Accurate silhouettes – do polyhedral models suffice?

Chris Heunen

September 2, 2003

Abstract

We investigate whether polyhedral models suffice for accurate silhouette computation. Therefore, we set up a theoretical framework, a mathematical foundation, to compare various algorithms computing silhouettes on spatial and topological accuracy. Within this carefully constructed environment, we can argue on a formal basis why some silhouette-computing algorithms are better than others. In doing so, we distinguish image space algorithms and object space algorithms. We show that only object space algorithms that respect topology are accurate enough for general applications.

1 Introduction

Silhouettes are a key part in non-photorealistic rendering [GG99]. They are also important in other applications, like mesh simplification [LE97], collision detection, sweeping, and various graphical user interfaces. Various algorithms have been developed to compute such silhouettes [BE99, HZ00, NM00, RC99, Her99]. Some of them assume the commonly used polyhedral model, where objects are represented by an approximation with (flat) polygons. Especially in real-time rendering applications, where one often sacrifices rendering quality to gain speed [MKT+97, HZ00], the polyhedral model is often the representation of choice. In most cases (of photo-realistic rendering), this model has proven fairly adequate. When computing silhouettes, however, a polyhedral representation of the object might not be sufficient for correct results.

1.1 Contributions

In this article, we investigate if there are any shortcomings to computing silhouettes based on polygonal approximations. Therefore, we examine various techniques, that represent a general underlying method. Each will be discussed on the topological feasibility of the result. In doing so, the main focus will be on the theoretical aspect. We shall argue, on a formal basis, why some algorithms 'work', where others do not – that is, to what extent the algorithms yield the desired results. If any other features like computation time or memory usage stand out, however, we will not neglect to mention them.

1.2 Image vs Object space

First, we should make a clear distinction between algorithms that operate in object space (section 4), and ones that operate in image space (section 3); the difference being the moment in the pipeline when the algorithm is applied. Object space algorithms work directly in the 3-dimensional space of the model – they try to squeeze every bit of information out, and yield full-fledged 3-dimensional silhouette curves. Image space algorithms, in contrast, start *after* projection, in the 2-d pixel array, with an additional z-buffer. As a consequence, their results are also 2-d pixel arrays (plus the unchanged z-buffer)[1].

Since we are working from a theoretical point of view, we analyse the results of both variants against the actual, *unprojected*, three-dimensional outlines. After all, they both have essentially the same information; the depth information lacking in the projection in image space can be obtained by the supplied z-buffer. Moreover, almost every application of computing silhouettes, except perhaps non-photorealistic rendering, requires 3-d silhouette information.

1.3 Structure of this paper

Before we can compare the various algorithms in a scientific manner, we need to acquire the means to formalize the comparison. This basis is established in section 2, and sets the stage for the rest of the article. We then successively consider image space algorithms in section 3 and object space algorithms in section 4. Finally we draw conclusions in section 5.

2 Mise-en-scène

In this section, we set up a mathematical basis, a theoretical framework. In the rest of this paper, this framework will be used to make an in-depth analysis of various algorithms. This framework regards the 'real' surface (2.1), a polygonal approximation (2.2), and silhouettes of both the surface and the approximation (2.3). We define an error metric to compare approximations to the actual silhouettes in 2.4, and conclude with a discussion about their topology in 2.5.

[1] Image space algorithms usually rely heavily on their input being discrete pixels, since they often use edge detection. A 'projection space algorithm' has not yet been tried to our knowledge, but it would fail probably.

2.1 Underlying surface

If we want to be able to compare any approximation, we should first consider the ideal. In our case this is the underlying surface, as one usually wants to depict real-world, or at least, continuous, objects. In order to avoid anomalies, we assume this to be a fairly 'civilized' surface. That is, the *surface* S we will look at has a piecewise continuous differentiable *parametrization* $f : I \times J \to \mathbb{R}^3$, with I and J intervals in $\mathbb{R}$: the domain $I \times J$ can be split into a finite number of subdomains $I_i \times J_i$, on which f is a C^1 function. Thus, S consists of a finite union of smooth patches. Furthermore, if S is closed, we choose the *surface normal* $\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}$ always to point 'outside' (and we call this normal ∇S from now on).

Notice that this is not a serious limitation; all objects one would usually like to model have this property. A trivial example is the torus with inner radius R and outer radius r, with parametrization $f : [0, 2\pi) \times [0, 2\pi) \to \mathbb{R}^3$, $(u, v) \mapsto (R\cos u + r\cos u\cos v, R\sin u + r\sin u\cos v, r\sin v)$.

