Boolean Operations on Point-Sampled Geometry

Zoltán Péter Gaál*

Faculty of Informatics
Eötvös Loránd Science University
Budapest / Hungary

Abstract

In computer graphics boolean operations have been around for years. In contrast, point sampled geometry models represent a new and stirring side of graphics which is researched mostly in the last couple of years. Papers about the combination of CSG and surfels only appeared recently, and it's hard to find a generalized overview of the possible choices. Therefor our goal is to fill this hole, to give an overview of the algorithms and give a way to merge them. In addition, we also try to focus on methods those give the most speed in terms of interactivity. We present implementation details and analyze efficiency. Finally, we propose a new algorithm for speeding up the update of space partitioning data structure needed for real time CSG operation.

Keywords: boolean operations, surfels, point-based geometry

1 Introduction

Boolean operations have been researched mainly for CAM applications. There are many robust algorithms which work on boundary represented solids. They handle the most extreme cases using tolerances [9]. There are also algorithms to hardware accelerate the evolution of the operation[7].

The trend for ever-denser 3D models has brought up an increased interest in algorithms that work on point primitives. Point sampled geometry can be described as a cloud of points placed stochastically on the surface of a solid. These points can be generated by using 3D scanners. Scanners can produce not only 3D positions, but normal vectors, material properties like diffuse coefficients too. At first these clouds were transformed into triangle or other polygon meshes and curved-face bounded solids. Recently the set of points is used for both geometrical representation and as rendering primitives.

*gaal_z@inf.elte.hu

2 The surface

Formally, the input $P = \{\mathbf{p}_i \mid 1 \leq i \leq n\}$ is a set of points. There might also exist scalar features for all the points like the material or color attributes. Using this point cloud we may introduce an S continuous Moving Least Square surface (MLS) that approximates the surface of the original solid [3]. S is given as the fixed points of a $\Psi_P : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ projection. To define the projection, first we have to find a local supporting plane of an $\mathbf{r} \in P$:

$$H = \{\mathbf{x} \mid \mathbf{n} \cdot \mathbf{x} - D = 0\}$$

by minimizing the weighted square distance

$$\sum_{\mathbf{p} \in P} (\mathbf{p} \cdot \mathbf{n} - D)^2 \phi(\|\mathbf{p} - \mathbf{q}\|)),$$

where $\mathbf{q}$ is the projection of $\mathbf{r}$ onto H. ϕ is the kernel of the MLS surface, and usually chosen as the Gaussian:

$$\phi(d) = e^{-d^2/h^2}$$

The H plane also defines an f_i height for all $\mathbf{p}_i \in P$ as $f_i = \mathbf{n} \cdot (\mathbf{p_i} - \mathbf{q})$. Finding the coefficients of the polynomial approximation g by minimizing the least square error for the

$$\sum_{i=0}^{n} (g(x_i, y_i) - f_i)^2 \phi(\|\mathbf{p} - \mathbf{q}\|)),$$

we may write Ψ_P as

$$\Psi_P(\mathbf{r}) = \mathbf{q} + g(0,0)\mathbf{n}.$$

The MLS projection requires a lot of calculations [2]. Fortunately, taking into consideration some ideas, we may speed up the algorithms:

1. The eigen-analysis of the weighted covariance matrix $B \in \mathbb{R}^{3\times3}$,

   $$b_{jk} = \sum_i \phi_i (\mathbf{p}_{i_j} - \mathbf{r}_j)(\mathbf{p}_{i_k} - \mathbf{r}_k),$$

   where $\phi_i = \phi(\|\mathbf{p}_i - \mathbf{r}\|)$ are fixed, gives a good approximation for the normal vector of the H plane [8].
   This analysis can be also used to extract topology feature information like edge, planarity. For more information see [4].

