Modifications of the Binary Space Partitioning Tree for Geographic Information Systems

P.J.M. VAN OOSTEROM, TNO Physics and Electronics Laboratory, 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 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 detail levels. This tree is one of the few spatial data structures that does not organize space in a rectangular manner. An application of the reactive data structure in thematic mapping is given. A prototype system is being 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* (GISs). There are many applications that use GIS technology, among them: 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. However, most existing systems lack these data structures. A *Reactive Data Structure* is a data structure with the following two properties:

- *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 [19, 3]. Several spatial data structures are described in the literature and are implemented in existing GISs.
- *Detail levels:* Too much details on the display will hamper the operator's perception of the important information. Also, unnecessary details 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 details (new objects). Conversely, when zooming out, it removes fine details from the display. We call this operation *logical zoom* as opposed to the ordinary zoom which only changes the size of objects. There is some literature on data structures with detail levels, for instance strip trees [1] and multi-scale line-trees [12].

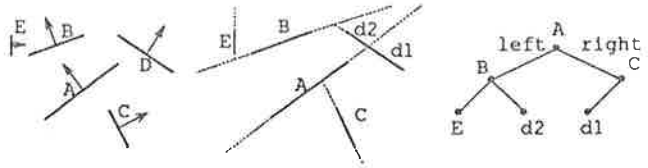
Reactive data structures are the basic building blocks for seamless, scaleless geographic databases[11]. The data structure presented in this paper is a modification of the binary space partitioning (BSP) tree. A short description of the original BSP-tree is given in section 2, together with some minor 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. Section 5 gives an typical application of the reactive data structure. The balancing of the BSP-tree is discussed in section 6 for both the static and the dynamic case. Section 7 contains the first practical results from our implementation. Finally, the pros and cons are discussed in section 8.

2 The BSP-tree and some variations on it

2.1 The original BSP-tree

The original use of the Binary Space Partitioning (BSP) tree was in 3D Computer Graphics [8, 7]. 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

*Also: Department of Computer Science, University of Leiden, The Netherlands, Email: OOSTEROM@HLERUL5.BITNET.



a. 2D scene b. convex sub-spaces c. BSP-tree

Figure 1: The building of a BSP-tree

BSP-tree itself represents a organization of space by a set *convex* sub-spaces in a binary tree. This tree is useful during spatial search and other spatial operations. Figure 1a 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 1b. This process is repeated for each of the two sub-spaces with 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 1b shows the resulting organization of the space, as a set of (possibly open) convex sub-spaces. The corresponding BSP-tree is drawn in Figure 1c. In the 3D case supporting planes of flat polygons are used to split the space instead of lines.

The choice which line segment to use for dividing the space, very much influences the building of the tree. It is preferred 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 Pascal-like code of a program that builds a BSP-tree. The program *BuildTree* is a variation of the traditional method (non-incremental) for building a BSP-tree [8]. 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 current 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

```
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

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 $O(n^2)$ [8] 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 section 7.

2.2 The object BSP-tree

The BSP-tree, as discussed so far, is only suited for storing a collection of (unrelated) line segments. In a modeling 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 a splitting line segment. These leaf nodes only correspond with 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 modeling system HIRASP [15]. Because of the spatial organization, the hidden surfaces can be "removed" in $O(n)$ time with n the number of polygons in the tree [16]. The object BSP-tree is also well suited to perform the set operations [17]: union, difference and intersection, as used in Constructive Solid Geometry (CSG) systems. The *map overlay operation* in a GIS (described in [19]) has strong relationships with these set operations.

