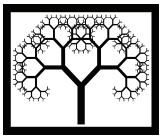
Paper 150



Proceedings of the Sixth International Conference
on Engineering Computational Technology,
M. Papadrakakis and B.H.V. Topping, (Editors),
Civil-Comp Press, Stirlingshire, Scotland

Application of Binary Spatial Partitioning to Computer Aided Design

A.A. Stamos
School of Civil Engineering
National Technical University of Athens, Greece

Abstract

Computer Aided Design (CAD) applications usually need to store and process dynamically and efficiently various drawing elements. Typically the quad-tree data structure is used in 2 dimensions and the octant-tree data structure in 3 dimensions. Here the Binary Spatial Partitioning (BSP) tree data structure is proposed for CAD applications. BSP scales better in irregular geometries such as long narrow engineering drawings, it has less overhead, it is easier to implement, it and it is easily extended to 3 and higher dimensions. Modifications to BSP are proposed to adapt it better to interactive environments. Elements are not broken when a BSP node is split keeping BSP size small. Physical elements size limit the depth and the fragmentation of the BSP trees. Semi-infinite BSP nodes delay the BSP structure finalization, improving BSP balance. The knowledge of drawing extents ensures BSP balance when the BSP structure is rebuilt after a drawing regeneration. Finally implementation related issues are addressed.

Keywords: binary spatial partitioning, computer aided design.

1 Introduction

The need to search for items such as numbers and text arises in many applications, and often it is crucial to do the search fast and efficiently. There are many algorithms for searching, each one with advantages and disadvantages, as it can been seen in the rich literature ([1], [2]). The need to search for items also arises in the CAD (Computer Aided Design) applications, the items being geometric elements. The geometric elements may be non-dimensional such as a single point, 1-dimensional such as a line segment, 2-dimensional such as a polygon, or in the case of 3d Cad, 3dimensional such as a prism. The search of geometric elements is certainly more difficult than, and different to, the search for numbers or text. Of the many algorithms that can be found in the literature (for example in [3]) this paper

focuses to Binary Space Partitioning (BSP – [4]) tree, a structure which is able to store the elements as well as search for them. The classical structure for this is the quad-tree in 2d ([5]) and the oct-tree in 3d ([4]). The BSP scales better in irregular geometries such as long narrow drawings which, for example, often arise in highway design.

2 Theoretical Background

2.1 Quadtrees

The classical structure for element storing is the quad-tree and octrees in 3 dimensions. The concept of quad-tree is that when the number of elements in a drawing pass a threshold value, the area of the drawing is divided to 4 rectangles and each element is stored into the rectangle it belongs to. This way the number of the elements is always manageable, and the search operations, such as element intersection, need not examine all the elements, but only the elements that lie within certain rectangles, with obvious speed enhancement. If the number of the elements of a rectangle gets big, this rectangle is divided to 4 sub-rectangles and so on. If a certain element spans two, three or four sub- rectangles, the common wisdom is to split the element into two, three, or four segments, and put each segment into the rectangle it belongs to ([6]).

The same procedure may be extended in 3 dimensions with octrees. The volume of the "drawing" is divided to 8 parallelepipeds and each one of them stores the 3d elements that lie within it.

2.2 Binary Spatial Partitioning trees (BSP)

The concept of Binary Spatial Partitioning trees (BSP), typically used in hidden surface removal ([7]), is similar to the quad-tree with the difference that the area of the drawing is divided to 2 rectangles. The area is split with a line which may be horizontal or vertical, whichever splits the area more evenly (with respect to either the shape or the number of elements). Each rectangle may be further divided to two sub-rectangles with a split line that may or may not be parallel to the previous split line.

The BSP can be easily extended easily to 3 and higher dimensions. The n dimensional hyperspace is divided to two hyper spaces with a (n-1) dimensional hyperplane normal to any of the axes of the hyperspace. Each hyperplane division is described fully by the axis normal to the hyperplane (a single integer number) and the coordinate x_i of the point of intersection of the hyperplane and the axis (a single real number).

In this work the hyperplanes are assumed to be parallel to the coordinate axes, and they split the the hyperspace into two equal sub-hyper spaces. This is not only simpler, but it makes the computation faster for dynamic BSP, such as those found on CAD systems.

