SIACG 2002 - 1st Ibero-American Symposium on Computer Graphics
1-5 July 2002, Guimarães - Portugal

Determining orientation of Laser scanned surfaces

João Fradinho Oliveira Anthony Steed
Department of Computer Science, University College London
Gower Street, WC1E 6BT London
{Joao.Oliveira, A.Steed}@cs.ucl.ac.uk

Abstract
Real 3D data acquired from scanning technology provide interesting 3D models for research and industrial applications. However before these models can be used, a surface needs to be fitted to a point cloud of an unknown object, this process might create some undesirable properties, such as triangle normals pointing in incorrect directions. We present a robust algorithm that reliably fixes these triangle normal problems on non-manifold, and self-interesting surfaces of scanned objects.

Keywords
Normal fixing, vertex ordering, non-manifold, self-intersection, Complex Boundary vertex

1. INTRODUCTION

Fitting and orienting[*] a 3D surface to a point cloud that comes from an unknown scanning device can pose a difficult problem. This is particularly the case when the scanner returns only point location information and not range data or surface direction[ϕ]. In practice even non-scanned objects, modelled by hand can have problems with inconsistent normals.

However in Computer Graphics one often renders only triangles that are facing the viewer to speed up the rendering. In such systems having inconsistent normals produces visually incorrect results (Figure 12, leftmost column, and Figure 13, rightmost column, show white gaps in the original model where the triangle normals are pointing inwards to the model). Some 3D software viewers avoid this problem simply by rendering both front-facing and back-facing triangles but clearly this is not ideal since the rendering speed is halved.

Since the object is a scan, it should be possible to get the correct visual result by rendering each triangle from only one of its two sides. Techniques that can attempt to determine which side is the correct side often rely on counting surface intersections of rays to determine inside/outside directions. These techniques face a difficult problem with non-manifold and self-intersecting surfaces since it is often the case that clusters of degenerate surfaces float inside an object where for example the surface did not have enough sample data.

[*] Triangles specified with vertex order consistent with the right hand rule, point in the direction of the thumb.

[ϕ] this is particularly likely if the data was obtained from a system with several different sensors, such as the whole body scanner described in [Horiguchi98])

Some previous work is mentioned in section 2. In this paper we present a robust algorithm that reliably fixes normals of degenerate surface scans in section 3. In section 4 we present some implementation issues. In section 5 we present results. Finally in section 6 we conclude and lay out future work .

2. PREVIOUS WORK

A collection of points on a flat surface can be given an arbitrary global surface orientation, depending on which side of the surface was used when applying the right hand rule. With a closed surface the decision is no longer arbitrary, because there is an orientation where all the triangles are specified to point towards the outside and thus only the exterior is rendered when back-face culling. Therefore it becomes important to reliably determine what direction is inwards, and what is outwards. Simply using a triangle normal to intersect the rest of the object, and counting the number of intersections, could give an idea of whether for example a triangle normal is pointing outwards (one triangle intersection), or inwards (two triangle intersections, possibly more), but problems arise when there are multiple disconnected surfaces in the object which cross these intersection paths. Self-intersecting surfaces resulting from noisy data can also limit the reliability of this test. For example consider a flat surface, completely defined by counter clockwise order. If this flat surface intersects and shares geometry with other surfaces that are noise artefacts, then the side criteria could swap at those locations and locally inconsistent normals will be produced. There is not much published work on fixing inconsistent normals. There are commercial tools available but unfortunately no details on their operation or quality when dealing with real data such as that of laser scans. Furthermore, some solutions require the user to choose the directions manually. This can be quite tedious if the object is composed of hundreds of patches, located dispersedly in the model. We compare our results with a widely free distributed tool called ivnorm [Bell95] in section 5.

3. DETERMINING ORIENTATION

Our algorithm has three distinct phases. The first one, described in section 3.1 creates normal groups, removing the problems of non-manifold parts. The second in section 3.2 in which rays are used in conjunction of opposite directed rays to find a reliable test for the normal group direction. And the third in 3.3 where we triangulate holes created from deleting non-manifold parts.