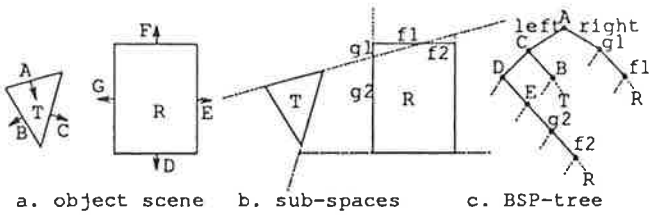


Figure 3: The building of a multi-object BSP-tree

2.3 The multi-object BSP-tree

We want to exploit the spatial organization properties of the BSP-tree in a Geographic Information System. 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 already discussed object BSP-tree. Instead of a boolean, the leaf nodes now contain an identification (name). This identification tells to which object the convex sub-space, represented by the leaf node, belongs. We call this type of BSP-tree the *multi-object BSP-tree.*

Figure 3a 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 3b. The BSP-tree of Figure 3c 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 3c. A disadvantage of this BSP-tree is that the representation of one object is scattered over several leaves. See for example rectangle R in Figure 3. The following list summarizes the properties of the multi-object BSP-tree:

- Each node in the tree corresponds with a convex sub-space.
- 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.
- 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, else the right branch is followed. This strategy results in one straight path from the root to a leaf node. So, in case of a balanced tree with n internal nodes, the search takes $O(log\ n)$.

3.2 The rectangle search

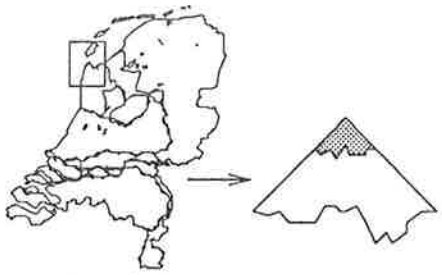
In many applications the user wants to select all objects within a certain rectangle R. The rectangle search is also necessary during the display of (a part of) a map on a rectangular screen Basically, the traversal of the tree is the same as in the pick operation. At an internal node, the left branch is followed if there is an overlap between rectangle R and the left sub-space. And, of course, 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 don't have to examine the data that are outside our region of interest.

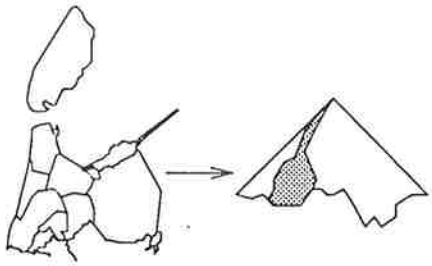
4 The detail levels

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

We first make an observation of the BSP-tree created with the function AddLine. An early inserted line segment 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 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



a. The global data



b. The details

Figure 4: The place of global and detailed data

data (details) 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" represents the entire BSP-tree and the gray region stands for the part of the BSP-tree that contains the data of the corresponding map.

We will use a case to illustrate the way the reactive BSP-tree functions. The case deals with the boundaries of administrative units. 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 store the boundaries of these administrative units in the BSP-tree, starting with the highest level, then the next to highest level, and so on. When we display this map, the number of detail levels depends on the map scale. That is, if we assume the size of the screen fixed, the size of the region we want to display. The larger the region we want to display, the less detail levels will be shown. A heuristic rule: the total amount of geometric data to be displayed is constant.

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:

- 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 we are interested in, then we can skip this branch, because it contains only data of a lower level.
- 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 section 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.

