An Algorithm Recreating 3D Triangle Mesh Faces from Its Edges

Marek Zabran*
Supervised by: Libor Vasa†

Department of Computer Science and Engineering
University of West Bohemia
Pilsen / Czechia

Abstract

3D triangle meshes are usually represented by a set of points with geometrical coordinates and a set of faces represented by triplets of these points. Multiple algorithms reconstructing a set of faces from a set of points and a set of edges were proposed in the literature, however, none of these can effectively reconstruct the set of faces using only the set of edges.

In this paper, such an algorithm is presented, which recreates a set of non-oriented triangle faces from only the set of its edges. The input is expected to be a closed 3D edge-manifold triangle mesh of any genus. The algorithm is simple, purely topological and runs in $O(n)$.

We present several practical examples demonstrating that it is capable of reconstructing faces even from fairly large input data, as well as input data that is prone to errors in reconstruction due to a high occurrence of possible inner faces.

Keywords: Graph Processing, Topological Mesh, Inner Face Removal

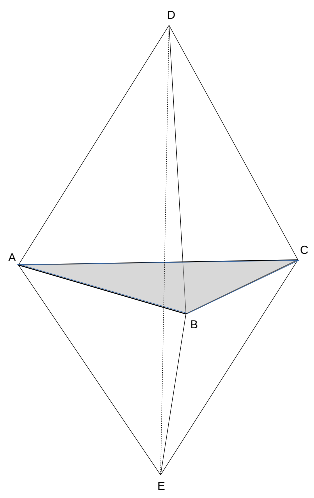

Figure 1: Two connected tetrahedra forming a hexahedron. Gray triangle indicates the inner face. DE edge is only for illustration. Double Tetrahedron mesh.

1 Introduction

A conventional triangle mesh [2] is a type of polygon mesh [2]. It comprises of a set of points given by geometric coordinates and a set of triangle faces represented by triplets of these points.

However, should these faces not be clearly specified, for example should they be represented by only a sketch (wireframe), a problem arises. In such case, it is possible to detect individual edges, but it is generally not clear which edges form a face. To find these faces using only the set of edges requires an algorithm which uses only topologic information, for geometric information might not be available.

1.1 Inner face

This may sound like a trivial problem that is solvable by generating all possible triangles. Unfortunately, the number of these triangles is generally bigger than the number of actual faces, due to the possible presence of the so-called *inner faces*.

An inner face (fig. 1), also called *internal face*, *interior face* or *pseudo face*, is a special false face which can be created, but which does not exist physically and therefore should not be included in the set of faces. It may only be formed by more than trihedral vertices [8], i.e. vertices of degree higher than 3.

1.2 Contribution

The algorithm presented in this paper should be able to exactly reconstruct all faces from a set of edges of any closed 3D edge-manifold triangle mesh, albeit with loss of face orientation. It may be also used to solve the dual problem, which is inner face removal. Many algorithms to solve inner face removal problem does exist, but their time complexities are mostly $O(n^2)$ or higher, whereas the complexity of the presented algorithm is $O(n)$.

Possible applications of this algorithm include mesh compression, damaged mesh reconstruction, inner face re-

*zabran@students.zcu.cz
†lvasa@kiv.zcu.cz

moval and possibly mesh generation from line drawing.

1.3 Structure

In Section 1, the tackled problem and terminology is specified. In Section 2, known algorithms solving similar problems are discussed and an alternative algorithm based on planarization is detailed. In Section 3, a new algorithm is proposed, fully described and a simple example of its usage is provided. In Section 4, testing of the new algorithm and results of these tests are presented.

1.4 Terminology

A *vertex* is a point in 3D space. Throughout this paper, its geometrical information is ignored and only it's topological information is relevant.

An *edge* always connects two vertices and represents a border between two faces.

A *triangle* is a cycle of three edges and three vertices. It may also be considered a face loop [8].

A *mesh* in this paper means a closed 3D edge-manifold triangle mesh, or simply any triangle mesh where every edge is incident with exactly two of its triangles. Note that this requirement is slightly less strict than the usual definition of a 2-manifold, which commonly also requires each vertex to be only incident with a single fan of triangles. For the purposes of our algorithm, this condition is not necessary.

A (proper) *face* is a specific triangle which is part of a mesh, i. e. part of the surface.

