

FZD-514

Polyhedral Surface Approximation of Non-Convex Voxel Sets and Improvements to the Convex Hull Computing Method

Henrik Schulz

März 2009

Wissenschaftlich-Technische Berichte
FZD-514 2009 · ISSN 1437-322X

WISSENSCHAFTLICH-TECHNISCHE BERICHTE



Forschungszentrum
Dresden Rossendorf

Wissenschaftlich-Technische Berichte
FZD-514
März 2009

Henrik Schulz

**Polyhedral Surface Approximation of
Non-Convex Voxel Sets and Improvements
to the Convex Hull Computing Method**

Bibliothek D 120



10029070X



**Forschungszentrum
Dresden Rossendorf**

Polyhedral Surface Approximation of Non-Convex Voxel Sets and Improvements to the Convex Hull Computing Method

Henrik Schulz

Forschungszentrum Dresden - Rossendorf
Department of Information Technology
PF 510119, 01314 Dresden, Germany
h.schulz@fzd.de

Abstract

In this paper we introduce an algorithm for the creation of polyhedral approximations for objects represented as strongly connected sets of voxels in three-dimensional binary images. The algorithm generates the convex hull of a given object and modifies the hull afterwards by recursive repetitions of generating convex hulls of subsets of the given voxel set or subsets of the background voxels. The result of this method is a polyhedron which separates object voxels from background voxels. The objects processed by this algorithm and also the background voxel components inside the convex hull of the objects are restricted to have genus 0. The second aim of this paper is to present some improvements to our convex hull algorithm to reduce computation time.

1 Introduction

An often arising problem in the field of three-dimensional image analysis is the efficient encoding of the surface of a digital object which is given as a set of voxels. The most popular approach is that of the triangulation, not only because of the simplicity of triangles but also because of the existing hardware support for tasks in the field of computer graphics.

A widely used approach to triangulate voxel objects is the Marching Cubes Algorithm by Lorensen and Cline [1]. It has a very low time complexity, i.e. it is linear in the number of voxels, which makes this algorithm applicable in practical tasks. But it has also two important drawbacks. First, the number of generated triangles is in most cases greater than the number of surface elements (faces) of the original voxel image and second, the orientation of the triangles is limited to a few directions. This is not desirable when an approximation of the original object (before digitization) is needed, which has a smooth surface with a constant curvature, for example.

Other triangulation methods use a divide-and-conquer approach. The algorithm described in [2] is applicable not only in 2-dimensional spaces to produce Delaunay triangulations, but also in higher dimensions, i.e. also in the 3-dimensional case. The algorithm separates the input data into two subsets and constructs the triangulation of the subsets recursively.

Another possibility consists in creating the Voronoi diagram [3] of a set of points using the duality between Delaunay triangulations and Voronoi diagrams. Efficient

algorithms for constructing Voronoi diagrams in 2D are well known [4]. The concept can be easily adopted to 3D.

Sometimes there exists the necessity to generate a more economical surface than a triangulation. Especially when triangulations would have lots of coplanar triangles, a polyhedral surface (a surface containing faces with more than three edges) would be much more efficient. The problem of approximating polyhedral surfaces is also well studied in the field of computational geometry [5, 6, 7].

In [8] we have already shown that for a convex voxel set the convex hull is such a polyhedral surface. In this paper we present an improvement of this algorithm to approximate non-convex objects, too.

The formal task to be solved is the following: Given a set V of voxels we want to create a closed polyhedral surface H containing V with the minimum surface area. Since there can be more than one polyhedron with the minimum surface area, we search the one with the minimum number of faces.¹ The polyhedral surface H shall separate object voxels (interior) from background voxels (exterior) in such a way that no object voxel lies outside H and no background voxel lies inside H . Voxels lying on H , especially the vertices of the polyhedron, have to be marked as being object voxels or background voxels. For the first criterion (minimum surface area) we will only present an approximation here. The second criterion (separation) is stated to provide the possibility to exactly restore the original voxel set V from the polyhedron H . An efficient data structure to store the polyhedral surface is the cell list [10].

In the next Section we present the basic definitions used in this paper. In Section 3 we shortly present the algorithm for the construction of the convex hull and we give some new improvements to this algorithm concerning the reduction of the computation time. The main part of this paper follows in Section 4 where we present our method to construct polyhedral surfaces for non-convex digital objects through recursive repetitions of generating convex hulls of subsets of the given voxel set or subsets of the background voxels. In Section 5 we show some example images and experimental results. The paper is closed with a conclusion in Section 7 and a bibliography.

2 Basic Definitions

The algorithm presented here is based on the theory of abstract cell complexes (AC complexes) introduced to the field of image analysis by Kovalevsky [10]. Most of the basic notions of this theory relevant to the topic of polyhedral surfaces are gathered in the Appendix to [8].

Let V be a given set of voxels in a Cartesian three-dimensional space. The voxels of V are specified by their coordinates. Our aim is to construct the convex hull K of V and a modification H of K which represents the polyhedral surface of V . We consider the convex hull and the modified hull as abstract polyhedra according to the following definition:

Definition AP: An *abstract polyhedron* is a three-dimensional AC complex containing a single three-dimensional cell whose boundary is a two-dimensional combinatorial manifold without boundary. The two-dimensional cells (2-cells) of the polyhedron are its *faces*, the one-dimensional cells (1-cells) are its *edges* and the zero-dimensional cells (0-cells) are its *vertices* or points. [8]

¹Example: The faces of a cube (squares) can be subdivided into coplanar triangles. The surface area does not change, but the number of faces increases.

An abstract polyhedron is called a *geometric* one if coordinates are assigned to each of its vertices. We shall call an abstract geometric polyhedron an AG-polyhedron. Each face of an AG-polyhedron PG must be planar. This means that the coordinates of all 0-cells belonging to the boundary of a face F_i of PG must satisfy a linear equation $H_i(x, y, z) = 0$. If these coordinates are coordinates of some cells of a Cartesian AC complex A then we say that the polyhedron PG is embedded into A or that A contains the polyhedron PG .

Definition CP: An AG-polyhedron PG is called *convex* if the coordinates of each vertex of PG satisfy all the linear inequalities $H_i(x, y, z) \leq 0$ corresponding to all faces F_i of PG . The coefficients of the linear form $H_i(x, y, z)$ are the components of the outer normal of F_i . [8]

A cell c of the complex A containing the convex AG-polyhedron PG is said to *lie in* PG if the coordinates of c satisfy all the linear inequalities $H_i(x, y, z) \leq 0$ of all faces F_i of PG .

Definition CH: The *convex hull* of a finite set V of voxels is the smallest convex AG-polyhedron PG containing all voxels of the set V . "Smallest" means that there exists no convex AG-polyhedron different from PG which contains all voxels of V and whose all vertices are in PG . [8]

For the differentiation between voxel sets being convex or not, we need to define what a convex voxel set actually is.