3.1 Normal group creation

Non-manifold edges, e.g. edges that are shared by more than two triangles, create a problem when performing surface connectivity queries such as determining the adjacent connected triangle of another triangle at an edge. This ambiguity can generate problems when trying to group a large area of connected triangles. Often these inconsistencies in surface scans are the line of contact of several noisy small surfaces. These edges can cause fragmented smaller groups of triangles, which increase the computation demands of determining more correct orientations for those groups. Furthermore they can inconsistently propagate vertex-order criteria of a given direction. To overcome this problem we simply remove all triangles connected to non-manifold edges, and we flag the vertices that became borders in result of these deletions. This flag at vertex level allows us to preserve holes of the original model, and only fill created holes. For determining non-manifold edges we use the connectivity/marking data structure adopted by [Garland99]. Figure 1, illustrates the procedure. For every edge in the model, we zero the face markers of every triangle connected to each of the edge's vertices. One can build lists of faces at vertex level by taking one triangle at a time, and adding the triangle index to each of it's three vertices own face lists. Next we increment by one the face markers of every triangle associated with the first vertex of the edge. Finally we increment by one all the face markers of the triangles associated with the second vertex of the edge. If there are more than two triangles with a face mark value of 2, then we delete all the triangles with a face mark value of 2.

Once the triangles associated with non-manifold edges are deleted, we pick the first triangle of the object, mark it with the current number of the group, retrieve the three adjacent triangles and force the vertex order on them to be consistent with the picked triangle, and recursively apply the same procedure to the retrieved triangles that have not been marked yet. When there are no more connected triangles, the recursion will stop and return to the main loop, where the triangles that have been marked are skipped until an unmarked one starts a new group.

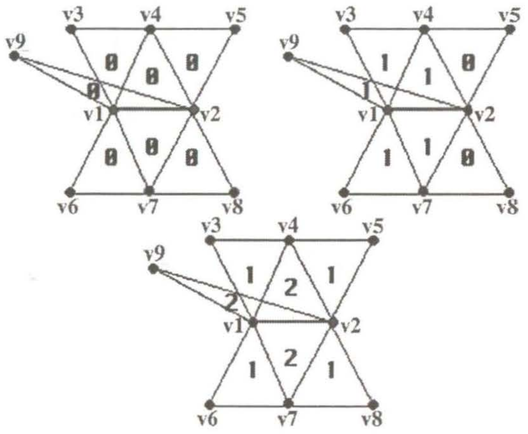

Figure 1: Detecting non manifold edges

```
void fixallnormals(GObject *theobject) {
        deletenonmanifoldregions(theobject); fgroup=1;
        for(i=0; i<theobject->farray->size; i++) {
            f=atFaceArray(theobject->farray, i);
            if(f->wt==0) { /*first ever vertex order specification*/
                fixvertexnormals(theobject,f, fgroup); fgroup++;
            }
        }
}
void fixvertexnormals(GObject *theobject, Face *f, int fgroup) {
        f->wt=fgroup; v1id=f->first->i; v2id=f->first->next->i;
        v3id=f->first->next->next->i;
        nf1=getadjacentfaceatedge(v1id, v2id, f, theobject);
        forcevertexorder(f, nf1);
        nf2=getadjacentfaceatedge(v2id, v3id, f, theobject);
        forcevertexorder(f, nf2);
        nf3=getadjacentfaceatedge(v3id, v1id, f, theobject);
        forcevertexorder(f, nf3);
        if(nf1->wt==0){fixvertexnormals(theobject,nf1,fgroup);}
        if(nf2->wt==0){fixvertexnormals(theobject,nf2,fgroup);}
        if(nf3->wt==0){fixvertexnormals(theobject,nf3,fgroup);}
}
```

Figure 2: Pseudo-code of normal grouping

3.2 Reliable ray tests

The previous phase, on average, successfully groups 98% of the model into one surface group (Table 1, 2nd and 4th column). Given that one test can potentially determine the orientation of the whole group, this allows for some freedom to choose reliable test rays for all our tests. For simplicity a ray is defined with two points: a starting point (Cx, Cy, Cz) and a second point (Bx, By, Bz) following a particular triangle orientation $\vec{N}$ or $-\vec{N}$, see Figure 3. A point (Px, Py, Pz) on the ray can be found with the following equations:

$$Px = Cx + \alpha(Bx - Cx) \qquad (1)$$
$$Py = Cy + \alpha(By - Cy)$$
$$Pz = Cz + \alpha(Bz - Cz), \text{ where } 0 \leq \alpha < +\infty$$

In our case we are interested in finding a point of the ray that intersects a plane:

$$a(Px)+b(Py)+c(Pz)+d=0 \qquad (2)$$

substituting (1) in (2),

$$a(Cx+\alpha(Bx-Cx))+b(Cy+\alpha(By-Cy))+c(Cz+\alpha(Bz-Cz))+d=0$$

and solving for α:

$$\alpha=\left(\frac{-a(Cx)-b(Cy)-c(Cz)-d}{a(Bx-Cx)+b(By-Cy)+c(Bz-Cz)}\right) \qquad (3)$$

Care needs to be taken with the denominator of (3), as the ray might be parallel to the plane and not intersect, yielding a zero dot product between the plane normal and the ray's orientation.

