A Reactive Data Structure for Geographic Information Systems

Peter van Oosterom
TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG The Hague, The Netherlands *and*
Department of Computer Science, University of Leiden,
P.O. Box 9512, 2300 RA Leiden, The Netherlands
(Email: OOSTEROM@HLERUL5.BITNET).

January 13, 1989

Abstract

We introduce 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 partioning tree that includes detail levels. This tree is one of the few spatial data structures that do not organize the space in a rectangular manner. An application of the reactive data structure in thematic mapping is given.

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 has to be based on an appropriate data structure. However, most existing systems lack these data structures. We introduce the term *Reactive Data Structure* for 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 [10]. 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

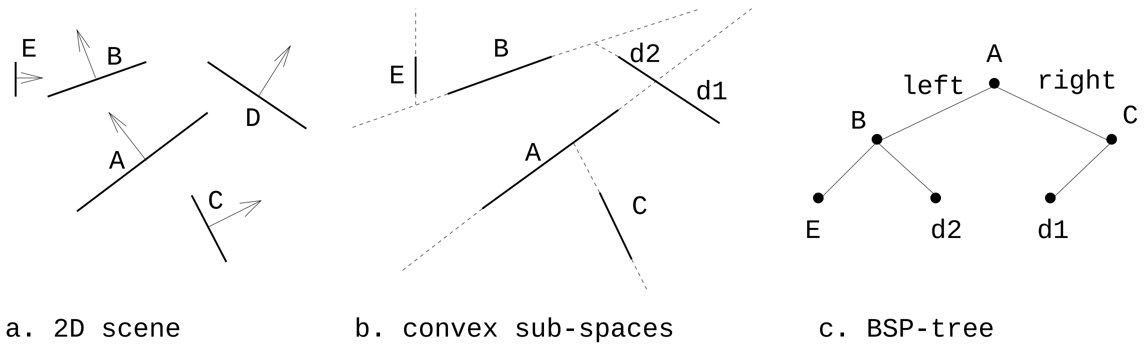


Figure 1: The building of a BSP-tree

of the map, the objects are enlarged, and more details are drawn (new objects). Conversely, when zooming out, fine details are removed from the display. We call this operation *logical zoom* in contrast with the ordinary zoom which only enlarges. There is some literature available on data structures with detail levels, for instance strip trees [1] and multi-scale line-trees [6].

The data structure presented in this paper is a modification of the 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. In section 5, an application of the reactive data structure is given. Finally, the pros and cons are discussed in section 6.

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 [5, 4]. 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, the 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 is not yet used for splitting), is split into two parts. 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 of which line segment is used to split the space, very much influences the building of the tree. We want the BSP-tree to be balanced and have as few nodes as possible. These two wishes are conflicting, 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.

The appendix contains the Pascal code of a program that builds a BSP-tree. The program *BuildTree* is a variation of the traditional method for building a BSP-tree [5]. The procedure *SplitLine* and the functions *LineSituation, CreateNode* and *GetLine* are not included in the appendix, 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 possible that the line segment has to be split first.

The splitting of line segments has a serious drawback. If we have m line segments in a scene, then it is possible that we end up with $O(m^2)$ 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.

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. A polygon is defined by a set of line segments, which together make up the boundary of the polygon. The boundary of an object can be stored in a BSP-tree. The BSP-tree is extended with 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 tells 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 [7]. Because of the spatial organization, the hidden surfaces can be "removed" in $O(n)$ time with n the number of polygons in the tree [8]. The object BSP-tree is also well suited to perform the set operations: union, difference and intersection, as used in Constructive Solid Geometry (CSG) systems [9]. The *map overlay operation* in a GIS (described in [10]) can be compared with these set operations.

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

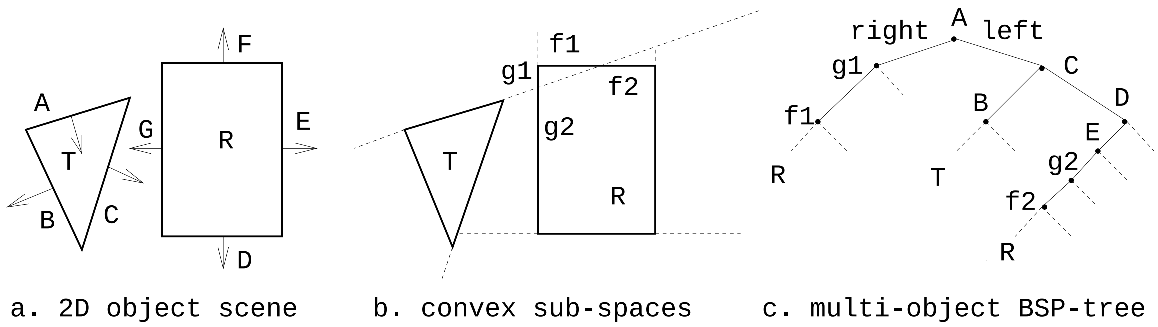


Figure 2: The building of a multi-object BSP-tree

inherent to the problem to organize the space, we expect a good spatial organization. Maps always contain multiple objects; for example several countries on the map of Europe. Because we deal with multiple objects we have to modify the concept of the 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. This type of BSP-tree is called *multi-object BSP-tree*.