Definition DCS: A *digital half-space* is the set of all voxels whose coordinates satisfy a linear inequality. A *digital convex subset* of the space is a non-empty intersection of digital half-spaces. [8]

3 Convex Hull

It is well known that every non-convex set can be considered as the sum or the difference of convex sets. We use this property of non-convex sets to extend our incremental convex hull computing method presented in [8] to construct an abstract polyhedron from a non-convex set of voxels by subtracting small convex hulls from an initial convex hull. This is motivated by the imagination of modelling the surface through pressing faces of the convex hull onto the voxel object.

3.1 Constructing the Convex Hull

The first step to build a non-convex abstract polyhedron consists in creating the convex hull of the given voxel object V as described in [8]. The algorithm for constructing the convex hull consists of two parts: in the first part a subset of vectors v pointing to voxels must be found which are candidates for the vertices of the convex hull. The coordinates of the candidates are saved in an array L . The second part constructs the convex hull of the set L .

From the point of view of AC complexes the given set V is the set of three-dimensional cells (3-cells) of a subcomplex M of a three-dimensional Cartesian AC complex A . The complex A represents the topological space in which our procedure is acting. It is reasonable to accept that M is homogeneously three-dimensional according to the following definition:

Definition HN: An n -dimensional AC complex A is called *homogeneously n -dimensional* if every k -dimensional cell of C with $k < n$ is incident to at least one n -cell of C . [9]

The problem of finding the vectors v can be defined as follows: A 0-cell is called a *convex* 0-cell iff it is incident to exactly one 3-cell of M (Figure 1). All 3-cells incident to at least one convex 0-cell are the candidate vectors v . The vectors v are stored in L .

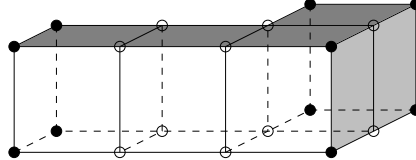


Figure 1: Four voxels and their convex 0-cells (depicted as black disks). Non-convex 0-cells are depicted as circles. The voxel in the center of the front row is not incident to any convex 0-cell and thus it is not a candidate vector.

As already mentioned, the second part of our algorithm is that of constructing the convex hull of the set L of the candidate vectors v found by the first part.

To build the convex hull of L we first create a simple convex polyhedron spanning four arbitrary non-coplanar voxels v of L . It is a tetrahedron. It will be extended step by step until it becomes the convex hull of L . We call it the *current polyhedron CP*.

The surface of the current polyhedron is represented with the data structure called the *two-dimensional cell list* [10]. The cell list of a two-dimensional complex consists in the general case of three sublists. The k th sublist contains all k -dimensional cells (k -cells), $k = 0, 1, 2$. The 0-cells are the vertices, the 1-cells are the edges, the 2-cells are the faces of the polyhedron. Each entry in the k th sublist corresponds to a k -cell c^k . The entry contains indices of all cells incident to c^k . The entry of a 0-cell contains also its coordinates.

The cell list according to this definition contains much redundancy, because it contains for a pair of two incident cells c^k and c^m both the reference from c^k to c^m and from c^m to c^k . The redundancy makes the computation faster, because cells incident to each other may be found immediately, without a search. However, for the exact reconstruction of the voxel set from the cell list the redundancy can be eliminated to make the encoding more economical.

The next step in constructing the convex hull is to extend the current polyhedron while adding more and more voxels, some of which become vertices of the convex hull. When the list L of the candidate vectors is exhausted, the current polyhedron becomes the convex hull of M . The extension procedure is based on the notion of visibility of faces which is defined as follows.

Definition VI: The face F of a convex polyhedron is *visible* from a cell c , if c lies in the outer open half-space bounded by the face F , i.e. if the scalar product (N, w) of the outer normal N of the face F and the vector w pointing from a point Q in F to c , is positive. If the scalar product is negative then F is said to be *invisible* from c . If the scalar product is equal to zero then F is said to be *coplanar* with c . [8]

To extend the current polyhedron the algorithm processes one voxel after another. For any voxel v it computes the visibility of the faces of the polyhedron from v . Consider first the simpler case when there are no faces of the current polyhedron,

which are coplanar with v . The algorithm labels each face of the current polyhedron as being visible from v or not. If the set of visible faces is empty, then the voxel v is located inside the polyhedron and may be discarded. If one or more faces are visible, then the polyhedron is extended by the voxel v and some new faces. Each new face connects v with one edge of the boundary of the set of visible faces. A new face is a triangle having v as its vertex and one of the edges of the said boundary as its base. All triangles are included into the cell list of the current polyhedron while all visible faces are removed. Also each edge incident to two visible faces and each vertex incident only to visible faces is removed.

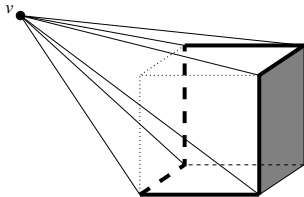


Figure 2: The current polyhedron (a cube) being extended by the voxel v as a new vertex.

In Figure 2 the boundary of the visible subset is shown by bold lines (solid or dashed). The edges shown by dotted lines must be removed together with the three faces visible from v . The remaining faces, edges and vertices keep unchanged. The algorithm repeats this procedure for all voxels in L .

Consider now the problem of coplanar faces. There are three variants to treat a face coplanar with a new voxel v . The first variant treats coplanar faces as visible ones. In such a situation the algorithm removes all coplanar faces and replaces them by new triangles connecting the boundary edges with the voxel v . As a result the number of coplanar triangles is rather small (see Figure 3).

Secondly, we can treat coplanar faces as invisible ones. In this situation the algorithm keeps the coplanar faces and creates more and more triangles. For a large number of coplanar voxels the number of faces created by this method is large, too (see Figure 4).

Both alternatives, treating coplanar faces as visible and invisible respectively, require a subsequent step of merging coplanar triangles since our aim is to produce a polyhedral approximation of the objects surface, but not necessarily a triangulation. If a triangulation is desired, the step of merging coplanar faces must be omitted.

The third option to handle coplanar faces is a little bit more sophisticated and treats them neither as visible nor as invisible ones. We call it *extension method* since the algorithm has to extend a coplanar face towards the new voxel v . This procedure is similar to the construction of the two-dimensional convex hull of v and the coplanar face. The result of this method to treat a coplanar face is a convex face with more than three edges. The number of generated faces then is as small as possible (see Figure 5).

In the following we want to present two examples processed with all three variants to compare computation time and the number of faces generated. The first example is a cube whose sides are non-parallel to the coordinate planes. Figures 3 and 4 show the results for the first two variants before merging coplanar triangles. After the merging step all three results are identical (see Figure 5). The second example is a digital ball (see Figures 6 to 8). The difference between the first two variants is not as big as in the cube example, but the number of faces generated is significantly greater than the number of faces generated with the extension method and the number of faces after merging respectively.