Only in scientific visualization this could sometimes be a problem[2]. Fractals, for instance, do not belong to this class, but then again, they are not visualisable on a finite medium anyway. In general, even in scientific applications, the functions under consideration are usually continuous. However, 'degenerate cases' like the Möbius band and the Klein Bottle are excluded by this definition because they have no orientation and self-intersect.

2.2 Polygonal approximation

If a polygon is contained in a supporting plane $Ax + By + Cz + D = 0$ then we call the vector (A, B, C) that polygon's *normal*. A *polyhedral model* S_i is a finite union of connected (flat) polygons (see figure 1). Let f_i be the (piecewise linear) parametrization of S_i.

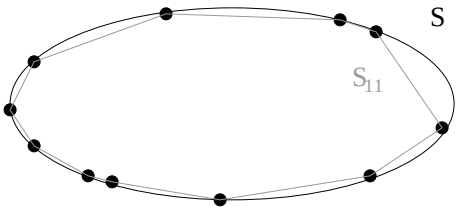


Figure 1: 2-dimensional illustration of a (grey) polygonal approximation S_{11} with 11 vertices to a (black) surface S

We could define some error metric E between the surface S and a polygonal approximation S_i now, but for our purpose it is only important that one exists. (The quadric and the metric measure are popular ones [EDD+95]).

Henceforth, $S_1, S_2, \ldots$ will be a row of polygonal approximations of S, to an arbitrary degree of accuracy[3] – i.e. $\lim_{n\to\infty} E(S, S_n) = 0$.

2.3 Outlines

Now we are ready to define *silhouettes*. Although silhouettes are a highly intuitive concept, a rigid definition exists, that stems from optics.

2.3.1 Precomputation

Besides silhouettes, more types of *outlines* are usually considered, typically also *boundaries* and *creases* (see figure 3). We will not go into detail here, but intuitively, boundaries occur where the parametrization f 'ends', and a crease is a region where the surface normal ∇S abruptly changes. But boundaries and creases can be precomputed – they do not change as the object is transformed (i.e. are invariant under rotation, translation, scaling, and even projection). So they are of little interest here. One could simply check all polygon edges: if it is not adjacent to any other polygon, it is a boundary, and if it is shared by exactly two polygons whose dihedral angle is above the threshold, it is a crease [BE99]. Instead, we focus on the silhouettes.

2.3.2 Silhouettes

A *silhouette point* is a point $\mathbf{p}$[4] on the surface S where the angle between the surface normal $\nabla S(\mathbf{p})$ and the *view vector* from the camera to the point is 90 degrees (see figure 2). Thus, if the camera is at $\mathbf{c}$, then $\mathbf{p}$ is a silhouette point if and only if $\langle \nabla S(\mathbf{p}), \mathbf{p} - \mathbf{c} \rangle = 0$. We assumed S to be C^1, so the Implicit Function Theorem guarantees us that all silhouette points lie on a finite union of curves. These curves we call the *silhouettes*. In a polygonal approximation, a silhouette is an edge between a front-facing and a back-facing polygon. That is, an edge between one polygon with $\langle (A, B, C), \mathbf{p} - \mathbf{c} \rangle \geq 0$, and one polygon with $\langle (A, B, C), \mathbf{p} - \mathbf{c} \rangle \leq 0$, (A, B, C) being the normal of a polygon. When the inner product equals 0, the entire polygon consists of silhouette points (although the difference between an entire silhouette polygon and a single silhouette point does not show when seen from $\mathbf{c}$).

