First Year Report

Simon McPartlin,
Department of Computer Science,
University of Edinburgh,
James Clerk Maxwell Building,
King's Buildings,
Edinburgh, EH9 3JZ
Scotland.
Email: sdnm@dcs.ed.ac.uk

November 1, 1993

Contents

1 Introduction

This report describes the work undertaken during my first year of study towards a Ph.D. Section 2 briefly reviews the topics I have looked at during this time. Section 3 describes in more detail current research in the areas I have chosen to concentrate on, Binary Space Partitions and Computational Geometry. Section 4 details work I have currently completed within these areas, with Section 5 outlining the aims and goals for the next two years.

2 Topics

The first year of study has been spent looking at two largely unrelated topics within the field of Computer Graphics. The change in direction, from the study of theoretical aspects of Computer Graphics to data structures for Computer Graphics, is the result of a number of factors which I will attempt to outline.

2.1 Theoretical Aspects of Computer Graphics

The semantics of rastering primitive graphic objects, such as lines, may be understood on several levels of abstraction. To the implementor of raster algorithms this understanding will be on a pixel level, while for most users the understanding will be on a higher abstract level. One of the largest research areas within "theoretical" computer graphics is concerned with this relationship.

The level of abstraction chosen for current research into the specification of computer graphics systems has been that of an abstract machine with abstract drawing primitives. This level of detail is clearly of interest to both users and implementors of such packages, but is insufficient in many respects. As E. Fiume observes, "Nowhere in any of the standards document is there a precise characterisation of an image, nor the effect of rendering a scene in a particular way. Thus the true semantics of a graphics system, the image produced in response to an input scene, is left entirely up to each implementation", [Fiume, 1989]. Current research into specification of graphics systems, [Mallgren, 1983, Gnatz, 1984, Carson, 1984, Duce, 1988, Reade, 1992], has failed to address this problem.

In contrast, research into rendering primitive graphic objects, in particular lines, has focused on the lowest level of abstraction – pixel level – in an attempt to solve this problem, [Fiume, 1987, Luby, 1988, Bresenham, 1988, Fiume, 1989, Eker, 1990].

A major aspect of this research, the rendering of lines, is described by M. Luby; "The motivating question is how to properly render straight line segments on a graphics screen. As a first approximation, the screen can be thought of as the Euclidean plane. However the screen is really a rectangular array of pixels, each of which can be either on or off, and a straight line segment is displayed as a path of on pixels. Thus, Euclidean geometry is approximated by a grid geometry. The essential difference between Euclidean geometry and a grid geometry is that in the Euclidean plane there is *infinite precision* and in a grid geometry there is only *finite precision*".

Line segments within the Euclidean plane exhibit a number of important properties which should ideally be inherited by the rendered line. One very important property of Euclidean geometry is that lines are *exactly straight*, i.e. the slope of a line is the same at every point along the line. In any rendering the slope of the resulting line at any particular point is either horizontal, vertical or diagonal. Thus, with the exception of horizontal, vertical and diagonal lines, preserving this property is not possible. The fundamental differences between the continuous Euclidean plane and the discrete screen grid are clear.

These differences highlight a significant problem surrounding the specification of graphics systems. Often the user of a graphics system will be working on a model defined in continuous Euclidean space, and viewing an image defined in a discrete grid space. While a specification of the graphic system can tell the user about the defined model, and possibly whether distinct models define the same scene, they can tell the user little about the image without the specification explicitly describing, at a pixel level, each of the rendering algorithms used, and thus resulting in a specification which is little more than a description of an implementation.

An additional problem is observed by Fiume, when considering line drawing algorithms which are non-monotonic[1]. It is demonstrated that such algorithms are unsuitable for applications that require all model specifications denoting the same scene to produce the same image, [Fiume, 1989]. It is further proved that no line drawing algorithm can preserve monotonicity in addition to satisfying fairly loose "straightness"and "width" properties. Thus the problem of producing a specification for a graphic system which satisfactorily describes the relationship between scenes and images appears not to be feasible.

It was felt at this point that further research into this area was likely to be fruitless and that enough time was left to pursue a new topic.

It should be noted that I have given only a very brief outline of the work undertaken in this area – it seems appropriate to spend the major part of this report describing the current topic of study.

2.2 Binary Space Partitions and Computational Geometry

After looking at various areas of current graphics research, the topic of data structures for geometric objects was settled on. More specifically I decided to focus on a particular data structure, the binary space partition. Current research in this area covers many disciplines within Computer Science including algorithm design, computational complexity, and computational geometry, in addition to application research. Having worked extensively with the data structure during a fourth year project, it was felt that the combination of theory and practice, together with a wide range of open questions, could provide the foundation for an interesting and rewarding research project.

[1] A line drawing algorithm is *monotonic* if the pixels selected for a line $L' \subseteq L$ are a subset of those selected for the line L.

3 Background

The majority of the sections on BSP trees is taken directly or paraphrased from the following sources: Section 3 and 3.1.3, [Naylor, 1990a]; Section 3.1.1, [Foley *et al.*, 1990]; Section 3.1.2, [Paterson and Yao, 1989]; Section 3.1.4, [Naylor, 1993]; Section 3.1.7, [Naylor, 1992b].

3.1 Binary Space Partitions

Algorithms for geometric-based computing (computer graphics, geometric modelling, robotics, etc.) depend fundamentally upon the representation used for geometric sets; as in all areas of computer science, the data structure and the algorithms are inseparable. An important subclass of geometric sets is polytopes: the dimension-independent class whose 3D members are polyhedra and 2D members are polygons. A promising new alternative for representing polytopes is the *binary space partitioning tree* (BSP tree), [Fuch *et al.*, 1980, Foley *et al.*, 1990].

The principal qualities that are considered desirable for any representation are space and time efficiency as well as algorithmic simplicity. While space and time efficiency have traditionally been the metrics receiving the greatest attention, the complexity of the algorithms, which affects the difficulty of implementation and the suitability of designing special hardware, can be just as important.

Given a representation, one must then devise algorithms to provide whatever operators are desired. For geometric sets, affine transformations and boolean set operations are fundamental. Of equal importance is the rendering operator: the mapping of geometric sets defined in continuous space to images defined in a 2D discrete space.

Currently, the two dominant representational schemes are constructive solid geometry and boundary representations. Constructive solid geometry defines a geometric model as an expression on primitive objects using affine transformations and boolean set operations for operators. Referencing of common subexpressions is also used, providing 'instancing'. Its combinatorial representation is a directed acyclic graph, and it is considered to be 'unevaluated', i.e. it is an expression defining a value, but it is not the value itself. In contrast, boundary representations are evaluated representations. They represent the boundary of the set explicitly; however the interior and exterior must be inferred. Their combinatorial representation is a hierarchy of lists where each level corresponds to sets of a particular dimension (faces, edges, vertices), and arcs between levels encode geometric incidence.

BSP trees, currently limited to representing linear polytopes, provide an alternative evaluated representation, [Thibault, 1987, Naylor *et al.*, 1990, Naylor, 1990a]. Compared to boundary representations, they appear at this point to provide a notable improvement in algorithmic complexity. The algorithms that operate on BSP trees are simple, recursive, of similar form, and dimension-independent. While boundary representations require unrelated and relatively complicated algorithms for such operations as spatial search, set operations and visible surface determination, all of these are integrated and simple using BSP trees. Although meaningful comparisons in space and time efficiency are presently difficult to make, achieving interactive speeds using a BSP tree-based solid modeller can be accomplished with relative ease, [Naylor, 1990b, Naylor, 1992a]. Current applications employing BSP trees are wide ranging and include:

ray tracing, [Naylor and Thibault, 1986, Thibault, 1987, Lin, 1987]; radiosity, [Campbell and Fussell, 1990, Campbell, 1991]; shadow-generation, [Chin and Feiner, 1989, Chin and Feiner, 1992]; image representations, [Radha *et al.*, 1990, Radha *et al.*, 1991a, Radha *et al.*, 1991b, Naylor, 1992b]; geographical information systems, [Ooterom, 1990]; and motion planning, [Tokuta, 1991].

3.1.1 Informal Description

The binary space partitioning (BSP) tree data structure, was developed to provide an efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary view point. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the view point changes, but the objects do not.

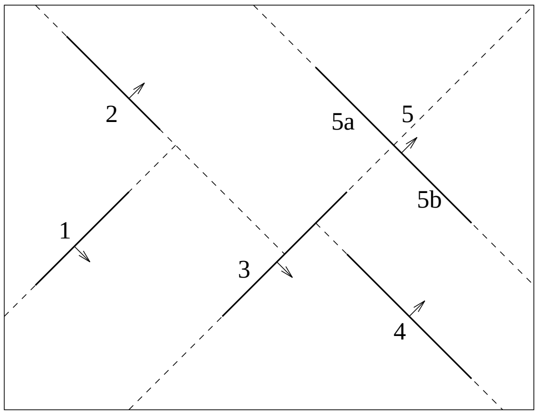
The data structure is based on the observation that a polygon will be scan-converted correctly (i.e., will not incorrectly overlap or be incorrectly overlapped by other polygons) if all polygons on the other side of it from the viewer are scan-converted first, then it, and then all polygons on the same side of it as the viewer.

```
Build_Tree( polygons ) -> B_tree
{
  if polygons == NULL then
     bsp_tree = NULL;
  else {
     root = Select_Hyperplane( polygons );
     (inside, on, outside) = Partition( polygons, root );
     bsp_tree = Make_Node( on );
     bsp_tree->left = Build_Tree( inside );
     bsp_tree->right = Build_Tree( outside );
  }
  return bsp_tree;
}
```

Figure 1: BSP tree construction algorithm.

The BSP tree's root is a polygon selected from those to be displayed. The root polygon is used to partition the environment into two half-spaces. One half-space contains all remaining polygons in front of the root polygon, relative to its surface normal; the other contains all polygons behind the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane and its front and back pieces are assigned to the appropriate half-space. One polygon each from the root polygon's front and back half-space become its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. Pseudocode for the tree building process is shown in figure 1; figure 2 illustrates a tree.

A BSP tree may be traversed in a modified in-order tree walk to yield a correct priority-order polygon list for an arbitrary view point. Consider the root polygon. It divides the remaining polygons into two sets, each of which lies entirely on one side of the root's plane. Thus, the algorithm needs to only guarantee that the sets are displayed in the correct relative order to ensure both that one set's polygons do not interfere with the other's and that the root polygon is displayed properly and in the correct order relative to the others. If the viewer is in the root polygon's front half-space, then the algorithm must first display all polygons in the root's rear half-space (those that could be obscured by the root), then the root, and finally all polygons in the front half-space (those that could obscure the root). Alternatively, if the viewer is in the root polygon's rear half-space, then the algorithm must first display all polygons in the root's front half-space, then the root, and finally all polygons in the rear half-space. If the polygon is seen on edge, either display order applies. Each of the root's children is recursively processed by this algorithm. [Gordon and Chen, 1991] describes an alternative, and more efficient, image-space display algorithm.



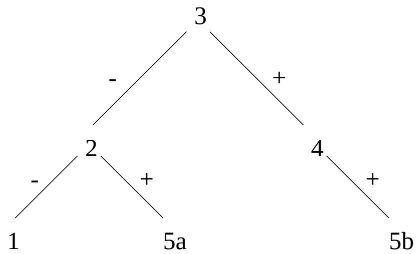


Figure 2: BSP tree.

