Smoothing the Silhouettes of Polyhedral Meshes by Boundary Curve Interpolation

Wu Sing On
31st August, 1999

Supervisor:
Dr. Wenping Wang

Department of Computer Science and Information Systems,
Faculty of Engineering,
The University of Hong Kong,
Pokfulam Road,
Hong Kong.

Submitted for the degree of Master of Philosophy
at the University of Hong Kong in August, 1999

Acknowledgement

I want to thank my supervisor Dr. Wenping Wang, who gave me the opportunity to study as a research student and finish this work. He has been giving me invaluable guidance since I was an undergraduate student. His advice on this work is crucial and many of the ideas are indeed owed much to him.

I would also like to thank C. H. Poon, who is also a research student of Dr. Wang. He stayed in the Graphics Lab late at night like I did and made me feel better when I encountered some obscure programming bugs and was crazy debugging since I had someone to talk to at least. His suggestions are quite useful also. I also want to thank K. H. Yeung, who is another student of Dr. Wang. We had discussed briefly on some implementation stuff and he gave some useful ideas.

Although I do not know him personally, I also want to thank Mr. Brian Paul for his excellent work on writing Mesa and making its source code publicly available, so that I can have something to base on instead of writing all the code from scratch. It is really incredible to be able to implement such a 3D graphics library.

I am also grateful to Gracelyn Chan, who virtually took care of my everyday life in those days when I did not know whether it was day or night, and whether I should take breakfast or dinner, but just knew I had to write, write and write.

1. Introduction

Efficient and accurate rendering of smooth curved surfaces[1] is a fundamental problem in computer graphics. It is because much of computer graphics is concerned with modeling the real world, and many real-world objects are inherently smooth on a certain scale [FOLE90]. For interactive computer graphics applications nowadays such as multimedia, computer animation, virtual reality, computer games, and so on, we not only want to render smooth surfaces such that they look like their real counterparts as much as possible, but we also want to do it as fast as possible.

To render a smooth surface for the purpose of interactive display, it is now almost a standard practice to first obtain a polyhedral mesh, which is a set of polygons, typically triangles, as an approximate representation of the curved surface, and then pass the vertex data including the vertex coordinates, normal vectors, color values, etc., to the graphics sub-system, which is then responsible for rendering the mesh as a 2D image. The reason for using polyhedral approximations for curved surfaces is that commercial graphics hardware, which is dedicated to rendering smoothly shaded polygons efficiently, is now commonly available. This kind of hardware is virtually a hardware implementation of the standard scan-line algorithm[2] integrated with Gouraud shading [GOUR71], together with texture mapping and more recently, even bump mapping. Workstations or personal computers equipped with such graphics hardware can render smoothly shaded polygons at a very high speed. This is arguably the primary reason for the computer graphics community to represent every 3D objects they want to render by polygons, or a polyhedral mesh. Even though people do not have access to graphics hardware, they would still like to use polygons and the scan-line algorithm implemented in software since the algorithm itself is fast and simple. Though techniques for rendering curves surfaces other than using polyhedral approximations and the scan-line algorithm do exist, no one seems to bother to use them for interactive computer graphics applications, probably due to their inherent complexity and inefficiency. It is almost certain that polyhedral meshes will continue to be used for representing smooth curved surfaces in computer graphics.

[1] We will use the terms smooth curved surface, smooth surface and curved surface interchangeably.
[2] [FOLE90] gives a detailed account and references for the scan-line algorithm and the various versions of it.

However, using polyhedral meshes to represent curves surfaces for rendering is not without its problems. A polyhedral mesh rendered may not look as faithful to the curved surface it represents as one might expect. One important criterion for the mesh to look 'realistic' is the smoothness of its shading. A curved surface should have a continuous, smooth shading. Simply shading the polygons individually as they are obviously does not work, since it gives a faceted appearance instead of a smooth one. Fortunately, much effort was devoted to smoothly shading a polyhedral mesh to simulate a smooth surface, while still using the fast and simple scan-line algorithm. This is the well-known Gouraud shading [GOUR71] and Phong [BUIT75] shading methods. These methods give a smoothly shaded appearance to a polyhedral mesh so that it looks like a curved surface in terms of shading, though they both suffer more or less from the well-known Mach band effect[3], especially Gouraud shading. Phong shading in general is superior in quality to Gouraud shading, but at the same time computationally more expensive. This is probably the reason why only Gouraud shading is commonly available in commercial graphics hardware.