An *inner face* is a specific triangle which cannot be part of a correct mesh due to the rule encompassing our meshes that is stated two paragraphs above. Note that all proper faces are triangles, yet not all triangles are necessarily proper faces, since some triangles might be inner faces.

A vertex is *trihedral* when it connects three edges and thus also three faces. The count of edges it connects is its degree (hedrality).

2 Related work

A common case of mesh reconstruction is reconstruction from a 2D line drawing. In such case, geometric information, which involves mainly vertex coordinates and the slope between edges, may be obtained (either 2D or 3D through the process of inflation) and used for face detection [8]. As a polygon mesh is generally not a triangle mesh, faces in such case consist of more vertices and are detected as shortest planar cycles.

A good example of such reconstruction algorithm is a popular approach based on Dijkstra's algorithm, or a more refined algorithm proposed by Varley and Company [8], [7]. In these algorithms, each non-oriented edge is divided into two oriented edges and all of them are to be concatenated to make faces. Unlike other similar algorithms, these are simpler and are not of exponential complexity, but fail in some more complex cases by merging faces together or being unable to build a face if two of its different edges of opposite orientation have already been used. It also does not possess any effective mechanism protecting it from inner faces and runs in $O(n^5)$.

In our case, the mesh consists of triangles and a triangle is always planar. This makes the problem considerably easier, simplifying it to the inner face removal problem, but it also makes using the above mentioned algorithms completely unnecessary. Not only are they slow, meant for a different kind of mesh and using geometrical information we do not want to use, but they also provide no solution to avoid creation of inner faces other than prioritizing trihedral vertices. Surely, triangles surrounding a trihedral vertex have to be faces[1], but a mesh may not contain any trihedral vertex at all in the first place[2].

Proper faces of a mesh could also be found using graph planarization. Planarization [5] is a process in which a general graph is equipped with 2D coordinates associated with graph vertices, such that no edges cross. Tutte's algorithm or any other algorithm in [10] may be used to find such planarization if it exists by ad hoc generating geometrical coordinates of each vertex. In the planarized generated graph, all elemental triangles[3] are faces, which means the mesh can be generated using this approach in $O(n^2)$ at worst or about $O(n \cdot log(n))$ at best.

3 The proposed algorithm

This above mentioned planarization algorithm certainly works, yet it actually transforms a purely topological problem to a geometric one and then solves it to obtain a topological result. From this point of view, the planarization based algorithm seems unnecessarily complicated and makes one wonder, whether it is really necessary to use such complicated approach and whether it would not be possible to find an algorithm solving the problem purely topologically. Due to many observations and experiments, such an algorithm has been found.

In the Section 1.4, simple rules for vertices, edges and faces have been defined. The most important elements of this new algorithm originate from these rules. Other than that, the main philosophy of [8], the growing crystal theme, has been adapted in a form of a dynamic creation of faces and elimination of inner faces at the same time. This ensures successive elimination of triangles from simpler to harder cases, similar to solving certain puzzles, such as sudoku.

There are three rules our algorithm is based on:

[1]Note: Proof in Section 3.1.

[2]Note: Examples provided in section 4.1.

[3]Triangles which contain no other vertex inside and the border triangle.

3.1 Rule 1

A triangle with at least one trihedral vertex is always a face.

This rule has been already used in the sketch based algorithms. Sadly, same as in the case of these algorithms, should the mesh contain no trihedral vertex (which can and does happen often), this rule does not help at all.

Evidence: In (our) mesh the least number of faces a vertex can connect is three[4]. So, should there be only three triangles connected to a vertex, all of them have to be faces. However, this does not mean there can be only three faces connected to a vertex, there can be arbitrarily many.

3.2 Rule 2

Should an edge be present in exactly two triangles, both of these triangles are faces.

This is so far the most capable and important rule we have found, as it can be used repeatedly every time the set of triangles is reduced. Benefits of this rule are so significant that the other rules can be completely ignored and the algorithm will not be affected[5].

Evidence: Each edge separates exactly two faces. Therefore, should an edge appear in exactly two triangles, both of them have to be faces. Should an edge appear in less than two triangles, it indicates that the algorithm has failed.

This rule allows marking some triangles as inner faces (triangles incident with edges where two other triangles were already marked as proper faces) and removing them from further processing. Subsequently, the rule can be used as long as at least one edge appearing in exactly two triangles exists in the mesh. For such situation not to happen, *every* edge would have to be part of an inner face.