3.1.2 Formal Definition

Let H be a collection of hyperplanes, organised as a binary tree and labelled as illustrated in figure 3. H defines a binary partition P, in which $\Re^d$ has been recursively partitioned, first by the root hyperplane H_ϵ, and then by the subtrees rooted at H_0 and H_1. Each hyperplane $H_i \in H$, $i \in \{0,1\}^*$, partitions $\Re^d$ into two open half-spaces H_i^- and H_i^+, together with H_i itself; these hyperplanes are referred to as *partitioning hyperplanes*. For each node v in the tree, the convex region defined by the intersection of all the open half-spaces defined at the (proper) ancestor nodes of v is denoted by $R(v)$, and the intersection of $R(v)$ with the hyperplane H_v is denoted by H_v^s and referred to as the *subhyperplane* at node v. The components of a partition P are thus the convex regions $R(v)$ for each leaf node v of the tree, together with the subhyperplanes H_v^s for every internal node v.

Given a collection of faces Γ (i.e. polytopes of dimension $d-1$ or less), a binary partition P provides a natural decomposition of Γ. A binary space partition of Γ is defined to be a restricted form of binary partition P in which the set of subfaces $\Gamma \cup R(v)$ for all leaf nodes v of P is empty, and thus in which subfaces lie within the hyperplanes H_v of one or more of the internal nodes v of P.

The size, $|P|$, of a BSP P is defined as the number of internal nodes. The partition complexity of Γ is defined to be the minimum size $|P|$ over all possible BSPs P of Γ.

It can be observed that at each internal node v the influence of the partitioning hyperplane H_v is restricted to the region $R(v)$. Thus, rather than partitioning $\Re^d$, the BSP tree rooted at v partitions $R(v)$, a subregion of $\Re^d$. By observing that a BSP tree induces a partition over a *particular* region of space, a more intuitive and recursive definition of a BSP tree can be achieved: A BSP tree represents a region of space R recursively by partitioning R into two subregions, using a partitioning hyperplane, which are represented in turn by BSP trees. The recursion is terminated when the region R is sufficiently simple, e.g. homogeneous.

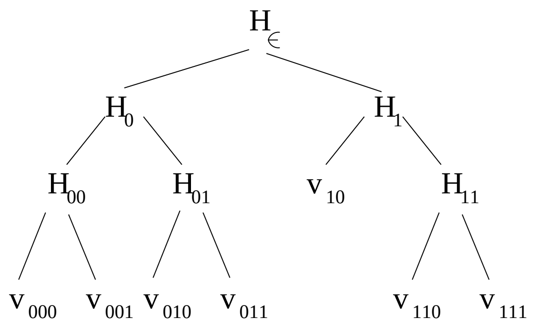


Figure 3: Labelled BSP tree.

3.1.3 Representation of Geometric Sets

Often the motivation for inducing a partitioning on a space is to model polytopes. To define polytopes, the attribute *classification* = {*interior, exterior*} can be assigned to each cell of the partition, i.e. leaf of the BSP tree. Figure 4 shows a BSP tree and the polygon it defines.

Defining a polytope as the closure of a union of (convex) cells allows a single tree to represent the most general notion of linear geometric sets: they can have concavity, any genus, any number of connected components, non-manifold boundaries, and unbounded interiors.

The boundary of the set lies in the partitioning subhyperplanes, and can be obtained either through neighbourhood operations or can be represented explicitly by polygons, each stored at the internal node whose subhyperplane contains them. A unified approach can be achieved by using $d-1$ dimensional trees at nodes to explicitly store the boundary of a d dimensional tree, as required.

A closed boolean algebra on BSP trees, i.e. *bsp tree <set op> bsp tree* $\rightarrow$ *bsp tree*, has been recently completed, [Naylor *et al.*, 1990].

Partitioning trees are alone among the principal methods for representing geometry in combining the representation of a set with the geometric search structure required for efficient spatial operations such as set operations and visibility.

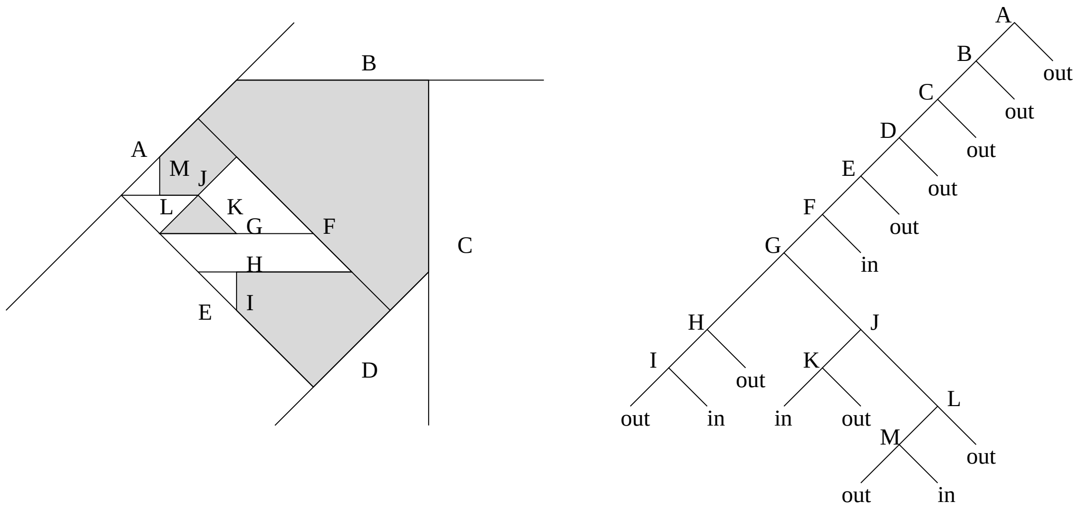


Figure 4: Polygon and BSP tree.

3.1.4 Representation of Functions

A more generalised view of partitioning trees sees them as representations of functions whose domain and range are continuous spaces of finite dimensions d_1 and d_2 respectively: $f : X \in S^{d_1} \Rightarrow Y \in S^{d_2}$. The partitioning tree partitions the domain into a hierarchical collection of subdomains. Within each subdomain a value-continuous function f_i defines the value of f within the subdomain. All points in S^{d_1} at which f is value-discontinuous are contained within partitioning subhyperplanes.

A partitioning tree also provides a structure enabling a hierarchical representation of f. As an example, consider modelling polytopes. At the cells of the partitioning, each f_i is a boolean valued constant function indicating whether or not the cell is in the set. The polytope is the set of points $P = \text{closure}(c_i \mid c_i$ is an in-cell). Thus f is the characteristic function f_χ for the set P.

3.1.5 Complexity and Efficiency

A partitioning tree may be interpreted as specifying a program for exploring the structure induced on a space (e.g. by objects). Consequently, there are many trees which represent the same spatial structure but provide different searches of the space. The issue of efficiency for operations using partition trees is thus transformed into the issue of constructing partition trees providing efficient searches of the space, as required by the operations.

Results from combinatorial geometry, [Edelsbrunner, 1987], provide the first measures of tree size. It is shown that, in the worst case, n hyperplanes can partition d dimensional space into $\Theta(n^d)$ non-empty d dimensional cells.

Recent complexity results, [Paterson and Yao, 1989], [Paterson and Yao, 1990a], [Paterson and Yao, 1990b], [Paterson and Yao, 1992], [d'Amore and Franciosa, 1992], have reduced these worst-case bounds when using BSP trees to represent polytopes. It has been shown that the partition induced on two dimensional space by n edges can be represented by a BSP tree of size $\Theta(n \log n)$, with a result of $\Theta(n^2)$ for three dimensions and n polygonal faces.

These worst-case result describe the partition of d dimensional space by hyperplanes, which by definition span all of d space, whereas the actual partition is due to subhyperplanes spanning a subregion of this space. Indeed, empirical results indicate that the expected size of partitioning trees for representing polytopes in three dimensions is closer to $O(n \log n)$ than the $O(n^2)$ bound, [Naylor, 1993]. An examination of the apparent discrepancy between the worst-case complexity and empirical results, may be found in Section 4.2.

It is put forward by Naylor that generating an optimal tree, like most combinatorial problems, appears intractable: "Our problem can in fact be considered to be harder than those problems which require enumeration of the power set since we have ordering: thus, for n faces lying in n unique hyperplanes, there are at least $n!$ different trees." Current algorithms for efficient tree construction are based on sampling heuristics and expected case models, [Naylor, 1993, Krishnaswamy *et al.*, 1990]. A discussion of current approaches to tree construction follows in Sections 4.2.2.

3.1.6 Set Operations

Algorithms for implementing set operations between BSP trees are described in [Naylor *et al.*, 1990]. The basis for such operations is the merging of the two partitions described by the trees, as illustrated in figure 5, with the semantics of the set operation determining the attributes for each cell.

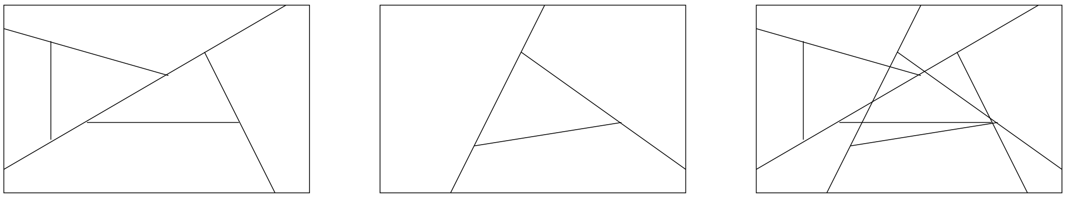


Figure 5: Merging partitions

Merging of trees is achieved recursively by partitioning one tree, say $T1$, with the root partitioning hyperplane of the other, say T_2. The result will be two trees T_1^- and T_1^+, defined on the same subspaces as T_2's left and right subtrees. The merging then proceeds recursively with T_1^- and T_2's left subtree, and T_1^+ and T_2's right subtree. When a cell, i.e. leaf, is reached, an operation dependent *cell <op> bsp tree* routine is used. For set operations on trees representing characteristic functions, this amounts to simply selecting either the cell or the tree, possibly complemented. For more general trees the *cell <op> bsp tree* attribute merging routine may be more complex.

The number of cells in the merged partition gives an upper bound on the size of the resulting tree. It should be noted, though that the worst-case bound on tree size, $|T_1||T_2|$, can occur only when each cell of T_1 intersects each cell of T_2. Of greater interest is the expected case. As with all BSP construction processes condensation of cells is possible, resulting in trees possibly smaller than the merged partition. It can be observed that at each stage in the merging process there is a choice as to which tree to partition. The problem of choosing the tree, at each stage, which will result in the smallest output tree has received little attention, but is closely related to the construction of optimal trees, Section 4.2.1.

3.1.7 Image Tree

As we have seen, a BSP tree can be used to generate a total visibility priority ordering on the faces of a scene with respect to any view point. This ordering can subsequently be used to generate continuous two- or three-dimensional BSP tree representations of the image visible from a particular view point, [Naylor, 1992b]. These BSP trees are called *image trees*.

The two-dimensional image tree can be constructed by considering each face in the front-to-back order, and:

1. Projecting the face onto the 2D projection plane and letting the attributes of the face be its colour.

2. Forming the two-dimensional space union of the faces in priority order: *image = Union Sets(image, face)*, where the attributes of the image take precedent over those of the face.

