence of one basic operation, the ray-scene intersection test. This computation can be expressed as answering the query: given a scene comprising a set of objects and a ray defined by an origin point and a direction vector, does the ray intersect some object, and what is the first (closest) object intersected? Other information may also be required if an object is intersected, such as the point of intersection and the normal to the object's surface at the point of intersection.

The computing of ray-scene intersections has traditionally been speeded up by many methods. Some hybrid algorithms reduce the number of intersection computations required by combining ray tracing with other types of rendering, such as standard scan-conversion techniques. Others reduce the number of such computations by more efficient ray tracing, minimizing the number of rays traced. Still others use parallel processors to perform more than one intersection computation at once.

While the methods mentioned above reduce the cost of ray tracing by cutting down the number of intersection computations, none of them actually speed up the basic ray-scene intersection test, which is the issue addressed by this thesis. The simplest, brute force method of determining the ray-scene intersection is to test the ray against each object, remembering which object, if any, has the nearest point of intersection. This has been vastly improved with the use of scene structuring [Rubi80], [Wegh84], [Glas84], [Kapl85], [Kay86], [Fuji86], [Glas87c], [Gold87], [Glas88]. Scene structuring relies on the fact that the intersection test is basically a multi-dimensional search. It can be speeded up by use of a search tree, which imposes an ordering upon the objects comprising the scene. This ordering reduces the number of ray-object intersection tests required.

Before further discussion of search trees for ray tracing, an introduction to the ray tracing algorithm is warranted. The next section provides a very brief overview of ray tracing. A good in-depth introduction to ray tracing may be found in [Glas87a].

1.1. Introduction to Ray Tracing

The images a person sees are created by energy emanating from light sources, bouncing off objects in the environment, and hitting his or her eye. A computer generated image of a given scene can be obtained by placing an imaginary eyepoint and a viewing screen into the three dimensional scene of objects. The scene is mapped onto the viewing screen by following rays of light leaving the light sources, bouncing off objects, through the screen, and into the imaginary eye.

However, it is much too expensive to trace rays in this way from the light source following their paths as they bounce off and travel through objects, because the small probability of a given ray eventually hitting the eye requires such a large number of rays to be traced that infinite resources would be required. Therefore standard ray tracing algorithms trace rays backward from the eye, through the screen, to the scene. The colour of each pixel of the image is determined independently from other pixels, by tracing a ray from the eye through the corresponding virtual pixel on the viewing screen, and computing what object, if any, it hits. Basic models of surface physics are used to determine the colour and intensity of light travelling from the intersection point to the eye, and thereby colouring the pixel in question.

The computation of light colour and intensity typically involves the surface normal at the intersection point, the eye vector (direction to the eye from the point), the light vectors (the directions to the light sources from the point), and the intensity of light hitting the point from the light sources, this last term accounting for shadows. In most ray tracers, the intensity of light hitting a point from any particular light source is set to zero if the point is shadowed by an opaque object, and set to the intensity of the light source otherwise. To check if an object is in shadow, a ray-scene intersection test is performed to determine if there is any opaque object intersected by the ray from the point of intersection to the light source.

More complex ray tracers model other effects such as reflection and refraction by tracing additional rays which represent backward reflection and transmission of the original ray. These rays may recursively spawn other reflection and transmission rays in attempts to provide a more realistic model. Many antialiasing techniques require sampling of extra rays. The general rule in ray tracing is that increased realism and picture quality requires more ray-scene intersection tests, indicating an obvious motivation for efforts to reduce the cost of ray-scene intersection computations.

This thesis examines methods to reduce the cost of ray-scene intersection tests through the use of space subdivision hierarchies. Various algorithms to improve performance are advanced.

Chapter 2
Background

This chapter provides some background necessary for the discussion of space subdivision for ray tracing. It begins with the basic geometry and algebra of rays and planes, and then introduces the idea of a bounding volume. Next a description of the two major hierarchical data structures for ray tracing is given, with emphasis on space subdivision hierarchies. Finally, a survey of previous work on space subdivision is presented.

2.1. Rays and Planes

A **ray** is defined by a point origin and a direction vector, both of which consist of three coordinates. A ray consists of all points which are the sum of the origin and a non-negative multiple of the direction vector. A ray can be expressed in the form:

$$R(t) = (R_x, R_y, R_z) = O + t \cdot D,$$

where

$$0 \le t = \textit{the parametric distance along the ray}$$

$$O = (O_x, O_y, O_z) = \textit{Origin of ray R}$$

$$D = (D_x, D_y, D_z) = \textit{Direction vector of ray R}$$

A **plane** can be defined by a normal to the plane and a distance from the origin. An equation defining a plane is:

$$N \cdot P - d = 0,$$

where

$\cdot$ *represents dot product*

$N = (N_x, N_y, N_z) =$ *the unit normal vector to the plane*

$d =$ *distance of plane from origin*

$P = (P_x, P_y, P_z) =$ *any point on the plane*

The plane consists of all points P which satisfy this equation.

The intersection of a ray with a plane can be obtained by first determining the value of the parameter t of the ray where the ray intersects the plane, then substituting this into the equation of the ray, to find the intersection point. The value of t may be calculated with the following equation:

$$t = \frac{d - N \cdot O}{N \cdot D}$$

If $N \cdot D$ is zero, then the ray is parallel to the plane and either does not intersect the plane or is embedded in it. In the case where the plane in question is perpendicular to either the x, y, or z axis, the equation is simpler (no dot-product), as in the following case, where we assume the normal is (1,0,0):

$$t = \frac{d - O_x}{D_x},$$

where

$x = d$ *is the simple form of the equation of the plane*

Since we are dealing with these types of planes extensively, let us term a plane that is perpendicular to a major axis (x, y, or z) a **major plane**.

2.2. Bounding Volumes

Scenes are modelled with a variety of different implicitly and explicitly defined objects and surfaces. They range from simple objects, such as spheres, ellipses, triangles, polygons, and parallelpipeds, to more complex surfaces such as cubic patches, spline surfaces, and implicit functions. For all but the simplest of these, an intersection test of a ray with the object is a nontrivial computation.

To speed up the intersection test, a bounding volume is placed around the object. The bounding volume is typically a very simple type of object with an easy intersection test, such as a sphere or a parallelpiped which has sides perpendicular to a major axis (bounded by major planes). In order to determine if a ray intersects a particular object, the ray is first tested against the object's bounding volume. If the ray does not intersect the bounding volume, it is obvious that it does not intersect the object inside, and the intersection test is finished. If the ray does intersect the bounding volume, the ray must be tested against the object in the usual manner. In this way, the bounding volume is used as a test for trivial rejection of a ray-object intersection.

The type of object chosen as bounding volume must have a sufficiently simple ray intersection test, yet must also provide a tight fit around the objects to be enclosed. The bounding volume must be tight enough around the objects that enough rays are trivially rejected to provide savings at least as great as the increased cost of computing the ray-bounding volume intersection test.

A common type of object for bounding volumes is a rectangular parallelpiped (each side is perpendicular to a major axis). The specification of this type of bounding volume (hereafter referred to as a **box**) requires only six values, a low and high value for each of the three coordinates. The intersection test of a ray with a box is quite simple, essentially clipping the ray against the box. It involves finding the range of the parametric value *t* for which the ray is within the box. If the range is empty, the ray does not intersect the box. A non-empty range indicates intersection. A simple coding of this algorithm is given in Figure 2.1.

Computation of a box bounding volume is usually a simple operation also. The tightest fitting box for any object is defined by the maximum and minimum coordinate values for each of the three coordinates. For most explicitly defined objects and surfaces, including polygons, spheroids, and parallelpipeds, this represents a very straightforward computation.

2.3. Hierarchical Data Structures

Scene structuring is usually accomplished with a hierarchical data structure. There are two main classes of hierarchy applicable to ordering the scene, which are duals of each other. The **object hierarchy** subdivides the objects composing the scene, recording the space that each object inhabits. Space subdivision, or **volume hierarchy**, subdivides space, recording the objects that inhabit each region of space.

2.3.1. Object Hierarchies

An improvement over the exhaustive search of the ray with a set of objects is to compute a bounding volume enclosing all of the objects in the set. Then, any rays which do not intersect the bounding volume can be rejected, thereby introducing some savings. Additional savings can be obtained by dividing the objects into two groups, and for each group, computing a bounding volume enclosing all of the objects in the group. A ray is tested against each bounding volume and if it does not intersect, the enclosed objects need not be tested. The improvement here is a result of sometimes being able to quickly reject half the objects (or all of them, in some cases) from the intersection tests.

```
FUNCTION RayBoxIntersect( RayOrigin, RayDirection, Box ) : BOOLEAN;
CONST
    Dimensions = 3;

VAR
    tmin, tmax, tlow, thigh, tenter, texit : REAL;

BEGIN

(* set the range of t to [0..+∞] *)

    tmin := 0;          tmax := +∞;

    FOR coord := 1 TO Dimensions DO
        IF RayDirection[coord] <> 0.0 THEN

        (* calculate the range of t between the two parallel planes,
       store in tlow, thigh                                             *)

            t1 := (Box[1,coord] - RayOrigin[coord]) / RayDirection[coord];
            t2 := (Box[2,coord] - RayOrigin[coord]) / RayDirection[coord];

            tlow := MIN( t1, t2 );
            thigh  := MAX( t1, t2 );

        (* get intersection of range so far, and range [tlow..thigh] *)

            tmin := MAX( tmin, tlow );
            tmax := MIN( tmax, thigh );
        ENDIF;
    ENDFOR;

(* if intersection occurs forward of origin AND
  the range is nonempty, then there is an intersection *)

    IF tmin <= tmax THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    ENDIF;
END RayBoxIntersect;
```

Figure 2.1.

Given the speed gain in dividing a group of objects into two, it seems reasonable that even more gains would result from subdividing each of the two groups into smaller groups. A **hierarchical extents tree** (hereafter referred to as an **HE tree**) effectively provides a recursive subdivision of objects in this manner ("extent" is just another name for bounding volume). This type of hierarchy is often termed an object hierarchy. The root of the tree corresponds to a bounding volume containing all of the objects in the scene. The children of a node correspond to a set of bounding volumes that divide the objects contained in the node's bounding volume. When the number of objects in a node's bounding volume is one, the node is given a single child where the object is actually stored. Although reference is made to objects enclosed by, or contained within, a node's bounding volume, it should be observed that objects are actually only stored in the leaves. An example of a small scene and a simple HE tree is given in Figure 2.2.

Given a ray and an HE tree, the intersection algorithm is a simple recursive process. The ray is tested against the root node's bounding volume and, if it intersects the bounding volume, the children are recorded as candidates for testing. This test is recursively applied to all children of any node intersected, thereby traversing the part of the tree where possible intersections with an object might occur. When a leaf node is encountered, the ray is tested with the object stored in the leaf, and the nearest intersection of a ray with an object is recorded during the traversal. This algorithm can often be speeded up by ordering the nodes to be tested according to the nearness of intersection with the nodes' bounding volumes.

The efficiency of the tree depends highly on the choice of bounding volumes during subdivision. In order to obtain an efficient HE tree, the subdivision process should divide a group of objects such that the corresponding bounding volumes are as small as possible and do not overlap. An indication of the validity of the second constraint arises from looking at the extreme case

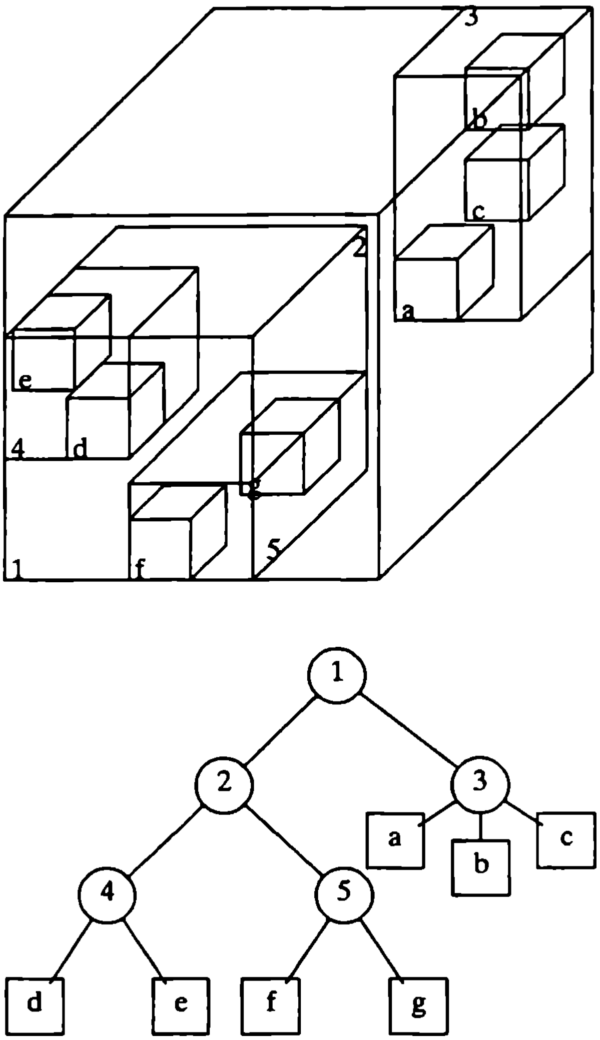


Figure 2.2
Letters a through g represent, the objects.
Numbers 1 through 5 represent interior nodes.

of overlap where all of the children of a node have the same bounding volume, which would have to be the same as that of the parent. In this case, it can be seen that the subdivision into children has effectively not subdivided the objects at all. If a ray intersects the parent node, it must intersect all of the children, giving no savings. The validity of the first constraint, using small bounding volumes, is more apparent. Smaller bounding volumes are less likely to be intersected by a ray, and therefore result in trivial rejection more often.

Based on these observations, any algorithm for constructing an HE tree should be devised with a goal of minimizing the size and overlap of bounding volumes. Because many systems define a scene in a hierarchical format, early HE tree construction techniques simply chose the bounding volumes defined by the groupings of the object hierarchy. This proved to result in quite inefficient trees, so other types of algorithms for generating efficient HE trees are now being explored. Examples of these algorithms may be found in [King86] and [Gold87].

2.3.2. Volume Hierarchies (Space Subdivision)

Object hierarchies recursively subdivide the objects into disjoint subsets, recording the space inhabited by these subsets. The dual of this is space subdivision, which subdivides the object space into disjoint subregions, recording the objects which inhabit these subsets of space. To see how this is done, let us examine the **octree**, a common type of space subdivision hierarchy, or volume hierarchy. Initially, the tree consists of only one node, representing the bounding volume containing all of the objects in the scene, exactly the same as the root of an HE tree. Using three **slicing planes** (one perpendicular to each of the three major axes), the bounding volume is divided into eight smaller ones. Eight children of the root are created (hence **octree**), one for each new bounding volume, and each object is placed in whichever child encloses it. This space subdivision can be seen in Figure 2.3.

Each of the children may be recursively subdivided. If a node contains no objects, then there is no advantage to subdividing it, and therefore, only nodes containing objects are subdivided. The bounding volumes associated with nodes are usually referred to as **voxels**, which is the three-dimensional analog of a pixel. Sometimes an object belongs in more than one voxel. In this case, one obvious technique is to split the object into new objects that do

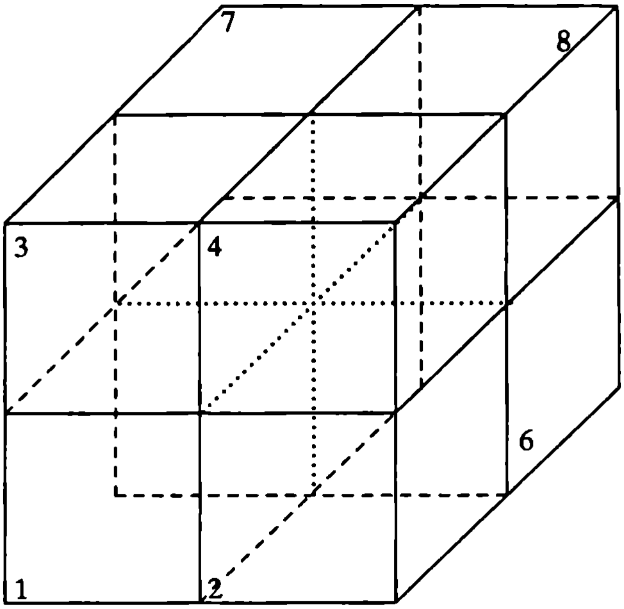


Figure 2.3

not belong in more than one node's voxel. However, this can be very costly in terms of time and space requirements, so usually the object, or a pointer to the object, is stored in both nodes [Glas84], [Kapl85], [Fuji86]. This technique introduces problems, which are discussed later.

As with the HE tree, when the octree subdivision process is complete, the resulting structure has all of the objects stored in the leaves, and none in the interior nodes. However, unlike the HE tree, a single leaf may contain more than one object. Figure 2.4 illustrates a simple scene and an octree representation of it.

The ray intersection algorithm for an octree may be implemented as a recursive process very similar to that of the HE tree. If a ray intersects the root node, it is tested against each of the children. This is recursively applied to the children of intersected nodes similar to the HE tree algorithm. When a leaf node is encountered, all of the objects stored in it are tested for intersection, and the nearest, if any, is recorded. The octree allows testing nodes in the order that the ray passes through them because it subdivides space into

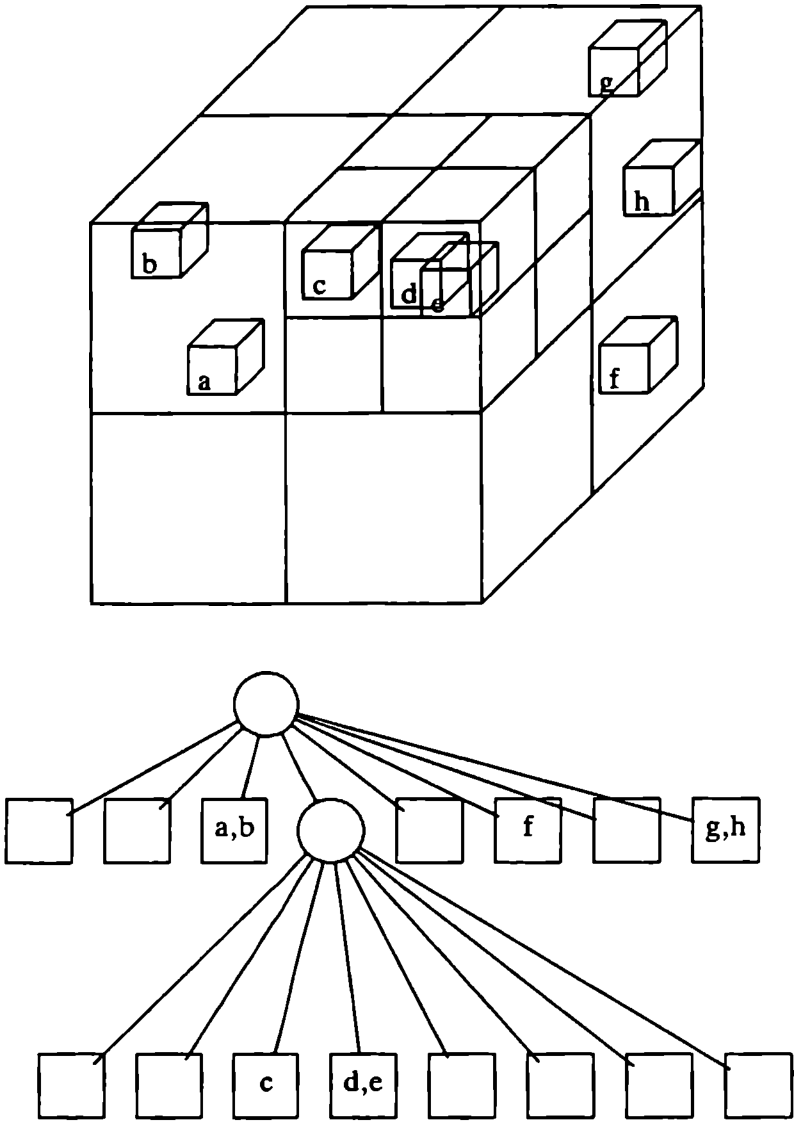


Figure 2.4

disjoint regions. Therefore the algorithm halts as soon as it finds a leaf which has an intersected object. This is an advantage over the HE tree. Although the HE tree may be traversed in order of most likely intersection, the algorithm cannot be halted as soon as a leaf with an intersected object is found, because the objects contained in the nodes may overlap.

The choice of slicing planes for each axis of subdivision in an octree may be any arbitrary plane within the current box, or may be the plane that is halfway between the sides of the box (the **spatial median**), as is the case in Figures 2.3 and 2.4. Choosing the spatial median means that the positions of the planes need not be stored in each node, because they can be generated from knowing the limits of the node. The tradeoff here is more compact storage in return for a restriction on the subdivision process. Traversing a spatial median octree is analogous to performing a binary search of space, since the search range is sliced in half at each node.

There is an important clarification to be made concerning the determination of whether a certain object belongs in a given node of an octree. An object belongs in a node only if the **surface** of the object intersects the node's box. The reason for this is that the point of intersection of a ray with an object cannot occur within a box that does not contain some part of the surface. Consider the object and box in Figure 2.5. The box is entirely within the object. Rays can only intersect the object outside this box. Therefore the object need not be stored in the node associated with the box, because the intersection of the ray with the object will be correctly found in some other node which contains part of the surface of the object.

Octrees have a problem during traversal peculiar to space subdivision hierarchies. Depending on the implementation, an object may be stored in more than one node, and may not necessarily be totally enclosed by any particular node. Therefore, an intersection test of a ray with an object may find an intersection point outside the current node. The algorithm as described would assume that this is the nearest point of intersection and halt. However, because the intersection point is outside the node, we have no guarantee that there is not a closer intersection point with some other object in the scene.

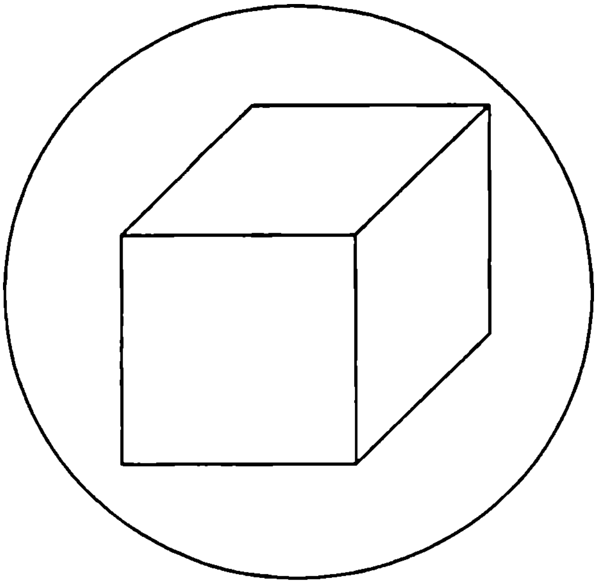


Figure 2.5

The solution is to always check a ray-object intersection point to see if it is within the node currently being examined. If it is not, we ignore the intersection point. This solution raises another problem, that of testing a ray with the same object more than one time. This inefficiency may be corrected by a ray-object cache, indicating whether the intersection test has been previously performed, and what the result was.

The two-way analog of the eight-way octree is the **k-d tree** or **bintree** [Same84]. The only difference is that where the octree divides a node into eight subnodes, using three splitting planes, a bintree divides a node into only two subnodes, using just one splitting plane. Any octree can be represented by a corresponding bintree. The subdivision of a node in an octree can be represented by 3 levels of subdivision of a node in a bintree. This is demonstrated in Figure 2.6.

There are a couple of advantages of bintrees over octrees. Although any octree can be represented by a bintree, not all bintree subdivisions can be represented exactly by an octree. Bintrees can therefore represent a larger

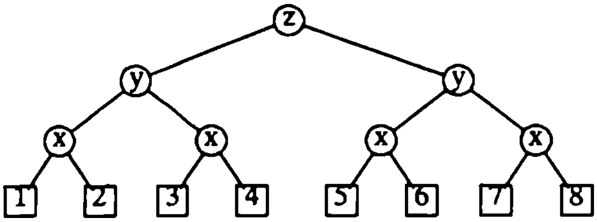


Figure 2.6
x, y, z represent splitting planes
1 through 8 denote corresponding children in octree

class of subdivisions than octrees. (It should be noted, however, that an arbitrary bintree subdivision can be accomplished with a degenerate octree, but possibly with regions of zero volume so that it is not exactly the same decomposition). Also, bintree algorithms tend to be simpler than those of octrees. This stems from the complexity of dealing with three simultaneous slicing planes per node, in the case of the octree, as compared to only one slicing plane, in the case of the bintree. Therefore, it is often more convenient and more efficient to use bintrees for space subdivision [Kapl85].

2.3.3. Static Versus Dynamic Structures

In computer animation, it is common for scenes to change from frame to frame, as objects appear, disappear, and change position, shape, colour, and other attributes. It is often desirable to allow the data structures ordering the scene to be updated to reflect these changes, rather than to rebuild the entire structure for each different frame of the image. An important issue when choosing a data structure to represent scenes is whether the structure allows dynamic modification as the scene changes, and whether the dynamic modification is more efficient than rebuilding a static structure each time the scene changes.

The issue of dynamic structures greatly adds to the complexity. The remainder of this thesis deals only with static structures, leaving the extension to dynamic structures an issue for later research. The restriction to static structures is not unreasonable, as static structures are appropriate in cases where the viewpoint changes often compared to the objects in the scene, or in very complex scenes or complex ray tracing algorithms, where the hierarchy construction time is insignificant compared to that consumed by the tracing of rays. The discussion of static issues may provide a reasonable foundation on which to base a treatment of dynamic structures.

One specific method of dealing with dynamic objects is to treat time as simply another dimension, with the data structure subdividing the objects in 4-space. Issues of this nature are not included in the discussions of this thesis, but extension to four or more dimensions is possible. A similar extension to 4 dimensions by Glassner [Glas88] was successful.

2.3.4. Multiprocessor Algorithms

There have been many designs for ray tracing on multiprocessor machines, each splitting the load over multiple processors in a different way. These methods can be partitioned into two main groups, based on how the work is divided among the various processors. **Image space** algorithms assign different subregions of the image space (groups of pixels) to different processors, requiring the entire scene description to be available to each processor, which computes its part of the image independently from the other processors. **Object** space algorithms assign different parts of the object space (the scene) to different processors. Each processor effectively acts as a leaf in a space subdivision strategy and only stores the objects within its particular subvolume of space. The passage of rays through subvolumes of space requires additional communication overhead in the form of messages from the processor that a ray is in to the next processor.

An important consideration in all multiprocessor algorithms is **load balancing,** which is the process of maximizing throughput by ensuring that all processors perform approximately equal amounts of work so that processor idle time is minimal. Load balancing appears in a static form and a dynamic form. Static load balancing attempts to divide the workload evenly among the processors before ray processing starts. Dynamic load balancing monitors the workload of the processors and changes the partitioning of the work if the workload becomes too unbalanced. Again, treatment of multiprocessor issues would greatly complicate the discussion, so extension of the ideas of this thesis to multiprocessor algorithms is mentioned to a very limited extent. [Dipp84] and [Nemo86] provide treatment of space subdivision issues on multiprocessors.

2.4. Previous Space Subdivision Algorithms

Octrees and bintrees have been used to speed up ray tracing with considerable success. Several papers describing the techniques and results have appeared in the literature. A survey of three papers relating to octrees and bintrees is presented here, with concentration on three aspects of the algorithms: construction, storage, and traversal of the structures. The remainder of this thesis examines the issues of space subdivision hierarchies with respect to these three areas.

2.4.1. Glassner — 1984

Andrew Glassner has published several papers dealing with space subdivision structures for ray tracing. His first paper [Glas84] represents one of the earliest published applications of octrees to ray tracing. His two later papers [Glas87c] and [Glas88] expand upon the ideas within this first paper but many of the basic concepts remain the same. The following sections primarily describe Glassner's original octree.

2.4.1.1. Glassner — Construction

Glassner implemented an octree which selects the spatial median slicing planes. This provides for a straightforward subdivision process, because no algorithm is needed to determine which plane should be used to subdivide a given node. His method of construction of the octree is a simple breadth first technique. Nodes which have more than a certain number of objects are subdivided, until a certain size of tree is reached.

The tree building is governed by two parameters: the maximum number of nodes and the threshold value used for determining whether to split. The threshold value is some small number greater than or equal to 1. Using a threshold value to govern subdivision is simple and straightforward, but does not necessarily subdivide the nodes that need it most. For instance, imagine a tree that has a few objects (less than or equal to the object threshold) in a node with a large bounding volume, and many objects in a node with a small bounding volume. Glassner's algorithm subdivides the smaller volume, rather than the large node. However, due to its relative size, it is likely that only a few rays go through the small node, while many intersect the large node. Therefore, subdividing the smaller gives very little performance gain. It is better to subdivide the larger node.

Glassner's algorithm fails to create a better tree because it does not explicitly take into account any measure of the chance of a ray intersecting a node. Glassner seems to have recognized this, because, in a more recent paper, he uses an improved algorithm [Glas87c]. He outlines a method whereby a node is subdivided if it contains more than a threshold number of objects, or is larger than a given volume. This method, although an improvement, is not very satisfactory as it seems that the choice of threshold is very critical to the performance of the tree. Selecting a threshold to obtain optimal performance for a given scene is difficult. In Chapter 3, an algorithm which performs this selection dynamically is proposed.

2.4.1.2. Glassner — Storage

Glassner assigns to each node in the octree a unique integer identifier based on the position of the node in the tree. The root is assigned the value 1, and each node u, is assigned the identifier pi, where p is the identity of the parent and i $(1 \leq i \leq 8)$ indicates which of the eight children u is. Therefore the children of the root would have the identifiers 11 through 18. The nodes are stored in a hash table, in blocks of eight siblings to conserve memory. The identifier of the parent of a given node is hashed to give the position in the table of the block of eight siblings. The desired node is found by using the last digit of the node's identifier as an index into the block. Collisions are resolved by separate chaining.

The information that is actually stored for a node is the id number, a subdivision flag indicating whether or not it is a leaf, the center (splitting planes), dimensions of the node, and an object list pointer. The dimensions of the node may be omitted and generated from the splitting planes as the tree is traversed, trading speed for storage. Similarly, if a spatial median scheme is used, the center field may also be omitted and generated as the tree is traversed. The object list pointer points to a list of object indices which constitute the list of all objects belonging to the node. In Glassner's implementation, it is just an index into an array containing object index lists for all nodes. Each list is terminated by a predefined nil value.

2.4.1.3. Glassner — Traversal

Glassner utilizes two basic procedures in the traversal of an octree. One procedure is given a point (x,y,z) and returns the leaf node which contains this point. The other procedure takes a leaf node, and returns a point (x,y,z) just beyond the point of exit of the ray in question. The point returned is computed so as to guarantee that it is within the next leaf node along the ray. A traversal calls these two functions, alternately using the output of one as the input of the other. This enumerates the leaf nodes intersected by the ray in order of nearness to the ray origin. The objects within the enumerated leaves are tested for intersection, and the algorithm halts at the first leaf with an intersected object.

The first procedure starts at the root node, with node identifier set to 1. By comparing the given point with the splitting planes stored at the node, it determines the id of the child in which the point belongs. The child node is found in the hash table, and if the subdivision flag is set, then the procedure is repeated because the node has children. Eventually a leaf node is found, which is returned as the leaf containing the point. This method of finding a node requires integer multiplications and divisions, as well as a real comparison, for each node in the path to the leaf. Since two consecutive leaves along the path of a ray generally share several nodes on their paths, Glassner's approach is inefficient, since it does not use this information to find leaves. Because the ray exit point can be used to determine a unique next leaf id, a simple optimization of Glassner's traversal algorithm is to perform a binary search for the next leaf.

The other procedure, which finds a point in the next leaf, performs an intersection test of the ray with the leaf voxel to find the *t* value of the ray at the exit point. This value is used to get the coordinates of the point of exit. Then a small offset is added to each coordinate whose corresponding side of the voxel contains the point of exit. This small offset is simply half the minimum length of the edges of all voxels in the tree, which is recorded during construction. By adding this offset to the point of exit, Glassner creates a new point which is within the boundary of the next leaf, but not on the boundary, thereby avoiding reporting more than one leaf node. This procedure is very similar to the ray-box intersection test, as is illustrated by the simple coding of the exit point algorithm given in Figure 2.7.

```
PROCEDURE RayBoxExit( RayOrigin, RayDirection, Box, VAR Exit : Point );
CONST
   Dimensions = 3;
VAR
   tmin, tmax, tlow, thigh, tenter, texit : REAL;
BEGIN

(* set the range of t to [0 to +∞] *)
   tmin := 0;        tmax := +∞;

   FOR coord := 1 TO Dimensions DO
      IF RayDirection[coord] <> 0.0 THEN

        (* calculate the range of t between the two parallel planes,
      store in tlow, thigh                                             *)
         t1[coord] := (Box[1,coord] - RayOrigin[coord]) /
                                     RayDirection[coord];
         t2[coord] := (Box[2,coord] - RayOrigin[coord]) /
                                     RayDirection[coord];

         tlow := MIN( t1[coord], t2[coord] );
         thigh  := MAX( t1[coord], t2[coord] );

        (* get intersection of range so far, and range [tlow..thigh] *)
         tmin := MAX( tmin, tlow );
         tmax := MIN( tmax, thigh );
      ENDIF;
   ENDFOR;

(* now we know the t value at point of exit (tmax)
  so determine the coordinates of the exit point *)

   FOR coord := 1 TO Dimensions DO
     (* if ray leaves box by the lower bound on this coordinate *)
      IF (t1[coord] = tmax) THEN
         Exit[coord] := t1[coord] - offset;
     (* if ray leaves box by the upper bound on this coordinate *)
      ELSIF (t2[coord] = tmax) THEN
         Exit[coord] := t2[coord] + offset;
      ELSE  (*  interpolate this coord at point of exit  *)
         Exit[coord] := RayOrigin[coord] + tmax * RayDirection[coord];
      ENDIF;
   ENDFOR;

END RayBoxExit;
```

Figure 2.7.

In performing the intersection test of the ray with the leaf's voxel, Glassner tests the ray against all six sides of the voxel. However any given ray needs to be tested against only three sides of the voxel, one per coordinate. By inspecting the signs of the components of the direction vector of the ray, it can be determined, for each coordinate, by which one of the two sides of a box the ray can possibly exit. This is independent of the size or position of the box, and is simply a function of the direction vector of the ray. This observation may be used to improve the exit point computation. Although reducing the computation by about half, the basic method would still require three divisions, three subtractions, and three comparisons for this operation, plus two multiplications and two additions for determining the point of intersection from the t value. These operations are performed quite frequently, many times per ray, and therefore the hierarchy traversal can constitute a major expense in Glassner's algorithm.

2.4.2. Kaplan — 1985

In 1985, Michael Kaplan published an article describing an implemention of a bintree which is very similar to Glassner's octree [Kapl85]. The following sections describe this bintree and how it differs from Glassner's octree.

2.4.2.1. Kaplan — Construction

Kaplan uses a method very similar to Glassner's, differing mainly in that he uses a bintree to represent the subdivision rather than an octree. A node is subdivided at the spatial median in each of the three coordinates and three levels of subnodes are created to represent this subdivision. Kaplan does not state why he chose the bintree representation of an octree. However, the traversal algorithm for a bintree is simpler, and a bintree typically results in fewer leaves than the corresponding octree.

Kaplan builds a bintree which he calls a **BSP tree**, a term which he borrows from Fuchs [Fuch80]. Fuchs organized polygons for visibility processing by creating a structure which he called a binary space partitioning tree. Each interior node contained one polygon which was used to subdivide the polygons in the corresponding region of space. Kaplan's version of the BSP tree is similar except that his slicing planes are determined by the position of the node in the tree and are major planes, whereas Fuchs' planes can have arbitrary normal vectors (and are usually planes chosen so as to be coplanar to the faces of individual objects within the scene).

Kaplan calls interior nodes slicing nodes, and leaves, box nodes or termination nodes, depending on their function. A box node is a standard leaf, as previously described, associated with a box in the object space and possibly containing objects. A termination node is a node representing space outside the object space. Reaching a termination node during traversal signals that the ray has left the scene without hitting an object.

The construction of the tree is governed by the same criteria as Glassner's later method [Glas87c]. A node is subdivided if it contains more than a threshold number of objects, or if it is larger than a threshold size. Kaplan suggests using 1 as the threshold number of objects. The problems with this approach are the same as those for Glassner's method.

2.4.2.2. Kaplan — Storage

Kaplan does not give much detail in his description of how his algorithm stores the bintree. However, it appears that he stores the data structure as an explicit tree, with two pointers at each node indicating the children.

2.4.2.3. Kaplan — Traversal

Kaplan uses the same traversal algorithm as Glassner. He does not specify how he finds a point in the next node, other than saying that the point of the ray's exit from the box is found and pushed far enough to place it in the next node.

2.4.3. ARTS — 1986

Fujimoto, Tanaka, and Iwata describe what they considered a significant speed breakthrough with regard to space subdivision structures for ray tracing [Fuji86]. Although their octree approach is very similar to Glassner's, their paper includes several ideas worth mentioning.

2.4.3.1. ARTS — Construction

Fujimoto, Tanaka, and Iwata created ARTS, which stands for Accelerated Ray Tracing System. The distinction of the ARTS method is the speed of its traversal algorithm, as opposed to the uniqueness of its octree. The traversal algorithm uses incremental integer arithmetic to enumerate the space through which a ray travels. It is a three dimensional adaptation of the standard two dimensional DDA (Digital Differential Analyzer) used to draw lines on bitmaps. This traversal algorithm can be used to traverse an octree, or a special structure the authors call **SEADS**, which stands for spatially enumerated auxiliary data structure. SEADS represents a uniform space subdivision, meaning that the object space is divided into equal sized boxes. The authors claim that the ray-scene intersection algorithm runs faster with SEADS than with the octree, but do not give details of the basis of comparison and provide little justification for this claim. The paper gives no details on the construction of the octree, but there is reference to both Glassner's and Kaplan's papers. Also, the traversal algorithm requires a subdivision algorithm that picks the spatial median as the slicing plane for a node. Therefore, it can be reasonably assumed that their construction of an octree is quite similar to those of Glassner and Kaplan.

2.4.3.2. ARTS — Storage

The ARTS paper depicts explicit storage of the octree as a tree. Two parallel arrays are used, where each set of eight entries corresponds to a single interior node's children. For instance, the first eight entries in the two arrays correspond to the eight children of the root. One array simply indicates node status, whether it is a subdivided node, empty leaf, or a leaf containing objects. The other array contains an index which is a pointer to a list of objects if the status indicates this is a leaf node containing objects, a pointer to the block of eight children if the status indicates subdivision, or undefined otherwise.

This linked list method of storage is superior to Glassner's hash table strategy. The ARTS method requires one byte for the status and four bytes for the index, per node. Similarly Glassner requires one byte for the subdivision flag and four bytes for the object list pointer. However, Glassner requires an additional four bytes (or more) for id and four bytes for a hash table link, for each eight nodes, plus space for the hash table of pointers. In terms of the number of nodes N in the tree, which is eight times the number of interior nodes, the memory required is

$$\mathit{HashTableSize}*4 + \frac{N}{8}*(4+4+8*(1+4)) \sim 6*N.$$

The ARTS method requires 5 bytes per node:

$$5*N,$$

which is about 16 percent less space than Glassner's. In addition to being more compact, the ARTS method has faster access than Glassner's, because of the explicit links to the children stored in the ARTS structure. Indexing is used to find the child of a node, rather than computing an index and searching a hash table as in Glassner's method.

