Kragujevac J. Math. 30 (2007) 327–342.

ALGORITHMS FOR INVESTIGATING OPTIMALITY OF CONE TRIANGULATION FOR A POLYHEDRON

Milica Stojanović and Milica Vučković

Faculty of Organizational Sciences, Jove Ilića, 11000 Beograd, Serbia
(e-mails: milicas@fon.bg.ac.yu, milica@fon.bg.ac.yu)

(Received October 25, 2006)

Abstract. The problem of finding minimal triangulation of a given polyhedra (dividing polyhedra into tetrahedra) is very actual now. It is known that cone triangulation for a polyhedron provides the smallest number of tetrahedra, or close to it. In earlier investigations when this triangulation was the optimal one, it was shown that conditions for vertices to be of the order five, six or for separated vertices of order four was only the necessary ones. It was shown that then if it exists the "separating circle" of order less then six, for two vertices of order six, cone triangulation is not the minimal one.

Here, test algorithms will be given, for the case when the given polyhedron has separating circle of order five or less.

1. INTRODUCTION

It is known how to divide any polygon with $n-3$ diagonals into $n-2$ triangles without gaps and overleaps. Such a division is called triangulation.

The generalization of this process to higher dimensions is also called triangulation. It divides polyhedron (polytope) into tetrahedra (simplices). Problem of triangulation in higher dimensions is much more complicated. It is impossible to triangulate

some nonconvex polyhedra [7] in three-dimensional space, and it is also proved that triangulations of the same polyhedron may lead to different numbers of tetrahedra [5], [8]. Considering the smallest and the largest number of tetrahedra in triangulation (the minimal and the maximal triangulation, respectively), the authors obtained values, which linearly, resp. squarely depend on the number of vertices. Some characteristics of triangulation in three-dimensional space are given by Chin, Fung, Wang [3], Develin [4] and Stojanović [9, 10, 11].

In this paper we shall consider convex polyhedra in which every 4 vertices are noncoplanar and all faces are triangular. Furthermore, all considered triangulations are face to face. The number of edges from the same vertex will be called the order or degree of the vertex.

In Section 2 previous results are summarized while in Section 3 method for finding separating circle with five or less edges is described. Section 4 is about graphs and their applications to this problem. Abstract data type (ADT) of graph is also presented [2, 6], including some elementary properties, and also two main data structures for representing graphs are given. In Sections 6 and 7 we will provide two graph algorithms. These algorithms work on the **graphs** of polyhedron representations.

We would like to thanks to dr Emil Molnr for improving the text and suggesting usage of the term separating circle.

2. PREVIOUS RESULTS

One of the triangulations, which gives a small number of tetrahedra, is the cone triangulation [8] described as follows.

One of the vertices is the common apex, which builds a tetrahedron with each triangular face of the polyhedron, except containing the vertex starting with.

By Eulers theorem, a polyhedron with n vertices has $2n-4$ faces if all of them are triangular. So, the number of tetrahedra in triangulation is $2n-10$ at most, since, for $n \geq 12$, each polyhedron has at least one vertex of order 6 or more. Sleator, Tarjan and Thurston in [8] considered some cases of "bad" polyhedra, which need a large

number of tetrahedra for triangulation. It is proved, using hyperbolic geometry, that the minimal number of triangulating tetrahedra is close to $2n - 10$. That value is tight for certain series of polyhedra, which exists for a sufficiently large n. Computer investigation of the equivalent problem of rotatory distance confirms, for $12 \leq n \leq 18$, that there exist polyhedra, with the smallest necessary number of tetrahedra equal to $2n - 10$. This was the reason why the authors gave a hypothesis that the same statement is true for any $n \geq 12$. To prove this hypothesis, it would be enough to check those cases where the cone triangulation of polyhedra gives the smallest number of tetrahedra, and to show how to improve that in other cases. With this aim, in [8] the authors gave a polyhedron example which has vertices of great order and for which there exists a triangulation better than the cone one. They also gave advices how to improve the method in this and some similar cases. Anyway, the polyhedra with vertices of great order give less than $2\,n-10$ tetrahedra in the cone triangulation, so, vertices of small order are considered in [9, 10, 11]. The obtained results are as follows:

Theorem 2.1. Let V be one of the vertices of a polyhedron P whose order is maximal. If the polyhedron P has a vertex of order 3 different and not connected with V, or a sequence of at least 2 vertices of order 4 connected with a chain, each of them not connected with V, then the cone triangulation of P with apex V will not give the smallest number of tetrahedra.