Thus the faces are projected to two-dimensional regions of the image plane, and the effect of higher priority faces occluding lower priority faces is achieved by letting the attributes of the image tree take precedence over those of the current face being "added" to the image. So visible surface determination has been reduced to ordered set operations on polyhedral faces. Non-refractive transparency can be supported by using a "merge attributes" method that blends colours according to their opacity.

In addition the set of visible edges can be generate by performing a closure operation which determines for each subhyperplane which subsets have a heterogeneous neighbourhood. This then provides a visible line algorithm as an additional step after the visible surface algorithm.

A three-dimensional image tree can be generated in several ways. One method first transforms each face into 3D post-perspective screen-space. Then each face is considered to define a polygonal cylinder, bounded at the near side by the face boundary, with sides orthogonal to the xy-plane, see figure 6. The union operation is now on 3D cylinders, and may be carried out in any order.

The algorithm can also be performed in model-space, in which case the cylinders are instead cones whose conical-vertex is the centre of projection. This approach has been used for shadow generation, a problem very closely related to visible surface determination, [Chin and Feiner, 1989, Chin and Feiner, 1992]. The resulting model-space image tree can then be transformed by the viewing transformation into screen-space.

An immediate advantage of using continuous space representations is that affine transformations can be applied with ease. Image trees can also be used in subsequent continuous space operations. Therefore, compositing operations can be performed on 3D images.

Set difference can be used for masking, and since our 3D image trees are of the same data type as any other of our geometric sets, the previously developed set operations can give us compositing immediately. Using 3D instead of 2D images frees the compositing from being simply a layering of images on top of each other, allowing the images to be interleaved.

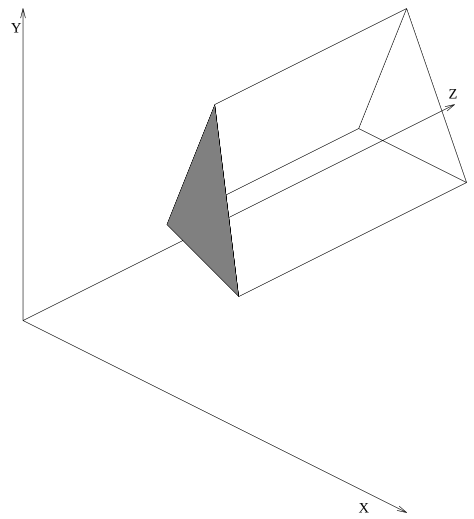


Figure 6: Cylinder defined by polygonal face.

3.2 Computational Geometry

Computational geometry [Mehlhorn, 1984, Edelsbrunner, 1987, Preparata and Shamos, 1988], is the branch of computer science that studies algorithms for solving geometric problems. In order to gain a greater understanding of space partitions and their complexity a study of related data structures and techniques from this field was undertaken.

General multidimensional search trees and dynamic search structures were first explored, Sections 3.2.1 and 3.2.2.

The next area looked at was arrangements of hyperplanes, an alternative and well studied spatial partitioning technique. A description of this technique is given in Section 3.2.3. Its relationship to the binary space partition is discussed in Section 4.2.

Other areas of research were studied, including general geometric algorithm techniques, Section 3.2.4, and hidden surface removal algorithms based on spatial partitions, Section 3.2.5.

3.2.1 Multidimensional Search Trees

Binary search trees are used to store data, in a binary tree, ordered with respect to a one-dimensional key, such that the following property is satisfied:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $\text{key}[y] \leq \text{key}[x]$. If y is a node in the right subtree of y, then $\text{key}[x] \leq \text{key}[y]$.

Viewed geometrically, binary search trees represent a partitioning of one dimensional space using the key values as partitioning hyperplanes. Since each key is of dimension 0, and thus we have no intersecting partitioning hyperplanes, a total ordering on the data can be achieved[2], with the binary search tree describing a search structure for the partitioned space. The search structure can be transformed using *rotation* operations to provide an alternative search, without altering the partition or data represented. Such rotation operations can be used to ensure that the tree is balanced.

Binary search trees of this form can be used to store data of any dimension by associating a one dimensional key with each item. For example, points in two dimensions could be stored by considering the x-coordinate as the key on which the data should be ordered. Of course the problem is that a binary search tree only provides help in searching within the one dimensional space occupied by the keys.

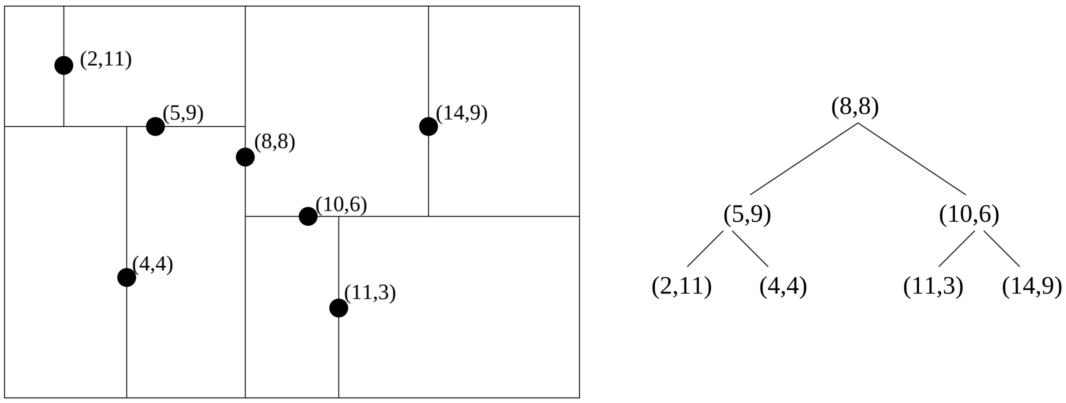


Figure 7: Two dimensional k-d tree.

k-d trees offer an alternative method for storing points in k-dimensional space. The essence of this data structure is the association of k key's with each data item, with each layer of the tree ordering the data with respect to a particular key. For example a 2-d tree may partition two dimensional space with respect to the x-coordinates of the points at the root, followed by the y-coordinates at the subtree roots, figure 7. A k-d tree thus performs a search of a k dimensional space. As with the binary search tree, rotation operations can be used to alter the search tree, though also the partition of space, while still preserving the correct tree ordering of the data.

The k-d tree can be seen to perform a one dimensional search on k axes cyclically. The fact that geometrically each partitioning of space appears to be in k dimensions is a result of *hardwiring* the orientation of each partitioning plane into algorithms using the tree, i.e. at the root of the tree algorithms must know that the data is partitioned by a plane parallel to the y-axis, leaving the data to define the final degree of freedom.

The benefits resulting from using hyperplanes of fixed orientation may not be obvious, so consider a k-d tree which partitions space using hyperplanes of arbitrary orientation.

Let the root of the k-d tree be a k dimensional point x, with the remaining points partitioned by a hyperplane $h(x)$ passing through x. Let x_i be a point lying inside hy-

[2]This assumes that keys are unique. If not, then it usual to arbitrarily order the data with equal key's.

perplane $h(x)$, and let x_j be a point lying outside hyperplane $h(x)$. We clearly cannot in general deduce anything about the position of x_j or x with respect to $h(x_i)$, and similarly for x_i or x with respect to $h(x_j)$. The traditional *k-d* tree ensures that such deductions can be made by forcing all hyperplanes h, at a given level of the tree, to be parallel and hardwiring this information into tree algorithms. The resulting ability to induce a total order on the data allows tree restructuring to take place in a straightforward manner. The root can be altered without affecting the ordering of the data, in fact the root can be viewed as a *pivot* for the data, having no affect on the overall ordering.

The obvious drawback is when higher dimensional data, such as line segments, are to be stored. In general it is not possible to ensure that all partitioning planes are parallel to a fixed hyperplane in addition to featuring a line segment.

3.2.2 Dynamic Data Structures

An area which has received a great deal of attention in recent years is the design of data structures and algorithms which can efficiently represent various geometric structures in a dynamic fashion, see for example [Overmars, 1983], [Mulmuley, 1991c], [Mulmuley, 1991b], [Goodrich and Tamassia, 1991], and [Teillaud, 1992].

Current work has focused on two areas: generating dynamic construction algorithms for existing static search structures; and using properties of the geometric structure to design dynamic algorithms and data structures. The application of these techniques to the BSP tree is discussed in Section 4.3.1.

The generation of dynamic data structures from static data structures is achieved by applying a variety of techniques which can be categorised as local rebuilding, partial rebuilding, and global rebuilding.

Local rebuilding is the simplest and most efficient technique, maintaining the required properties of the data structure by making local changes after an insertion or deletion. The rotation operation for one dimensional binary search trees, Section 3.2.1, is an example of such a technique.

When local rebuilding techniques are not an option, or are very complex, the technique of partial rebuilding is useful. After insertion or deletion of objects from a structure, the required properties of the data structure may be violated. The basic idea of this technique is the complete rebuilding of the portion of the data structure causing the violation.

In many cases, local rebuilding may be an expensive operation. If the data structure can accommodate additions and/or deletions fairly easily, though possibly violating a property based on efficiency of representation, e.g. balance for tree structures, then *lazy* rebuilding can used. That is, a weaker efficient representation requirement can be used, with rebuilding being applied only when this property is violated, e.g. only rebuilding when the tree becomes significantly unbalanced. Obviously the structure will not permit queries as efficiently, but this may be offset by a reduction in the update time.

A lazy approach of this style is even more applicable to the technique of global rebuilding, with the complete data structure rebuilt periodically, or as its efficiency as a representation falls below a threshold.

Alternative techniques, using properties of the problem itself, are naturally more diverse. Efficient dynamic solutions exist for many multidimensional problems including range queries, point location in arrangements of hyperplanes, convex hull and Voronoi diagram construction, and hidden surface removal.

A significant amount of research has explored the use of randomisation in dynamic data structures, [Mulmuley, 1991c, Mulmuley, 1991b, Teillaud, 1992]. I intend to investigate this area further.

3.2.3 Arrangements of Hyperplanes

The partitioning technique that has received the greatest attention in computational geometry is that induced by an *arrangement of hyperplanes*.

An finite set H of hyperplanes in d dimensions partitions space into a number of cells, of dimension d, and lower dimensional faces[3], [Edelsbrunner *et al.*, 1986, Edelsbrunner, 1987]. This partition is called the *arrangement*, $\mathcal{A}(H)$, of H. Figure 8 illustrates such an arrangement with the d dimensional cells numbered, and the $d-1$ dimensional faces forming the boundary of the cells.

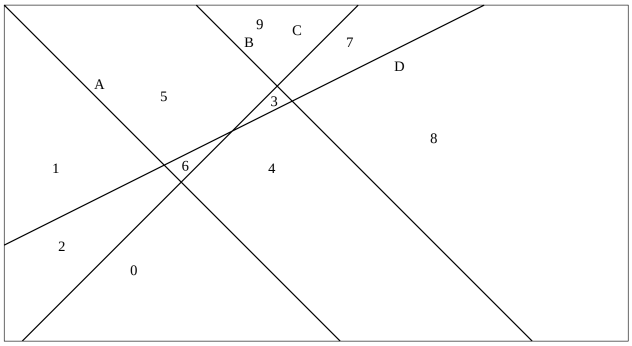


Figure 8: Arrangement of 4 hyperplanes in two dimensions.