2.3 CAD systems requirements

The CAD has different search requirements than the general problems addressed by the computational geometry. The key observation is that the user will almost always search for something local to their current environment, such as the mouse cursor or the visible part of the drawing. For example a user does not require all the intersections of all the elements in a drawing; they require the closest element intersection near the mouse cursor. Thus the goal is to minimize the number of elements to be examined for intersections, and then pass only these elements to the appropriate algorithm.
Typical CAD search requirements are listed below:

- Search for an element near a user defined point (select one).
- Search for elements which lie within a user defined rectangle (select window).
- Select elements which lie, at least partly, in a rectangle (select crossing window).
- Find the closest point of any element near a user defined point (select nearest).
- Search for intersections, endpoints, middle points, etc., near a user defined point (usually the mouse cursor).
- Search for elements that belong to certain layers, or other containers.
- Search for elements with certain text.
- Search for elements processing a certain attribute (for example circles of a given color).

The last three searches are not spatial and they are not covered here. It is instructive to see that all the spatial searches above are really rectangular in nature, or range searching in other words. Indeed if "near" is quantized, then the search for an intersection "near" a point, is transformed to the search within certain distance to the left, right, above or below the point. This is, of course, the definition of the interior of a rectangle.

The objective of this work is to minimize, or at least lower, the number of elements that must be checked for intersection or any other operation, without losing the simplicity of the BSP algorithm.

3 Application of BSP to CAD

3.1 Addition to BSP

The addition of an element to the BSP structure is described in [6]. Starting from the root of the BSP tree, it is checked if the element lies to the "left" sub-rectangle or sub-hyperspace (the one with the lower coordinates with respect to the axis normal to the split line or split hyperplane) or to the "right" sub-rectangle or sub-hyperspace (the one with the higher coordinates). If the sub-rectangle is leaf then the element is added there. Otherwise the same procedure continues recursively.
However, in this work, if an element spans the two sub-rectangles, then it is not split to two segments (one within each sub-rectangle), as it is commonly proposed in the literature. The element split is not used for many reasons:

- Many operations when applied to the whole element, such as the computation of the nearest point of a line segment to a user defined point, have the same computation cost with the same operation on either of the segments. Divide and conquer delivers no gain here.
- Worse, if the element is a circle, then the segments will be arcs. The operations on arcs are generally more difficult than on circles and thus the computation cost is bigger than the operation on a whole circle.
- The number of elements may grow significantly, with obvious memory and speed penalties.
- Some operations on the element may require action to all the element segments, such as element deletion, or color change. This means that a single cheap operation may be transformed to an expensive multitude of operations, which is exactly the opposite of the purpose of the BSP. Also, every segment of the element must be efficiently found in order to perform the operation. It can be done by keeping track of all the segments of each element using a container, and by storing a reference to the container along with each segment. Clearly, such complexity is to be avoided if possible.

In this work, it is proposed to store the element directly to the node that completely contains it, whether it is a leaf node or not. This simplifies the addition procedure and avoids all the above problems. It also provides a physical limit to the depth of the BSP tree, as the rectangles of leaf nodes may not be smaller than the smallest element (actually they will be bigger than this). As a result the data structure that is used in this paper is a k-d tree, ([4]) rather than a BSP.

3.2 Node split

If the number of elements stored in a node is bigger than a maximum threshold, then the node is split. Theoretically, the split line (or split hyperplane) may not halve the rectangle (or hyperspace) of the node, and indeed it may not be normal to one of the axes. The latter makes the shape of the children arbitrary, which greatly complicates the computation of element inclusion. The former raises the issue of the computation of the best position of the split hyperplane which may be expensive or may not even be feasible ([6]). None of them are applicable to the dynamic environment of a CAD system, since the arbitrary additions and deletions of elements invalidate the benefits they have.

The split line is normal to one of the axis (split axis) and halves the rectangle. Not all the elements will be transfered to the child nodes. If an element crosses the split line, then it is kept in the parent node. Thus the optimum split axis, is the axis which transfers the maximum number of elements to the children nodes.