Figure 2a shows a 2D scene with two objects, triangle T with sides ABC and rectangle R with sides DEFG. The space is organized as a set of convex sub-spaces. The result is shown in Figure 2b. The BSP-tree of Figure 2c 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 2c. A disadvantage of this BSP-tree is that the representation of one object is scattered over several leaves; for example rectangle R in Figure 2. The following list summarizes the properties of the multi-object BSP-tree:

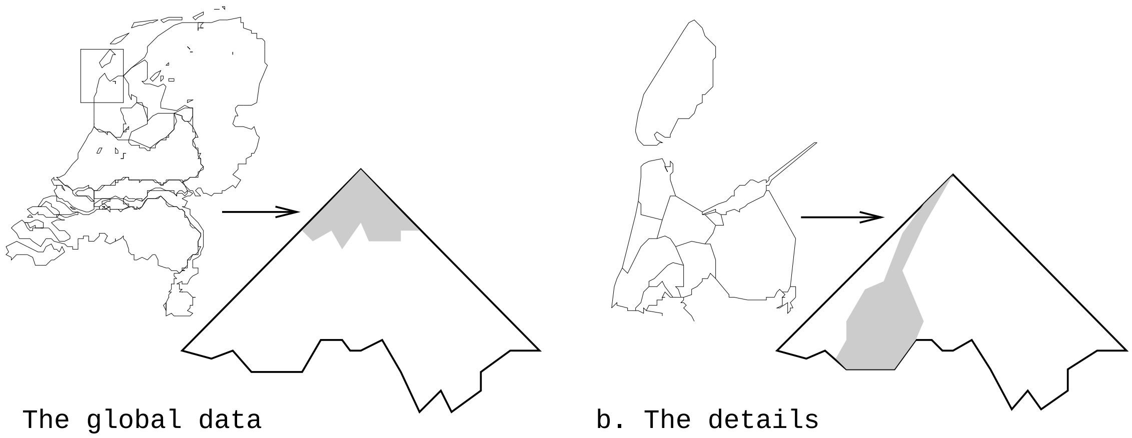
- Each node in the tree corresponds with a convex sub-space.
- Each internal node splits the convex sub-space into two convex parts: left and right. The convex sub-spaces become smaller when the tree is descended. Each internal node contains one line segment.
- Each leaf node corresponds with a convex 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

A map is displayed on the screen. The user selects 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 have to locate point P in the tree. This is done by descending



a. The global data b. The details

Figure 3: Global and detailed data in the reactive BSP-tree

the tree until a leaf node is reached. This leaf node contains the identification of the object. The descending of 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 results in one straight path from the root to a leaf node. If the tree is balanced and n is the number of internal nodes in the tree, then this search can be performed in $O(log\ n)$.

3.2 The rectangle search

The user wants to select all objects within rectangle R. This operation is also performed when (a part of) a map has to be displayed 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. This traversal can be accomplished by a simple recursive function. Only the nodes encountered during this traversal contain objects that are within the rectangle.

The efficiency of these operations is based on the fact that whole 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 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. A line segment that is inserted early, will end 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

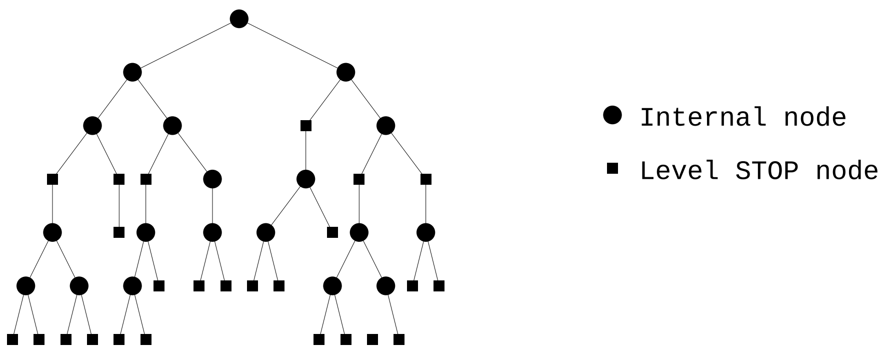


Figure 4: The reactive BSP-tree

be split a few times), before it reaches the correct position and is stored in a new node on a *lower* level of the BSP-tree. We use this property to create a *reactive BSP-tree*. If the global data is inserted first in the BSP-tree, then it will end up in the higher levels of the BSP-tree. The local data (details) are added later, so they end up in the lower levels of the BSP-tree. Figure 3 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 whole 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 highest level, and so on. When we display this map, the number of detail levels depends on the 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 must 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 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 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 4 shows a reactive BSP-tree with two detail levels.