	Q_1	Q_2
$S_1 \cup S_2$	$\{\mathbf{p} \in P_1 \mid \Omega_{P_2}(\mathbf{p}) = \text{False}\}$	$\{\mathbf{p} \in P_2 \mid \Omega_{P_1}(\mathbf{p}) = \text{False}\}$
$S_1 \cap S_2$	$\{\mathbf{p} \in P_1 \mid \Omega_{P_2}(\mathbf{p}) = \text{True}\}$	$\{\mathbf{p} \in P_2 \mid \Omega_{P_1}(\mathbf{p}) = \text{True}\}$
$S_1 \setminus S_2$	$\{\mathbf{p} \in P_1 \mid \Omega_{P_2}(\mathbf{p}) = \text{False}\}$	$\{\mathbf{p} \in P_2 \mid \Omega_{P_1}(\mathbf{p}) = \text{True}\}$
$S_1 \setminus S_2$	$\{\mathbf{p} \in P_1 \mid \Omega_{P_2}(\mathbf{p}) = \text{True}\}$	$\{\mathbf{p} \in P_2 \mid \Omega_{P_1}(\mathbf{p}) = \text{False}\}$

Table 1: Classification of surfels

2. We don't need to take into consideration all the points of P since the points far away from $\mathbf{r}$ modifies the value of 2 slightly because of the ϕ kernel. So they play no role in the definition of Ψ_P. Usually taking the k-nearest points gives a sufficient approximation.

3. If for some reason, e.g. in section 4.1, we still need to find the exact projection of an $\mathbf{r}$ point onto the MLS surface, we may use an iterative approach. Pauly suggested [8] to estimate $\mathbf{q}$ directly, which determines $\mathbf{n} = (\mathbf{x} - \mathbf{q})/\|\mathbf{x} - \mathbf{q}\|$ and $D = \mathbf{q} \cdot \mathbf{n}$. It's is done by a Newton-type iteration. First an initial value is chosen for $\mathbf{q}_0$, by finding the closest point in P to the $\mathbf{x}$ position. Then a H_0 plane is calculated for this $\mathbf{q}_0$ using the Eigen-analysis. The next estimation for $\mathbf{q}$ is computed using the orthogonal projection of $\mathbf{x}$ onto H_0. This procedure can be iterated to get $\mathbf{q}_{i+1}$ by projecting $\mathbf{x}$ onto H_i, which is the weighted least square plane fitting to the local neighborhood of samples of P around the previous $\mathbf{q}_i$ estimation. To avoid the oscillation a λ biasing factor is introduced such that instead of taking $\mathbf{q}_{i+1}$ as the next point of approximation, a $(1-\lambda)\mathbf{q}_i + \lambda\mathbf{q}_{i+1}$ is used. [8].

For the exact visualization the point cloud itself is not sufficient. Since drawing the points results in a hallowed picture. Where the sample density is less, the picture quality will decrease as well. To correct this a point is extended to a so called surfel. Actually a surfel is an elliptical disk, whose plane is the H local reference plane, the two axis of the disc are the same as the ones used to define the approximating g polynomial. Finally the radius of the disk refers to the local density of the sample. It's suggested to use the radius of the sphere containing the k-nearest point. Zwicker have studied this question and used the elliptical weighted average (EWA) surface splatting to solve the problem. In our example we've used a simplified, hardware accelerated, 2-pass method for the rendering.

Note that most of the time the input only contains the position of the points, but most of the algorithms which operate on point clouds require the normals of the surface at the input points. Therefore practically, it is worth to calculate the normals of the MLS surface using e.g. the k-nearest neighborhood in the preprocess step and store it in the point cloud data.

3 Boolean operation

The boolean operations, we'll take into consideration are the union, intersection and subtraction. The ideas discussed here can be easily extended to evolve whole CSG trees. So without restricting the generalization we might state that our goal is to find the P_0 surfels bounding the union, intersection or the subtraction of two solids represented by P_1, P_2. $P_0 = P_1 \Upsilon P_2$, where$\Upsilon \in \{cup \cap \setminus\}$.

Before going into details, let's have some more definitions. It's easy to see that, to get the evolution of an operation we have to choose a $Q_1 \subseteq P_1$ and a $Q_2 \subseteq P_2$ plus a set of newly generated sample points to have the P_0 representation. That is we have to

1. find the Q_i subsets

2. and extend the subset as desired.

