INT. J. GEOGRAPHICAL INFORMATION SYSTEMS, 1990, VOL. 4, NO. 2, 133–146

A modified binary space partitioning tree for geographic information systems

PETER VAN OOSTEROM

Department of Computer Science, University of Leiden,
P.O. Box 9512, 2500 RA Leiden, The Netherlands

Abstract. We present a reactive data structure, that is, a spatial data structure with detail levels. The two properties, spatial organization and detail levels, are the basis for a geographic information system (GIS) with a multi-scale database. A reactive data structure is a novel type of data structure catering to multiple detail levels with rapid responses to spatial queries. It is presented here as a modification of the binary space partitioning tree that includes the levels of detail. This tree is one of the few spatial data structures that does not organize space in a rectangular manner. A prototype system has been implemented. An important result of this implementation is that it shows that binary space partitioning trees of real maps have $O(n)$ storage space complexity in contrast to the theoretical worst case $O(n^2)$, with n the number of line segments in the map.

1. Introduction

In the past few years there has been a growing interest in geographic information systems (GIS). There are many applications that use GIS technology including automated mapping/facility management (AM/FM); command, control and communication systems (C^3S); war gaming; and car or ship navigation systems. A major advantage of a GIS over the paper map is that the operator (end-user) can interact with the system. To make this interaction both possible and efficient, the GIS must be based on an appropriate data structure. A reactive data structure is a data structure with the following two properties:

(1) *Spatial organization.* This is necessary for efficient implementation of operations such as selection of all objects within a rectangle, picking an object from the display, map overlay computations, and so on (van Oosterom 1988, Chrisman 1989). Several spatial data structures are described in the literature and are implemented in existing GIS (Bentley 1975, Guttman 1984, Günther and Bilmes 1988, Samet 1989).

(2) *Detail levels.* Too much detail in the display will hamper the operator's perception of the important information. Also, unnecessary detail will slow down the drawing process. When the operator wants to take a closer look at a part of the map, the system enlarges objects and shows more detail (new objects). Conversely, when zooming out, it removes fine detail from the display. We call this operation logical zoom as opposed to the ordinary zoom which changes only the size of objects. There is some literature on data structures with detail levels, for instance strip trees (Ballard 1981) and multi-scale line-trees (Jones and Abraham 1987).

Reactive data structures are the basic building blocks for seamless, scaleless geographic databases (Guptill 1989). The detail levels must be integrated in the spatial

organization, not by simply storing maps at several different scales (e.g., Waugh and Dowers 1988), each with its own separate spatial organization.

The data structure presented in this paper is a modification of the binary space partitioning (BSP) tree. A short description of the original BSP-tree is given in § 2, together with some minor modifications for the GIS environment. The next section shows how the basic spatial operations can be implemented efficiently by using a BSP-tree. Section 4 describes the most important difference with the original BSP-tree, the incorporation of detail levels. The balancing of the BSP-tree is discussed in § 5 for both the static and the dynamic case. Section 6 contains the first practical results from our implementation. Finally, the pros and cons are discussed in § 7.

2. The BSP-tree and some variations on it

2.1. *The original BSP-tree*

The original use of the BSP-tree was in 3D computer graphics (Fuchs *et al.* 1980, Fuchs *et al.* 1983). The BSP-tree reflects a recursive division of space. Each time a (sub-) space is divided into two sub-spaces by a splitting primitive, a corresponding node is added to the tree. The BSP-tree itself represents an organization of space by a set of convex sub-spaces in a binary tree. This tree is useful during spatial search and other spatial operations. Figure 1 (*a*) shows a 2D scene with some directed line segments. A 2D scene is used here, because it is easier to draw than a 3D scene. However, the principle remains the same. The 'left' side of the line segment is marked with an arrow. From this scene, line segment A is selected and the 2D space is split into two parts by the supporting line of A, indicated by a dashed line in figure 1 (*b*). This process is repeated for each of the two sub-spaces with the other line segments. The splitting of space continues until there are no line segments left. Note that sometimes the splitting of a space implies that a line segment (that itself has not yet been used for splitting) is split into two parts. Line D, for example, is split into d1 and d2. Figure 1 (*b*) shows the resulting organization of the space, as a set of (possibly open) convex sub-spaces. The corresponding BSP-tree is drawn in figure 1 (*c*). In the 3D case supporting planes of flat polygons are used to split the space instead of lines.

The choice of which line segment to use for dividing the space very much influences the building of the tree. It is preferable to have a balanced BSP-tree with as few nodes as possible. This is a very difficult requirement because balancing the tree requires that line segments from the middle of the data set are used to split the space. These line segments will probably split other line segments. Each split of a line segment introduces an extra node in the BSP-tree. It is not clear how we can optimize the BSP-tree, so further research is needed here.