Remark When V is connected with a vertex at the end of chain the cone triangulation is not the minimal one whenever the chain contains at least 3 vertices.

Besides the order of vertices it is also necessary to consider the order of *separating circle* (now is better then in [11] called it *separating ring*). Let us define the following:

- A *circle of p vertices* of the polyhedron P is a p-sided closed polygon $A_1, A_2, ..., A_p$ where A_i ($i = 1, \ldots, p$) are different vertices of P and A_iA_{i+1} ($i = 1, \ldots, p$-1), A_pA_1 are edges of P.

- Let c be a circle on the polyhedron P, moreover M and N two vertices of P

different from A_i. If all paths on P with end points M and N pass through some of the vertices A_i then we say that a *separating circle* c separates M and N. We also say that M and N are on the different sides of c. If the circle c does not separate the vertices M and N, they are on the same side of the circle.

Theorem 2.2. If a polyhedron contains a circle of p vertices, which separates vertices A and B of order $\nu(A)$ and $\nu(B)$, where $\nu(A) \geq \nu(B) > p$, then the cone triangulation with apex A is not the minimal one.

From everything mentioned before, it is clear that candidates for the minimal triangulation with $2n - 10$ tetrahedra, are polyhedra with all vertices of order 5 or 6, occasionally some of order 4 which are not connected between themselves, and with separating circles of order six or more.

Condition for order of circles will be considered here, while conditions for order of vertices are considered in [12].

3. METHOD OF FINDING SEPARATING CIRCLE OF ORDER AT MOST FIVE

The first of all it is necessary to find the series of neighbor circles in the following way: Start with a vertex of the polyhedron and take all its neighbor vertices. They are connected to form a circle - the first one in series. The new neighbor vertices of those in the first circle form the second circle. The third circle is formed of new neighbors of the vertices of the second circle, and so on. The process is finished when unused neighbors of the vertices of the last circle form a chain, not a circle.

If in this series of neighbors circles all of them (except may be the first and the last one) are of order six or more, then we are searching for a circle of smaller order.

Let p, q, r denote consequitive neighbour circles of order at least six. Let a circle now be inserted with $A_1, A_2, ..., A_k$ (k = 3, 4,5) and edges A_iA_{i+1} (i = 1, ... , k-1), A_kA_1 . Then there are the following combinatorial possibilities:

a) Vertex A_1 lies on p, A_2, A_3 are on q;

b) Vertices A_1, A_2 lie on p, A_3, A_4 are on q;

c) Vertices A_1, A_3 lie on p, A_2, A_4 are on q;

d) Vertex A_1 lies on p, A_2, A_3 are on q and A_4 is on r;

e) Vertices A_1, A_2, A_3 lie on p, A_4, A_5 are on q;

f) Vertices A_1, A_3, A_4 lie on p, A_2, A_5 are on q;

g) Vertex A_1 lies on p, A_2, A_3, A_4, A_5 are on q;

h) Vertex A_1 lies on p, A_2, A_5 are on q, A_3, A_4 are on r.

In cases when vertices A_i are positioned on two of neighbor circles p and q, our inserted circle $A_1, ..., A_k$ have to separate other vertices of the circle p from those of circle q.