2.4.3.3. ARTS — Traversal

The traversal of the octree is the key to the speed of the ARTS method. Space is partitioned into small voxels of a fixed size. Using incremental integer arithmetic, the algorithm determines which of these voxels that a ray travels through and, using these, which leaves the ray intersects. The size of the voxels is appropriately chosen so that the smallest leaf node in the octree is a nonnegative power of two times the size of the small voxels. The splitting planes of the octree coincide with faces of the small voxels, allowing a straightforward mapping of a small voxel to a leaf node; determination of the leaf node containing a given voxel can be performed by inspection of the bits of the index of the enumerated voxel. When one of the small voxels is enumerated, the leaf in which it resides is found and processed. More voxels are enumerated until a voxel is found outside the leaf, and the next leaf is found and processed. Rather than search from the root each time a leaf is to be found, as Glassner does, the ARTS system traverses upwards from the previous leaf only as far as required and then down to the leaf in question. The authors claim this can be done quite efficiently using byproducts of the incremental integer arithmetic algorithm, but do not elaborate on how this is done. The traversal of an octree is therefore very efficient.

Chapter 3
Improvements

This chapter outlines a number of methods for improving the efficiency of space subdivision data structures, with individual concentration on the construction, storage, and traversal phases.

3.1. Construction

As seen in Chapter 2, current methods of construction for bintrees and octrees are quite simple, providing fast generation of the data structure. However, the construction of the bintree or octree is typically insignificant compared to the computation spent in actually traversing the tree to determine ray-object intersections. Therefore it would be advantageous to devote a greater amount of time to creating a more efficient tree, under the assumption that the extra time would then be saved during the tree traversal.

3.1.1. The Surface Area Metric

To devise such an efficient construction algorithm requires some understanding of efficiency of bintrees and octrees. Currently the only reported efficiency measures are from actual use [Glas88] or from simulations [King86]. Such simulations trace a set of rays through a data structure representing a scene of objects, recording the speed either in terms of actual processing time or the number of basic operations required. The latter is less implementation-dependent and, thus, is more reliable as a general measure. The basic operations are usually the total number of nodes visited and the number of objects tested for intersection. However, such simulations provide no insight into why a particular tree is efficient; it only provides a numerical measure of the efficiency.

A first insight into space-subdivision efficiency may be derived from Stone's [Ston75] observation that the number of rays likely to intersect a convex object is roughly proportional to its surface area, assuming that the ray origins and directions are uniformly distributed throughout object space, and that all origins are sufficiently far from the object. This observation has been used to provide a measure of the likelihood that a ray will intersect a bounding volume in a HE tree [Gold87]. Similarly, we can derive a prediction of the number of objects, interior nodes, and leaves intersected in a given scene and space subdivision hierarchy, thereby doing away with the need for the costly Monte Carlo simulations.

Let us assume that for a given scene and tree, we are dealing with exactly those rays that intersect the bounding volume for the entire scene, the voxel of the root node. Therefore every ray intersects the root voxel, and the probability of a ray intersecting the root node is 1. The probability of a ray intersecting any interior or exterior node is equal to the surface area of the node divided by the surface area of the root. This results in the following intersection estimations:

$$\textit{number of interior nodes hit per ray} = \frac{\sum_{i=1}^{Ni} SA(i)}{SA(root)}$$

$$\textit{number of leaves hit per ray} = \frac{\sum_{l=1}^{Nl} SA(l)}{SA(root)}$$

$$\textit{number of objects tested for intersection per ray} = \frac{\sum_{l=1}^{Nl} SA(l)*N(l)}{SA(root)}$$

where

Ni denotes the number of interior nodes

Nl denotes the number of leaves

$SA(i)$ denotes the surface area of interior node i

$SA(l)$ denotes the surface area of leaf node l

$N(l)$ denotes the number of objects stored in leaf l

Given these measures of the node, leaf, and object visits performed during traversal of the tree, an estimate of the cost of the tree can be obtained. The costs associated with these three components depend on the particular implementation of the traversal algorithm and may be determined theoretically or experimentally. The total cost of a particular tree is determined from the three sums above and the three related costs, which are assumed to be constants for a given implementation. This is expressed as

$$\textit{cost of tree} = \frac{C_i * \sum_{i=1}^{Ni} SA(i) + C_l * \sum_{i=1}^{Nl} SA(l) + C_o * \sum_{l=1}^{Nl} SA(l) * N(l)}{SA(root)}$$

where

$C_i =$ *cost of traversing an interior node*

$C_l =$ *cost of traversing a leaf*

$C_o =$ *cost of testing an object for intersection with a ray*

This cost function assumes that rays do not intersect any objects, but also represents an upper bound for rays that intersect objects. The cost function implies that if an object occurs in two or more leaves, it is tested for intersection each time a ray intersects one of these leaves. Therefore a given object may be tested against the same ray several times. As observed in Chapter 2, this is usually unacceptable, and is avoided by caching objects intersected against a ray, so that each object is tested at most once per ray. The cost function given above must be modified to account for this caching.

To derive the correct cost function, we require a measure of the probability that a ray intersects at least one leaf from the set of leaves within which a particular object resides. This is equivalent to determining the probability that a ray intersects the volume defined by the union of the set of leaves. Because this union may be non-convex, the probability of ray intersection must be estimated by finding a convex region to approximate the non-convex region. Figure 3.1 depicts an object, the set of leaves containing it, and a convex region approximating the non-convex region.

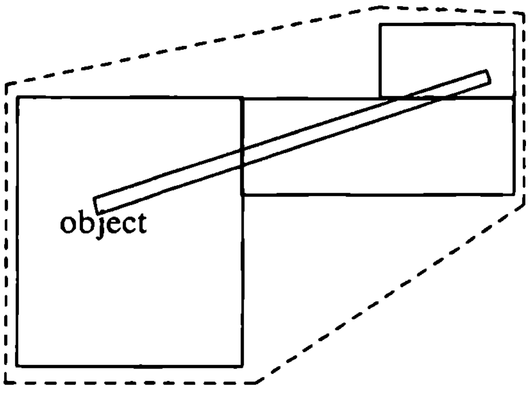


Figure 3.1.

A simpler approximation is the sum of the areas of the projection of the set onto the six faces of the root bounding volume divided by the root bounding volume's surface area. This is illustrated by Figure 3.2 which depicts the same object as Figure 3.1 but includes the root bounding volume and the projection of the set of leaves onto this bounding volume. For a convex object, this measure is exactly equal to its surface area divided by the root bounding volume's surface area, which is the correct measure. Therefore, we can use this approximation for the set of leaves for all objects, whether the set of leaves for each object is convex or not. This makes the object portion of the

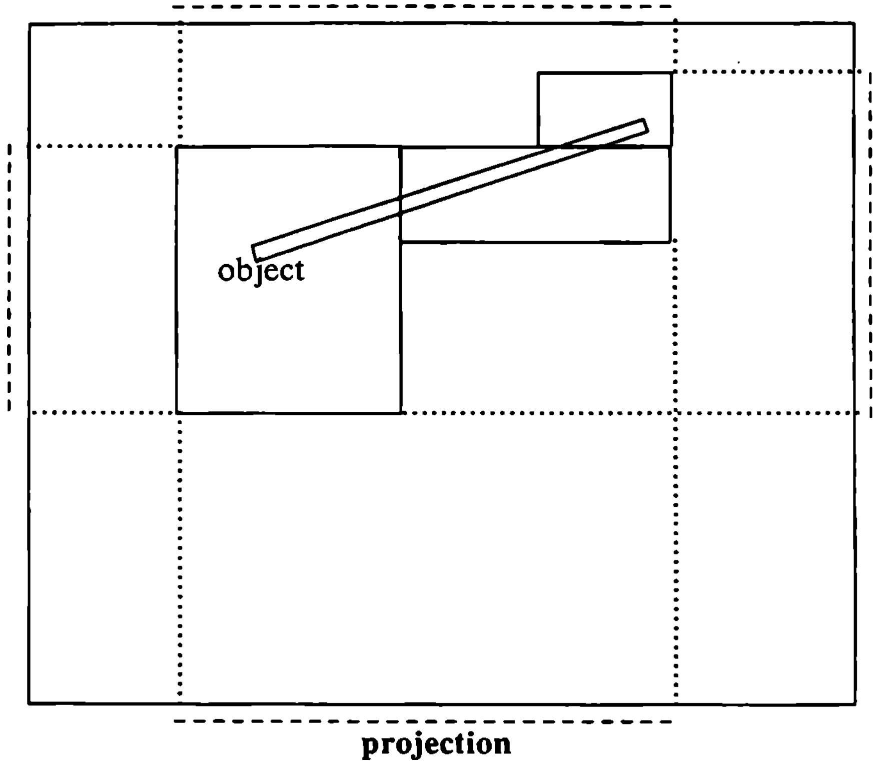


Figure 3.2.

cost of a tree:

$$\textit{object cost per ray} = \frac{C_o * \sum_{o=1}^{N_o} SAset(S_l(o))}{SA(root)}$$

where

N_o *is the number of objects*

$S_l(o)$ *is the set of leaves in which the object o resides*

$SAset(s)$ *denotes the approximate surface area of the set s*

If we assume that the above costs are accurate, we can use these equations to govern the construction of the tree.

Consider the octree methods of Glassner or ARTS, which subdivide nodes in a breadth-first fashion. As the tree is built up from a tree of just one node, the above equations can be used to calculate the overall decrease in cost created by subdividing a particular node. This calculation involves computation of the surface area of the node, the surface areas of its children, and the number of objects belonging in each child, if it were to be created. The calculation is quite simple because each child of a node has exactly one quarter of the surface area of its parent. A reasonable heuristic would be to subdivide whichever node would decrease the cost of the tree most, rather than using a breadth-first selection mechanism. A bound on the size of the tree would still be required to control the amount of space used. Such a heuristic would perform the subdivision where it is needed most, that is, where the most gain would be achieved, as opposed to a breadth-first method which is somewhat more random in its choice of a node to split. This method effectively performs a one-step lookahead to determine the cost of splitting a particular node. The expense of this step may be eliminated by using the current cost of a particular node as its estimated gain and changing the subdivision process to select the node which has the highest cost. The results of this zero-step lookahead can be expected to be similar to the highest-gain method, because in general, the higher the cost of a node, the higher the gain by subdividing it. However, the zero-step lookahead is computationally cheaper.

The cost function can be used not only to select the node splitting as described above, but also to select the position of the splitting plane within a node. Traditionally, this has been the spatial median, with one exception being Heckbert's median split algorithm [Heck82]. Heckbert chooses the object median in a k-d tree where the objects are colour triplets (points). Unlike the other methods described, which are applied to ray tracing, Heckbert uses the space subdivision structure to select a set of colours to be used in a lookup table to approximate a larger set of colours, which are the data being organized. One obvious reason why the spatial median is usually chosen as the splitting plane is that it does not require storage of the position of the splitting plane. Another reason may be that no one has suggested a good algorithm for selecting splitting planes.

The cost equation developed above can also be applied to selecting "good" splitting planes, where the splitting planes are not restricted to being the spatial median. The extra degree of freedom granted by allowing splitting planes to be anywhere within a box may provide a more efficient tree and may make up for the added memory requirements of this splitting plane selection.

In the following discussions of splitting planes, we will, for simplicity, only consider the bintree, since it uses only one splitting plane per node. Only major planes are used as splitting planes and we ignore, for now, the possibility that an object may belong in more than one node. Let us look at the subdivision along a particular axis c when we have to choose a value for some parameter b, where $b = 0$ corresponds to the lower bound of the coordinate c of the box and $b = 1$ is the upper bound. Choosing $b = 0.5$ is equivalent to selecting the spatial median (Figure 3.3). Let us look at the cost savings as a function of this parameter b. We observe that the internal node and leaf node components of this cost savings function are constant with respect to b. Therefore, for the purposes of minimizing cost, one can minimize the following function:

$$f(b) = LSA(b)*L(b) + RSA(b)*(n-L(b)) - SA*n$$

where n is the number of objects in the node, $L(b)$ is the number of objects to the left of the plane at b, and $n-L(b)$ is the number to the right. The surface area of the left and right subnodes are $LSA(b)$ and $RSA(b)$, respectively, and the surface area of the node itself is SA. The first term represents the probability that a ray intersects the left subnode multiplied by the number of

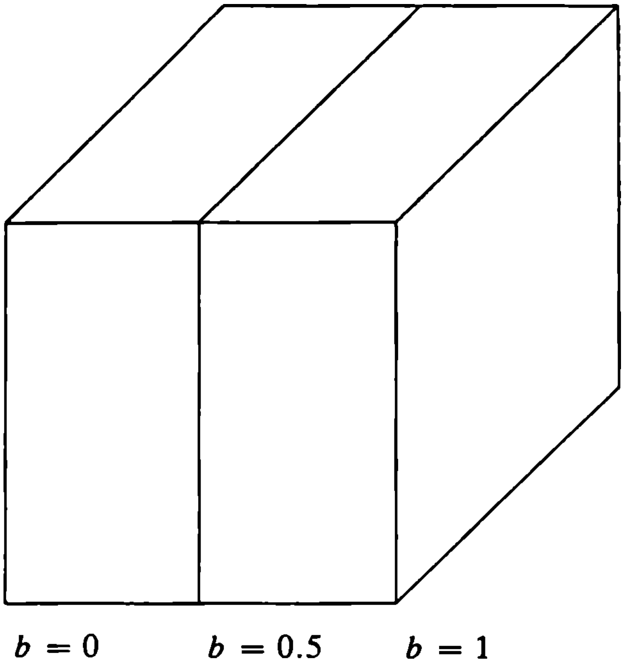


Figure 3.3

intersection tests performed in the left subnode. The second term is a similar quantity for the right subnode. The "$SA*n$" term is the amount of work required if the node were not subdivided and thus is an amount of work saved by changing the original node from a leaf to an internal node, hence the minus sign. This last quantity is a constant with respect to b, so it may be removed from the function, resulting in:

$$f(b) = LSA(b)*L(b) + RSA(b)*(n-L(b))$$

To find a "good" splitting plane, one might evaluate this function at several different positions, and choose the position with the minimum value. However, let us examine the behaviour of this function. The value of this function at the spatial median (b=0.5) is

$$n*LSA(0.5),$$

because $LSA(0.5) = RSA(0.5)$. Curiously enough, the value of this equation at the **object median**, where half of the objects are on each side of the slicing plane, $(L(b)=\frac{n}{2})$, is also

$$(LSA(b)+RSA(b))*\frac{n}{2} = n*LSA(0.5)$$

because $LSA(b)+RSA(b)$ is a constant, $LSA(0.5) = RSA(0.5)$, therefore $LSA(b)+RSA(b)$ may be substituted by $2*LSA(0.5)$. This shows that picking the object median results in the same gain as picking the spatial median. Intuitively, one might assume that picking the object median would be a reasonable heuristic for choosing an arbitrary splitting plane, but the above observation indicates that it is equivalent to the standard spatial median subdivision.

The optimum heuristic is to pick the slicing plane which minimizes $f(b)$. Differentiating with respect to b gives

$$f'(b) = LSA'(b)*L(b) + LSA(b)*L'(b)+ n*RSA'(b) -$$
$$RSA'(b)*L(b) - RSA(b)*L'(b)$$

which can be simplified by substituting $-LSA'(b)$ for $RSA'(b)$ (because $LSA(b) + RSA(b)$ is a constant), giving,

$$f'(b) = (2*L(b)-n)*LSA'(b) + (LSA(b) - RSA(b))*L'(b)$$

Since $L(b)$ is a discontinuous function, $L'(b)$ is not defined. However, for the purposes of minimization of $f(b)$, we can assume that $L'(b)$ is always nonnegative (the number of objects stored in the left subnode cannot decrease as b increases). Let us try to narrow down the range in which the minimum might lie. Let us investigate the case where the object median lies at some point $b<0.5$. To the left of the object median, $f'(b)$ is negative, because $L(b)<\frac{n}{2}$ and $LSA(b)<RSA(b)$. To the right of the spatial median, $f'(b)$ is positive, because $L(b)>\frac{n}{2}$ and $LSA(b)>RSA(b)$. Therefore the minimum must occur between the object median and the spatial median in the case where the object median is to the left of the spatial median.

A similar proof can be used for the other case where the object median is to the right of the spatial median, thereby proving that for any node and set of objects within it, the optimum slicing plane occurs between the object median and the spatial median, reducing the required search range. Figure 3.4 represents a set of 10 objects within a single node in two dimensions, the spatial and object medians, and the narrowed range of search for the optimum splitting plane.

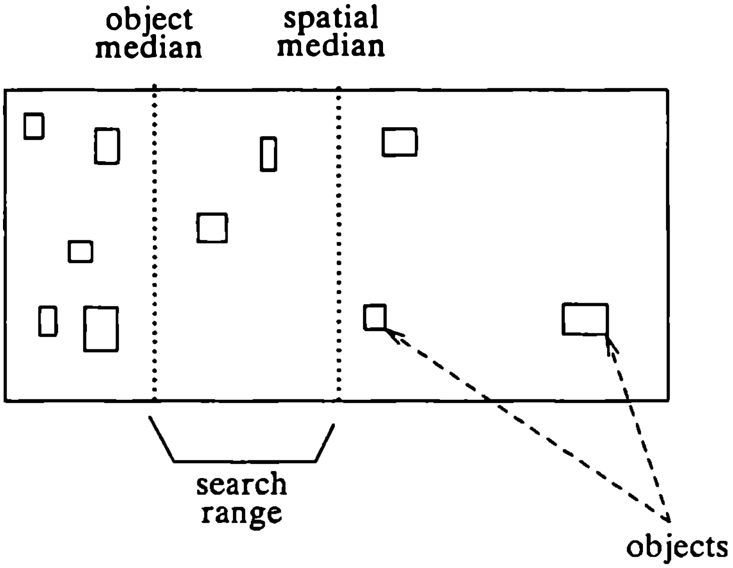


Figure 3.4

The optimum splitting plane occurs within the reduced range, and at the upper or lower edge of one of the objects within this range, rather than in the middle of "white space". To take advantage of this reduced range, one must first find the object median, which is easy if the objects are sorted, but otherwise requires a binary or Fibonacci search of the space. If one does not want to perform this search, one can determine how many objects are on each side of the spatial median, thereby determining on which side of the spatial median the object median occurs. This allows one to cut the search space in half. In the cases of small numbers of objects, one can try splitting planes at the limits of each object within the appropriate half and record the maximum. For large numbers of objects, one might try a small set of splitting planes, at equally spaced intervals, or even randomly selected, within the appropriate half. Alternatively, a cheap heuristic is to select the splitting plane midway between the object median and the spatial median.

3.1.2. Objects Spanning the Slicing Plane

If we relax our restriction about dealing only with cases where objects belong in exactly one leaf, the equation gets slightly more complicated. The increased cost function is now

$$f(b) = LSA(b)*L(b) + RSA(b)*R(b) + SA*(n-L(b)-R(b)) - n*SA$$

(which can be reduced to $-b*R(b) - (1-b)*L(b)$). $L(b)$ represents the number of objects that belong in the left subnode but not in the right, $R(b)$ represents the number of objects that belong in the right subnode but not the left, and $n-L(b)-R(b)$ is the number of objects that belong in both subnodes, so called **spanning objects**.

The first two terms in the above equation represent an estimate of the number of nonspanning objects tested for intersection due to the left and right subnodes. The third term represents the number of spanning objects tested, assuming that the algorithm uses an object cache to avoid multiple intersection tests with the same object. Note that we approximate the probability that spanning objects are tested for intersection by the surface area of the original node, because a ray that intersects this node causes ray-object tests with the spanning objects independent of which subnodes are intersected. As before the term $-n*SA$ represents the cost of the undivided node.

The reduced search space may be used if there are no spanning objects at the spatial median and an object median with no spanning objects can be found. Otherwise, one may apply the simple method of sampling the function value at various positions to determine a minimum. The minimum position may not be easy to find using numerical techniques, because the second derivative is not necessarily negative, causing the function to be nonmonotonic on either side of the minimum. This can happen when two objects overlap along the axis of the appropriate coordinate. However, if there is no such overlap, then the set of planes at the upper and lower limits of each of the objects contain the optimal plane, and the cost function is monotonic decreasing before the optimal plane, and monotonic increasing after. A binary or Fibonacci search can then be performed to determine the optimal splitting plane.

In a bintree approach, it is necessary to determine the optimum axis of subdivision as well as the optimum splitting plane for this axis. The above algorithm can be used to select the optimum slicing plane and gain for each of the axes. The axis with the maximum gain is chosen as the axis of subdivision for this node. For an octree, the problem of selecting the three optimum slicing planes is more complicated, because of the need to deal with three axes at once. One solution is to apply the above algorithm to each axis independently to select a "good" slicing plane for each axis.

We have now seen how a "good" splitting plane may be selected, and how to choose the next node for subdivision. This leads to an overall algorithm for construction of a bintree or octree with arbitrary slicing planes.

3.1.3. Optimal Trees

It is worthwhile investigating optimal space subdivision trees, but it is first necessary to define optimality. A tree may be optimal with respect to space or time (search cost), or a combination of both. The tree that is optimal in terms of space for a given scene is a tree of one leaf containing all the objects in the scene, clearly an impractical criteria for optimality. Likewise, the tree that is optimal with respect to time may require impractical amounts of storage. Therefore the definition of optimality should combine the space and time factors. One might define optimality as the most compact tree that provides the desired query time, or as the most efficient tree for a given amount of space, which is the definition used in the following discussion.

3.1.3.1. Spatial Median Subdivision of Point Objects

Let us investigate the optimal spatial median subdivided octree, defined as the most efficient tree for a given size. The size of an octree is defined as the number of interior nodes. Let us restrict the objects to non-coincident uniformly distributed random points. This choice of object type guarantees that each object belongs in exactly one leaf. Subdividing a node results in a predictable gain, assuming the relative costs of visiting leaves and internal nodes, and of intersection tests are given. For instance, the decrease in the number of intersection tests due to splitting a node into eight subnodes is $\frac{3}{4} * SA(node) * \#objects\ in\ node$.

The algorithm for producing an optimal octree starts with a single leaf enclosing the entire scene and repetitively splits whichever node results in the highest gain until the number of interior nodes is equal to the desired number. This algorithm relies on the observation that optimal trees are built up from smaller optimal trees. More formally, for any optimal octree of size n and for any m greater than n there must be an optimal octree of size m which contains the smaller optimal octree as a subtree. This can be proved by realizing from the previous formulas that the gain in subdividing a child of a node must be less than or equal to the gain in subdividing the node itself.

This observation is used in an inductive proof. Clearly the optimal octree of size 1 is the leaf tightly enclosing the scene, which must be a subtree of all optimal octrees of size greater than 1. If an optimal octree of size $n+1$ does not have a given optimal octree of size n as a subtree, there must be a subdivided node in the size n optimal octree which is a leaf L in the size $n+1$ optimal octree. By virtue of the greatest-gain subdivision process, there must be some node N in the size $n+1$ optimal octree whose children are leaves, such that the gain due to the subdivision of N is less than or equal to the gain which results from subdividing the leaf L.

Therefore, within the larger optimal octree, the node N can be replaced by a leaf, and its former children used to split the leaf L, resulting in a nonnegative overall decrease in the cost of the octree. This rearrangement of the nodes within the size $n+1$ optimal octree results in a new octree which is still of size $n+1$, and is at most as costly as the original, and hence is still a size $n+1$ optimal octree. This process of moving nodes can be repeated until we have an optimal octree of size $n+1$ which has the given size n optimal octree as a subtree.

As a larger optimal octree is built up from smaller ones, the algorithm eventually separates all points such that every leaf has exactly one or zero points within it. After this, the algorithm repetitively subdivides the largest leaf which has an object in it, because the number of objects in all such leaves is equal. Eventually all the non-empty leaves are on the same level, and the algorithm effectively performs the same breadth first subdivision process that results from Glassner's or the ARTS method with the object threshold set to zero (subdivide if greater than zero objects in the node).

For octrees that have reached this stage, the estimated number of ray-object intersection tests is equal to the number of objects * $\left(\frac{1}{4}\right)^l$, where l is the number of levels in the octree, with a size 1 octree corresponding to $l=0$. Other characteristics of the octree can be estimated also. Let n be the number of random point objects in the scene, and h be the height of an optimal octree representing the scene ($h=0$ corresponds to a 1-node octree). We assume that the optimal octree is sufficiently large that each leaf has exactly one or zero objects stored in it, and every leaf with an object within it is at level h.

Before deriving estimates for optimal octrees, we first devise some measures for complete octrees of arbitrary height l, also assuming spatial median subdivision. At level l, the number of leaves is 8^l. The number of occupied leaves $O(l)$ is:

$$O(l) = \sum_{i=1}^{min(8^l, n)} i*P(i, 8^l, n)$$

where $P(i, p, n)$ is the probability that the n randomly distributed points result in exactly i occupied leaves given that the total number of leaves is p. This is computed as:

$$P(i, p, n) = P(i-1, p, n-1)*\frac{p-(i-1)}{p} + P(i, p, n-1)*\frac{i}{p}$$

where $P(i, p, n)$ is 0 if $n<i$, and $P(1, p, n)$ is $\left(\frac{1}{p}\right)^{n-1}$.

We can now use the formula for the number of occupied leaves in a complete octree to compute some statistics for the optimal octree of height h. The number of empty leaves at level l in the optimal octree is equal to the total number of leaves in the corresponding complete tree (8^l) minus two quantities. The first quantity is the number of leaves that are subdivided and hence are not empty leaves at level l. This is the number of non-empty leaves in the corresponding complete tree at level l, which is $O(l)$. The second quantity represents the number of leaves at level l which are missinig because one of their ancestors is a leaf. This is the sum of the number of empty leaves at each level less than l weighted by the number of level l leaves these larger leaves represent. The resulting formula for $E(l)$, the number of empty leaves at level l in the optimal octree is:

$$E(l) = 8^l - O(l) - \sum_{i=l-1}^{0} 8^{l-i} * E(i)$$

The number of occupied leaves in the optimal octree is n, because all leaves contain exactly zero or one objects. $I(l)$, the number of interior nodes at level l in the optimal octree, is the number of occupied leaves in the corresponding complete octree.

Using these estimates the following statistics regarding the size and performance of the optimal spatial median octree of height h can be estimated:

Size of Tree:

$$\begin{aligned} \textit{number of interior nodes in octree} &= \sum_{l=0}^{h-1} O(l) \\ &= \sum_{l=0}^{h-1} \sum_{i=1}^{min(8^l,\, n)} i * P(i,\, 8^l,\, n) \\ \textit{number of empty leaves in octree} &= \sum_{l=0}^{h} E(l) \\ &= \sum_{l=0}^{h} \left(8^l - O(l) - \sum_{i=l-1}^{0} 8^{l-i} * E(i) \right) \\ \textit{number of non-empty leaves in octree} &= n \end{aligned}$$

Performance of Tree:

$$\textit{estimated interior node visits} = \sum_{l=0}^{h-1} \left(\frac{1}{4}\right)^{l} *O(l) =$$

$$= \sum_{l=0}^{h-1} \left(\frac{1}{4}\right)^{l} * \sum_{i=1}^{min(8^l,\, n)} i*P(i,\, 8^l,\, n)$$

$$\textit{estimated occupied leaf visits} = n* \left(\frac{1}{4}\right)^{h}$$

$$\textit{estimated empty leaf visits} = \sum_{l=0}^{h} \left(\frac{1}{4}\right)^{l} E(l)$$

$$\textit{estimated object tests} = n* \left(\frac{1}{4}\right)^{h}$$

We now have an algorithm to produce the optimal spatial median octree for a given size, and we also have an estimate of the number of fundamental operations performed. Using these equations and estimated costs of visiting interior nodes, leaf nodes, and objects, the size which results in the most efficient octree can be determined.

3.1.3.2. Arbitrary Subdivision of Point Objects

If we assume that we have point objects and we remove the restriction of spatial median subdivision, there is no similar algorithm for the construction of an optimal octree. The problem is that the optimal slicing plane must be found for each node. The algorithm given previously for selecting the slicing plane position results in the optimal slicing plane assuming that the children are not subdivided. There is no guarantee that this is the optimal slicing plane if the node has grandchildren.

However, one can estimate the efficiency of the optimal octree without knowing how to construct it. Because the slicing planes can be arbitrarily positioned, the leaves can tightly enclose the objects, which are points. Therefore the estimated number of objects tested for intersection is zero. Since the positions of the slicing planes within a node do not affect the sum of the eight new children's surface areas, it can be estimated that the number of interior nodes and leaves intersected is roughly equivalent to that of the corresponding optimal spatial median octree. Even for non-point objects, the number of objects tested for intersection can be expected to be significantly less for the optimal arbitrary tree due to the ability to provide leaf nodes that tightly enclose the objects. Because the ray-object intersection test is typically the most expensive operation, the optimal arbitrary octree is probably significantly faster than the corresponding spatial median octree. This provides a motivation for arbitrary subdivision instead of spatial median subdivision, even if an optimal algorithm cannot be devised.

3.1.4. Evaluation of Conventional Octree Construction Method

If we assume that objects do not intersect each other and that each leaf has at most one object, the estimated cost of the optimal octree can be used to measure the performance of Glassner's or the ARTS algorithm. However, these assumptions are not valid in practice, because objects often overlap and are so close to other objects that leaves may have many objects stored within them. Hence there is no obvious estimate of the efficiency of Glassner's construction method.

The surface area metrics can be used to illustrate a deficiency with Glassner's method. This deficiency arises from the use of a value greater than zero as the object threshold. This means that only a node which has more than the threshold number of objects is subdivided. A node which has only one object in it, for example, is not subdivided further. Consider Figure

3.5. Every ray that intersects the left subnode is tested for intersection with the object within it. Because the cost of a leaf visit is less than the cost of an object test, it is more advantageous to subdivide this node, reducing the likelihood that a ray is tested against this object. This deficiency can be remedied somewhat by setting the object threshold to zero, which causes the algorithm to subdivide every non-empty node. However, additional traversal cost is introduced.

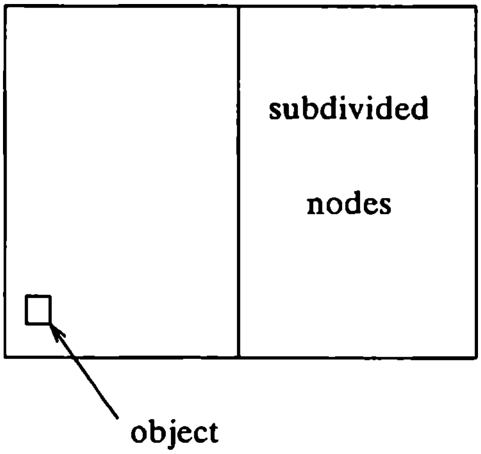


Figure 3.5.

3.1.5. Arbitrarily Oriented Splitting Planes

In 1986, Kay and Kajiya [KayK86] showed how to use precomputation to remove the dot product calculation from the intersection test of a ray with an arbitrarily oriented plane. The computation of the parameter t of a ray was given in Chapter 2 as:

$$t=\frac{-d-N\cdot O}{N\cdot D}$$

For a given ray, $N\cdot O$ and $N\cdot D$ can be precomputed for a set of plane normals. Then each intersection of the ray with a plane having one of these normals requires only one subtraction and one division, the same as the equation for major planes. Because multiplication is faster than division on many machines, divisions are often eliminated by precomputing a reciprocal to be used in multiplication. For our discussion, we assume that division is equivalent to multiplication, because it can be implemented as such.

Our discussion of slicing planes so far has been limited to major planes, but Kay and Kajiya's paper indicates that it is possible to use a set of arbitrarily oriented splitting planes at little extra cost. The surface area metrics can be applied to arbitrary planes to determine the optimum orientation, as well as position, of a splitting plane. The extra degree of freedom gained by allowing arbitrary orientations may result in a more efficient tree, because the surface areas of the voxels containing objects can be minimized further.

The conventional traversal algorithms used by Glassner and Kaplan would not work without modification, because of the necessity of finding a point strictly within the next leaf, as opposed to on or within the next leaf. Because the sides of a voxel are no longer orthogonal to each other, one cannot simply push the ray exit point perpendicular to the exit plane because there is no guarantee that it is within the next leaf. The exact ray exit point must be used, requiring a traversal algorithm which handles this ambiguous point (belonging in two or more leaves) and avoids an infinite loop of processing due to a cycle in the list of enumerated leaves. The algorithms of Glassner, Kaplan, and Fujimoto et al. cannot be used without modification to check for cycles. The basic recursive traversal algorithm given in Chapter 2 can be used without modification.

3.1.6. Scene Complexity and Efficiency

There are many references in the literature to scene complexity, which usually take the form of qualitative statements about the amount of work required to render a scene, or the amount of visual detail of the image. Typically, one talks about **visual complexity** and **computational complexity**. Visual complexity typically deals with issues such as how many pixels compose the image and how they vary in colour and position, while computational complexity deals with how much work is required to render the image. The surface area idea is now extended to provide measures of these two complexities.

We can define the **visual complexity** of a given scene as the probability that a ray that strikes the bounding volume of the scene strikes an object within the scene. The surface area of each object in the scene can be used as estimate of the likelihood of it being intersected by a ray. Alternatively, one might use the object bounding volumes' surface areas instead of the objects themselves, for simplicity. The numerical measure of the visual complexity is the surface area measurement of the set of all bounding volumes, using the SAset function defined previously, divided by the surface area of the scene bounding volume. As described previously, this is calculated by projecting all bounding volumes onto each of the six faces of the scene bounding volume, and dividing the area of the union of these projections by the surface area of the scene bounding volume. A value near 1 means that the majority of rays entering the scene hit an object and almost all pixels in the image are hit. A value near 0 indicates that few rays hit objects and thus that the image is nearly empty. This measure is approximately equal to the fraction of pixels that have colour in the final image, independent of the rendering algorithm, which makes it useful as a general-purpose visual complexity measure.

One may also use this visual complexity measure as a basis of comparison for a rendering efficiency measure. It is reasonable to expect that, ideally, the amount of work performed in rendering should be proportional to the visual complexity. In the case of ray tracing, optimal performance occurs when the number of intersection tests for each ray is zero or one, depending on whether it hits an object or not. By dividing the number of ray-object tests per ray by the visual complexity, one may get a measure of efficiency, where a value of one represents optimal efficiency.

The optimal case is usually not attainable, and it is more desirable to have a general purpose computational complexity measure. **Computational complexity** can be defined as the amount of work involved in rendering the image and approximated as the average number of objects that map to a pixel. A justification of this as a rendering algorithm-independent measure is provided by the observation that the amount of work in ray tracing, for example, is more related to the number of objects the average ray goes near than to the number of rays that hit objects. In other rendering algorithms, such as z-buffering, the work required is proportional to the average number of objects that map to a pixel. One estimate of this computation might be the sum of the object surface areas (or bounding volumes) divided by the scene bounding volume surface area. The computational complexity, being a sum of surface areas, is greater than or equal to the visual complexity, which is in a sense a union of the same surface areas. Computational complexity provides a measure of the average amount of work required to render a scene, as opposed to the minimum amount required, which is provided by the visual complexity.

3.1.7. Load Balancing in Multiprocessor Algorithms

Another application of the surface area metric is in load balancing of multiprocessor algorithms. Static load balancing of an object space algorithm attempts to subdivide space so that each processor performs an equal amount of work. The surface area of a bounding volume times the number of objects within it provides a good estimate of workload for static load balancing. A similar idea may be used for dynamic load balancing of object space multiprocessor algorithms.

3.2. Storage

The simplest and most obvious method of storing the bintree (octree) is as an explicit tree with two (eight) pointers per node. This has a large space requirement, motivating the more compact octree schemes of Glassner [Glas84] and Fujimoto et al. [Fuji86]. It was demonstrated in Chapter 2 that Fujimoto's linked list method was superior to Glassner's hash table method. One deficiency with Glassner's method is the need to perform multiplications and divisions to construct a node identifier, due to the base 10 nature of the identifier. A better encoding of leaf identifiers is to use three bits per level to indicate the position of the leaf, plus four bits to store the level of the leaf. Thus a 32-bit identifier can represent leaves in a tree which has up to nine levels of subdivision. This type of identifier can be manipulated with fast bit operations, an improvement over Glassner's method.

The particular storage method for a tree has a marked effect on the speed of traversing the tree. Traversal of the tree is mainly movement from one leaf to another. The tree should be stored in such a manner as to minimize the computation involved in determining the next leaf intersected. This traversal cost can be decreased by storing links to neighbours on each of the six faces of each leaf. Samet [Same84] describes similar links in region quadtrees, which are termed **ropes**. For the purposes of the following discussion, let each face of each leaf have exactly one neighbour, defined as the smallest node (interior or leaf) whose voxel's surface totally encloses the face of the leaf in question. By this definition, the neighbours of a leaf are not necessarily leaves. However, this definition guarantees that each leaf has exactly one neighbour per face (except leaves on the boundary of the scene, which have none). Figure 3.6 illustrates the neighbours of four leaves.

During traversal of the structure it is now necessary to determine the face exited. The neighbour link of this face is followed and if the neighbour of the face is a leaf, processing of the objects within the leaf is performed. If the

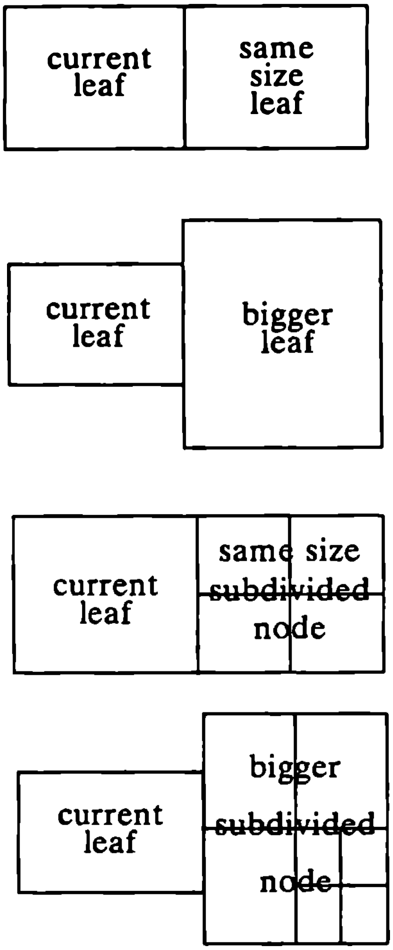


Figure 3.6
The node to the right of the leaf is the leaf's neighbour for the righthand face.

neighbour is an interior node, then the exit point of the current leaf must be computed and used to descend the neighbour's subtree to find the appropriate leaf. This strategy eliminates all upward traversal of the tree and some downward traversal. In general, when a ray travels from one area to an area of equal or less fine subdivision, then the neighbour is a leaf and the hierarchy traversal cost is zero. It is only when travelling to an area of higher subdivision that there is any hierarchy cost. In this case the cost is less than the corresponding cost of conventional methods, because the upward traversal to the common ancestor is eliminated, and some of the downward traversal may be avoided (about equal to the upward traversal eliminated). Therefore, the neighbour links reduce the hierarchy cost significantly, at the added expense of six pointers per leaf.

A further modification of the neighbour links is to redefine the neighbours of a face as **all** leaves adjacent to that face. Now, all neighbours are leaves, but any given face may have more than one neighbour, which requires more memory per leaf than the previous link strategy. However, in the case of spatial median subdivision, the amount of memory required is now less than 12 pointers per leaf, only twice that of the former links method. The upper bound of 12 pointers per leaf stems from the observation that, although some faces have a large number of neighbours, others have only one neighbour, with the average being two pointers per face. This is illustrated in Figure 3.7, which shows $n+1$ faces, and $2*n$ links, and hence $\frac{2*n}{n+1}$ links per leaf, which is less than two pointers per face. With arbitrary subdivision, the number of pointers per face may be higher, because Figure 3.7 no longer covers all possible subdivision cases.

The storage of the neighbours for a leaf consists of six integers representing the number of neighbours of each face, plus a list of pointers to the neighbours of each face. Alternatively, the neighbours could be stored in a two dimensional bintree to quickly determine the appropriate neighbour for a given exit point.

