Boolean Operations on Surfel-Bounded Objects using Constrained BSP-Trees

Marcus A. C. Farias [1] Carlos E. Scheidegger [2] João L. D. Comba [1] Luiz C. Velho [3]

[1] Instituto de Informática, UFRGS
[2] Scientific Computing and Imaging Institute, University of Utah
[3] Instituto de Matemática Pura e Aplicada

Abstract

Point-based modeling and rendering is an active area of research in Computer Graphics. The concept of points with attributes (e.g. normals) is usually referred to as surfels, and many algorithms have been devised to their efficient manipulation and rendering. Key to the efficiency of many methods is the use of partitioning schemes, and usually axis-aligned structures such as octrees and KD-trees are preferred, instead of more general BSP-trees. In this work we introduce a data structure called Constrained BSP-tree (CBSP-tree) that can be seen as an intermediate structure between KD-trees and BSP-trees. The CBSP-tree is characterized by allowing arbitrary cuts as long as the complexity of its cells remains bounded, allowing better approximation of curved regions. We discuss algorithms to build CBSP-trees using the flexibility that the structure offers, and present a modified algorithm for boolean operations that uses a new inside-outside object classification. Results show that CBSP-trees generate fewer cells than axis-aligned structures.

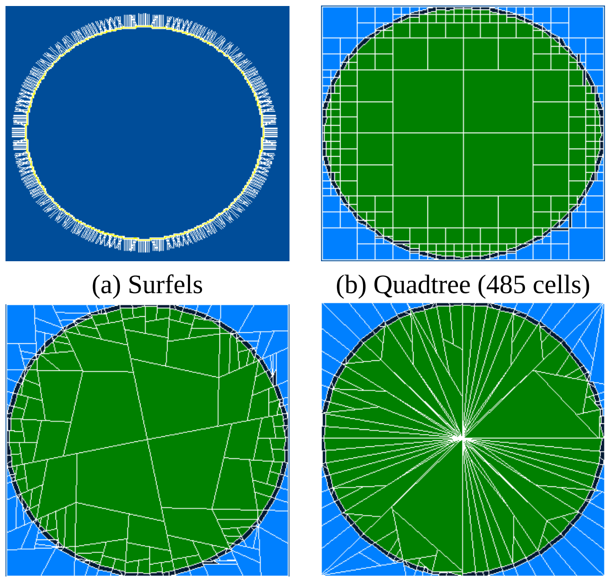

(a) Surfels (b) Quadtree (485 cells)
(c) CBSP-tree 1(341 cells) (d) CBSP-tree 2(254 cells)

Figure 1. Circle example. (a) Surfels and (b) Corresponding Quadtree. (c) and (d) shows two possible CBSP-trees.

1. Introduction

Object representation using point samples is becoming very common in Computer Graphics [12, 9]. The increased detail needed for models and 3D scanners are reasons to consider using points as geometric primitives instead of polygons (triangles). Points with associated attributes (e.g. normals) of an object surface are usually called *surfels* (surface elements) [14] (figure 1.a).

Surfel-bounded objects can be created in many ways (e.g. scanner acquisition). The ability to combine surfel-bounded objects using boolean operations is also very important, and efficient algorithms have been proposed to solve this problem [1, 2]. In such algorithms an important geometric query involves checking which parts of an object are inside another object. Spatial data structures such as octrees and KD-trees are used to reduce the natural quadratic complexity of this problem. In addition, boolean operations in such structures are simplified since they only have axis-aligned cuts.

In order to circumvent the problem of creating too many cells along curved parts of the model, a hybrid structure is proposed in [1] that uses a quadtree (or octree) with cuts that are not aligned with the coordinate axes at the leaves. This solution requires special treatment of the leaves, specially when computing boolean operations.

In this work we introduce Constrained BSP-trees (CBSP-trees) as a variation of BSP-trees that impose certain conditions on the directions of cuts. CBSP-trees can be seen as an intermediary structure between the BSP-tree [6] and the KD-tree [4]. The KD-tree corresponds to a hierarchical representation of binary partitions of space by axis-aligned cuts. On the other hand, BSP-trees also represent binary partitions of space, but with no restriction on the directions of cuts. This is one important aspect that differentiates BSP-trees from KD-trees: the direction of the partitioner cuts: **arbitrary** in a BSP-tree, **orthogonal** in a KD-tree. Another important difference is the complexity of the cells associated with regions in space: **bounded** in a KD-tree (all cells have four vertices in 2D), **unbounded** in a BSP-tree. The flexibility of the additional type of cuts allows the CBSP-tree to better adapt the shape of the objects represented, generating trees with fewer cells (figure 1). This is a more general framework that naturally handles non-orthogonal cuts (without special treatment such as in [1]).