Although these cases have more subcases, if we require vertices to have order not greater then six, then the only possible case is *e)* when p is circle of order six. For example in case *a)* vertices A_2 and A_3 have to be connected with as follows: themselves, each of them with neighbor vertex of q, each with at least one vertex of another neighbor circle, and with all vertices of circle p (of order at least 6) - both with A_1 and one more vertex. That means that sum of orders of vertices A_2 and A_3 is at least 14 so, at least one of them have order greater than 6.

4. GRAPH METHODS

In the present paper, the graph structure is used as a model for polyhedron representation [1]. Graph provides more natural and consistent approach for this class of algorithms.

Viewed abstractly, a graph *G = (V, E)* consists of a set *V* of *vertices* and a set *E* of edges connecting the vertices in *V*. An edge *e = (u, v)* is a pair of two vertices *u* and *v*. The vertices *u* and *v* are called *endpoints* of the edge *(u, v)*.

An *abstract data type* (ADT) is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies *what* each operation makes, but it does not describe the way. In modern object-oriented program languages, (such as Java and C#) an ADT can be expressed by an interface, which is simply a list of method declarations.

An ADT is realized by a concrete data structure, which is modeled in object-oriented program languages by a *class*. A class defines the data stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify *how* the operations are performed. A Java class is said to *implement an interface* if its methods give life to all of those of the interface.

As an abstract data type, *a graph* is a positional container whose positions are its vertices and its edges. Hence, the graph ADT stores elements at either its edges or vertices (or both). A position in graph is always defined relatively, that is, in terms of its neighbors.

To abstract and unify the ways of storing elements in the various implementations of a graph, we introduce the concept of *position* in a graph, which formalizes the intuitive notion of "place" of an element relative to others in the graph.

A position itself is an abstract data type that supports a simple *element()* method, which returns to the element that is stored at this position. We also use specialized iterators for vertices and edges. An iterator is an enumeration whose traversal order can be guaranteed in some way. In order to simplify the presentation, we denote with v a vertex position and with e an edge position.

There are admittedly a lot of methods in the graph ADT. Some methods, however, is unavoidable to a certain extent, since graphs are rich structures. We need different methods for accessing and updating some positions in a graph, as well as dealing with the relationships that can exist between these positions. We divide the graph methods into three main categories: general methods, accessor methods and methods for updating and modifying graphs. We do not discuss error conditions that may occur. In addition, we take into consideration only methods for dealing with

undirected edges.

We begin by describing the fundamental methods for a graph, which ignore the direction of the edges. Each of the following methods returns global information about a graph G:

numVertices()	Return the number of vertices in G
numEdges()	Return the number of edges in G
vertices()	Return an iterator of the vertices of G
edges()	Return an iterator of the edges of G

The following accessor methods take vertex and edge positions as arguments:

degree(v)	Return the degree of v.
adjacentVertices(v)	Return an iterator of the vertices adjacent to v.
incidentEdges(v)	Return an iterator of the edges incident upon v.
opposite(v,e)	Return the endpoint of edge e distinct from v.
areAdjacent(v,w)	Return whether vertices v and w are adjacent.

We can also allow for update methods that add or delete edges and vertices:

insertEdge(v,w)	Insert and return an undirected edge between vertices v and w
insertVertex(v)	Insert and return a new (isolated) numbering vertex v storing the object o at this position
removeVertex(v)	Remove vertex v and all its incident edges
removeEdge(e)	Remove edge e

In order to perform graph algorithms in a computer, we have to decide how to store the graph. There are several ways to realize the graph ADT with a concrete data structure. In this section, we discuss two popular approaches, usually referred to as the *adjacency list* structure and the *adjacency matrix*, [2, 6].

There is a fundamental difference between the *adjacency list* and the *adjacency matrix*. The adjacency list structure only store the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). This difference implies that, for a graph G with n vertices and m edges, an edge list or adjacency list representation uses $O(n + m)$ space, whereas an adjacency matrix representation uses $O(n^2)$ space.