Another important criterion for a rendered polyhedral mesh to look like a curved surface is the smoothness of its silhouette. It is a well-known problem that no matter how realistic the shading of a polyhedral mesh is, the mesh exhibits a conspicuously polygonal silhouette. Let us quote a sentence from page 739 of [FOLE90]:

> "*No matter how good an approximation an interpolated shading model offers to the actual shading of a curved surface, the silhouette edge of the mesh is still clearly polygonal.*"

[VANO97a] and [VANO97b] also have comments on the silhouette problem of polyhedral models:

> "*...using the interpolated normal vectors in the shading computations, yields a smoothly varying intensity distribution. There is an inherent mismatch, however, between the smoothness of the shading thus achieved and the non-smoothness of the geometry which is*

[3] Again [FOLE90] gives an account and reference for the Mach band effect.

particularly visible at silhouettes, showing as straight edges and non-smooth edge junctions at the silhouette vertices."

"*Images that are generated by means of the traditional Phong shading algorithm typically look very good provided the polygonal mesh is sufficiently dense. For larger polygons, several conspicuous artifacts may arise. The silhouette edge problem is probably the most notorious one, ...*"

Although the problem is well-known in the computer graphics community, it seems that little has been done to tackle the problem directly. The most obvious and most applied technique so far is to approximate a curved surface with a huge number of tiny polygons so as to produce a smooth silhouette. Using this method, if we want to have a silhouette guaranteed smooth, we virtually have to ensure that each polygon is smaller in size than a pixel on the computer screen. For an object which occupies a significant portion of the screen, we obviously need a really large number of polygons. This will increase the expense of rendering such an object both in terms of storage and computation. The latter is especially undesirable since the rendering time would become too long for interactive display. One more thing to note about the method of using more polygons is that shading is also improved.

Our work is concerned with tackling the problem of silhouette directly, while leaving shading alone. We have devised an algorithm which tries to solve the problem without increasing the number of polygons to render, thereby avoiding increasing the rendering time by a large extent.

We will first have a brief review of some related work (**2. Related Work**), followed by an account of the background and inspiration which lead us to the work presented here (**3. Background and Inspiration**). Then we will give an overview of the essential steps of our algorithm (**4. Overview of the Algorithm**), after which we will have detailed descriptions and explanations of the various steps of our algorithm (**5. Identification of Silhouette Polygons**, **6. Perturbation of Silhouette Vertices**, and **7. Scan-Conversion of Silhouette Polygons**), which form the major part of this writing. What follows is a discussion on issues related to the implementation of our algorithm (**8. Implementation Issues**). After this, we will try to examine the quality

(**9. Quality**) and performance (**10. Performance**) of the algorithm. The final part contains some suggestions on the future directions of this work (**11. Future Work**) and a conclusion (**12. Conclusion**).

2. Related Work

Besides the rather straightforward approach of using a large number of polygons to approximate a curved surface so as to obtain a smooth silhouette, there is some earlier work on scan-converting curved surfaces directly. Even if the method of using many polygons is to be used, problems arise from how to subdivide the original polyhedral mesh into more polygons if the underlying curved surface is unknown.

2.1 Direct Scan-Conversion of Curved Surfaces

Three scan-line methods for displaying parametrically defined surfaces are presented in [LANE80]. Parametrically defined surfaces are those defined by the function $\mathbf{x}(u, v) = (x(u, v), y(u, v), z(u, v))$. Scan-line algorithms can readily be applied to polygons because edges can be tracked as functions of y. Parametrically defined surfaces do not have this property. Unlike polygons, edges not only occur on the boundary, but also on the silhouette. (For polygonal objects any silhouette edge is also a boundary edge.) Two of the authors, Blinn and Whitted track edges by finding all the intersections of a scan line with the boundary and silhouette edges of a surface patch, so as to do scan-conversion. This inevitably involves the use of numerical root-finding methods like Newton-Raphson iteration, which is used by the authors. Efficiency cannot be expected from this kind of numerical iteration method and sometimes the roots cannot be found for some special cases. Whitted's algorithm cannot handle certain silhouette edges properly. Nevertheless, most models of real objects are well-behaved and their algorithms do a good job of rendering them, regardless of efficiency. As they deal with the parametric representations of surfaces directly, polygonal artifacts cannot appear and the silhouette is guaranteed smooth.