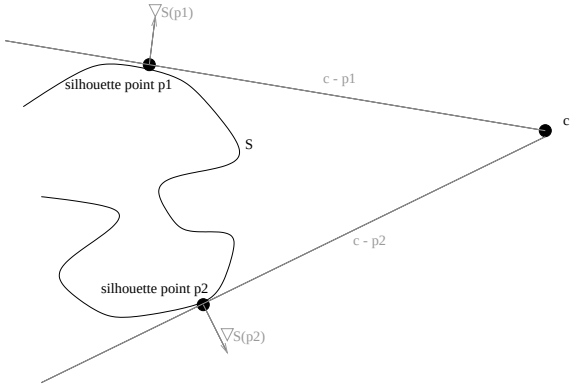


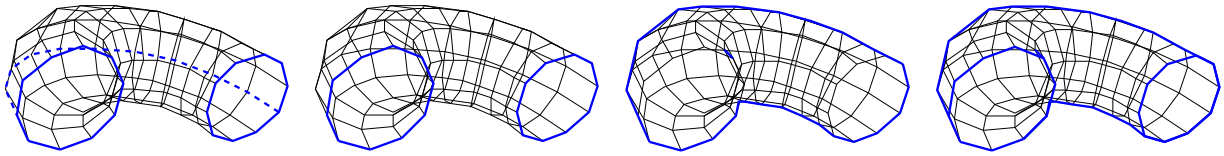
Figure 2: At a silhouette point, the view vector is perpendicular to the surface normal

Because they are defined in terms of the camera, silhouettes are view-dependent. Therefore they need to be recomputed every time the camera or the surface changes.

[2] When considering an implicit surface this is also problematic, but that is a field of study in itself [LC87].

[3] Though it is not necessary, for $j > i$, S_j usually has more edges (and vertices) than S_i. Note that even in the special case where f (of S) is the graph of a function of x and y, it is a NP-hard problem to decide whether an approximation with i vertices exists [HDD+93]. This is not a problem though, since any number of vertices will do for our purposes.

[4] We use the convention that bold faced letters denote vectors.



(a) (b) (c) (d)

Figure 3: On a capped half torus: (a) boundaries, (b) creases, (c) silhouettes, and (d) all outlines. Note that in (a), the parametrization is closed in one parameter only, so the dashed line is no boundary.

2.4 Error metric

Say the silhouettes of S, as seen from $\mathbf{c}$, are given by curves $s_1, \ldots, s_n : \mathbb{R} \to \mathbb{R}^3$. An algorithm computing (an approximation to) the silhouettes of S (from a particular viewpoint) yields a set of curves $\widetilde{s}_1, \ldots, \widetilde{s}_k$. The main question is whether $s_1, \ldots, s_n$ and $\widetilde{s}_1, \ldots, \widetilde{s}_k$ are equivalent (that is, whether $\bigcup_{i=1}^{n}\{s_i(x) : x \in \mathrm{Dom}_{s_i}\} = \bigcup_{i=1}^{k}\{\widetilde{s}_i(x) : x \in \mathrm{Dom}_{\widetilde{s}_i}\}$). If not, is it possible to compute the $\widetilde{s}_i$ so that they at least lie arbitrarily close to the s_i?

But how do we compare the $\widetilde{s}_i$ to the s_i? Well, since we are working in $\mathbb{R}^3$, we already have the Euclidean distance:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\langle \mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y} \rangle}$$

So, a natural way to define the distance d between a point $\mathbf{p}$ and a curve $s : I \to \mathbb{R}^3$ would be to take the distance between that $\mathbf{p}$ and the closest point on s:

$$d(\mathbf{p}, s) = \inf\{d(\mathbf{p}, s(t)) : t \in I\}$$

Using this, we can compare an entire curve $\widetilde{s} : \widetilde{I} \to \mathbb{R}^3$ to s:

$$d(\widetilde{s}, s) = \sup\{d(\widetilde{s}(t), s) : t \in \widetilde{I}\}$$

This formula is known as the one-sided Haussdorf distance [KLS96], because it is generally (when defined for unrestricted sets) not symmetric. But when defined for curves, it is indeed a metric[5]. As the metric for the deviation of an approximation to the actual surface we define

$$\delta((\widetilde{s}_1, \ldots, \widetilde{s}_k), (s_1, \ldots, s_n)) = \sum_{i=1}^{k} \min\{d(\widetilde{s}_i, s_j) : 1 \le j \le n\}$$

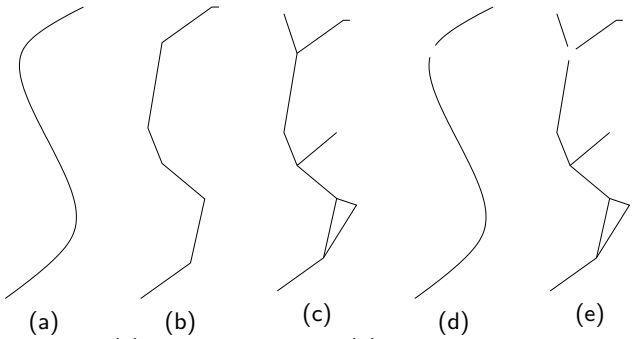
which we abbreviate to $\delta(\widetilde{\mathbf{s}}, \mathbf{s})$. So, the Euclidean distance leads to a (very natural) error metric δ that we can apply to the results of the various algorithms, to see how well they approximate the actual silhouettes.