5. ALGORITHM FOR FORMING NEIGHBOR CIRCLES

Input to the algorithm is an undirected graph G with n vertices and the specific starting vertex $s \in G$. This algorithm forms circles $C_0, C_1, \ldots, C_m$ and lists $L_0, L_1, \ldots, L_{m-1}$. Each C_i represents neighbor circle. Each list L_i store the edges whose connect pairs *(u,v) of vertices, where* $u \in C_i$ *and* $v \in C_{i+1}$.

1. We begin with initializing graph (circle) C_0 so it contains a specific vertex s
2. At each iteration, the algorithm forms a new graph (circle) C_{i+1} and a new list L_i. The process is repeated until the graph G becomes empty
3. For each vertex $w \in C_i$ we find its incident edges in the graph G. Algorithm for each discovered edge *(w,q)*
   i. insert sits endpoint q to graph C_{i+1}
   ii. inserts edge *(w,q)* to the list L_i
   iii. removes edge *(w,q)* from the graph G
4. Away, algorithm for each vertex $w \in C_{i+1}$ finds all vertices in G adjacent to vertex w; for each found vertex $v \in G$ adjacent to w we check if v in C_{i+1}. If vertex $v \in C_{i+1}$ then edge *(w,v)* is inserted in C_{i+1}.
5. When C_{i+1} is formed then all edges in C_{i+1} are removed from G.

We give the pseudo-code for this algorithm in Figure 1.

Algorithm for forming neighbor circles
Input: An undirected graph G with n vertices and a specific starting vertex $s \in G$
Output: Graphs $C_0, C_1, \ldots, C_m$ and lists $L_0, L_1, \ldots, L_{m-1}$

initialize circle graph C_0 to contain a specific vertex s;
i = 0;
while (G is not empty)
{ create circle graph C_{i+1} to initially be empty;

```
        initialize new empty list L_i;// list store edges which connect two adjacency
                                        graph C_i and C_{i+1}
        for each vertex w in C_i.vertices()
        {       for each edge e in G.incidentEdges(w)
                {       q = G.opposite(w,e);// returns the endpoint of edge e distinct
                                                from w
                        C_{i+1}.insertVertex(q); // Add vertex q to graph C_{i+1}
                        L_i.insertLast((w, q)); //edge (w,q) connects two vertices w and
                                                q, w ∈ C_i, q ∈ C_{i+1}
                        G.removeEdge((w,q)); // remove edge (w,q) from G
                }
        }
        for each w in C_{i+1}.vertices() // add edges to graph C_{i+1}
        {       for each v in G.adjacentVertices(w)
               {
                if (C_{i+1}.containsVertex(v)) then C_{i+1}.insertEdge((w, v));
                }
        }
        for each e in C_{i+1}.eges() {
                G.removeEdge(e); // remove edge e from G
        }
        i++;
} m=i-1;
writeLine("formed neighbor circles");
for ( i = 0; i<= m; i++)
{       writeLine("ordinal number of circle: 0, degree of circle: 1", i, C_i.degree());
        writeLine ("vertices of circle:");
        for each u in C_i.vertices() {
                write ( "0,",u);
        }
        writeLine ("edges of circle");
        for each edge (u,v) in C_i.edges() {
                write ("(0, 1)" u,v);
        }
        if ( (L_i is not empty) then
                writeln ("edges that connect two adjacent circles C_0 and C_1", i, i+1);
                for each edge (w,u) in L_i.elements() {
                        write ("( 0, 1)", w,u); i++;
                }
} return C_0, C_1, ..., C_m; L_0, L_1, ..., L_{m-1}
```

Figure 1. Pseudo-code for forming neighbor circles

Example. The undirected graph G with 14 vertices, shown below and vertex B_5

is used as algorithm's input.