Such mesh can be created e. g. from mesh in Fig. 1 by adding new edge between vertices D and E, but it would not be manifold. Manifold triangle mesh where all edges are part of an inner face should not exist, for to add a new inner face to a mesh with maximal inner faces, adding a new vertex is necessary, and to add this vertex, three new edges must be added as well, which will not be part of an inner face. Unfortunately, it is hard to find rigorous proof for this.

3.3 Rule 3

The fewer edges of a triangle appear in other triangles, the lower is the chance of this triangle being an inner face.

This is only a backup rule proposed for the eventuality of the second rule failing. So far, this never happened and it is likely impossible. Also, as shown in Section 3.4, Rule 3 can be only used in sequence with Rule 2.

[4]Otherwise it would not be a closed 3D edge-manifold triangle mesh.

[5]Without Rule 1, algorithm appears to be about 3-5 % slower. Rule 3 is only a backup rule and, so far, was not used.

Initially, we assumed that the edges appearing in proper faces always have to appear in fewer other triangles, than edges of an inner face, because edges of an inner face have to appear in more triangles, as they appear in both inner and proper faces. This is, however, not true in very dense meshes, such as Monster type meshes (fig. 9), where, in central areas, triangles appear such that all of their edges are in at least one inner face and thus in these areas proper faces and some inner faces may have the same degree[6]. This rule would not fail, should the mesh be not reconstructed randomly, but sequentially from the periphery to the center, which would require a more complex implementation[7].

Should the existence of a mesh, where degree[6] of every edge is higher then 2, be proven negative[8], the third rule should be completely ignored as the second rule is sufficient. Otherwise, the third rule can by used together with the second rule to make such meshes solvable[9], but in such case, the mesh able to make the algorithm fail could definitely still exist.

3.4 Examples

Let us show an example of using these rules on (fig. 1):

The set of edges contains: AB, BC, CA, AD, BD, CD, AE, BE, CE.

The set of triangles contains: ABC (inner face), ABD, BCD, CAD, ABE, BCE, CAE.

The set of vertices[10] contains: A(4), B(4), C(4), D(3), E(3).

Now, using only one of the rules for this example:

Using Rule 1:

ABC is not generated, because it contains no trihedral vortex (A, B and C are all tetrahedral).

Other faces are generated, because D and E are trihedral.

Using Rule 2:

AB appears in ABC, ABD and ABE.

BC appears in ABC, BCD and BCE.

CA appears in ABC, CAD and CAE.

AD appears only in ABD and CAD: ABD and CAD are newly generated.

BD appears only in ABD and BCD: BCD is newly generated.

CD appears only in BCD and CAD.

AE appears only in ABE and CAE: ABE and CAE are newly generated.

BE appears only in ABE and BCE: BCE is newly generated.

CE appears only in BCE and CAE.

Using Rule 3 step by step:

[6]Note: Explained in Section 3.5.

[7]Luckily, second rule does this automatically without other implementation requirements.

[8]Discussed in last paragraph of Section 3.2.

[9]It creates one face, so that second rule can be used once more.

[10]Note: The attached number is vertex degree.

Edge set[11]: AB(3, 2), BC(3, 2), CA(3, 2), AD(2, 2), BD(2, 2), CD(2, 2), AE(2, 2), BE(2, 2), CE(2, 2).

Triangle set[12]: ABC(9), ABD(7), BCD(7), CAD(7), ABE(7), BCE(7), CAE(7).

–

ABD has the lowest degree and is generated, edges are used.

Edge set: AB(2, 1), BC(3, 2), CA(3, 2), AD(1, 1), BD(1, 1), CD(2, 2), AE(2, 2), BE(2, 2), CE(2, 2).

Triangle set: ABC(8), BCD(6), CAD(6), ABE(6), BCE(7), CAE(7).

–

BCD has the lowest degree and is generated, edges are used. BD has been used twice and is removed.

Edge set: AB(2, 1), BC(2, 1), CA(3, 2), AD(1, 1), CD(1, 1), AE(2, 2), BE(2, 2), CE(2, 2).

Triangle set: ABC(7), CAD(5), ABE(6), BCE(6), CAE(7).

–

CAD has the lowest degree and is generated, edges are used. AD and CD has been used twice and are removed.