The paper is outlined as follows. Section 2 discusses previous work. CBSP-trees are presented in section 3, followed by the presentation of the boolean operations algorithm in section 4. Results are presented in section 5 followed by conclusions and directions for future work in section 6.

2. Related work

Points as primitives were discussed in [12], and more recently in the work of Grossman and Dally [9, 8] and Pfister et al. [14], where the concept of *surfel* is introduced. A good survey on the subject can be found in [3].

Adams and Dutré [1] described an algorithm to compute boolean operations on surfel-bounded objects. Surfels are stored in a hybrid structure, that uses an octree at intermediate nodes, and non-axis aligned cuts at the leaves. The octree speeds up computation of the inside-outside classification required by boolean operations. A resampling operator is used to improve the quality of the edges of the resulting object. Later, the same authors improved their solution using programmable graphics hardware (GPUs) [2]. Another surfel-based boolean operation algorithm that uses the GPU was described in [10], based on a depth peeling technique. Boolean operations using distance fields was described in [5]. Octrees are also used here to speed-up calculations.

The octree, used in [1] and [5], was the preferred choice of spatial data structure due to its simplicity, regularity and easy boolean operation algorithms. Competing structures, such as KD-trees[4], BSP-trees[6] can also be considered, and algorithms to perform boolean operations are described in [15, 13]. Each structure has its own advantages and disadvantages, and hybrid structures are often used as a way of combining the good features of each approach. In this work we propose the CBSP-tree to allow an adaptive combination of the features of several spatial data structures.

3. Constrained BSP-trees

3.1. Definitions

A CBSP-tree corresponds to a BSP-tree with a predicate that defines which directions of cuts are valid. For computing boolean operations with surfels we define a criterion to identify valid cuts as described below.

Definition 1 *Valid Cell: A cell in R^n is called valid if it is a convex polytope in R^n that contains at most 2^n vertices and $2n$ faces.*

For example, in 2D, cells have 3 or (usually) 4 vertices. In 3D, cells have at most (usually exactly) 8 vertices. A *valid cut* is defined as (figure 2):

Definition 2 *Valid Cut: A cut c defined by a hyperplane in space R^n is called valid when it partitions a valid cell in R^n into exactly 2 valid cells.*

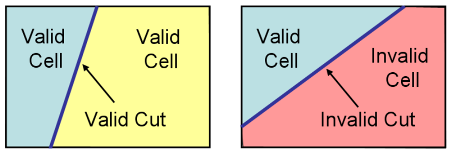


Figure 2. Valid and invalid cells (cuts)

Our decision to use the above definition of valid cut is directly related to the cells generated by the subdivision. Cells with bounded complexity can more efficiently be used in geometric problems such as point location, nearest neighbor search or intersection detection, as well as serve as better function approximations using interpolation.

3.2. Choosing Partitioners

The construction of CBSP-trees for representing surfels is controlled by a partition operation. We explore the added flexibility of CBSP-trees with a combination of cut selection strategies implemented using the *strategy pattern* [7].

The first selection mode uses a Principal Component Analysis [11] approach (called *PCA selection*). It consists in computing the eigenvectors representing the principal and secondary directions of the surfels positions. The corresponding eigenvalues represent the significance of each direction. Figure 3 shows principal and second component for a sequence of surfels. We use the direction given by the second component to define partitioner cuts.

Another strategy is to choose a valid cut from a list of candidates that span a subset of all possible directions in a

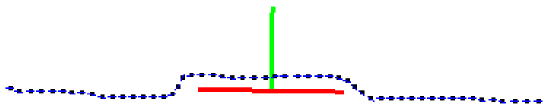


Figure 3. Principal components used to choose partitioners

given space. This is called *candidate selection*. We use the cut that maximizes empty space in one of its halfspaces. If none is found, we split the longest dimension, thus avoiding thin cells.