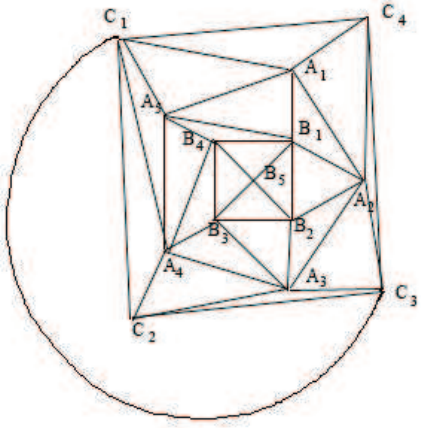

Figure 2. *Input to the algorithm*

The algorithm has generated the following output:

formed neighbor circles:

ordinal number of circle: 0 **degree of circle: 1**

vertices of circle: B_5

edges of circle:

edges that connect two adjacent circles C_0 and C_1:

$(B_5, B_1)(B_5, B_2)(B_5, B_3)(B_5, B_4)$

ordinal number of circle: 1 **degree of circle: 4**

vertices of circle: $B_1, B_2, B_3, B_4,$

edges of circle: $(B_1, B_2)(B_2, B_3)(B_3, B_4)(B_4, B_1)$

edges that connect two adjacent circles C_1 and C_2:

$(B_1, A_1)(B_1, A_2)(B_1, A_5)(B_2, A_2)(B_2, A_3)(B_3, A_3)$

$(B_3, A_4)(B_4, A_4)(B_4, A_5)$

ordinal number of circle: 2 **degree of circle: 5**

. . . . . .

ordinal number of circle: 3 **degree of circle: 4**

. . . . . .

6. ALGORITHM FOR FORMING INSERTED CIRCLES

Input to this algorithm are graphs $C_0, C_1, \ldots, C_m$ and lists $L_0, L_1, \ldots, L_{m-1}$, which are formed by preceedent algorithm (see previous Section). Each graph C_i represents neighbor circle. Each list L_i store the edges whose connect pairs (u, v) of vertices, where $u \in C_i$ and $v \in C_{i+1}$. A pseudo-code description of the main algorithm is given in Figure 3.

We now describe in detail our algorithm for forming inserted circles.

1. Algorithm begins with process traverses circles (graphs) $C_0, C_1, \ldots, C_m$ by considering order of each graph.

2. If algorithm finds the graph C_i of order six then for each vertex $v \in C_i$

   a) finds its two adjacency vertices x and y in C_i

   b) adds vertices *v, x, y* to temporary queue Q

   c) **calls** algorithm, (see pseudo-code in Figure 4.), for finding two vertices w and q in circle C_{i-1} neighbor to C_i, such that vertex w connects with vertex $x \in C_i$, as vertex q connects with vertex $y \in C_i$. Also vertex w mast be adjacent to vertex q. Algorithm returns queue D to contain five vertices of inserted circle which is forming.

   d) **calls** algorithm (see pseudo-code in Figure 5.), for forming inserted circle. The input in this algorithm are queue D and new empty graph P_k. Algorithm's output is formed inserted circle P_k of order five.

   e) Further main algorithm repeats the above steps 2.3 i 2.4 for circle C_{i+1} neighbor to C_i