The other two authors, Lane and Carpenter perform an adaptive subdivision of each surface patch until each patch is within a set tolerance of being a planar quadrilateral, which can then be scan-converted with an ordinary polygon scan-conversion algorithm. The tolerance normally depends on the resolution of the computer screen so unnecessary subdivisions are avoided. If the tolerance is set

appropriately, this algorithm also produces good looking images of curved surfaces with smooth silhouettes.

Although this kind of algorithm shares more or less the same goal as we do, they expect a mathematically well-defined surface as input, whereas we would like to accept a polyhedral mesh as input and render it so that it looks like a curved surface. Also, their use of numerical root-finding methods may pose significant efficiency problems for the purpose of interactive display.

2.2 Polygon Subdivision of Polyhedral Meshes

A method of subdividing an existing polyhedral mesh into one with more polygons is presented in [VANO97a]. This essentially can be considered as a reverse process of mesh simplification. This method makes a minimal assumption on the input polyhedral mesh: the input data comprises vertex coordinates and vertex normal vectors, and nothing else. In particular, the algorithm does not require any neighborhood information, in contrast to other subdivision algorithms and those mesh simplification algorithms. For each polygon, the authors use the vertex coordinates and normal vectors to construct a 4^{th}-degree triangular Bezier surface patch, which essentially replaces the polygon. This patch can easily be subdivided into several smaller polygons, with some new normal vectors which come smoothly between the original ones as they are obtained from the Bezier patch. The whole process can be applied recursively to produce more Bezier surface patches and therefore more polygons. The algorithm is a preprocessing step prior to rendering: it just produces a mesh with more polygons for the graphics sub-system to render. If the number of polygons is sufficiently large, the silhouette of a rendered mesh will be smooth. Also, as the new polygons and normal vectors are obtained from the original mesh, the new mesh with more polygons will look faithful to the original one.

3. Background and Inspiration

As mentioned earlier, rendered images of polyhedral meshes do not look faithful to the curves surfaces the meshes are supposed to represent mainly because of two factors: non-smoothness of shading (the Mach band effect, in particular) and polygonal silhouettes. The most common way to improve the appearance of polyhedral meshes so that they look more like curve surfaces is to use more polygons.