We observed that best results are obtained when we further subdivide space with cuts that are orthogonal to the previous chosen direction. In order to simulate this behavior, we chose to alternate between *PCA* and *candidate selection* strategies while building a CBSP-tree.

3.3. Stop Criteria

Partitioning stops when certain conditions are met. Two simple strategies have been tested. The first one uses an evaluation of a geometric criteria based on surfels alignment, while the second stops processing when a certain tree depth is reached In both cases, the area of the cell can be tested first, to avoid wasting time processing very small cells and also to avoid numerical errors. As expected, the number of surfels in a cell is used as stop criterion too. In our tests, the stop criterion was chosen so that partitioning is detailed while avoiding too many unnecessary cuts.

3.3.1 Surfels Alignment Criterion

In this strategy we first test the number of surfels. If the cell has less than lim surfels, we stop partitioning. The parameter lim can be configured as low as 1 or 2 (since two surfels will *always* be aligned), or a higher number can be chosen when that degree of precision is found to be too much.

If the cell has more than lim surfels, we check alignment using the PCA results based on an equation that relates the first and secondary components, such as the following formula: $sec\_eigval/(princ\_eigval + sec\_eigval)$. If the result is less than or equal to a given threshold, the surfels are considered to be sufficiently aligned.

3.3.2 Tree Depth Criterion

A simple alternative to stop partitioning is based on the depth of the tree. If the level of recursion when building the tree is less than $level$ (for example, $level = 4$), we continue dividing. When the tree depth is greater than $level$, we test the number of surfels in the cell. That means that partitioning stops when a minimum tree depth is reached *and* the cell has less than a given number of surfels.

3.4. Trim Partitioners

Cells containing surfels may require further processing after the stop criterion is met. In particular, when a cell contain nearly-aligned surfels, we can introduce additional cuts to limit the interior and exterior of the object. This can be accomplished by adding two parallel cuts such that all surfels lie in between them. This operation is called *trim* and the cuts used are called *trim partitioners*. However, some trim partitioners might violate the CBSP-tree restriction, generating invalid cells.[1]

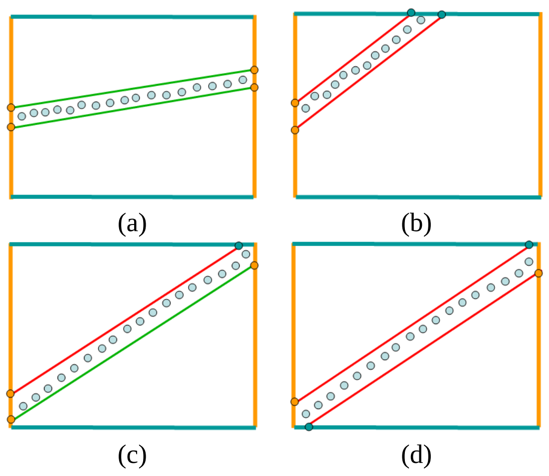


Figure 4. ***Trim*** **cases. (a) 2 valid cuts; (b) 2 invalid (same sides); (c) 1 invalid, 1 valid; and (d) 2 invalid (different sides)**

Trim partitioners are directly used if they are both valid. Otherwise, in order to keep the bounded complexity of the cells and all surfels in a small area, we add a few extra cuts to make the others valid. Figure 4 illustrates the different situations that can occur. The next sections describe how to handle invalid cuts.

3.4.1 Two Invalid Cuts, Same Sides

Figure 4.b shows two invalid partitioners (notice that they cut adjacent sides of the cell, rather than opposing sides). To correct them we first add the cut labeled *A* (figure 5) in a way that the original invalid cut now intersects opposite sides of the new cell. We can make one endpoint of *A* and *B* coincident, thus creating a cell with three sides (still

[1]Notice that, in [1], trim *does* generate cells that would not exist in normal octrees.

valid), but keeping all cells with the same number of sides simplifies the algorithm. The same idea is then applied to the other invalid cut, creating the cuts labeled *C* and *D*.