Edge set: AB(2, 1), BC(2, 1), CA(2, 1), AE(2, 2), BE(2, 2), CE(2, 2).

Triangle set: ABC(6), ABE(6), BCE(6), CAE(6).

–

ABC (inner face) has the lowest degree and is generated, edges are used. AB, BC and CA has been used twice and are removed.

Edge set: AE(2, 2), BE(2, 2), CE(2, 2).

Triangle set: (ABE(6), BCE(6), CAE(6)).

ABE, BCE and CAE cannot be generated because their edges AB, BC and CA have already been used. However, edges AE, BE and CE still remain. Algorithm fails. This shows why the third rule cannot be applied alone and only as a complement of the second rule.

3.5 Algorithm details

A vertex is defined by its ID and contains a list of edges it connects. Its degree is equal to the number of edges in its list.

An edge is defined by two vertices it connects in any order and contains a list of triangles it is part of, and the information, whether it has been already used once. Its degree is equal to the number of triangles in its list.

A triangle is defined by three vertices (or edges) it contains in any order. Its degree is equal to sum of its edge's degrees.

During the face generation from a triangle: All its edges are marked as used and if they have already been used before, all remaining triangles they are part of are removed, since they must be inner faces. The triangle is then removed from the triangle set and added to the set of proper faces.

[11]Note: The first number is edge degree[6], second is remaining number of uses.

[12]Note: The number refers to triangle degree[6].

During the triangle removal: Degree of all its edges is decremented. After that, it can be checked, whether the degrees of its edges are still bigger or equal to how many times the edges should be used. If it is not, the algorithm failed.

The algorithm proceeds as follows:

1. The set of edges is loaded (it is the required input).

2. The set of all triangles is generated. It means a search from every edge is made to the depth of three edges and should a new cycle be obtained in this way, it is inserted in the set of triangles. This is further called preprocessing[13].

3. The first rule is applied. Faces are generated from triangles which contain at least one trihedral vertex. (Used edges are marked, triangle set is reduced by triangles which can no longer be generated.)

4. The second rule is applied. Faces are generated from triangles which contain at least one edge with degree equal to the number of its necessary uses. This is repeated as long as any such triangle remains.

5. The third rule is applied. Should any triangle remain (and it should not) in the triangle set, then the one with lowest degree is generated. After that the algorithm continues with step 4.

6. As no triangle remain in the triangle set, the face set is complete.

3.6 Implementation details

The time complexity of this algorithm can be as good as $O(n)$, where n is the number of proper faces (eventually vertices or edges, these are directly or almost directly proportional depending on genus). This is confirmed for tested meshes by results in Table 1 as well as by graph in Fig. 17. But in order to achieve the complexity, the algorithm must be implemented in an effective way. The most important in this is the right use of sets. For the algorithm to remain $O(n)$ in all its parts, all operations with the sets[14] have to be $O(1)$. In our case, we used the .NET Hashset class[15] as hashing is probably the simplest way to solve the problem. The only difficulty was that the hashset needs $O(n)$ to create an enumerator to iterate through the set and this enumerator becomes unusable after every change in the set. This means that all the elements of the set have to be marked first and removed after the search of the whole set.

[13]It is not part of the algorithm, as it is unnecessary should the algorithm be used only for inner face removal.

[14]Operations like add(), remove(), contains(), get() etc.

[15]https://msdn.microsoft.com/en-us//library/bb359438(v=vs.110).aspx

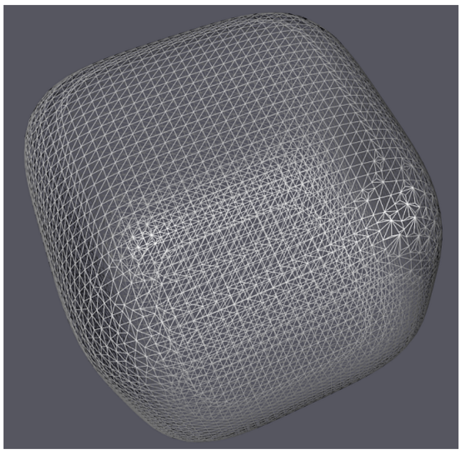

Figure 2: Cube mesh with complexly rounded edges. Implicit function isosurface obtained using the dual contouring algorithm [4].