Care must be exercised to avoid unbalanced BSP trees. If none or few elements are transfered to the children nodes, for example if less than 20% of the elements are transferred, then the split offers no advantage; it is just an expensive emulation of the linear search ([2]). The same might seem to be true, if all or most of the elements are transferred to one of the children nodes. However it is entirely possible that the same elements are then split evenly to 2 grandchildren (figure 1a), and if not, then they may be split evenly to 2 great-grandchildren and so on (figure 1b). Even if the

elements are concentrated in a very small area (hyperspace), the nodes will converge very rapidly (exponentially) to it (figure 1c).

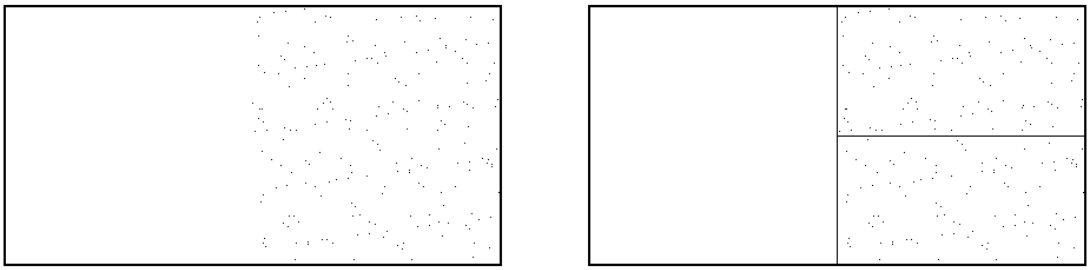


Figure 1a: Uneven children, even grandchildren.

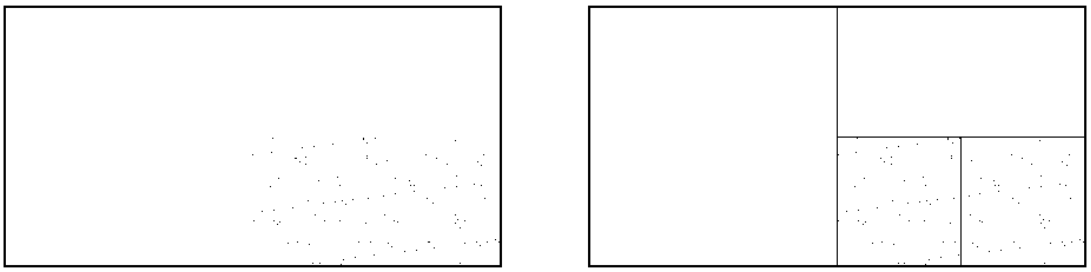


Figure 1b: Uneven children/grandchildren, even great grandchildren.

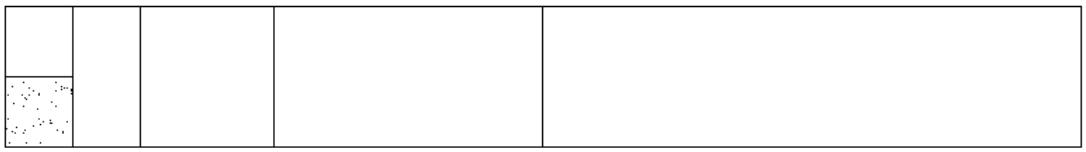


Figure 1c: Exponential convergence to concentrated elements.

If the node is not split, then a new element addition will cause the whole process of node split to be repeated. But it is almost certain that the split will be unsuccessful again, for the same reasons. In order to avoid such thrashing, if a node split is unsuccessful, the maximum threshold value of the node is increased by 50%.

The algorithm for node split is summarized below:

Split node n:
Let n.nelmax be the, already initialized, maximum threshold value of node n
Let n.nel=number of elements in node
If n.nel < n.nelmax: exit
Let m=number of dimensions
For i=1..m:
    Create split line perpendicular to axis i
    Let nc=number of elements that cross the split line
    Let ns[i] = n.nel-nc

End for
Let j be the axis with the max{ns}
*If ns[j] < 80% * n.nelmax:*
    *n.nelmax = n.nelmax * 1.5*
else:
    Split the node to children nodes n1 and n2 with split axis j
    Transfer elements to n1 and n2
    Split node n1
    Split node n2
End if
End split node.