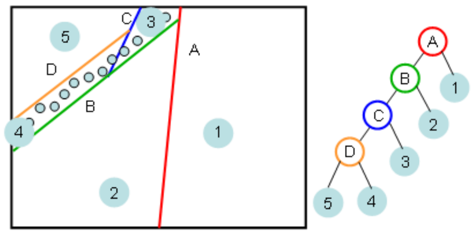


Figure 5. *Trim*: **correcting two invalid splits, when both intersect the same sides of the cell.**

3.4.2 One Invalid Cut

Figure 4.c shows a case where only one partitioner is invalid. As in the previous case, we add an extra cut (this time labeled *B*, see figure 6) that intersects the invalid one before it reaches the border of the cell. *B* could be any cut with a direction that makes the partitioning valid (such as a vertical cut). The one shown uses the middle of the valid cut as an endpoint, because it is simpler and always generates a valid cut.

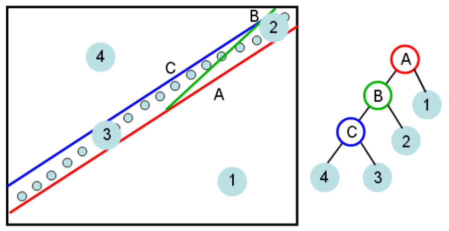


Figure 6. *Trim*: **correcting one invalid split**

3.4.3 Two Invalid Cuts, Different Sides

In figure 4.d, there are two invalid trim partitioners, but they intersect different sides of the cell, so we can't apply case A here. Instead, we cut the cell in half (preferably perpendicular to the longest dimension) and apply case B on both sides (figure 7). Notice that, although all examples used rectangular cells for illustration, they work for any cell with four sides.

4. Boolean Operations using CBSP-trees

4.1. Inside-Outside Classification

In order to compute boolean operations between CBSP-trees we first need to classify each leaf cell as *inside*, *outside* or in the *boundary* of the object. Every cell that contains surfels is classified immediately as a boundary cell. However, inside and outside cell classification is more involved and requires finding adjacent cells in the subdvision. By keeping a polygon that represents the geometry of each cell, we can use the following algorithm to identify neighbors for all leaf nodes:

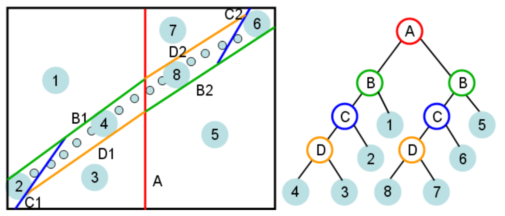


Figure 7. *Trim*: **correcting two invalid splits, each intersecting different sides of the cell.**

Algorithm 1 FindNeighbors

```
1: for each partitioner $p$ do
2:     $(C_l, C_r)$ = FindCellsIncidentToPartitioner(p);
3:     for each cell $c_l \in C_l$ and cell $c_r \in C_r$ do
4:         if $c_l$ is neighbor to $c_r$ by a segment $p_s$ of $p$ then
5:             Add $c_l$ and $p_s$ to the neighbor list of $c_r$ (and
               vice-versa)
6:         end if
7:     end for
8: end for
```

The highlighted partitioner in figure 8 has incident cells C_l={*2*, *6*, *7*} by its left half-space, and cells C_r={*3*, *8*} by its right half-space. Remaining cells (*1*, *5*, *4*, *9*) and any children they might have are not considered for further tests. Checking every cell from the left side against one in the right side allows us to conclude that cells *3* and *6* are neighbors by *neighbor segment c-d*. On the other hand, cells *7* and *3* are not neighbors because they do not share a common line segment. [2]

Once we find all cell neighbors, we calculate the average of the normal vectors of the surfels for every boundary cell. Since normals are not used individually, we refer to it simply as *normal vector*. Using the *normal vector* and the *neighbor segment*, a boundary cell can tell us whether a neighbor represents the space inside or outside the object. Suppose cell *6* is a boundary cell and cell *3* does not contain surfels (figure 9).

Notice that the line passing through the center of the surfels along the direction of the *normal vector* intersects a shared *neighbor segment*. This allows us to classify cell *3* as *outside* since the *normal vector* of cell *6* points to cell *3*. If

[2]neighbor segments are ignored if they are much smaller than cell sides

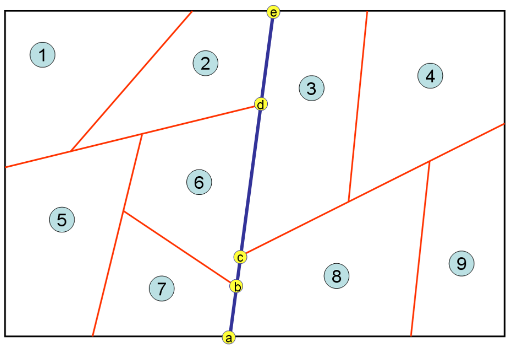


Figure 8. Finding neighbor cells

the vector had opposite direction, cell *3* would be classified as *inside*. Tests showed that using only the *normal vector* direction classifies too many cells in some cases.