Let Ω_P be the logical function that tells if an $\mathbf{x} \in \mathbb{R}^3$ is in the V volume being represented by P:

$$\Omega_P(\mathbf{x}) = \begin{cases} \text{True} & \text{,if} \quad \mathbf{x} \in V \\ \text{False} & \text{,if} \quad \mathbf{x} \notin V \end{cases} \tag{1}$$

The first step of the evaluation of the boolean operation is finding Ω_P. Using this logical function and table 1 we may classify the surfels into two groups: surfels which belong and surfels which do not belong to the final solid.

The simplest algorithmic method to evaluate function (1) for a point $\mathbf{x}$ is finding the nearest surfel t in solid P and evaluate

$$\Omega_P(\mathbf{x}) = \mathbf{n}_t(\mathbf{x} - \mathbf{x}_t) < 0,$$

where $\mathbf{n}_t$ is the normal of surfel t and $\mathbf{x}_t$ its position. The brute force implementation of this step results in an $O(n_1 n_2)$ algorithm, and since the surfel counts of each point cloud can reach 60K or 200K, this approach can be hardly qualified as interactive or realtime.

The second step of the algorithms tries to repair visual anomalies, caused by the fact that the MLS surface of solid P_1 can cut the ellipsis of the surfels in solid P_2.

Thus our goal is that, the classification of the surfels and the extension of the $Q_1 \cup Q_2$ union represents more precisely the intersecting part of two solids.

4 Approaches

4.1 Pre-interactive surface space method

Pauly suggested an analytical approach [8]. It slightly improves the brute force algorithm. An $\mathbf{r}$ surfel can be classified if we find its projection onto the MLS surface of the other object. As we know from differential geometry $\mathbf{r} - \mathbf{y}$ is aligned with the surface normal at $\mathbf{y}$, and thus $\Omega_P = (\mathbf{r} - \mathbf{y}) \cdot \mathbf{n}_y < 0$. Fortunately we don't need to map all the points of a solid. If the distance from $\mathbf{r}$ to the closest $\mathbf{s}$ surfel defining the other object is less than the local sample spacing ν_s, the algorithm gives a good result. Unfortunately if the distance exceeds the sample spacing the result might be incorrect. To avoid this we have to project such surfels onto the surface as explained in section 2 . This algorithm can be further improved since for all $\mathbf{c}$ points within $\|s - r\| - \nu_s$ distance from a classified $\mathbf{r}$ surfel gets the same classification. Formally:

$$\Omega_P(c) = \Omega_P(r) \forall c \in \{c \mid \|c - r\| < \|s - r\| - \nu_s\} .$$

In the second step of the boolean operation we need to refine the intersecting curve of the two surfaces. For this we have to find closest pairs in the Q_1, Q_2 sets. Usually the sample distribution of the two sets are different so an up-sampling operator have to be used. This is usually defined by some simple subdivision rule.

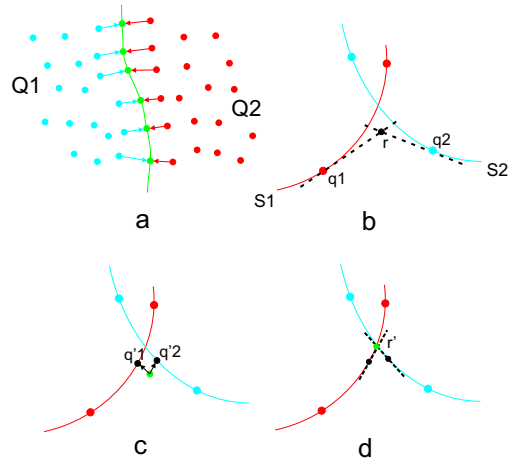

Figure 1: *Finding the intersection curve of two MLS surfaces.*

Using these closest pairs ($\mathbf{q}_1 \in Q_1$,$\mathbf{q}_2 \in Q_2$) a Newton-type iteration finds the exact samples on the intersecting curve as follows. We may define a closest point $\mathbf{r}$ to $\mathbf{q}_1$ and $\mathbf{q}_2$ on the intersection of the two tangent planes $\mathbf{q}_1$ and $\mathbf{q}_2$. Then using the Ψ_{P_1},Ψ_{P_2} operators we may project the r point onto the MLS surfaces. Repeating the process for the two new $\mathbf{q}_1^{'}$,$\mathbf{q}_2^{'}$ projections we get a better approximation for a point on the intersecting curve. As Pauly stated due to the quadratic convergence of the Newton iteration, this typically requires less than three iterations.