The outline of our test strategy is as follows:

1. For each normal group, choose one triangle to fire a ray from.
2. For the picked triangle create 3 random barycentric coordinates for the starting point C of the ray. Make sure the random point is not on one of the edges of the triangle as this would count as two intersections. Continue to create random barycentric coordinates if they fall on an edge.
3. Use the triangle normal $\vec{N}$ to calculate the second point A, that determines the direction of the ray.
4. Intersect the ray with all the triangles of the model. If you have a spatial data structure, query the data structure. If the ray hits an edge, go back to 2, with the same triangle. If not, record the number of hits (hitsA). If hitsA is one, proceed with the next group. Go to step 1.
5. Use the triangle normal to calculate a ray with the opposite direction $-\vec{N}$ of the created in 3.
6. The same as 4, compute two new rays for the same triangle if it hits an edge. If not, record the number of hits separately (hitsB).
7. Check to see if either hitsA or hitsB has the value of one. If hitsA has a value of one, it means that the group was oriented correctly, and we proceed with a triangle of the next group. If hitsA has not got a value of one, but hitsB has, then we reverse the vertex order for all the triangles in the group, and proceed to the triangle of the next group.
8. If neither hitsA or hitsB has a value of one, we go back to step 2, with another triangle of the same group. Hopefully this new triangle will be positioned in a more reliable location, away from self-intersecting surfaces. In principle, with surface models, it should be possible to find a triangle in the group, where one of the rays hits only one triangle, the one it started from. We also keep a count of how many triangles we have tried, and if we have tried all the triangles in the group, we reason with smallest value between hitsA and hitsB. If hitsA has the smallest value, and it is odd, then we proceed with the triangle of the next group. If it was even, we invert the vertex order of all the triangles in the group, as we do if hitsB had the smallest value and it is odd. Finally if hitsB has the smallest value and it is even, we do not invert the vertex order of the group and proceed to step1.

Care needs to be taken to avoid double counting of triangle intersections when a ray hits an edge. Shooting systematically the rays from the centre of a triangle is a bad strategy as illustrated in Figure 3. We use random barycentric coordinates described in the next section to generate our starting point for the ray. Note that in step 8, if neither hitsA or hitsB has a value of one, then either: a) the triangle is positioned in a way that it's rays hit another part of the surface (e.g. with a scanned upright human, the rays from one ankle could genuinely hit triangles in the opposite leg) or b) we are dealing with a triangle that is inside the model, in the context of laser surface scans, this would typically be a self intersection of the fited mesh. Since it is not possible to distinguish between the two cases, ultimately our search for a reliable one hit ray, allows us to cope with these degeneracies. We present results on four scanned models in section 5.

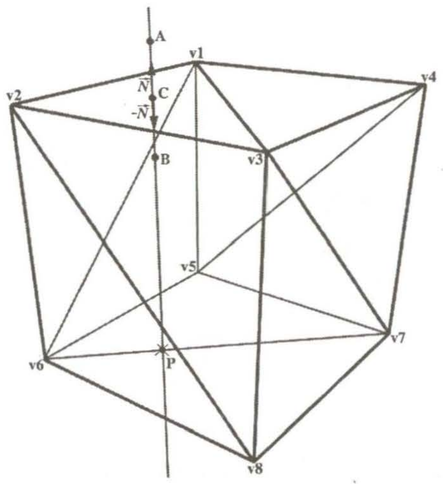

Figure 3: Ray B-C, originates from the centre of the top triangle and hits an edge in the bottom triangles.

3.3 Hole triangulation

Finally we triangulate holes resulting from the initial deletion of triangles around non-manifold edges. The first step of the hole triangulation process is to retrieve border edges that have their vertices flagged from the initial deletion process. These border edges can be found with a marking strategy similar to the one illustrated in Figure 1, with the difference being that an edge is classified as a border edge if there is only one triangle that shares the edge with the value of 2. Lists are made to track these detected border edges, and they are sorted according to the smallest index value of the vertex pair. This allows one to easily follow a connected edge sequence in the list. When the sequence is broken, e.g. an edge shares no values with the previous edge in the list, this indicates the start of a different hole in the model. It would be desirable to correct initial non-manifold configurations when triangulating their holes, one way to try to achieve this, would be to only triangulate a sequence of connected border edges whose border triangles were classified into the same normal group. This situation is illustrated in Figure 4. New triangles are created with the vertices of the border edges and the centroid "P".