4 Results

The test data was all .OBJ files of closed edge-manifold triangle meshes. Not only could this data by easily obtained in huge variety and easily processed, but it can also be easily used to verify that the face set has been correctly reconstructed. On the other hand, no other similar algorithm has been tested along with the above described one. The algorithm based on planarization mentioned in Section 2 could be used as such, but due to its specifics it can be assumed it would be slower being $O(n \cdot log(n))$ at best and could be only used on meshes with genus 0. As this new algorithm had been shown to run in linear time and is more universal, the planarization algorithm should end up losing. This is however very dependent on particular implementation of planarization and was not tested, as the implementation effective enough to stand against our algorithm would be significantly more difficult to prepare[16].

4.1 Test data

Following meshes were tested with results shown in Table 1:

Meshes to prove basic functionality - Double Tetrahedron (fig. 1), Icosahedron (fig. 3), Spheres (fig. 13), Figure (fig. 4) and Lion (fig. 11).

Meshes to prove identical results on meshes with same topology and only different coordinates - Bunny (fig. 8) and Helix (fig. 16) type meshes, found in MeshTest [9].

Random meshes from Thing10K [11], some of which are not genus 0 or consist of separate parts - Alien Egg (fig. 12), Octocat (fig. 14), Loop (fig. 15), Land (fig. 7), Water (fig. 6) and Gear (fig. 10).

A mesh with almost a million faces from the Smithsonian X 3D library - Palmyra (fig. 5).

[16]So far, we found no tested planarization algorithm specifically for inner face removal, so this all is only our speculation.

Figure 3: Icosahedron mesh.

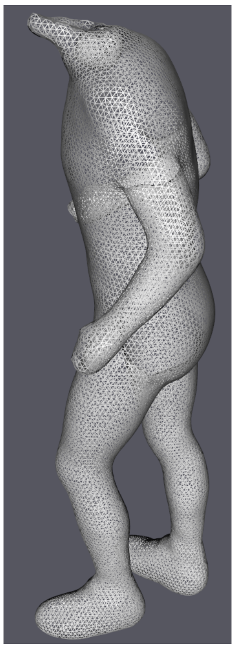

Figure 4: Figure. Reconstructed human figure [1], [6].

A mesh with related inner faces - Cube (fig. 2).

Meshes with very high amount of inner faces - Monster type meshes (fig. 9), named simply "Mesh"+vertex count. These has been generated by our own custom software designed to maximize the number of inner faces.

The algorithm seems to run in $O(n)$. It is fast and able to solve even complex meshes. Practical time complexity of this algorithm, should it be computed, would oscillate around n^1. Graph of linear regression (fig. 17 and fig. 18) shows mild descent, but this irregularity is very dependent on which meshes are included. Accurate practical time complexity computation would probably require many meshes with 100k+ faces.

Spheres and Cube meshes are processed in noticeably longer time. In the case of Spheres, it may be affected by the inaccuracy evident for smallest meshes such as Double Tetrahedron and Icosahedron, but it seems odd, since Mesh1000 has similar number of faces and is almost twice faster despite having a high amount of inner faces. In the case of Cube, it is probably caused by inner faces being interconnected with each other and thus obstructing their elimination. But it does not explain why Monster type mashes are processed so quickly despite of their complex structure.