Figure 2 contains a Pascal-like code of a program that builds a BSP-tree. The program BuildTree is a variation of the traditional method (non-incremental) for

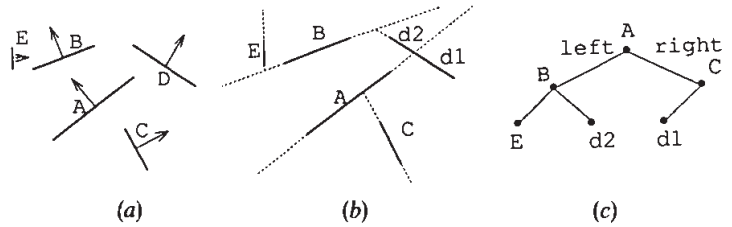

Figure 1. The building of a BSP-tree: (*a*) 2D scene, (*b*) convex sub-spaces, (*c*) BSP-tree.

```
Program BuildTree;
type BSP=^node;
     node=record
          segm: Line;
          l,r:  BSP
     end;
var  root: BSP;
     newsegm: Line;
root:= nil;
while GetLine(newsegm) do
     root:=AddLine(root,newsegm);
```

```
function AddLine(tree:BSP;segm:Line):BSP;
var Lsegm, Rsegm:Line;
begin
   if tree=nil then
      tree:=CreateNode(tree, segm)
   else
      case LinePosition(tree,segm) of
      LEFT: tree^.l:=AddLine(tree^.l,segm);
      RIGHT:tree^.r:=AddLine(tree^.r,segm);
      SPLIT:
         SplitLine(tree,segm,Lsegm,Rsegm);
         tree^.l:=AddLine(tree^.l,Lsegm);
         tree^.r:=AddLine(tree^.r,Rsegm);
   AddLine:=tree;
end;
```

Figure 2. Incremental BSP-tree building algorithm.

building a BSP-tree (Fuchs *et al.* 1980). The procedure SplitLine and the functions LinePosition, CreateNode and GetLine are not included, because their meaning will be clear. A node in the BSP-tree is represented by the record type *node*, which contains a line segment and pointers to the left and right child. Initially, the tree is empty. As long as GetLine can fetch a new line segment, it is added to the BSP-tree with a call to the function AddLine. AddLine checks whether the correct position in the BSP-tree is found. This is true if the pointer *tree* in the BSP-tree is *nil*. In that case a new node is created and added to the tree. Otherwise, LinePosition determines in which sub-tree the line segment has to be stored. The storage of the line segment is implemented by a recursive call to AddLine. It is possible that the line segment has to be split first.

The splitting of line segments has a serious drawback. If we have n line segments in a scene, then it is possible that we end up with $O(n^2)$ (Fuchs *et al.* 1980) nodes in the tree. It will be clear that this is unacceptable in GIS applications, in which we typically deal with 10 000 or more line segments. However, this is a worst case situation and the actual number of nodes will not be that large (see § 6). The BSP-tree is intended for interactive applications in which fast responses are required. Minimizing the memory usage is considered less important and so we have not paid much attention to it. However, the typical memory usage is about three to four times the size of the original map data. There are three reasons for this: the memory space required for pointers, the splitting of line segments and the replication of points, as the end of one line segment is often the beginning of another.

2.2. *The object BSP-tree*

The BSP-tree, as discussed so far, is suited only for storing a collection of (unrelated) line segments. In a modelling system it must be possible to represent a closed object, for example, (the interior of) a polygon in the 2D case or a polyhedron in the 3D case. The object BSP-tree is an extension to the BSP-tree to cater for object representation. It stores the line segments that together make up the boundary of the polygon. The object BSP-tree has explicit leaf nodes which do not contain line segments to split the sub-spaces any further. The leaf nodes correspond to the convex sub-spaces created by the BSP-tree. A boolean in a leaf node indicates whether the convex sub-space is inside or outside the object.

At the University of Leiden, we used the object BSP-tree in the 3D graphics modelling system HIRASP (Teunissen and van den Bos 1988). Because of the spatial organization, the hidden surfaces can be 'removed' in $O(n)$ time, with n the number of polygons in the tree (Teunissen and van Oosterom 1988). The object BSP-tree is also well suited to perform the set operations (Thibault and Naylor 1987): union, difference and intersection, as used in constructive solid geometry (CSG) systems. The map overlay operation in a GIS (van Oosterom 1988) has strong relationships with these set operations.

2.3. *The multi-object BSP-tree*

We wish to exploit the spatial organization properties of the BSP-tree in a GIS. In a GIS we usually deal with 2D maps. The line segments of the original data base are used to split the space in a recursive manner. By using data inherent to the problem to organize the space, we expect a good spatial organization. Maps always contain multiple objects; for example, countries on the map of Europe. Because we deal with multiple objects, we have to modify the concept of the object BSP-tree previously discussed. Instead of a boolean, the leaf nodes now contain an identifier (name). This identifier indicates the object to which the convex sub-space, represented by the leaf node, belongs. We call this type of BSP-tree the multi-object BSP-tree.