Arrangements in two-dimensions have been studied from various mathematical points of view, while arrangements in higher dimension have received considerably less attention.

Much of the interest in arrangements in d dimensions is due to a dual correspondence to configurations of points in d dimensions. Many problems for sets of points are more conveniently solved for the corresponding arrangement. Examples include computing Voronoi diagrams and largest convex subsets of point sets.

Tight upper bounds on the number of faces in an arrangement exist. In an arrangement of n hyperplanes in d dimensions, the number of k dimensional faces is given by:

$$f_k^{(d)}(n) = \sum_{i=0}^{k} \binom{d-i}{k-i} \binom{n}{d-i} \text{ , for } 0 \leq k \leq d$$

If the dimension d is considered to be a constant, then we have the more manageable $f_k^{(d)}(n) = \Theta(n^d)$. Algorithms exist which can construct a representation of an arrangement in optimal $O(n^d)$ time, with faces and incidences between faces represented explicitly.

[3] The cells and faces of an arrangement are defined by sets of points lying in the same positions (i.e. on, inside or outside) with respect to each hyperplane defining the arrangement.

3.2.4 Coherence Paradigm

Coherence is one of the best known paradigms in computational geometry, and has found application in a number of geometric algorithms, e.g. the wide range of scan line based algorithms, for examples see [Foley *et al.*, 1990]. The application of coherence with regard to BSP tree based algorithms is discussed in Section 5. A general outline of the approach, from [Mulmuley, 1991a], follows:

"Consider the problem of constructing a certain geometric configuration G. One approach is to build G incrementally, with the computation guided by a certain coherence parameter ("time") t, whose value increases, monotonically, throughout the construction. The (partial) construction at time t will be denoted by $G(t)$, and, by convention, $G(\infty) = G$ and $G(-\infty)$ is either an empty configuration or one that can be computed by some other method. We call a value a of the coherence parameter t critical if $G(t)$ undergoes a change at $t = a$. In a combinatorial problem, the set of critical values is discrete. Hence $G = G(\infty)$ can be computed if

1. the critical values of t can be efficiently generated in an increasing order, and

2. the incremental change in $G(t)$ at critical values can be accomplished efficiently.

In an alternative problem the goal may not be the construction of $G(\infty)$, but rather to maintain the time dependent configuration $G(t)$ as t increases in a fixed interval. For example, in the case of hidden surface removal with respect to a moving view point, $G(t)$ may denote the visible scene at time t."

The range of higher dimensional problems that have been efficiently solved using this approach is relatively small. The main difficulties being the generation of critical values, as well as the updating of $G(t)$. Coherence has been used to efficiently provide hidden surface removal with respect to moving view points, [Mulmuley, 1991a], see Section 5, k-level Voronoi diagram construction, and convex hull generation.

3.2.5 Output-Sensitive Hidden Surface Removal

Current research into the hidden surface removal problem has concentrated on the design of *output-sensitive* solutions, [Reif and Sen, 1988, Mulmuley, 1989, Bern *et al.*, 1990, Mulmuley, 1991a, Sharir and Overmars, 1992].

The problem of hidden surface removal can be viewed as finding a suitable partition of the view plane and labelling each region of the partition with the face that is visible there. When all scene faces are projected onto the view plane, several junctions will be formed between the xy-projections of face boundaries. Only a few of these junctions are visible. An efficient hidden surface algorithm should spend as little time as possible processing these invisible junctions. Such algorithms, with a complexity linked to the number of visible junctions, are termed output-sensitive. A brief outline of two output-sensitive algorithms, and coherence based solutions follow.

The algorithm proposed in [Sharir and Overmars, 1992] can be used to perform hidden surface removal for a set of n polygons for which a (partial) depth order is known. The main idea of the algorithm is quite simple, with the final view plane partition M constructed by incrementally adding the polygonal faces in a front-to-back order.

In more detail, the algorithm proceeds by choosing an initial constant positive integer parameter c and constructs the view plane partition M for the visible faces from the set of the first c faces. The algorithm then increments the parameter c, constructs the view plane partition for the next c visible faces and merges this partition with M. Once all the polygons have been processed, M will denote the required view plane partition for the whole scene. It should be noted that any each stage of the algorithm, all edges and vertices of M are part of the final partition. Details of the complexity and the role of the parameter c can be found in [Sharir and Overmars, 1992].

The image tree algorithm discussed previously, Section 3.1.7, is similar in many respects to this hidden surface algorithm, and extending the image tree algorithm using the above approach appears to be straightforward.

An alternative output-sensitive approach is given in [Mulmuley, 1989]. The algorithm proceeds by adding one face at a time in a random order. The crux of the algorithm lies in ensuring that if a junction q is computed at some instant during the algorithm, none of the faces added before that instant could obstruct q.

We first form an initial partition $\mathcal{H}_0$ of the view plane. $\mathcal{H}_0$ is simply the whole plane, and thus has just one region, labelled with the background face 0.

Starting with $\mathcal{H}_0$, we form a sequence of convex partitions $\mathcal{H}_0, \mathcal{H}_1, \ldots \mathcal{H}_n$, by "adding" the scene faces in a random order. This means that $\mathcal{H}_{k+1}$ will be obtained from $\mathcal{H}_k$ by adding to it a face f_{k+1}, randomly chosen from the set of remaining scene faces.

Each region R in $\mathcal{H}_k$ will be labelled with the face $f(R)$ which is "currently visible" in R among the $k+1$ faces, $f_0, f_1, \ldots, f_k$, that have been added so far. This means that the face $f(R)$ would be visible in R, if the scene were to consist only of faces $f_0, f_1, \ldots, f_k$. $\mathcal{H}_n$ will be the final visibility partition of the view plane.

A general approach to coherence based hidden surface removal for a moving view point and moving scene constructs a particular partition of space $\mathcal{H}$, induced by the initial set of polygonal faces A, and possibly a view point v, such that the set of visible faces, $\mathcal{F}$, is embedded within the partition. The initial set of visible faces are then determined using a conventional hidden surface removal algorithm, possibly output-sensitive. Then, by detecting when the *facial* structure of $\mathcal{H}$ alters, as a result of the view point moving or the scene changing, the possible accompanying changes in $\mathcal{F}$ can be detected. The details are fairly involved and so are omitted here. See [Mulmuley, 1991a] for a full description of two such algorithms.

The application of these output-sensitive approaches to hidden surface removal using BSP trees, and the use of coherence for arbitrary spatial partitions is described in Section 5.

4 Current Work

This Section surveys the work currently completed. It describes in detail an alternative view of the problems associated with constructing BSP trees, and discusses possible approaches to their solution.

The relationship between the partition of space, the search structure for the partition, and the data to be represented was first explored, Section 4.1.

These results and conclusions were applied to the representation of functions using arrangements, resulting in an alternative view of BSP trees, Section 4.2, and the construction of efficient trees, Sections 4.2.1 and 4.2.2.

Finally, the first steps on the way to a dynamic BSP data structure were made with operations for removing partitions and multidimensional tree rotation defined, Sections 4.3.3 and 4.3.2.

4.1 Data, Spatial Partitions, and Search Trees

When storing data in AVL, red-black or any other one dimensional binary search trees, a partitioning is induced on space. Balancing operations and restructuring of these trees does not affect the spatial partition but merely provides different searches of the space. The spatial partitioning induced by a set of data in one dimension is unique.

In higher dimensions, there may exist infinitely many different partitions of space which can be induced by a data set, with multidimensional search trees which can represent each spatial partitioning and data set. A useful view of these observations considers three entities:

- the Data to be represented.
- the Partition of space.
- the Search of space.

In order to gain a greater understanding of the underlying relationship between BSP trees and the data being modelled, the inter-dependencies of these three entities was considered.

A straightforward interpretation sees the spatial partition inducing a structure on space, with this structure represented – together with attributes defining the data – by a search structure. The role of the spatial partition can be seen as aiding the search structure by simplifying the attributes which must be assigned; partitioning space into regions where the data is value-continuous.

As we have seen, for a fixed set of data in one dimension there is a unique partition of space into regions where data is value-continuous, although there are a number of alternative search structures. The ability to transform between search structures representing the same data is thus captured by the rotation operations. In addition, the size of such a search structure is constant, being dependent only on the size of the spatial partition, allowing the effect of a rotation operation to be entirely predictable and thus enabling transformation to optimally balanced trees to be achieved efficiently.

In two dimensions the one-to-one relationship between the data and the partition of space is generally lost, e.g. consider k-d trees, described in Section 3.2.1. The loosening of this link can often be seen to result in a closer relationship between the spatial partition and the search structure. In dimensions higher than one, the common divide-and-conquer nature of the spatial partitions must be reflected by the search structure, in general removing any degree of freedom between the partition and the search structure. Again, considering the k-d tree in figure 7, it can be observed that the hierarchical partition of space dictates the search structure. This does not prohibit us from indirectly altering the partition, while preserving the data defined, by restructuring the search tree: clearly we are interested in the data defined by the search structure; the spatial partition is merely a tool with which to achieve this.

As with one dimensional trees, if k-d trees are used to store point data then the size of the search structure is constant, equal to the size of the data set, and the effect of rotation can be predicted, allowing optimally balanced trees to be generated efficiently.

If data of a higher dimension is to be stored, e.g. line segments, then the size of the search structure depends heavily on the partition, and is no longer constant for a fixed data set; as we saw in Section 3.1.5, trees of various sizes can be used to represent the same data set. The inability to predict the size of a tree for a specific data set, without resorting to construction of the tree, lies at the root of most problems associated with BSP trees. The problem of transforming between alternative BSP tree representations for fixed data sets was subsequently investigated, Section 4.3.3.

4.2 Partitioning Trees and Arrangements of Hyperplanes

As we have seen, an arrangement of hyperplanes partitions space into a number of cells and lower dimensional faces. Such a partition can be used to represent a function as in Section 3.1.4. If the arrangement is used to model a function f, then each cell has an associated value-continuous function f_i defining the value of f within the cell. Again, all points at which f is value-discontinuous are contained within the hyperplanes defining the arrangement.

Adopting this viewpoint, we can see that there is no need to distinguish between neighbouring cells for which f is value-continuous. By merging all such cells, the space will be partitioned into cells whose boundaries are the value-discontinuous points of f. Such a degenerate arrangement, where the faces form a subset of the faces of the full arrangement, is called a *skeleton arrangement*. This skeleton arrangement features the smallest number of cells necessary to represent f; it represents the minimum partition of space necessary. Conversely, the full arrangement represents the maximum partition of space required in the worst case.

While the skeleton arrangement is attractive, it offers little support for spatial queries. As an example consider modelling the characteristic function of a set of d dimensional polytopes. A boundary representation explicitly defines the $d-1$ dimensional faces of the skeleton arrangement from which the d dimensional cells can be inferred, but provides little structure for efficient spatial operations. This irregular partitioning of space makes if difficult for algorithms to take advantage of the possible geometric properties of more regular partitions. Clearly the efficiency of the representations must be matched by efficient algorithms.