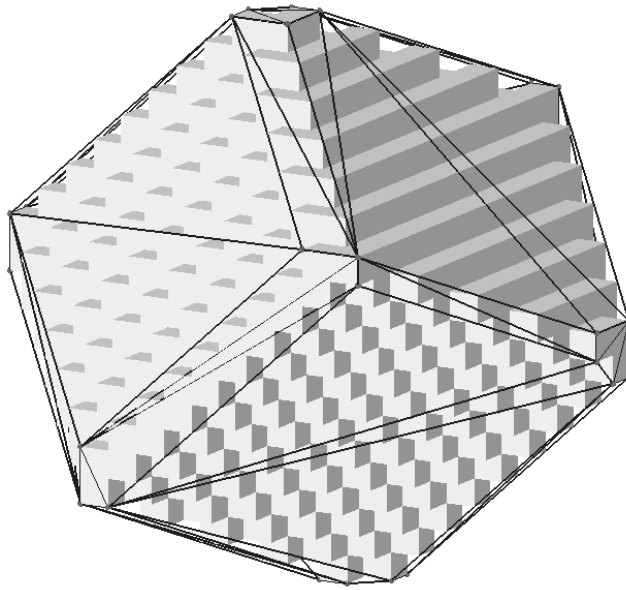


Figure 3: Convex hull of a cube as a triangulation. Coplanar faces have been treated as visible ones.

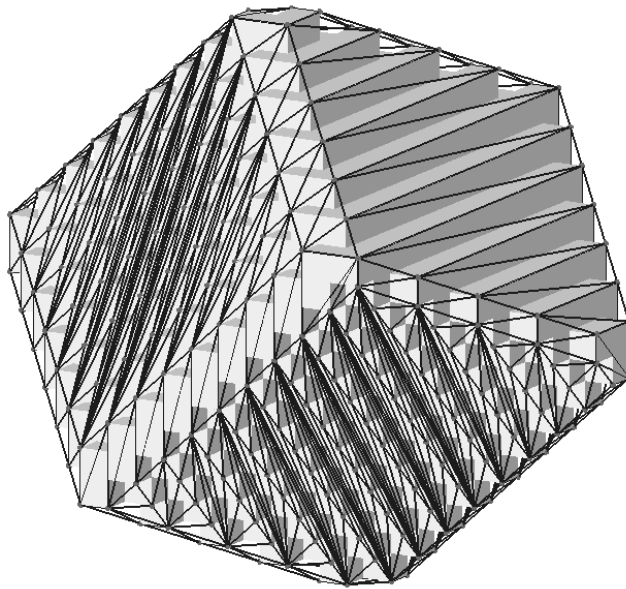


Figure 4: Convex hull of a cube as a triangulation. Coplanar faces have been treated as invisible ones.

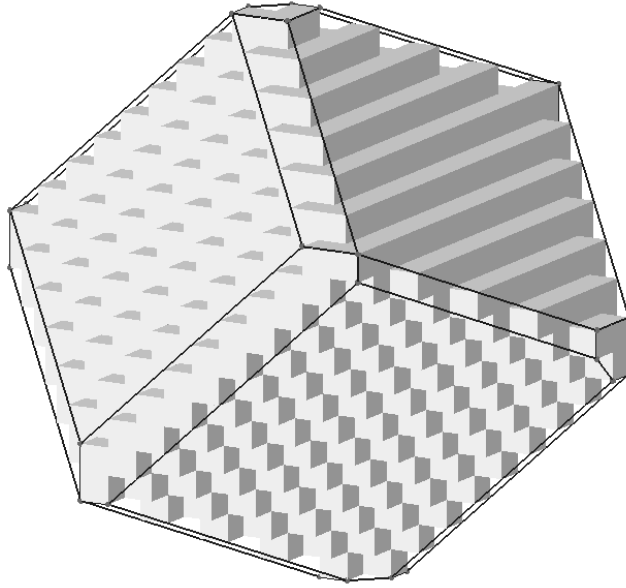


Figure 5: Convex hull of a cube. Coplanar faces have been treated with the extension method. After merging coplanar faces of the triangulated convex hull the result of all three methods looks the same.

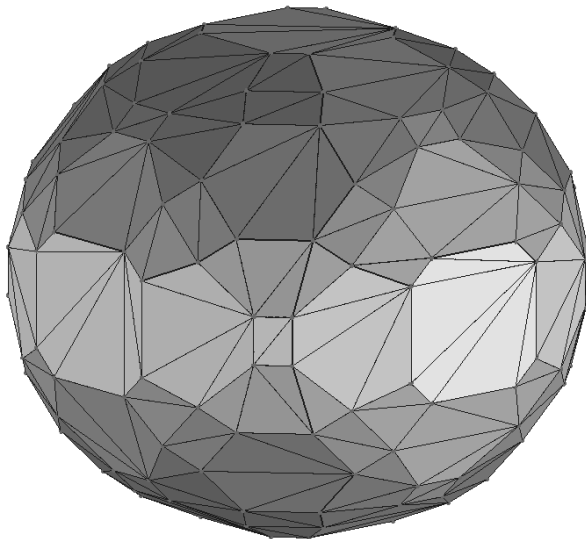


Figure 6: Convex hull of a digital ball as a triangulation. Coplanar faces have been treated as visible ones.

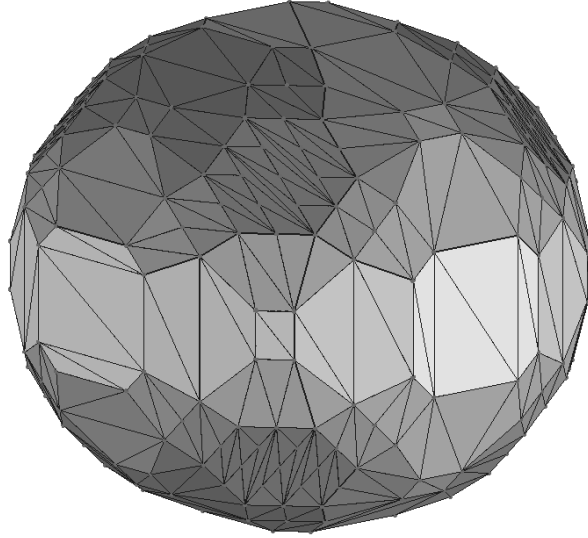


Figure 7: Convex hull of a digital ball as a triangulation. Coplanar faces have been treated as invisible ones.