This **complete neighbour links** scheme eliminates the hierarchical traversal altogether, because finding the next node only requires following the links. It introduces the additional cost of determining which link to follow, if a leaf has more than one neighbour on a given face. If a face has n neighbours, on average $\frac{n}{2}$ of them have to be tested (four real comparisons per test) to find the one intersected by the ray, assuming a linear search. If we assume that

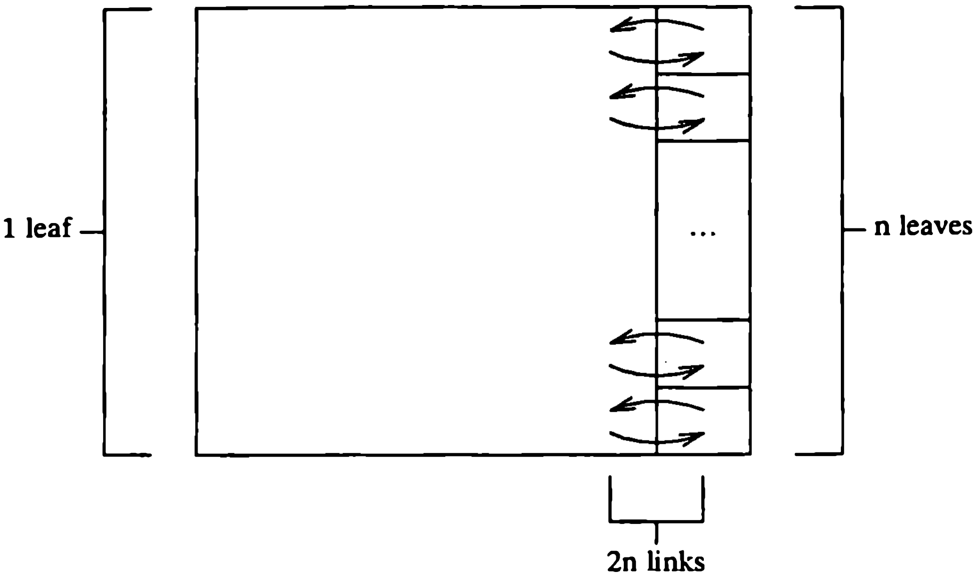


Figure 3.7
$n+1$ leaves, $2n$ links.

the number of neighbours of a leaf is proportional to its surface area (because neighbours share a common surface), then the estimated number of such tests per ray is proportional to

$$\sum_{l=1}^{Nl} sa(l)*sa(l) \sim \sum_{l=1}^{Nl} sa(l)^2.$$

With a roughly uniform subdivision, the volume of each leaf is $\frac{1}{Nl}$, where Nl is the number of leaves. The surface area of a leaf is proportional to the $\frac{2}{3}$ power of the volume, so the number of tests per ray is proportional to

$$\sum_{i=1}^{Nl}\left(\left(\frac{1}{Nl}\right)^{\frac{2}{3}}\right)^2$$

$$\sim \sum_{i=1}^{Nl} Nl^{\frac{-4}{3}}$$

$$\sim \frac{1}{\sqrt[3]{Nl}}$$

This provides the indication that increased subdivision leads to a decreased amount of work per ray for determining which neighbours to follow. In rough terms, each additional complete level added to an octree results in eight times as many leaves but half as much work for determining neighbours.

Better search performance may result by using a two dimensional bintree to search for the neighbours, or by performing a binary search on the sorted neighbour lists. Either of these two methods reduces the expected number of tests per face to log*n* complexity. Also, the form of the tests is single comparisons in the case of the two dimensional bintree, rather than four comparisons. The expected number of comparisons is therefore proportional to

$$\sum_{l=1}^{Nl} sa(l)\log sa(l)$$

$$\sim \sum_{l=1}^{Nl} \frac{1}{Nl^{\frac{2}{3}}} * \frac{-2}{3} * \log Nl$$

$$\sim \sqrt[3]{Nl} * \log Nl$$

Although it appears that the neighbour links approach may have large space requirements, there is a memory-speed tradeoff. Instead of defining links to occur at all leaves, one can define the links to occur at all interior nodes that only have leaves for children. This decreases the extra space to approximately one eighth of the original space requirements in the octree case, or one half in the case of a bintree. This method incurs the same traversal cost as the original neighbour links plus one additional upward traversal per leaf and possibly some extra downward traversal. Alternatively, the linking is defined to link the set of nodes at a particular height above the leaves. For example, links may be stored in all nodes which are a fixed distance n above the deepest leaf in their subtree. The case $n = 1$ corresponds to the above method of storing at all nodes that only have leaves for children. The amount of memory required is proportional to $(\frac{1}{8})^n$ in the case of an octree, yet the extra traversal cost is only proportional to n. A suitable value of n results in an appropriate tradeoff between space and the additional up and down traversals. For practical cases n can be quite small.

3.2.1. Multiprocessor Implementation

The complete neighbour links strategy is well suited to implementation on a multiprocessor system. Each processor can represent leaves, performing the ray-object intersection tests, and alerting the appropriate neighbour if the ray passes through without intersecting an object. Alternatively, a separate set of processors may be used to perform the neighbour computations.

3.2.2. Storage of Voxel Dimensions

Another issue is whether the voxel dimensions are stored in each leaf node. In order to enumerate the next leaf intersected by a ray, the limits of the current leaf voxel are usually required. The storage required for each voxel is six floating point numbers, which may be a significant cost. Glassner, Kaplan, and Fujimoto et al. have similar suggestions for avoiding this cost. The limits of each encountered voxel are computed by maintaining the limits of each interior node encountered, using the splitting planes of the parent to determine the limits of the children. In addition, with spatial median subdivision, the value of the splitting plane does not need to be stored in each interior node, as it may be computed as the midpoint of the limits of the node's box. The computational implications of these schemes are that a stack of voxel limits must be recursively maintained and updated for each movement from parent to child.

Another possible solution that avoids storing voxel limits performs some vertical traversal to compute the voxel for a given leaf. By following the path back through ancestors and examining splitting planes, the limits of a leaf voxel may be reconstructed. In two-thirds of the cases, it takes at most three upward traversals. With conventional traversal algorithms, the additional computation involved in these space-time tradeoffs may be acceptable, because they do not introduce additional traversals through the tree. However, the neighbour links traversal algorithm attempts to speed up leaf enumeration by eliminating traversal of the interior nodes of the tree, so introducing the additional cost of maintaining the voxel limits in the conventional recursive way would defeat the purpose of the links.

One obvious solution is to store the limits in each leaf. The additional space is six reals per leaf, which seems a significant amount. However, with the complete neighbour links strategy, the internal nodes are only required for finding the initial leaf for each ray. This means that the internal nodes can be eliminated and alternative strategies used to find the initial leaf.

For example, for rays that originate outside the scene, the first leaf can be found by using quadtrees on each of the six faces of the root bounding volume. For rays that originate within the scene, the ray can be extended to the edge of the root bounding volume, and leaves enumerated until the leaf containing the origin is found. This operation need only be performed for a small number of points, i. e., the eye point in most ray tracers, if the leaf containing the point of intersection is recorded when tracing a ray. The origin of all rays in a ray tracer is almost invariably either the eye or a point of intersection of a previous ray with the scene, for which the corresponding leaves is recorded from the previous search.

In a bintree representation, the number of internal nodes is equal to the number of leaves minus one. Depending on the implementation, each internal node may require up to two pointers for children, possibly a parent pointer, one entry for each splitting plane and/or splitting coordinate. The extra memory required by storing the voxel limits in leaves are therefore in the order of that saved by the elimination of interior nodes.

If we are using a spatial median strategy with an octree, a storage scheme which capitalizes on the fact that the sizes of the voxels are powers of two is available. One may use the identifier encoding scheme previously mentioned, which stores a node identifier in 32 bits. The limits of the voxel can be implicitly generated from this 32 bit identifier, because the level indicates the size, and the position information can be used to determine one corner of the voxel. The additional storage of this method is four bytes per leaf, unless we are using a hashing scheme, in which case the identifier is already required for the hashing and the additional cost is zero. This strategy can be used with either the conventional traversal algorithm or the neighbour links method.

3.2.3. Storage of Object Lists

Storage of the lists of objects that belong in each leaf have large space requirements. Glassner stores all the object lists in a single array of object indices, where each list ends with a NIL index (Figure 3.8).

Glassner's scheme provides a separate object list for each leaf. A more compact scheme would allow more than one leaf to point to the same object list. In cases where there are many duplicate leaf lists, this scheme would result in significant memory savings. There would be an additional cost during the traversal phase in order to identify duplicate lists. Additional savings may result if lists which are subsets of other lists are identified, and a pointer

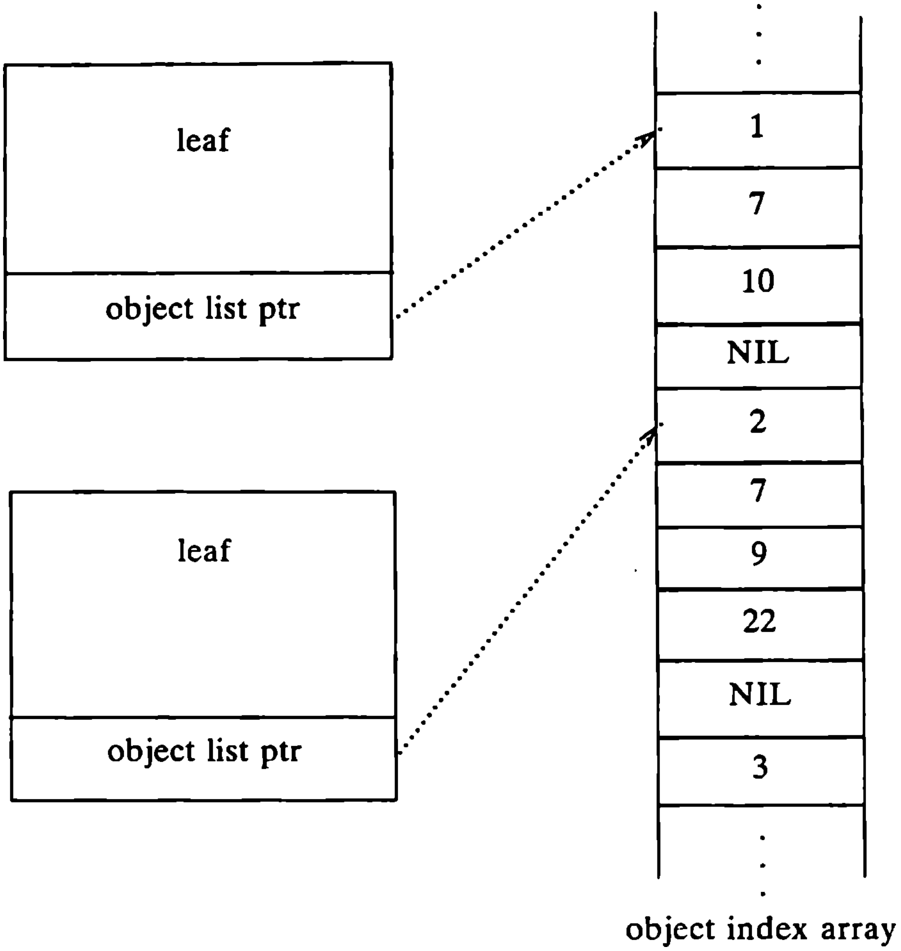


Figure 3.8
Glassner Object List Storage.

to the beginning of a sublist within a larger list used to avoid explicit storage of the sublist. The larger list would have to be organized so that the sublist is at the end.

Another compact scheme is to partition the set of objects into equivalence classes, where each equivalence class is a set of objects which belong in the same set of leaves. In the worse case, each equivalence class consists of one object, in which case this scheme is equivalent to the above many-to-one linking. The object list for a leaf is now a list of equivalence classes, rather than a list of object indices. Figure 3.9 depicts this data structure, which not only allows a many-to-one mapping of leaves to an intermediate level of object list, but also a many-to-one mapping of this intermediate object list to the equivalence classes.

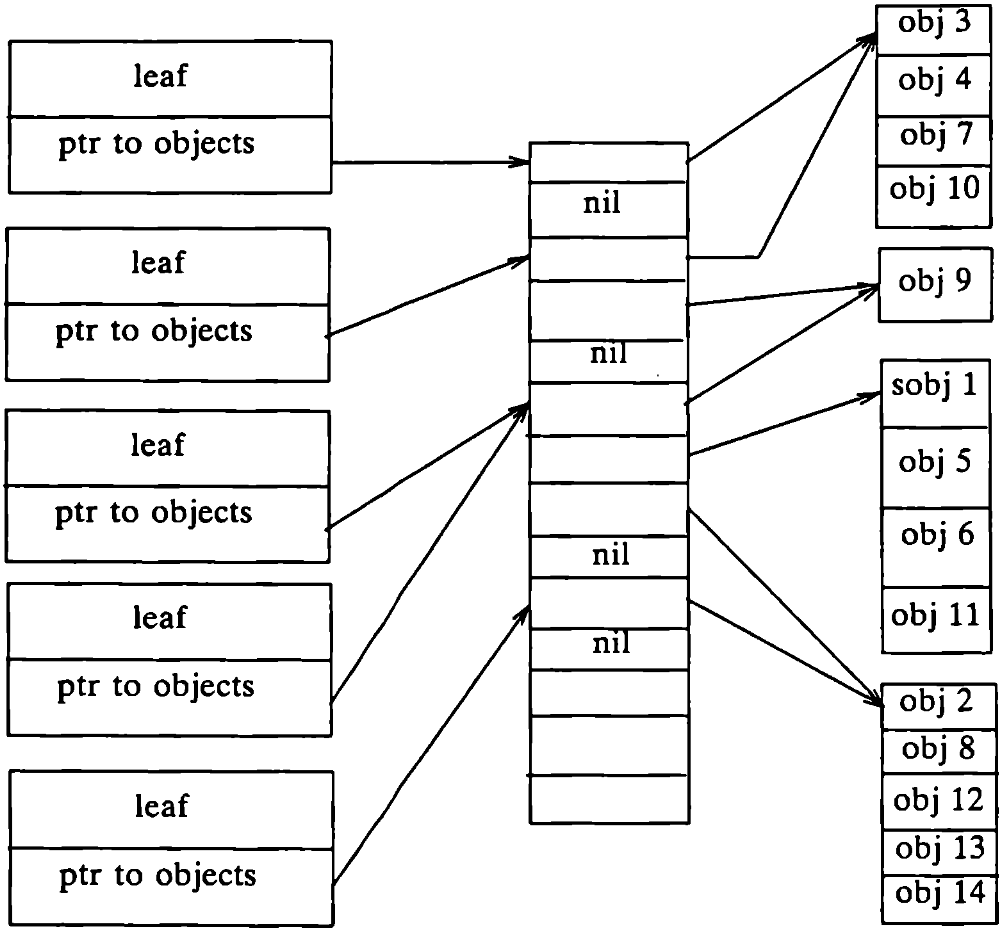


Figure 3.9
Equivalence Class Object List Storage.

3.3. Traversal

Now let us examine some methods of increasing the efficiency of the traversal phase of bintrees and octrees.

3.3.1. Precomputation of Intersection Distances

Traversal of a spatial median subdivided tree can be speeded up by precomputing information for each possible sized voxel in the tree. In a spatial median subdivided octree of height h there are at most h different voxel sizes. This is also true of a bintree where the axis of subdivision cycles through the coordinates as the level increases. For a given ray, we can compute and store the parametric distance along the ray for each of the three pairs of faces for each possible voxel size. The traversal algorithm would then be recursive, examining the intersection of the ray with interior nodes to determine which children to descend. In order to do this, the intersection of the ray with the voxel edges would have to be recursively maintained.

Figure 3.10 depicts an interior node in a bintree, for which we know these intersection points in terms of the parametric value t. Adding the precomputed value for the distance to the splitting plane for this node size to the value of the left edge, the value of t at the intersection with the splitting plane can be determined. Determinating which child to examine or the order of examination can now be accomplished with two comparisons.

3.3.2. Exit Point Computation

One major cost of traversal of a bintree or octree is the computation of the exit point of the ray from the current leaf. A simple coding of this computation was provided in Figure 2.6. As mentioned previously, the speed of this computation can be enhanced by realizing that for a given ray, the ray never exits the box by the lower bound of a coordinate if the direction vector of the ray is positive in that coordinate, and similarly for the upper bound, if the ray direction is negative in that coordinate. This means we need to test the ray against only three of the six sides of a box, which can be determined from examining the direction vector of the ray. Even more savings results if we eliminate the intermediate calculation of the parameter t. With

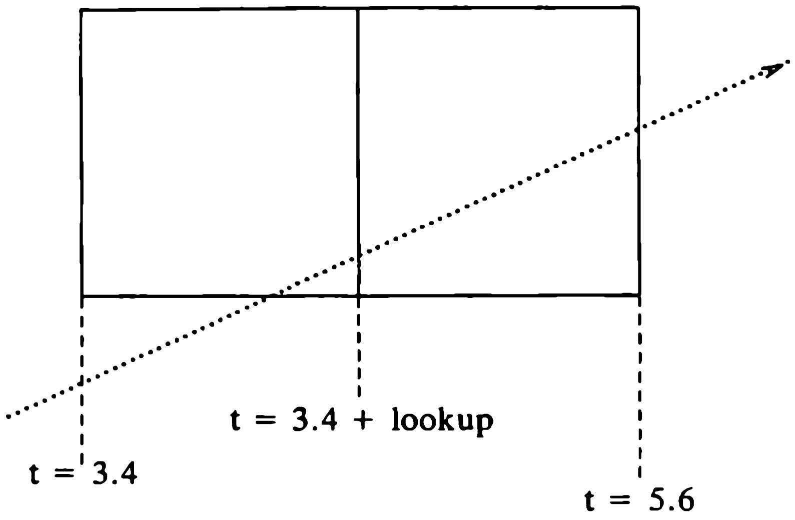


Figure 3.10
Precomputation for Fast Traversal

precomputation, it is possible to compute the value of any coordinate of the ray at a given value of another coordinate with one multiplication and one addition. This means, for instance, that we can compute the y value of the ray where it intersects the x face of the box (the edge perpendicular to the x axis where the ray could possibly exit the box) with one multiplication and one addition. By comparing this value of y with the y limits of the box and repeating this for other coordinates, it can be determined from which face the ray exits. Let us assume that the components of the ray direction are positive and that we have precomputed the values,

$$A_{x,y}, B_{x,y}, A_{x,z}, B_{x,z}, A_{y,x}, B_{y,x}, A_{y,z}, B_{y,z}, A_{z,x}, B_{z,x}, A_{z,y}, B_{z,y}$$

where, for example, the x-value of the intersection of the ray with the plane $y=Y1$ is computed as

$$x = A_{x,y} * Y1 + B_{x,y}$$

The pseudocode for determining which face is exited and the x,y,z value of the point of exit is given in Figure 3.11. Note that this procedure assumes that the coordinates of the ray direction are all positive. Similar code is required for the other cases of ray directions.

```
PROCEDURE RayBoxExitPoint( RayOrigin, RayDirection, Box,
                           VAR Exit : Point;  VAR leavesBy : FaceName );
CONST
    Dimensions = 3;

VAR
    tmin, tmax, tlow, thigh, tenter, texit : REAL;

BEGIN

    $Y_x := A_{y,x} *Box_{High,x} + B_{y,x}$;
    IF $Y_x \leq Box_{High,y}$ THEN          (* does not leave by y face *)
     $X_z := A_{x,z} *Box_{High,z} + B_{x,z}$;
     IF $X_z \geq Box_{High,x}$ THEN   (* leaves by x face *)
            leavesBy := xFace;
            $Z_x := A_{z,x} *Box_{High,x} + B_{z,x}$;
        ELSE                                             (* leaves by z face *)
            leavesBy := zFace;
            $Y_z := A_{y,z} *Box_{High,z} + B_{y,z}$;
        ENDIF;
    ELSE
        $Y_z := A_{y,z} *Box_{High,z} + B_{y,z}$;
        IF $Y_z \leq Box_{High,y}$ THEN       (* leaves by z face *)
            leavesBy := zFace;
            $X_z := A_{x,z} *Box_{High,z} + B_{x,z}$;
        ELSE                                             (* leaves by y face *)
            leavesBy := yFace;
            $X_y := A_{x,y} *Box_{High,y} + B_{x,y}$;
            $Z_y := A_{z,y} *Box_{High,y} + B_{z,y}$;
        ENDIF;
    ENDIF;

END RayBoxExit;
```

Figure 3.11.

This procedure takes on average two comparisons and $3\frac{1}{3}$ multiplications and additions to determine the point of exit of the ray from the box and one of the faces on which this point exists. This procedure does not add an offset to the point to guarantee that it is within the next leaf, because the neighbour links method does not require it. It could be added to the procedure at a cost of two additional comparisons and one addition, which could then be used with Glassner's algorithm to speed it up.

A further improvement to the traversal algorithm is to reduce the number of floating point operations even further. Consider Figure 3.12, which depicts a set of two dimensional leaves and a ray passing through it. The ray is tested against the indicated plane four times, once for each leaf with that plane as a limit. These redundant computations can be reduced by keeping track of flags to indicate whether the limit planes are the same as for the previous leaf, in conjunction with a recursive traversal like that of the ARTS method. One flag per coordinate is required to indicate whether the intersection test of the corresponding bounding plane with the ray should be performed. Each flag should be set after performing the intersection test and reset whenever traversal from one leaf to the next leaf changes the corresponding bounding plane.

The detection of this bounding plane change is simple. Assuming that the ray has positive x direction, the flag for the x coordinate should be reset whenever a downward traversal meets the left subnode of a node that is divided along the x axis, and whenever later upward traversal reaches this node again. This results in at most one ray-plane test per slicing plane, rather than three per leaf. Experimental results indicate that this can reduce the average number of ray-plane tests to two per leaf.

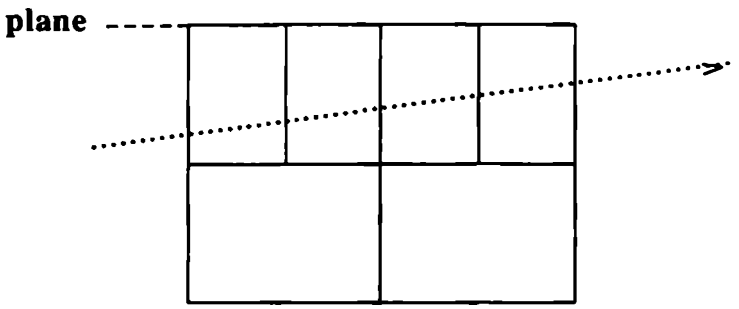


Figure 3.12.

Another method of speeding up the determination of the face exited also requires some precomputation. By inspecting the entrance point of the ray with a voxel, it is sometimes trivial to determine which face is exited. Consider Figure 3.13. If the ray enters within the square region A, then it must exit through the x face. If it enters below the dotted line, then it cannot exit via the y face. Similarly for the dashed line and the z face. The values of these cutoffs depend on the ray direction and the size of the voxels, and must be precomputed for three sides of each different size voxel, for each ray.

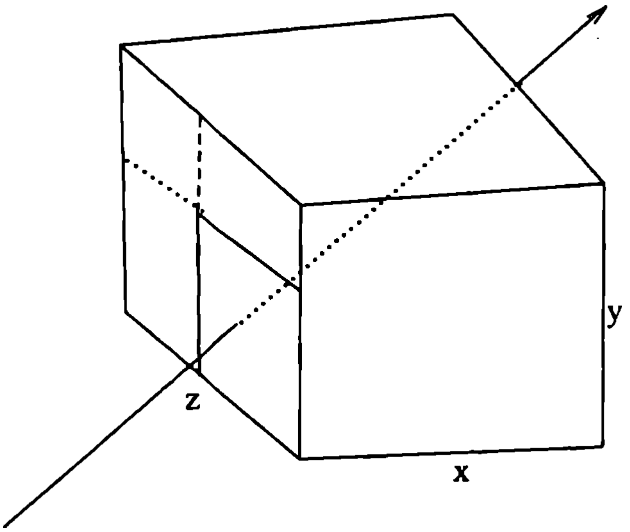


Figure 3.13.

3.3.3. Multi-ray Traversal

A common method of improving the speed of highly repetitive operations such as the traversal of the bintree or octree is to capitalize on coherence, which is the similarity of an operation with previous operations. Ray-to-ray coherence refers to the likelihood that the current ray being traced is similar to the previous ray traced [Spee85]. Taking advantage of such coherence requires us to be able to reuse some of the computation performed for the previous ray so as to save computation time for the current ray. All the space subdivision algorithms described so far do not exploit ray-to-ray coherence in any way. Each ray is dealt with separately, generating a unique tree traversal. It may be advantageous to deal with several rays during one tree traversal, thereby dividing the traversal overhead over several rays.

One way to exploit ray-to-ray coherence is to trace several rays having a common origin in one tree traversal. Consider Figure 3.14, which depicts rays and bintree nodes in two dimensions for simplicity. The rays displayed have the same origin, and intersect the large displayed box. The splitting plane for this box is also shown. If the rays are organized by direction, it is simple to determine the subsets A and B, which are the rays that intersect the left subnode and the right subnode, respectively. For example, the subset A is the set of rays which have slopes that are on the counterclockwise side of point 1, which can be determined by inspecting the vector from the origin to point 1. This process can be applied recursively to the left subnode, then the right subnode, thereby traversing the tree with many rays. As the nodes visited get smaller (deeper in the tree), the number of rays from the original group intersecting the nodes decreases. When this number drops below some threshold value, the rays are traced singly, in the conventional manner, through the small nodes.

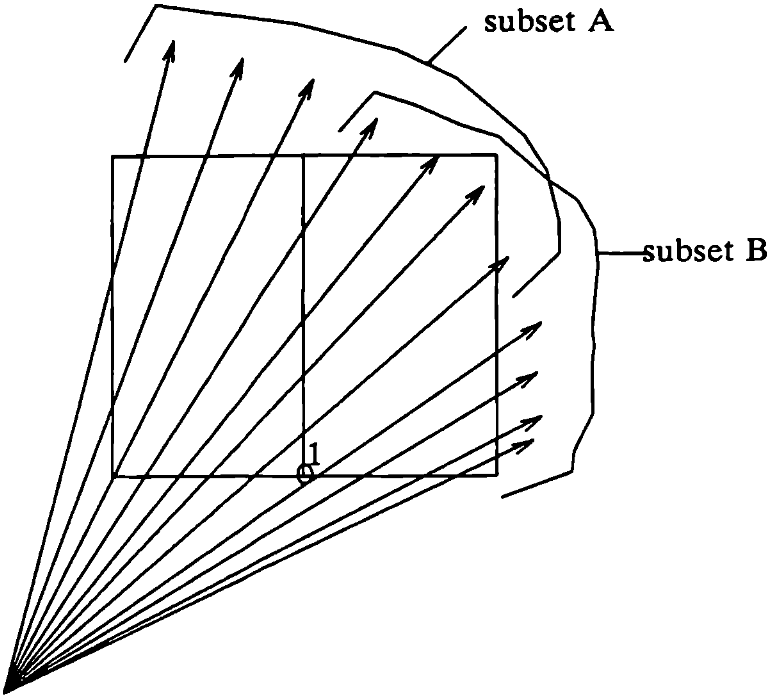


Figure 3.14

This method requires extra space for the rays being traced. However, if a sufficient number of rays are used per tree traversal, the average overhead per ray can be reduced. This method can be applied to all ray tracing algorithms, because even the simplest of ray tracers has of the order of 100,000 rays originating from a single point, the eye. Additionally, distributed ray tracing and other types of antialiasing often trace over 10 rays per pixel.

For this traversal algorithm to be faster than a conventional method, there must be many sets of rays with identical origins; in ray tracing situations this condition is often satisfied. Even more improvement can be achieved by modifying this multi-ray traversal algorithm to work with groups of rays with origins that are not necessarily coincident, but close to each other. This can be accomplished by first finding the bounding volume containing the origins of the set of rays. The rays are still organized with respect to slope, the only difference being the computation of the range of slopes which intersect a given subnode.

In the previous method the slope of the lines from the subnode's corners to the common point of origin are found and used to determine the range of valid slopes. In the revised method, the minimum and maximum slopes of the lines from the subnode's corners to **each** corner of the origin bounding volume is found and used to determine the range of valid slopes. The previous method is equivalent to this revised method where the origin bounding volume happens to be a point. Figure 3.15 depicts the two slopes defining the range of rays which could possibly intersect the left subnode. Note that in the revised algorithm, it is now possible that some rays are within the valid range of slopes, yet do not intersect the box in question. The probability of these false hits increases as the size of the origin bounding volume increases. So for optimum efficiency, rays must be grouped so that there are enough rays to realize a gain during traversal, but the origins must be close enough to constrain the number of false hits to a minimum.

Both of these traversal algorithms effectively provide parallel traversal of rays through the tree at a cost of the order of a single ray traversal. A similar parallel computation can be used to compute ray-bounding volume and ray-polygon intersections, and perhaps ray intersections with other types of objects.

There are other methods of image rendering similar to ray tracing, which effectively trace volumes rather than rays. Amanatides [Aman84] traces cones which approximately cover the area of the individual pixels, thus providing an antialiased image. Heckbert and Hanrahan [Heck84] sweep rectangular beams through the scene to get a resolution-independent tiling of the image. These techniques involve testing the scene for intersection with cones and pyramids, respectively. Algorithms for ray-tracing bintrees and octrees are easily extendible to handling cones or pyramids in place of rays, while still allowing traversal in order of nearness to ray origin. In fact, the multi-ray

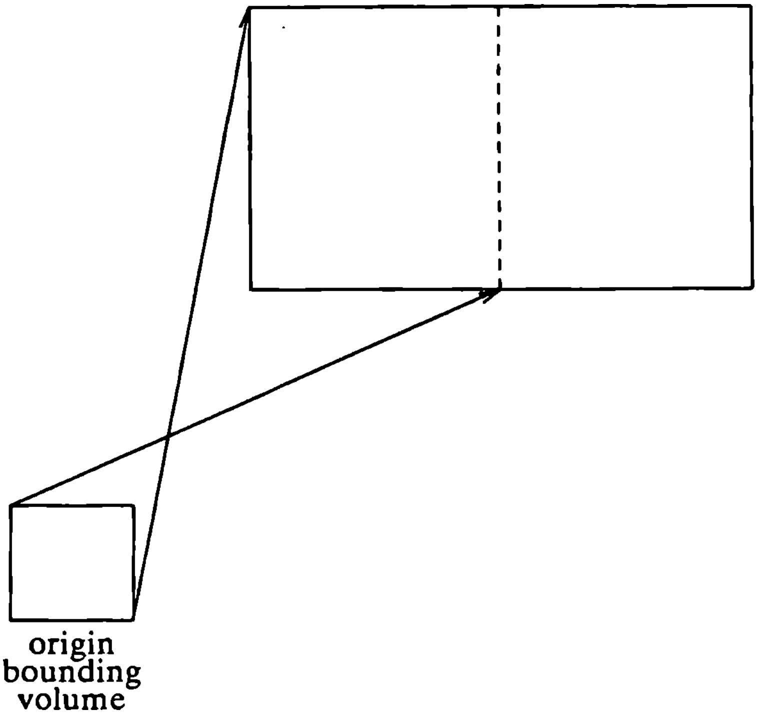


Figure 3.15

traversal algorithm mentioned above is very similar to sweeping a rectangular beam through a scene, which is performed in Heckbert's and Hanrahan's beam tracing.

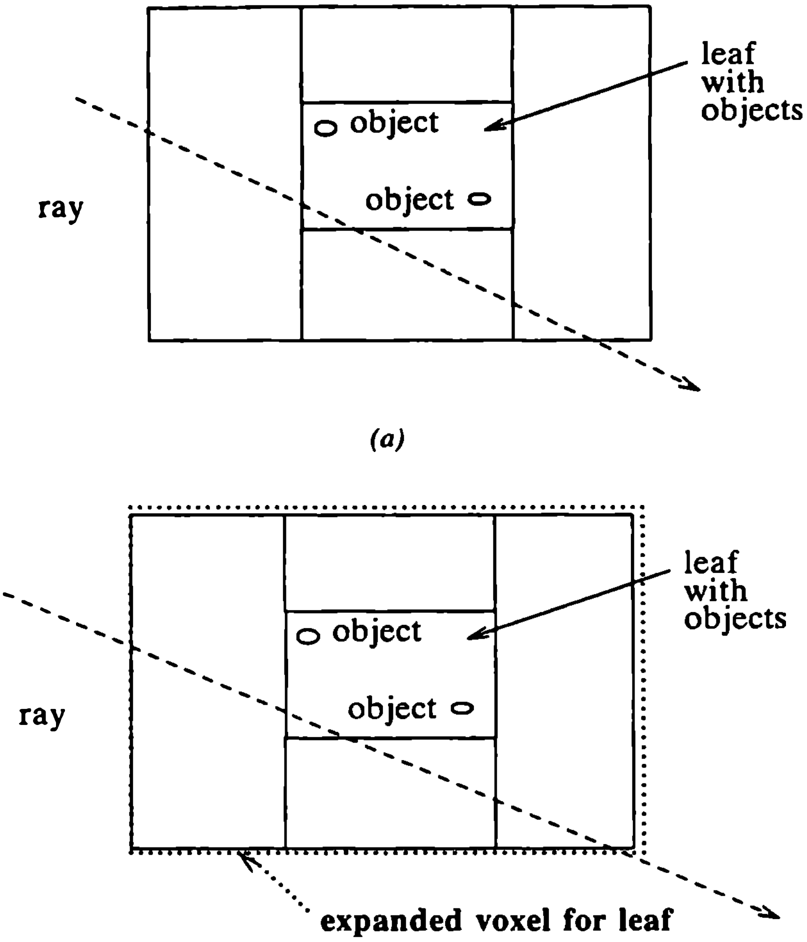
3.3.4. Expanded Leaves

As a ray travels through leaves, it incurs a cost due to the ray exit point computation and the determination of the next leaf. A very simple way of speeding up this propagation of the ray through space is to expand the voxel dimensions of leaves as much as possible. The path of slicing planes which lead to a particular leaf strictly determines the space that the leaf represents. However, if the voxel limits are explicitly stored within the leaf, they do not necessarily have to correspond to the voxel limits implied by the slicing planes. Under certain circumstances, the stored voxel limits can be expanded to be larger than, yet still enclose, the implicit voxel. It will be demonstrated that this can reduce the computation involved in the traversal, by enumerating fewer leaves. As long as the expanded voxel totally encloses the original one, and the extra volume added to the voxel encloses no objects, a traversal algorithm such as that of Fujimoto et al. or Glassner can be used with almost no modification.

As before, a given point maps, via the slicing planes, into one leaf. The advantage of this approach arises from the fact that computing the ray exit point from an expanded voxel may possibly give a point within a leaf that is a few leaves farther along the ray, so it may take just one ray exit point computation and one traversal to the next leaf where previously it would have taken several. Figure 3.16 illustrates this graphically, where Figure 3.16a represents the standard voxel dimensions in a very simple scene subdivision. Figure 3.16b represents the same subdivision, but the voxel stored in the leaf node containing the objects is expanded as shown. The implication in Figure 3.16b is that if the ray intersects the original voxel and does not intersect the objects, then the exit point of the expanded voxel is computed, traversing the scene with two less leaf visits.

There is more than one possible expansion for a given leaf. The best one should be selected, according to some criteria, such as largest surface area of the expanded leaf, or largest sum of surface areas of leaves enclosed by the added volume. The expanded leaf idea may be applied to empty leaves also, assuming that empty leaves are explicitly stored. The notion of expanded leaves may be used not only with the conventional algorithms, but also with the neighbour links method, and any traversal algorithm which uses an exit point computation.

Figure 3.17 depicts a part of the scene containing one object which belongs in many leaves. With traditional methods the gain in tightly enclosing the object (fewer intersection tests) is offset somewhat by the additional leaves required (more leaves to traverse). However, by expanding the leaves



(a)

(b)

Figure 3.16

containing the object to the leaf shown, the traversal cost is reduced significantly, while retaining the tight enclosure around the object.