File	Vertices	Edges	Inner faces	Faces	Time $[s]$	Time/face $[\mu s]$	Preprocessing $[s]$	Prep/face $[\mu s]$	Prep/alg
Palmyra.obj	492465	1477425	678	984950	12.576	12.768	11.481	11.657	91.30%
Alien_Egg.obj	38788	117504	0	78336	1.096	13.990	0.909	11.601	82.93%
Bunny.obj	35946	107832	0	71888	0.893	12.417	0.782	10.885	87.66%
Helix.obj	30534	91590	0	61060	0.735	12.035	0.591	9.683	80.46%
Octocat.obj	20125	60369	30	40246	0.500	12.413	0.389	9.662	77.84%
Water.obj	18853	56817	1063	37878	0.469	12.389	0.388	10.256	82.78%
Figure.obj	15830	47490	2	31660	0.396	12.517	0.314	9.922	79.27%
Land.obj	14738	44208	49	29472	0.360	12.230	0.274	9.306	76.09%
Gear_Hypoid.obj	14566	43698	106	29132	0.364	12.490	0.274	9.413	75.36%
Cube.obj	5858	17568	116	11712	0.536	45.783	0.155	13.268	28.98%
Loop.obj	3040	11520	0	7680	0.111	14.411	0.063	8.141	56.50%
Lion.obj	2213	6633	10	4422	0.053	11.921	0.034	7.631	64.01%
Spheres.obj	974	2910	0	1940	0.038	19.766	0.016	8.285	41.92%
Icosahedron.obj	12	30	0	20	0.000	9.217	0.000	4.891	53.06%
Double_Tetrahedron.obj	5	9	1	6	0.000	34.300	0.000	14.587	42.53%
Mesh1000.obj	1000	2994	996	1996	0.024	12.181	0.020	10.055	82.55%
Mesh2000.obj	2000	5994	1996	3996	0.049	12.355	0.041	10.337	83.67%
Mesh3000.obj	3000	8994	2996	5996	0.075	12.501	0.064	10.690	85.52%
Mesh4000.obj	4000	11994	3996	7996	0.099	12.344	0.086	10.807	87.55%
Mesh5000.obj	5000	14994	4996	9996	0.127	12.741	0.112	11.181	87.75%
Mesh6000.obj	6000	17994	5996	11996	0.155	12.921	0.138	11.502	89.02%
Mesh7000.obj	7000	20994	6996	13996	0.173	12.372	0.204	14.546	117.58%
Mesh8000.obj	8000	23994	7996	15996	0.202	12.633	0.207	12.935	102.39%
Mesh9000.obj	9000	26994	8996	17996	0.230	12.768	0.267	14.835	116.20%
Mesh10000.obj	10000	29994	9996	19996	0.259	12.944	0.289	14.458	111.69%
Mesh11000.obj	11000	32994	10996	21996	0.305	13.865	0.338	15.369	110.85%
Mesh12000.obj	12000	35994	11996	23996	0.343	14.297	0.373	15.530	108.62%
Mesh13000.obj	13000	38994	12996	25996	0.373	14.359	0.402	15.454	107.62%
Mesh14000.obj	14000	41994	13996	27996	0.402	14.346	0.451	16.096	112.20%
Mesh15000.obj	15000	44994	14996	29996	0.429	14.306	0.481	16.019	111.98%
Mesh16000.obj	16000	47994	15996	31996	0.444	13.884	0.524	16.388	118.04%
Mesh17000.obj	17000	50994	16996	33996	0.454	13.349	0.542	15.957	119.53%
Mesh18000.obj	18000	53994	17996	35996	0.485	13.469	0.542	15.067	111.87%
Mesh19000.obj	19000	56994	18996	37996	0.533	14.019	0.601	15.820	112.85%
Mesh20000.obj	20000	59994	19996	39996	0.556	13.893	0.630	15.752	113.38%
Mesh21000.obj	21000	62994	20996	41996	0.561	13.351	0.670	15.964	119.57%
Mesh22000.obj	22000	65994	21996	43996	0.606	13.767	0.691	15.698	114.02%
Mesh23000.obj	23000	68994	22996	45996	0.622	13.522	0.763	16.580	122.61%
Mesh24090.obj	24090	72264	24086	48176	0.665	13.796	0.785	16.293	118.10%

Table 1: Testing data with their properties, average time for processing and preprocessing (of 100 tests), this time per face count, and ratio between preprocessing and algorithm processing (*Prep/alg*). Preprocessing means generating a set of all triangles.

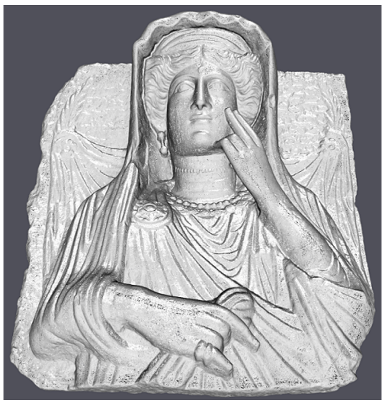
Figure 5: Palmyra mesh.
Smithsonian institution [3].

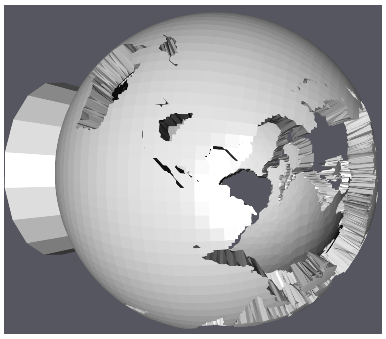
Figure 6: Earth without land.
Water.stl in [11].