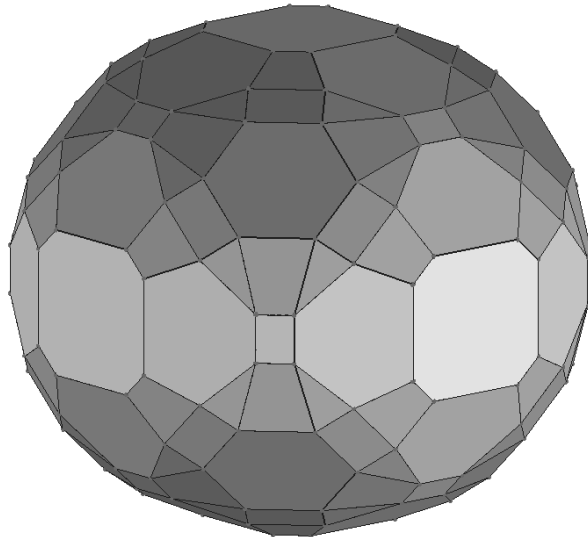


Figure 8: Convex hull of a digital ball. Coplanar faces have been treated with the extension method.

Table 1: Comparison of the computation time of the three variants of treating coplanar faces.

Object	Computation time		
	Triangulation (coplanar = visible)	Triangulation (coplanar = invisible)	Extension Method
Ball ($r = 12$)	0,110s	0,210s	0,080s
Ball ($r = 16$)	1,812s	2,293s	0,971s
Ball ($r = 20$)	1,932s	5,357s	1,462s
Ball ($r = 24$)	4,937s	8,072s	3,435s
Cube ($a = 5$)	0,040s	0,130s	0,010s
Cube ($a = 10$)	0,180s	8,031s	0,170s

Table 2: Comparison of the numbers of faces created with the three variants of treating coplanar faces.

Object	Number of faces of the convex hull		
	Triangulation (coplanar = visible)	Triangulation (coplanar = invisible)	Extension Method
Ball ($r = 12$)	140	236	74
Ball ($r = 16$)	428	492	218
Ball ($r = 20$)	380	636	170
Ball ($r = 24$)	524	668	290
Cube ($a = 5$)	60	204	24
Cube ($a = 10$)	60	716	22

The examples show clearly that the best way to treat coplanar faces is to use the extension method since the number of faces created is as small as possible. The number of faces of the current polyhedron has a significant impact on the computation time (see Table 1) since the algorithm has to compute the visibility of every face for every new voxel v .

If a triangulation is desired the best way is to treat coplanar faces as visible ones since the number of new faces is smaller than the number of faces created with the method which interprets coplanar faces as invisible ones.

The procedure of adding new faces to the current polyhedron ends after processing all candidate vectors in L . With this step the convex hull is completely constructed and the first part of creating a polyhedral surface of the given voxel object V ends.

3.2 Reducing computation time through sorting vertex candidates

As already shown in the previous section, the runtime of the convex hull creation algorithm depends considerably on the method of treating coplanar faces. The computation time spent for the extension of coplanar faces is much less than the time saved by reducing the number of faces of the polyhedron.

Another runtime reduction can be achieved by a suitable selection of the vertices of the initial convex hull, i.e. the tetrahedron. The idea is to maximize the initial polyhedron and also the subsequent hulls to have many vertex candidates, which are no vertices of the convex hull, already located inside the polyhedron. For those

vertex candidates the algorithm has to compute the visibility only. Since they are located inside the current polyhedron, there is no need to extend it, which means that no new faces have to be created. It is easy to see that this method is promising only for objects whose convex hulls have few vertices related to the number of vertex candidates.

For the maximization of the initial hull we first need to have the list L of vertex candidates sorted lexicographically. This is always the case since L is created by systematically going through the cell complex to find the candidates. Hence the first vector \vec{v}_1 in L is the one with minimum x -coordinate and minimum y -coordinate amongst all vectors with this x -value and minimum z -coordinate amongst all with this y -value. We cannot take the absolute minimum values for all three coordinates since such a cell does not necessarily belong to the object. The first candidate in L is also the first one in our new sorted list L_s . The second candidate \vec{v}_2 is the vector with maximum x -coordinate and minimum y - and z -coordinates chosen the same way as for the first candidate. If there is one vector with minimum x -, y - or z -coordinate only then the next greater value has to be taken. For the third and the fourth candidate \vec{v}_3 and \vec{v}_4 we choose the ones with maximum y -coordinate and z -coordinate respectively. I.e. the four vertices of the tetrahedron are:

$$\vec{v}_1 = \begin{pmatrix} x_{min} \\ y_{min} \\ z_{min} \end{pmatrix}, \vec{v}_2 = \begin{pmatrix} x_{max} \\ y_{min} \\ z_{min} \end{pmatrix}, \vec{v}_3 = \begin{pmatrix} x_{min} \\ y_{max} \\ z_{min} \end{pmatrix}, \vec{v}_4 = \begin{pmatrix} x_{min} \\ y_{min} \\ z_{max} \end{pmatrix}$$

The next four vertex candidates in L_s are chosen in a similar way with maximum values in their coordinates, i.e.:

$$\vec{v}_5 = \begin{pmatrix} x_{max} \\ y_{max} \\ z_{max} \end{pmatrix}, \vec{v}_6 = \begin{pmatrix} x_{min} \\ y_{max} \\ z_{max} \end{pmatrix}, \vec{v}_7 = \begin{pmatrix} x_{max} \\ y_{min} \\ z_{max} \end{pmatrix}, \vec{v}_8 = \begin{pmatrix} x_{max} \\ y_{max} \\ z_{min} \end{pmatrix}$$

With these eight vectors the algorithm produces an octahedron which is extended with all other candidate vectors of L_s containing the remainder of all cells of L in the same order.

As already mentioned this method has no benefit for objects whose all candidates are vertices of the convex hull. Digital balls are examples for such objects. But the computation time for choosing eight elements from a sorted list is negligibly small in relation to the advantage of saving a lot of work spent for creating and removing faces. In the general case, especially when considering non-convex objects, the number of vertices of the convex hull is significantly smaller than the number of vertex candidates.

The object in Figure 9 has been created from the sorted list L_s . Coplanar faces have been treated as invisible faces. With this sorted list L_s the number of faces of the triangulated convex hull can be reduced from 716 to 622, which leads to a reduction of computation time from 8,031s to 6,209s.

The experiments in this section have shown that a significant reduction of the computation time can be achieved using the concept of treating coplanar faces with the extension method instead of treating them as visible ones [8]. The preparation step of sorting the vertex candidates and choosing minimum and maximum coordinate values does not change the methodology of the algorithm, but brings another runtime reduction of the implementation.