3.3 Node fusion

In the dynamic environment of a CAD system, element deletions may empty, or nearly empty, one or more nodes. In order to avoid expensive emulation of a linear search, if the number of elements of a child node and its sibling is less than a minimum threshold, the elements of the children are added to the parent, and the children nodes are deleted – the nodes and their parent fuse. To avoid immediate re-split of the parent, the number of elements of the children and the parent should be less than the parent's maximum threshold. It is also necessary that the node has a parent, which means that it is not the root node. In order to avoid checking yet another condition, the minimum threshold of the root node could be set to a negative number.

If the fusion is unsuccessful, then another element deletion will cause the whole process of node fusion to be repeated. But it is almost certain that the fusion will be unsuccessful again, for the same reasons. In order to avoid such thrashing, if the node fusion is unsuccesful, the minimum threshold value of the node is decreased by 50%. Of course, the minimum threshold can not be less than 1.

The algorithm for node split is summarized below:

Fuse node n:
Let n.nelmin be the, already initialized, minimum threshold value of node n
Let n.nel=number of elements in node
If n.nel > n.nelmin: exit
If n is the root node: exit (this is not necessary if root.nelmin is set to -1 *)*
Let n2 be the sibling of node n
If n2.nel > n2.nelmin: exit
Let p be the parent of node n
*If n.nel+n2.nel+p.nel > 80% * p.nelmax and n.nelmin > 3:*
    n.nelmin = n.nelmin / 1.5
else:
    Add the elements of n to p
    Add the elements of n2 to p
    Delete n and n2
    Fuse node p

End if
End fuse node.

3.4 Element inclusion in window

If an element is represented by its minimum spanning rectangle (MSR), or points A and B in figure 2a, then the element is not inside a window with points A_w and B_w, if one the following conditions holds:

$x_B < x_{Aw}$ or $x_A > x_{Bw}$ or

$y_B < y_{Aw}$ or $y_A > y_{Bw}$

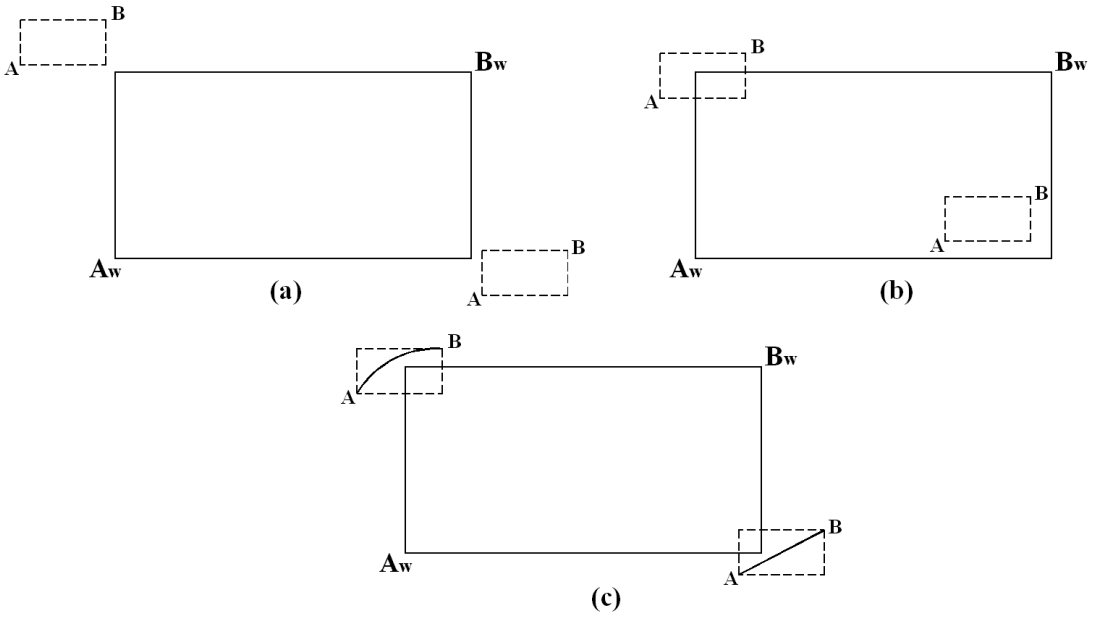


Figure 2: MSR is outside window (a), MSR crosses or is inside window (b), MSR crosses window but the element is outside (c).