Notice that we have set up δ to compare curves in $\mathbb{R}^3$, but we have never actually used that the dimension of this $\mathbb{R}$-vectorspace is 3. We could interpret δ as a metric on curves in $\mathbb{R}^2$ just as well.

2.5 Topology

Approximation in a spatial sense is nice. But if the approximation $\tilde{\mathbf{s}}$ fundamentally differs in structure from the actual silhouette $\mathbf{s}$, we are not content. Small structural differences might still considerably distort the computation based on the approximation $\tilde{\mathbf{s}}$. Therefore it seems worthwhile to compare the topology of $\tilde{\mathbf{s}}$ to $\mathbf{s}$, to see if any peculiarities are introduced in the approximation $\tilde{\mathbf{s}}$, which did not exist in the $\mathbf{s}$.

The types of topologies of approximations to the actual silhouettes (figure 4(a)) we will encounter are limited: they are either correct (figure 4(b)) or have 'crossroads' (see figure 4(c)). Intuitively, these are clearly not topologically equivalent. This intuition is easily proved using the Component Theorem: if topological spaces are equivalent, leaving out any point in either one should result in the same number of components as leaving it out in the other.



(a) (b) (c) (d) (e)

Figure 4: (a) Actual silhouettes, (b) approximation with the same topology as (a), (c) approximation with a different topology as (a): (d) shows that leaving out a point in (a) yields 2 components, whereas in (e) the same action can yield 3 components[6].

3 Image space algorithms

The easiest way to compute silhouettes is perhaps in image space, using existing graphics packages to do all the hard work. By rendering images in different ways and then post-processing, an image space algorithm can produce fairly convincing results quickly. We shortly explain how image space algorithms work, considering the two major features of renderers used, namely the z-buffer in 3.1, and the so-called *normal-buffer* in 3.2. In 3.3 we carry out the analysis which shows if image space algorithms yield feasible results.

3.1 Using depth

Most renderers (like OpenGL) generate a z-buffer, where the intensity of every pixel represents the relative depth of that point in the model. This can be used to detect silhouettes. The idea behind using the z-buffer is that the variation in depth between adjacent pixels is usually small over

[5] A metric in the space of *curves* in $\mathbb{R}^3$, that is.

[6] If the point that was left out was on a 'loop', rather than a 'crossroad', the result will be 1 component, which is not equivalent either.

a continuous surface, whereas it is large between surfaces. Thus, one can detect C^0 discontinuities, i.e. silhouettes, by applying an edge-detection filter on the z-buffer (usually a Sobel filter, like in [ST90]). Algorithms implementing this approach include [Her99, Cur98, NM00, RC99].

3.2 Using surface normal

Instead of interpolating depth, resulting in a z-buffer, one could also interpolate the surface (polygon) normal, resulting in a *normal-buffer* [Her99]. The edges in this normal-buffer now correspond with changes in surface orientation, i.e. C^1 discontinuities. Augmenting the edges extracted from the z-buffer with these edges from the normal-buffer results in detection of C^0 and C^1 discontinuities, that is, silhouettes and creases.

Raskar and Cohen [RC99] further note that only the first two layers of (visible) polygons are needed and utilizes that to gain speed. But the principle limitations drawn in the following still apply. So, we will not elaborate this in more detail.

3.3 Accuracy

Let us have a look at what happens if we look upon an algorithm as in 3.1 and 3.2 using the mathematical framework we set up in section 2. As mentioned in section 1.2, a fair comparison of algorithms is only made in 3 dimensions. However, since image space algorithms are mainly used only to proceed with resulting projected silhouettes (for example in photo-realistic rendering), we also consider the behavior in 2 dimensions.