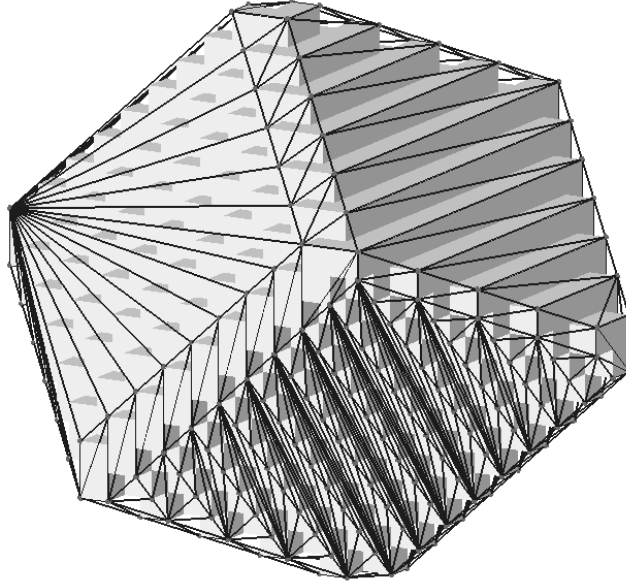


Figure 9: Triangulated convex hull of the cube created from the sorted list L_s . Coplanar faces have been treated as invisible ones (cf. Figure 4).

4 Non-convex Sets of Voxels

4.1 Finding Concavities

As already mentioned, the convex hull is a good means to encode the surface of a convex object. The convex hull of a convex set of voxels (according to Definition DCS) never contains voxels of the background. If the given voxel object is not convex, we have to find components of the set of background voxels included into the convex hull. These components can be cavities, concavities and tunnels [11].

Definition CO: A *concavity* is a component of background voxels inside the convex hull which intersects exactly one connected set of faces of the convex hull.

Due to this definition a concavity does not have to be a convex protrusion of the background, i.e. it does not have to be convex.

Definition CA: A *cavity* is a component of background voxels inside the convex hull, which does not intersect any face of the convex hull.

Definition TU: A *tunnel* is a component of background voxels inside the convex hull which intersects more than one connected set of faces of the convex hull. If the tunnel intersects exactly two connected sets it is called a *non-branched* tunnel. If the number of connected sets of faces of the convex hull which are intersected by the component is greater than two, the tunnel is called a *branched* one.

Here we only deal with concavities and cavities. The latter are a trivial problem and will be mentioned later on. Tunnels are part of future work.

We use the algorithm described in [12] to find the components of the set of background voxels inside the convex hull and classify them by the number of the

connected sets of faces of the convex hull which are intersected by the component.

To check whether a component of background voxels intersects a set of faces of the convex hull, we just have to compute the visibility of faces from the 0-cells of the background component. If every 0-cell of a component has no visible faces then the component is entirely located inside the hull and thus it is a cavity. If one or more 0-cells have visible faces or one or more 0-cells are coplanar with a set of faces then the component intersects the hull, i.e. it is a concavity or a tunnel (see Figure 10).

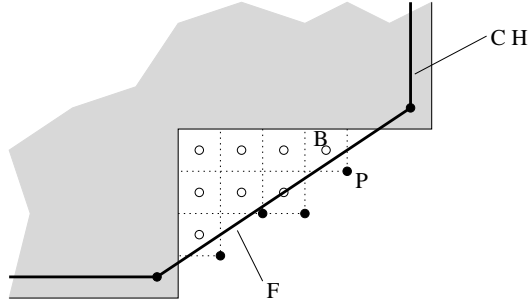


Figure 10: Convex hull CH of an object (gray shaded). Some of the 0-cells of the background voxel component are located outside CH and thus there exists a visible face.

The components are labeled and thus we can modify the convex hull by treating one component of background voxels after another.

4.2 Modification of the Convex Hull

After all components of background voxels inside the convex hull are found, we can modify the convex hull. As already mentioned, the convex hull is a convex polyhedron. After the first modification we can no longer speak of the convex hull, because it is no longer a convex polyhedron. Hence we call the modified hull *current polyhedron* again until it becomes a polyhedron with the desired properties.

To modify the current polyhedron we first need to know which faces are intersected by the current background voxel component. This can be determined by using again the notion of visibility. In the previous step we have labeled a component if its 0-cells have visible faces and now we label the faces which are visible from the 0-cells of the current background voxel component. This means that a face F becomes labeled if the following criteria are all satisfied:

1. The face F is visible from some of the 0-cells of the background voxel component or some 0-cells are coplanar with the face F .
2. If there is a 0-cell P outside H and a background voxel B inside or on H with $P \in Cl(B)$, then the projection of B onto the face F is inside the boundary of F .
3. The 0-cell P does not lie on the boundary of the face F .

We want to mention that the steps of labeling the background voxel components and labeling the corresponding faces can be merged together.

A topology preserving operation called "pressing-in" is applied to the set of labeled faces (see Figure 11).

Definition PR: *Pressing-in* towards a non-empty set of cells located inside a polyhedron H is a topology preserving operation which replaces a connected set S_1 of faces of H by a new connected set S_2 of faces in such a way that the boundaries of the sets S_1 and S_2 are identical. [13]

In the simplest case the set S_1 consists of one face only and thus it can be interpreted as the base of a pyramid which has the set S_2 as its sides. The apex of the pyramid (P in Figure 11) is located inside the polyhedron. In the general case the set S_1 consists of several faces and the destination of the pressing-in is not necessarily a single cell.

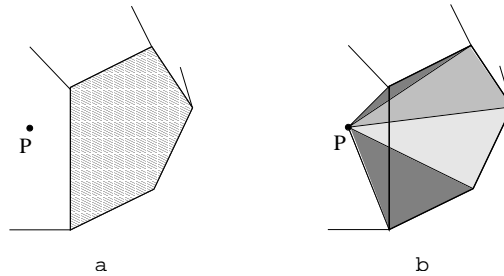


Figure 11: A cell P inside the polyhedron (a) and the resulting polyhedron after pressing-in (b).

We perform the pressing-in by constructing a polyhedron around the background voxel component. As mentioned in the Introduction, we want to apply our convex hull algorithm recursively to modify the convex hull of the voxel object. This means that we now create the convex hull of the background voxel component and modify it again and again until there are no object voxels outside the polyhedron and no background voxels inside it. To do so we have to combine the cell lists of the current polyhedron and that of the current polyhedron of the background voxel component. But this is not trivial. Definition PR implies that we can identify faces of both polyhedra, but this is not possible in the general case (see Figure 12a).

To avoid identifying faces of these two polyhedra, which do not have identical boundaries, we do not construct the convex hull of the background voxel component independently. A more precise approach consists in spanning the convex hull by starting with the labeled faces of the current polyhedron. This means that we span an initial polyhedron with this set of faces and a voxel of the background component, which lies on the inside of the set of labeled faces (Figure 12b). This initial polyhedron can be extended in the same way as the tetrahedron being the initial convex hull. The result is the convex hull of the set of labeled faces and the set of voxels of the background voxel component lying inside the polyhedron before applying the pressing-in operation (Figure 12c).

Another problem is that of the recognition of 0-cells outside a non-convex polyhedron, because faces of such a polyhedron can be visible from 0-cells inside a non-convex polyhedron. Therefore we need another method than the visibility approach to decide for each 0-cell whether it is located in the inside or not. An easy approach is to compute a ray from the current 0-cell to a point which is certainly outside the polyhedron (i.e. a point on the boundary of the space) and just to count the number of intersections of the ray with the polyhedron. If this number is odd then the current 0-cell lies inside. This method is of course only applicable for polyhedra with no self-intersections, but since we only deal with binary voxel images, self-intersections do not appear.