For example, consider the problem of locating the cell featuring a given point in a full arrangement and a skeleton arrangement. In a full arrangement we can take advantage of the fact that each partition spans the complete space. By testing the position of the point with respect to a partition, we can immediately reject all cells lying on the

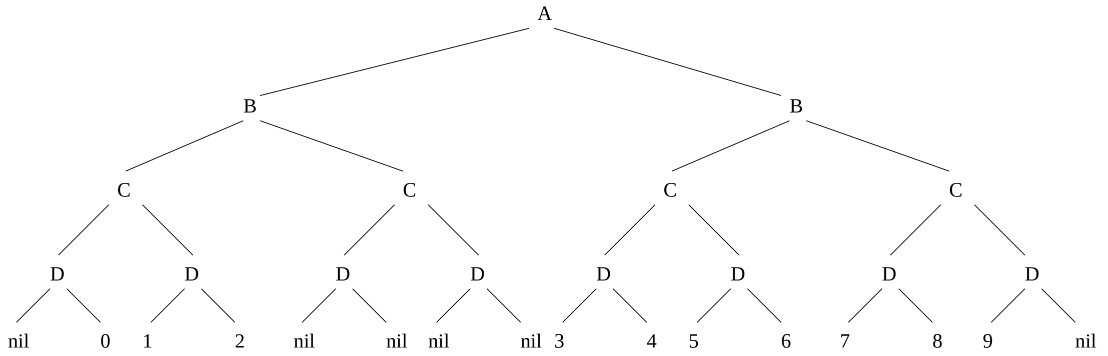


Figure 9: Computation tree for point location.

other side of the partition; if the point lies inside the partition then clearly the point cannot be located within any cells lying outside the partition. This can be repeated with each partition until a single cell is left. In contrast, the partitions within the skeleton arrangement provide little such help.

The operation of point location within a full arrangement, such as figure 8, can be naturally represented as a binary tree, figure 9, with each internal node representing a location test, with the left/right branch followed by points inside/outside the partition. Several paths of figure 9 lead to empty cells and thus feature unnecessary location tests, partitioning space into these empty regions. For example, the B node in the subtree to the left of root A is unnecessary since all points inside A are also inside B. Taking advantage of such redundant tests is difficult without building a data structure. Figure 10 shows the data structure for the computation in figure 9, with paths condensed as required.

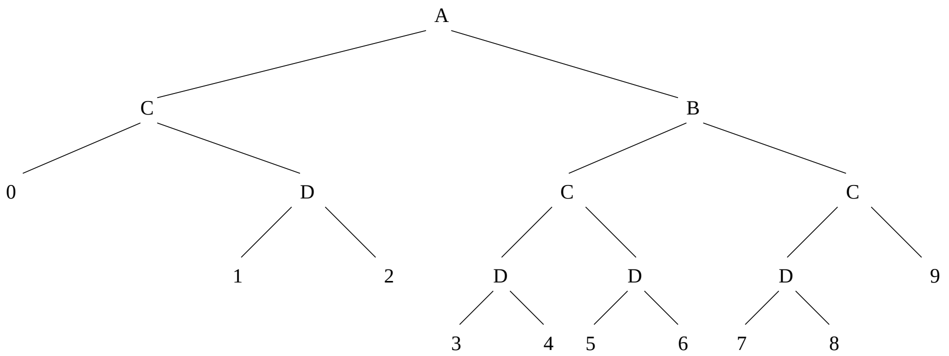


Figure 10: Data structure for point location.

It can be observed that the partitions of the arrangement not only separate regions where f is not value-continuous but also where f *is* value continuous. Consider figure 11, a representation of a function g, where similarly shaded regions denote subdomains where g is continuous. It can be clearly seen that each partition contains points at which g is value-discontinuous as well as points where g is value-continuous, e.g. hyperplane A separates regions 0 and 4, where g is value-discontinuous, as well as region 1 and 5, and, 2 and 6, where g is value-continuous. Figure 12 shows the data structure for g, with the definition of g within the cells at the leaves.

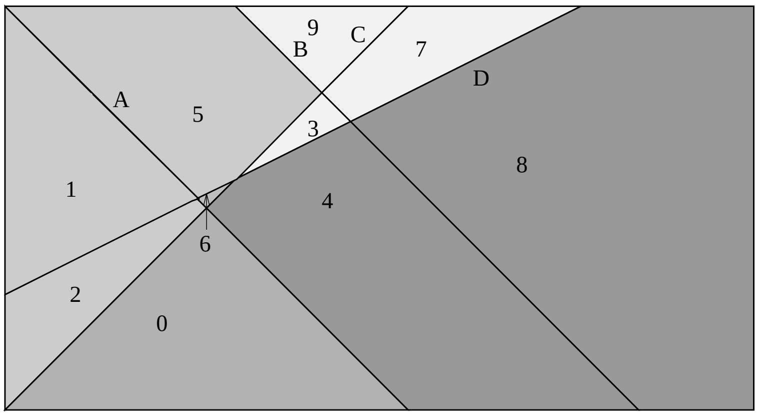


Figure 11: Representation of function g.

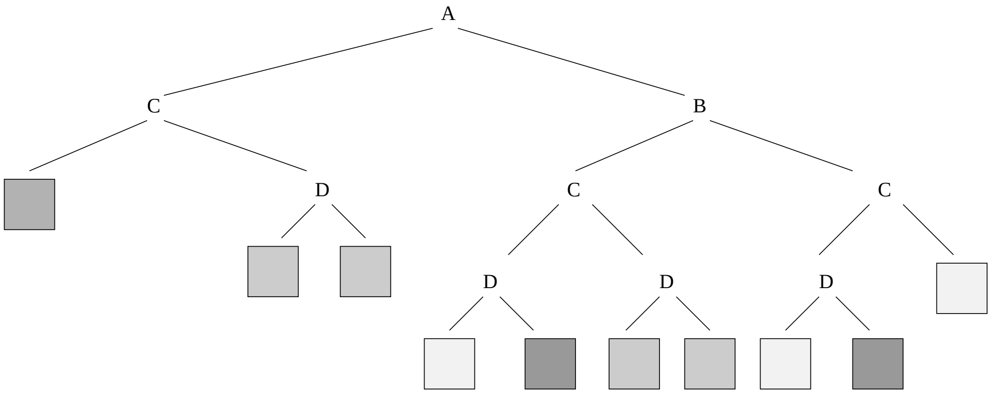


Figure 12: Data structure for function g.

Further condensation of the tree can be achieved by removing nodes partitioning regions where g is value-continuous, e.g. the D node in the left subtree of the root. Figure 13 shows the data structure after this condensation. This data structure is in fact a binary space partition tree for the function g. So a binary space partition can be viewed as an efficient data structure for operations on a function represented by a full arrangement.

Let us consider the two main data structures defined above: the point location structure, or arrangement tree, e.g. figure 10, and the condensed point location structure, or binary space partition, e.g. figure 13.

For a fixed function f, there are a number of distinct arrangement trees and BSP trees which can be used to represent f. All arrangement trees for f are of the same size

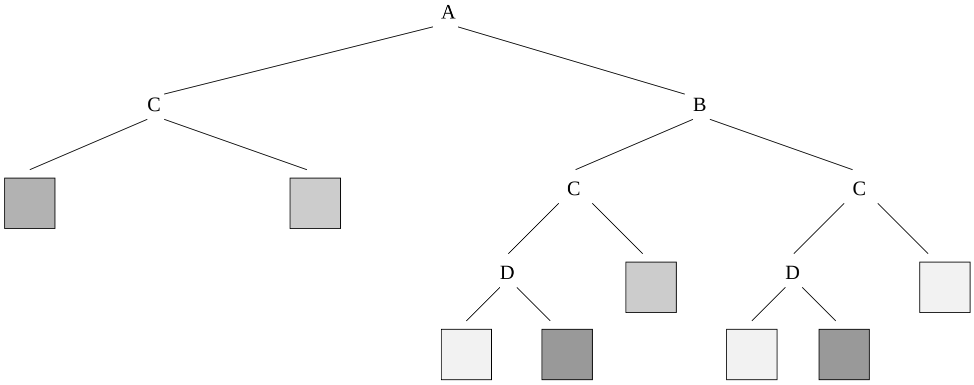


Figure 13: Condensed data structure for function g.

and describe the same spatial partition, the only area for variation being the order of the location tests, and the balance of the tree. In contrast, the BSP trees for f may be of different sizes and describe different spatial partitions. Each arrangement tree has a dual BSP tree, which can be generated by merging cells as described above. The construction of optimal or efficient BSP trees can be seen as the selection of the arrangement tree which, when condensed, results in an optimal or efficient BSP tree. Current worst-case BSP tree measurement results correspond to the size of arrangement trees which are themselves BSP trees, that is, representations of functions for which no merging of cells can be achieved. The ability to select an arrangement tree which is the dual of an optimal or at least efficient BSP tree is discussed below.

4.2.1 Optimal Trees

It is clear that there is no "optimal" BSP tree. The goodness of a tree is very much dependent on the tree's application. In this Section, and throughout this report, we will refer to an "optimal" BSP tree as a BSP tree which minimises a given *evaluator* function. This evaluator function may measure the balance or size of a BSP tree, for example.

Figure 14 shows an exhaustive approach to the construction of an optimal BSP tree. The complexity of this algorithm is extremely large, with an lower bound of $\Omega(2^n n!)$ for an arrangement of n hyperplanes, ignoring the possibly expensive evaluating and condensing functions. This reflects the large number of distinct arrangement trees, with the exhaustive evaluation of each possible arrangement tree ensuring the correctness of the algorithm for every possible evaluation function.

An alternative approach would be to interleave the construction of possible arrangements with the condensation. By doing this, we are avoiding constructing arrangements which condense to the same BSP trees. This could be achieved by constructing optimal left and right subtrees only for the subset of faces lying inside and outside the chosen root hyperplane. Clearly the number of distinct BSP trees is smaller than the number of distinct arrangement trees.

The complexity of the improved algorithm is difficult to predict since it depends not only on the number of hyperplanes, but also on the position of the faces on the hyperplanes.

```
Optimal_Build_Tree( set, evaluator ) -> B_Tree
{
  best = infinity;
  if set == NULL then
     optimal_tree = NULL;
  else
     for each hyperplane in set do
     {
        (inside, on, outside) = Partition_Set(hyperplane, set);
        tmp_tree = Make_Node(on);
        tmp_tree->left = Optimal_Build_Tree(inside, evaluator);
        tmp_tree->right = Optimal_Build_Tree(outside, evaluator);

        value = evaluator(tmp_tree);
        if ( value < best ) {
           best = value;
           copy(tmp_tree, optimal_tree);
        }
     }
  return optimal_tree;
}
```

Figure 14: Exhaustive algorithm for constructing optimal arrangement tree.

It is clear that these algorithms exhaustively construct all possible BSP trees in a bottom-up style: constructing an optimal tree by considering each possible root with optimal subtrees for the remaining faces. A top-down approach, such as the one illustrated in figure 15, is conventionally a natural approach to the efficient construction of tree-style data structures. Such an iterative approach is appealing, but requires the ability to efficiently determine the optimal root for the tree. Clearly, the choice of the root may depend on the optimal subtrees for the remaining faces. The details required to evaluate the suitability of a root may therefore require construction of the subtrees, resulting in an exhaustive algorithm.

For example, consider attempting to construct a balanced BSP tree. In selecting the optimal root, the size and balance of the optimal subtrees for the remaining polygons must be considered. As we have seen, predicting the size of optimal BSP trees, even asymptotically, does not appear to be possible. Thus, to construct a balanced BSP tree, the intractable exhaustive approach seems to be the only option.