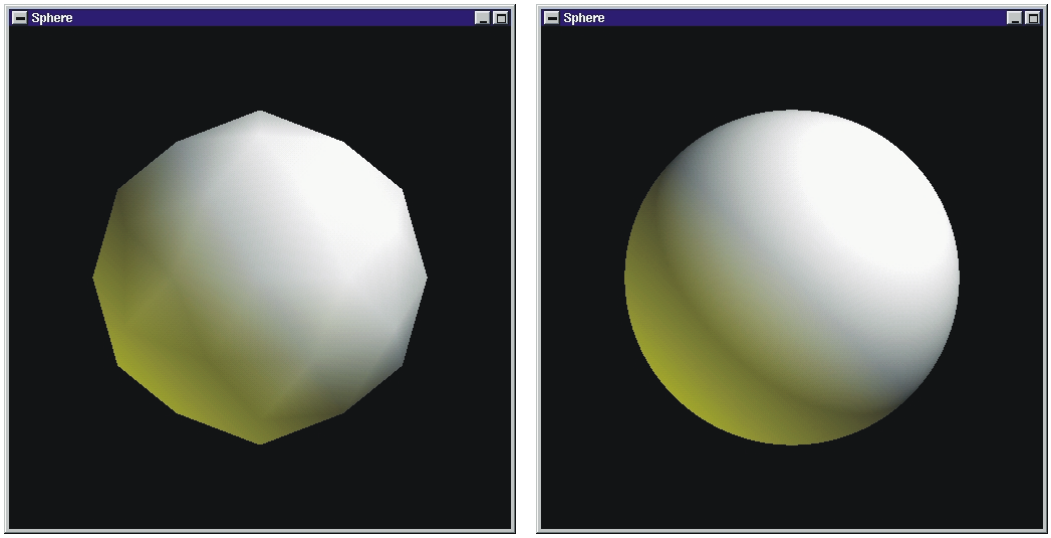


Figure 1

This way virtually improves the quality of both shading and silhouette, and if we want to guarantee the silhouette to be smooth, we must ensure that each polygon is smaller than a pixel.

Our work is mainly inspired by the observation that when we keep increasing the number of polygons of a polyhedral mesh so as to improve its resemblance to a curved surface, after the polygons has become sufficiently numerous, we are just improving the smoothness of the silhouette, without much apparent improvement in shading. Empirically speaking, shading has been quite good already with fewer polygons (see Figure 1). This observation naturally leads to the idea that increasing the number of polygons just to improve the silhouette is somewhat overkill. We would like to tackle the problem of polygonal silhouettes independent of shading, and independent of the number of polygons a polyhedral mesh has. That means no matter how good or how bad the quality of shading is, and no matter how many and how few polygons a polyhedral mesh has, we want to make the silhouette look smooth. The

number of polygons certainly still matters in terms of shading, for if we have to restrict ourselves to either Gouraud shading or Phong shading or whatever other interpolated shading, the only way for us to improve the smoothness of shading is to use more polygons. Our work also implies that the silhouette and shading are improved separately from each other, unlike what is commonly being done now. How to improve the quality of shading is probably another important topic, but we would like to leave it alone for the time being. We would like to concentrate on the silhouette: render the silhouette such that it appears as a smooth, closed curve rather than a polygon. Our algorithm achieves this by interpolating the vertices on the silhouette, together with their normal vectors, which must be provided for shading anyway, to obtain a smooth curve (one with G1 continuity).

4. Overview of the Algorithm

The major steps of our method of rendering a polyhedral mesh such that its silhouette is guaranteed smooth are given below:

1. Identify silhouette polygons

We define a *silhouette polygon* as a polygon which is visible (front-facing) and have at least one neighbor which is invisible (back-facing), where a neighbor of a polygon is defined as another polygon which shares an edge with it. An edge shared by an invisible neighbor is called a *silhouette edge*, of which the two vertices are called *silhouette vertices*. We must first of all distinguish silhouette polygons from other 'interior' polygons. In this step we also identify silhouette edges and silhouette vertices, but for convenience we refer generally to this step as 'identify silhouette polygons' or 'identification of silhouette polygons'.

2. Perturb the silhouette vertices

This essentially means that we modify the position of the silhouette vertices in the 3D space, or more specifically, in the eye coordinates using the OpenGL terminology, if necessary, so that they lie in the real silhouette of the curved surface that the polyhedral mesh represents. The necessity of this step may seem hardly comprehensible at first glance and a detailed explanation on this will be given later.

3. Scan-convert the silhouette polygons in the new way

For each silhouette polygon, we scan-convert it in such a way that its silhouette edge(s) appear(s) appropriately curved rather than straight. The curve is obtained by interpolation of the silhouette vertices with their (projected) normal vectors. The interpolation is done in 2D, or the window coordinates using the OpenGL terminology. The final result will be that straight silhouette edges are replaced by curve segments which join together with each other with G1 continuity and the silhouette of the rendered polyhedral mesh will be a closed, smooth curve rather than a polygon.

4. Scan-convert the other polygons in the conventional way

For those non-silhouette polygons, we just scan-convert them using the conventional scan-line algorithm since they have nothing to do with the silhouette. Obviously this step does not necessarily come before or after the step just described above. The process of rendering silhouette and that of non-silhouette polygons can certainly be interleaved with each other.