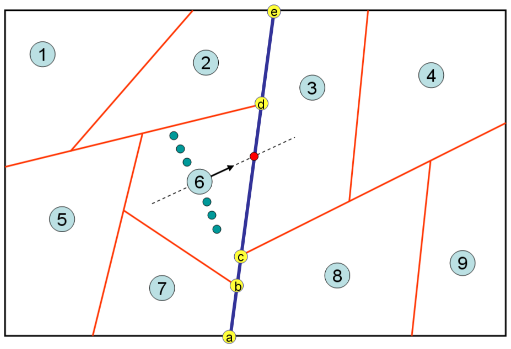


Figure 9. Classifying cell 3 as "outside"

If a cell is incorrectly missed by the above procedure, it is classified along the remaining unclassified cells by a simple flood-fill algorithm. Figure 10 shows an example of the final result obtained by the inside-outside classification algorithm.

4.2. Boolean Operations Algorithm

The *boolean operation* is performed as described in [1]. First we classify all surfels of one object as inside or outside the other (and vice-versa). This classification requires a point-location algorithm that finds the cell reached by a given surfel. If the surfel reaches a boundary cell, we use a classification based on the closest surfel. Adams and Dutré [1] suggest the use of a resampling operator to refine the surfel into smaller surfels, which is not fully implemented in our work since it will be more useful for the 3D case.

In some cases a surfel-by-surfel test is not necessary. For instance, when a cell associated with one tree node does not intersect the other tree (or intersects only with *outside* (*inside*) leaf cells of the other tree), all surfels in that cell (or its children) can be classified as *outside* (*inside*).

After this classification is finished, it is simple to combine surfels using boolean operations. For example, when performing a union operation, the result will have all surfels that were classified as *outside* (from both objects). Table 1 shows the parts of each object kept in the result.

Operation	**Surface of A kept**	**Surface of B kept**
$A \cup B$	Outside B	Outside A
$A \cap B$	Inside B	Inside A
$A - B$	Outside B	Inside A (inverted normals)
$B - A$	Inside B (inverted normals)	Outside A

Table 1. Parts of surfaces kept when performing boolean operations [1].

5. Results

Several 2D objects have been tested using CBSP-trees and quadtrees, with different partitioning strategies and stop criteria. We compared the number of leaf cells in the tree and a summary of results is shown in table 2. Every input file used the same stop criterion with both the CBSP-tree and the quadtree, in order to make the results comparable. A video showing some of the results can be found at `http://www.inf.ufrgs.br/~macfarias/cbsp`.

Input	**CBSP Leaves**	**QT Leaves**	**Stop criterion**	**Tree Constr. Time**
circle	254	485	Max. tree depth: 4 and max. 8 surfels	19ms
logo	1527	2333	Surfels alignment or less than 1 surfel	69ms
ufrgs	1315	1589	Surfels alignment or less than 8 surfels	61ms
koch	2220	2487	Max. tree depth: 4 and max. 8 surfels	110ms
coke	5417	7688	Surfels alignment or less than 2 surfels	292ms

Table 2. CBSP-Quadtree comparison.

The input file named *circle* obtained best results with almost unconstrained cuts (figure 1). Other input files obtained best results with most axis-aligned cuts (except in the *trim* operation, of course). See examples in figure 11. Still, the CBSP-tree has fewer leaf nodes since quadtrees often partition empty space when creating four cells. Boolean operation examples are shown in figure 12.