4.2 Interactive object space method

Adams and Dutre [1] proposed an interactive method for boolean operations. The points are considered as surfels with a finite radius. Their method partitions the space that is defined by the bounding box of the surfels by an octree data structure. The surfels are assigned into exactly 1 leaf node of the octree. The nodes are "colored" or labeled as inside, outside or border cells in a preprocess step (see section 6.1 for further details). Nodes that classified as borders are further subdivided by 2 planes. This 2 parallel planes is determined by enclosing the presented surfels in the cube, therefore the 3 separate volumes can be colored also as inside, outside and border areas.

In the first step of the boolean operation (see section 3), the algorithm does not test the surfels of P_1 individually, whether or not they belongs to the inside, outside or border area of P_2, but instead the octree nodes of P_1 are tested against the octree of P_2. If e.g. a node only intersects with the interior cells of P_2, all surfels of this node can be classified as inside surfel. If the situation is not so obvious, testing the surfels of this node one at a time is unavoidable. If the surfel belongs to an inside or outside node, the decision is easy. If it belongs to a border cell, it is tested against the 2 parallel planes. If it is still classified as border, it is tested using the normal of the nearest surfel of P_2. This is degraded to the brute force way, since the nearest surfel is searched in the whole P_2.

In the second step of the boolean operation, we need to find the exact intersecting curve. Adams suggested to use a resampling operator that replaces each intersecting surfels by some smaller surfels. During this refinement the surfels are considered as discs. This way the intersecting curve will be approximated by more surfels.

4.3 Interactive image space method

A different approach was used by Martin Wicke in 2004. He proposed a solution [5] which does not need any resampling operator in object space, only different clipping methods in image space.

The method classifies the surfels into 4 categories:

- **Surface surfels**, that must be present in the result (union, intersection, etc.) without any modification
- **Inside edge surfels**, which also belong to the result, but have an intersection with the other solid
- **Outside edge surfels**, which do not belong strictly to the result, but have cross sections with the edge
- **Outside surfels**, which do not belong to the boolean result.

Outside surfels can be discarded from rendering. Surface surfels can be rendered with any hardware accelerated splatting algorithm. Edge surfels are treated specially with a fine tuned software renderer, since they may clip

*	Grid (m^3)	Kd-tree	Octree
building time	$O(n)$	$O(n\log(n))$	$O(n\log(n))$
finding surfel	$O(n/m)$	$O(\log(n))$	$O(\log(n))$
k-nearest neighborhood	$O(k)$	$O(n\log(n))$	$O(n\log(n))$
in-out classification	yes	yes	yes
managing dynamic scenes	yes	restructuring	restructuring

Table 2: Property summery of space partitioning structure.

each other. This classification is calculated each time the different solids change their position.

The bulk of the computation is done in the rendering phase. The rendering contains 3 passes. In the first phase, the edge surfels are rendered to a "clip buffer", which has the same resolution as the final image. Each cell (pixel) of the clip buffer contains a list of surfels, which would be visible in that pixel. With the aim of the clip buffer, the clipping partners can be determined. Note that two surfels do not clip each other if they belong to the same object.

The next pass renders clipped surfels with a special software renderer that can handle not only the clipping of two surfels (as in the approach presented in section 4.1), but can render corners or arbitrary number of intersections. The inside/outside test determines whether a fragment needs to be clipped.

In addition, Wicke improved the inside/outside test that decides which side of the surface a point belongs to. The most accurate solution is the MLS projection operator. However, it is not useful in interactive systems. As it has been mentioned the common practice is to find the closest surfel and determine the status of the point by inspecting the normal vector of the closest surfel. It is proposed that by using the two closest surfels, the classification errors can be reduced.

The final pass of the rendering uses hardware splatting for surface surfels and finally the image is merged with the result of the previous phase.