5. Identification of Silhouette Polygons

In general, polygons which are front-facing but have at least a neighbor which is back-facing are considered silhouette polygons (see Figure 2). Conceptually, we can determine whether a polygon is a silhouette polygon or not by just examining the polygon itself and its neighbors. For example, see Figure 3, where the shaded triangle is shown with its three neighbors. If $\Delta V_0V_1V_2$ is back-facing, then it is not a silhouette

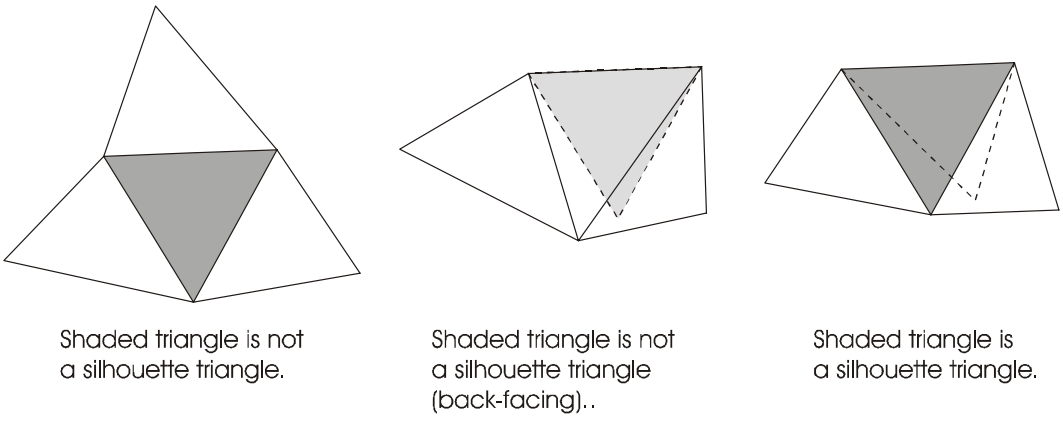


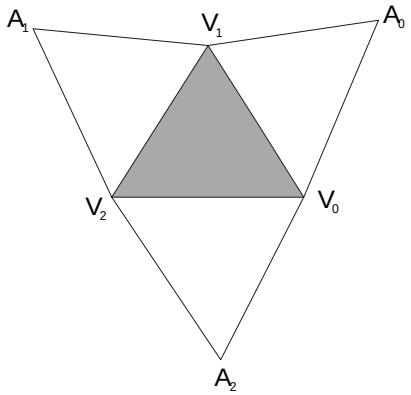
Figure 2

triangle. Otherwise (i.e., $\Delta V_0V_1V_2$ is front-facing), if, say, $\Delta V_0A_0V_1$ is back-facing,

then $\Delta V_0V_1V_2$ is a silhouette triangle. Moreover, we know that V_0V_1 is a silhouette edge and V_0, V_1 are silhouette vertices.

5.1 Technicality

To summarize the scheme of determining whether a triangle is a silhouette triangle or not, we want to define something called an *auxiliary vertex*. A triangle $V_0V_1V_2$ has an



$\Delta V_0V_1V_2$ with its 3 neighbors: $\Delta V_0A_0V_1$, $\Delta V_1A_1V_2$ and $\Delta V_2A_2V_0$

Figure 3

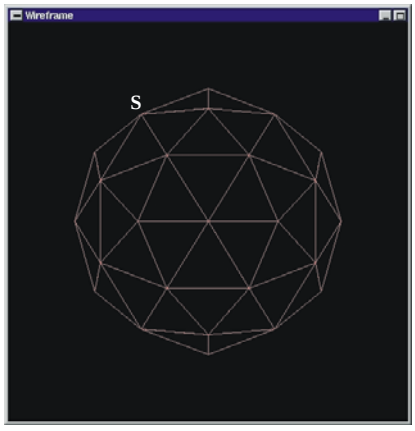
auxiliary vertex A_i ($0 \le i \le 2$) if and only if $\Delta V_iA_iV_{(i+1) \bmod 2}$ is a neighbor of $\Delta V_0V_1V_2$. In Figure 3, A_0, A_1 and A_2 are auxiliary vertices of $\Delta V_0V_1V_2$. Then, given a front-facing triangle $V_0V_1V_2$ with auxiliary vertices A_0, A_1 and A_2, if for some i ($0 \le i \le 2$) $\Delta V_iA_iV_{(i+1) \bmod 2}$ is back-facing, then $\Delta V_0V_1V_2$ is a silhouette triangle, and $V_iV_{(i+1) \bmod 2}$ is a silhouette edge, and V_i, $V_{(i+1) \bmod 2}$ are silhouette vertices. For deciding whether a triangle is front-facing or not, we use the common method of computing the signed area of the triangle in window coordinates and checking its sign, as described in [WOO96]. This scheme can easily be generalized to other polygons.