At this stage of the modification we have the current polyhedron and a second

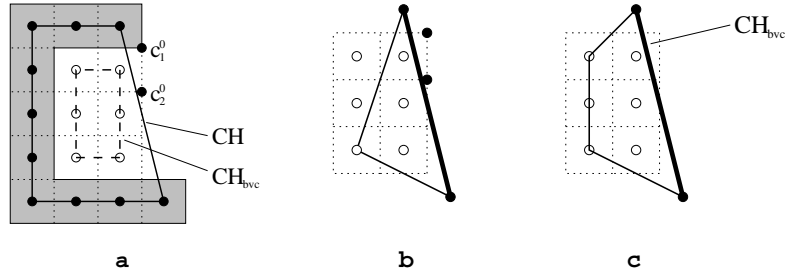


Figure 12: (a) Convex hull CH of a given set of voxels and the convex hull CH_{bvc} of the background voxel component. Two 0-cells c_1^0 and c_2^0 of the background voxel component are located outside CH . (b) Initial convex hull of the set of background voxels created from a fixed face (bold line). (c) Resulting convex hull CH_{bvc} of the background voxels.

smaller polyhedron which has a connected set of faces in common with the first one (see Figure 13a). Now we perform our idea of recursivity by interpreting the second polyhedron as the current polyhedron and we change the roles of object and background voxels. Now we can apply the same algorithm to the new current polyhedron and thus we search for object voxels inside this polyhedron for which we have to do a pressing-in. This leads to a protuberance to the outside of the first polyhedron.

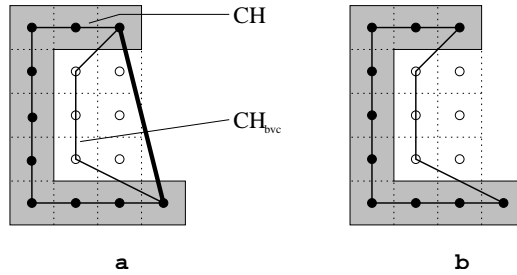


Figure 13: CH and CH_{bvc} with a common face (a) and the resulting polyhedron after the modification step (b).

The result of the algorithm is a polyhedron \hat{H} which separates object voxels from background voxels meaning that no object voxel lies outside \hat{H} and no background voxel lies inside \hat{H} (see Figure 13b). Voxels lying on \hat{H} , especially the vertices of \hat{H} , have to be marked in the cell list as being object or background voxels to preserve the possibility to reconstruct the object from the cell list of \hat{H} .

As already mentioned in Section 4.1, the algorithm can also deal with cavities, because it is a trivial problem. According to Definition CA cavities have no connection to the outer surface of the polyhedron and thus we can independently compute the convex hull of this background voxel component and modify it, if the cavity is a non-convex one. After applying the algorithm to the cavity we have to change the orientation of the normal vectors of the faces to ensure that they point to the outside of the surface of the voxel object, which means that they point to the inside of the cavity.

Given an object represented as subcomplex S of an AC complex A the modification algorithm can be summarized as follows:


```

create H=CH(S)
if H separates S and A\S {
    return H
} else {
    modify_polyhedron(H)
    return H
}

function modify_polyhedron(H) {
    find concavities inside H
    for each concavity S' {
        create CH(S')
        if CH(S') separates S' and A\S' {
            merge H and CH(S')
        } else {
            change roles of object and background voxels
            modify_polyhedron(CH(S'))
        }
    }
    return H
}

```

Algorithm 1. The modification algorithm.

It is easily seen from the algorithm pseudocode that the complexity of this method not only depends on the number of voxels but also on the number of concavities and their degree of non-convexity (see Section 4.3). A complete study on the complexity of the algorithm will be given in Section 6.

We want to mention that our algorithm has an important drawback. At this level of development it is not able to deal with a class of objects whose surfaces have a genus greater than 0 or whose background voxel components have a surface with a genus greater than 0, such as tori or mushrooms. This is justified by the fact that the pressing-in does not work for a set of faces composing a cycle. We are currently working on an improvement of our algorithm to solve this problem.

The correctness of the algorithm can be shown as follows: the first step to create a polyhedral surface \hat{H} which separates the voxels of a given object S and its complement $A \setminus S$ is to compute the convex hull $CH(S)$. The correctness of the convex hull algorithm has already been shown in [8]. When the convex hull is computed two cases can occur: (1) $CH(S)$ does not contain any voxel of the set $A \setminus S$ and thus it is convex and (2) $CH(S)$ contains voxels of $A \setminus S$. In the first case the convex hull is a polyhedron which separates the voxels since no voxel of S lies outside $CH(S)$ and no voxel of $A \setminus S$ lies inside and thus the algorithm stops here. In the second case the algorithm detects the class of the background voxel components and if these components are concavities it modifies the convex hull using the concept of pressing-in. This procedure moves all voxels of a concavity S' to the outside of the current polyhedron P through a substitution of faces of P with a new set of faces which lies completely inside P . This is done recursively for each concavity. In each step the algorithm deals with a current polyhedron P and a second smaller polyhedron P' lying completely inside P . The second polyhedron is smaller than the first one in the sense that it includes a fewer number of voxels. This means that by reducing the number of voxels in the current polyhedron the number of recursive steps is finite and the algorithm stops after a finite number of steps with a separating polyhedron \hat{P} as its result.

4.3 Degree of Non-Convexity

In Section 4.1 we already presented a classification of non-convex objects regarding the number of connected sets of faces of the convex hull into concavities, cavities and tunnels. The number of such components of background voxels inside the convex hull can be used as a measure to express the complexity of the object. But solely using this measure does not distinguish between components of different complexity. For a good classification of the complexity of the objects we need a second measure which expresses the intricacy of the background voxel components.

For this purpose we introduce the degree of non-convexity, defined as follows:

Definition NK: The *degree of non-convexity* $G_{nc}(S)$ of a non-convex subcomplex S of an AC complex A is defined recursively:

1. For every convex object S it holds:

$$G_{nc}(S) = 0 \quad (1)$$

2. The degree of non-convexity of an object S which is non-convex and contains exactly one component S' of background voxels, is greater by one than the degree of non-convexity of S' :

$$G_{nc}(S) = G_{nc}(S') + 1 \quad (2)$$

3. The degree of non-convexity of an object S which is non-convex and contains more than one components of background voxels, is greater by one than the maximum degree of non-convexity of its background voxel components:

$$G_{nc}(S) = \max_i \{G_{nc}(S'_i)\} + 1 \quad (3)$$

Table 3 presents some examples for objects with different degree of non-convexity. The examples are only 2-dimensional, but the concept is similar in the 3-dimensional case.