The conditions hold even if any of the coordinates of the window approach infinity. The conditions can also be easily extended to higher dimensions, adding similar inequalities for each higher dimension. If none of the condition holds, then the rectangle either crosses, or is completely inside the window (figure 2b). The same is not necessarily true for the element, as for example a line segment, which may be outside the window even if none of the above conditions holds (figure 2c). To actually determine if an element crosses or is completely inside the window, more computations may be needed, which depend on the nature of the element ([8]). However, these computations are performed only if none of the above conditions are met.

3.5 Infinite and semi-infinite nodes

When a new drawing is instantiated it contains only the root BSP node. The boundaries of the root node are undefined because there are no elements. If an element is added, its minimum spanning rectangle (MSR) of the element could define the boundaries of the node. But this would be unfortunate because the node is not full and the next elements could be disjoint to the MSR of the first element, and thus not in the node.

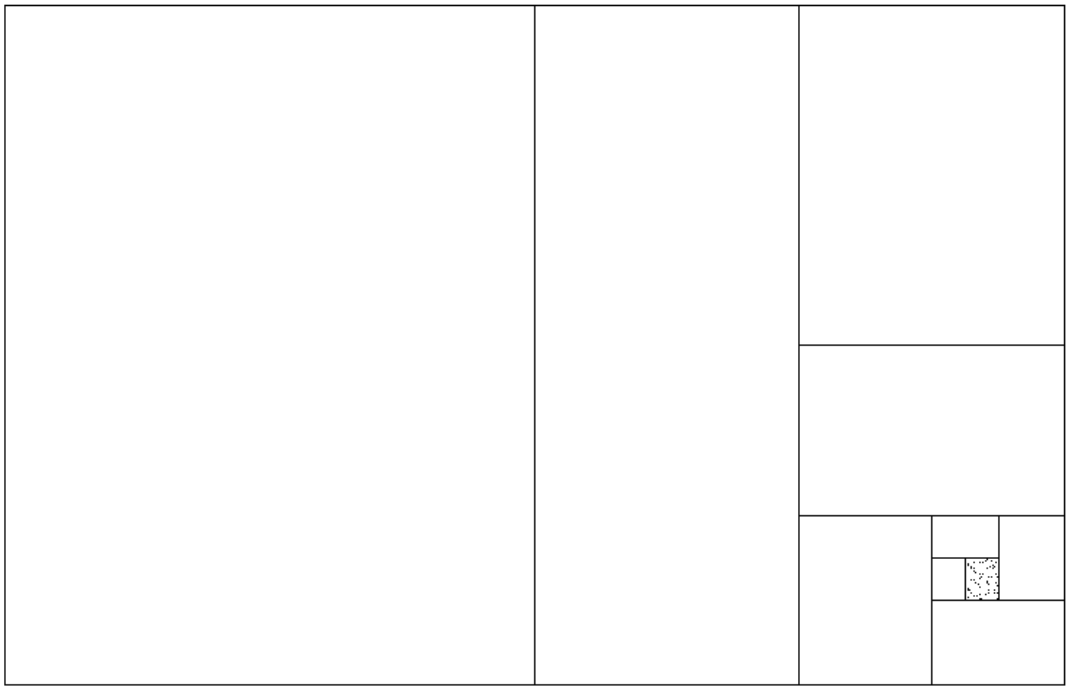


Figure 3: Element-less nodes when the root node boundaries approach infinity.

A better strategy is to define the boundaries of the node as infinite, or more conveniently to some very large positive and negative number. However, when the node becomes full, the location of the split line will be grossly out of focus, and will lead to many nested element-less nodes (figure 3). To overcome this inefficiency, the global MSR, which is the MSR of all the elements so far, is separately kept. When the boundaries of an infinite nodes are needed, then the boundaries of the global MSR are used instead.
When an infinite node is split, its children will be semi-infinite. All their edges (hyper-edges) will be infinite except one. If a semi-infinite is split, a new finite edge will be introduced to its children, and so on until finite nodes are created. In a semi-infinite node the global MSR will be used only for its infinite edges (figure 4).