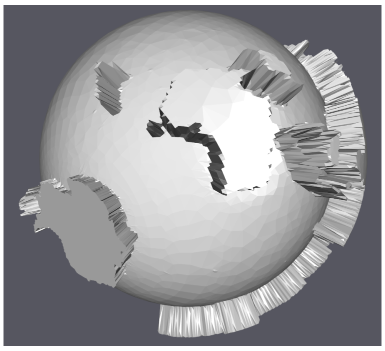
Figure 7: Earth with no water.
Land.stl in [11].

Figure 8: Bunny mesh.
bunny in [9].

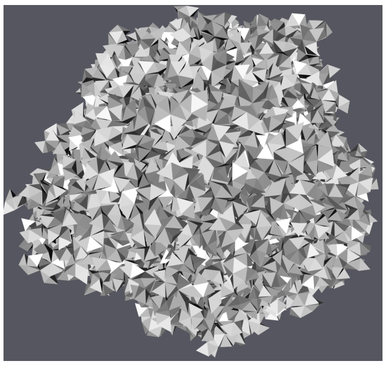
Figure 9: Monster type mesh.

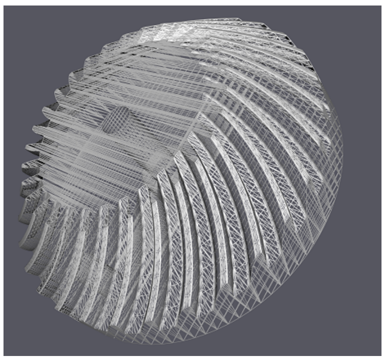
Figure 10: Gear mesh.
gear_hypoid_left.stl in [11].

Figure 11: Lion mesh.
Courtesy of Ivo Hanak.

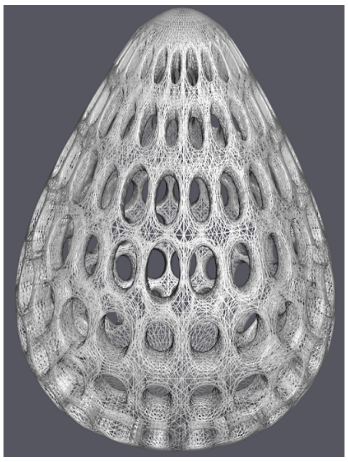
Figure 12: Alien egg mesh.
alien_egg.stl in [11].

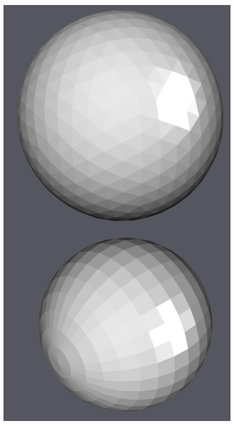
Figure 13: Spheres; Models obtained by polar parameterization and icosahedron subdivision.

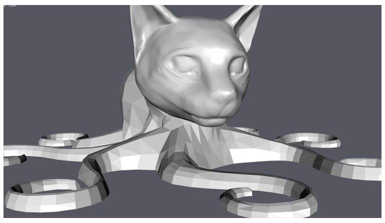
Figure 14: Octocat mesh.
Octocat-v2.stl in [11].

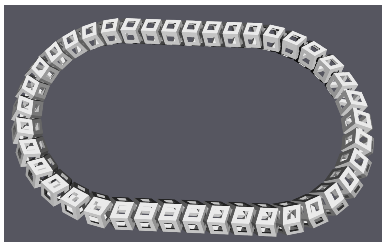
Figure 15: Loop mesh.
loop.stl in [11].

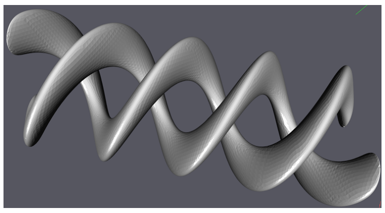
Figure 16: Double helix.
out in [9].