Since the degree of non-convexity is defined without being restricted to concavities, it can be applied to all three classes of background voxel components.

5 Examples

We have implemented the described algorithm and we have tested it with several objects of different size and complexity.

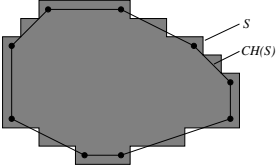
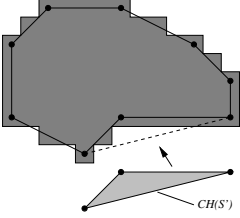
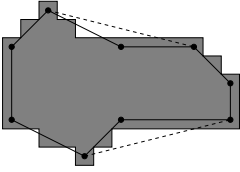
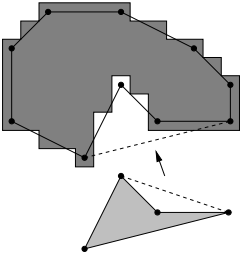
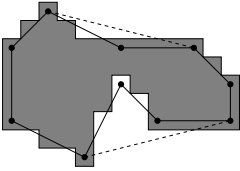
One simple example is presented below in Figure 14-17 to show how intermediate and final results look like.

Figure 14 (left) shows the voxel object. The voxels are interpreted as cells of an Euclidean complex and thus they are represented as small cubes. The object consists of 4949 voxels. On the right hand side of Figure 14 there is the convex hull CH surrounding the voxels, i.e. the centers of the cubes. The convex hull consists of 288 faces, 530 edges and 244 vertices.

The convex hull CH of the given voxel object includes one component of background voxels. The convex hull CH_{bvc} of these background voxels is shown on the left in Figure 15. It is created by starting with a set of 9 connected faces of CH . Finally it consists of 32 faces and it is surrounding 835 background voxels, but also 424 object voxels.

In this situation we apply our algorithm to the polyhedron shown in Figure 15 (left). We interpret the set of background voxels now as the set of object voxels

Table 3: Some examples for objects with different degree of non-convexity

object	degree of non-convexity
	<p>$G_{nc}(S) = 0$ since $CH(S)$ does not contain any components of background voxels.</p>
	<p>$G_{nc}(S) = 1$ since $CH(S)$ contains exactly one component S' of background voxels with $G_{nc}(S') = 0$.</p>
	<p>$G_{nc}(S) = 1$ since both concavities inside $CH(S)$ have a degree of non-convexity of 0.</p>
	<p>$G_{nc}(S) = 2$ since the concavity inside $CH(S)$ itself is non-convex, i.e. it has a degree of non-convexity of 1.</p>
	<p>$G_{nc}(S) = 2$ since the maximum degree of non-convexity of all concavities inside $CH(S)$ is 1.</p>

with the 424 object voxels as the set of background voxels inside the polyhedron. The background voxels are composing two components. The convex hulls of these components are shown on the right of Figure 15.

The left image in Figure 16 shows the polyhedral surface after applying only the first modification of the convex hull, which corresponds to the unification of CH and CH_{bvc} , where the set of labeled faces is removed. This polyhedron includes no voxels of the background, but two sets of object voxels are located on the outside of the polyhedron.

On the right hand side of Figure 16 the resulting polyhedron after all modification steps is presented. The resulting polyhedron consists of 300 faces, 566 edges and 268 vertices.

Figure 17 shows the triangulation of this object with the Marching Cubes method. It consists of 5184 triangles.

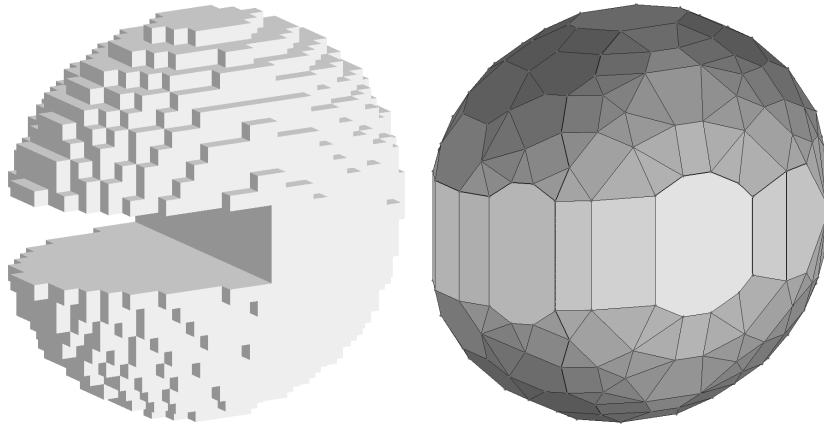


Figure 14: Example "pac-man". Left: The voxel object. Voxels are depicted as small cubes. Right: Convex hull of of the centers of the small cubes.

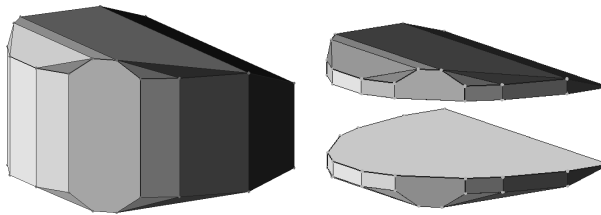


Figure 15: Left: Convex hull of the background voxels lying inside the convex hull of the given object. Right: Convex hulls of the object voxels lying inside the convex hull of the background voxels.

One of our primary goals concerning this algorithm is the efficiency of the encoding. Therefore we have compared the results of our algorithm with that of the Marching Cubes triangulation method, which is very often used in practical applications. As already mentioned in [8] we assume that 4.5 integers have to be saved for each triangle within this method. Our algorithm produces a non-redundant cell list containing 3 integers per vertex (its coordinates) and N_{F_i} integers per face, where N_{F_i} is the number of vertices of face i . Table 4 presents the obtained values for the memory requirements.

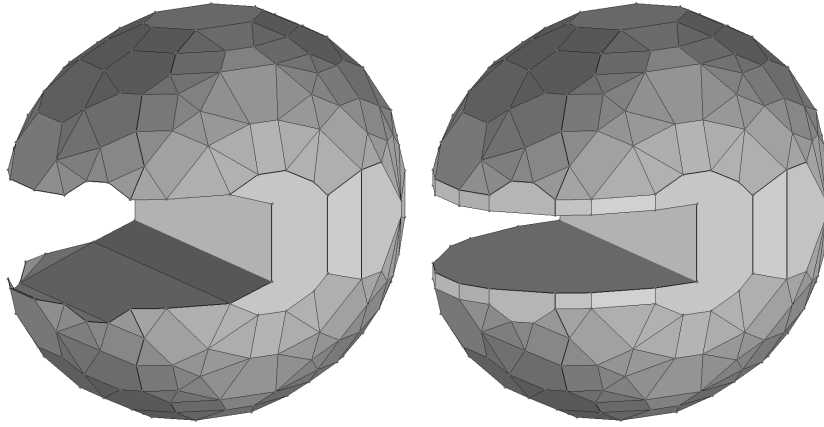


Figure 16: Left: Polyhedron after subtracting the convex hull of the background voxels only. Right: Resulting polyhedron after all modification steps.