In contrast, consider attempting to construct a BSP tree that at each node v partitions the space $R(v)$ into regions of approximately equal size. In these circumstances the optimal root can be chosen without considering the subtrees, and so the top-down approach can be used.

Since the majority of applications will be dependent on some measure of tree size, be it depth or number of internal nodes, efficient optimal tree construction using the top-down approach does not appear to be generally possible.

```
Optimal_Build_Tree( set, evaluator ) -> B_Tree
{
  hyperplane = Select_Optimal_Root(set, evaluator);
  (inside, on, outside) = Partition_Set(hyperplane, set);
  optimal_tree = Make_Node(on);
  optimal_tree->left = Optimal_Build_Tree(inside, evaluator);
  optimal_tree->right = Optimal_Build_Tree(outside, evaluator);

  return optimal_tree;
}
```

Figure 15: Top-down tree construction.

4.2.2 Efficient Trees

As a result of the seemingly intractable problem of constructing optimal trees, most current research has focused on building efficient BSP trees, [Naylor, 1993, Krishnaswamy *et al.*, 1990]. These algorithms are based, in the main, on a form of static sampling. Given the data to represent, a top-down approach is adopted with a subset of possible partitioning roots considered. An evaluation of each candidate then attempts to select an optimal or efficient partition for the root.

In addition to the problems associated with this top-down approach discussed above, i.e. the inability to accurately evaluate a candidate root, the selection of a reasonable candidate set is not trivial. The use of heuristics to generate the candidate set and to approximately evaluate each candidate is seen as the only option. A description of various heuristics together with empirical results is featured in [Krishnaswamy *et al.*, 1990].

It can be observed that a BSP tree generates a hierarchical decomposition of the space, with the regions along any path of the tree strictly decreasing in volume. This sequence of regions can thus be interpreted as a sequence of approximations that converges to the region corresponding to the leaf node. [Naylor, 1993] describes an alternative tree construction approach which aims to construct trees corresponding to a series of accurate approximations of the data. The motivation for this approach is the minimisation of the expected cost of particular operations on trees, e.g. point location, line segment classification, or a weighted set of such elementary operations. The tree construction is guided by an expected cost model for a particular operation. Point location for example, is modelled in the following way:

$$E_{\text{cost}}(T) = \begin{cases} 0 & \text{if } T \text{ is a cell} \\ 1 + p^- * E_{\text{cost}}(T^-) + p^+ * E_{\text{cost}}(T^+) & \text{otherwise} \end{cases}$$

where p^- and p^+ represent the probability of a point lying in the regions defined by T^- and T^+ respectively, given that it lies in the region defined by T.

While p^- and p^+ can be computed exactly, the values for $E_{\text{cost}}(T^-)$ and $E_{\text{cost}}(T^+)$ can only be estimated. The problem associated with all current construction methods is evident again, though the ability to compute the p^- and p^+ values makes these expected case models more accurate than traditional approaches.

"The simplest characterisation is that we want short paths to high probability cells, i.e. large cells, and symmetrically we will accept long paths to low probability cells, i.e. small cells corresponding to detail." This alternative notion of "good" trees is clearly not based on the balance or directly linked to tree size, and enables more accurate expected case models to guide tree construction. The construction of trees which minimise tree traversal operations, though, corresponds to the traditional notion of "good" tree construction.

The underlying static nature of these algorithms is significant. As a result it is not possible to construct "good" trees dynamically, that is extending existing "good" trees by adding objects will not generally result in a "good" tree for the extended scene. The static nature of these algorithms imposes a great deal of importance on tree root selection, which due to the nature of the problem may have been selected using little more than a guess. This situation is made worse by the "finality" of each choice. The prospects for a more flexible, dynamic construction process are discussed in Section 4.3.1.

4.3 Application of Computational Geometry

The section discusses the application of the techniques described in Section 3.2.2 to generate a dynamic BSP tree algorithm. The use of coherence is discussed in Section 5.

4.3.1 Dynamic Techniques

Section 3.2.2 outlined a number of techniques for transforming a static data structure into a dynamic data structure. Balanced binary search trees, e.g. red-black trees, provide an example of such a data structure, transformed using an efficient local rebuilding technique, the rotation operation. The binary search tree can be updated, by additions or deletions, and automatically rebalanced by a sequence of rotations. Figure 16 illustrates the rotation operation for binary search trees.

In order to apply such an approach to BSP trees, we need to design operations for removing data from a tree, to allow deletion, and transforming the tree, to allow restructuring or rebalancing.

4.3.2 Removing Spatial Partitions

It was observed previously that all the faces of the scene represented by a BSP tree are featured on the subhyperplanes of the tree nodes. In order to delete a face of the scene, let us consider the node holding the subhyperplane associated with the face.

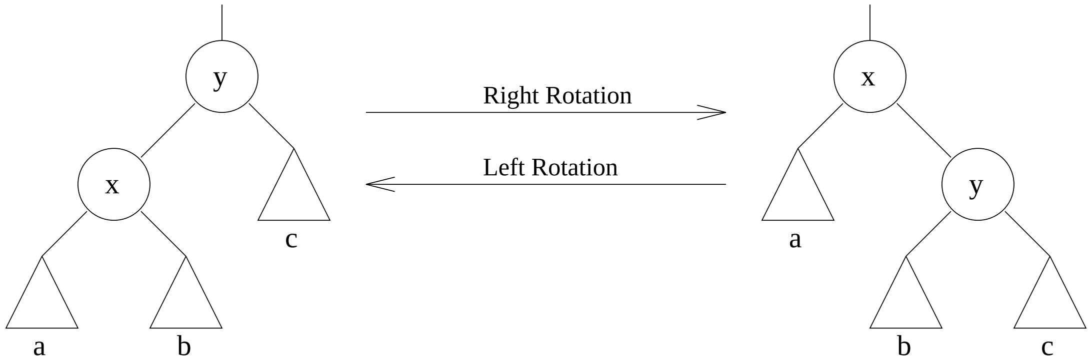


Figure 16: Rotation on a binary search tree

Several alternative methods can be used to produce a BSP tree for the scene with the face deleted. The most obvious is a *lazy* form of global rebuilding: the face can simply be marked as deleted, with the complete tree rebuilt when the number of deleted faces becomes, say, greater than the number of actual faces. This method provides deletion in constant time, once the node holding the face has been found. Of course the associated increase in tree size, due to the unnecessary partition, and subsequent deterioration in query time may offset this gain in situations where deletions occur frequently. It should be noted, though, that if other faces share the subhyperplane then the partition at this node is still necessary. In this case the trivial deletion method is optimal, preserving the properties of the tree without causing an unnecessary increase in size.

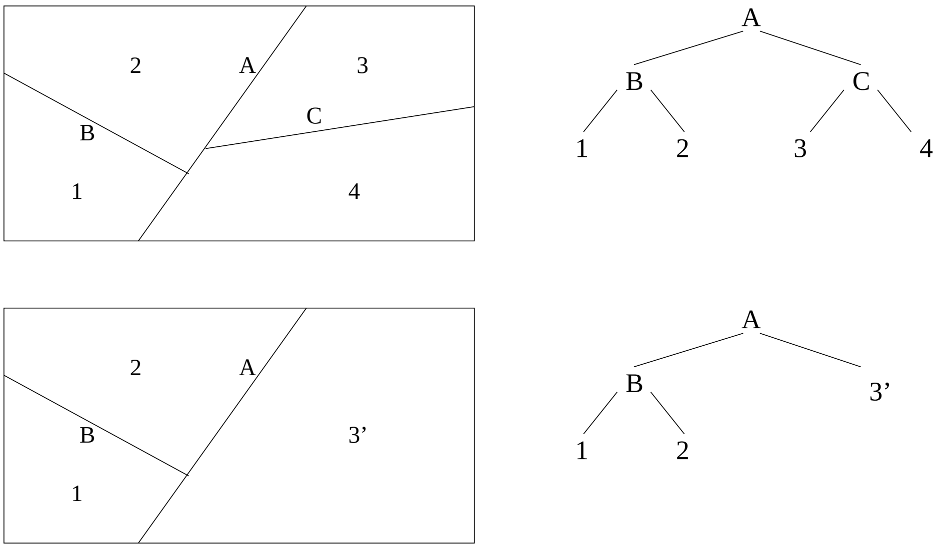


Figure 17: Deletion of a partition.

In addition, the method can take advantage of several special cases. If the node has no children, and features no other faces, then the node can be seen to partition the subspace into regions featuring no objects, and can be therefore simply deleted. Another trivial case occurs when the node has exactly one child. In this case, the node was used to partition the subspace into an empty region and a non-empty region. Therefore the tree can be updated correctly by replacing the node with the child subtree.

The lazy form of deletion is thus only required when the face to be deleted is the only face on a subhyperplane whose node has two non-empty children.

Since the only portion of the tree requiring alteration is the subtree rooted at the node holding the face, the technique of partial rebuilding, outlined in Section 3.2.2, seems applicable.

The removal of the root of the subtree, the node featuring the face, leaves two subtrees, each defined over neighbouring but independent subspaces. These trees can be merged together to produce a subtree for the subregion of the deleted node, which can replace the deleted node. Figure 17 illustrates the process.

This deletion operation produces a valid BSP tree, with the deleted face removed, but the tree may no longer be an efficient representation of the scene, properties such balance may now be violated.

4.3.3 Multidimensional Rotation

An ideal BSP construction approach would allow trees to be constructed dynamically, with the trees minimising the chosen evaluation function. Such a dynamic construction approach would clearly require the ability to restructure an existing tree when extending it with new objects in order to preserve the optimality of the tree.

It was pointed out in Section 4.1 that the ability to transform between search structures representing the same data is captured by rotation operations. Such an operation for BSP trees can be defined. Figure 18 shows a typical BSP tree. A rotation operation, or perhaps more accurately a swap operation, can be achieved by swapping the order in which the partitioning planes A and B are applied.

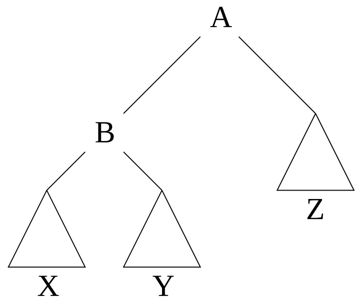


Figure 18: BSP tree to be restructured.

Unlike one dimensional search trees, the relationship between the hyperplane A and the data on B tells us nothing about the relationship between the hyperplane B and the data on A. A case analysis of the relationship is therefore necessary to describe the possible trees resulting from the rotation. Figure 19 illustrates the possible resulting trees. The following definitions define each of these trees:

- Consider partitioning Z with hyperplane B. Let Z_{-b}, Z_b, and Z_{+b} denote the resulting BSP trees lying inside, on, and outside B respectively.
- Let node B' represent the data held at nodes B and Z_b.