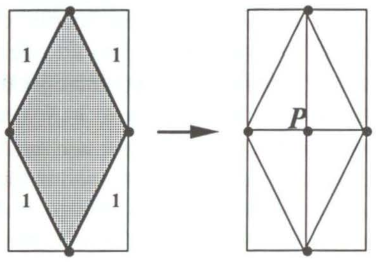

Figure 4: left: border vertices with border-edge valence of 2, right: hole triangulation

Connecting the border vertices to the centroid, does not ensure non-self intersection of the resulting surface. Unfortunately neither does it ensure non-manifold edge creation. Figure 5 shows a connected edge border sequence whose border triangles were tagged to the same normal group "1". The vertex A, has a valence of 4 border edges connected to it, instead of 2 as in border vertices in Figure 4. Although the border edge sequence is valid, the border vertex sequence is not. The image on the right shows four darkened triangles that share the resulting non-manifold edge P-A.

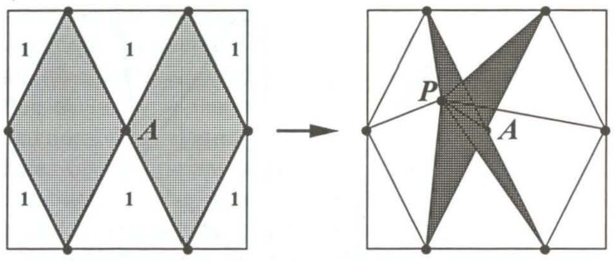

Figure 5: left: *Complex Boundary vertex* "A" with border-edge valence of 4, right: hole triangulation with non manifold edge PA

We have tried to triangulate sequences that stopped at border vertices with valence higher than 2, *Complex Boundary vertices*. But unfortunately in all the scanned models, all the resulting border vertices have a valence of 4. An object that similarly exhibits this property is the Sierpinski triangle (Figure 6), where all but the three corners on the silhouette of the object have a border edge valence of 4. It is not clear what benefits other hole triangulation schemes such as [Held01] and [Schroeder92] can offer in this situation. Border vertices that have a border-edge valence higher than 2 are likely to create non-manifold configurations. For completeness we would like to add the vertex classification: *Complex Boundary* to Schroeder's five: Simple, Complex, Boundary, Interior Edge, Corner. We note that *Complex Boundary* vertices were created through the decimation of a complete edge, and that although they don't have non-manifold edges connected, they are complex. In the end we consider that the initial mesh in these cases is already non-manifold and choose to accept non-manifold edge creation, using our robust centroid triangulation scheme, to avoid visible holes in the model.

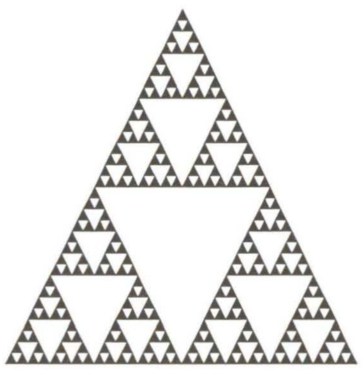

Figure 6: Sierpinski triangle, border edge valence > 2

4. IMPLEMENTATION ISSUES

As mentioned in the previous section, one needs to be careful with ray edge hit condition. A classical problem in raytracing is illustrated in Figure 7.

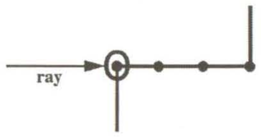

Figure 7: Multiple ray edge hit

In the situation above, five triangles are hit, where it should count only as one. We detect edge hits between a ray and a triangle by forming two vectors **v1** and **v2** with the intersection point and two vertices of the triangle (Figure 8), if either angle ∂1, ∂2 or ∂3 formed between one of these vectors and an edge is smaller or equal to half a degree, we classify the intersection as an edge hit, and discard the ray. Figure 8 illustrates the two vectors **v1** and **v2**, and how an initial random starting point is computed. We call a random number generator three times, and divide each number by the sum of the three. A point can then be calculated by:

$$P(\beta 1, \beta 2, \beta 3)=P1+\beta 2*(P2-P1)+\beta 3*(P3-P1) \quad (4)$$

where, $$\beta 1+\beta 2+\beta 3=1 \quad (5)$$

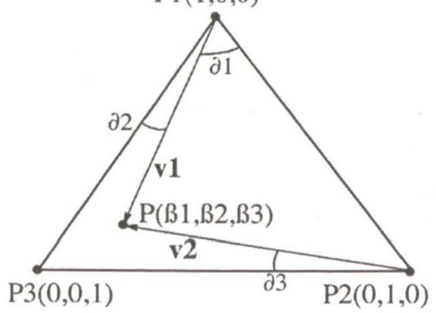

Figure 8: Barycentric coordinate point and edge nearness tolerance

5. RESULTS

We have tested our algorithm in several surface models (Figure 9,10) and in larger laser scanned models, with simple topology (Figure 11) and more complex (Figure 12, 13). In this work we have used data from a Hamamatsu Body Lines scanner which offers 1-2 mm accuracy over approximately regular samples at 5 mm spacing over 400 horizontal slices of the body [Horiguchi98]. We have also used a surface reconstruction software called Cocone, freely available at [Dey02], based on [Amenta00] to fit a surface to the scanned point cloud. The experiments were carried out on a PowerBookG4 500MHz, 1Gb RAM. This computer is capable of computing 1 million ray triangle intersections in ~7 seconds (including the edge hit test). Numerical results for all the models can be found in Table 1. The table shows that reliable tests from our algorithm are affordable even with larger models. The time increases with how degenerate the model was, for instance Igor2 (fifth row of Table 1), has fewer triangles than Igor3, but has more normal groups (column 3) and takes longer. We expect that with even larger scanned models, the number of mesh degeneracies will remain the dominant time factor. The third column of Table 1 shows that the problem of real data is finding the orientation relationship between several surfaces and not just one. The fifth columns indicates how easy it is to get the intersection counts wrong with a one ray strategy as it is very likely to hit an edge in a dense model, and considering the number of groups to test. The sixth column, shows how useful it was to use our two opposing orientation rays strategy, it shows that calculating the ray opposite to a triangle normal was determinant in finding the correct orientation of half of the normal groups.

The following figure presents an inconsistent symmetrical object on the left and shows results after applying our algorithm on the right.

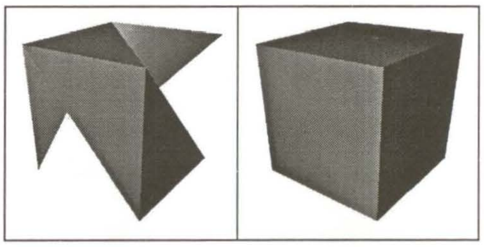

Figure 9: left: inconsistent normals, right: normals after applying our algorithm

Figure 10 shows a Mobius strip with inconsistent normals (top), and results after applying our algorithm (below). The transition of the normals from *outside* to *inside* can clearly be seen. The object is completely front facing from this view, and completely invisible on the other side facing the viewer.

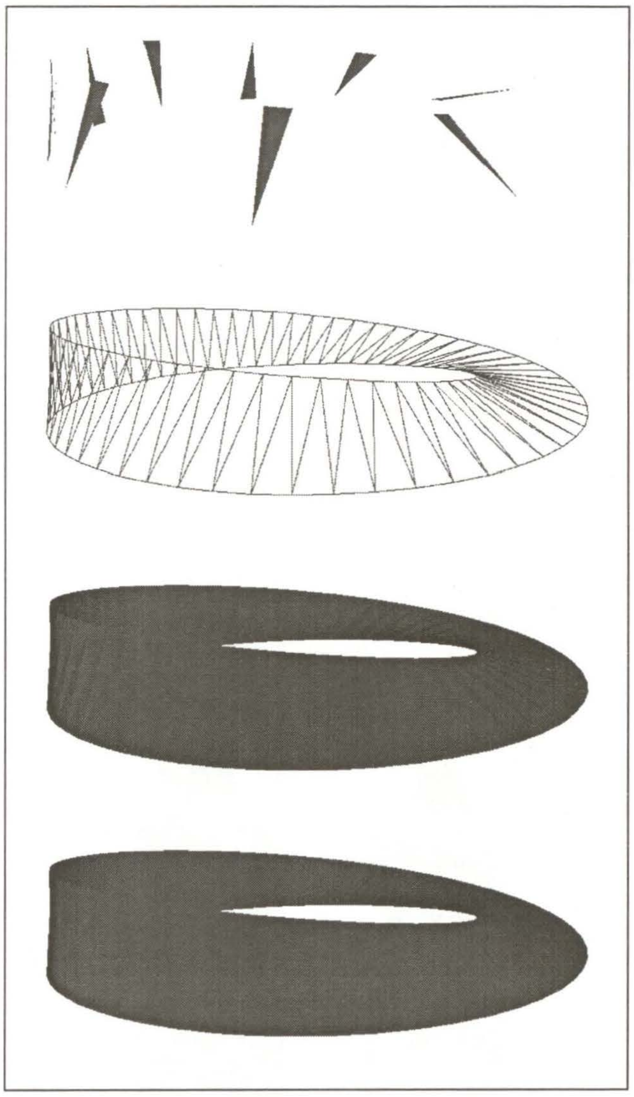

Figure 10: from top: Mobius strip with inconsistent normals, wireframe results after applying our algorithm, flat shading results, Gourard shading results