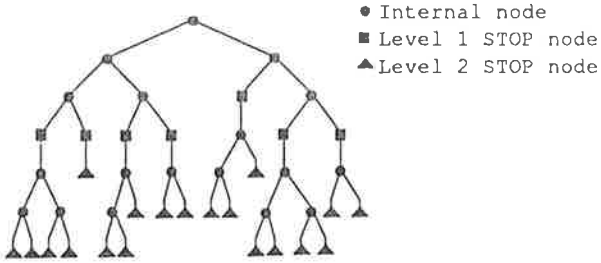


Figure 5: The reactive BSP-tree

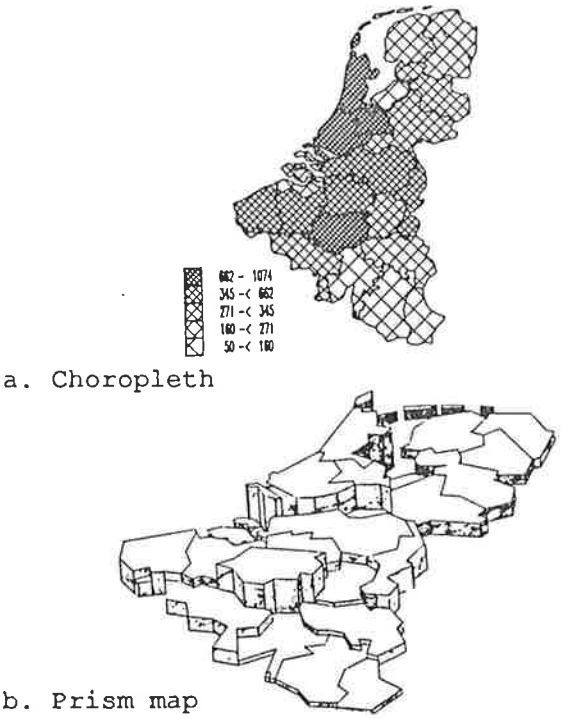
A drawback of the reactive BSP-tree is that it only supports a part of the map generalization process[14]. It removes unimportant lines, but it draws important lines with the same number of points on every scale. As far as we know, there is no elegant solution to this problem. It is possible to store 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 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 [5].

5 An application

In this section we describe some additional uses of the reactive data structure in thematic mapping. We will expand the case of the previous section, to make it possible to visualize census data of administrative units. We show how a choropleth map and a prism map can be produced. Figure 6 gives an example of these map types. We use the reactive BSP-tree with the level STOP nodes. A level STOP node corresponds with a convex part of an administrative unit at that level. The identification of the administrative unit is stored in the level STOP node. The census data is not stored in the BSP-tree, because the BSP-tree scatters objects (administrative units) over several leaves, see section 2. The census data is available at each detail level.

5.1 Choropleth

After the user has decided which region and which census variable has to be displayed, the GIS determines the de-



a. Choropleth

b. Prism map

Figure 6: Two map types for thematic mapping

tail level. A choropleth map colors administrative units depending on their value of the wanted census variable. All we have to do to produce a choropleth map, is traverse the BSP-tree for the selected region and level. When we reach a level STOP node of the desired level, we know to which administrative unit the corresponding convex sub-space belongs. The required census variable is retrieved and the convex sub-space is filled with the right color.

The BSP-tree does not offer an explicit representation of the convex sub-spaces. This is solved by maintaining a *temporary data structure* during the traversal of the BSP-tree. This temporary data structure represents the (open) convex sub-space that corresponds with the current node. Each time we take a step down in the BSP-tree the temporary data structure is updated. At depth k the temporary data structure represents a convex polygon with no more than k edges. So, a step downwards from level k to level $k+1$ takes $O(k)$ time, the insertion of a new edge in a convex polygon with k edges. The steps upwards take no processing time, because the intermediate results are stored along the current path in the BSP-tree. In the case of a balanced BSP-tree, the height of the tree is $O(\log n)$ with n the number of line segments (nodes) in the BSP-tree. Summing all steps for the whole BSP-tree results in $O(n \log n)$ processing time. So, displaying the whole BSP-tree while coloring the convex sub-spaces takes $O(n \log n)$ time. It is possible to store the explicit representation of the convex sub-space in the level STOP node. This reduces the time to generate the choropleth to $O(n)$, but increases the storage requirements.

5.2 Prism map

The prism map [6] is an attractive map to look at and it offers the possibility to display an extra variable through the height of the prisms. A prism map is a set of 3D-objects. Before the prism map is generated, the user has to indicate from which direction he wants to look at the prisms.

Basically, we produce the prism map in the same manner as the choropleth map. Instead of coloring the convex sub-space, we lift it up to the desired height. If the convex sub-space has k sides, then each side will result in a 3D rectangular polygon. Together with the top of the prism, this results in $k+1$ 3D polygons, which must be displayed. Before a polygon is displayed it is projected from 3D to 2D, in order to calculate the actual coordinates on the screen. A number of convex prisms form one prism on the map, as the same convex sub-spaces form together the administrative unit. This means that the "internal" sides of the prism need not be drawn. We can recognize the internal sides if we label those sides of the convex sub-spaces that are part of line segments.