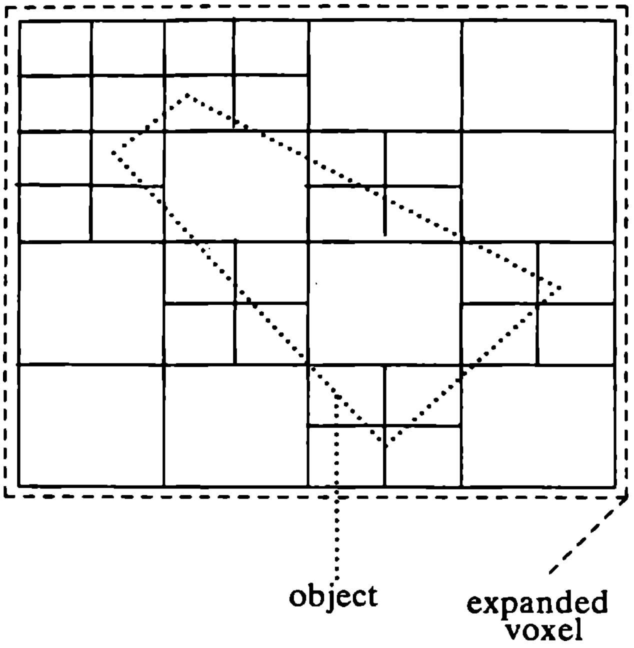


Figure 3.17

Chapter 4
Implementation

4.1. Surface Area Metric Verification

As a preliminary implementation, the validity of the predictions of the surface area metrics given in the previous chapter were tested. A set of 100 boxes with random sizes and positions were created, where each box was a standard rectangular parallelpiped. 100,000 random rays were traced through the bounding volume enclosing the boxes. These rays had origins outside the bounding volume, and were directed towards the bounding volume. The statistics recorded are presented in graphical form in Figure 4.1, where each point represents the surface area of a box and the number of rays which intersected the box. The number of rays intersecting a box is thus shown to be directly proportional to its surface area,

number of rays intersecting box = surface area * 27.5,

std. dev. = 5.2 %   correlation coef. = .995

This graph illustrates that the number of rays intersecting a box is proportional to its surface area, assuming random rays. However, this does not prove that the estimates of interior and leaf nodes intersected are correct, because the search is truncated as soon as an intersection is found. The number of object tests also cannot be assumed to be proven, because the estimate is derived from an approximation of a possibly concave set of leaves by a convex volume. To test the validity of these estimates, a further simulation was performed. Random scenes of objects, and random bintrees were created, and used to trace random rays as in the previous simulation. The estimated numbers of interior nodes, leaves, and objects visited were

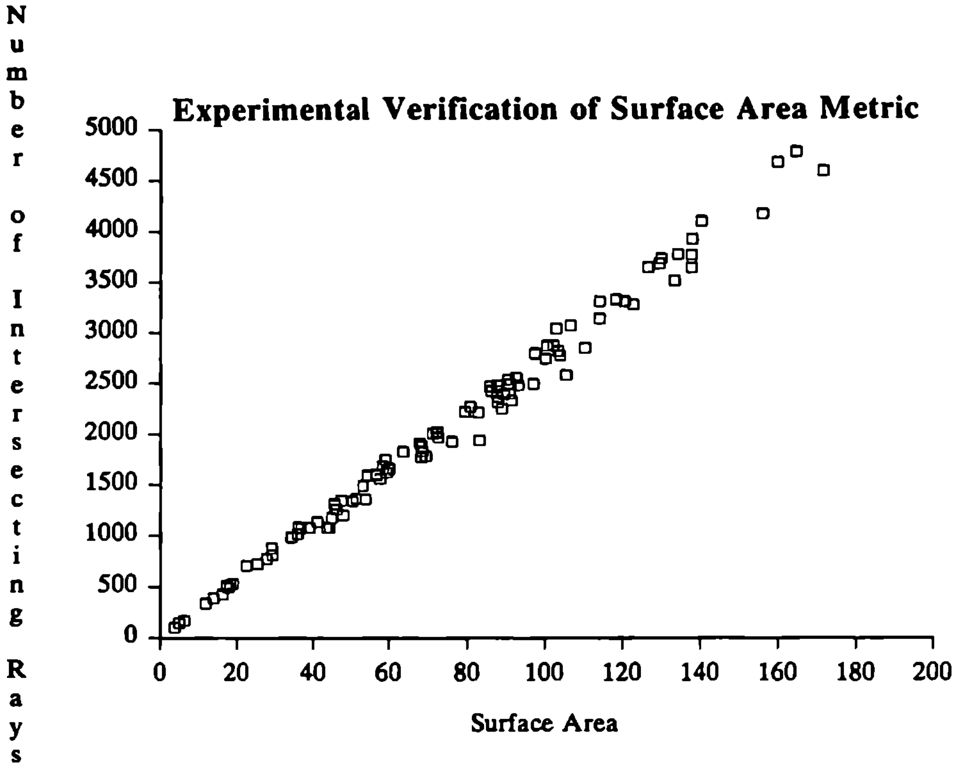


Figure 4.1

compared with the actual numbers from the ray tracing. Each scene contained a random number of objects between 10 and 500, with random distribution in size (described in a later section) from .01 to 1. The bintree created for the scene contained a random number of nodes between 10 and 1000, where nodes were subdivided in random order along a random axis at a random position within the corresponding voxels. 529 random scenes were created and 10000 rays were traced for each scene. Figures 4.2, 4.3, and 4.4 graphically depict the actual statistics versus the estimates, indicating the accuracy of the estimates. The following are the resulting equations:

number of interior nodes intersected: actual = estimated *.752

std. dev. = 12.7 % correlation coef. = .945

number of leaves intersected: actual = estimated *.831

std. dev. = 14.1 % correlation coef. = .900

number of object tests: actual = estimated *1.03

std. dev. = 9.5 % correlation coef. = .985

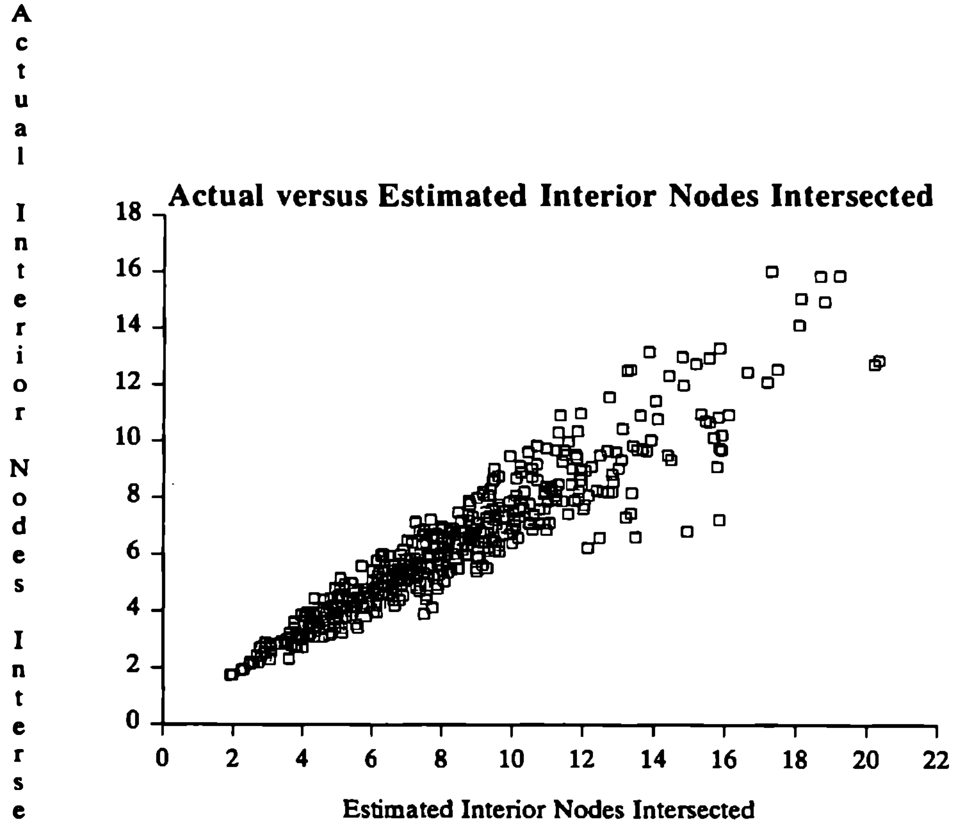


Figure 4.2

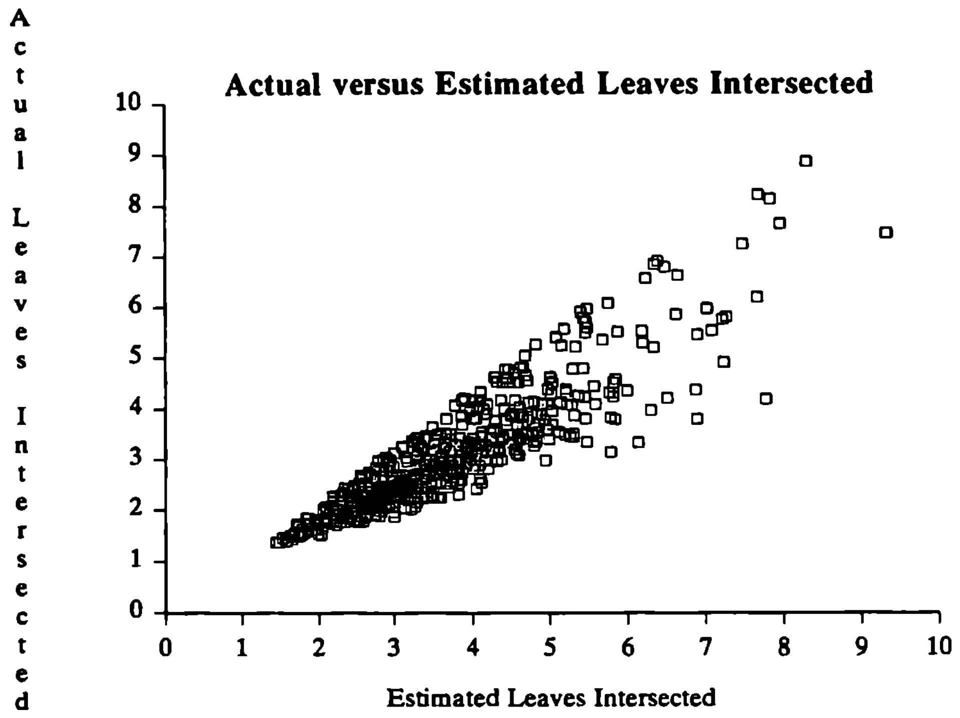


Figure 4.3

The graphs show that in all cases the actual number is proportional to the estimated number. In the case of the number of interior nodes and leaves intersected, the estimate actually provides an upper bound rather than an average case estimate. This is understandable, as the derivation of the estimate assumes that the rays hit no objects. The constants of proportionality may therefore be used in conjunction with the surface area metrics to give a more accurate estimate of the average number of interior nodes and leaves intersected. The estimate of the number of objects intersected was shown to be quite accurate, with a constant of proportionality close to 1. One reason that this estimate provided an average case estimate, rather than an upper bound, is that there are too few objects in the scene. Truncating the search as soon as an intersection was found probably did not save many intersection tests because each ray may have intersected only zero or one objects. Therefore the estimate provided an average-case estimate. With denser scenes, the

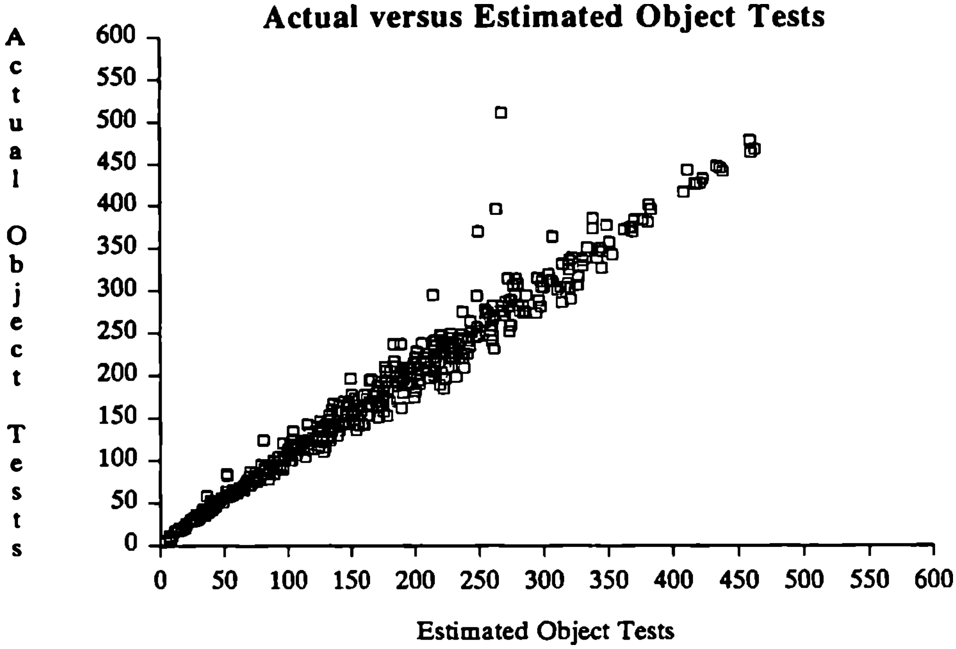


Figure 4.4

object intersection estimate should probably be scaled down in the same way as the interior and leaf node estimates.

4.2. Construction Algorithms

Having verified the surface area metric as reasonably accurate, construction of space subdivision trees was investigated. Four new construction algorithms, as well as Kaplan's algorithm, were implemented for purposes of comparison and evaluation. All algorithms were implemented on bintrees for simplicity and generality, but can easily be extended to octrees. The construction algorithms consist of two algorithms where the spatial median is chosen as the slicing plane, two algorithms where the slicing plane can be in an arbitrary position, and Kaplan's algorithm as a standard of comparison. These algorithms are:

K Kaplan's Algorithm — (zero degrees of freedom in the slicing plane selection) — this is simply Kaplan's algorithm with a threshold value of one. Nodes are subdivided until they contain zero or one objects, in a breadth-first order. The maximum height of the tree was set to 30, which was felt to be large enough not to restrict the growth, yet provide a practical bound. This is roughly equivalent to the value of 10 used by Kingdon [King86] as the maximum height of his octrees, because three levels of a bintree correspond to one level of an octree.

AA Arbitrary Acyclic — (two degrees of freedom) — slicing planes can be anywhere within the node, and a node may be divided along any of the three axes. The optimal slicing plane is determined by sampling at nine equally spaced intervals within the node, recording the maximum value of the function given in the previous chapter. Nine is an arbitrary parameter chosen so as to attempt to focus on the optimal plane, yet not incur unreasonable amounts of computation. A node is subdivided along whichever axis provides the greatest gain, and nodes are subdivided according to highest gain.

AC Arbitrary Cyclic — (one degree of freedom) — same as Arbitrary Acyclic, except that the first level of subdivision always occurs along the x axis, the second along the y axis, the third along the z axis, cycling through the three axes.

SA Spatial Median Acyclic — (one degree of freedom) — same as Arbitrary Acyclic, except that the spatial median is always chosen as slicing plane.

SC Spatial Median Cyclic — (zero degrees of freedom) — same as Arbitrary Cyclic, except that the spatial median is always chosen as slicing plane.

These algorithms were encoded as simply as possible without any attempts to optimize the code. It was felt that it was more important that the code be correct, and our emphasis was verification, rather than efficiency.

Statistics on the trees were recorded at every 50 nodes during the construction of the tree. The statistics include the number of interior nodes, the number of empty leaves, the number of non-empty leaves (containing one or more objects), the estimated number of leaves visited, estimated number of interior nodes visited, and the estimated number of objects tested for intersection.

4.3. Scenes

The ultimate goal of the strategies for building the space subdivision structures is to improve performance in actual ray tracing systems. The performance should therefore be evaluated with scenes that represent a reasonable sample of all scenes subjected to ray tracing. Kingdon [King86] uses a set of seven general scene types for a similar data structure evaluation. These scene types have associated parameters to control the number of objects and distribution of object sizes and positions. One can vary the parameters in an attempt to represent the range of typical object distributions within average scenes. For the purposes of comparison and evaluation of bintree construction algorithms, Kingdon's scene types are used, as well as similar selections for the scene parameters. In order to reduce the amount of computation for these tests, only five of Kingdon's seven scene types were used. The two types not used were less general, hierarchical object scenes.

The object distributions are based on three simple random number generators: U^3, which selects a random point within a unit sphere; U^0, which selects a random point on the unit sphere; and U^e, which returns the output of U^0 scaled by a Gaussian distributed random number with a mean of 0 and variance of 1. The five scene types are:

small spherical

a set of triangles whose first vertices are U^3 distributed in space and whose other two vertices are $.01*U^0$ distributed offsets from the first point,

large spherical

a set of triangles whose first vertices are U^3 distributed in space and whose other two vertices are .333*U^0 distributed offsets from the first point,

small Gaussian

a set of triangles whose first vertices are .333*U^e distributed in space and whose other two vertices are .01*U^0 distributed offsets from the first point,

large Gaussian

a set of triangles whose first vertices are .333*U^e distributed in space and whose other two vertices are .333*U^0 distributed offsets from the first point,

three random vertices

a set of triangles whose vertices are U^3 distributed in space, creating a set of dense, interpenetrating triangles.

The small spherical and small Gaussian scenes contain triangles that are roughly $\frac{1}{200}$ times the width of the scene, while the large spherical and large Gaussian scenes contain triangles approximately $\frac{1}{6}$ times the width of the scene, attempting to simulate the limits of object sizes in typical scenes. The Gaussian distribution provides a cluster of objects while the normal distribution provides more spread out objects. Six instances of each scene were used, varying only in the number of objects comprising the scene. The numbers used were 256, 512, 1024, 2048, 4096, 8192. Kingdon uses these numbers plus two other smaller scene sizes (64 and 128). It was felt that the six numbers used were sufficient to measure performance without requiring an impractical amount of computer resources. The maximum number of nodes was set according to the amount of time and memory required and range from 2000 to 8000 nodes, depending on the scene type. Also, for some scene types, only the first five scene sizes were used, to limit computer usage.

4.4. Neighbour Links

A neighbour links strategy was implemented, using the simple definition of neighbours which gives exactly one neighbour per face, as opposed to the complete neighbour links strategy. One instance of each of the five scene types was used to build an arbitrary acyclic type bintree, with the neighbour links for each leaf computed. All scenes had 1000 objects and the bintrees constructed contained 1000 nodes. After building the bintrees, 10000 random rays were traced and the number of parent-to-child and child-to-parent movements were recorded, for each of the conventional traversal algorithms and the neighbour links method. These numbers indicate the savings in traversal cost by using the neighbour links strategy.

Chapter 5
Results

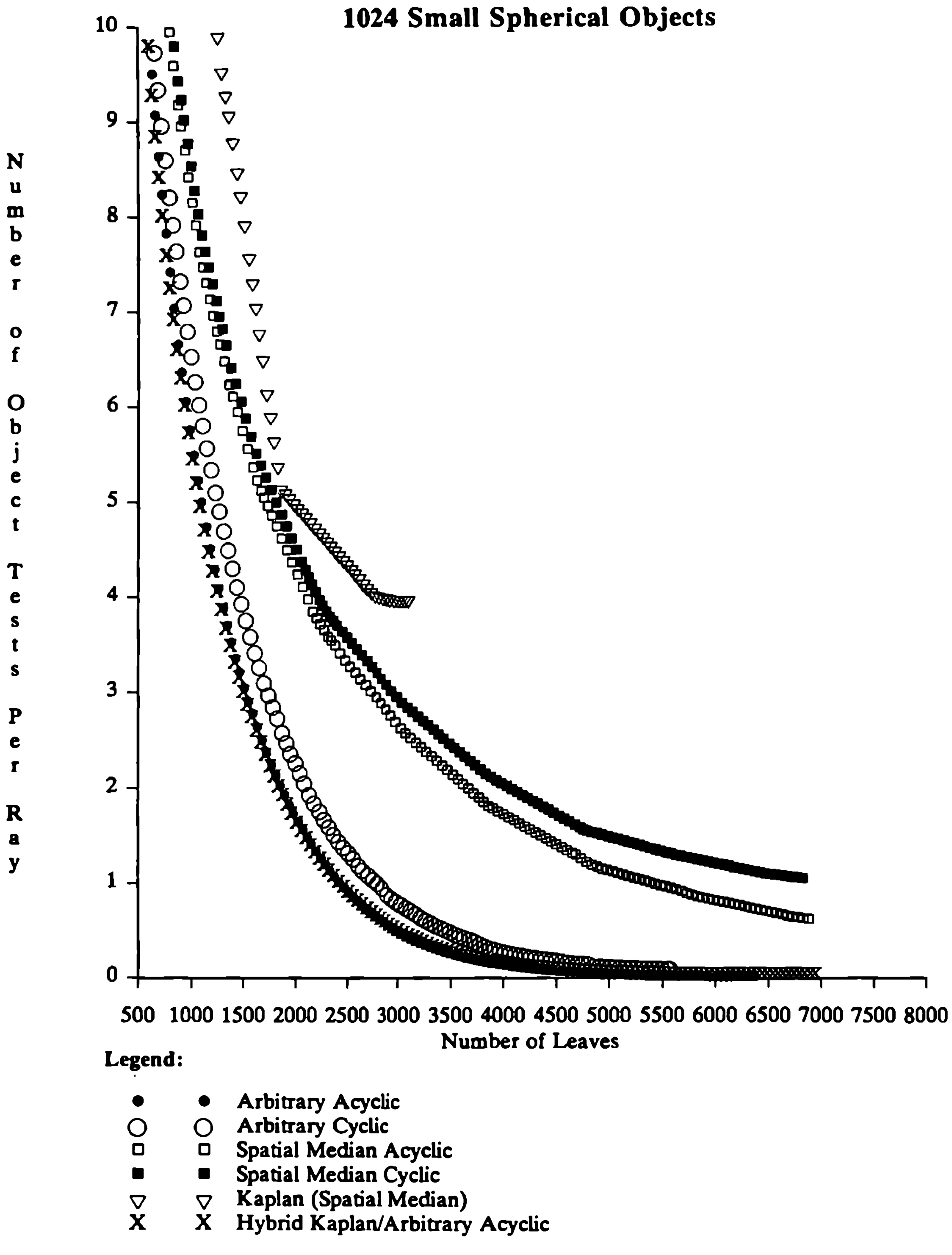
5.1. Construction Algorithms

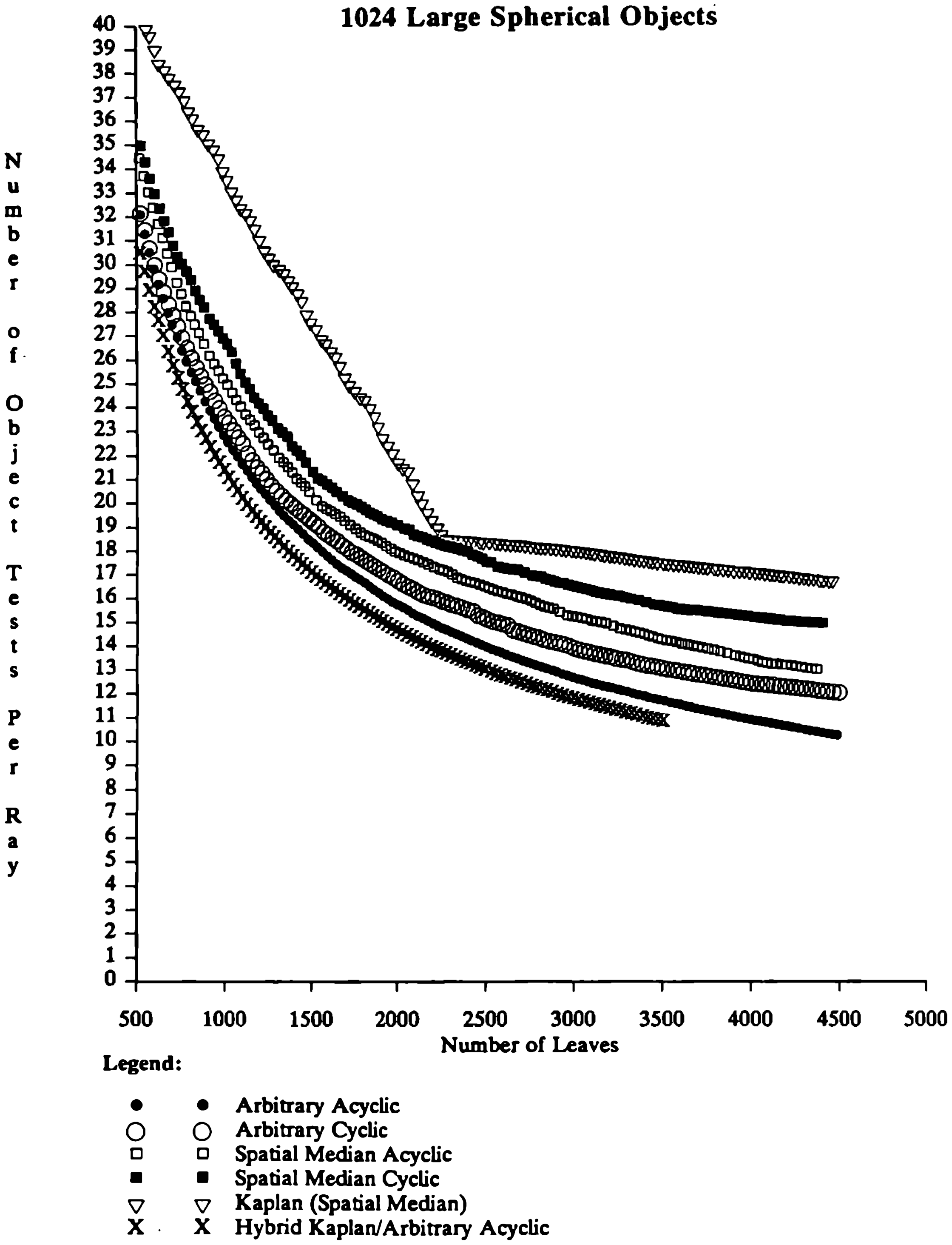
Data from the construction simulations is included in graphical form in the appendix. In summary, the estimated number of nodes and leaves visited for a given scene were very similar over all five algorithms, as is evident from examining the graphs. Overall, the arbitrary acyclic algorithm performed slightly better than the rest, in terms of number of nodes and leaves visited. However, the number of objects intersected varied widely over the different construction algorithms. For this reason and because the object cost is typically higher than the other two costs, let us concentrate on the number of objects intersected in order to evaluate the algorithms' performance.

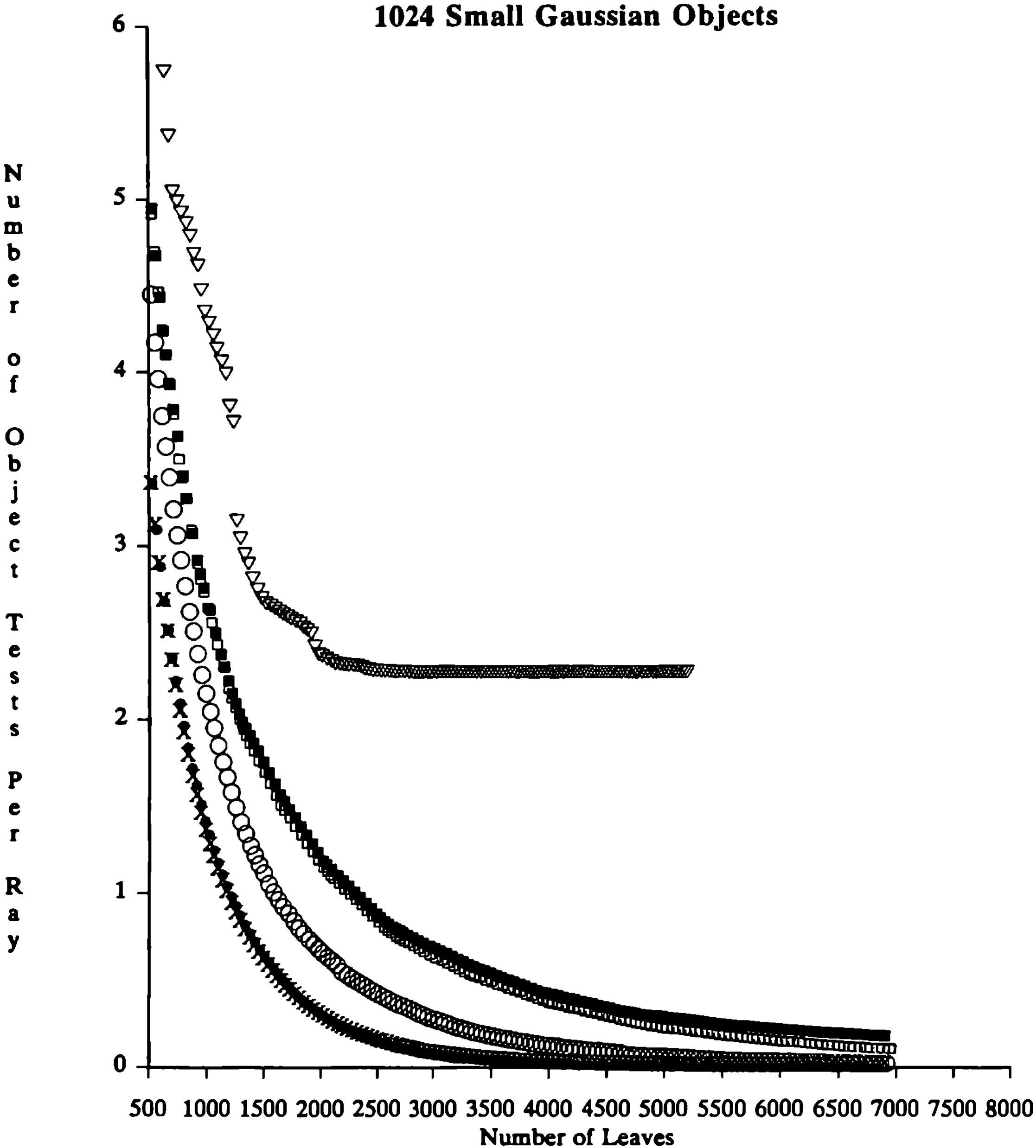
For the small spherical and small Gaussian scene types, the arbitrary acyclic algorithm performed the best, providing up to three orders of magnitude reduction in the number of objects tested for intersection. For the large spherical and large Gaussian scene types, the arbitrary acyclic algorithm was also the best, but only up to one order of magnitude better. However, for the scenes consisting of three random vertices, the Kaplan method performed best. The general rule seems to be that the arbitrary acyclic algorithm performs best for scenes with non-overlapping small objects, while Kaplan's performs best for denser scenes with interconnected objects.

The explanation for this behavior is that the arbitrary acyclic algorithm is a greedy algorithm, governing the subdivision by only looking one step in advance. If subdividing a node is not immediately advantageous, then it is not subdivided, even if subjecting the node to two levels of subdivision would be advantageous. Kaplan's algorithm, by virtue of its breadth-first nature and an inability to evaluate the benefit of subdividing a node, may subdivide a node many times, resulting in a gain where the arbitrary acyclic algorithm would not.

These observations indicate that a hybrid of the arbitrary acyclic and Kaplan's algorithms might provide optimum performance in all scene types. A hybrid implementation was performed where the arbitrary acyclic algorithm was applied to a node first to determine an optimum splitting plane. If it does not find a speed gain above a certain threshold dependent upon the surface area of the node, then the spatial median is chosen. The coordinate is dependent on the level of the node, similar to Kaplan's method except that nodes are only subdivided with one level of subdivision at once (rather than three levels). This forces the algorithm to assume that subdividing a node results in a decrease in cost, even if the one-step lookahead indicates an increase. Thus, a node which the original algorithm does not find advantageous to subdivide may be subdivided by the hybrid algorithm, resulting in a tree with a higher cost than if the node remained a leaf. The children of this node may then be subdivided, possibly resulting in an overall decrease in the cost of the tree. The above process is used, as in the other algorithms, only to determine the splitting plane, splitting coordinate, and estimated gain, if it were to be subdivided. The selection of the next node to subdivide is, as in the arbitrary acyclic algorithm, the node which has the highest estimated gain. However, when the hybrid algorithm resorts to selecting the spatial median, the gain associated with this split is set at the threshold, rather than the actual value, which would be lower. This hybrid algorithm was run on each of the five scene types containing 1024 objects, except for the scene type containing three random vertices, which had only 64 objects for efficiency. As the following graphs indicate, it performs better overall than any of the other algorithms (it was outperformed slightly by the arbitrary acyclic algorithm in the case of a large Gaussian scene). It is interesting to note that the portions of the graphs pertaining to Kaplan's algorithms often contain line segments and abrubt changes of slope. These are due to the fact that after some point in the construction of the tree, Kaplan's algorithm essentially builds the tree level by level. The line segment portions correspond to individual levels, and the abrupt changes in slope correspond to the filling of a level.

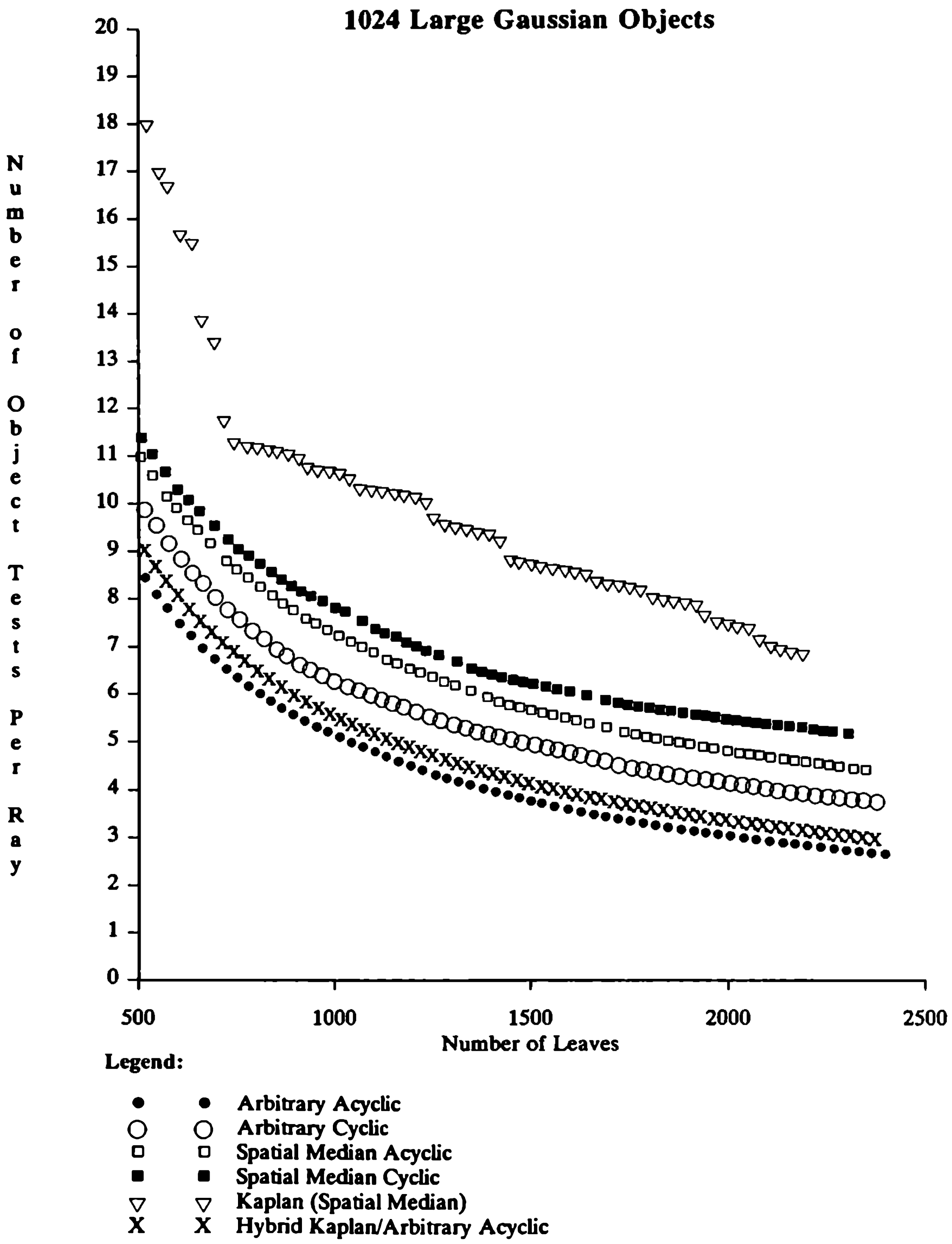

1024 Small Spherical Objects
Number of Object Tests Per Ray
0
1
2
3
4
5
6
7
8
9
10
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)
X X Hybrid Kaplan/Arbitrary Acyclic

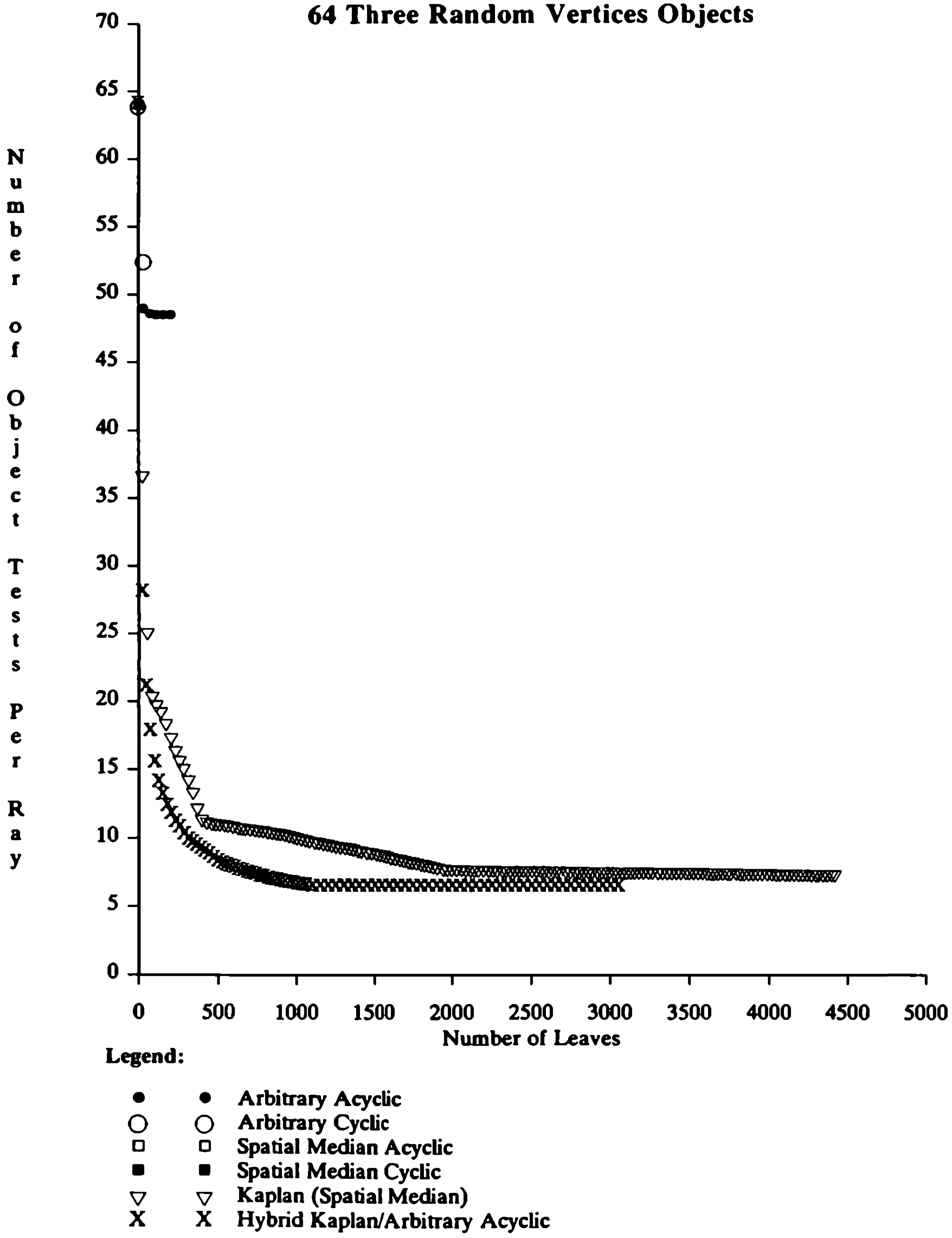

1024 Large Spherical Objects
Number of Object Tests Per Ray
40
39
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)
X X Hybrid Kaplan/Arbitrary Acyclic



Legend:

● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)
X X Hybrid Kaplan/Arbitrary Acyclic


1024 Large Gaussian Objects
Number of Object Tests Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)
X X Hybrid Kaplan/Arbitrary Acyclic



64 Three Random Vertices Objects

At the end of each simulation, the total number of object instances (number of objects stored at the leaves) were recorded. The arbitrary algorithms produced near optimum numbers, that is, only 10 or 20 percent more object instances than objects, while Kaplan's and the other two spatial median algorithms produced trees with up to ten times as many object instances as objects. The reason for this is the implicit motivation to keep objects in as few leaves as possible, provided by the cost function used in selecting the splitting plane for arbitrary subdivision.

5.2. Neighbour Links

The following table summarizes the number of parent-to-child and child-to-parent traversals recorded from the neighbour link simulation.

Up / Down Traversals, 1000 nodes			
Scene Type	Up	Down	Neighbours Down
1000 Small Spherical	36.35589981	36.38169861	9.951199532
1000 Large Spherical	15.85369968	20.09070015	8.987500191
1000 Small Gaussian	33.94269943	33.94810104	10.09840012
1000 Large Gaussian	24.50469971	25.91119957	8.954000473
64 3-Random Verts	15.17660046	19.25169945	9.043399811

Table 5.1.

If it is assumed that the cost of a single upward traversal is equivalent to a single downward traversal, then these numbers show that the neighbour link scheme decreases the traversal cost to between $\frac{1}{7}$ and $\frac{1}{4}$ of the cost of an ARTS-type traversal method.

5.3. Summary

It has been demonstrated that the cost of ray tracing space subdivision trees can be represented by the number of interior nodes, leaves, and objects visited per ray, and the respective costs of these visits. This thesis reports new construction algorithms which represent considerable improvement over conventional methods, in terms of reducing the number of nodes, leaves, and objects visited by a ray. Similar improvements may be provided by using arbitrarily oriented slicing planes (not just major planes) to add more freedom to the subdivision process. The efficiency of traversal has been improved by attacking its two main costs, the processing of interior nodes in bintrees and octrees and the computation of the ray exit point. The neighbour link strategy introduced in this thesis significantly reduces the number of interior nodes visited, while a new coding of the ray exit point computation that is arithmetically less complex is provided. The efficiency of traversal may also be improved by the multiple ray traversal scheme. Finally, the number of leaves traversed is decreased further by allowing expanded leaves, which can be employed in conjunction with any of the traversal schemes.

5.4. Suggested Further Work

Some of the ideas in this thesis have been implemented only to a limited extent, while others have not been implemented at all. Further implementation is definitely required in order to evaluate and refine the suggestions. As stated earlier there are some issues not dealt with in this thesis. The ideas herein should be examined with respect to other areas such as higher-dimensional data structures and/or dynamic data structures, and multi-processor algorithms.

The construction algorithms advanced in this thesis, while providing improvements, are very primitive. The basic problem with them is that they choose the optimal slicing plane for each stage of subdivision, which is not necessarily the optimal overall subdivision. In effect, the algorithms choose local maxima, without using global information. Research should be applied to improving the subdivision process by making the process less localized. A simple way is to perform lookahead of several levels of subdivision per node, as opposed to one. Other methods should also be investigated.

The new construction algorithms, as well as the conventional Kaplan algorithm, perform badly as the objects become more densely distributed and interpenetrating. Methods of selecting splitting planes for such situations should be investigated. A lookahead of several levels of subdivision and varying the different slicing planes involved would, perhaps, be applicable to these types of situations, although probably computationally expensive.