Figure 3 (*a*) presents a 2D scene with two objects, triangle T with sides ABC, and rectangle R with sides DEFG. The method divides the space in the convex sub-spaces of figure 3 (*b*). The BSP-tree of figure 3 (*c*) is extended with explicit leaf nodes, each representing a convex part of the space. If a convex sub-space corresponds with the 'outside' region, then no label is drawn in figure 3 (*c*). If no more than one identification per leaf is allowed, then only mutually-exclusive objects can be stored in the multi-object BSP-tree, otherwise it is also possible to deal with objects that overlap. A disadvantage of this BSP-tree is that the representation of one object is scattered over

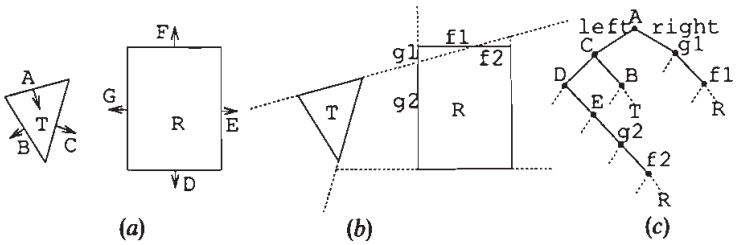

Figure 3. The building of a multi-object BSP-tree: (*a*) object scene, (*b*) sub-spaces, (*c*) BSP-tree.

several leaves, e.g., rectangle R in figure 3. The following list summarizes the properties of the multi-object BSP-tree:

(1) Each node in the tree corresponds with a convex sub-space.
(2) Each internal node splits the convex sub-space into two convex parts: left and right. Further down the tree, the convex sub-spaces become smaller. Each internal node contains one line segment.
(3) Each leaf node corresponds with a convex sub-space which will not be split. A leaf node does not contain a line segment but it does contain an object identification.

3. The basic spatial operations

In this section we will explain how the (multi-object) BSP-tree is used in implementing two spatial operations: the pick and the rectangle search.

3.1. *The pick operation*

Consider a system that displays a map on the screen. The user generates a point $P=(x, y)$ with an input device such as a mouse or tablet. He wants to know which object he pointed at. To solve this problem we locate point P by descending the tree until a leaf node is reached. This leaf node contains the identification of an object. Descending the tree is quite simple: if at an internal node point P lies on the left side of the line segment, then the left branch is followed, otherwise the right branch is followed. This strategy results in one straight path from the root to a leaf node. So, in the case of a balanced tree with n internal nodes, the search takes $O(\log n)$ time (see § 5). This is probably the best result one could wish.

3.2. *The rectangle search*

In many applications the user wants to select all objects within a certain rectangle R. The rectangle search is also necessary during the display of (a part of) a map on a rectangular screen. Basically, the traversal of the tree is the same as in the pick operation. At an internal node, the left branch is followed if there is an overlap between rectangle R and the left sub-space; the right branch is followed if there is an overlap between the right sub-space and the rectangle R. If there is overlap with both sub-spaces, then both branches must be followed. A simple recursive function accomplishes this traversal.

The operations are efficient because parts of the tree are skipped. In an unstructured collection of data we would have to visit every item and test if we 'accept' this item based on its geometric properties. Using the BSP-tree we do not have to examine the data that are outside our region of interest.

4. The detail levels

We need detail levels, as argued in the introduction, if we want to build a usable interactive GIS. The detail levels must not introduce redundant data storage and must be combined with the spatial data structure. Not only must the geometric data be organized with detail levels but the same applies to the related application data (non-geometric). However, we will focus our attention on the geometric data.

We first make an observation of the BSP-tree created with the function AddLine. A line segment inserted early on ends up in one of the top levels of the BSP-tree. A line segment inserted later on must first 'travel down' the tree (and if necessary be split a few

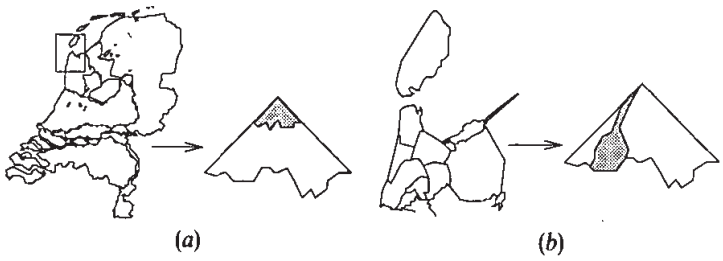

Figure 4. The place of global and detailed data: (*a*) the global data, (*b*) the details.

times) before it reaches the correct position on a lower level of the BSP-tree. We use this property to create a reactive BSP-tree. If the global data are inserted first in the BSP-tree, they will end up in the higher levels of the BSP-tree. The local data (detail) are added later, so they end up in the lower levels of the BSP-tree. Figure 4 depicts this situation for a map of The Netherlands. The rectangle in the global map shows the position of the detailed map. The mountain-like object represents the entire BSP-tree and the grey region stands for the part of the BSP-tree that contains the data of the corresponding map.

We will use a case involving the boundaries of administrative units, to illustrate the way the reactive BSP-tree functions. In The Netherlands there are six hierarchical levels of administrative units, ranging from the municipalities (the lowest level) to the whole country (the highest level). We insert the boundaries of the administrative units in the BSP-tree, starting with the highest level, then the next to highest level, and so on. During the display of the map, the number of detail levels shown depends on the size of the selected region. The larger the region we want to display, the fewer detail levels will be shown. A heuristic rule for this is: the total amount of geometric data to be displayed should be constant measured by the number of coordinates involved.