3. The above process repeats while the circles C_i of order six exist

4. When the process is terminated, the algorithm returns inserted circles $P_0, P_1, \ldots, P_k$

Algorithm for forming inserted circles
Input: Graphs $C_0, C_1, \ldots, C_m$ and lists $L_0, L_1, \ldots, L_{m-1}$
Output: graphs (inserted circles) $P_0, P_1, \ldots, P_k$

```
i=0;
while (i < n)
{       findCircleOrderSix = false;
        j=i;
        while ((j <= n) && (not findCircleOrderSix)) //finding circle of the order
                                                        six
        {       if (C_j.numVertices()) == 6) then
                        findCircleOrderSix = true;
                else j++;
        {
        i = j;
        if (not findCircleOrderSix) then
                writeLine ("No more circles of the order six");
        else
                k=0; // ordinal number of inserted circle
                // for each vertex v finds its two adjacency vertices in C_i
                for each vertex v in C_i.vertices()
                {       Let Q be an initially empty queue;
                        Q.AddRear(v); // Add vertex v at the rear of the queue
                        for each vertex w in C_i.adjacentVertex(v)
                                Q.AddRear(w);
                        {
                        call FindTwoVerticesOnNeigbourCircle(C_{i-1}, C_i, L_{i-1}, Q, D)
                        if ( (D is not empty) && (D.size() == 5) ) then
                                initialize new graph (inserted circle) P_k;
                                call FormInsertedCircle (D, P_k);
                                k++;
                        call FindTwoVerticesOnNeigbourCircle(C_{i+1}, C_i, L_i, Q, D);
                        if ((D is not empty) && (D.size() == 5))) then
                                initialize new graph (inserted circle) P_k;
                                call FormInsertedCircle(D, P_k);
                                k++;
                }
                i++;
}
k= k-1;
return P_0, P_1, ..., P_k;
```

Figure 3. *Pseudo-code for forming inserted circles (main algorithm)*

Algorithm **FindTwoVerticesOnNeigbourCircle** // finding two vertices on the neighbor circle U of the circle P

Input: Graph U, graph P, list S and queue Q

Output: queue D

```
Let D be an initially empty queue;
If (Q is not empty) && (Q.size() == 3) then
D ← Q; // copy Q to D
//remove three neighbor vertices from the front of the queue Q
v = Q.RemoveFront();
x = Q.RemoveFront();
y = Q.RemoveFront();
Let T be an initially empty list;
Let G be an initially empty list;
findVertices = false;
// finding all vertices in circle U adjacent to P, which are connect with vertex x ∈ P,
as to with vertex y ∈ P

    for each edge (w,q) in S.elements()
{       if (w == x) then T.insertLast (q);
        if (q == x) then T.insertLast (w);
        if (w == y) then G.insertLast (q);
        if (q == y) then G.insertLast (w);
}
if ((T is not empty) && (G is not empty)) then
        for each vertex w in T. elements()
        {       // finding two adjacent vertices w and q in graph U
                for each vertex q in G.elements() {
                    if (U.areAdjacent(w,q)) then
                        if ((S.ContainsElement((v,w)) or (S.ContainsElement(v,q)) then
                        // found all vertices of inserted circle of the order five
                                    D.insertRear(w);
                                    D.insertRear(q);
                                    findVertices =true;
                                    break;
                }
                if (findVertices) then break;
        }
return D
```

Figure 4. *Pseudo-code for finding two vertices on the neighbor circle*

Algorithm **FormInsertedCircle** // forming inserted circle
Input: queue D and empty graph M_k
Output: formed graph M_k

// remove vertices from the front of the queue D
v = *D.removeFront();*
x = *D.removeFront();*
y = *D.removeFront();*
w = *D.removeFront();*
q = *D.removeFront();*

// Insert vertices into graph P_k
$P_k.insertVertex(v)$; $P_k.insertVertex(x)$; $P_k.insertVertex(y)$; $P_k.insertVertex(w)$; $P_k.insertVertex(q)$;

// insert edges to P_k connecting pairs of vertices
$P_k.insertEdge(v, x)$; $P_k.insertEdge(v, y)$; $P_k.insertEdge(x, w)$; $P_k.insertEdge(y, q)$; $P_k.insertEdge(w, q)$;

writeLine("ordinal number of inserted circle: 0", k);

writeLine ("edges of inserted circle");
for each edge *(u,w)* in $P_k.edges()$ **{**
write ("(0, 1)" *u,w*);
}
return P_k;