A drawback of the reactive BSP-tree is that it only supports a part of the map generalization process. Unimportant lines are removed and small regions are grouped,

a. Choropleth b. Prism map

Figure 5: Two map types for thematic mapping

but important lines will be drawn 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 every detail level. However, the storage of the same line at multiple levels introduces undesired 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 [2].

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 the administrative units. We show how a choropleth map and a prism map can be produced. An example of those map types is given in Figure 5. The reactive BSP-tree with the level STOP nodes is used. 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 detail level is determined. On a choropleth map the administrative units are colored. The color depends on the value of the census variable in the administrative unit. 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.

We do not have 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 in the BSP-tree the temporary data structure is updated. We will show that this takes *O(log n)*. If the BSP-tree is balanced, the height of the tree is *O(log n)* with n the number of line segments (nodes) in the BSP-tree. So, a convex sub-space has no more than *O(log n)* edges. The insertion of an edge in such a convex sub-space (polygon) takes *O(log n)*. Displaying the whole BSP-tree while coloring the convex sub-spaces takes *O(n log n)* time, because there are n nodes in the BSP-tree. 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 [3] is an attractive map to look at and it offers the possibility to display an extra variable by the height of the prisms. Before the prism map is generated the user has to indicate from which direction he wants to look at the prisms. A prism map can be compared with a set of 3D-objects.

Basically the prism map is produced 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, just 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 these if we label those sides of the convex sub-spaces, that are part of line segments. This can be done without overhead.

The "hidden surface" problem is usually quite difficult and time consuming to solve. However, if we slightly change the manner in which the BSP-tree is traversed, the hidden surface problem is solved (in combination with the Painters-algorithm). The different traversal does not cost any extra processing time and ensures that prisms further away from the viewing point are drawn first. This results the "removal" of the hidden surfaces in the prism map, for more details on this topic see [8]. Normally, the whole 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 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. Because of this fast response, the end-user will be stimulated to take other views of the data.

6 Conclusion

The data structure we presented is one of the few that combines the two difficult requirements: spatial organization and detail levels. The reactive BSP-tree fulfills, up to a certain extend, those requirements. The BSP-tree also introduces some problems, as we have seen in the previous sections. And we can think of more problems: How should a single unconnected point be stored in the BSP-tree? How can the BSP-tree be balanced? How will the BSP-tree behave if we insert very large amounts of irregular geometric data? The BSP-tree is a static data structure. This is not really a problem, because the maps in most GIS applications are also static. In order to gain experience we are currently working on a prototype GIS that is based on a BSP-tree. We are interested in the size and the performance of the BSP-tree. We know that the BSP-tree is far from perfect, but we hope that is serves as a source of inspiration to generate more ideas. A reactive data structure 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.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems*, 1988. Orlando, Florida (In press).

A Appendix: BuildTree

```
program BuildTree;

type  LineSegment = record x1, y1, x2, y2: real end;
      Pos = (LEFT, RIGHT, SPLIT);
      BSP = ^node;
      node = record
        segm: LineSegment;
        l, r: BSP {Left end Right child in BSP−tree}
      end;

var   root: BSP;
      newsegm: LineSegment;

procedure SplitLine
(tree: BSP; segm: LineSegment; var Lsegm, Rsegm: LineSegment); forward;
function CreateNode(tree: BSP; segm: LineSegment): BSP; forward;
function LinePosition(tree: BSP; segm: LineSegment): Pos; forward;
function GetLine(var newsegm: LineSegment): boolean; forward;

function AddLine(tree: BSP; segm: LineSegment): BSP;
var   Lsegm, Rsegm: LineSegment;
begin
   if tree = nil then {Position found in BSP−tree, create node}
      tree := CreateNode(tree, segm)
   else begin {Position not yet found, go further down the tree}
      case LinePosition(tree, segm) of
         LEFT: tree^.l := AddLine(tree^.l, segm);
         RIGHT: tree^.r := AddLine(tree^.r, segm);
         SPLIT: begin
            SplitLine(tree, segm, Lsegm, Rsegm);
            tree^.l := AddLine(tree^.l, Lsegm);
            tree^.r := AddLine(tree^.r, Rsegm);
         end
      end
   end;
   AddLine := tree;
end;

 .
 .
begin
   root := nil;
   while GetLine(newsegm) do root := AddLine(root, newsegm)
end.
```