3.3.1 In 3 dimensions

Suppose we are rendering a picture of resolution $n \times n$, mapped onto $[0,1] \times [0,1] \subset \mathbb{R}^2$, and for arguments sake, assume the edge detection process of 3.2 really yields silhouettes. If we had some way of connecting neighboring 'edge pixels' into curves ([NM00] and [Cur98] provide such a method), one might be tempted to let $n \to \infty$, using that the distance between a pixel's midpoint and its border is at most $\frac{\sqrt{2}}{2n}$, to get

$$\delta(s, \widetilde{s}) \leq \sum_{i=1}^{k} \delta(s, \widetilde{s}_i) \leq \sum_{i=1}^{k} \frac{\sqrt{2}}{2n} = \sqrt{2}\frac{k}{2n}$$

where k is the number of 'black pixels' in the edge map (see figure 5(a)). (After all, the deviation of any curve connecting two 'black pixels' from the actual curve must still be within those pixels). This converges to 0, since $k \ll n^2$ because the actual silhouettes are infinitesimally thin compared to the plane as $n \to \infty$. So, $\delta(s, \widetilde{s}) \leq \varepsilon$ for an arbitrary ε.

Using the z-buffer, one could then back-project the curves found to $\mathbb{R}^3$. This back-projection is error-prone, but even if it would be perfect, we have lost too much information along the way. Consider, for example, the surface parameterized by $f(u,v) = (u, v\sin v, v)$, $0 \leq v \leq 4\pi$, as seen from $\mathbf{0}$ (see figure 5(b)). An image space algorithm could never detect the two overlapping silhouette points, because they are projected on the same pixel. For 3-dimensional output computations, image space algorithms just start from too little input.

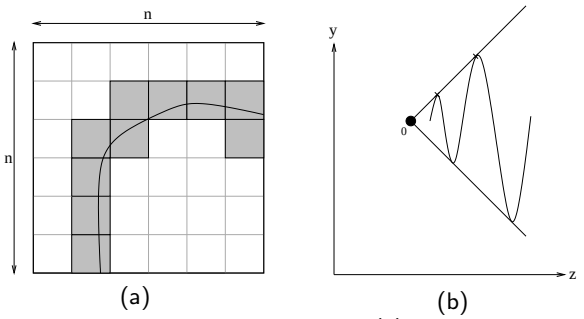


Figure 5: Image space algorithms – (a) naively achieving accuracy using rendering resolution, (b) a situation where image space algorithms are doomed to fail

3.3.2 In 2 dimensions

When we compare the curves with the metric that our δ induces on $\mathbb{R}^2$ (see 2.4), however, things work out better. Since S_n approximates S to an arbitrary ε, so do the projections of $\tilde{\mathbf{s}}$ and $\mathbf{s}$. So, image space algorithms are fine, when 2-dimensional output is used only, i.e. when all you care about is the projection, whose topology might not be pleasing – since it is completely ignored, the topology of the output might well be different from the actual curves. Moreover, there is no way at all to inspect topology in the projection (let alone correct), since overlapping (intersections of) silhouettes, which are perfectly 'legal' can look exactly the same as a 'crossroad' (see 2.5).

Apart from that, the method of cranking up the rendering resolution is of course extremely crude. Most renderers are not designed to render beyond a certain range of resolutions. Moreover, computation time rises at least quadratically with the resolution.

4 Object space algorithms

In an object space algorithm, in contrast to image space algorithms, one wants to build up the $\widetilde{s}_i$ directly in the 3-dimensional space from the s_i. The simplest approach to computing silhouette curves would be to replace the smooth surface by a polygonal approximation, and find the silhouette edges of that. However, there can be significant differences between using S and S_i (as described in 2.5). We discuss this widely adopted method in section 4.1. In section 4.2 we look at a more sophisticated method, and we see that this technique is always topologically pleasing. Finally, we consider some methods to speed up the process of finding silhouettes in object space in section 4.3.

4.1 Brute force

Silhouette edges are simple to find in a polygonal approximation, as elaborated in 2.3.2. We simply iterate over *all* edges in the model, and look at the polygons adjacent to each edge. Consider two of those two polygons, and say they have normals $\mathbf{n}_1$ and $\mathbf{n}_2$. We recapitulate from section

2.3.2: in this case the edge is a silhouette edge is if and only if $\langle \mathbf{n}_1, \mathbf{n}_2 \rangle \leq 0$.

The silhouette edges thus found are contained within the same polygons that also contain the actual silhouettes. So, for every $\tilde{s}_i : \tilde{I}_i \to \mathbb{R}^3$, we have $\delta(\tilde{s}, s) \leq \sum_{i=1}^{k} \min |\tilde{I}_i|^2$. And every polygon's area converges to zero if we take denser approximations. More precisely, because $\lim_{n\to\infty} E(S, S_n) = 0$, also $\lim_{n\to\infty} |\tilde{I}_i| = 0$. And since $x \mapsto x^2$ is continuous, also $|\tilde{I}_i|^2 \to 0$ as $n \to \infty$. So, $0 \leq \lim_{n\to\infty} \delta(\tilde{s}, s) \leq \lim_{n\to\infty} \sum_{i=1}^{k} |\tilde{I}_i|^2 = 0$.