The BSP-tree is traversed with an adapted 'rectangle search'-algorithm, to display all objects in a certain region up to a certain detail level. The algorithm has to know where one detail level stops and where the other begins. This can be achieved by extending the BSP-tree in one of the following manners:

(1) Add to each node a label with the corresponding detail level. If, during the traversal of the BSP-tree a detail level is reached that is lower than the one in which we are interested, then we can skip this branch because it contains only data of a lower level.
(2) After inserting the global data (highest level) into the BSP-tree, add special nodes, called level STOP nodes, to the BSP-tree. The level STOP nodes contain no splitting line segment and can be compared with the leaf nodes of the multi-object BSP-tree (see § 2.3). Then the next highest level is added to the BSP-tree, again followed by level STOP nodes. This process is repeated for each detail level. Figure 5 shows a reactive BSP-tree with two detail levels.

A drawback of the reactive BSP-tree is that it supports only a part of the map generalization process (Shea and McMaster 1989). It removes unimportant lines but it draws an important line with the same number of definition points on every scale. As far as we know, there is no elegant solution to this problem in the context of BSP-trees. It is possible to store a generalized version of a line at multiple detail levels in the same BSP-tree. However, the storage of the same line at multiple levels introduces unwanted

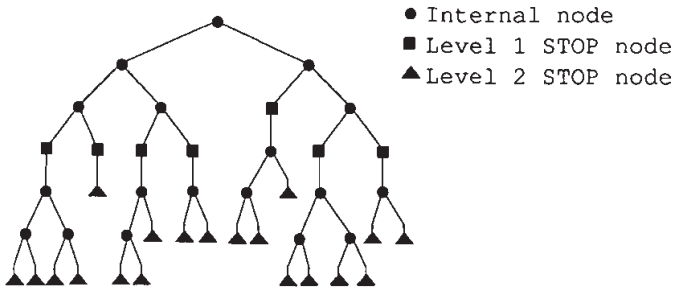

Figure 5. The reactive BSP-tree.

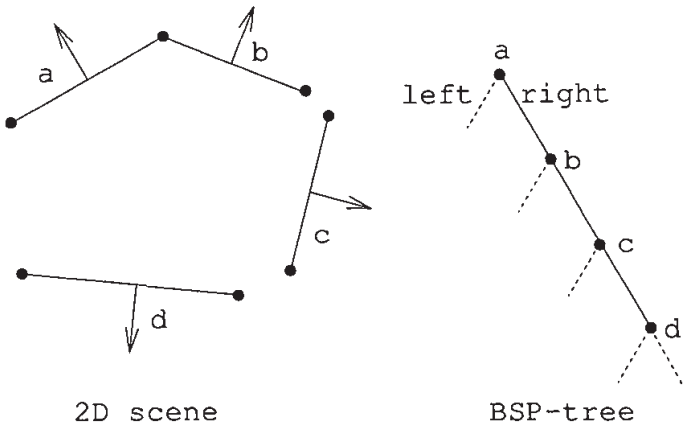

Figure 6. Unbalanced BSP-tree.

redundancy. The generalized version of a line can be computed specially for every level with a line generalization algorithm, for instance, with the Douglas–Peucker algorithm (Douglas and Peucker 1973).

In another paper (van Oosterom 1989) we have described some uses of the reactive BSP-tree in thematic mapping and showed how choropleth maps and prism maps (Franklin and Lewis 1978) can be produced.

5. Balancing the BSP-tree

The arguments in the previous sections assume a balanced BSP-tree but the algorithm in figure 2 will not necessarily generate such a BSP-tree. In fact, in some situations it is impossible to generate a balanced BSP-tree, see for example the 'convex scene' of figure 6. We can solve this only by inserting first some invisible auxiliary splitting line segments, for example, a line with line segments a and b to the left and c and d to the right (not drawn in figure 6).

A balanced BSP-tree might result in a tree with more nodes because of the splitting process. Sometimes, a slightly less balanced BSP-tree with fewer split line segments is to be preferred. This raises the question: 'What is the best BSP-tree for GIS?' There is no easy answer to this question but as long as both the measure in which the tree is out of balance and the number of split line segments remain within 'reasonable' bounds, it is our experience that the BSP-tree is well suited for GIS applications. The next two subsections describe several strategies for balancing BSP-trees in the static and the

dynamic case respectively. Dynamic balancing is not as important as in many other applications that use balanced trees, because the maps in most GIS applications are static.

5.1. *Static balancing*

In our implementation the line segments are, per detail level, inserted in the same order as they are stored in the original map file. In the case of the map of The Netherlands, this results in a region by region insertion of the map data into the BSP-tree. For example, if the northernmost region is inserted first, then the paths that correspond with the area above the northernmost region will not grow when inserting the other regions. In this manner the BSP-tree gets out of balance. A simple solution is to insert the line segments, per detail level, in truly random order.

Another, more expensive, solution for balancing the BSP-tree is taken from Fuchs *et al.* (1983). A few potential roots for the tree are tried and the one that gives a balanced division is selected. Fuchs uses this solution in combination with the original (non-incremental) version for building a BSP-tree. Balancing the BSP-tree and minimizing the number of splits are two objectives that do not always agree. Thibault and Naylor (1987) describe some heuristics for evaluating the candidates.