The "hidden surface" problem is usually quite difficult and time consuming to solve. However, if we slightly change the way in which the BSP-tree is traversed, the hidden surface problem is solved easily (in combination with the Painters-algorithm). The different traversal does not cost any extra processing time and ensures that prisms farther away from the viewing point are drawn first. This results in the "removal" of the hidden surfaces of the prism map. For more details on this topic see [16].

Normally, the entire BSP-tree is traversed in $O(n)$, but we have to maintain the temporary data structure that contains the explicit representation of the current convex sub-space. So, we can produce a prism map of the whole scene in $O(n \log n)$. If explicit representations of the convex sub-spaces are stored, then a prism map can be produced in $O(n)$, which is quite fast. This fast response stimulates the end-user to take other views of the data.

6 Balancing the BSP-tree

The arguments in the previous sections assume a balanced BSP-tree. The algorithm in Figure 2 will not necessarily generate a balanced BSP-tree. In fact, in some situations it is impossible to generate a balanced BSP-tree, see for example the "convex scene" of Figure 7. 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 7).

A balanced BSP-tree might result in a tree with more nodes because of the splitting process. Sometimes, a

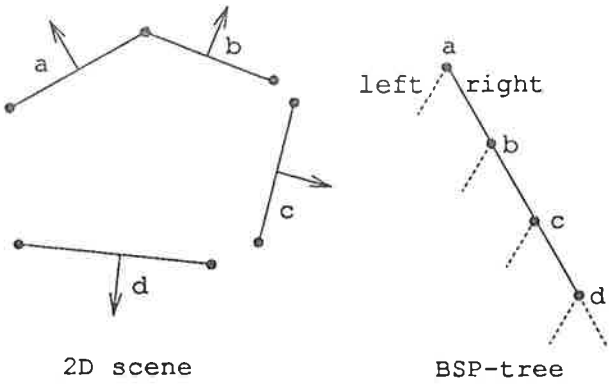


Figure 7: Unbalanced BSP-tree

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 GISs?" 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, the BSP-tree is well suited for several 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.

First, some general remarks on balancing. There are two main criteria for balancing binary trees. *Height balancing:* the height of the left sub-tree may not differ more than a fixed number h from the height of the right sub-tree. *Weight balancing:* the number of nodes in the left sub-tree may not differ more than a fixed number w from the number of nodes in the right sub-tree. As height balancing and weight balancing are strongly correlated, both will suit the needs of GISs in practice. Most theoretic proofs assume $O(\log n)$ the height of the tree. As weight balancing implies a form of height balancing, both are sufficient.

6.1 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 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 northern most region is inserted first, then the paths that correspond with the area above the nor most region will not grow when inserting the othe ions. In this manner the BSP-tree gets out of bala \ simple solution is to insert the line segments, per d level, in truly random order.