The advantage of this approach is that it can easily handle the rendering of a multiple level CSG tree without the need to generate the solids at the intermediate levels. Other good feature is that it is possible to zoom arbitrarily to the edges without visual glitches. However, since the rendering performance depends linearly on the number of visible edge pixels, if we zoom to the edge, the rendering time falls rapidly. Note that in the object space method (section 4.2), the result of the boolean operation is computed only once, after the objects are animated. In that case, when only the camera moves, the result can be rendered with using only minor hardware computational efforts. This suggests that the object space approach is better for walk-through applications.

5 Data structure

Let's go back to the interactive object space method, which we discussed in section 4.2. The rest of this paper is focused strictly on this approach. It's obvious that, there's no way to handle 50k-300k of points without having any data structure more complex than a simple set. We have no connectivity or alignment information about the surfels, thus the only way to organize them is to use some kind of space partitioning.

Generally we have 2 choices: a grid-like structure and the well-known space partitioning trees.

5.1 Grid

A grid subdivides the space into a $n * n * n$ cells. A cell may contain any number of surfels (figure 2/a). Since the structure of a grid is static, it can be used efficiently for dynamic scenes. The grid needs no restructuring only the surfels in subject have to be moved from one cell to another.
The k-nearest neighborhood can be also found fast because the neighborhood of a cell can be accessed in $O(1)$ step. Thus, if the current cell does not contains enough points, the 9 cells around this one can be accessed in constant time.

But the strict structure is also a drawback since the subdivision does not depends on the local sample density. It's possible that a cell contains hounders of points while other cells contains only 1-2 points. By increasing the value of n, the 'crowded' cells become less dens, but the other parts had to be partitioned further as well increasing the memory requirements.

5.2 Hierarchical tree

The other approach is to use a hierarchical space partitioning tree. In common we might state that these structures are built recursively as follows:

- Let the whole scene be the root of the tree and also the starting point of the recursion. (Later we'll refer to the unpartitioned parts of the space as *cells*.)
- Find the appropriate dividing planes for the current cell and divide it into subcells. Substitute the original cell with a node in the graph having the subcells as the children.

- This partitioning is repeated till we reach a maximum depth level in the recursion or a minimum in the number of surfels contained in the current cell.

We may differ the *data-driven* and the *space-driven* partitioning. In the first case the space is partitioned adaptively to the sample density rather than regularly in space as it is in the second case. See figure 2/c,d.

Both of them have their advantages and disadvantages. In the following section we'll focus on the space-driven approach, especially on the axis-aligned octree, but these algorithms can be easily adopted to the similar space partitioning hierarchies. We'll show that, using the information represented by an empty cell (that's a part of the space contains no boundary) may dramatically increase the efficiency of bool evaluation.

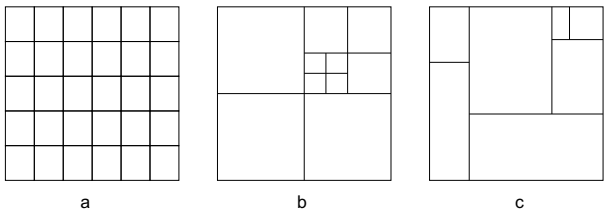

Figure 2: *Typical examples for grid (a), space-driven (b) and data-driven (c) space partitioning.*

6 Octree coloring

For each solid we construct an axis-aligned octree as described in 5.2. Than the cells of these trees are classified as being inside outside or on the border. This tree coloring is usually performed in a preprocess step, however it must be executed again, when the user places the interacted object to its final position and wants to start a new boolean operation on the result of the last phase.

6.1 Per level coloring

This classification is done as follows [1]. First the non-empty leaf nodes are classified as boundary cells. Than all the other (empty) cells are classified from the leaves to the root of the tree according to the neighboring cells. It is important to have a common understanding on what the neighboring relations means. The neighbors of a cell C are those child nodes of its parent, which can be accessed from node C by an orthogonal route. Diagonal nearby cells do not count. It means that each cell has exactly 3 neighbors. Other assumption of this algorithm is that the point cloud is closed and evenly sampled. A resampling process should be run on the solid, if it is not the case.

There are three different cases for the classification of a node:

1. A cell has only one non-empty neighbor
2. A cell has more than one non-empty neighbor
3. A cell has no non-empty neighbor

In the first case the cell is classified according to the non-empty neighbor. We find the closest s surfel to the cell and if it looks toward the cell, it is colored as outside, otherwise it is colored as inside.