Model name	# triangles	# Normal Groups	# triangles in largest group	# ray starts on edge	# opposite orientation rays with single hits	# rays shot	Time (s)
Cube	12	1	12	1	0	12	<1
Mobius strip	120	1	120	1	1	240	<1
Mannequin	31662	18	31544	55	6	2461786	20
Igor1	66164	44	65806	89	20	8703944	69
Igor2	62982	80	62280	160	43	16182298	126
Igor3	68590	58	67977	116	26	11445967	91

Table 1 – Numerical results

6. FUTURE WORK

As mentioned in the beginning of section 3.2, the fact that more than 90% of the triangle's orientation can be determined with a few ray tests allows for extra ray tests to find a reliable test. Although we generate new random points when a ray starts at an edge, or hits an edge, we have not explored more exhaustive searching on the triangle to find a position that would yield a one triangle hit ray. We plan to use a spatial data structure [Arvo87] to query only specific parts of a model with our rays, and hope that the time saved will allow for more tests in small problematic areas such as concavities, with internal surfaces inside, where a particular starting ray position could determine a reliable one hit triangle test. Currently the worst case situation of our algorithm would be if the model had a double hull, which does not occur in surface scans. Nevertheless our algorithm can be changed in step 8, to not fallback and try all the triangles in a group, instead it could just rely on the last part of step 8, using the first pair of *edge hit free* rays calculated for the group to compute an answer for generic models. Finally, regarding the initial deletion of non manifold triangles, it would be interesting to attempt to separate the connecting surfaces by creating new vertices with small shifts in coordinates for each surface, hence eliminating non manifold configurations without deletion, and small error.

7. ACKNOWLEDGEMENTS

The authors would like to thank Hamamatsu for the loan of the Body Lines scanner to the Department. In addition, one of us [JFO] would like to thank the Fundação Calouste Gulbenkian and JNICT/PRAXISXXI, for financial support. Finally we would like to thank Dr. W. B. Langdon for some advice on the random barycentric coordinate generator, Francis Gioia for his comments and Tamal Dey for providing Cocone. For the models we would like to thank Jorge Fradinho Oliveira for the Igor scans, Bernhard Spanlang for the manequin scan, and Richard Marsden for the mobius strip model.

8. REFERENCES

[Amenta00] N. Amenta, S. Choi, T. K. Dey and N. Leekha. "A simple algorithm for homeomorphic surface reconstruction".Proc. 16th ACM Symposium on Computational Geometry, 213-222.

[Arvo87] Arvo, J., and D. Kirk, "Fast Ray Tracing by Ray Classification", SIGGRAPH 87, 55-64.

[Bell95] www.webhistory.org/www.lists/www-vrml.1995q2/1779.html, accessed 21st March 2002.

[Dey02] www.cis.ohio-state.edu/~tamaldey/cocone.html

[Garland99], Michael Garland, Quadric-Based Polygonal Surface Simplification, Ph.D. Thesis, Tech. Rept. CMU-CS-99-105.

[Held01] M Held, "FIST: Fast Industrial-Strength Triangulation of Polygons", *Algorithmica* 30(4): 563-596.

[Horiguchi98] Horiguchi C, Hamamatsu, BL (Body Line) Scanner, *International Archives of Photogrammetry and Remote Sensing*, Vol XXXII, Part5.

[Schroeder92] Schroeder W J, Zarge J A, and Lorensen E, "Decimation of Triangle Meshes", *Proceedings of SIGGRAPH 92*, 65-70.