A different approach is to insert first a few auxiliary split lines, which try to divide the space in a fair manner. The

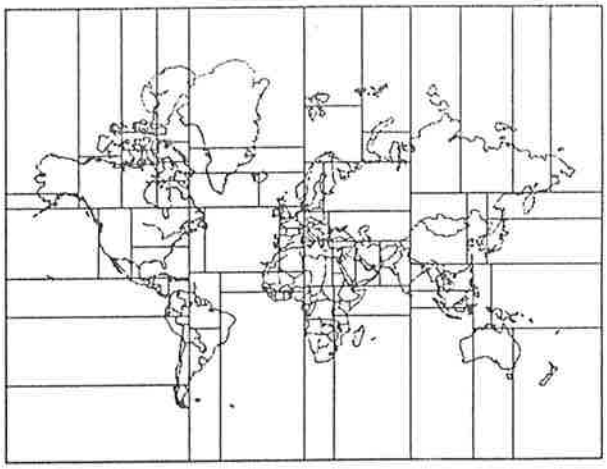


Figure 8: KD-tree with large bucket size

map data line segments are inserted after the auxiliary lines and end up in the proper region. 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, this number of splits are probably relatively small. The auxiliary split lines could be taken from a coarse raster. Note that, in principle, these lines are unbounded and that the order in which they are inserted is important. Assume that the map data space is $\{(x,y)|0 \leq x \leq 1 \wedge 0 \leq y \leq 1\}$, then 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. 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 – 1,000). KD-trees are balanced trees for storing points, Bentley [2] gives a clear description of them. Because the KD-tree is only suited to store points, it is built from the points that define the line segments. Figure 8 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 9), which does not always splits along one of the main axes. The BSP-tree is already suited to store 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:

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

With means $\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i, \mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i$, variances $\sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_x)^2, \sigma_y^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mu_y)^2$ and co-

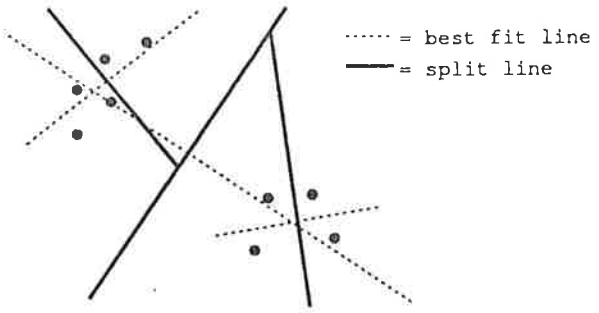


Figure 9: Generalized KD-tree

variance $\mathrm{Cov}(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y)$ defined in the usual manner[13] we get:

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

for $\sigma_x^2 > \sigma_y^2$. In 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 in the right sub-space. This process repeats itself for all sub-spaces until they contain less points than the bucket-size. This results in a perfectly balanced tree for storing points in $O(n(\log n)^2)$. The sorting causes 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 $O(n \log n)$. However, this does not necessarily result in perfectly balanced tree.

Another solution for balancing the BSP-tree is taken from Fuchs et al.[7]. 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 et al.[17] describe some heuristics for evaluating the candidates.

6.2 Dynamic Balancing

The emphasis in this subsection is on inserting line segments in a BSP-tree, while keeping it balanced. Deleting and changing line segments is less important, because they occur less frequent in GIS applications. Even without considering the balance of the BSP-tree, really deleting a line segment can be very difficult. The line segment is the root of a (sub) BSP-tree and the replacement of this root by an other line segment affects the whole sub-tree in a drastic manner. This is not a problem in case of an empty or a very small sub-tree, but otherwise it could require the complete rebuild of the sub-tree. A deletion can be simulated by marking the line segment invisible, just like 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, that is, with the function *AddLine*, see Figure 2. During the insertion the balance status of the visited nodes have 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 case of a split line segment, there are several leaves that correspond with the new line segment. So, there may 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 the sub-tree that corresponds with the deepest unbalanced node has to be reorganized. (This continues until the root is reached.) A solution might be to perform a complete rebuild 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 in case of "convex scenes" it is even impossible as explained in the previous subsection.

The root of the new sub-tree is an auxiliary line and it is made in the similar way as 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 number 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 parts as large as possible from the old to the new sub-tree. In order 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 center lies halfway the line segment and the radius is the smallest value such that all line segments of the sub-tree lie within the circle. In order to further increase the computational efficiency the square of the radius is stored 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.

7 Practical Results

In this section we present the first results of our implementation. Note that this is just a prototype GIS and not all functions are present yet. The prototype is a "main mem-

ory 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. Probably, the structure will resemble the Cell Tree described by Günther [9, 10].

Fuchs et al.[8] show that if n line segments may result 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 a new (long) line segment results in a lot of leaves of 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 enormous storage requirements: $S(n) = O(n^2)$. This is unacceptable in the case of GISs in which n is typically very large, e.g. 10,000 – 100,000.

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. In order to gain experience we are now working on a prototype GIS that is based on a BSP-tree. 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[4] 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 near Harderwijk in The Netherlands. These three maps are from completely different sources, but they produce very similar results. The table below shows some of the key figures when no measures for balancing are taken.