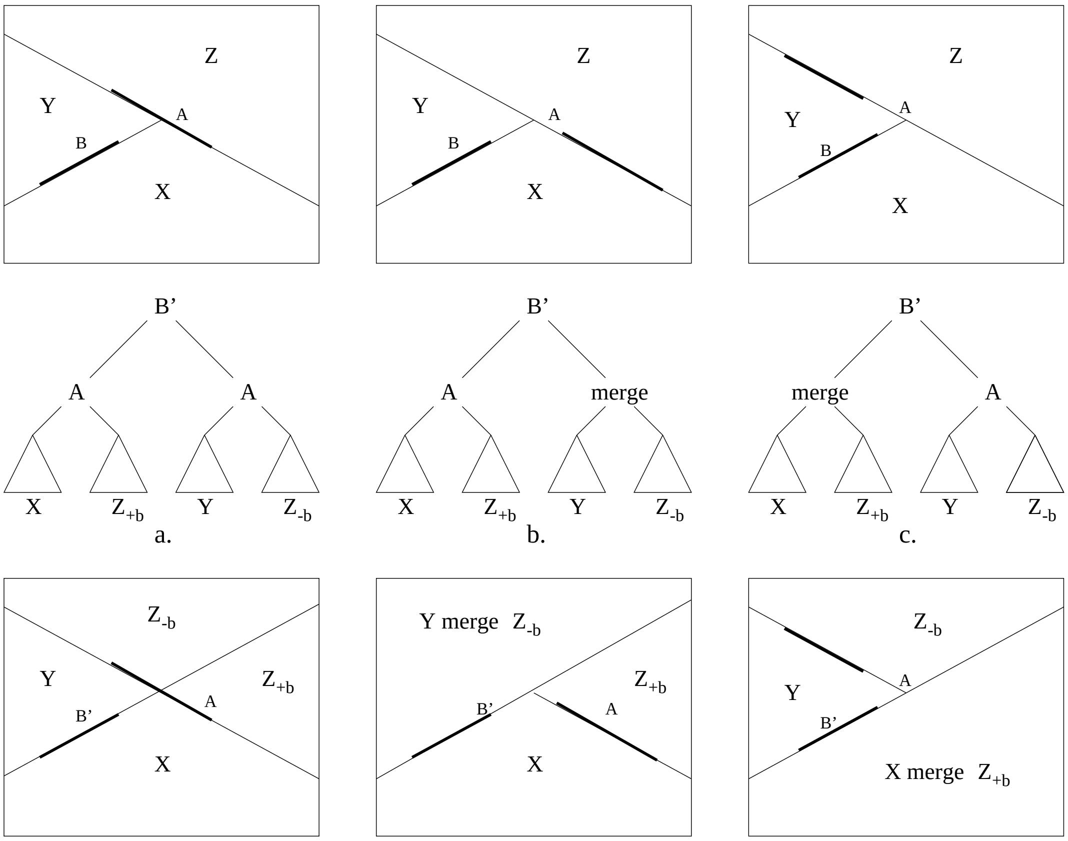


Figure 19: Possible resulting trees

As pointed out, the transformation ability provided by this operation is of greatest benefit when an accurate prediction of its effect can be made. Some upper bounds on the complexity of resulting trees can be made, with a maximum increase in size of $|Z|$. More accurate estimations of the tree size can be made by considering each of the resulting trees in figure 19. Some additional definitions and observations can be made.

Let the number of faces lying on the plane B be $m+1$. Note that this is the number of times that the polygon with plane B was split up by the partitions within this tree. By promoting B to the root of the tree we are rejoining all these fragments. If the number of fragments lying in tree Z is m_z then in the worst case we have $|Z_{+b}| = |Z_{-b}| = |Z| - m_z$.

The worst case cost of merging trees was discussed in Section 3.1.6. In b. and c. in figure 19 the merging is of a special nature since we have a partition, A, which separates the trees to be merged. Thus if we consider situation b., the trivial merging using partition A gives us a subtree of size $|X| + |Z_{-b}| + 1$. The worst-case complexity for merging gives us a possible subtree of size $|X||Z_{-b}|$. The problem is that we would *expect* the merging to result in a tree of size $< |X| + |Z_{-b}| + 1$, but cannot guarantee this by using the general merging operation, and are restricted to a tree of size $|X| + |Z_{-b}| + 1$ by using the trivial merging. A more detailed examination of the merging operation and its associated complexity is clearly required.

Using the trivial merging of trees, an upper limit on the resulting tree size of $|X| + |Y| + 2|Z| - 2m_z + 3$, an increase in size of $|Z| - 2m_z + 1$. A significant amount of work clearly still needs to be done in order to achieve an accurate measure of the resulting trees in particular circumstances, with a greater understanding of the merging operation holding the key.

Ideally we would like to know whether the tree resulting from a rotation is an improvement, with regard to some tree evaluation function, before carrying out the restructuring. The definition of the rotation operation is the first step in the definition of dynamic BSP tree data structure, providing the ability to transform between alternative representations for the same data. The next step involves determining which transformations improve the representation, and designing tree construction algorithms which can take advantage of such transformations. The current application of such rotation operations in balancing multidimensional search structures and generating dynamic data structures is currently being looked at, [Overmars, 1983, Mulmuley, 1991b, Mulmuley, 1991c], with this goal in mind.

5 Proposal

The static nature of the BSP tree data structure has been highlighted within this report. The efficient hierarchical partition of space used to represent data enforces an order on the partitioning hyperplanes, in contrast to the less efficient partition induced by an arrangement of hyperplanes, resulting in multiple representations for fixed data, and the seemingly intractable problem of selecting the optimal representation for a particular application.

The major goal of this research is the design of a more flexible data structure for efficiently representing data using binary partitions. While BSP trees have been seen to provide a useful and efficient representation of fixed geometric sets, I would like to turn my attention to the efficient representation of dynamic geometric sets. The introduction of dynamic data structure capabilities such as tree restructuring and data deletion, forms the first stage of this goal, and while basic operations providing these capabilities have been designed, a significant amount of work is required in order to fully appreciate the limits, efficiency and application of these operations. Further stages will involve the design of efficient dynamic tree construction algorithms.

An area which has received little attention is the use of BSP trees for the representation of non-linear geometric objects. The current use of hyperplanes allows space to be partitioned into two distinct regions, but clearly alternative non-linear partitions can be used in a similar way, allowing a wider set of geometric objects to be represented. The possible dependence on linear partitions inherent in current BSP tree based applications and properties of tree traversals, together with a fuller exploration of non-linear BSP trees, is another area I propose to look at in detail.

An additional goal is the application of computational geometry techniques to provide more efficient BSP tree applications. Current applications use BSP trees to provide an ordering of the scene faces, and while this is a powerful and useful feature of the data structure, few alternative uses of the spatial partition have been proposed. Spatial partitions have been demonstrated to be a powerful tool in solving many geometric problems and the adaption of such techniques to general binary partitions shall be explored. The use of multiple BSP and image trees for scene representation and parallel BSP implementations are additional areas of interest which I intend investigating.

A outline of some of the short term goals follows.

Rotation operations have been used in many one dimensional search structures, providing the ability to transform between alternative representations of the same data. The use of rotation to generate optimal trees relies not only on the transformation ability but on the ability to predict the outcome of a transformation, and to predict the sequence of rotations which will result in an optimal representation.

At this stage, the transformation between alternative BSP representations has been achieved, Section 4.3.3. Predicting the outcome of such a transformation is the next stage. Predicting details of BSP tree size and balance is, in general, a recurring problem, affecting tree construction and merging. The gap between the worst-case bound on tree size of $\Theta(n^2)$ and the best-case bound on tree size of $\Theta(n)$ is at the root of this problem. It is observed that empirical results indicate that the expected-case bound lies around $O(n \log n)$, and that "there is no example yet of a single manifold comprised of n faces generating worst-case behaviour", [Naylor, 1993]. As a result efficiently predicting the size of a BSP tree, without construction, appears to be a difficult task in general.

The use of transformations to produce more efficient BSP trees appears to be a more achievable goal. The tree construction approach described in [Naylor, 1993], based on creating a series of approximations of the data, uses local optimisations to construct BSP trees. While this will not necessarily lead to a optimal tree, it ensures a degree of efficiency for the representation. Obtaining the local information to guide these local decisions is straightforward and can be achieved efficiently. Transformations where we can predict the local affect thus seem a possibility.

The use of transformation for more general optimisation is an area requiring significantly more work. The reliance on predicting the size of merged trees is the main problem. This problem can be reduced by requiring the ability to predict whether a transformation will improve the tree or not. In certain circumstances this can be achieved, and a fuller investigation into these possibilities shall be carried out.

The exploitation of coherence is crucial in most efficient graphic applications and algorithms, yet has received little attention with respect to BSP trees. The growth in research into algorithms exploiting coherence in dimensions higher than one has introduced several techniques and approaches which may be suitable for BSP trees. For example, the coherence based hidden surface removal algorithm described in Section 3.2.5, [Mulmuley, 1991a], constructs a spatial partition representing the complete scene, embedding a partition representing the visible scene. The spatial partition represented by a BSP tree, together with the image tree representing the visible scene, correspond very closely to these structures, and adapting the algorithm for use with BSP and image trees is a distinct possibility.

A Recent Work

A.1 Partially Ordered Sets and Binary Space Partitions

A.1.1 Partially Ordered Sets

"A *partially ordered set* or *poset* $\mathcal{P}$ is an algebraic system in which a binary partition $x \geq y$ (read: x includes y) is defined, which satisfies the following postulates.

- For all x, $x \geq x$ (reflexive property)
- $x \geq y \wedge y \geq x \Rightarrow x = y$ (antisymmetric property)
- $x \geq y \wedge y \geq z \Rightarrow x \geq z$ (transitive property)

A binary relation which satisfies these postulates is called an *inclusion relation* or *order relation.*

It should be observed that the inclusion relation need not be defined for each pair of elements of $\mathcal{P}$. It is sufficient that it be defined for some pairs of elements.

Any poset $\mathcal{P}$ can be represented by a *Hasse diagram* in which distinct elements of $\mathcal{P}$ are represented by distinct points and in which each relation $x \geq y$ is represented by a line or lines which descend steadily from x to y. Nevertheless the fact that a point x is above the point y in a Hasse diagram is not sufficient to guarantee that $x \geq y$. In fact, x and y are not related unless the diagram has at least one line which steadily descends or steadily ascends, from one point to the other." [Rutherford, 1965].

A.1.2 Binary Space Partition

"The binary space partitioning (BSP) tree data structure, was developed to provide an efficient method for calculating the visibility relationship among a static group of 3D polygons," [Foley *et al.*, 1990]. A first impression would appear to link the BSP tree with a Hasse diagram – both are representatives of an ordering, with the nature of the represenations similar.

Let us consider the back-to-front order of polygons, with respect to an arbitrary view point, as given by a BSP tree. With a little thought, it is clear that the BSP tree implys a relationship between each and every polygon in the tree, generalising the relationships represented by a Hasse diagram – x and y are related if there is a path from x to y. It can be shown that the order represented by a BSP tree is a partial order, satisfying the postualtes given in Section A.1.1.

Definition 1 *A relation $x \geq_1 y$, defined to be true when there exists a node a_x such that y lies in a_x's left subtree or equals y, and x lies in a_y's right subtree or equals x, captures this order. Note that the node a_x is unique for any pair x, y*[4]*.*

[4]This operator trivially satisfies the first and second postulates. Let us consider the relations $x \geq_1 y$ and $y \geq_1 z$. Assuming these relations hold, let a_x and a_y denote the nodes in the above definition of the operator. Note that a_x and a_y are both ancestors of y. If a_y is a descendant of a_x then a_y, and therefore z, must lie in a_x's left subtree. Alternatively, if a_y is an ancestor of a_x then y, a_x and therefore x lie in a_y's right subtree, with z in a_y's left subtree, or equal to a_y. Finally if $a_y = a_x$, then z lies in a_y's left subtree, or equals a_y, while x lies in a_y's right subtree, or equals a_y. So by case analysis, the third postulate, $x \geq_1 y \wedge y \geq_1 z \Rightarrow x \geq_1 z$, is also satisfied. The order defined by the BSP tree is therefore a partial order.