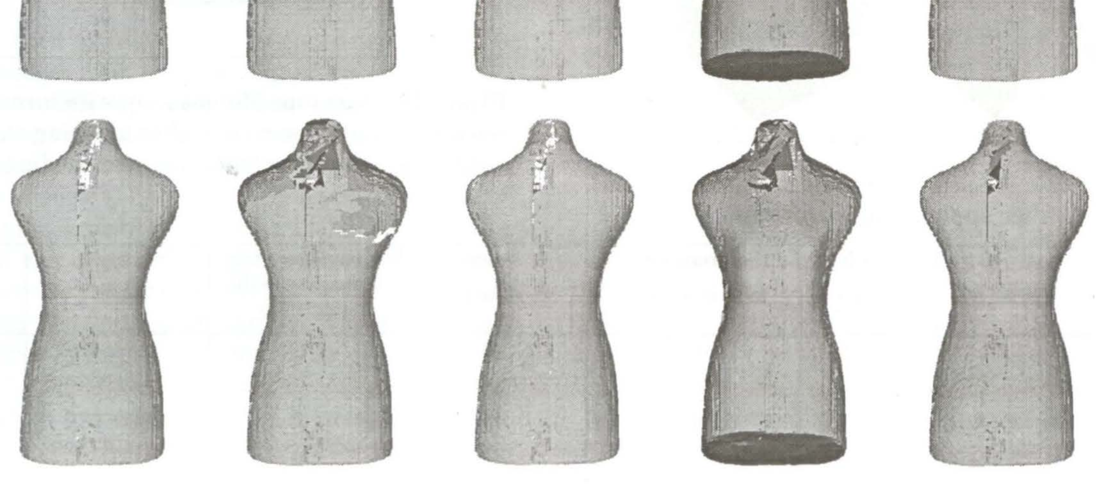

Figure 11: Mannequin (top: front bottom: back) – from left to right: original model, Ivnorm[default], Ivnorm[counterclockwise], Ivnorm[clockwise], our result.

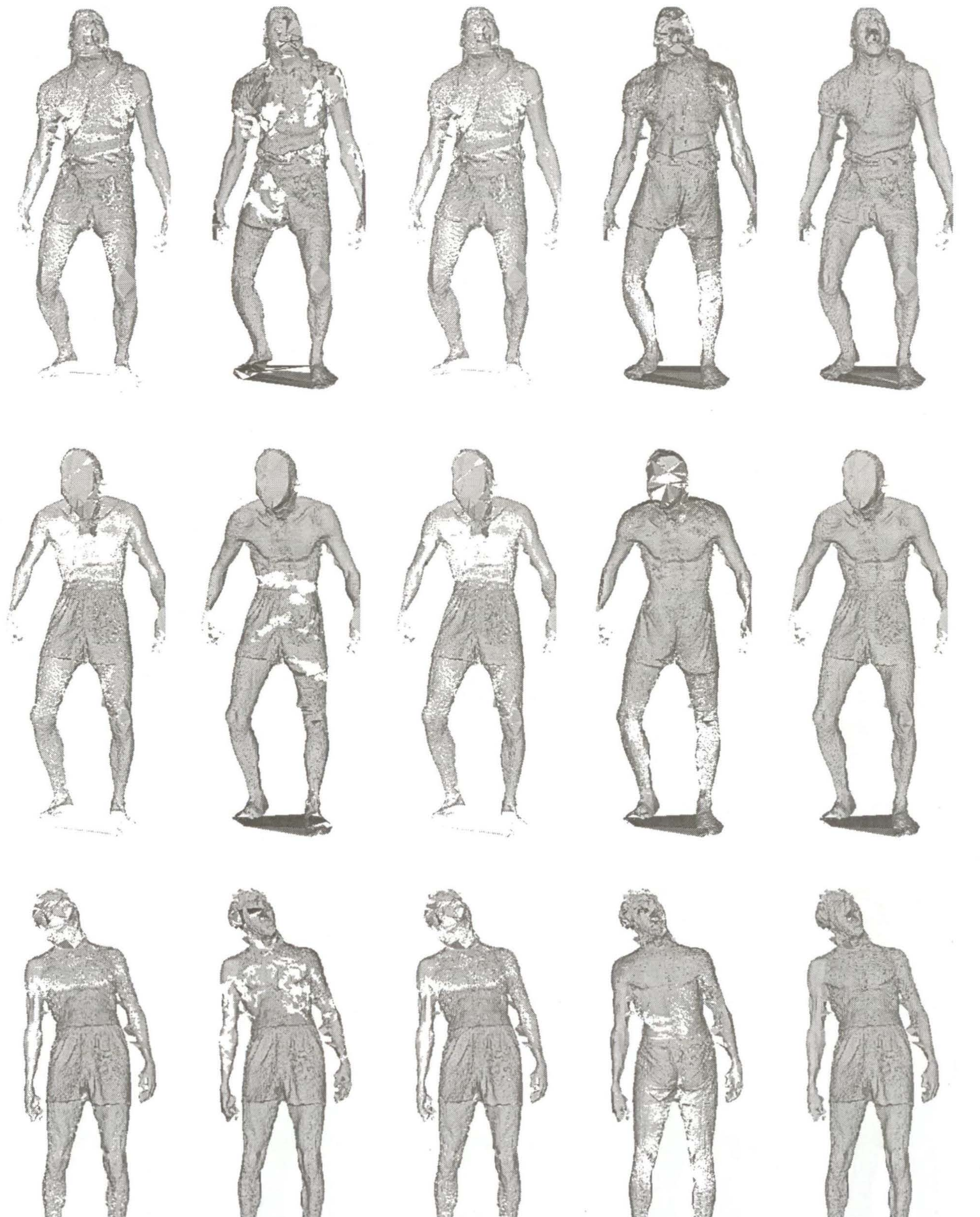

Figure 12: Igor 1, 2, 3 (front) – from left to right: original model, Ivnorm[default], Ivnorm[counterclockwise], Ivnorm[clockwise], our result.