The global MSR can be cheaply updated when a new element is added. When an element is deleted the MSR may or may not be affected. To actually find the MSR when an element is deleted is computationally expensive and it is avoided in this

paper. Thus the MSR will be inaccurate but usually not crossly inaccurate. The MSR is computed again when a drawing is read from disk, or when a regeneration of the drawing is triggered by the user.

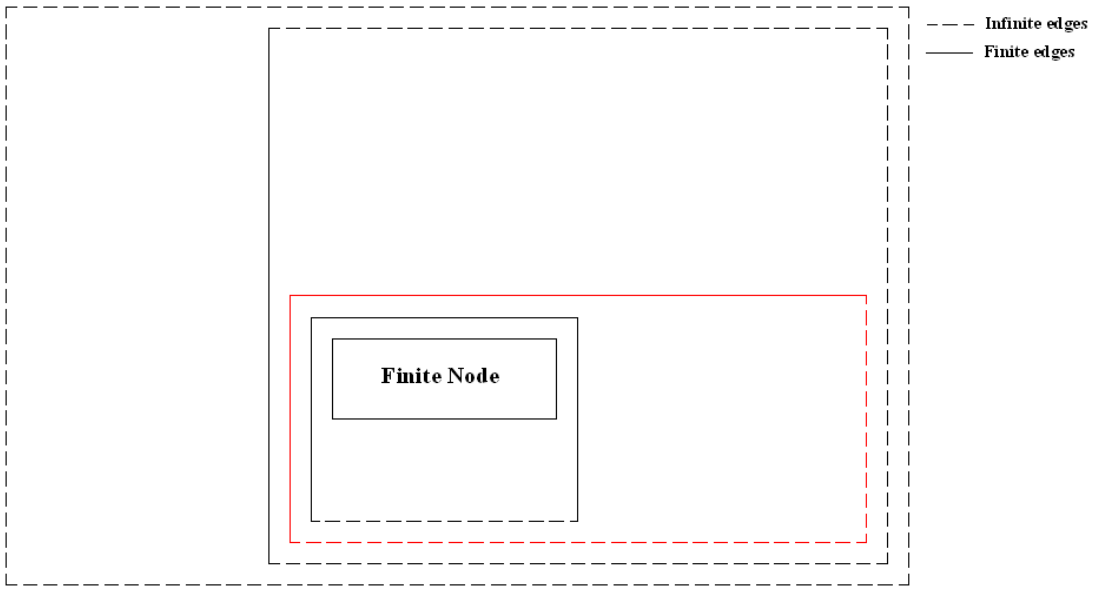


Figure 4: BSP nodes with infinite boundaries (dashed) mapped to global MSR.

3.6 Range searching

When a search for all the elements within a rectangle is needed (range searching), this rectangle will be inside the root node, which is practically infinite. The elements of the root node are checked to see if they are inside the Search Rectangle (SR). Then, if the SR is completely inside a child node of root, the process is repeated recursively for this child node. Otherwise, if the SR crosses the split line of root, it is split to 2 SRs by the split line. The process is then repeated recursively for both children nodes with the appropriate SR. The process stops when no children nodes are left.

Note that all elements of a node are checked for inclusion in the appropriate SR, which may be relatively expensive, depending on the type of the element. But often in the CAD environment, the SR is not very strict, and the pending operation on the elements which are sought, is cheaper than the inclusion computation. For example when the nearest midpoint of the nearest element is sought, it is equally correct but cheaper to check all the elements of a node for the nearest midpoint, than finding which elements of the node are inside the SR and then finding the nearest midpoint of them. In fact, even if the inclusion computation were slightly cheaper than the pending operation, the advantage would be offset by the fact that, in many elements, both the inclusion computation and the pending operation would be done.

Thus the range searching algorithm has a restriction mode, which returns either the elements inside the window or all the elements of the nodes it visits. The algorithm is summarized below:

Range search (SR, restrict):
Let S is an empty set
Search_Node(root, S, SR, restrict)
End range search.