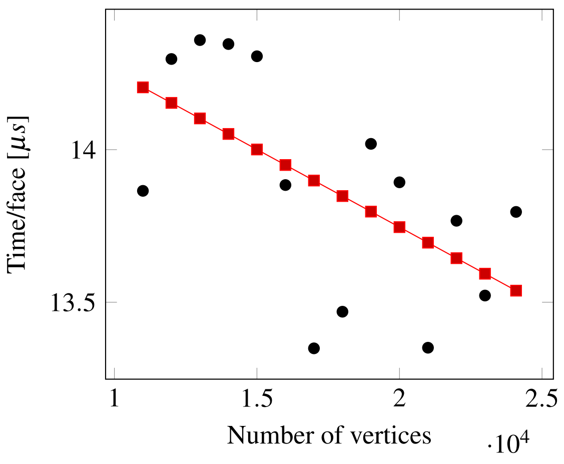

Figure 17: Graph showing linear processing time for Monster type meshes.

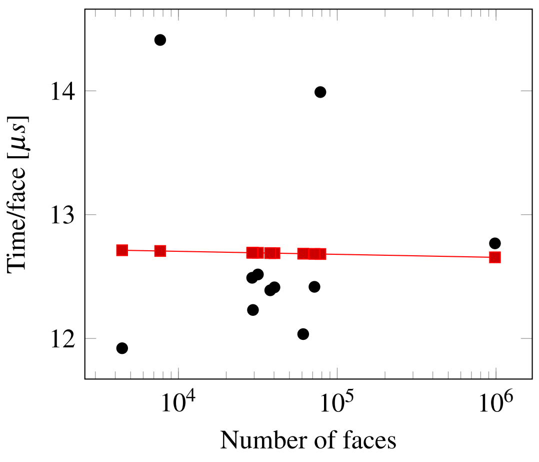

Figure 18: Graph showing linear processing time for rest of the meshes.

5 Conclusions

A simple, purely topological algorithm of $O(n)$ complexity which recreates a set of non-oriented triangle faces from only the set of edges was proposed, implemented and tested. It appears to be both fastest and most universal (in terms of mesh complexity) algorithm solving the problem we know about. So far, no closed edge-manifold triangle mesh has been found on which it fails. Experiments demonstrate it runs in $O(n)$ and the following properties were observed:

Processing time grows linearly with the number of proper faces for meshes bigger than 1000 faces, while the processing time of small meshes fluctuates due to random effects. For most of the meshes, time required to process one face is approximately $13\mu s$ on current consumer level PC.

Meshes processing is affected neither by mesh genus, nor by the number of separate components it consists of.

The number of inner faces in comparison to true faces also does not seem to affect efficiency.

Some meshes take noticeably more time to process from still unknown reasons and relevance of some of the algorithm supplementary elements (Rule 1 and 3) has not yet been accurately tested.

To find out more details about this algorithm and its more efficient implementation, further tests are needed.

Acknowledgement

This paper was written using Overleaf.
Tables generator was used to generate Table 1.
Mesh screenshots has been made using MeshMagic.
Bibliography has been made using Citation Machine.
Meshes were converted to .obj using Spin 3D.

References

[1] Nizam Anuar and Igor Guskov. Extracting animated meshes with adaptive motion estimation. In *Vision, Modeling, and Visualization*, pages 63–71, 2004.

[2] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lvy. *Polygon Mesh Processing*. AK Peters, 2010.

[3] Smithsonian Institution. Funerary relief bust of haliphat. https://3d.si.edu/model/fullscreen/p1b-1474716020541147811544479O-0.

[4] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Trans. Graph.*, 21(3):339–346, July 2002.

[5] Mauricio G. C. Resende and Celso C. Ribeiro. *Graph planarization*, pages 908–913. Springer US, Boston, MA, 2001.

[6] Peter Sand, Leonard McMillan, and Jovan Popović. Continuous capture of skin deformation. *ACM Trans. Graph.*, 22(3):578–586, July 2003.

[7] Peter A.c. Varley. Implementing the new algorithm for finding faces in wireframes, 2009.

[8] Peter A.c. Varley and Pedro P. Company. A new algorithm for finding faces in wireframes. *Computer-Aided Design*, 42(4):279–309, 2010.

[9] Libor Vasa. Software for comparing models meshtest, 2010.

[10] Luca Vismara. Planar straight-line drawing algorithms. In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization.*, pages 193–222. Chapman and Hall/CRC, 2013.

[11] Qingnan Zhou and Alec Jacobson. Thingi10k, 2016. https://ten-thousand-models.appspot.com/.