Since the concept of a hierarchical extents tree is simply the dual of space subdivision, many of the ideas within this thesis can be applied to HE trees. The multiple ray traversal scheme can be applied easily to HE trees. In fact, tracing multiple rays through an HE tree would be less complex than for space subdivision trees. There are also many possibilities for defining construction algorithms based on the surface area metric. One heuristic for selecting two bounding volumes to partition a set of objects would be to first find the two objects of the set which imply a bounding volume with the largest surface area. These two objects should not, therefore, belong to the same bounding volume. They can be used as the starting point for the algorithm. One by one, each of the remaining objects are added to one of the two sets. The choice depends on which one results in the minimum sum of the surface areas of the two bounding volumes surrounding the two sets. This can be extended to any number of partitions. A partitioning algorithm of this sort may be a useful basis for a construction algorithm for HE trees. In general, the surface area metric is useful for constructing any sort of structure that is to be used for ray tracing or for answering queries involving object-object intersection tests.

Although this thesis has concentrated heavily on using octrees and bintrees for ray tracing, the space subdivision trees described may be useful for other image rendering algorithms. As mentioned in Chapter 3, octrees and bintrees may be used for cone tracing and beam tracing rendering algorithms. As general structures for organizing multidimensional data, space subdivision trees may be applied to many problems which involve searching in multidimensional space.

References

[Ahuj84] Ahuja, N. and Nash, C., Octree Representations of Moving Objects. *Computer Vision, Graphics, and Image Processing*, 26, 1984, pp. 207-216.

[Anam84] Anamatides, J., Ray Tracing with Cones, *Computer Graphics*, 18(3), July, 1984, pp. 129-135.

[Carl85] Carlbom, I., and Chakravarty, I., A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects, *IEEE Computer Graphics and Applications*, 5(4), April, 1985, pp. 24-31..

[Cook84] Cook, R. L., Porter, T., and Carpenter, L., Distributed Ray Tracing, *Proceedings of SIGGRAPH '84, July, 1984, pp. 137-145.*

[Dipp84] Dippé, M.A.Z., and Swensen, J., An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis, *Proceedings of SIGGRAPH '84, July, 1984, pp. 149-158.*

[Dipp85] Dippé, M.A.Z., and Wold, E.H., Antialiasing Through Stochastic Sampling, *Computer Graphics*, 19(3), July, 1985, pp. 69-78.

[Doct81] Doctor, L., and Torborg, J., Display Techniques for Octree-Encoded Objects, *IEEE Computer Graphics and Applications*, 1(3) July, 1981, pp. 29-38.

[Dyer82] Dyer, C.R., The Space Efficiency of Quadtrees, *Computer Graphics and Image Processing*, 19(4), August, 1982, p. 335-348.

[Dube86] Dubetz, M.H., *Ray Tracing Algorithms for Computer Graphics*, Ph.D. Thesis, University of Alberta, 1985.

[Fiel85] Field, D.E., *Fast Hit Detection for Disjoint Rectangles*, University of Waterloo Research Report, December, 1985.

[Four82] Fournier, A., Fussel, D., and Carpenter, L., Computer Rendering of Stochastic Models, *Communications of the ACM*, 25(6), June, 1982, pp. 371-384.

[Fuch80] Fuchs, H., Kedem, Z.M., and Naylor, B.F., On Visible Surface Generation By A Priori Tree Structures, *Proceedings of SIGGRAPH '80*, July 14-18, 1980.

[Fuji86] Fujimoto, A., Tanaka, T., and Iwata, K., ARTS: Accelerated Ray-Tracing System, *IEEE Computer Graphics and Applications*, 4(10), October, 1984, pp. 15-22.

[Glas84] Glassner, A.S., Space Subdivision for Fast Ray Tracing, *IEEE Computer Graphics and Applications*, 4(10), October, 1984, pp. 15-22.

[Glas87a] Glassner, A.S., An Overview of Ray Tracing, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, pp. 1-20.

[Glas87b] Glassner, A.S., Surface Physics for Ray Tracing, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, pp. 1-26.

[Glas87c] Glassner, A.S., Spacetime Ray Tracing for Animation, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, pp. 1-17.

[Glas88] Glassner, A.S., Spacetime Ray Tracing for Animation, *IEEE Computer Graphics and Applications*, March, 1988, pp. 60-70.

[Gold87] Goldsmith, J. and Salmon, J., Automatic Creation of Object Hierarchies for Ray Tracing, *IEEE Computer Graphics and Applications*, May, 1987, pp. 14-20.

[Hain87] Haines, E., Essential Ray Tracing Algorithms, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, 1-41.

[Hall83] Hall, R. A., and Greenberg, D. P., A Testbed for Realistic Image Synthesis, *IEEE Computer Graphics and Applications*, 3(8), November, 1983, pp. 10-20.

[Hanr87] Hanrahan, P., A Survey of Ray-Surface Intersection Algorithms, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, pp. 1-29.

[Heck82] Heckbert, P.S., Color Image Quantization for Frame Buffer Display, *Proceedings of SIGGRAPH '82*, July 26-30, 1982, pp. 297-307.

[Heck84] Heckbert, P.S., and Hanrahan, P., Beam Tracing Polygonal Objects, *Computer Graphics*, 18(3), July, 1984, pp. 119-127.

[Jack80] Jackins, C. L., and Tanimoto, S. L., Oct-Trees and Their Use in Representing Three-Dimensional Objects, *Computer Graphics and Image Processing*, 14, 1980, pp. 249-270.

[Kapl85] Kaplan, M. R., The Uses of Spatial Coherence in Ray Tracing, *SIGGRAPH '85 Course Notes 11*, July 22-26, 1985.

[Kaji83] Kajiya, J. T., New Techniques for Ray Tracing Procedurally Defined Objects, *Computer Graphics*, 17(3), July, 1983, pp. 91-102.

[Karl84] Karlsson, R. G., *Algorithms in a Restricted Universe*, University of Waterloo Research Report, November, 1984.

[Kay86] Kay, T. L., and Kajiya, J. T., Ray Tracing Complex Scenes, *Computer Graphics*, 20(4), August, 1986, pp. 269-277.

[King86] Kingdon, S. J., *Speeding Up Ray-Scene Intersections*, Master Thesis, University of Waterloo, 1986.

[Lee85] Lee, M. E., Redner, R. A., and Uselton, S. P., Statistically Optimized Sampling for Distributed Ray Tracing, *Computer Graphics*, 19(3), July, 1985, pp. 61-65.

[MacD86] MacDonald, J. D., *Quadtrees in Computer Graphics*, Bachelor Thesis, St. Francis Xavier University, March, 1986.

[Nemo86] Nemoto, K., and Takao, O., An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing, *Proceedings of Graphics Interface '86*, 1986, pp. 43-48.

[Rubi80] Rubin, S. M., and Whitted, T., A Three-Dimensional Representation for Fast Rendering of Complex Scenes, *Computer Graphics*, 14(3), July, 1980, pp. 110-116.

[Same84] Samet, H., The Quadtree and Related Hierarchical Data Structures, *Computing Surveys*, 16(2), June, 1984, pp. 187-260.

[Spee85] Speer, L. R., DeRose, T. D., and Barsky, B. A., A Theoretical and Empirical Analysis of Coherent Raytracing, *Proceedings of Graphics Interface '85*, Montreal, Quebec, 1985, pp. 1-8.

[Spee87] Speer, L.R., A Survey of Algorithms for Fast Raytracing, *SIGGRAPH 1987 Introduction to Ray Tracing Course Notes*, July, 1987, 1-18.

[Ston75] Stone, L., *Theory of Optimal Search*, Academic Press, New York, 1975, pp. 27-28.

[Swee86] Sweeney, M.A.J., and Bartels, R.H., Ray-Tracing Free-Form B-Spline Surfaces, *IEEE Computer Graphics and Applications*, 6(2), February, 1986, pp. 41-49.

[Wegh84] Weghorst, H., Hooper, G., and Greenberg, D. P., Improved Computational Methods for Ray Tracing, *ACM Transactions on Graphics*, 3(1), January, 1984, pp. 52-69.

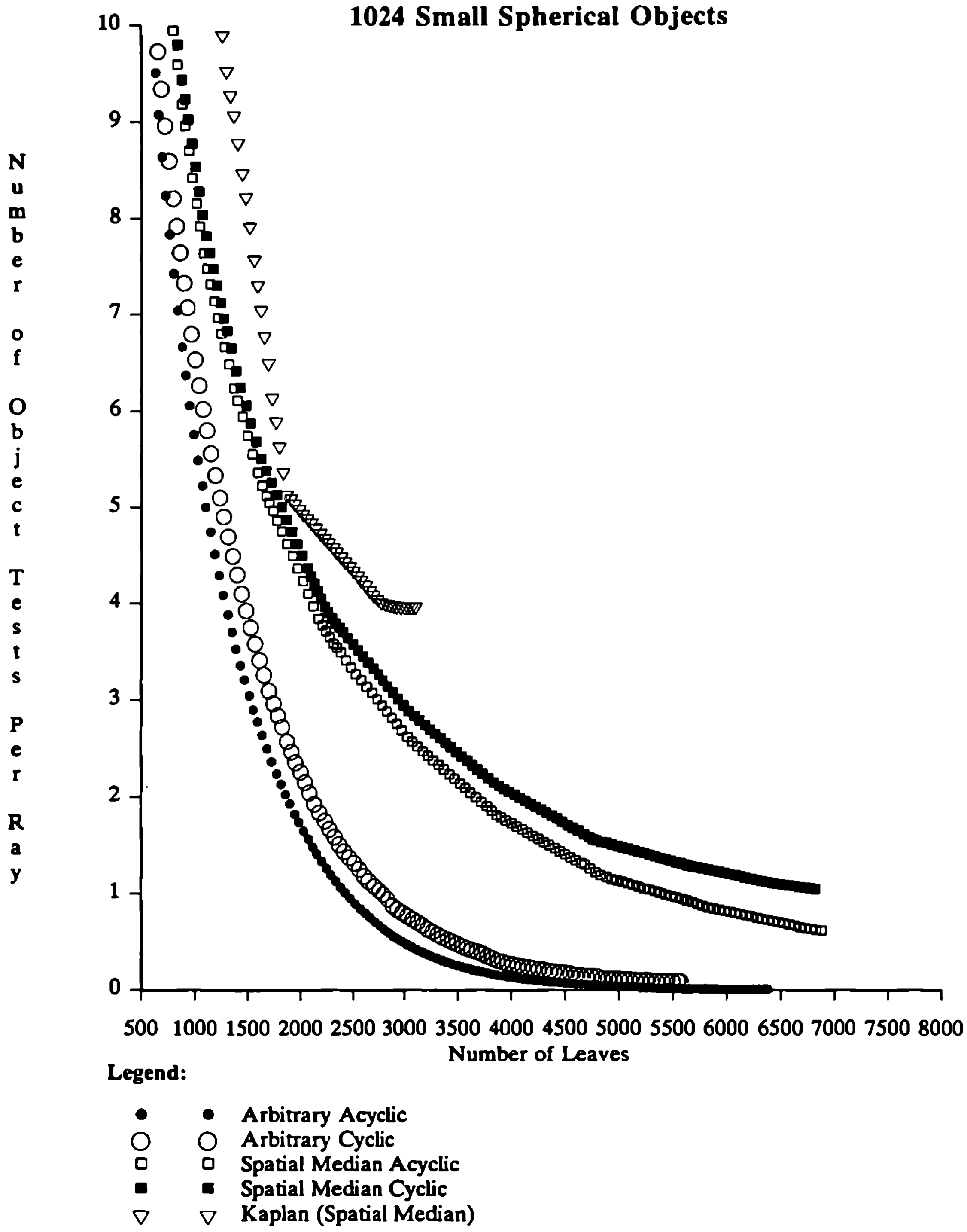
[Wyvi86] Wyvill, G., Kunii, T.L., and Shirai, Y., Space Division for Ray Tracing in CSG, *IEEE Computer Graphics and Applications*, 6(4), April, 1986, pp. 28-34.

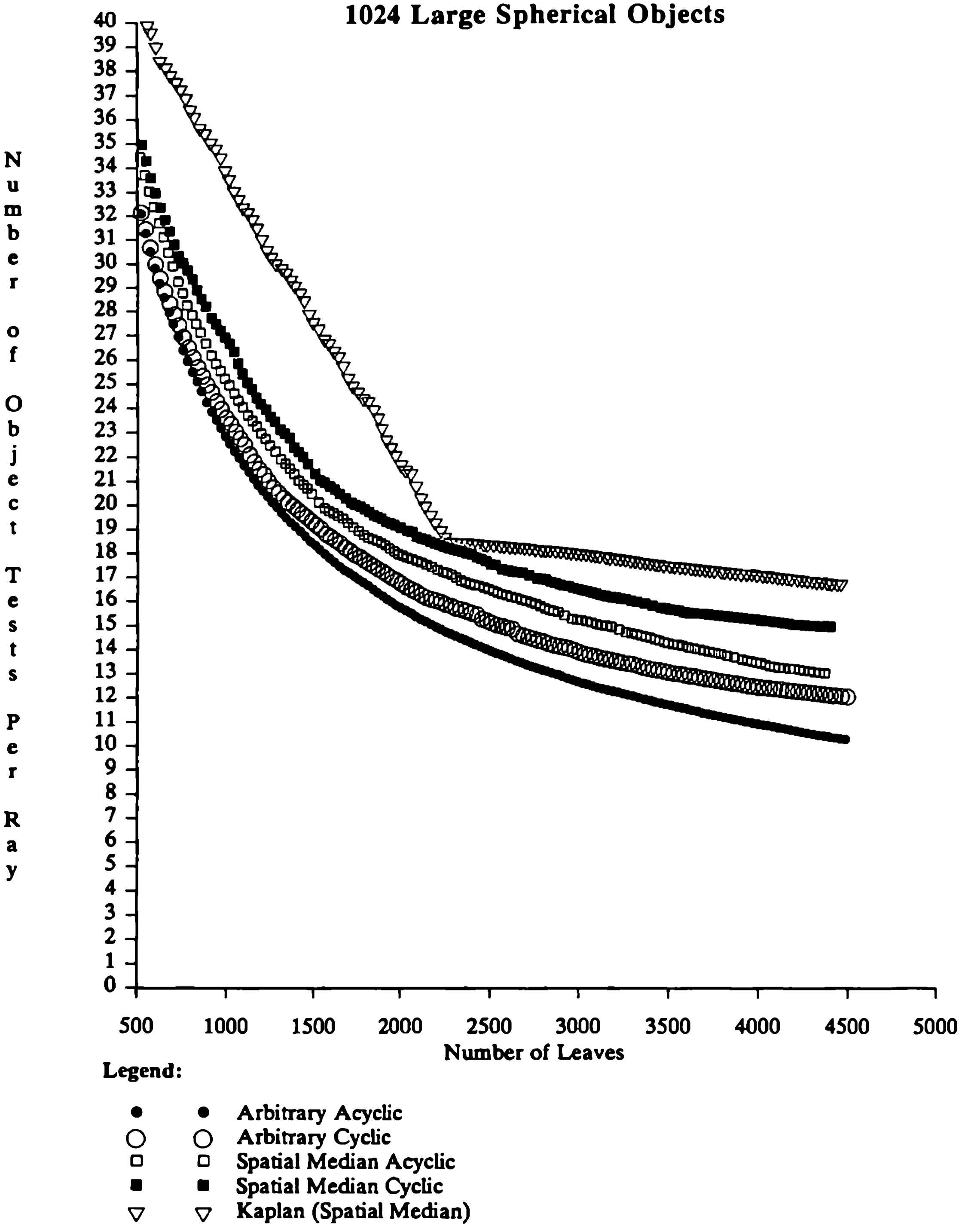
[Whit80] Whitted, T., An Improved Illumination Model for Shaded Display, *Communications of the ACM*, 23(6), June, 1980, 343-349.

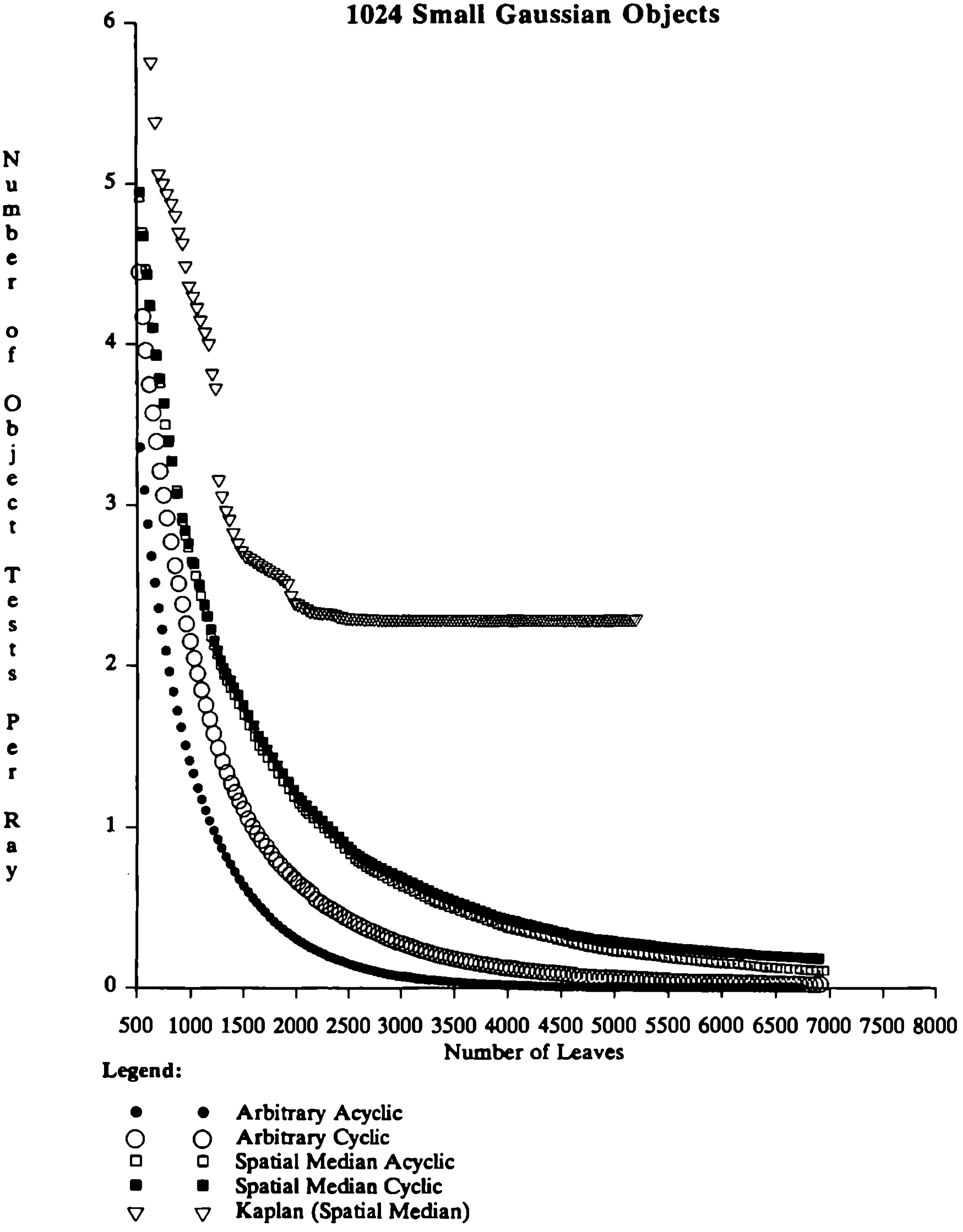
Appendix A
Graphical Results of Construction Algorithms

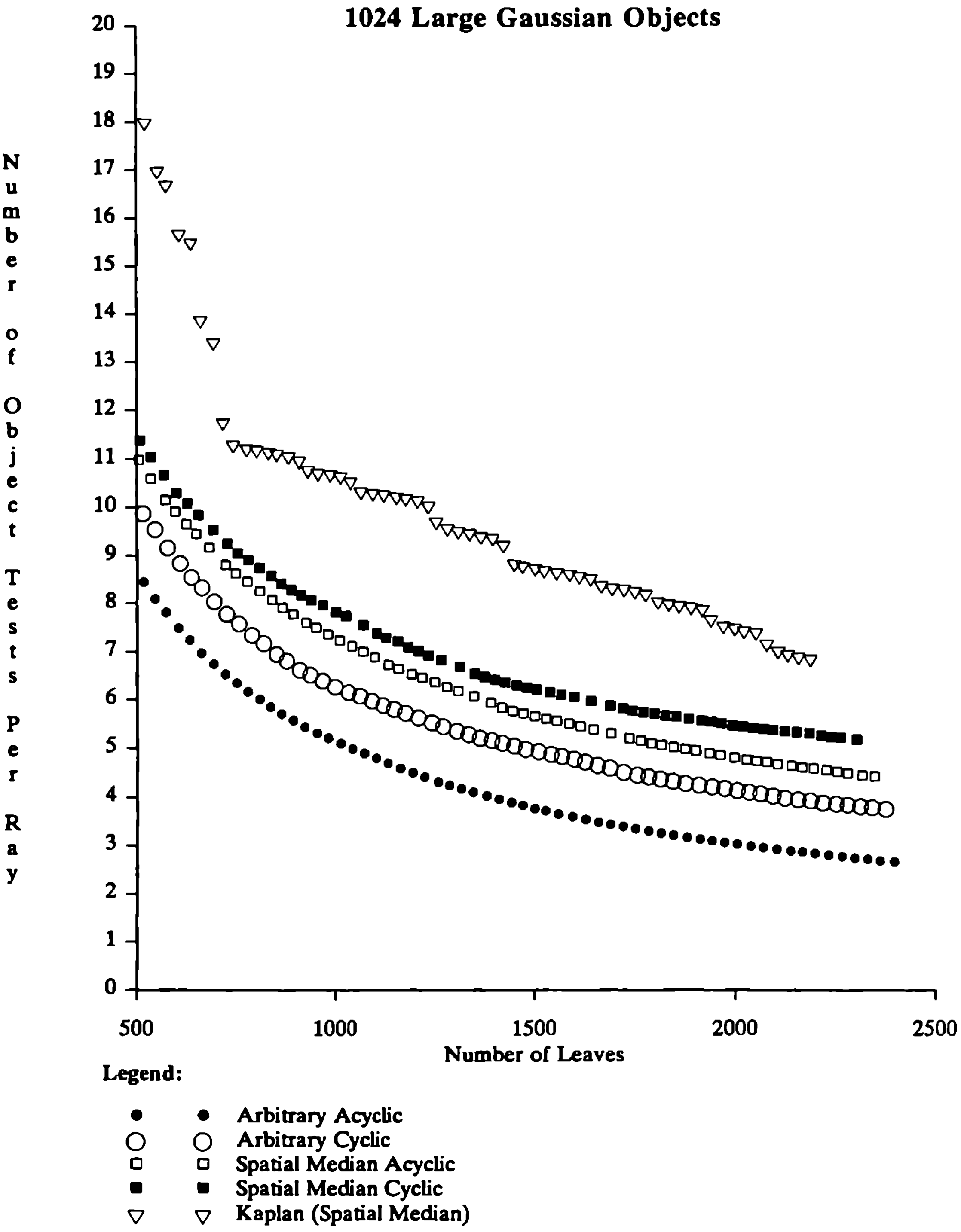
Objects

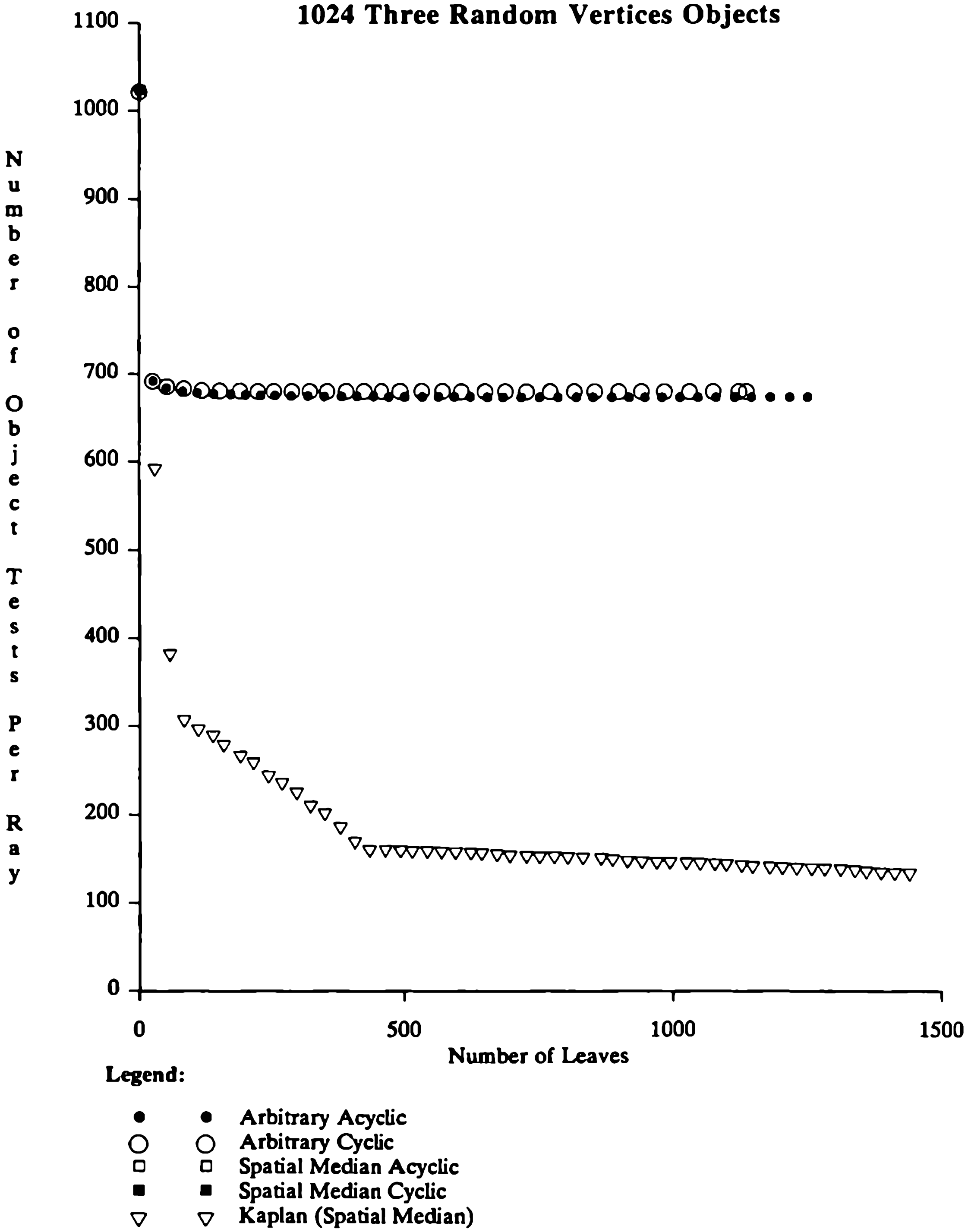
The following are graphs of estimated number of object intersection tests per ray traversal versus the number of leaves (empty plus non-empty). Only the data for 2 sizes of each scene are presented, due to space considerations. These scene sizes are 1024 objects and the largest number of objects run, except in the TRV case where data for 64 and 1024 objects are presented. The graphs of these scene sizes are representative of the other sizes as well. The following are for scene size 1024.


1024 Small Spherical Objects
Number of Object Tests Per Ray
10
9
8
7
6
5
4
3
2
1
0
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


1024 Large Spherical Objects
Number of Object Tests Per Ray
40
39
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

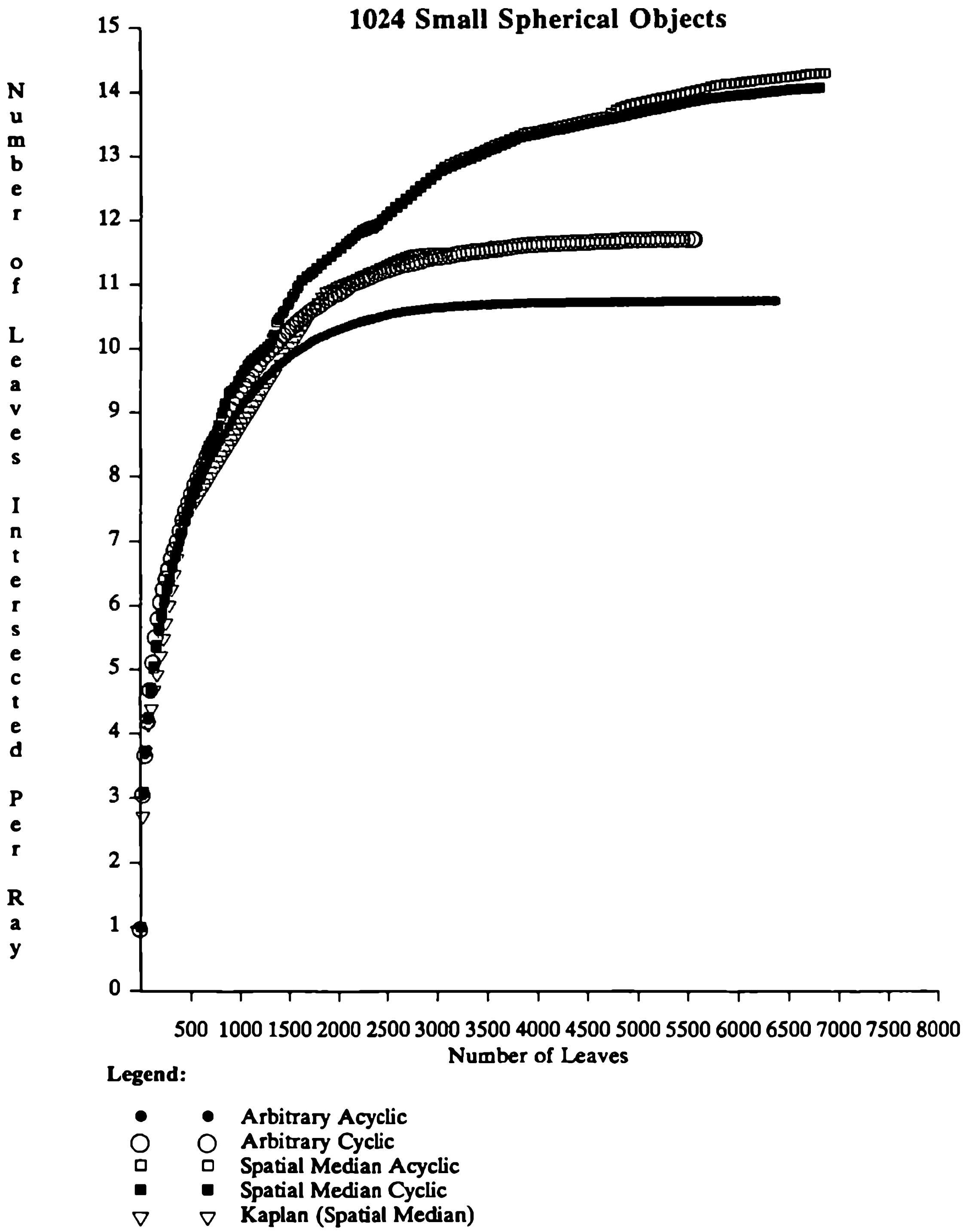

1024 Small Gaussian Objects
Number of Object Tests Per Ray
0
1
2
3
4
5
6
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

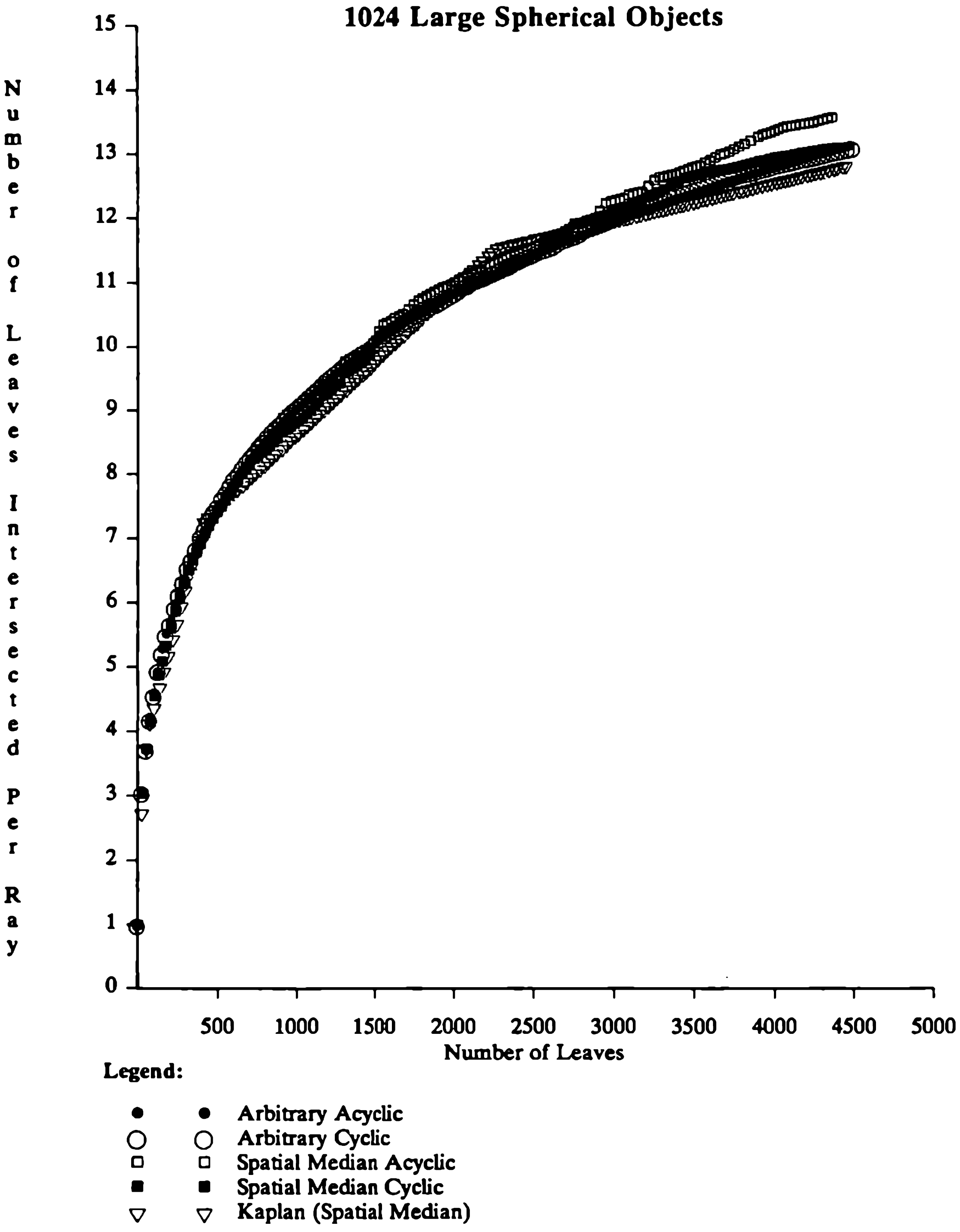

1024 Large Gaussian Objects
Number of Object Tests Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

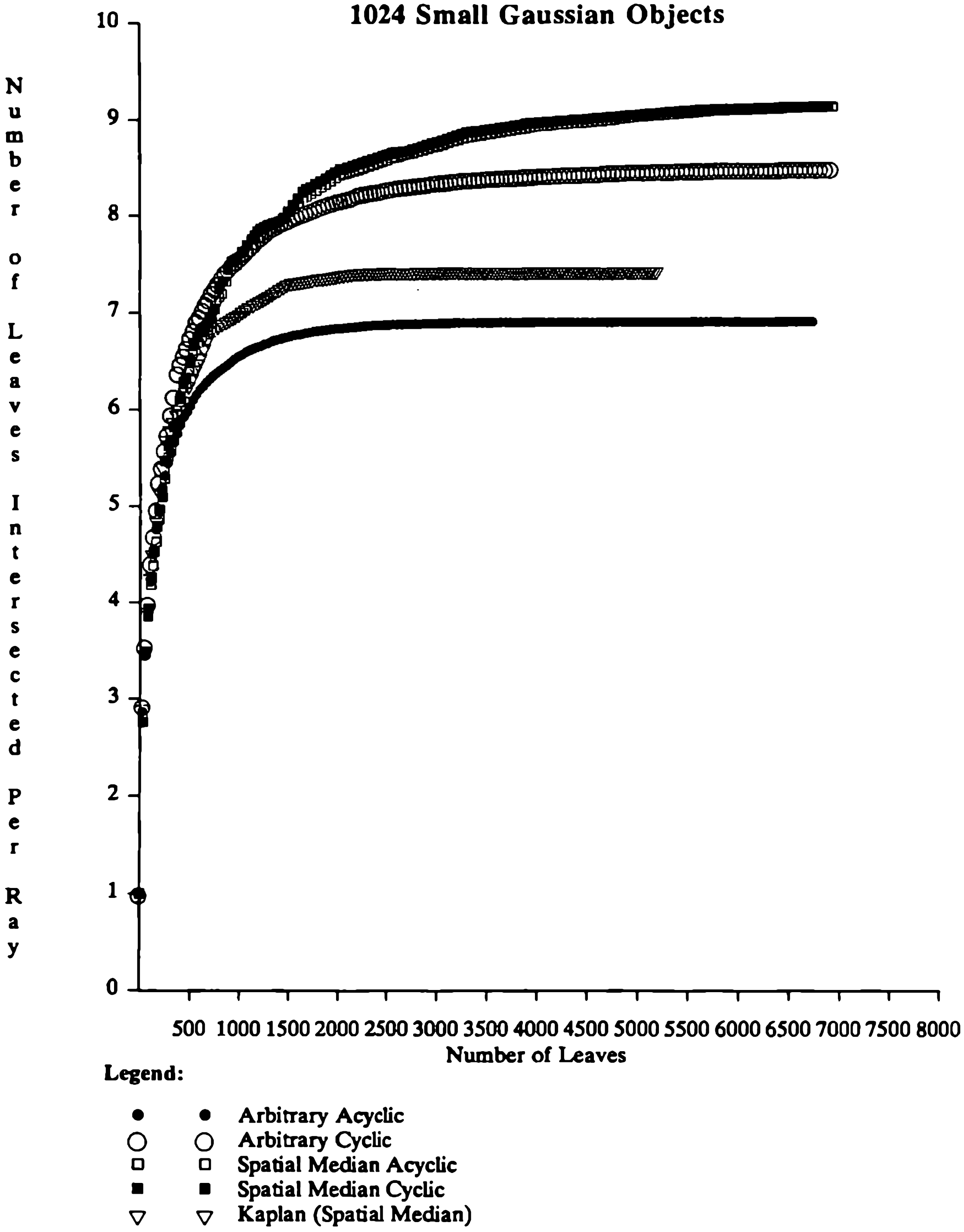

1024 Three Random Vertices Objects
Number of Object Tests Per Ray
1100
1000
900
800
700
600
500
400
300
200
100
0
0
500
1000
1500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