That is if we note the center of the current and the neighboring cells as $\mathbf{c}_c$ and $\mathbf{c}_n$, and the normal of the surfel as $\mathbf{n}_s$, than the cell is declared inside [outside] if $(\mathbf{c}_n - \mathbf{c}_c) \cdot \mathbf{n}_s > 0$ $[(\mathbf{c}_n - \mathbf{c}_c) \cdot \mathbf{n}_s < 0]$.

In the second case we consider only one of the neighboring cells as before.

In the last case we classify the cell as the neighborhood. Since a node in an octree contains at least one non-empty cells, this classification can be done. If we are in the third case and the neighbor has no color yet, we skip the current cell and color the neighboring cells. Because of the existence of the non-empty cell in the node, in the worst case we need 3 steps to classify the whole node.

By this coloring of the tree we've got a draft representation of the interior of the solid bounded by the surfels. However, in a last step, this can be further improved by partitioning only the boundary nodes of the octree. Finding an average normal for a boundary cell by $\mathbf{n} = \sum \mathbf{n}_s / \| \sum \mathbf{n}_s \|$, we may define two parallel planes those surround the surfels: $\mathbf{p}_i = \mathbf{n} \cdot \mathbf{x} - d_i, i = 1, 2$; where $d_1 = \min(\mathbf{n} \cdot \mathbf{x}_s)$, $d_2 = \max(\mathbf{n} \cdot \mathbf{x}_s)$ and $\mathbf{x}_s$ are the positions of the surfels. Than similarly as for the cell classification we may classify the three parts of the cells as inside, outside, border.

Thus we have an even better approximation for the interior (exterior) of the solid, which will be useful when we classify the surfels of the other solid as being interior or exterior.

6.2 Octree flood fill

In the first and in the second case of the per level coloring approach having a center of a cell in our hand, we have to find the closest surfel to get the normal of the surface. As the number of the surfels increases, this activity takes too much time and becomes the bottleneck. We try to reduce the number of times this procedure is executed by classifying a cell according to its classified neighbor. If the cell has not got a marked neighbor, we leave the cell untouched for further processing. In the same time, we extends the domain of the "neighbor" relationship to be valid across different levels of the octree.

Our algorithm works like a 2D drawing application (e.g. GIMP), which paints a closed area bounded by a border-line. However in our case the algorithm should work in 3D and the border-line is defined by the nodes of the octree that are still classified as boundary cells.

The flood fill starts by finding a suitable "seed" node, which should achieve the following requirements:

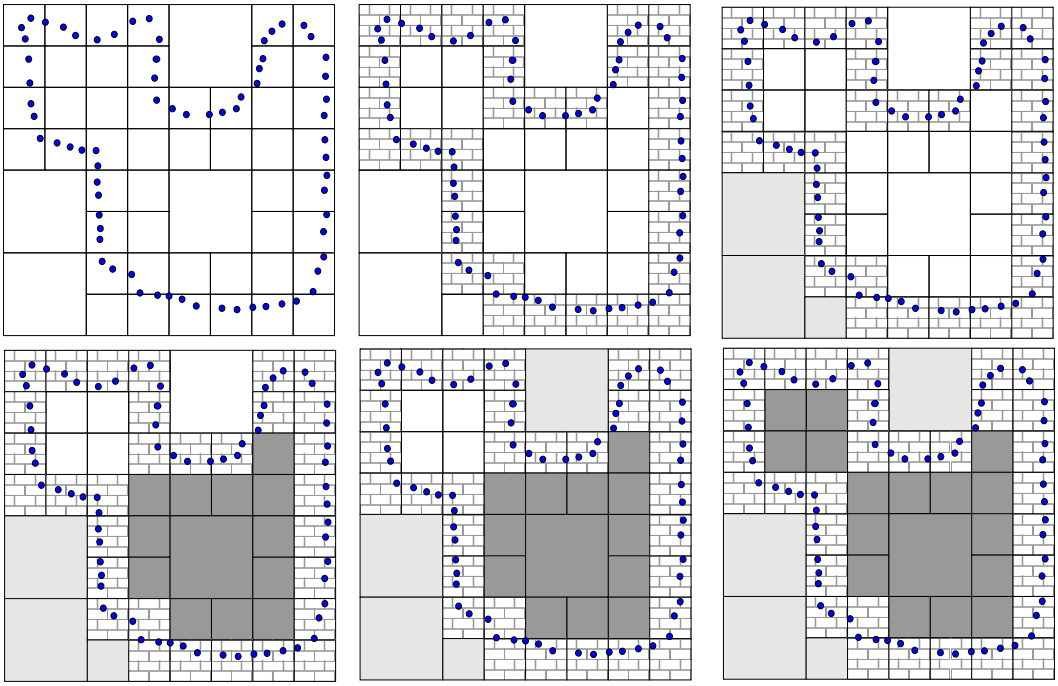

Figure 3: *The illustration of the flood fill octree coloring in 2D for depth d = 3.*

1. It is a leaf node
2. It is not marked yet
3. It has a neighbor node which is a border

As before, the status of the seed node is determined by finding the nearest surfel, and testing the normal vector against with.

The seed node is then put into a queue and an iteration is started. In each step of the iteration a node is fetched from the top of the queue. We than find the neighbors of the node, but this time we seek for next cells that can be sit in any other level of the octree. One way of finding such cells is generating points which are situated from epsilon distance from the bounding box of the current node and find that leaf of the octree, which contains this point. If the recently found neighbor cell is not assigned any status yet, we mark it by the status of the seed node and add the cell to the queue.

If the queue becomes empty, another seed node is chosen and the iteration starts again as it is illustrated on figure 3. Note that, it is possible that the octree has isolated "inside" areas, but this method can handle such cases too.

7 Implementation

It was not our goal to create an all-in-one complete code from MLS surface reconstruction, edge detection to point set ray-tracing or to implement the different kind of resampling operators, or to strive for experiencing the different kind of beautiful rendering and splating techniques.

number of surfels	per-level (msec)	flood fill (msec)	gain
40K	128	112	14%
62K	187	161	16%
134K	243	202	20%

Table 3: Profiling results of the tree building and coloring

Instead, we created an application which cover small areas from that huge topic mentioned before. To illustrate the classification of the surfels by tree-coloring, we have implemented the storage of points in an octree. This octree can be colored according to the 6.1 or 6.2 section.

Finally the rendering process had to be implemented, since without it the program cannot be used. First a simple point based rendering was created without taking any effort to create nice pictures. Later a simple, hardware accelerated algorithm was born. The surfels are represented as an alpha textured quad. Since the texture is constant no real EWA splatting or similar method is used [6]. Only a simple alpha-blending function is applied on a Gaussian-like texture.

The rendering process is a 2 pass method. In the first pass only the Z-buffer is filled to avoid the blending of the undesired parts. In the second phase the frame buffer is filled using the pregenerated Z-buffer and alpha-blending the surfels. This way the result was quite good.

With the help of the space partitioning, we classified the surfels in the other solid and vica-versa, and we can achieve boolean operations with interactive rates.