However, there is no basis to claim that this brute force method is also topologically accurate. Since we have not demanded anything about the normals of the polygonal approximation S_n in relation to the normals of the actual surface S, it could be that only one polygon's normal is 'wrong', and its surrounding polygons are 'right'. In that situation, all edges of this 'wrong' polygon will be tagged as silhouette edges. Implementations show that this happens quite frequently. No matter how dense the approximation is, the topology will differ. Instead of a continuous curve, a curve with 'crossroads' (see 2.5) results, a 'twig with branches' if you will (see figure 6: the blue silhouette edges have 'crossroads', whereas the actual, dashed grey, silhouettes do not).

4.2 Interpolating the straightforward

Hertzmann and Zorin [HZ00] use a small refinement of this naive approach, by linearly approximating the silhouettes. Remember that silhouettes are defined as the zero set of $g(\mathbf{p}) = \langle \nabla S(\mathbf{p}), \mathbf{c} - \mathbf{p} \rangle$. Assume the true surface normal $\nabla S(\mathbf{p})$ is known at each vertex $\mathbf{p}$, so $g(\mathbf{p})$ can be computed at each vertex. By linearly interpolating g over all edges, and connecting the resulting points, we obtain the $\widetilde{s}_i$ (see figure 6, the red lines). The $\widetilde{s}_i$ will now consist of line segments inside each polygon of the polygonal approximation. So, following the very same analysis as in 4.1, we see that this method also spatially approaches the actual silhouettes.

Moreover, the $\widetilde{s}_i$ connect points in the interior of the edges of the mesh, and form either closed loops or non-intersecting chains connecting points on the boundaries or creases, similar in structure to the actual silhouette curves.

Thus, the $\widetilde{s}_i$ will have the same topology as the s_i! This guarantees a convincing image of S (but it may of course not accurately reflect features smaller than one polygon).

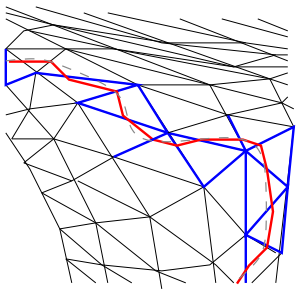


Figure 6: The topology of the non-interpolated silhouettes (blue) differs from the actual silhouettes (dashed grey), whereas the linear interpolation (red) has the same topology as the actual silhouettes.

4.3 Speeding up

Both the methods of 4.1 and 4.2 require a complete traversal of the polygonal model. So, they must have a time complexity of at least $\mathcal{O}(n)$, if n is the number of polygons[7]. By being clever, one can reduce this to $\mathcal{O}(\sqrt{n})$ in at least two ways: by using geometric duality (4.3.1), or via a gaussian map (4.3.2).

4.3.1 Using geometric duality

Hertzmann and Zorin [HZ00] suggest a method to speed up the finding of silhouettes, making use of geometric duality. They carry out the calculation in projective space, rather than in the affine $\mathbb{R}^3$. This way, there is no need to completely traverse the polygonal approximation in order to find silhouettes, but only the polygons that actually contain silhouette lines. As this method does not alter the resulting set, it does not compromise accuracy. If m is the number of silhouettes, this method is $\mathcal{O}(m)$ in time. In a typical setting (i.e. in an average rendering), this is roughly the same as $\mathcal{O}(\sqrt{n})$, where n is the number of polygons.

4.3.2 Via a Gaussian map

Benichou and Elber [BE99] propose a different speedup, effectively based on the same. Their idea is to transform the problem to another context, where a subdivision is possible. First, for all vertices $\mathbf{p}$, $\nabla S(\mathbf{p})$ is projected onto the unit cube (via the Gaussian sphere). Next, the problem is divided into six smaller ones, since a cube has six faces. This method is also $\mathcal{O}(m)$ in time, where m is the number of silhouettes, which is typically $\mathcal{O}(\sqrt{n})$, where n is the number of polygons. This is achieved through an $\mathcal{O}(n)$ precomputation.