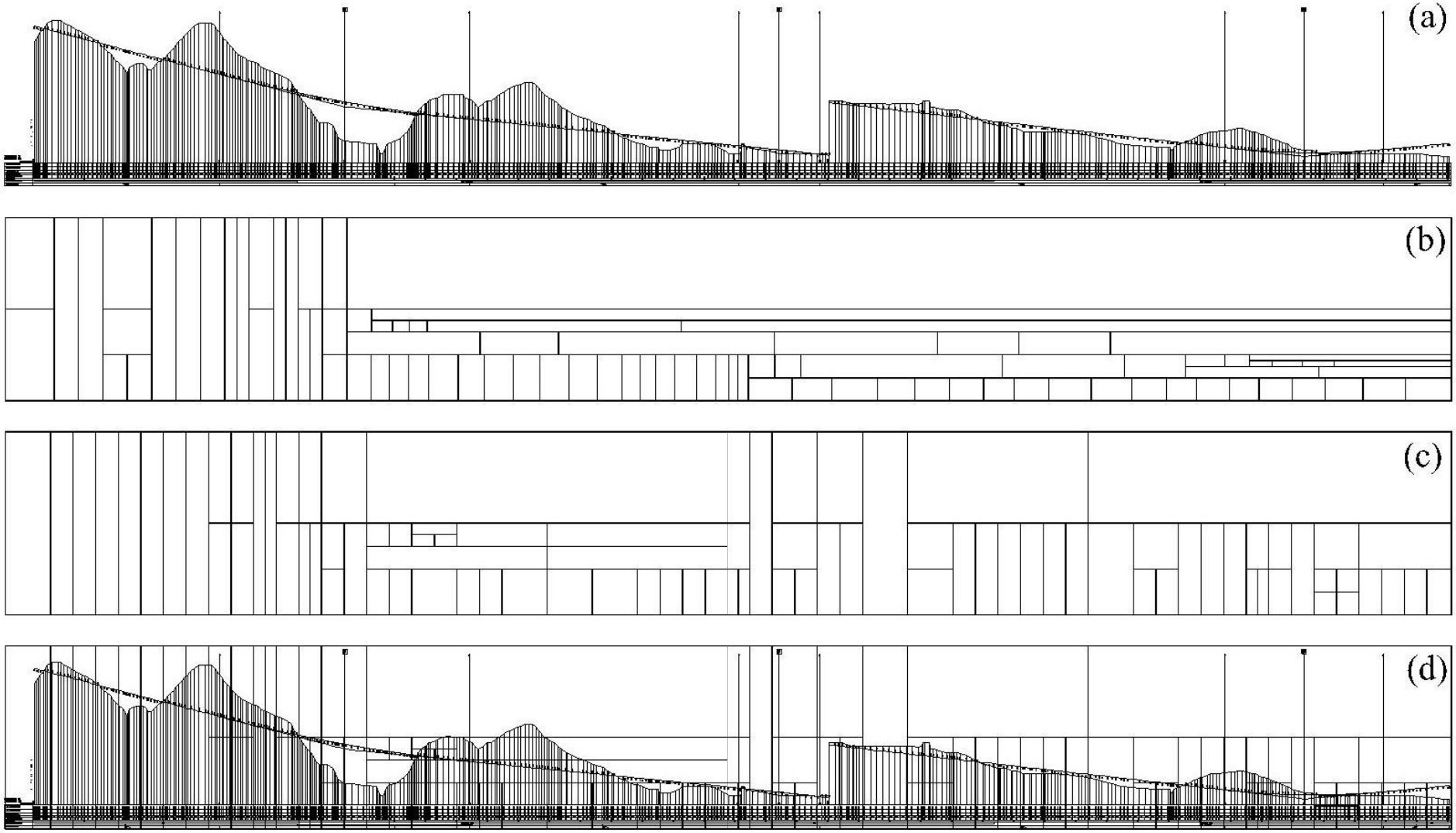
Search_Node(n, S, SR, restrict):
For each element e of n:
    If restrict == False or e in SR: add e to S
End for
If n has no children: exit
Let n1, n2 be the children nodes of n
If SR in n1:
    Search_Node(n1, S, SR, restrict):
Else if SR in n2:
    Search_Node(n2, S, SR, restrict):
Else:
    Split SR to SR1 and SR2
    Search_Node(n1, S, SR1, restrict):
    Search_Node(n2, S, SR2, restrict):
End if
End Search_Node.