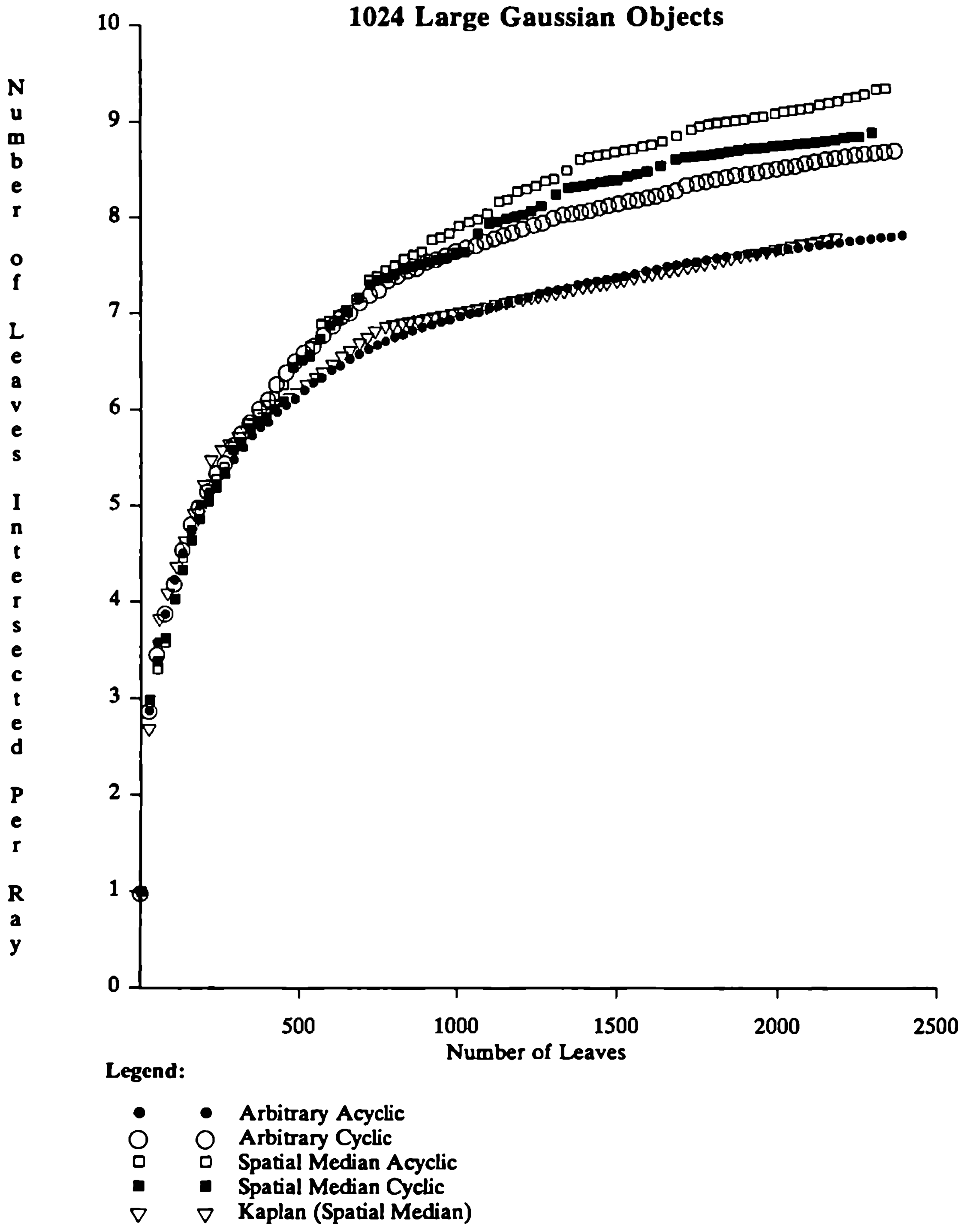
Leaves

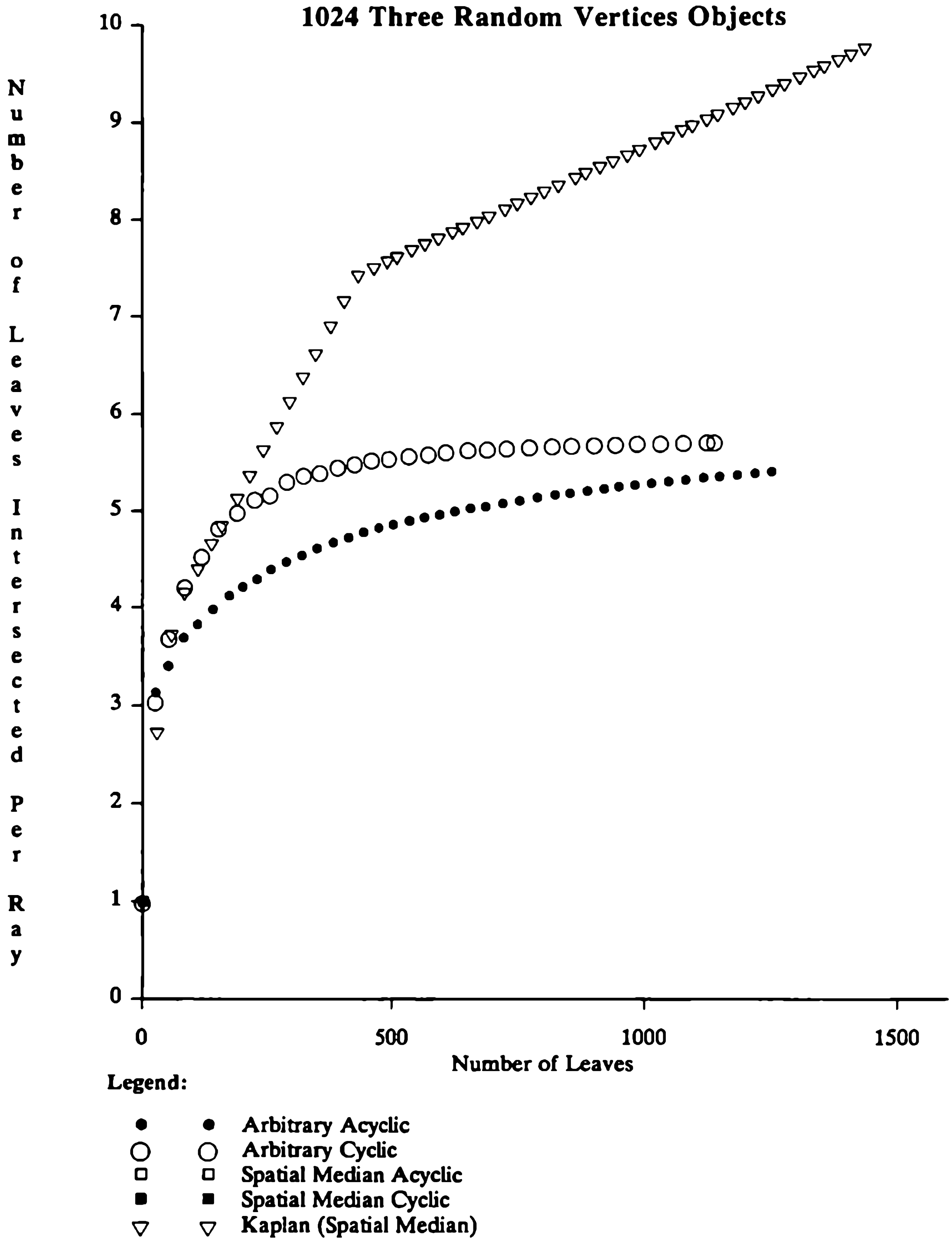
The following are the analogous graphs for number of leaves intersected per ray.


1024 Small Spherical Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


1024 Large Spherical Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

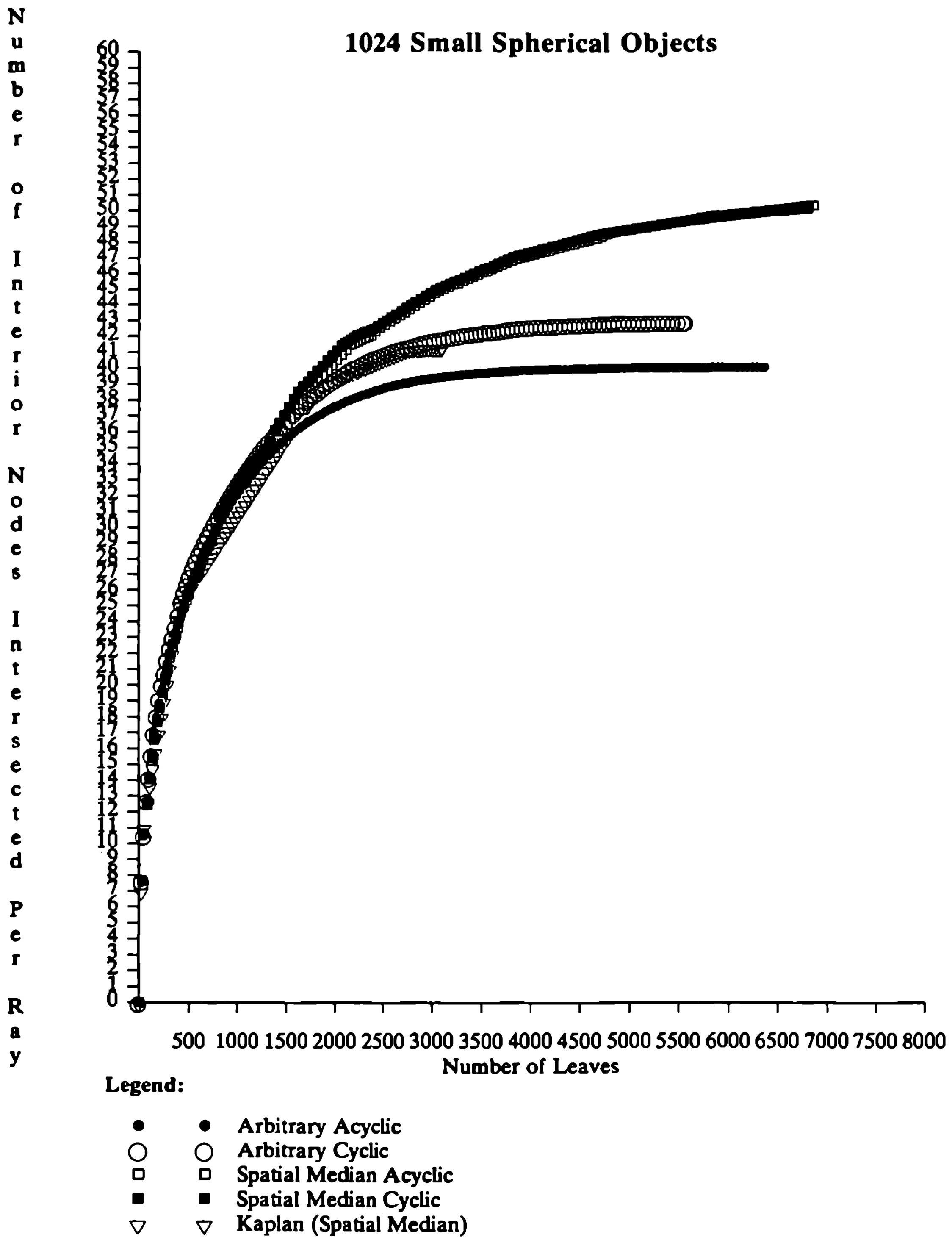

1024 Small Gaussian Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

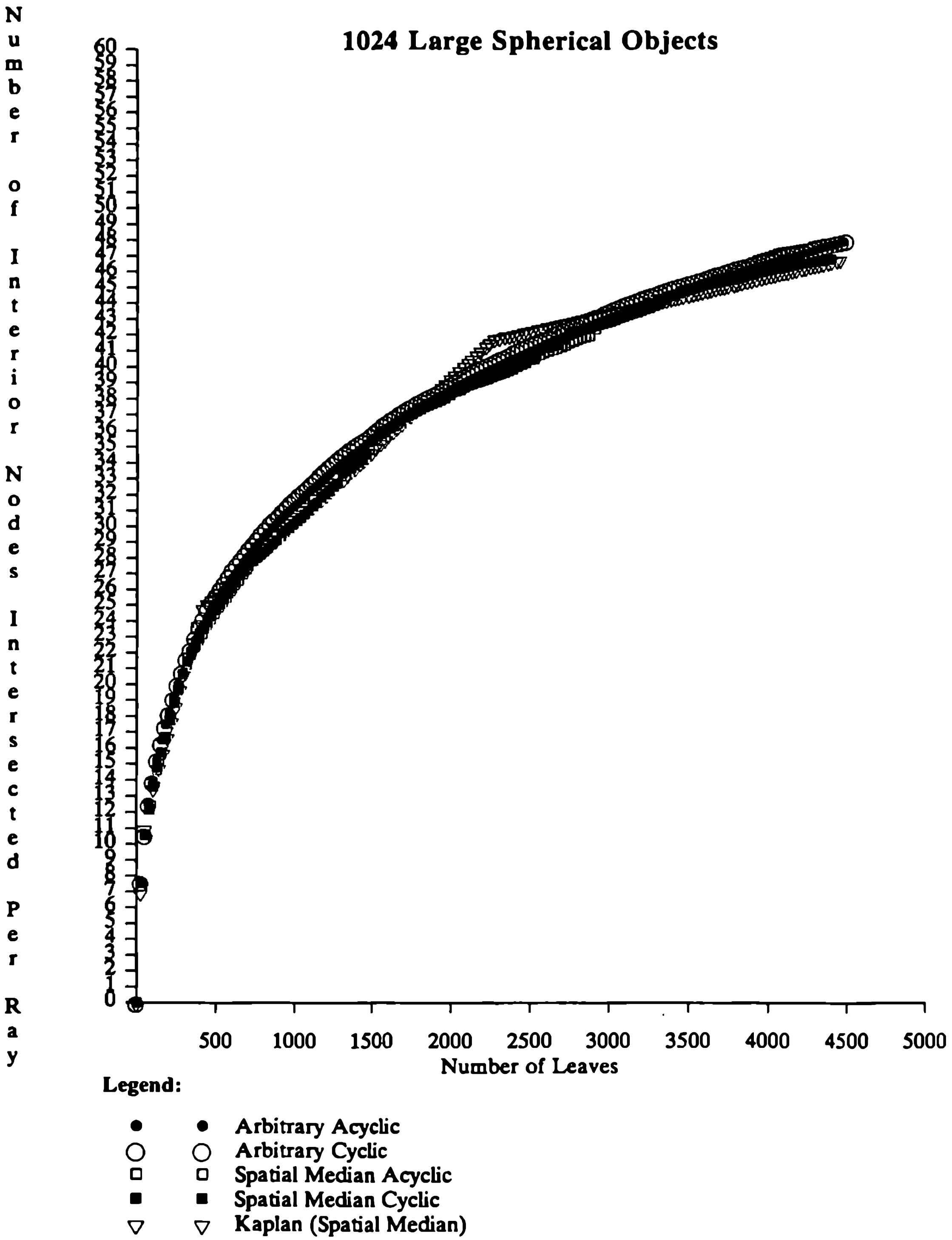

1024 Large Gaussian Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

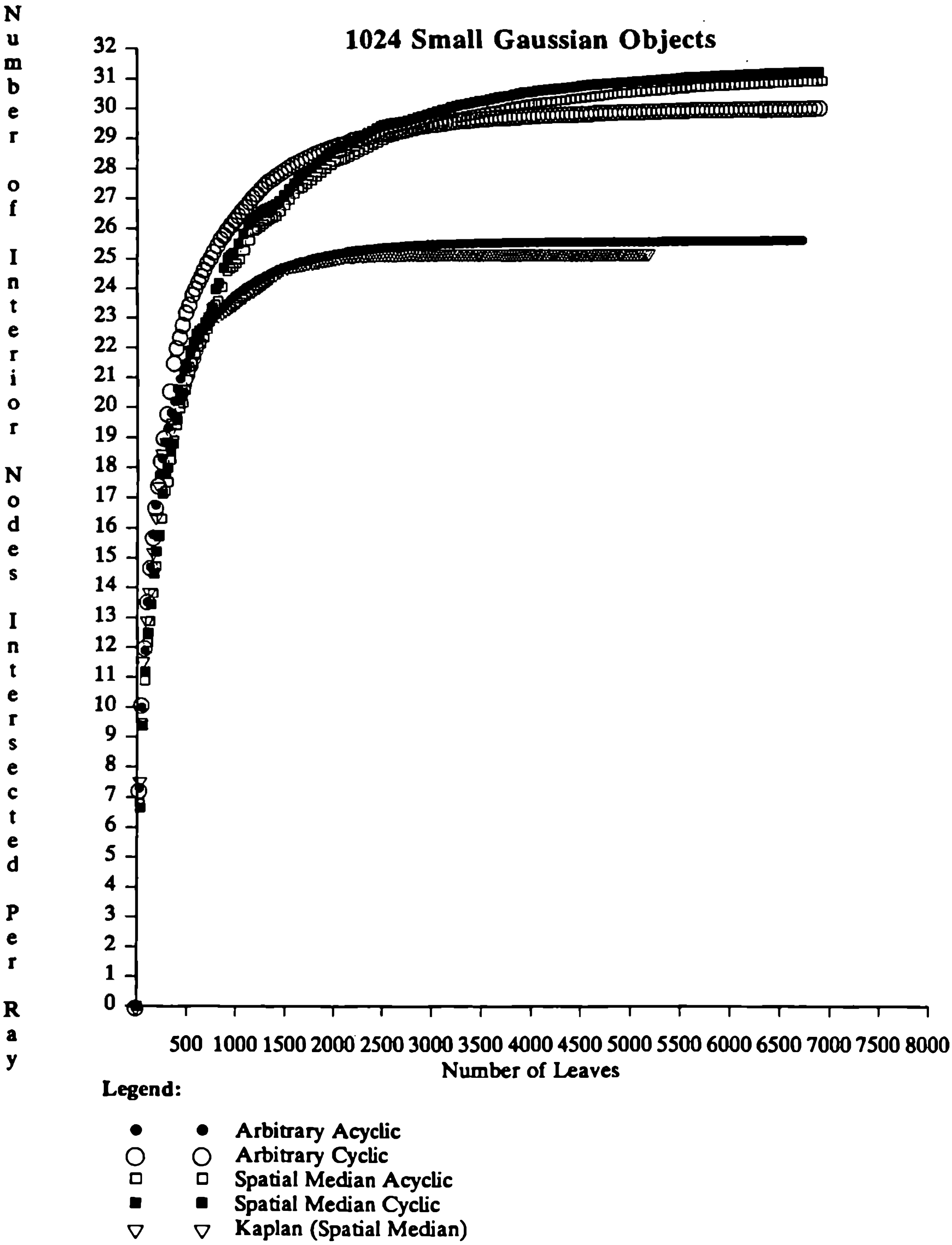

1024 Three Random Vertices Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
0
500
1000
1500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

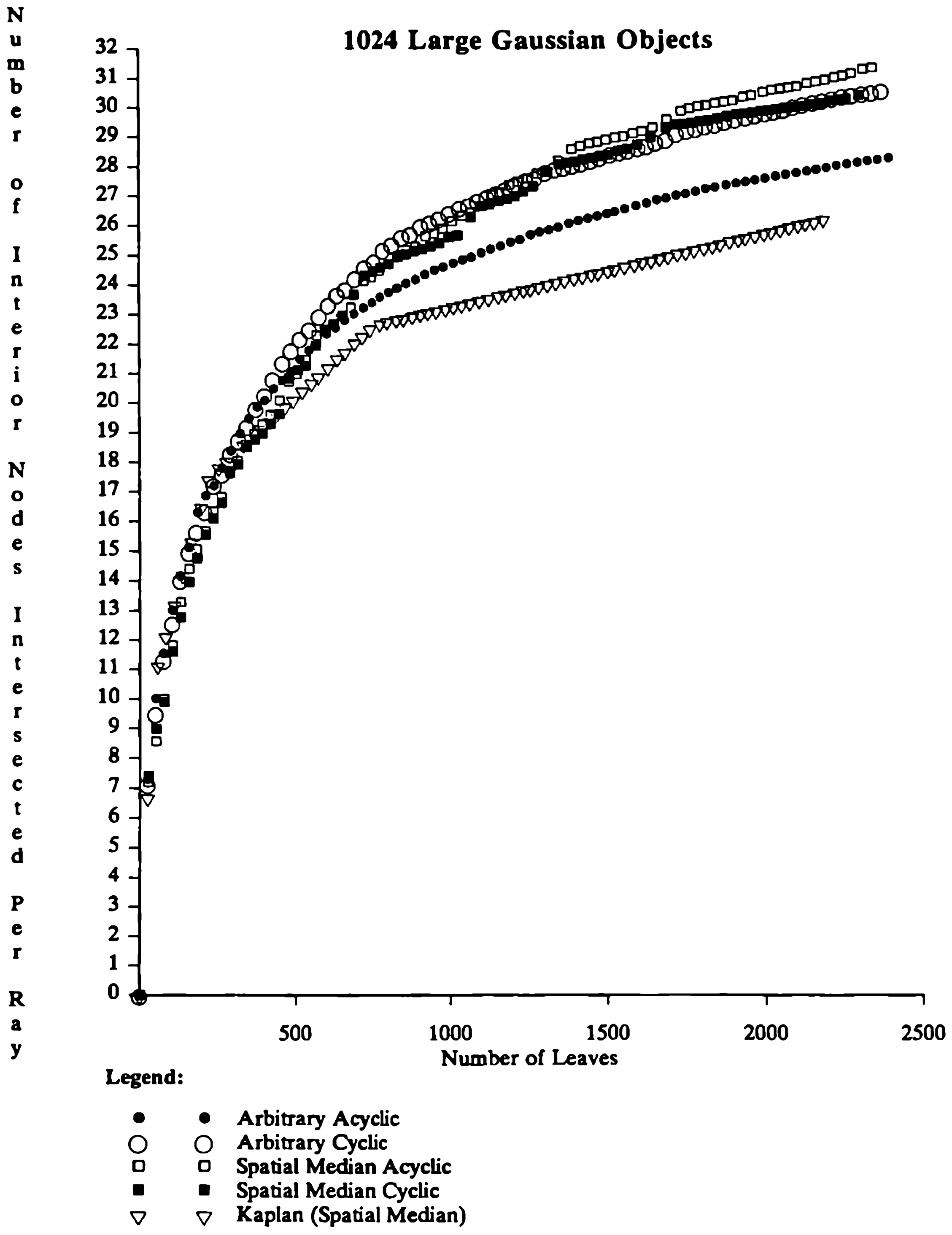
Nodes

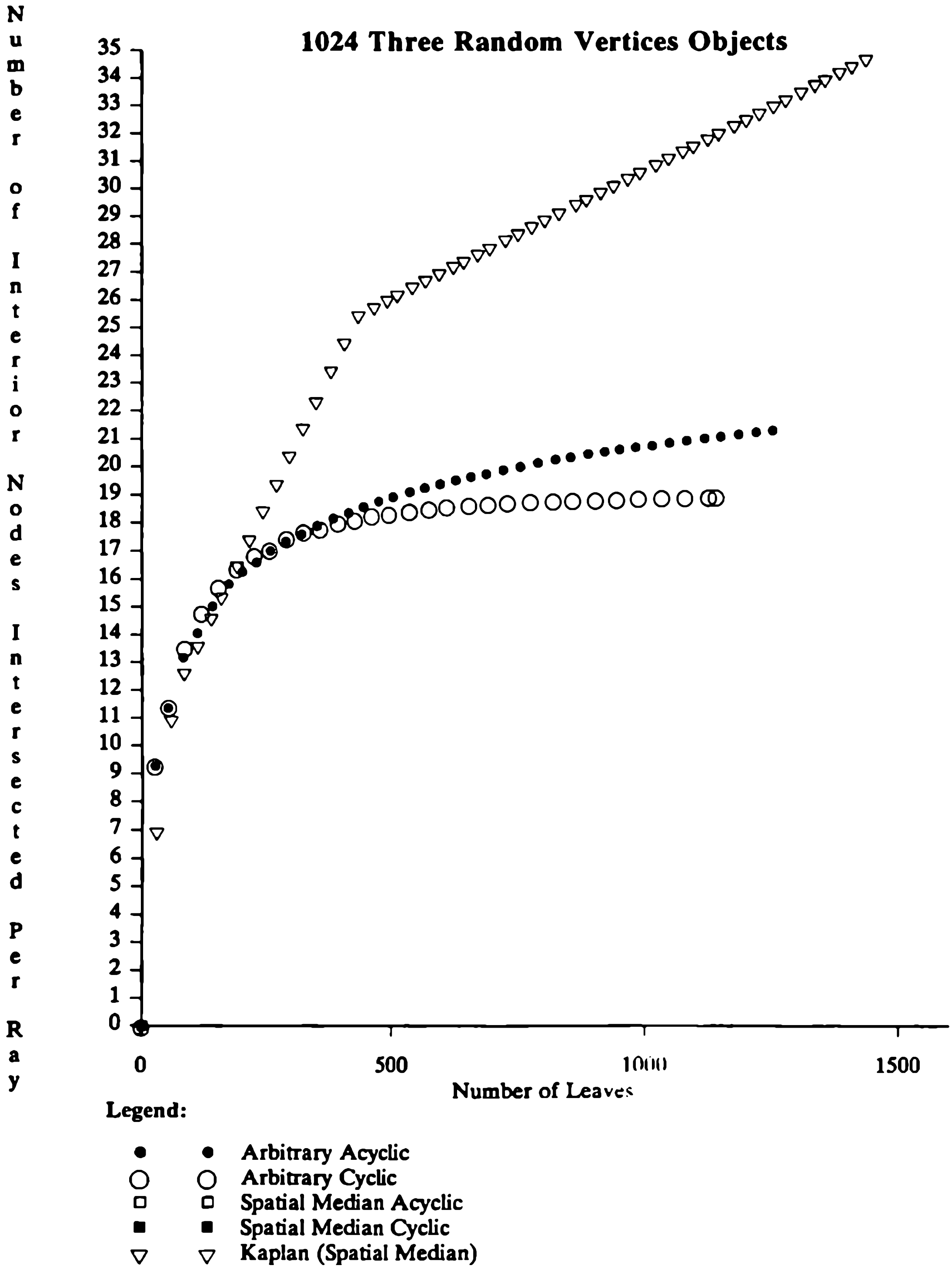
The following are the analogous graphs for number of interior nodes intersected.


1024 Small Spherical Objects
Number of Interior Nodes Intersected Per Ray
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


1024 Large Spherical Objects
Number of Interior Nodes Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


1024 Small Gaussian Objects
Number of Interior Nodes Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

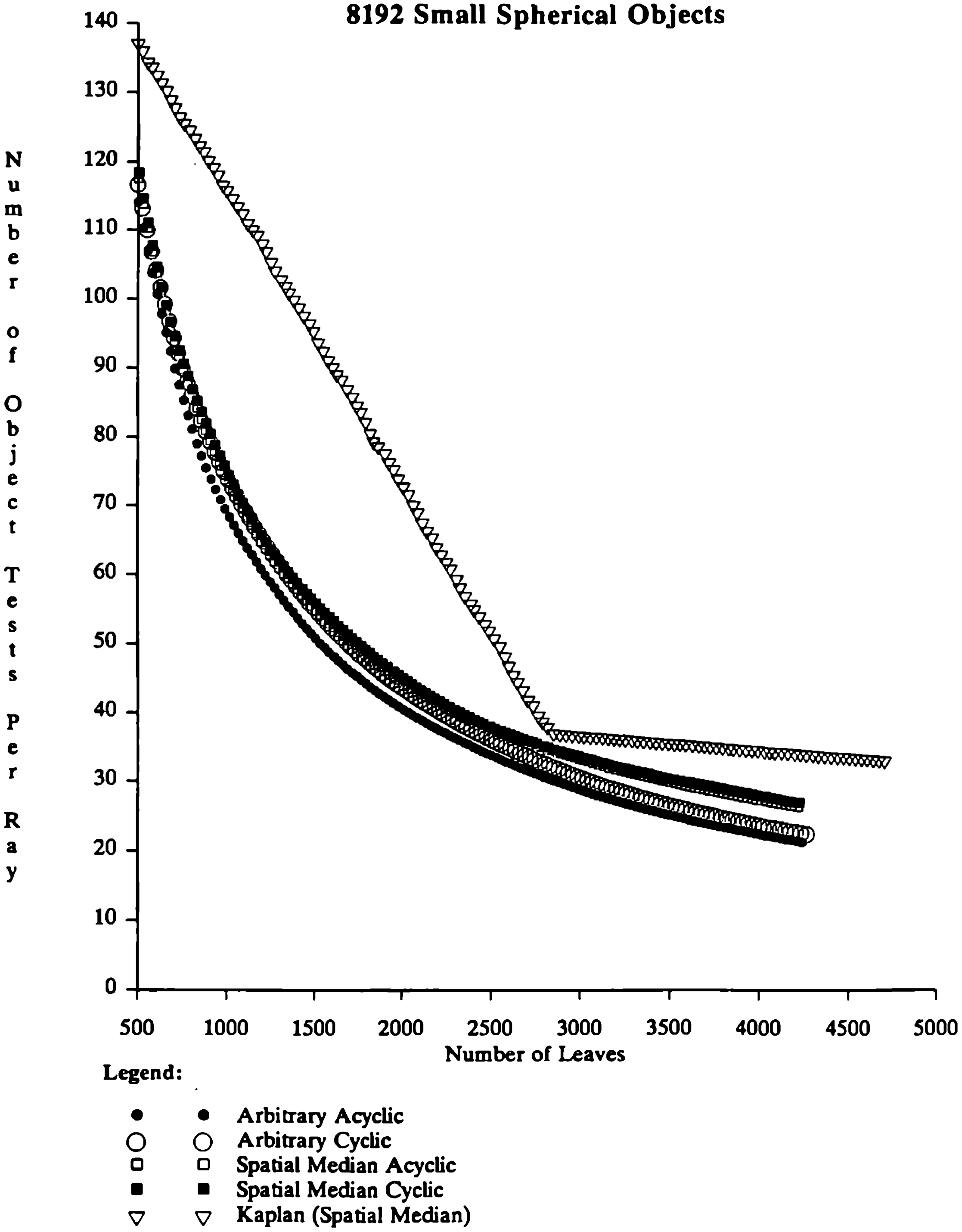

1024 Large Gaussian Objects
Number of Interior Nodes Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

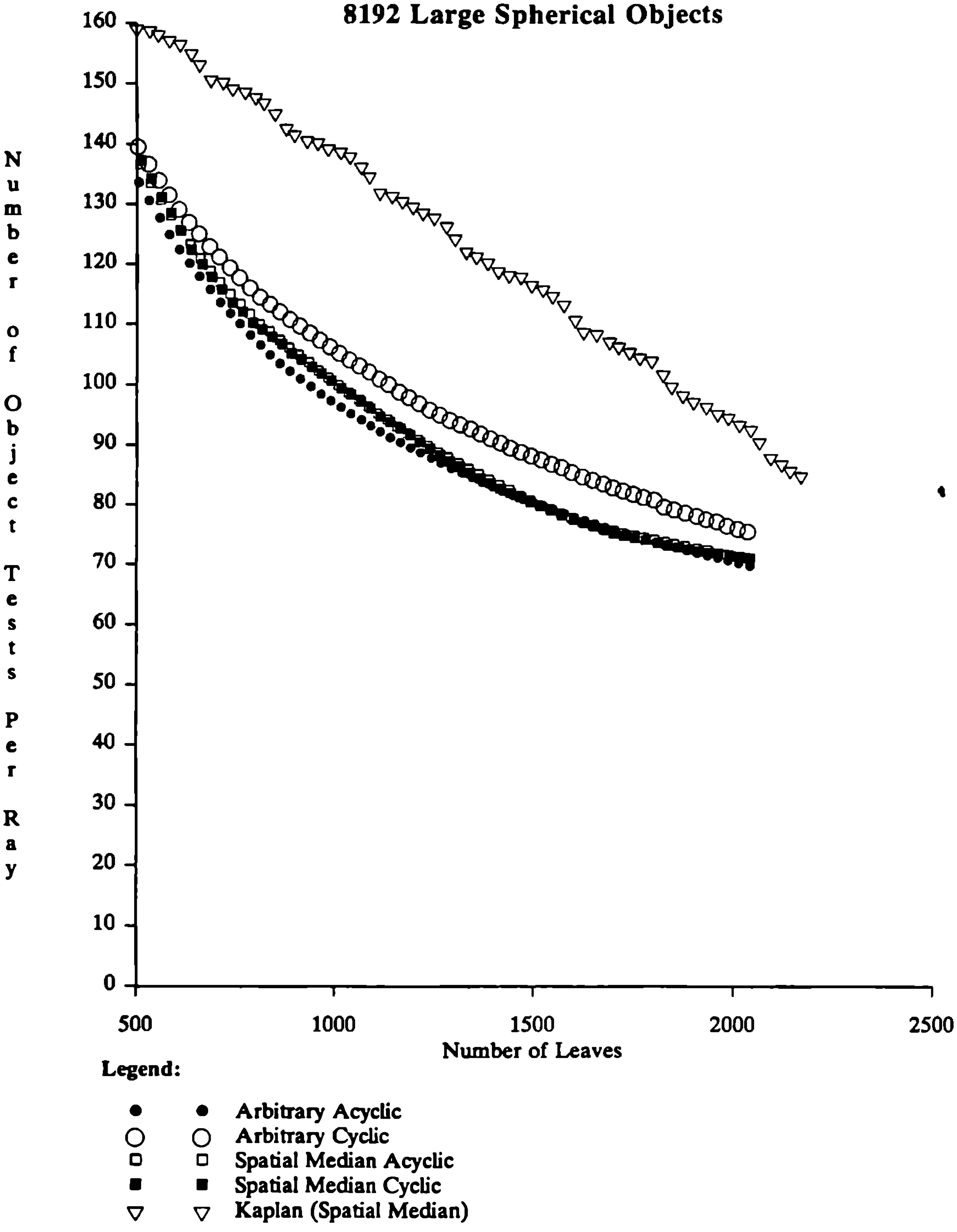

1024 Three Random Vertices Objects
Number of Interior Nodes Intersected Per Ray
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
500
1000
1500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

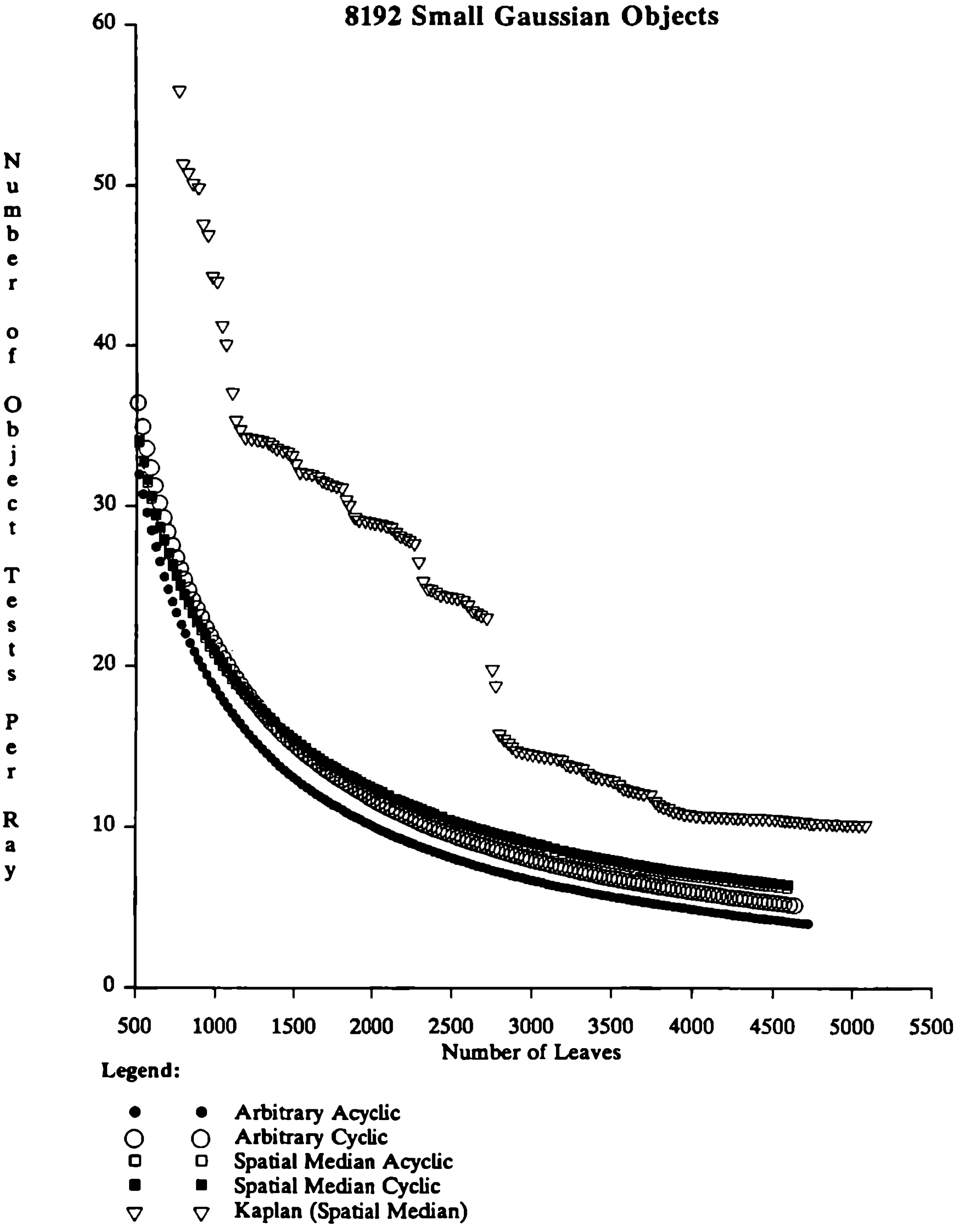
Large Scenes

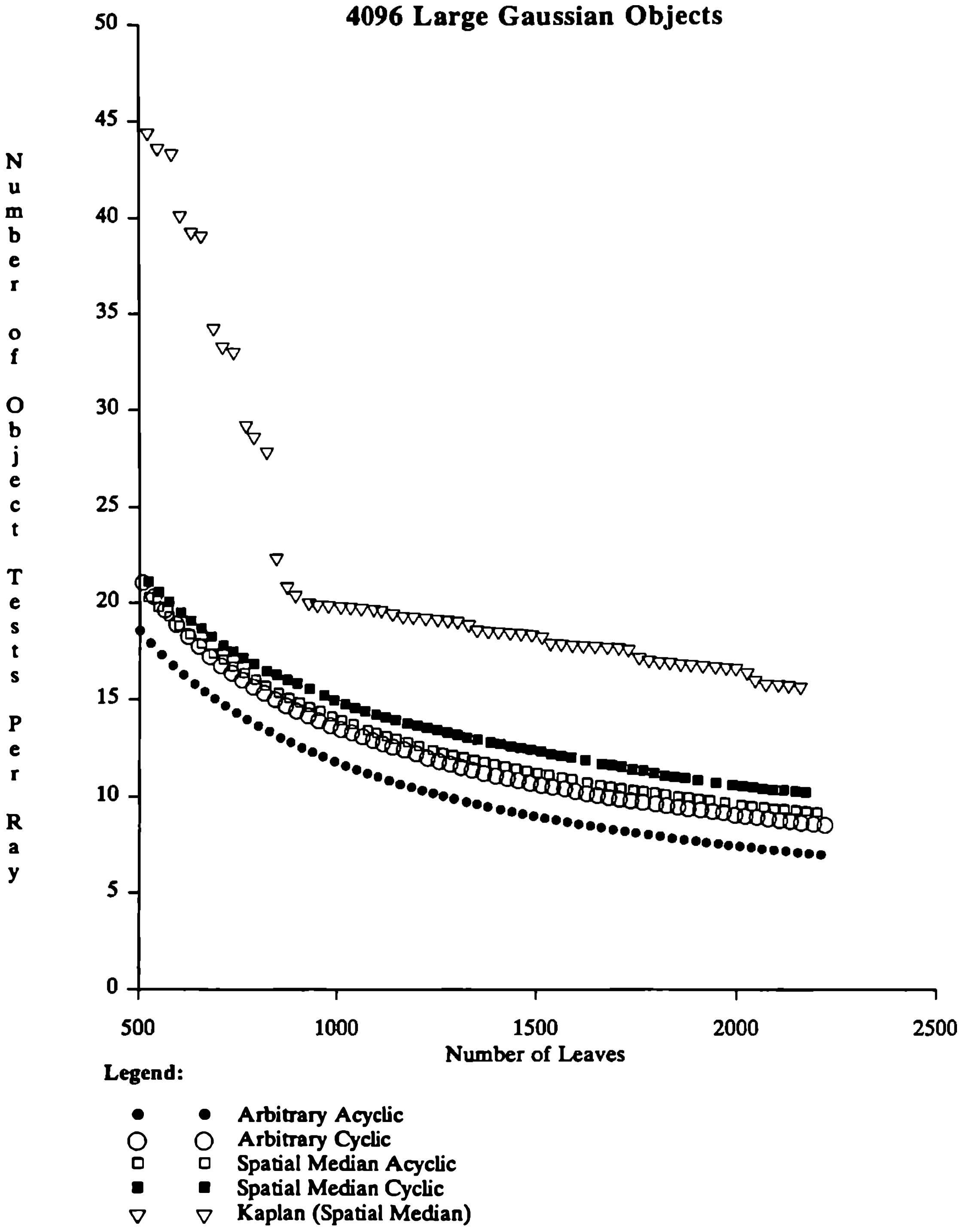
The following are for the largest scene size used for each scene, except for the TRV case, for which the graph is for 64 objects.