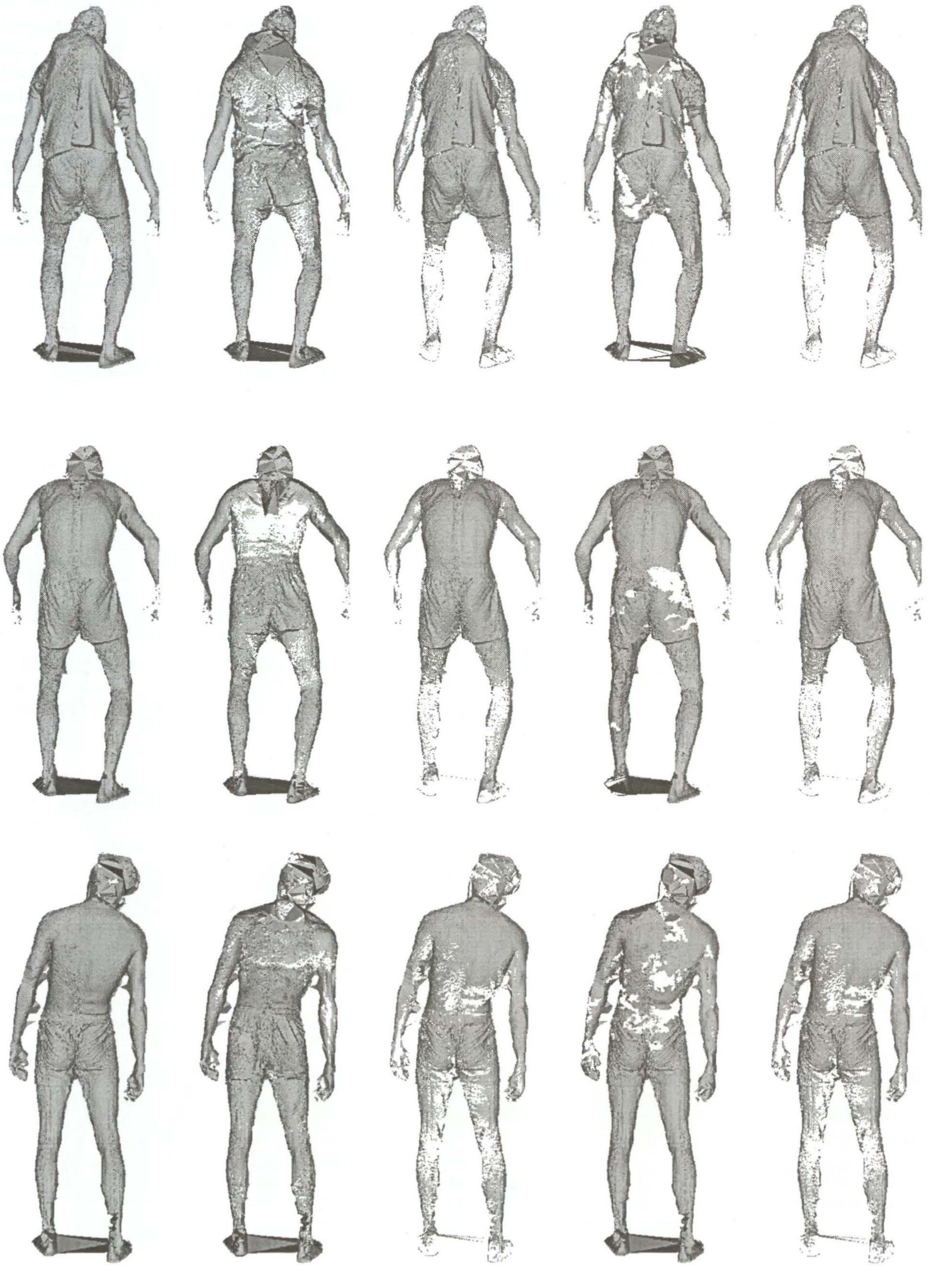

Figure 13: Igor 1, 2, 3 (back) – from left to right: our result, Ivnorm[clockwise], Ivnorm[counterclockwise], Ivnorm[default], original model.