Figure 5. *Pseudo-code for forming inserted circle*

Example. Input to this algorithm is polyhedron shown in Figure 6., with neighbor circles A_{01}; A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}; A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}; A_{31} and edges that connect two adjacent circles: $A_{01}A_{11}$, $A_{01}A_{12}$, $A_{01}A_{13}$, $A_{01}A_{14}$, $A_{01}A_{15}$, $A_{01}A_{16}$; $A_{11}A_{21}$, $A_{12}A_{21}$, $A_{12}A_{22}$, $A_{12}A_{23}$, $A_{13}A_{23}$, $A_{13}A_{24}$, $A_{14}A_{24}$, $A_{14}A_{25}$, $A_{14}A_{26}$, $A_{15}A_{26}$, $A_{16}A_{26}$, $A_{16}A_{21}$; $A_{21}A_{31}$, $A_{22}A_{31}$, $A_{23}A_{31}$, $A_{24}A_{31}$, $A_{25}A_{31}$, $A_{26}A_{31}$.

The algorithm is generates the following output:

ordinal number of inserted circle: 0

edges of inserted circle:

$(A_{13}, A_{14})(A_{13}, A_{12})(A_{14}, A_{26})(A_{12}, A_{21})(A_{26}, A_{21})$

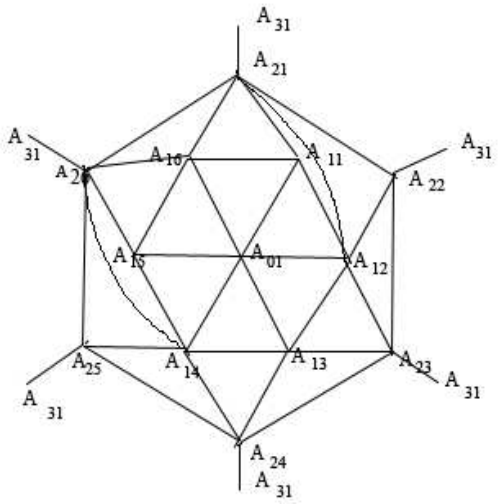

Figure 6. *Input to the algorithm*

References

[1] B. G. Baumgart, *A Polyhedron Representation for Computer Vision*, In Proceedings of the 1975 National Computer Conference. AFIPS Conference Proceedings, vol. **44**. AFIPS Press, Reston, Va. (1975).

[2] M. Goodrich, R. Tamassia, *Data Structures and Algorithms in Java*, Second Edition. John Wiley & Sons (2001).

[3] F. Y. L. Chin, S. P. Y. Fung, C. A. Wang, *Approximation for minimum triangulations of simplicial convex 3-polytopes*, Discrete Comput. Geom., **26**, No.4 (2001), 499–511.

[4] M. Develin, *Maximal triangulations of a regular prism*, J. Comb. Theory, Ser. A **106**, No. 1 (2004), 159–164.

[5] H. Edelsbrunner, F. P. Preparata, D. B. West, *Tetrahedrizing point sets in three dimensions*, J. Simbolic Computation, **10** (1990), 335–347.

[6] J. McConnell, *Analysis of Algorithms: An Active Learning Approach*, Jones and Bartlett Publishers (2001).

[7] J. Ruppert, R. Seidel, *On the difficulty of triangulating three - dimensional non-convex polyhedra*, Discrete Comput. Geom., **7** (1992), 227–253.

[8] D. D. Sleator, R. E. Tarjan, W. P. Thurston, *Rotatory distance, triangulations, and hyperbolic geometry*, J. of the Am. Math. Soc., Vol. **1**, No 3. (July 1988).

[9] M. Stojanović, *Triangulation of convex polyhedra by small number of tetrahedra*, Proceedings of the 10^{th} CONGRESS OF YUGOSLAV MATHEMATICIANS, Belgrade (21-24.01.2001), 207–210.

[10] M. Stojanović, *Algorithms for triangulating polyhedra with a small number of tetrahedra*, Mat. Vesnik, **57** (2005), 1–9.

[11] M. Stojanović, *Triangulations of some cases of polyhedra with a Small Number of tetrahedral*, Kragujevac J. Math., **31** (2007). (!!!!)

[12] M. Stojanović, M. Vučković *Algorithms for finding orders of the vertices of a given polyhedra and incidence structures*, manuscript.