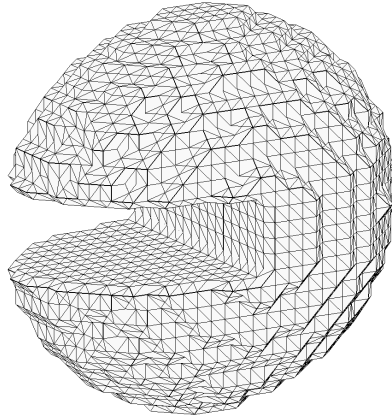


Figure 17: Marching Cubes triangulation of the object of Figure 14.

Table 4: Comparison of the memory requirements of the cell list and those of the triangulation for the example of Figure 14

	Faces	Vertices	Integers to be saved
MC-triangulation	5184	~2592	23328
Modified Convex Hull	300	268	1936

6 Complexity

6.1 Complexity of the convex hull algorithm

O'Rourke has shown that the complexity of an incremental convex hull algorithm in three dimensions is $O(v^2)$, where v denotes the number of vertices of the polyhedron [4]. Our method is also an incremental algorithm and all vertices of the convex hull are voxels of S and therefore the complexity of the algorithm can be considered as $O(n^2)$, where n denotes the number of voxels of S .

Intuitively one could presume that the algorithm has linear time complexity since it has to run through the list L of voxels only once when spanning the convex hull. But during the steps of the extension procedure the number of faces to be tested increases linearly and leads to the overall time complexity of $O(n^2)$.

6.2 Complexity of the modification algorithm

Let n be the number of voxels of the object S , k the number of concavities inside $CH(S)$ and $m = G_{nc}(S)$ the degree of non-convexity of S (see Section 4.3). The complexity of the modification algorithm is composed of the following parts: Inside the `modify_polyhedron` subroutine (see Algorithm 1) we have to find the concavities, which can be done going twice through the list of all voxels, i.e. in $O(n)$ time. For each of the k concavities we have to create the convex hull. As already mentioned this is of order $O(n^2)$. Merging polyhedra as well as changing roles of object and background voxels has a complexity of $O(n)$. Since `modify_polyhedron` is called recursively the complexity of the whole routine is $O(m \cdot k \cdot n^2)$, but in practice it can be observed that it behaves as one of quadratic complexity since m as well as k are much less than n .

7 Conclusion

In this paper we present a new algorithm for computing a polyhedral surface approximating a 3-dimensional digital object represented as a set of voxels. The resulting polyhedral surface is an abstract polyhedron which is a particular case of an abstract cell complex. The polyhedron is encoded by the non-redundant version of the well-known 2-dimensional cell list which is a good tool to save topological and geometric information efficiently and without redundancy. The cell list also provides the possibility to exactly reconstruct the voxel object.

Furthermore we present some improvements to our convex hull algorithm which lead to faster computation through sorting the list of vertex candidates. We also discuss the three variants to treat coplanar faces and we show that the extension method is the fastest one. Some examples for triangulated convex hulls are presented.

With the classification of background voxel components inside the convex hull and the degree of non-convexity we introduce a measure to distinguish between non-convex objects of different complexity.

The algorithm presented in this paper still needs further development. There are some drawbacks, especially the one mentioned in Section 4.2, concerning surfaces with a genus greater than 0. Also the labeling criteria are still under investigation.

The algorithm can be applied in a variety of tasks. Especially in the field of 3-dimensional image analysis and computer graphics it can be used to visualize voxel sets by polyhedra and to store large sets of voxels efficiently and without any loss of information.

Acknowledgement

A preliminary version of the paper was presented at the 12th International Workshop on Combinatorial Image Analysis, Buffalo, NY, USA, April 2008 [13].

References

- [1] W.E. Lorensen, H.E. Cline, Marching Cubes: A High-Resolution 3D Surface Construction Algorithm, *Computer Graphics* 21(4) (1987) 163-169.
- [2] P. Cignoni, C. Montani, R. Scopigno, DeWall: A Fast Divide & Conquer Delaunay Triangulation Algorithm in E^d , *Computer Aided Design* 30(5), Elsevier (1998) 333-341.
- [3] F.P. Preparata, M.I. Shamos, *Computational Geometry - An Introduction*, Springer-Verlag 1985.
- [4] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press 1994.
- [5] P.K. Agarwal, S. Suri, Surface Approximation and Geometric Partitions, *SIAM Journal on Computing* 27(4) (1998) 1016-1035.
- [6] H. Brönnimann, M.T. Goodrich, Almost Optimal Set Covers in Finite VC-Dimension, *Discrete and Computational Geometry* 14 (1995) 263-279.
- [7] G. Das, M.T. Goodrich, On the Complexity of Optimization Problems for 3-dimensional Convex Polyhedra and Decision Trees, *Computational Geometry: Theory and Applications* 8 (1997) 123-137.
- [8] V.A. Kovalevsky, H. Schulz, Convex Hulls in a 3-dimensional Space, in: R. Klette, J. Žunić (Eds.), *Combinatorial Image Analysis*, Lecture Notes in Computer Science, Vol.3322, Springer-Verlag, 2004, pp. 176-196.
- [9] V.A. Kovalevsky, A Topological Method of Surface Representation, in: G. Bertrand, M. Couprie, L. Perrotton (Eds.), *Discrete Geometry for Computer Imagery*, Lecture Notes in Computer Science, Vol.1568, Springer-Verlag, 1999, pp. 118-135.
- [10] V.A. Kovalevsky, Finite Topology as Applied to Image Analysis, *Computer Vision, Graphics and Image Processing* 45(2) (1989) 141-161.
- [11] S. Svensson, C. Arcelli, G. Sanniti di Baja, Characterising 3D Objects by Shape and Topology, in: I. Nyström, G. Sanniti di Baja, S. Svensson (Eds.), *Discrete Geometry for Computer Imagery*, Lecture Notes in Computer Science, Vol.2886, Springer-Verlag, 2003, pp. 124-133.
- [12] V.A. Kovalevsky, Algorithms in Digital Geometry Based on Cellular Topology, in: R. Klette, J. Žunić (Eds.), *Combinatorial Image Analysis*, Lecture Notes in Computer Science, Vol.3322, Springer-Verlag, 2004, pp. 366-393.
- [13] H. Schulz, Polyhedral Surface Approximation of Non-Convex Voxel Sets through the Modification of Convex Hulls, in: V.E. Brimkov, R.P. Barneva, H.A. Hauptman (Eds.), *Combinatorial Image Analysis*, Lecture Notes in Computer Science, Vol.4958, Springer-Verlag, 2008, pp. 38-50.