A different approach is to insert first a few auxiliary split lines, which try to divide the space in a fair manner. The map data line segments are inserted after the auxiliary lines and end up in the proper regions. We mark the auxiliary lines as invisible. A disadvantage of the auxiliary lines is that they themselves may cause the split of line segments. However, experience has shown that this number of splits is relatively small. Sometimes the total number of nodes is even slightly reduced (though not significantly), in spite of the insertion of auxiliary split lines. Assume that the map data space is $\{(x, y) | 0 \leqslant x \leqslant 1 \wedge 0 \leqslant y \leqslant 1\}$, we first insert the lines $x = \frac{1}{2}$ and $y = \frac{1}{2}$, then $x = \frac{1}{4}$, $x = \frac{3}{4}$, $y = \frac{1}{4}$ and $y = \frac{3}{4}$, and so on. Together, these auxiliary split lines form a coarse raster. Note that the order in which they are inserted is important. The insertion of 15 horizontal and 15 vertical lines reduced the (maximum) depth of a BSP-tree of The Netherlands (map 1, see § 6) from 80 to 35. The disadvantage of these auxiliary split lines is that they still result in unbalanced trees if the distribution of the map data is not uniform.

A more radical approach is first building a K-dimensional (KD) tree with large bucket size (e.g., 100–1000). KD-trees, which are clearly described by Bentley (1975), are balanced trees for storing points. The KD-tree divides a set of points into two sub-sets of (nearly) equal size by either a horizontal or a vertical line in the 2D case. This is recursively repeated for each sub-set until it is small enough. Because the KD-tree is only suitable for storing points, it is built from the points that define the line segments. Figure 7 shows the KD-tree of a map that contains about 30 000 points. The split lines in the KD-tree are the auxiliary lines for the balanced BSP-tree, and the KD-tree is thrown away.

We could also use a generalized version of the KD-tree (see figure 8), which does not always split along one of the main axes. The BSP-tree is already suitable for storing split lines that have an arbitrary orientation. So, it might be better to split along a line orthogonal to the 'best' fit line. The set of points consists of $p_i = (x_i, y_i)$ for i from 1 to n. The general form of a line l that makes an angle α with the positive x-axis is: $x \sin\alpha - y \cos\alpha = c$. The distance from point p_i to line l is: $|x_i \sin\alpha - y_i \cos\alpha - c|$. We should minimize the function

$$f(\alpha, c) = \sum_{i=1}^{n} (x_i \sin\alpha - y_i \cos\alpha - c)^2$$

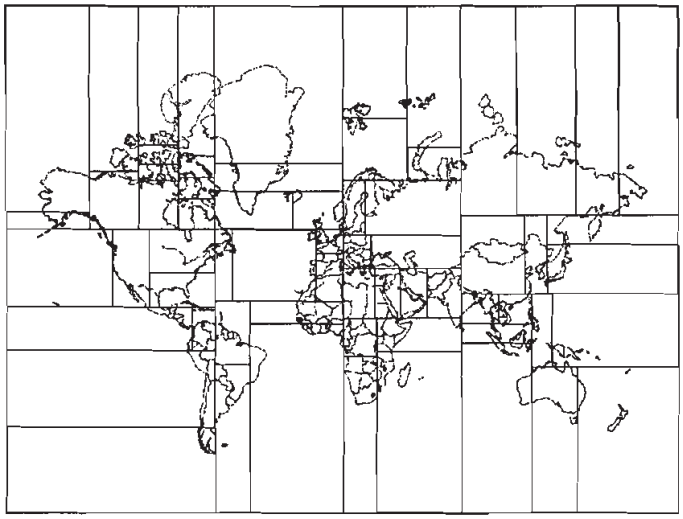

Figure 7. KD-tree with large bucket size.

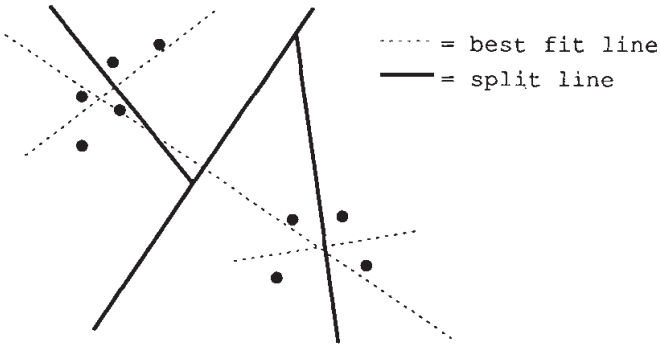

Figure 8. Generalized KD-tree.