6. Conclusions and future work

In this paper we introduced the concept of a CBSP-tree as a more flexible spatial data structure. We used

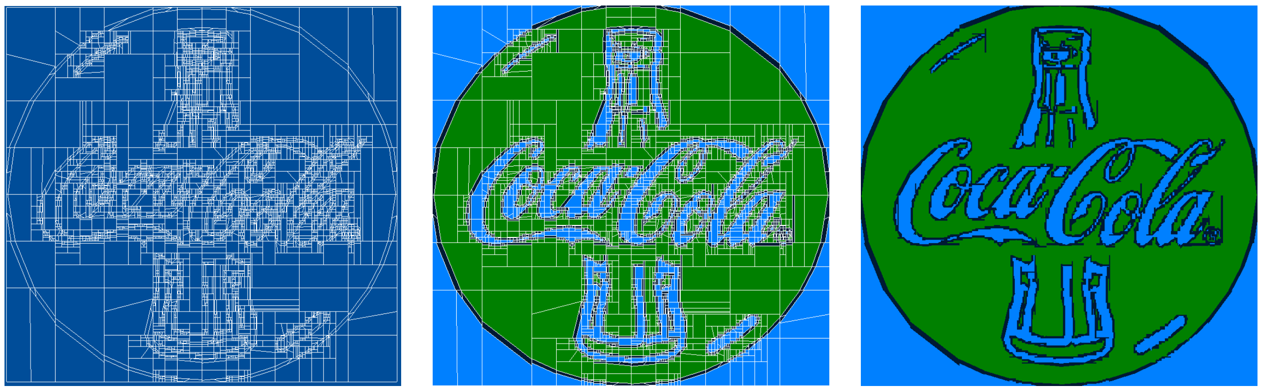


Figure 10. Inside-outside algorithm example. Left: subdivision. Center: cell classification. Right: cell classification, with polygon lines hidden

the problem of computing boolean operations on surfel-bounded solids to evaluate our proposal. The experiments show that the ability of the CBSP-tree to better adjust to the shape of objects leads to a subdivision that has a smaller number of cells. The fact that we can use cuts that are not aligned with the axes does not mean that they need to be always used. In fact, in some of the cases, axis-aligned cuts were dominant over the total number of cuts. However, in some objects (such as the circle) non-axis-aligned cuts are important to reduce the number of cells. The CBSP-tree handles the *trim* cuts in a more general way than the hybrid octree used in [1].

We are extending our approach and concepts to 3D and considering a GPU implementation. We are also analysing the time complexity of the algorithms. We plan to explore CBSP-trees in other problems, such as the representation of distance-fields (ADFs), texture mapping, and rendering with proxy-geometry.

References

[1] B. Adams and P. Dutré. Interactive boolean operations on surfel-bounded solids. In *ACM Transactions on Graphics*, pages 651–656, 2003.

[2] B. Adams and P. Dutré. Boolean operations on surfel-bounded solids using programmable graphics hardware. In *Eurographics Symposium on Point-Based Graphics 2004*, June 2004.

[3] M. Alexa, M. Gross, M. Pauly, H. Pfister, M. Stamminger, and M. Zwicker. Point-based computer graphics siggraph 2004 course notes. In *SIGGRAPH 2004*, 2004.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[5] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, pages 249–254. ACM Press, 2000.

[6] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[8] J. P. Grossman. Point sample rendering. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998.

[9] J. P. Grossman and W. J. Dally. Point sample rendering. In *9th Eurographics Workshop on Rendering*, pages 181–192, 1998.

[10] J. Hable and J. Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In *ACM Transactions on Graphics, Proceedings of SIGGRAPH 2005*, 2005.

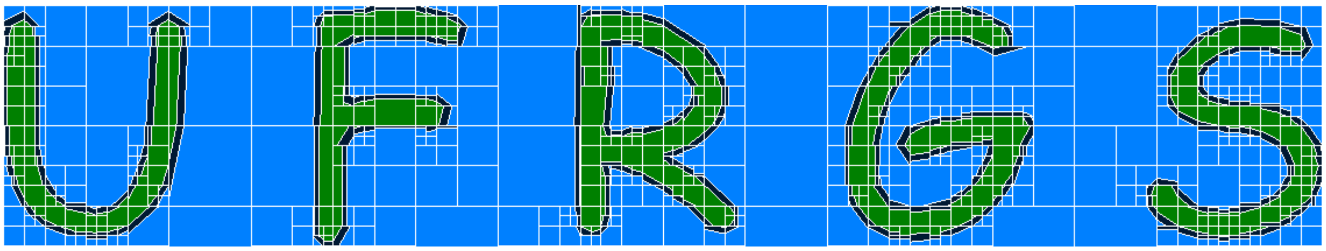
[11] I. T. Jolliffe. *Principal Component Analysis*. Springer, 2nd edition, 2002.