3.7 Implementations issues

In the description of algorithms various parameters were used, like the minimum and maximum threshold. The best values of these parameters can be determined empirically, and they depend on the typical drawings and typical use of the CAD system. They also depend on the programming language used. Good parameter values for compiled languages, are different to good parameter values for interpreted languages. This is because, in interpreted languages, built-in iteration of built-in containers is more efficient than user-defined iteration of user-defined container ([9]). As a result, the number of elements in a BSP node is much bigger in interpreted languages. This implies that, in the case where the total number of elements is not very big, only the root node of BSP will contain elements, rendering the algorithm to an imitation of the exhaustive search.

4 Application and Conclusions

In order to test the presented BSP algorithm, the algorithm was implemented in an open source CAD ([10]), and a long narrow drawing was taken from real life highway design. The drawing presents a section along the axis of the Egnatia highway. Its width is 470cm and its height 58cm (figure 5a). The drawing has some elements (lines) which almost span the height of the drawing and other lines which span its width. There are also isolated long lines in nonuniform positions which destroy the regularity of the drawing. The total number of elements of the drawing is 5776.


(a)
(b)
(c)
(d)

The elements of the drawing were entered to an empty BSP tree, as they were produced by a civil engineering application. In total, 179 BSP nodes were created, the distribution of which are shown in figure 5b. Then, a search for the elements contained in a window was made. The window length represented 7% of the width of the drawing. The search returned 835/5776 elements or about 14% of all the elements, and visited 30/179 nodes or about 17% of all the nodes.

However, if the dimensions of the drawing are known in advance, for example if the drawing is read from a dxf file, the BSP can exploit this information to initialize the global MSR. The result is 175 nodes, with improved node distribution as shown in figures 5c and 5d. The same search returned 607/5776 elements or about 10% of all the elements, and visited 20/175 nodes or about 11% of all the nodes.

The results are certainly promising, but more experimental work needs to be done in order to fine tune the various parameters. The parameters may also be fine tuned for various types of drawing, such as the typical drawings in civil engineering. Finally, the optimum parameter values can also be computed by techniques like simulated annealing ([11]), although the computation time will probably be very long.

References

[1] D.E. Knuth, “The Art of Computer Programming, Volume 3, Sorting and Searching”, Addison-Wesley, Reading, MA, 1975.

[2] R. Sedgewick, “Algorithms in C++”, Addison-Wesley, ISBN 0-201-51059-6, 1992.

[3] F.P. Preparata and M.I. Shamos, “Computational Geometry – An Introduction”, Springer-Verlag, ISBN 0-387-96131-3 Springer-Verlag New York Berlin Heidelberg Tokyo, ISBN 0-387-96131-3 Springer-Verlag Berlin Heidelberg New York Tokyo, 1985.

[4] M. Berg, M. Kreveld, M. Overmars, and O. “Computational Geometry”, 2nd revised edition, Springer-Verlag. ISBN 3-540-65620-0, 2000.

[5] R. Finkel and J.L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", Acta Informatica 4 (1), 1-9, 1974.

[6] B. Wade, “The BSP Frequently Asked Questions”, http://www.opengl.org/resources/code/samples/bspfaq/, Last Update: 09/20/2001.

[7] M. Paterson and F. Yao, “Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modelling”, Discreet Computational Geometry, 5(5), 485-503, 1990.

[8] A. Boehm and T. Theoharis, “Graphics, Principles and Algorithms”, University of Athens, Symmetry Publications, ISBN 960-11-0004-0, 1999.

[9] A. Marteli and D. Ascher, “Python Cookbook”, O'Reilly, ISBN 0-596-00167-3, eds. 2002.

[10] A.A. Stamos, “ThanCad, a 2dimensional CAD”, EuroPython, Vilnious, 2007.

[11] Press H. Wiliam, Teukolski A. Saul, Vetterling T. Wiliam, Flannery P. Brian, “Numerical Recipes in Fortran”, Cambridge University Press, ISBN 052143064X, 1992.