With means

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i,$$

variances

$$\sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu_x)^2, \quad \sigma_y^2 = \frac{1}{n}\sum_{i=1}^{n} (y_i - \mu_y)^2$$

and covariance

$$\mathrm{Cov}(x, y) = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y)$$

defined in the usual manner (Mood *et al.* 1974) we get

$$\alpha = \tfrac{1}{2}\arctan\left(\frac{2\,\mathrm{Cov}(x, y)}{\sigma_x^2 - \sigma_y^2}\right), \quad c = \mu_x \cos\alpha + \mu_y \sin\alpha$$

for $\sigma_x^2 > \sigma_y^2$. In the case that $\sigma_x^2 < \sigma_y^2$, $\frac{1}{2}\pi$ must be added to α. If both variances are equal, then $\alpha = \pm\frac{1}{4}\pi$ depending on the sign of the covariance. The points are sorted according

to the position of their projections on line l. All points up to the median are put in the left sub-space and the others are put in the right sub-space. This process repeats itself for all sub-spaces until they contain fewer points than the bucket-size. This results in a perfectly balanced tree for storing points in $O(n(\log n)^2)$ time. The sorting requires some extra pre-processing time. If the set is split into two parts by using a split line through (μ_x, μ_y) and orthogonal to line l, then the building of the generalized KD-tree takes $O(n \log n)$ time. However, this does not necessarily result in a perfectly-balanced tree.

Paterson and Yao (1989) prove that if the original line segments are disjoint, then it is possible to build a BSP-tree with $O(n \log n)$ nodes and depth $O(\log n)$ using an algorithm requiring only $O(n \log n)$ time. They (recursively) use a horizontal auxiliary split line defined by the median value of the y-coordinate, and, with the notion of 'free cuts', they prove their theorem. Their results can easily be generalized to line segments representing a map and touching each other only in begin and end points. Better partitions may be achieved by using more general auxiliary split lines, e.g., lines similar to the ones of the generalized KD-tree. It is doubtful whether this BSP-tree can be built in $O(n \log n)$ time. Among other difficulties, the detection of free cuts will become harder.

5.2. *Dynamic balancing*

The emphasis in this subsection is on inserting line segments in a BSP-tree, while keeping it balanced. Deleting and changing lines segments is less important, because they occur less frequently in GIS applications. Even without considering the balance of the BSP-tree, deleting a line segment can be very difficult. If the line segment is the root of a (sub) BSP-tree, then the replacement of this root by another line segment affects the whole sub-tree in a drastic manner. This is not a problem in the case of an empty or a very small sub-tree but otherwise this could require the complete rebuilding of the sub-tree. A deletion can be simulated by making the line segment invisible, as with an auxiliary line. It will be clear that this is not a practical solution when the number of deletes and changes is relatively large compared to the actual number of line segments.

For dynamic balancing, the nodes in the BSP-tree have to be extended with information about their balancing status. This is a single integer that contains for example the value of the expression #NodesLeft − #NodesRight. The dynamic insertion of a line segment starts in the same manner as in the normal situation, i.e., with the function AddLine, see figure 2. During the insertion the balance status of the visited nodes has to be updated. However, because of this insertion it is possible that nodes on the path from the root to the new leaf get out of balance. Note that, in the case of a split line segment, there are several leaves that correspond with the new line segment. There may thus be multiple paths from the root that have to be considered during the restoration of the balance and, as a consequence, the weight associated with sub-trees may increase with more than one.

In order to restore the balance, a sub-tree that corresponds with an unbalanced node has to be reorganized. A solution might be to perform a complete rebuilding of a sub-tree based on (exhaustive) search for a good root in the sub-tree. This could be done in a way comparable with the method Fuchs describes to balance the BSP-tree. This is not only very time-consuming but is even impossible, in the case of 'convex scenes', as was explained in the introduction of this section. Therefore, we decided to use another technique.

The root of the new sub-tree is an auxiliary line and it is made in a similar way to that in which a line in the generalized KD-tree is created. If, during the calculation of

this auxiliary line, a point that is an end-point of more than one line segments is also counted more than once, then the numbers of line segments to the left and to the right are equal. In order to preserve as much as possible of the (balanced) structure of the old sub-tree, we should try to move as large parts as possible from the old to the new sub-tree. To simplify the test whether a part fits in the new sub-tree, a circle is stored in each internal node of the BSP-tree. The centre lies halfway along the line segment and the radius is the smallest value such that all line segments of the sub-tree lie within the circle. The computational speed is further increased by storing the square of the radius instead of the radius itself. This circle, together with the BSP-tree structure of the new sub-tree, makes it easier to move parts of the old sub-tree.

6. Practical results

In this section we present the first results of our implementation. Note that this is only a prototype GIS and not all functions are present yet. The prototype is a 'main memory implementation', that is, the complete map is stored in a data structure of a running program. Especially for large data sets, it would be useful to perform a redesign of the prototype in order to minimize the number of disk accesses during a tree search. The most obvious strategy is to take a group of nodes (a small sub-tree) of the BSP-tree and store it on one disk-page. If each disk-page is considered to be one node in the 'external implementation', then the resulting structure is a multi-way tree. Probably, this structure will resemble the cell tree described by Günther (1988) and Günther and Bilmes (1988).

Fuchs *et al.* (1980) show that n line segments may result in the worst case in $O(n^2)$ line segments in the BSP-tree because of the splitting process. This happens when the line segments are, relative to each other, long and have unfortunate orientations and positions. The insertion of one (long) line segment results in many new leaves in the BSP-tree. In a balanced tree this is no problem for the query time: $Q(n) = O(\log n^2) = O(2 \log n) = O(\log n)$. However, it results in an enormous storage requirement: $S(n) = O(n^2)$. This is unacceptable in the case of GIS in which n is typically very large, e.g., 10 000–100 000. As already noted in the previous section, in the case of disjoint line segments the storage requirement is $S(n) = O(n \log n)$.