The order represented by a BSP tree is actually a *total* order on the set, with the operator defined above describing the relations between elements in positions of the order, but offers no support in the construction of the order. This total order is not unique, as we have seen repeatedly, and evolves from the general relationship between elements of the set, and the associated partial order. Consider an alternative definition of the ordering operator.

Definition 2 $x \geq_2 y$ = *(y lies inside or on the plane of x) or (x lies outside or on the plane of an element z and $z \geq_2 y$).*

This definition captures in the most general way the relationships between elements of the set – x includes y if either y lies inside x's plane or there exists an element z such that z doesn't include x but does include y. The ordering generated may be circular with $x \geq_2 y \wedge y \geq_2 x \wedge x \neq y$. It appears that the BSP tree represents this ordering by *selecting* which pairs to define operator $\geq_2$ over in order to generate a non-circular total ordering for the elements. Defining an operator which *selects* a total order from a number offered by the partial order is a non-trivial task. It seems likely that this task corresponds to the general construction of BSP trees. While $\geq_1$ describes the relationships between elements of a constructed tree, $\geq_2$ describes the complete range of relationships which may be used in construction. An investigation into the differences between the two operators, particularly the cases where they disagree, may shed more light on the problem of general tree construction.

Axiom 3 $x \geq_1 y \Rightarrow x \geq_2 y$

Proof
Case analysis on $x \geq_2 y$.

- $x = a_x$;
  - $y = a_x$;
    The reflexive property ensures that the definitions agree in this case.
  - y lies in a_x's left subtree;
    Since y lies in the left subtree rooted at x, y must lie outside the plane of x, so $x \geq_2 y$.
- x lies in a_x's right subtree;
  - $y = a_x$;
    x lies in the right subtree rooted at a_x, so x must lie inside the plane of a_x, and since $y \geq_2 a_x$, we have $x \geq_2 y$.
  - y lies in a_x's left subtree;
    As before, x lies in the right subtree rooted at a_x, so x must lie inside the plane of a_x. Since y lies in the left subtree rooted at a_x, y must lie outside the plane of a_x, so $a_x \geq_2 y$ and therefore we have $x \geq_2 y$.

□

Axiom 4 $x \geq_2 y \not\Rightarrow x \geq_1 y$

Proof

Trivial examples can be found such that $x \geq_2 y \wedge y \geq_1 x \wedge x \neq y$. Since $\geq_1$ is antisymmetric, $\geq_1$ must differ from $\geq_2$ on these occasions.

□

These results reinforce the notion that $\geq_1$ is obtained from $\geq_2$ by restricting the domain of the operator and breaking circular relationships between elements. Let us now consider the Hasse-style diagram[5] for operator $\geq_2$. $\geq_2$ is defined for each pair of elements in the set and so we have a fully connected graph. The extraction of a BSP tree from this graph, together with the complications resulting from portions of objects being related, is an area requiring further investigation.

[5]The diagram will not be a true Hasse diagram since $\geq_2$ is not an inclusive operator

References

[Bern *et al.*, 1990] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[Bresenham, 1988] J.E. Bresenham. Anomalies in incremental line rastering. In Rae A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 329–349. Springer-Verlag, 1988.

[Campbell and Fussell, 1990] A.T. Campbell and D.S. Fussell. Adaptive mesh generation for global diffuse illumination. In *Proceedings of the ACM SIGGRAPH, Computer Graphics, 24(4)*, pages 155–164, August 1990.

[Campbell, 1991] A.T. Campbell. *Modeling Global Diffuse Illumination for Image Synthesis.* PhD thesis, Department of Computer Sciences University of Texas at Austin, December 1991.

[Carson, 1984] G.S. Carson. An approach to the formal specification of graphics systems. *Computers and Graphics*, 8(1):51–58, 1984.

[Chin and Feiner, 1989] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. In *Proceedings of the ACM SIGGRAPH, Computer Graphics, 23(3)*, pages 99–106, July 1989.

[Chin and Feiner, 1992] N. Chin and S. Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In David Zeltzer, editor, *Symposium on Interactive 3D Graphics, Computer Graphics, 25(2)*, pages 21–30, March 1992.

[d'Amore and Franciosa, 1992] F. d'Amore and P.G. Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. *Information Processing Letters*, 44(5):255–259, December 1992.

[Duce, 1988] D.A. Duce. Formal specification of graphics software. In Rae A. Earnshaw, editor, *Theorectical Foundations of Computer Graphics and CAD*, pages 543–574. Springer-Verlag, 1988.

[Edelsbrunner *et al.*, 1986] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15:341–363, May 1986.

[Edelsbrunner, 1987] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, 1987.

[Eker, 1990] S. Eker. *Formal Foundations for the Design of Rasterisation Algorithms and Architectures.* PhD thesis, School of Computer Studies, University of Leeds, 1990.

[Fiume, 1987] E. Fiume. Bit-mapped graphics: A semantic and theory. *Computers and Graphics*, 11(2):121–140, April 1987.

[Fiume, 1989] E. Fiume. *The Mathematical Structure of Raster Graphics.* Academic Press, 1989.

[Foley *et al.*, 1990] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, second edition, 1990.

[Fuch *et al.*, 1980] H. Fuch, A. Kedem, and B. Naylor. On visible surface generation by a priori tree structure. In *Proceedings of the ACM SIGGRAPH, Computer Graphics, 14(3)*, pages 124–133, July 1980.

[Gnatz, 1984] R. Gnatz. Approaching a formal framework for graphics software standards. *Computers and Graphics*, 8(1):39–50, 1984.

[Goodrich and Tamassia, 1991] M.T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proceedings of the Symposium on Theory of Computing*, pages 523–533, 1991.

[Gordon and Chen, 1991] D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, pages 79–85, September 1991.

[Krishnaswamy *et al.*, 1990] R. Krishnaswamy, G.S. Alijani, and Su Shyh-Chang. On constructing binary space partition trees. *Cooperation, ACM 18th Annual Computer Science Conference Proceedings*, pages 230–235, February 1990.

[Lin, 1987] J. Lin. Fast ray tracing on BSP-trees. Master's thesis, Department of Computer Science, University of Central Florida, Orlando, 1987.

[Luby, 1988] M.G. Luby. Grid geometries which preserve properties of Euclidean geometry: A study of line drawing algorithms. In Rae A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 397–432, 1988.

[Mallgren, 1983] W.R. Mallgren. *Formal Specification of interactive graphics programming languages.* ACM distinguished dissertations. MIT Press, 1983.

[Mehlhorn, 1984] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, 1984.

[Mulmuley, 1989] K. Mulmuley. An efficient algorithm for hidden surface removal. In *Proceedings of the ACM SIGGRAPH, Computer Graphics, 23(3)*, volume 23, 1989.

[Mulmuley, 1991a] K. Mulmuley. Hidden surface removal with respect to a moving view point. In *Proceedings of the Symposium on Theory of Computing*, pages 512–522, 1991.

[Mulmuley, 1991b] K. Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling. In *Proceedings 32nd Annual Symposium on Foundations of Computer Science*, pages 216–227, 1991.

[Mulmuley, 1991c] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *Proceedings 32nd Annual Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[Naylor and Thibault, 1986] B. Naylor and W. Thibault. Application of BSP trees to ray-tracing and CSG evaluation. Technical Report GIT-ICS 86/03, Georgia Institute of Technology, 1986.

[Naylor *et al.*, 1990] B. Naylor, J. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. In *Proceedings of the ACM SIGGRAPH, Computer Graphics, 24(4)*, pages 115–124, August 1990.

[Naylor, 1990a] B. Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer-Aided Design*, 22(4), May 1990.

[Naylor, 1990b] B. Naylor. SCULPT: An interactive solid modeling tool. In *Proceedings of Graphics Interface*, pages 138–148, May 1990.

[Naylor, 1992a] B. Naylor. Interactive solid geometry via partitioning trees. In *Proceedings of the Graphics Interface*, pages 11–18, May 1992.

[Naylor, 1992b] B. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of the Graphics Interface*, pages 201–212, May 1992.

[Naylor, 1993] B. Naylor. Constructing good partitioning trees. In *Proceedings of the Graphics Interface*, 1993.

[Ooterom, 1990] P. Van Ooterom. A modified binary space partitioning tree for geographical information systems. *International Journal of Geographical Information Systems*, 4(2):133–146, April 1990.

[Overmars, 1983] M.H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, 1983.

[Paterson and Yao, 1989] M.S. Paterson and F.F. Yao. Binary partitions with applications to hidden surface removal and solid modeling. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry*, pages 23–32, 1989.

[Paterson and Yao, 1990a] M.S. Paterson and F.F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Computational Geometry*, 5:485–503, 1990.

[Paterson and Yao, 1990b] M.S. Paterson and F.F. Yao. Optimal binary space partitions for orthogonal objects. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 100–106, 1990.

[Paterson and Yao, 1992] M.S. Paterson and F.F. Yao. Optimal binary space partitions for orthogonal objects. *Journal of Algorithms*, 13:99–113, 1992.

[Preparata and Shamos, 1988] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 2nd edition, 1988.

[Radha *et al.*, 1990] H. Radha, R. Leonardi, B. Naylor, and M. Vetterli. Image representation using binary space partitioning trees. In *Proceedings of the SPIE - The International Society for Optical Engineering, Visual Communications and Image Processing '90*, volume 1360 pt1, pages 639–650, October 1990.

[Radha *et al.*, 1991a] H. Radha, R. Leonardi, and M. Vetterli. A multiresolution approach to binary tree representations of images. In *International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2653–2656, April 1991.

[Radha *et al.*, 1991b] H. Radha, R. Leonardi, M. Vetterli, and B. Naylor. Binary space partitioning tree representation of images. *Journal of Visual Communication and Image Representation*, 2(3):201–221, September 1991.

[Reade, 1992] C. Reade. Process algebra specification of graphics standards. Technical Report 92/062, Rutherford Appleton Laboratory, 1992.

[Reif and Sen, 1988] J. Reif and S. Sen. An efficient output sensitive hidden surface removal algorithm and its parallelization. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 193–200, 1988.

[Rutherford, 1965] D.E. Rutherford. *Introduction to Lattice Theory*. Oliver & Boyd, 1965.

[Sharir and Overmars, 1992] M. Sharir and M.H. Overmars. A simple output sensitive algorithm for hidden-surface removal. *ACM Transactions on Graphics*, 11(1):1–12, January 1992.

[Teillaud, 1992] M. Teillaud. *Towards Dynamic Randomized Algorithms in Computational Geometry*. PhD thesis, INRIA, July 1992.

[Thibault, 1987] W. Thibault. *Application of Binary Space Partitioning Trees to Geometric Modeling and Ray-Tracing*. PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, 1987.

[Tokuta, 1991] A.O. Tokuta. Motion planning using binary space partitioning. In *Proceedings of IROS, IEEE/RSJ International Workshop on Intelligent Robots and Systems*, volume 1, pages 86–90, November 1991.