[12] M. Levoy and T. Whitted. The use of points as display primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, Jan. 1985.

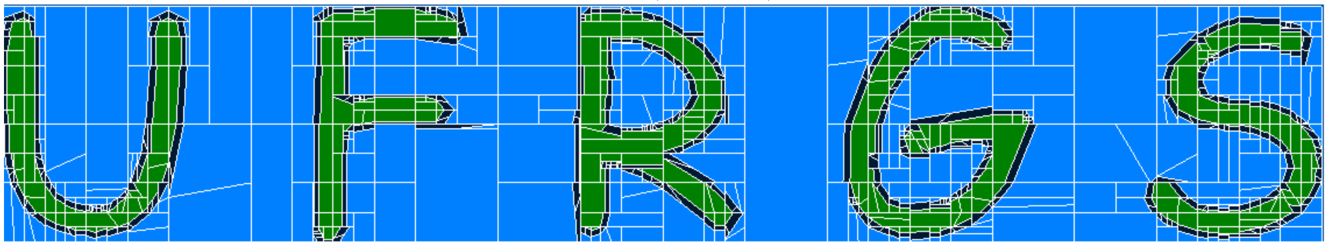
[13] B. F. Naylor, J. Amanatides, and W. C. Thibault. Merging BSP trees yields polyhedral set operations. *Computer Graphics*, 24(4):115–124, Aug. 1990.

[14] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 335–342, 2000.

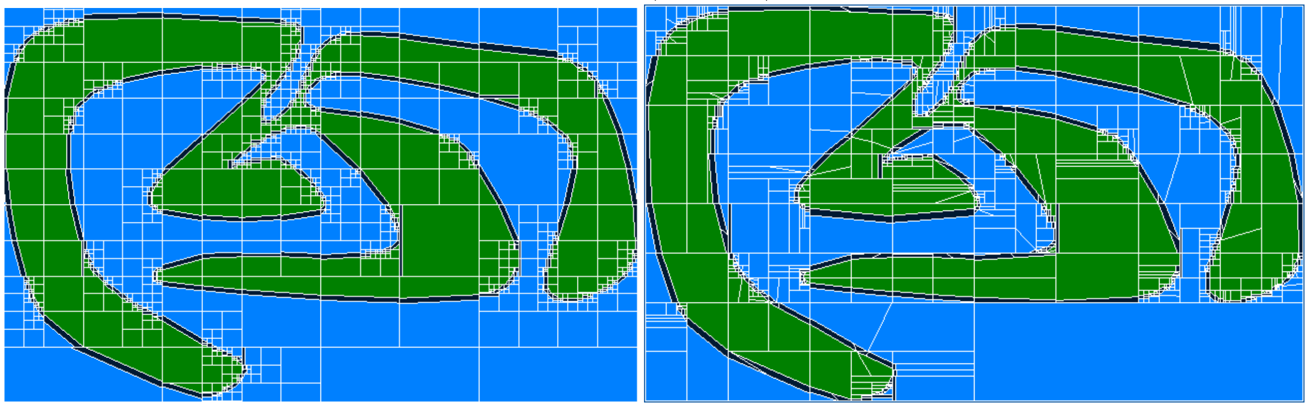
[15] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.



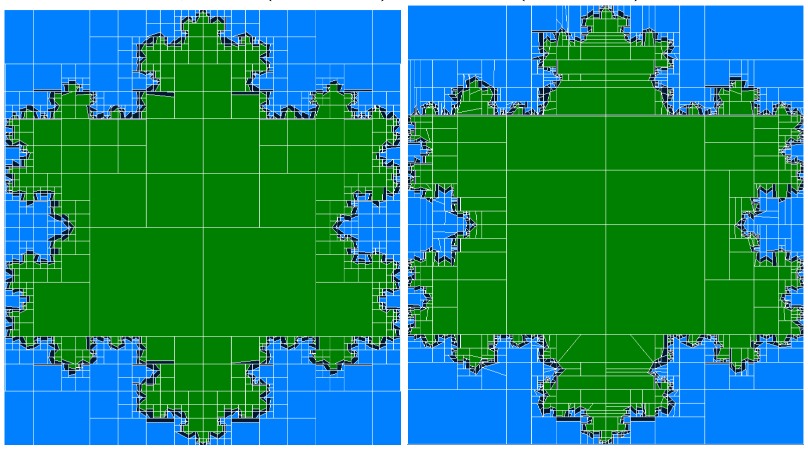
Quadtree(1589 cells)