How will the BSP-tree behave when we insert very large amounts of irregular geometric data? In contrast with the worst case, we expect that the number of splits in the practical GIS situation to be far less because the line segments are relatively short. We are interested in the size and the performance of BSP-trees built with real map data. Map 1 is the map of The Netherlands as drawn in figure 4. Map 2 contains the data from World Data Bank I. The area and line features from DLMS DFAD (DMA 1977) are used in map 3. The latitude ranges from 52° 12′ to 52° 24′ and the longitude from 5° 30′ and 6° 00′; this is the region hear Harderwijk in The Netherlands. These three maps are from completely different sources but they produce very similar test results. The table shows some of the key figures when no measures for balancing are taken.

The expansion factor is defined by $f = L/I$ and the theoretic minimum depth by $d_{th} = \lceil {}^2\log L \rceil$. The maximum depth d_{max} and the average depth d_{avg} are measured values. There is a simple relationship between the number of leaves in the BSP-tree and the number of inserted, degenerated, and split lines: $L = I - D + S + 1$. This is due to the property of binary trees that 'the number of external nodes is one more than the number of internal nodes' and is corrected for split-line segments and degenerated line-segments. Degenerated line-segments are line segments in the original data set with the

	Map 1	Map 2	Map 3
Inserted line segments (I)	13350	108966	5456
Degenerated line segments (D)	3	50	4
Split line segments (S)	6313	48173	2586
Leaves (L)	19661	157090	8039
Expansion factor (f)	1·47	1·44	1·47
Max depth (d_{max})	80	234	60
Average depth (d_{avg})	27·7	63·2	30·5
Theoretic depth (d_{th})	15	18	13

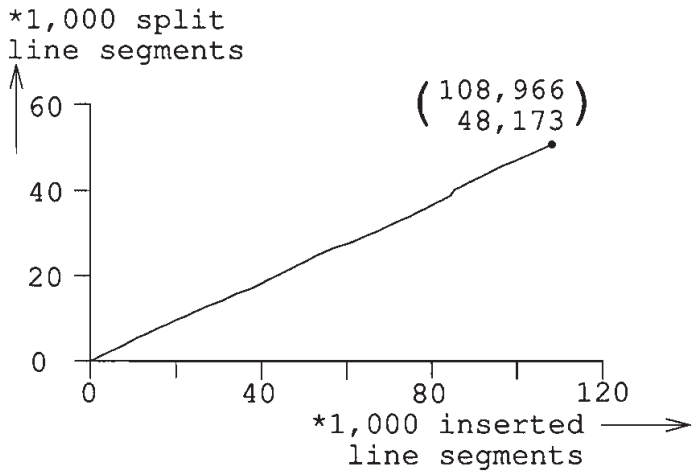

Figure 9. Split lines as function of inserted lines.

end point equal to the begin point or at least within a distance smaller than a relative accuracy epsilon, as used by our program.

Figure 9 shows the number of split line segments as a function of inserted line segments for the data from World Data Bank I. One might expect that the more line segments are already inserted in the BSP-tree, the bigger the chance that a new line segment has to be split. However, this is not true. The straight line in figure 9 means that the chance that a new line segment has to be split is independent of the number of line segments already inserted. This is a remarkable result because it implies that BSP-trees of real maps have linear $O(n)$ storage space complexity in contrast to the worst case quadratic $O(n^2)$ and the $O(n \log n)$ for disjoint line segments. The constant associated with this $O(n)$ storage space complexity is modest and stable, somewhere between 1·4 and 1·5. An intuitive explanation for this is that the line segments have some 'point-like' characteristics because they are small compared to the whole map. When the line segments reach their final position they gradually get back the line characteristics. We must verify this with more maps from different independent sources. Another approach to proving this $O(n)$ storage space complexity is the development of a statistical model.

7. Conclusion

The data structure presented is one of the few that combines two difficult requirements: spatial organization and levels of detail. Because of its generality it enables incorporation of other spatial organization techniques in the BSP-tree, e.g., the

raster structure, the quadtree (Samet 1989) or the KD-tree (Bentley 1975). A surprising result of our implementation is that BSP-trees of real maps seem to have no more than about $1{\cdot}5*n$ nodes instead of the worst case $O(n^2)$ nodes where n is the number of line segments in the map.

We are aware of the fact that the reactive BSP-tree is far from perfect but we hope that it serves as a source of inspiration to generate more ideas. One important guideline we have derived from our work on the BSP-tree is that, in a spatial organization tree, the more important objects must be stored in the higher levels of the tree. Using the insertion order, as done in § 4, is not a very elegant method to achieve this because the solution is limited to static data. A reactive data structure (van Oosterom 1989) need not be based on a BSP-tree; other solutions are possible. We are also working on the development of a reactive data structure based on an object-oriented approach to GIS (van den Bos and Laffra 1989, van Oosterom and van den Bos 1989).