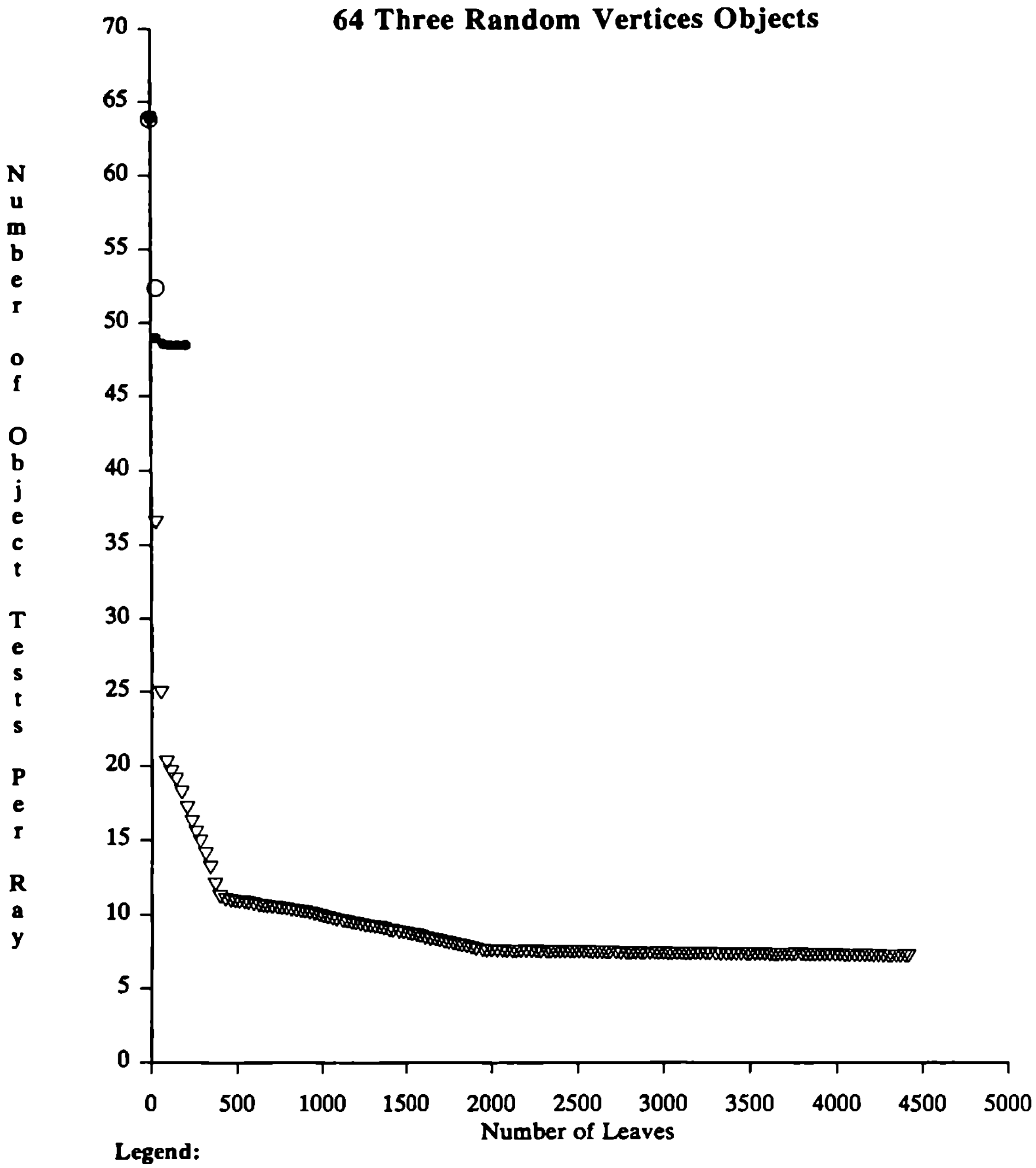
Objects


8192 Small Spherical Objects
Number of Object Tests Per Ray
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


8192 Large Spherical Objects
Number of Object Tests Per Ray
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


8192 Small Gaussian Objects
Number of Object Tests Per Ray
0
10
20
30
40
50
60
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


4096 Large Gaussian Objects
Number of Object Tests Per Ray
50
45
40
35
30
25
20
15
10
5
0
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

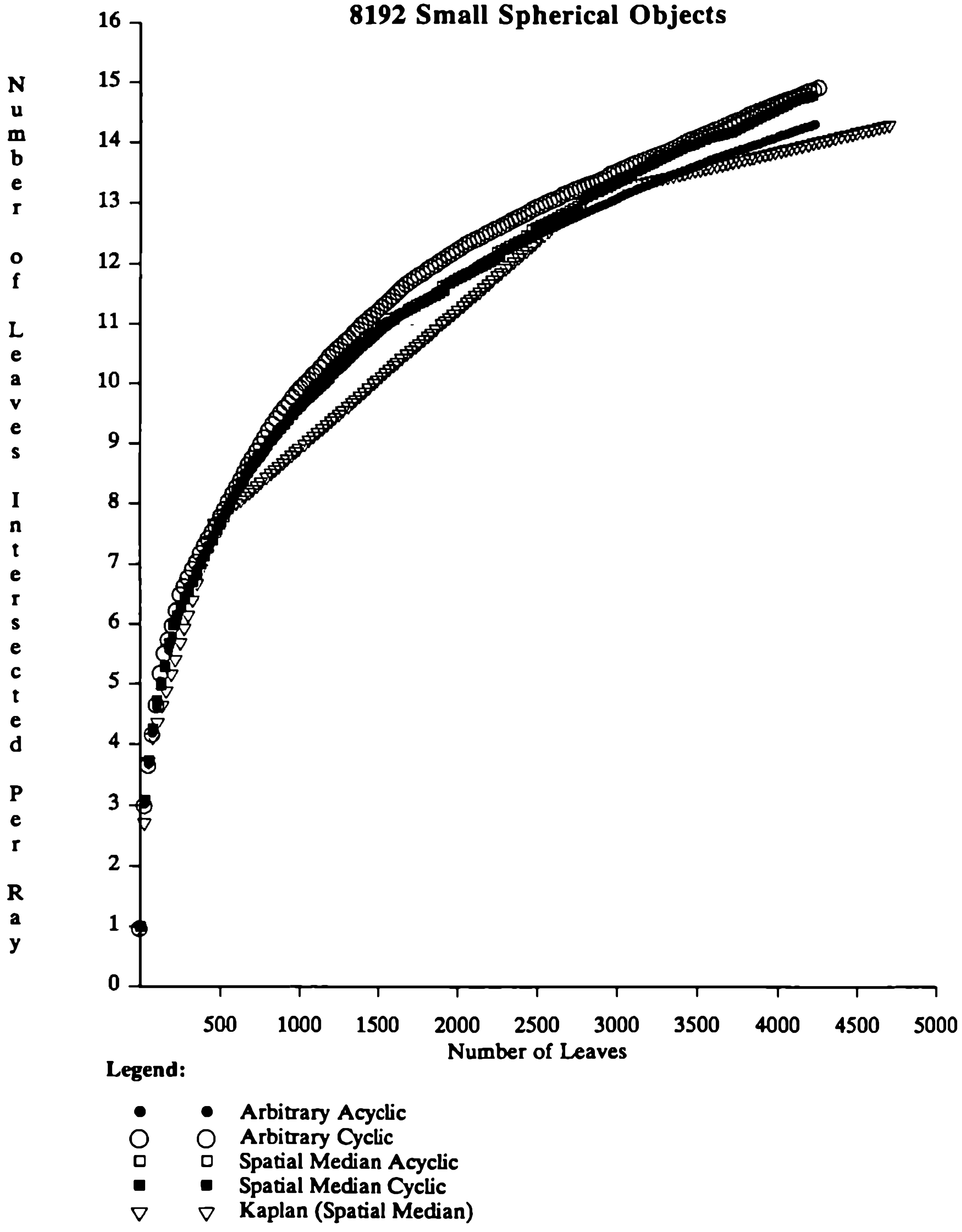


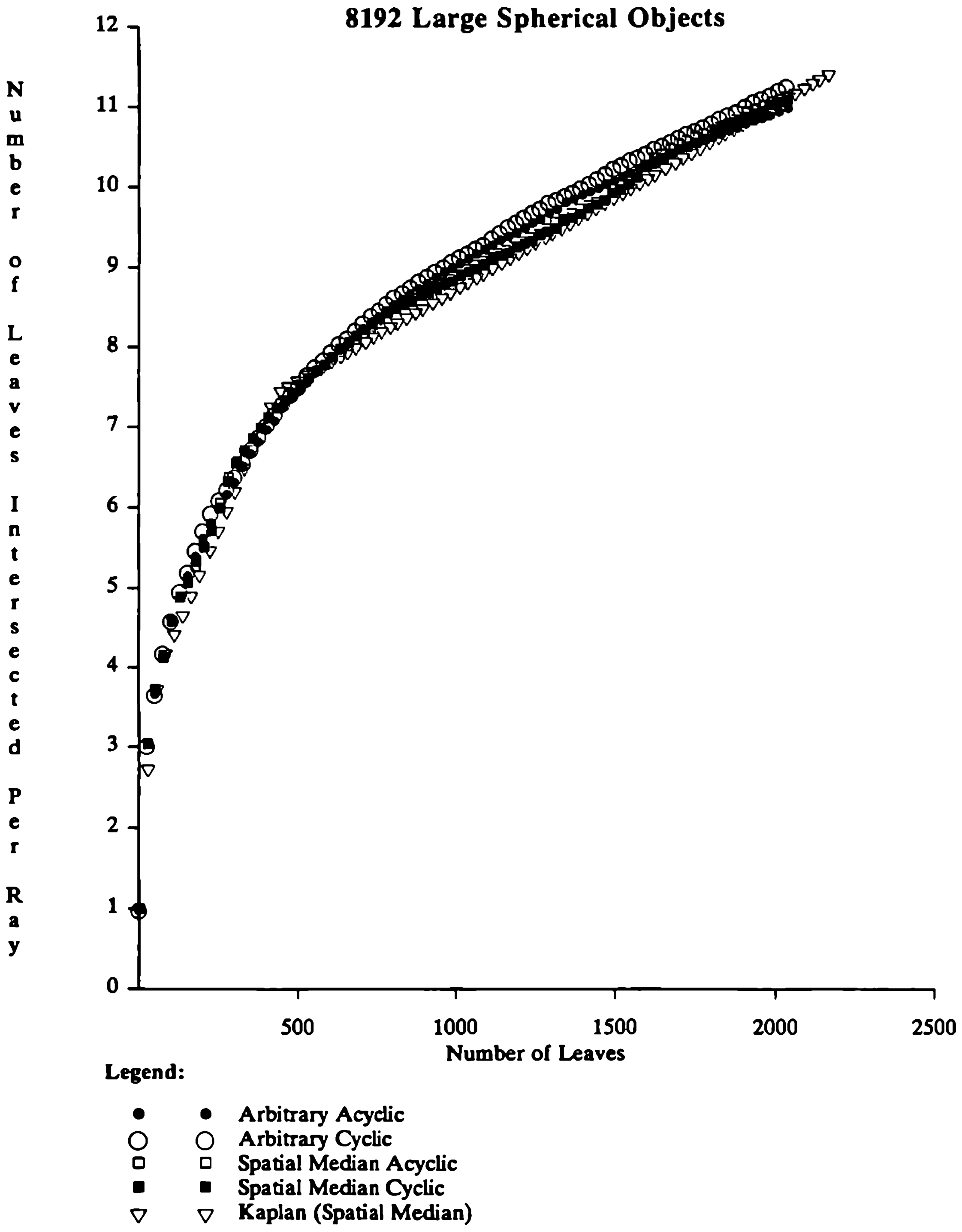
Legend:

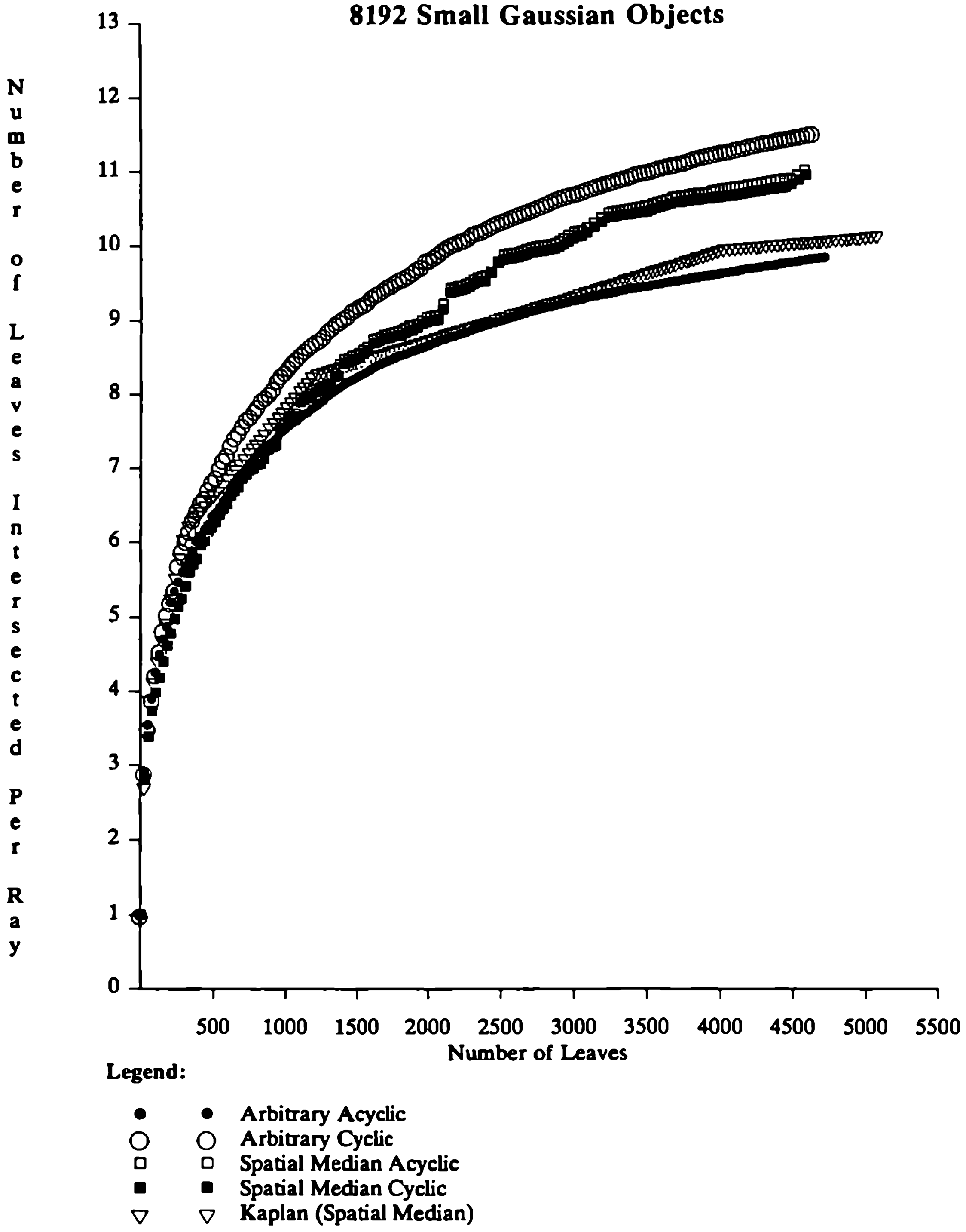
●	●	Arbitrary Acyclic
○	○	Arbitrary Cyclic
□	□	Spatial Median Acyclic
■	■	Spatial Median Cyclic
▽	▽	Kaplan (Spatial Median)

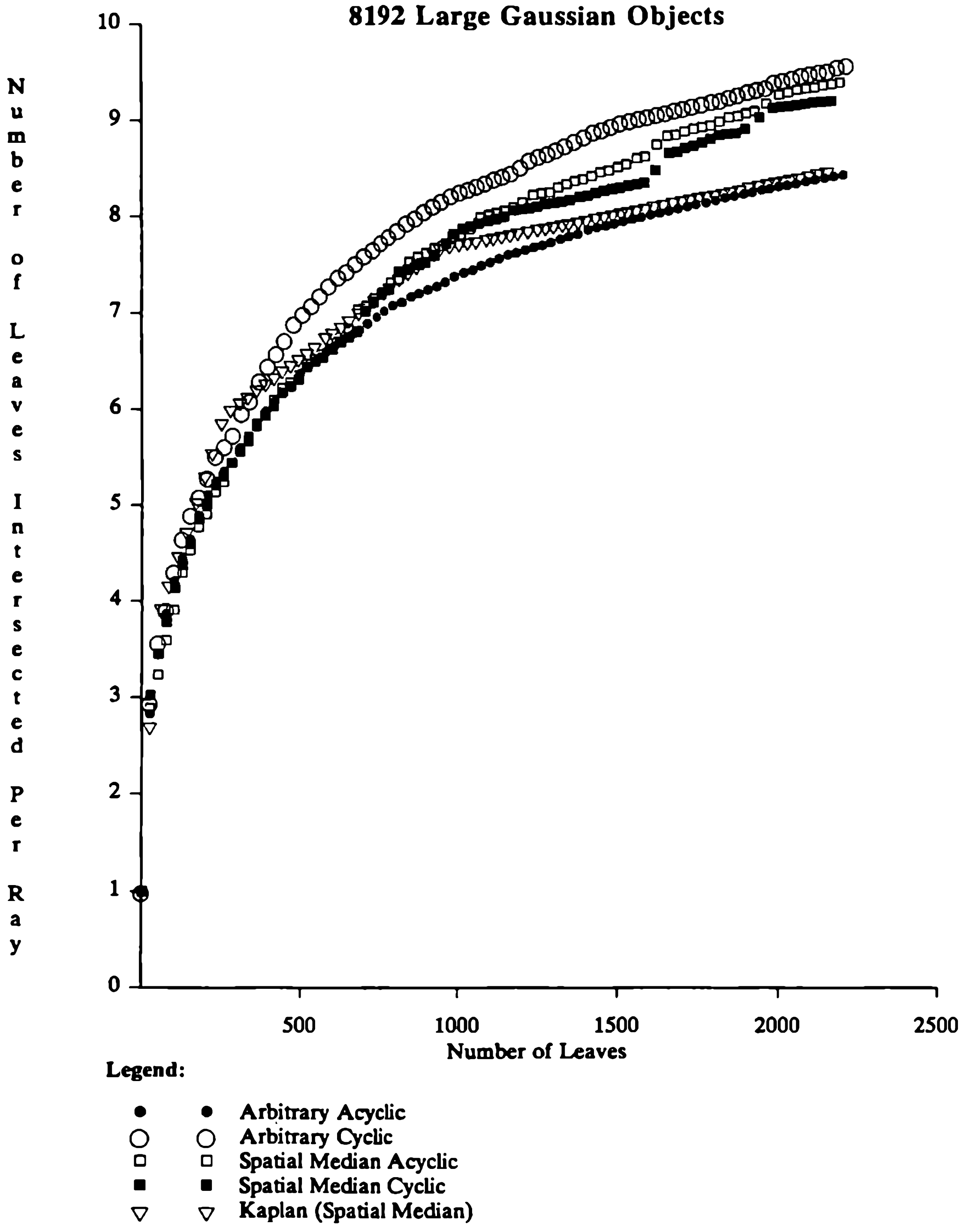
Leaves

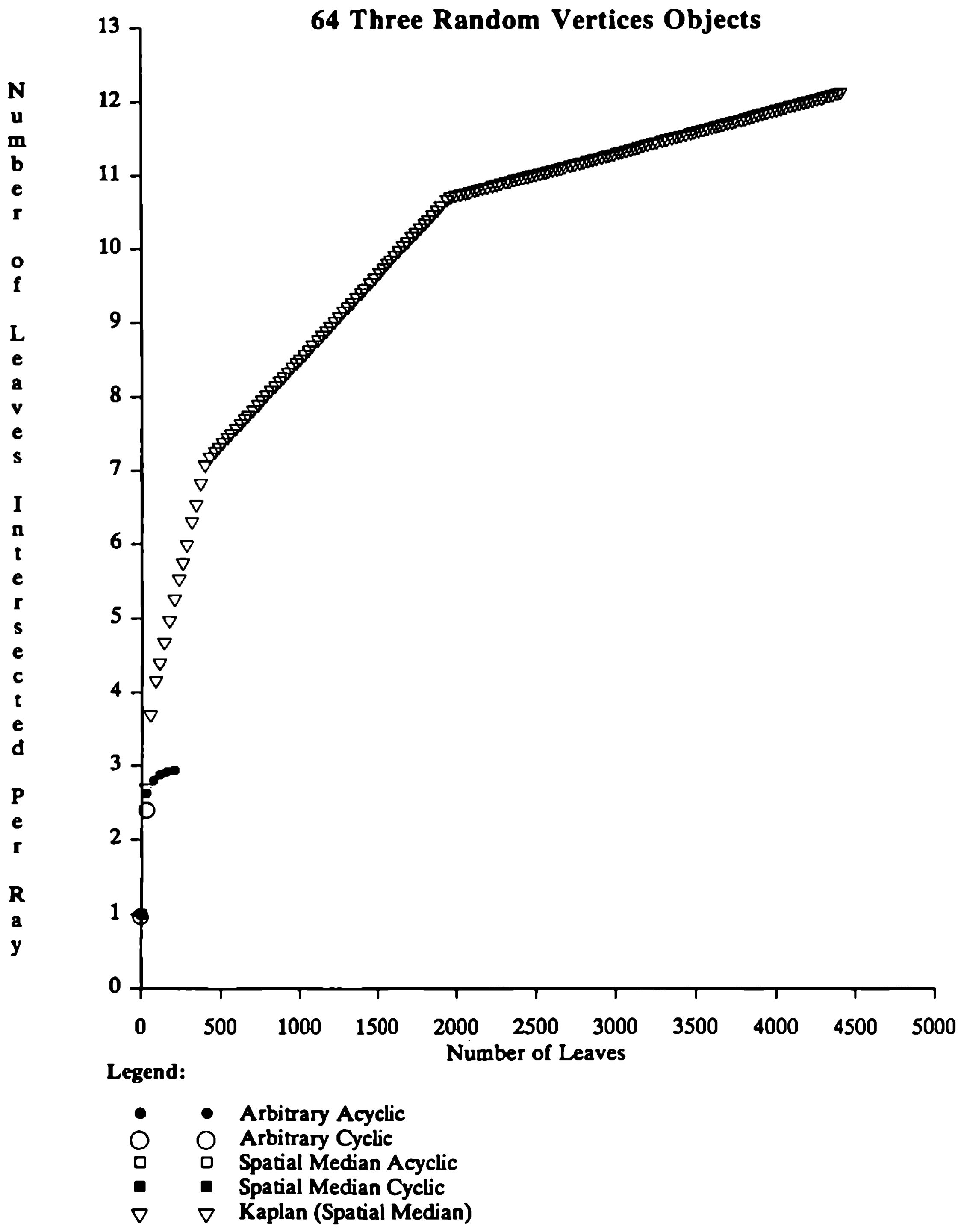
The following are the analogous graphs for number of leaves intersected per ray.


8192 Small Spherical Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


8192 Large Spherical Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

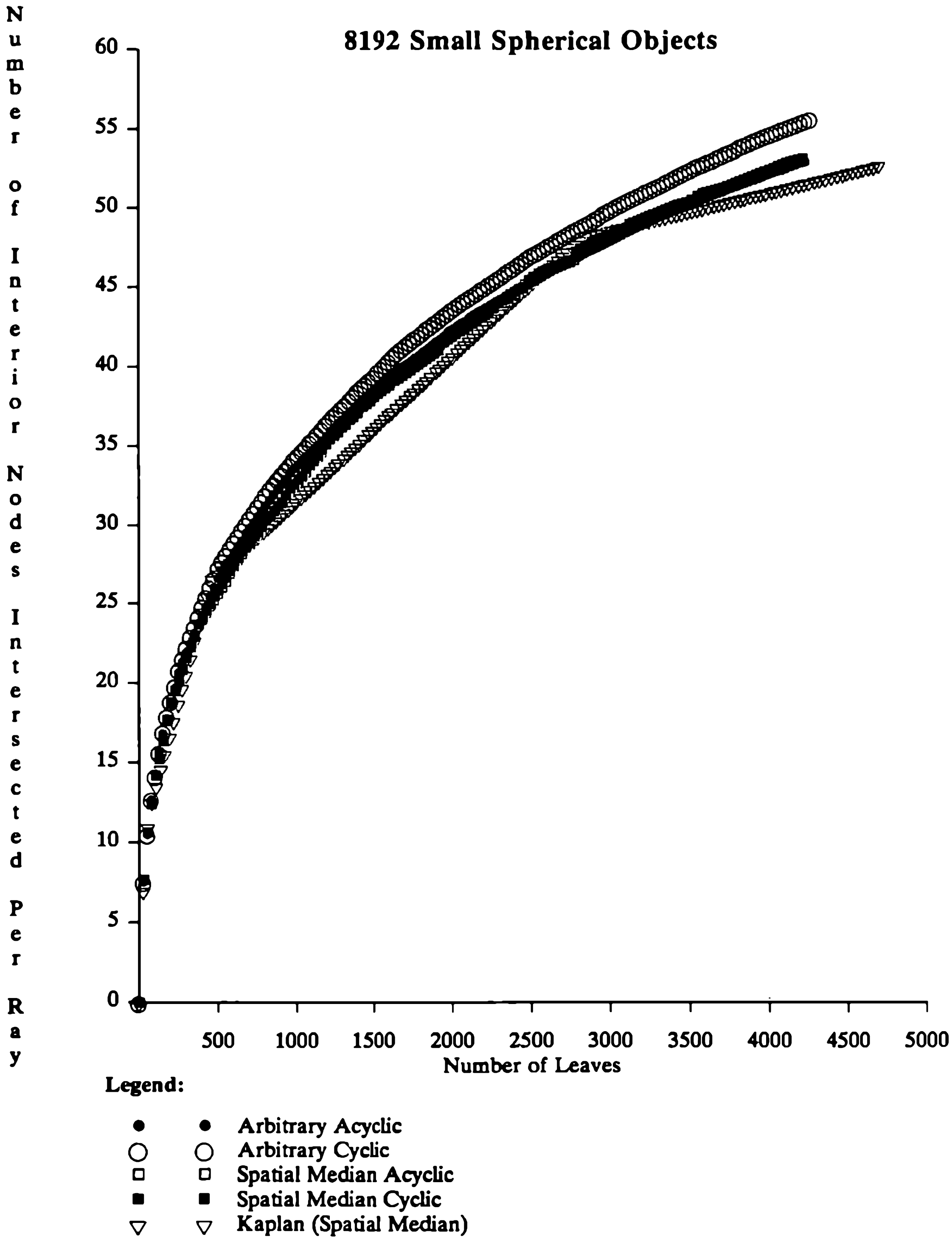

8192 Small Gaussian Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

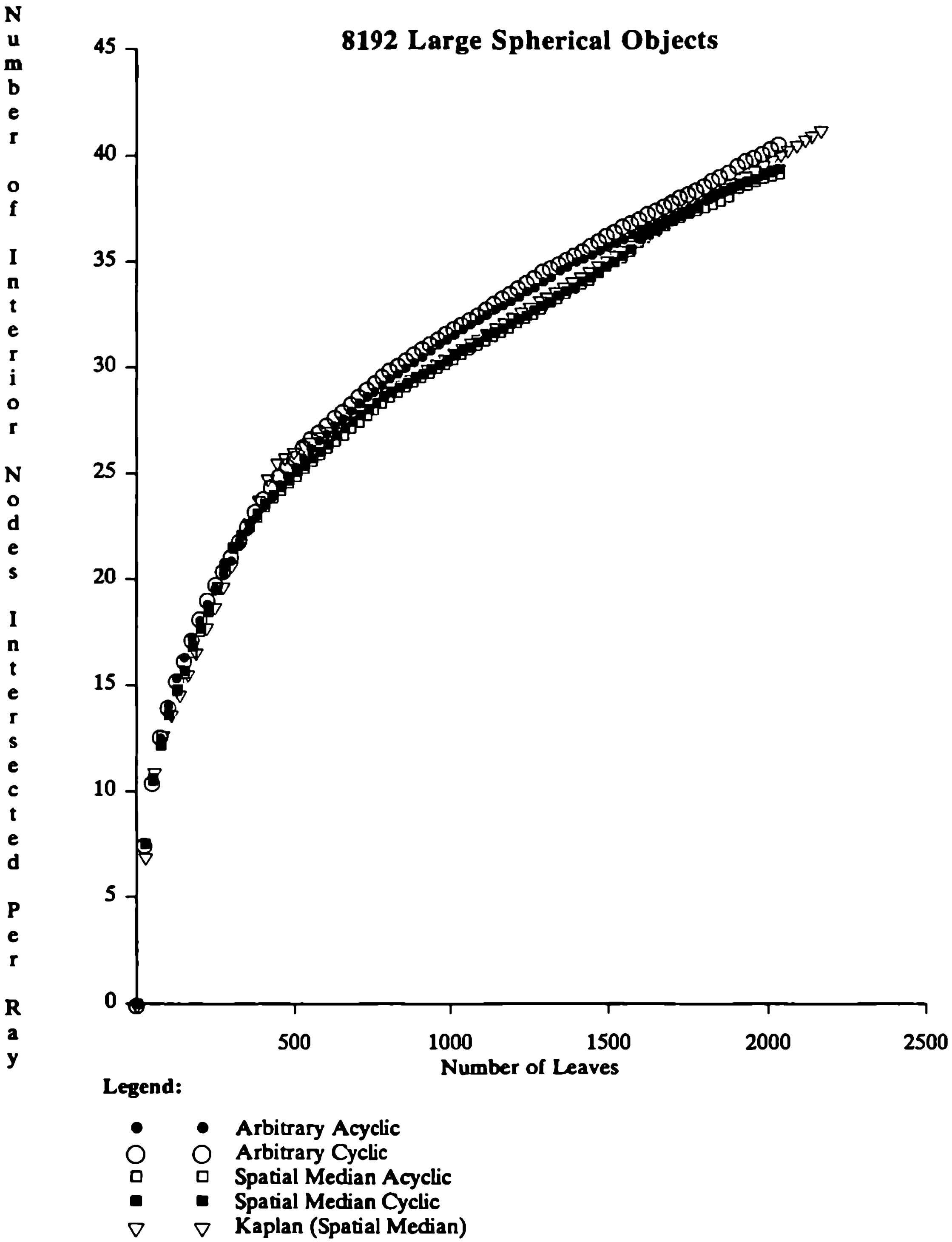

8192 Large Gaussian Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

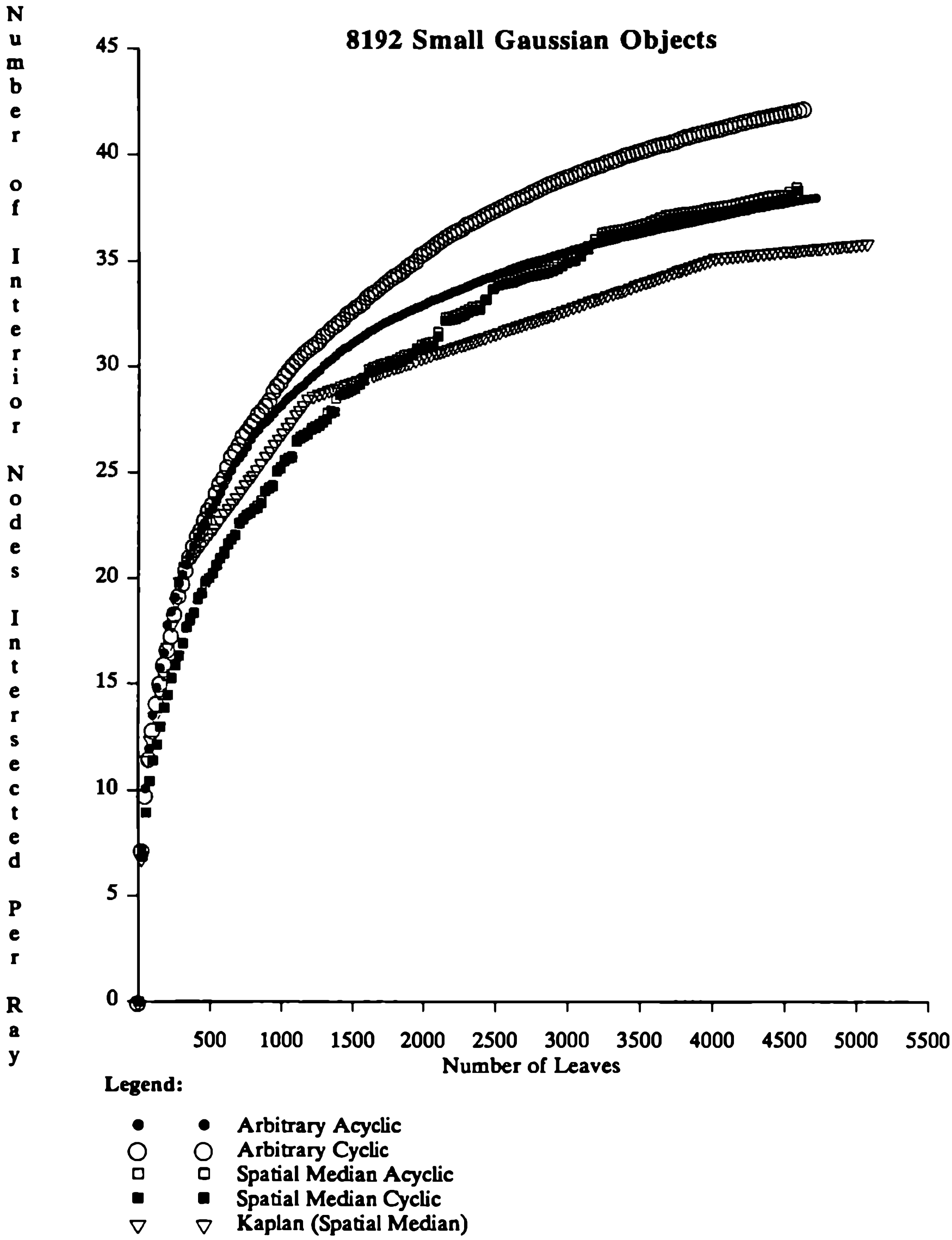

64 Three Random Vertices Objects
Number of Leaves Intersected Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
13
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

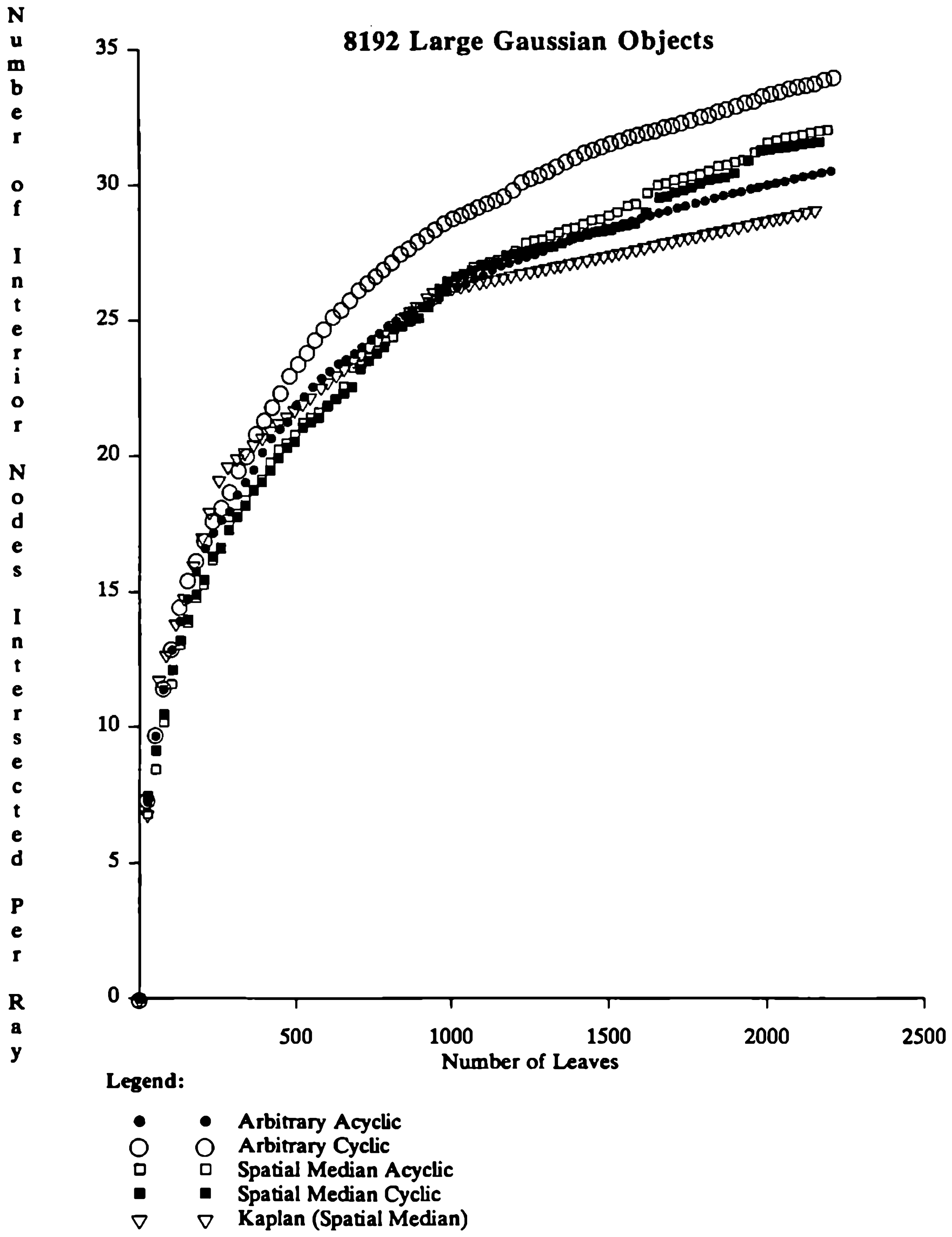
Nodes

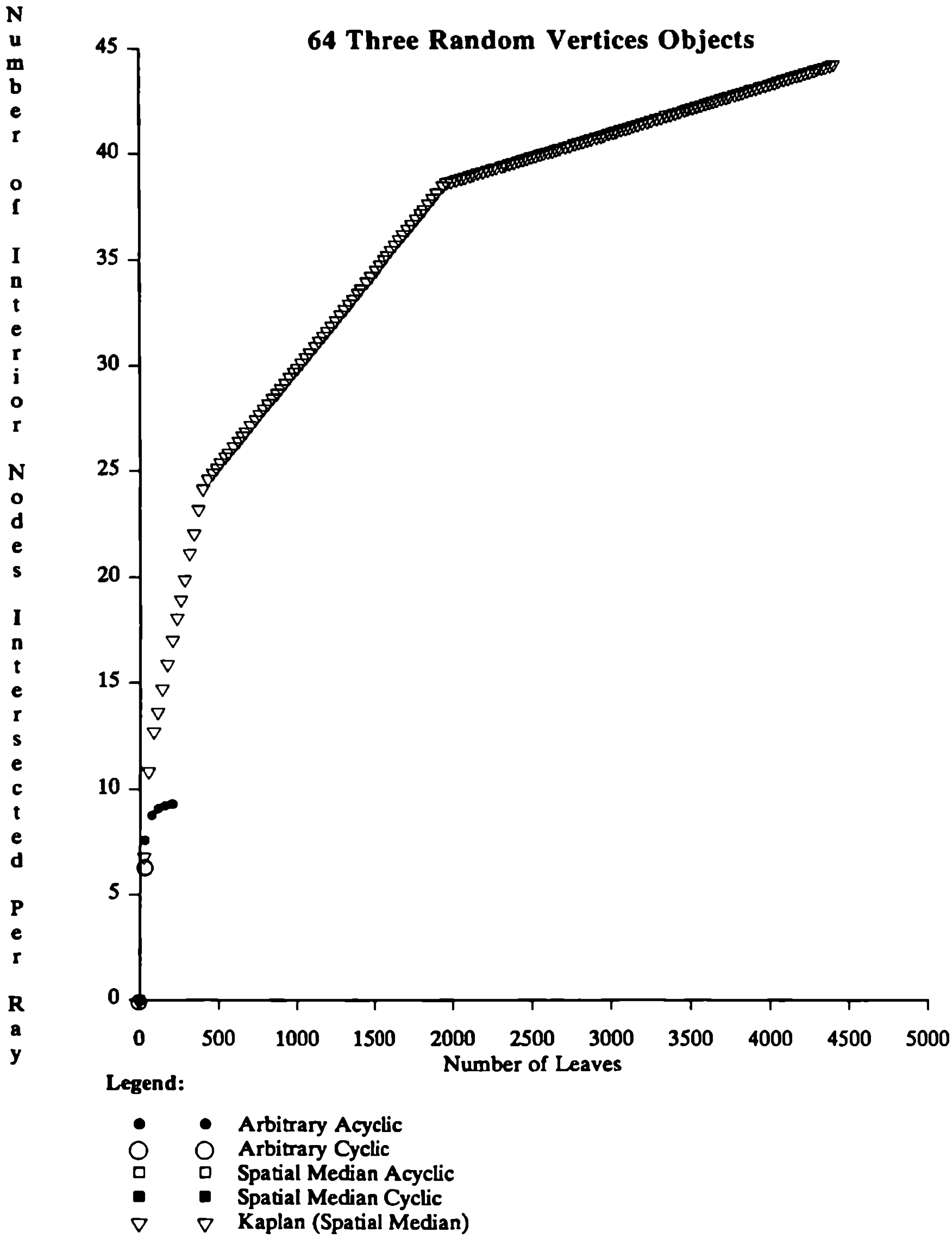
The following are the analogous graphs for number of interior nodes intersected.