	Map 1	Map 2	Map 3
inserted lines (I)	13,350	108,966	5,456
degenerated lines (D)	3	50	4
split lines (S)	6,313	48,173	2,586
leaves (L)	19,661	157,090	8,039
expansion (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

The expansion factor is defined by $f = L/I$ and the theoretic minimum depth by $d_{\text{th}} = \lceil {}^2\log L \rceil$. 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 end point equal to the begin point or at least within a distance smaller than a relative accuracy *eps*, as used by our program.

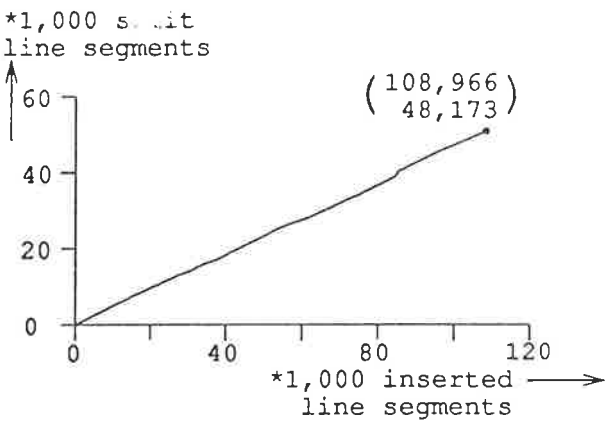


Figure 10: Split lines as function of inserted lines

Figure 10 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 change that a new line segment has to be split. However, this is not true. The straight line in Figure 10 means that the chance that a new line segment has to be split is independent of the number of already inserted line segments. This is a remarkable result, because it implies that BSP-trees of real maps have $O(n)$ storage space complexity in contrast to the worst case $O(n^2)$ with n the number of line segments in the map. 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 the 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.

8 Conclusion

The data structure presented is one of the few that combines the two difficult requirements: spatial organization and detail levels. Because of its generality it enables incorporation of other spatial organization techniques in the BSP-tree. For example: the raster structure, the quadtree or the KD-tree. A surprising result of our implementation is that BSP-trees of real maps seem to have no more than about $1.5 * n$ nodes instead of the worst case $O(n^2)$ nodes with n the number of line segments in the map. We know that the reactive BSP-tree is far from perfect, but we hope that it serves as a source of inspiration to

generate more ideas. A reactive data structure [20] need not be based on a BSP-tree, other solutions are possible. We are also working on development of a reactive data structure based on an Object-Oriented approach to GIS [18, 21].

9 Acknowledgments

Our thanks go to TNO Physics and Electronics Laboratory, 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

[1] Dana H. Ballard. Strip trees: A hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, May 1981.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[3] Nicholas R. Chrisman. Spatial indexing schemes for GIS. *International Journal of Geographical Information Systems*, 1989. Submitted for acceptance.

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

[5] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.

[6] Wm. Randolph Franklin and Harry L. Lewis. 3D graphic display of discrete spatial data by PRISM maps. *ACM Computer Graphics*, 12(3):70–75, August 1978.

[7] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. *ACM Computer Graphics*, 17(3):65–72, July 1983.

[8] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *ACM Computer Graphics*, 14(3):124–133, July 1980.

[9] Oliver Günther. *Efficient Structures for Geometric Data Management*. Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1988.

[10] Oliver Günther and Jeff Bilmes. The implementation of the cell tree: Design alternatives and performance evaluation. Technical Report TRCS88-23, University of California, Santa Barbara, October 1988.

[11] Stephen C. Guptill. Speculations on seamless, scaleless cartographic data bases. In *Auto-Carto 9*, pages 436–443, April 1989.

[12] Christopher B. Jones and Ian M. Abraham. Line generalisation in a global cartographic database. *Cartographica*, 24(3):32–45, 1987.

[13] Alexander M. Mood, Franklin A. Graybill, and Duane C. Boes. *Introduction to the Theory of Statistics*. McGraw-Hill, 1974.