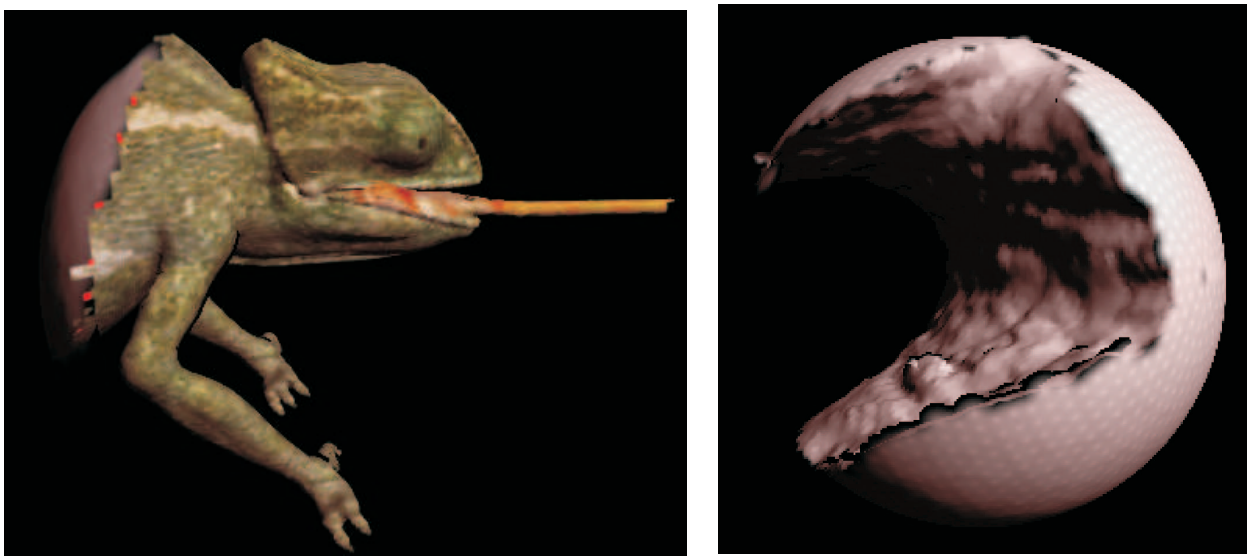

Figure 4: *Dense models can be used in boolean operations*

When working on relatively sparse solids (3K), 9fps was achieved easily without too much optimalisation. The brute-force approach was 4 times slower, however as the number of surfels increased the advantage of space partitioning becomes more self-evident. On larger models, like the chameleon model (100K) 4fps was reached.

The benefit of our flood filling algorithm is not exploited in the interaction time, but in the preprocess or and in the update phases. The comparison of the per-level coloring and the flood filling is presented in table 3, which shows that by increasing the number of surfels the algorithm becomes more and more useful.

As Adams have reported, we confirm also that the fastest interaction is achieved by choosing the octree depth to 4. In case of more than 100K surfels, the depth of 5 can be considered also.

Figure 5: *Different visualisation possibilities in our program*

8 Conclusion

The aim of this paper was threefold. First of all, we intended to give an overview on the current state of boolean operations on surfel bounded solids. We discussed and explored deeply the interactive method presented by Adams, and finally we've tried to improve his work.

9 Acknowledgments

This present work represents part of my Master Thesis which discusses B-rep, volumetric and point cloud geometric models in connection with boolean operations. I would like to thank György Antal, my supervisor for all his help and his precious orientation. Thanks go also to Incode Ltd. for providing the pleasant environment and other facilities.

References

[1] Bart Adams, Philip Dutre. *Interactive Boolean Operations on Surfel-Bounded Solids*. ACM, 2003.

[2] Marc Alexa, Darmstadt Daniel, Shachar Fleishman, David Levin, Claudio T. Silva, Johannes Behr. *Point Set Surfaces*. IEEE Visualization, 2001.

[3] Mark Pauly, Leif P. Kobbelt, Markus GrossShape, Richard Keiser. *Shape modeling with Point-Sampled Geometry*. Computer Graphics Proceedings, Annual Conference Series. ACM Press / ACM SIGGRAPH, 2003.

[4] Mark Pauly, Markus Gross, Richard Keiser. *Multiscale Feature Extraction on Point-Sampled Surfaces*. EUROGRAPHICS, 2003.

[5] Markus Gross, Martin Wicke, Matthias Teschner. *CSG-Tree Rendering for Point-Sampled Objects*. Computer Graphics and Applications, Proc. Pacific Graphics PG'04, Seoul, Korea, pp. 160-168, 2004.

[6] Matthias Zwicker, Jeroen van Baar, Markus Gross, Hanspeter Pfister. *Surface Splatting*. SIGGRAPH, 2001.

[7] J O'Sullivan, O'Loughlin. *Real-Time Animation of Objects Modelled using Constructive Solid Geometry*. CG, 1999.

[8] Mark Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. A dissertation submitted to the Federal Institute of Technology (ETH) of Zurich, 2003.

[9] Mark Segal. *Using Tolerances to Gurantee Valid Pilyhedral Modelling Results*. Computer Graphics, Vol.24, No.4, 1990.

[10] Szirmay-Kalos László, Csonka Ferenc, Antal György. *Háromdimenziós grafika, animáció és játékfejlesztés (Hungarian book)*. ComputerBooks, 2003.