8192 Small Spherical Objects
Number of Interior Nodes Intersected Per Ray
60
55
50
45
40
35
30
25
20
15
10
5
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)


8192 Large Spherical Objects
Number of Interior Nodes Intersected Per Ray
0
5
10
15
20
25
30
35
40
45
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

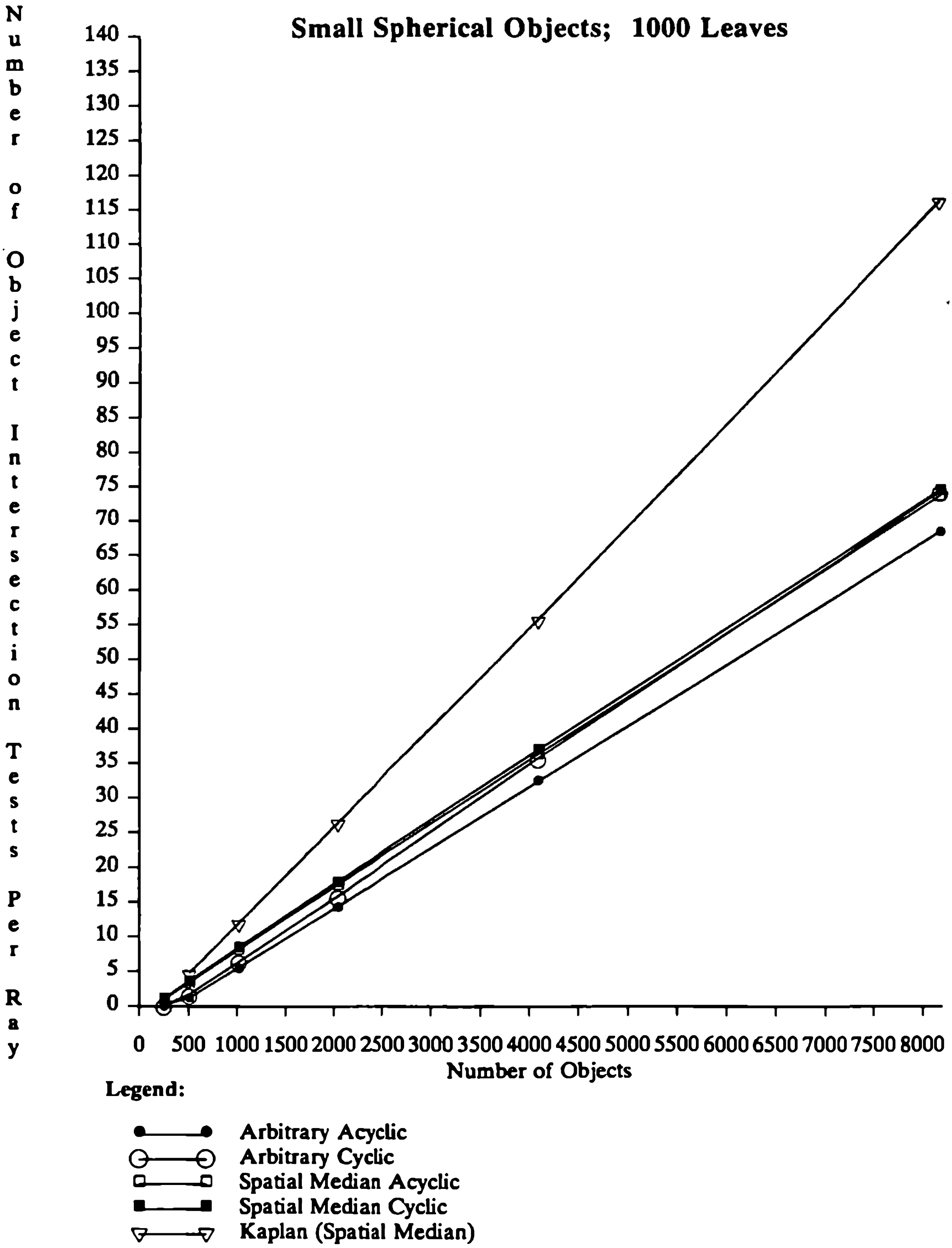

8192 Small Gaussian Objects
Number of Interior Nodes Intersected Per Ray
0
5
10
15
20
25
30
35
40
45
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

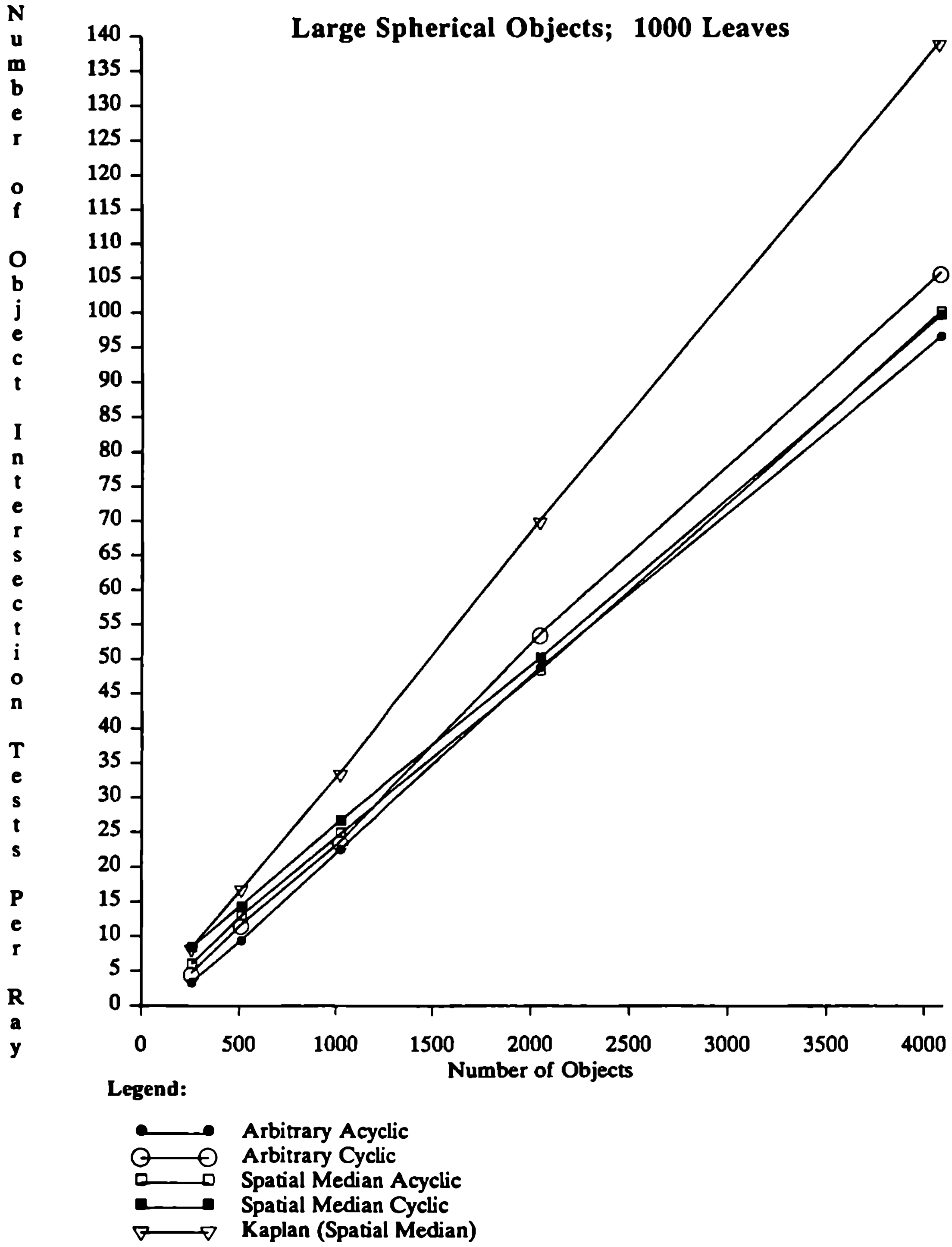

8192 Large Gaussian Objects
Number of Interior Nodes Intersected Per Ray
0
5
10
15
20
25
30
35
500
1000
1500
2000
2500
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

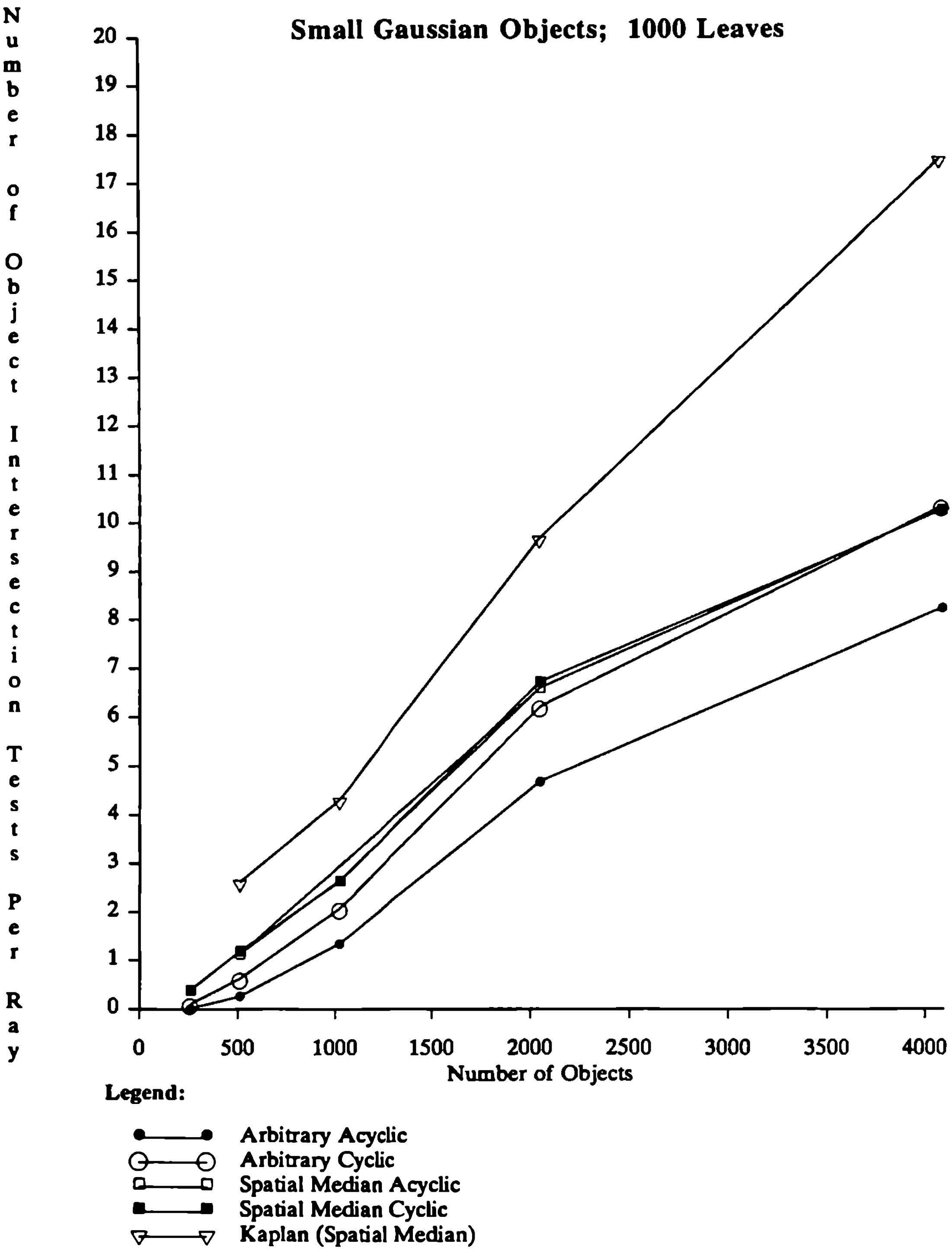

64 Three Random Vertices Objects
Number of Interior Nodes Intersected Per Ray
0
5
10
15
20
25
30
35
40
45
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Number of Leaves
Legend:
● ● Arbitrary Acyclic
○ ○ Arbitrary Cyclic
□ □ Spatial Median Acyclic
■ ■ Spatial Median Cyclic
▽ ▽ Kaplan (Spatial Median)

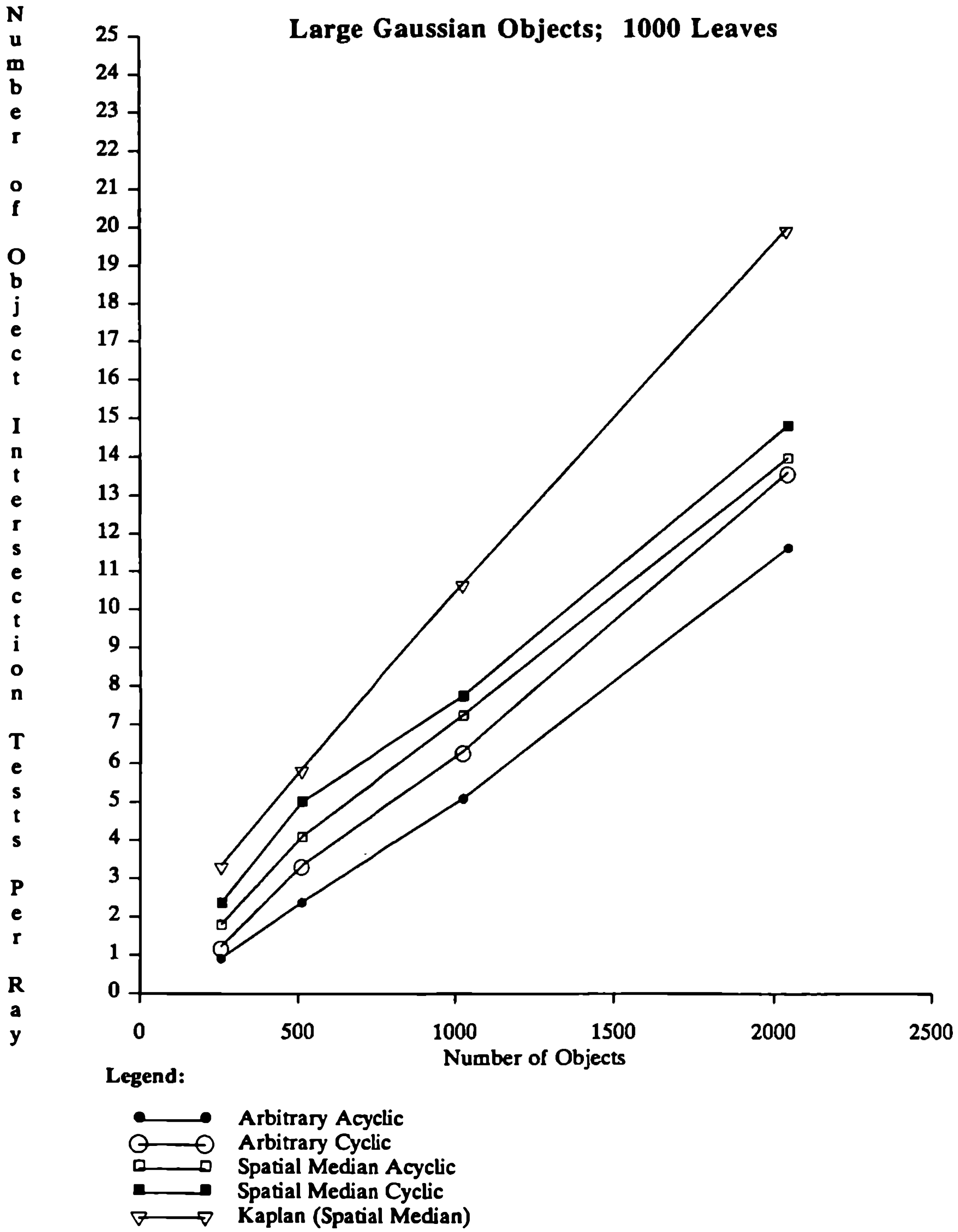
Appendix B
Varying the Number of Objects

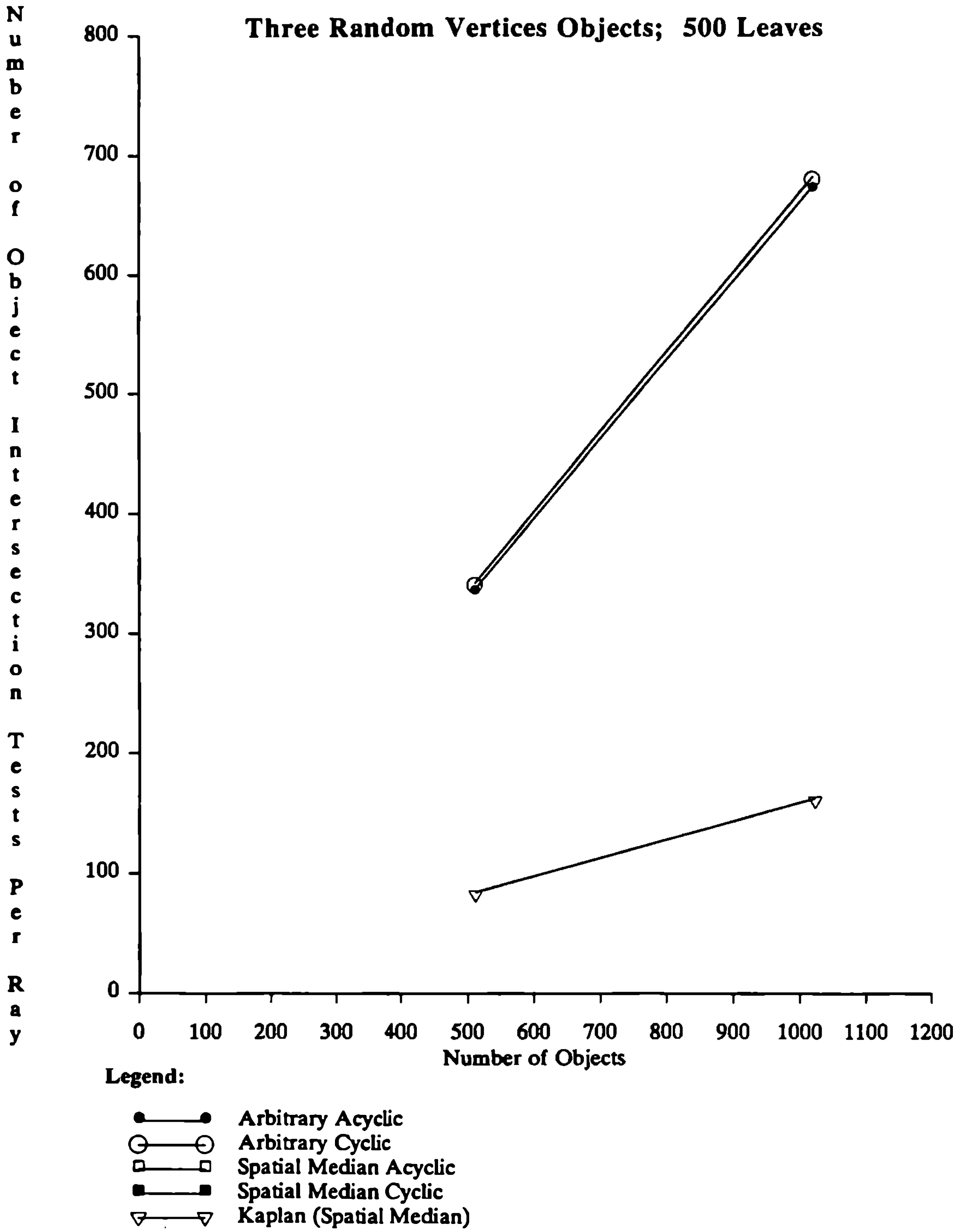
In order to get a grasp of how the number of objects affects performance, the following plots depict the number of object tests per ray, as a function of the number of objects comprising the scene. All graphs are for trees with 1000 leaves (empty plus non-empty), except for the three random vertices scene type, which is graphed at 500 leaves.


Small Spherical Objects; 1000 Leaves
Number of Object Intersection Tests Per Ray
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
Number of Objects
Legend:
●——● Arbitrary Acyclic
○——○ Arbitrary Cyclic
□——□ Spatial Median Acyclic
■——■ Spatial Median Cyclic
▽——▽ Kaplan (Spatial Median)


Large Spherical Objects; 1000 Leaves
Number of Object Intersection Tests Per Ray
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
0
500
1000
1500
2000
2500
3000
3500
4000
Number of Objects
Legend:
●——● Arbitrary Acyclic
○——○ Arbitrary Cyclic
□——□ Spatial Median Acyclic
■——■ Spatial Median Cyclic
▽——▽ Kaplan (Spatial Median)


Small Gaussian Objects; 1000 Leaves
Number of Object Intersection Tests Per Ray
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
500
1000
1500
2000
2500
3000
3500
4000
Number of Objects
Legend:
● Arbitrary Acyclic
○ Arbitrary Cyclic
□ Spatial Median Acyclic
■ Spatial Median Cyclic
▽ Kaplan (Spatial Median)

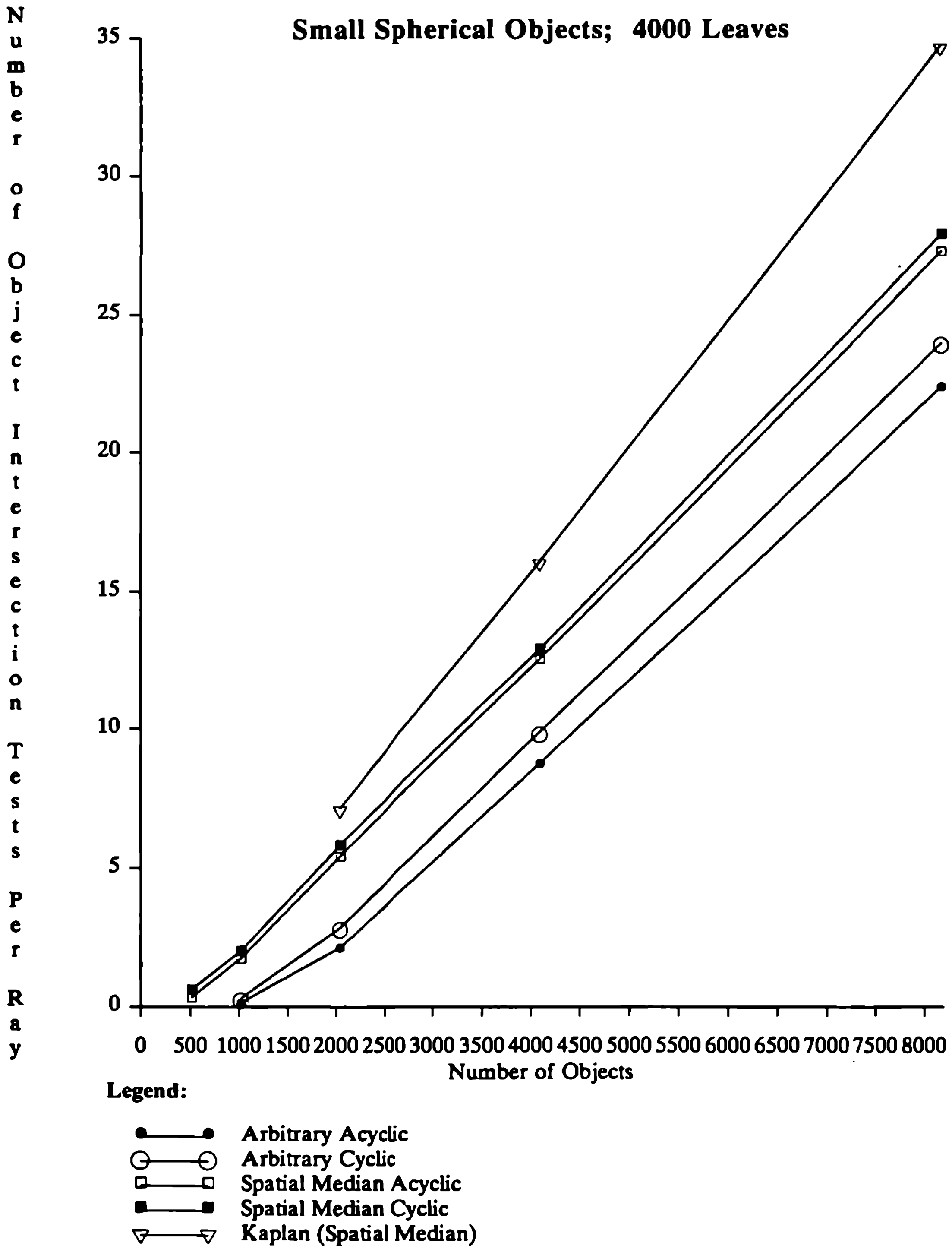

Large Gaussian Objects; 1000 Leaves
Number of Object Intersection Tests Per Ray
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
500
1000
1500
2000
2500
Number of Objects
Legend:
Arbitrary Acyclic
Arbitrary Cyclic
Spatial Median Acyclic
Spatial Median Cyclic
Kaplan (Spatial Median)

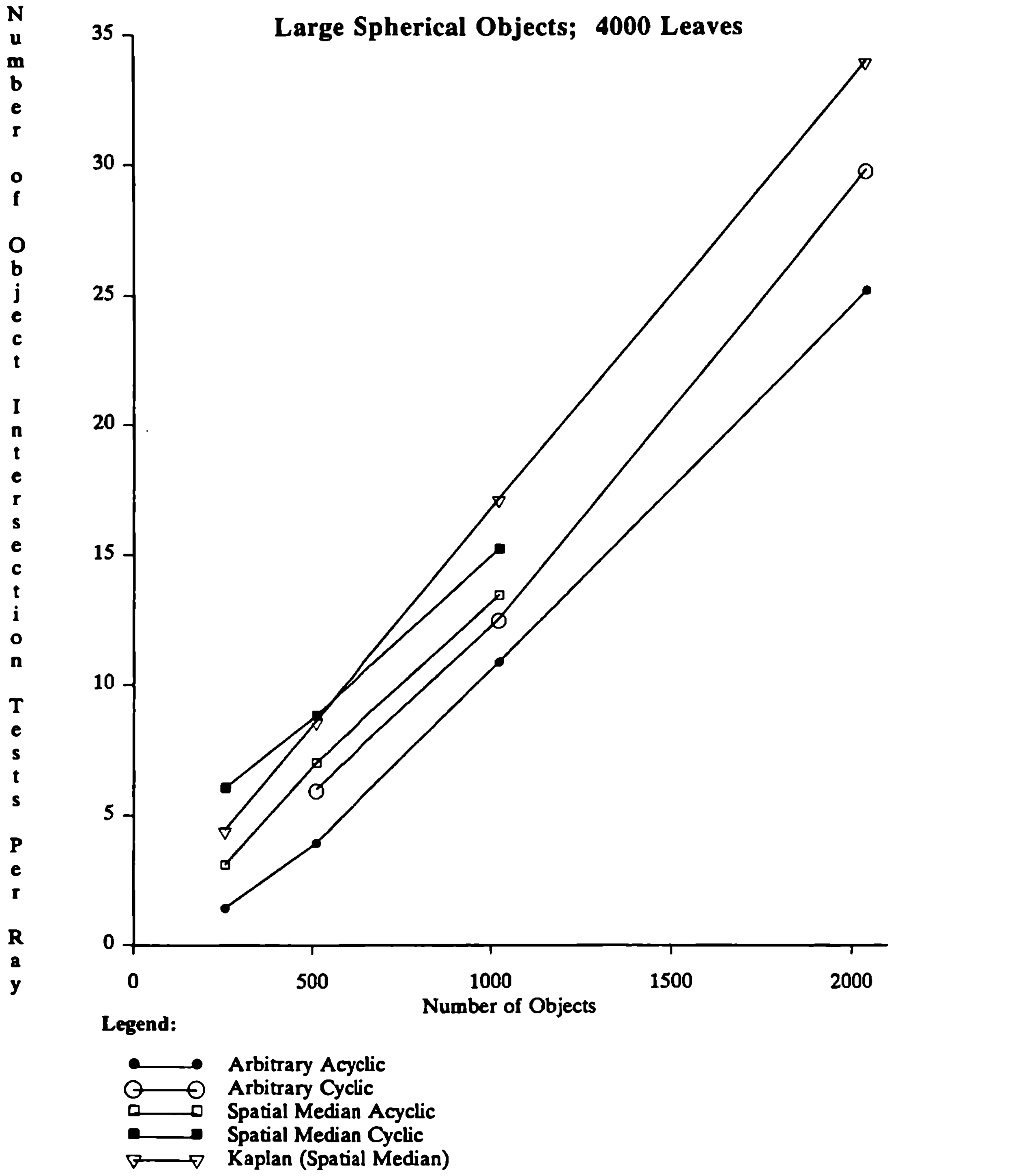


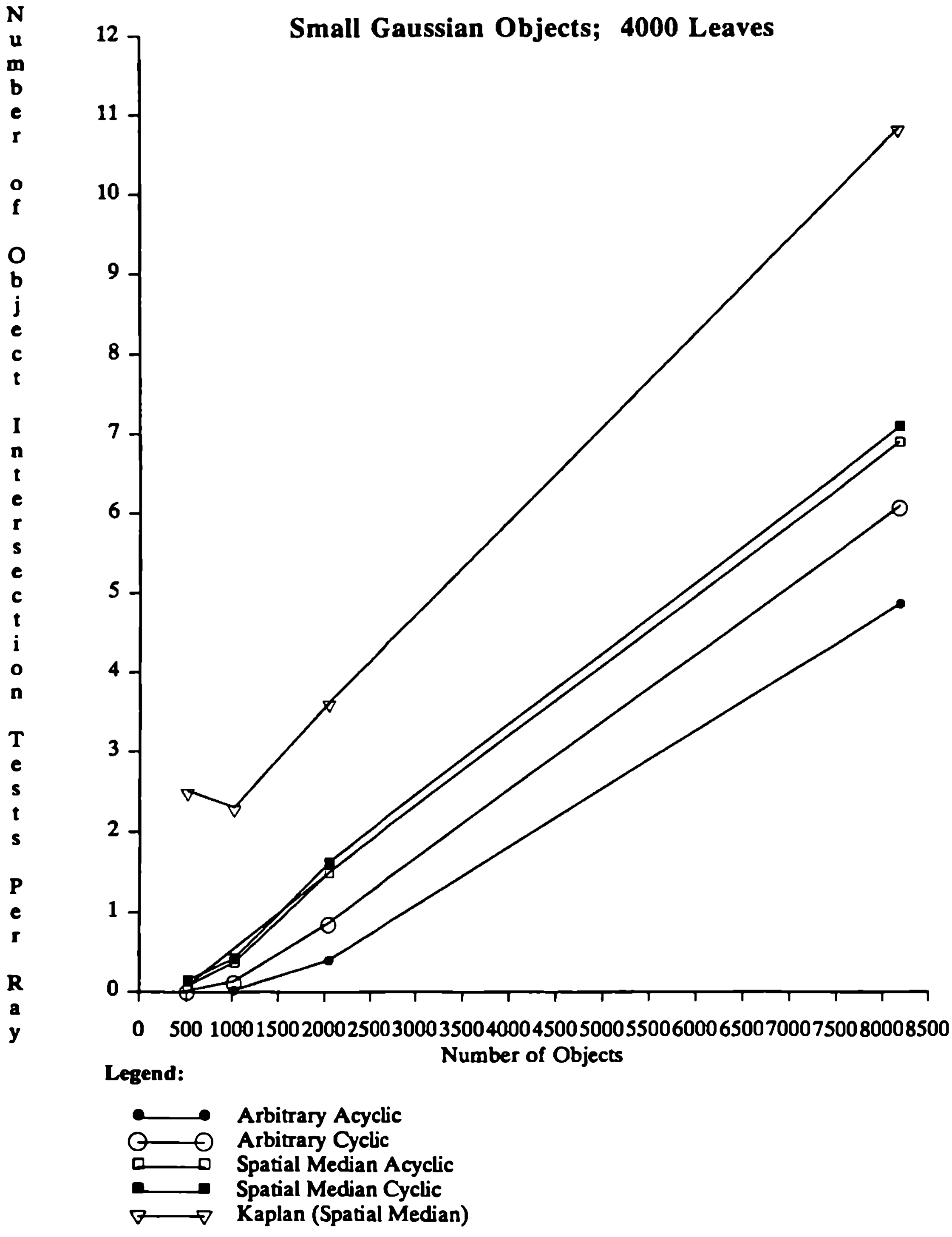
Three Random Vertices Objects; 500 Leaves
Number of Object Intersection Tests Per Ray
800
700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
Number of Objects
Legend:
Arbitrary Acyclic
Arbitrary Cyclic
Spatial Median Acyclic
Spatial Median Cyclic
Kaplan (Spatial Median)

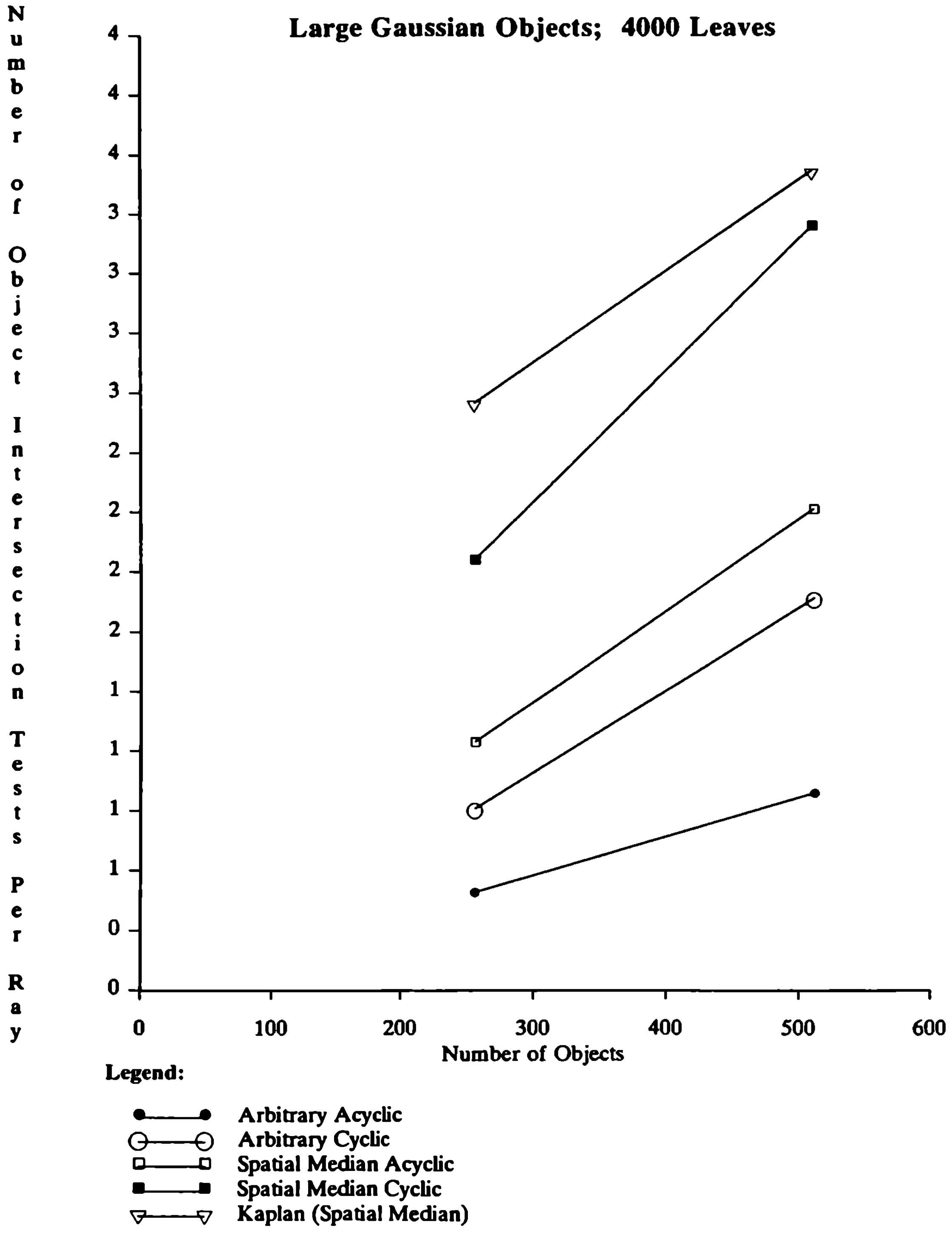
Large Trees

The following are graphs of 4000 leaves (empty plus non-empty). The three random vertices scene type does not have any data for 4000 leaves, and hence has no graph.


Small Spherical Objects; 4000 Leaves
Number of Object Intersection Tests Per Ray
0
5
10
15
20
25
30
35
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
Number of Objects
Legend:
Arbitrary Acyclic
Arbitrary Cyclic
Spatial Median Acyclic
Spatial Median Cyclic
Kaplan (Spatial Median)


Large Spherical Objects; 4000 Leaves
Number of Object Intersection Tests Per Ray
35
30
25
20
15
10
5
0
0
500
1000
1500
2000
Number of Objects
Legend:
●——● Arbitrary Acyclic
○——○ Arbitrary Cyclic
□——□ Spatial Median Acyclic
■——■ Spatial Median Cyclic
▽——▽ Kaplan (Spatial Median)


Small Gaussian Objects; 4000 Leaves
Number of Object Intersection Tests Per Ray
0
1
2
3
4
5
6
7
8
9
10
11
12
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
Number of Objects
Legend:
Arbitrary Acyclic
Arbitrary Cyclic
Spatial Median Acyclic
Spatial Median Cyclic
Kaplan (Spatial Median)


Large Gaussian Objects; 4000 Leaves
Number of Object Intersection Tests Per Ray
4
4
4
3
3
3
3
2
2
2
2
1
1
1
1
0
0
0
100
200
300
400
500
600
Number of Objects
Legend:
Arbitrary Acyclic
Arbitrary Cyclic
Spatial Median Acyclic
Spatial Median Cyclic
Kaplan (Spatial Median)