However, Benichou and Elber [BE99] yield silhouettes composed from edges, i.e. not interpolated. So, they face the same problem as image space algorithms; namely that the topology of $\tilde{\mathbf{s}}$ differs from that of $\mathbf{s}$. This can be fixed though: interpolation can be done on the unit cubes faces as well.

5 Conclusion

We considered piecewise C^1 surfaces, and have set up a mathematical framework to compare silhouette computation algorithms based merely on their specification. Especially the spatial error metric δ works nicely when reasoning formally about the ('extent of') correctness of algorithms.

Image space algorithms for detecting silhouettes are usually fast and easy, but only yield acceptable results when further calculations are solely based on the projection. Moreover, they may introduce singularities. So, from a topological point of view, image space algorithms are not ac-

[7] This is the same as $\mathcal{O}(n)$, where n is the number of edges, since a triangle has only 3 edges, and every polygon can be triangulated (in linear time or less).

ceptable. Image space algorithms are mostly used for non-photorealistic rendering, where speed and rough results are often more important than accuracy.

When aiming for a $x \times y$ rendering resolution, image space algorithms are typically $\mathcal{O}(xy)$ in time (*after* the time complexity of the rendering itself, over which such methods have no control).

Object space algorithms, in contrast, are more involved, but can yield (very) accurate results. Among object space algorithms, we can distinguish two kinds: those that are topologically correct, and those that might not be. Topological accuracy can be achieved by linear interpolation, as in [HZ00] – without compromising computation time.

Object space algorithms can be improved by clever techniques. If n is the number of polygons in a model, accurate silhouette edges can be computed in $\mathcal{O}(\sqrt{n})$ time. Typically this is significantly slower than image space algorithms, since existing rendering packages are entirely optimized.

All in all, the answer to the title, "Do polyhedral models suffice for accurate silhouettes?", thus is: "Yes, polyhedral models suffice, but only if used clever enough.".

6 Future work

The result of this article can be seen as a form of existential proof. We know now (i.e., we can *prove*) that accurate silhouette computation algorithms exist. The challenge that lies ahead is to create faster ones, while preserving topologically correct results. Knowing that sound methods exist to compute them, silhouettes can be used safely in ever more applications as basic building blocks.

Acknowledgements

The author would like to thank Peter Klok for helping the research get started, Jozef Hooman for the tedious task of spotting small grammatical and logical errors in many iterations, Martijn Grooten for providing a crash course in elemental topology, and Lotte Hollands for valuable discussions on parts of the mathematical content of this article.

References

[BE99] F. Benichou and G. Elber. Output sensitive extraction of silhouettes from polygonal geometry. *Proc. Pacific Graphics*, pages 60–69, 1999.

[BJ98] Thomas F. Banchoff and Ockle Johnson. The normal euler class and singularities of projections for polyhedral surfaces in 4-space. *Topology*, 37(2):419–439, 1998.

[Cur98] C.J. Curtis. Loose and sketchy animation. *Visual proceedings of Siggraph '98*, page 317, 1998.

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics*, 29(Annual Conference Series):173–182, 1995.

[GG99] A. Gooch and B. Gooch. *Non-Photorealistic Rendering, chapter 8: Using Non-Photorealistic Rendering to Communicate Shape*. A.K. Peters, 1999.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. *Computer Graphics*, 27 (Annual Conference Series):19–26, 1993.

[Her99] A. Hertzmann. Introduction to 3d non-photorealistic rendering: Silhouettes and outlines. *SIGGRAPH '99 Course Notes*, 1999.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 517–526. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[KLS96] Reinhard Klein, Gunther Liebich, and W. Strasser. Mesh reduction with error control. *Proceedings of Visualisation '96*, pages 311–316, 1996.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, (4):163–169, July 1987.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31(Annual Conference Series):199–208, 1997.

[MKT+97] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Computer Graphics*, 31(Annual Conference Series):415–420, 1997.

[NM00] J.D. Northrup and L. Markosian. Artistic silhouettes: A hybrid approach. *ACM SIGGRAPH*, 2000.

[RC99] Ramesh Raskar and Michael Cohen. Image precision silhouette edges. *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[ST90] T. Saito and T. Takahashi. comprehensible rendering of 3-d shapes. *Proceedings of SIGGRAPH '90 (Dalles, Texas, August 6–10)*, 24(4):197–206, 1990.

[Whi55] H. Whitney. On singularities of mappings of euclidean spaces. *Annals of Mathematics*, pages 374–410, 1955.