CBSP-tree(1315 cells)

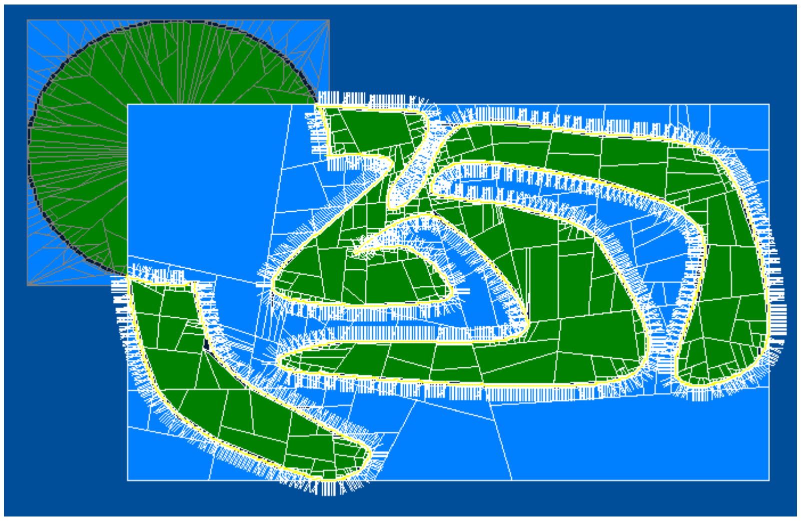


Quadtree(2333 cells), CBSP-tree(1527 cells)

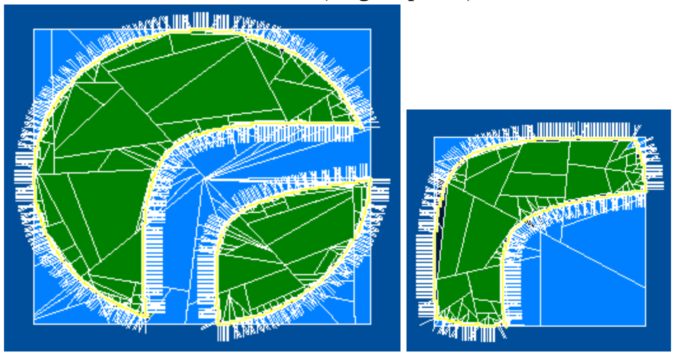


Quadtree(2487 cells), CBSP-tree(2220 cells)

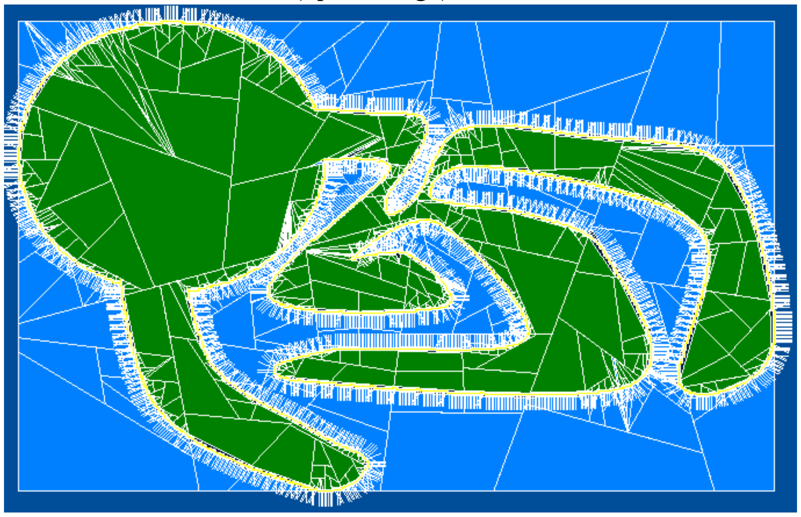
Figure 11. UFRGS, Logo and Koch results



Difference (Logo-Sphere)



Difference (Sphere-Logo) and Intersection



Union

Figure 12. Boolean Operation Examples