Acknowledgments

Our thanks go to TNO Physics and Electronics Laboratory, P.O. Box 96864, 2509 JG The Hague, The Netherlands, employer of Peter van Oosterom, for giving him the opportunity to perform this research at the Department of Computer Science, University of Leiden. Andre Smits of TNO, Jan van den Bos, Chris Laffra, Hans Jense and Dony van Vliet of the University of Leiden made many valuable suggestions in reviewing this document.

References

BALLARD, D. H., 1981, Strip trees: A hierarchical representation for curves. *Communications of the Association for Computing Machinery*, **24**, 310.

BENTLEY, J. L., 1975, Multidimensional binary search trees used for associative searching. *Communications of the Association for Computing Machinery*, **18**, 509.

CHRISMAN, N. R., 1989, Spatial indexing schemes for GIS. *International Journal of Geographical Information Systems* (submitted).

DMA, 1977, Product specifications for digital landmass system (DLMS) data base. Technical report, Defense Mapping Agency Aerospace Center, St. Louis, Missouri.

DOUGLAS, D. H., and PEUCKER, T. K., 1973, Algorithms for the reduction of points required to represent a digitized line or its caricature. *Canadian Cartographer*, **10**, 112.

FUCHS, H., ABRAM, G. D., and GRANT, E. D., 1983, Near real-time shaded display of rigid objects. *Computer Graphics*, **17**, 65.

FUCHS, H., KEDEM, Z. M., and NAYLOR, B. F., 1980, On visible surface generation by *a priori* tree structures. *Computer Graphics*, **14**, 124.

FRANKLIN, WM. R., and LEWIS, H. L., 1978, 3D graphic display of discrete spatial data by PRISM maps. *Computer Graphics*, **12**, 70.

GÜNTHER, O., 1988, *Efficient Structures for Geometric Data Management* (Lecture Notes in Computer Science, No. 337) (Berlin: Springer-Verlag).

GÜNTHER, O., and BILMES, J., 1988, The implementation of the cell tree: Design alternatives and performance evaluation. Technical Report TRCS88-23, University of California, Santa Barbara.

GUPTILL, S. G., 1989, Speculations on seamless, scaleless cartographic data bases. *Proceedings of AutoCarto 9 held in Baltimore, Maryland, on 2–7 April 1989*, edited by K. E. Anderson (Falls Church: American Society Photogrammetry and Remote Sensing, American Congress of Surveying and Mapping), pp. 436–443.

GUTTMAN, A., 1984, R-trees: A dynamic index structure for spatial searching. *Association for Computing Machinery SIGMOD*, **13**, 47.

JONES, C. B., and ABRAHAM, A. M., 1987, Line generalisation in a global cartographic database. *Cartographica*, **24**, 32.

MOOD, A. M., GRAYBILL, F. A., and BOES, D. C., 1974, *Introduction to the Theory of Statistics* (Tokyo: McGraw-Hill).

PATERSON, M. S., and YAO, F. F., 1989, Binary partitions with applications to hidden-surface removal and solid modelling. *Proceedings 5th Association for Computing Machinery Symposium on Computational Geometry* (New York: Association for Computing Machinery Press), p. 23.

SAMET, H., 1989, *The Design and Analysis of Spatial Structures* (Reading: Addison-Wesley).

SHEA, K. C., and MCMASTER, R. B., 1989, Cartographic generalization in a digital environment: When and how to generalize. *Proceedings of AutoCarto 9 held in Baltimore, Maryland, on 2–7 April 1989*, edited by K. E Anderson (Falls Church: American Society Photogrammetry and Remote Sensing, American Congress of Surveying and Mapping), p. 56.

THIBAULT, W. C., and NAYLOR, B. F., 1987, Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, **21**, 153.

TEUNISSEN, W. J. M., and VAN DEN BOS, J., 1988, HIRASP a hierarchical interactive rastergraphics system based on pattern graphs and pattern expressions. *Proceedings of Eurographics held in Amsterdam, The Netherlands, on 24–28 August 1987*, edited by G. Maréchal (Amsterdam: North-Holland), p. 393.

TEUNISSEN, W. J. M., and VAN OOSTEROM, P. J. M., 1988, The creation and display of arbitrary polyhedra in HIRASP. Technical Report 88-20, University of Leiden, Department of Computer Science, Leiden.

VAN DEN BOS, J., and LAFFRA, C., 1989, PROCOL: A parallel object language with protocols. *SIGPLAN Notices*, **24**, 95 Conference Proceedings OOPSLA '89.

VAN OOSTEROM, P., 1989, A reactive data structure for Geographic Information Systems. *Proceedings of AutoCarto 9 held in Baltimore, Maryland, on 2–7 April 1989*, edited by K. E. Anderson (Falls Church: American Society Photogrammetry and Remote Sensing, American Congress on Surveying and Mapping), pp. 665–674.

VAN OOSTEROM, P., and VAN DEN BOS, J., 1989, An object-oriented approach to the design of Geographic Information Systems. *Computers and Graphics*, **13**, 409.

WAUGH, T. C., and DOWERS, S., 1988, High-speed interactive mapping in large topographic databases. *Proceedings of Eurocarto Seven held in Enschede, The Netherlands, on 20–22 September 1988*, edited by J.-C. Muller (Enschede: ITC Publication), p. 97.