[14] K. Stuart Shea and Robert B. McMaster. Cartographic generalization in a digital environment: When and how to generalize. In *Auto-Carto 9*, pages 56–67, April 1989.

[15] Wim J.M. Teunissen and Jan van den Bos. HIRASP a hierarchical interactive rastergraphics system based on pattern graphs and pattern expressions. In *Eurographics*, pages 393–404, 1988. Amsterdam, The Netherlands.

[16] Wim J.M. Teunissen and Peter J.M. van Oosterom. The creation and display of arbitrary polyhedra in HIRASP. Technical Report 88-20, University of Leiden, Department of Computer Science, July 1988.

[17] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153–162, July 1987.

[18] Jan van den Bos. PROCOL – A protocol-constrained concurrent object-oriented language. Special Issue on Concurrent Object Languages, Workshop Concurrency, OOPSLA '88, San Diego. *SigPlan Notices*, 24(4), April 1989.

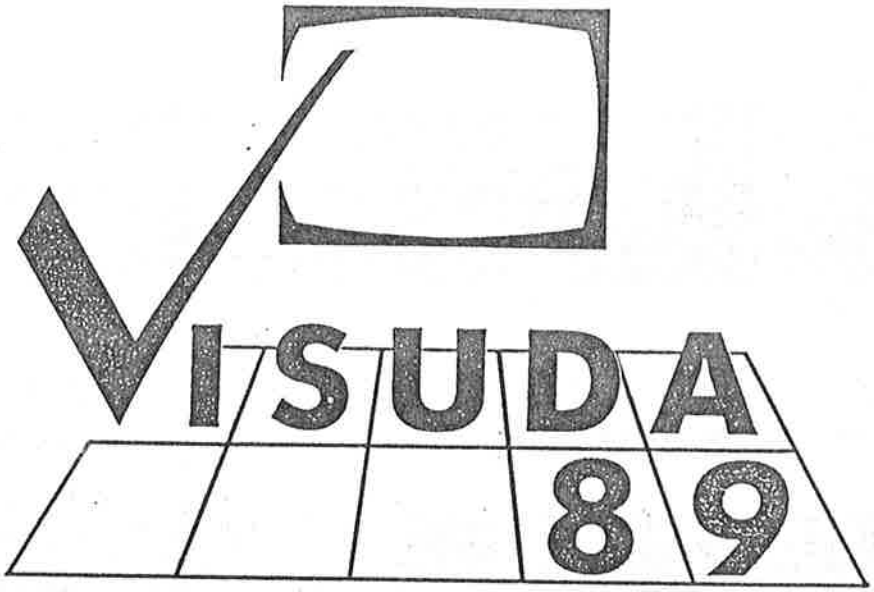
[19] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems, Orlando, Florida*, pages 104–118, September 1988.

[20] Peter van Oosterom. A reactive data structure for Geographic Information Systems. In *Auto-Carto 9*, pages 665–674, April 1989.

[21] Peter van Oosterom and Jan van den Bos. An object-oriented approach to the design of Geographic Information Systems. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, July 1989. Submitted for acceptance.

PROCEEDINGS

ACTES



Conférence internationale et exposition sur l'informatique graphique pour la défense, l'administration et les grands projets

International conference and exhibition on visual computing for defense, government and large projects

CIP Porte Maillot - Paris - France
13-16/6/89

Sous le haut patronage de /*Under the auspices of*
Ministère de la Défense
Ministère de la Recherche et de la Technologie

Avec le support de/*Sponsored by*
Association Nationale de la Recherche Technique (ANRT)
World Computer Graphics Association (WCGA)

et le soutien de/*cosponsored by*
Association Science et Défense
A.F. MICADO
Deutsche Gesellschaft für Wehrtechnik (DWT)

Organisé par/*Organized by*
E.S. International Communucations - 16 avenue Bugeaud - 75116 Paris - France
Tel : (33) 1.45.53.26.67 - Telex : 642632 ANRTF - Fax : (33) 1.47.04.25.20.