5.2 Practical Consideration

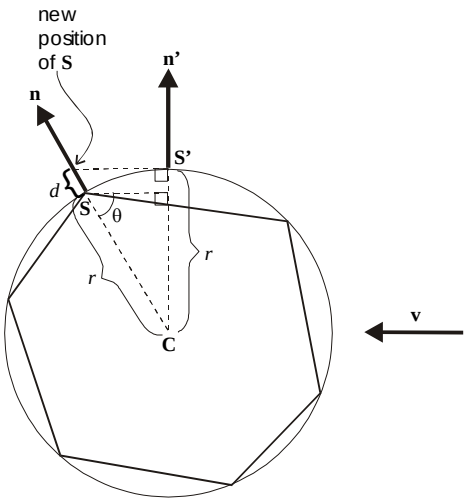
It is obvious that in order to determine whether a polygon is a silhouette polygon or not, we must have its neighbors available. This requirement deviates from the standard practice in computer graphics since the conventional graphics pipeline does not require any neighborhood information to render polygons, and therefore does not possess such information. In fact, the graphics sub-system does not have a notion of a 'mesh' or an 'object' since it treats all polygons independently. It accepts polygons

one by one and renders them one by one in the order it receives them, without knowing which is adjacent to which (with the possible exception of the primitive types triangle strips and quadrilateral strips, which conveys partial neighborhood information, but for the sole purpose of saving extra computation).

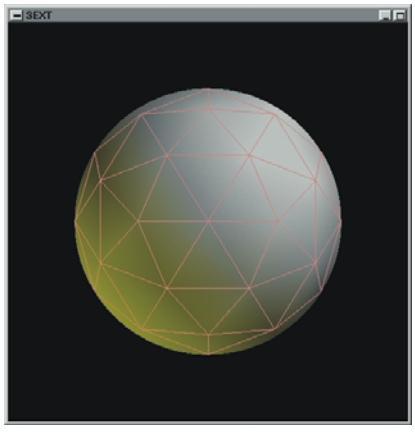
To be able to identify silhouette polygons, we have to provide extra information, which the graphics sub-system must be able to utilize. For each triangle to be rendered, as mentioned above, we have to provide at most three more triangles, which are its neighbors, and we must know which edge of it is shared by which neighbor. This can easily be done by providing auxiliary vertices as defined earlier. The graphics sub-system must be modified in a way such that it can accept and understand such information and use it to identify silhouette triangles (and hence silhouette edges and vertices). This modification has a number of implications both



(a)



(b)



(c)

Figure 4

in terms of implementation and performance of our method, which will be discussed in sections 8 and 10.

6. Perturbation of Silhouette Vertices

As a reasonably accurate model of a curved surface, a polyhedral mesh should have its vertices sampled from the curved surface itself. That means every vertex of a polyhedral mesh should also lie in the curved surface the mesh is supposed to approximate. Although this can be taken for granted in most cases, after viewing projection, a silhouette vertex of a rendered polyhedral mesh may not be in the real silhouette of the curved surface.

6.1 Unrepresentative Silhouette Vertices

For simplicity and ease of understanding, we use a sphere as an example to explain this phenomenon. Imagine that we are viewing a polyhedral mesh which is meant to represent a sphere (see Figure 4a). For any silhouette vertex of the mesh, there exists a unique cross-sectional plane that passes through the vertex itself and is spanned by the normal vector of the vertex and the view vector (a vector pointing from the viewpoint to the vertex). Figure 4b shows such a polygonal cross-section of the mesh together with the 'real' circular cross-section, which the polygonal one is supposed to represent. In Figure 4b, **S** is the same silhouette vertex **S** of the mesh shown in Figure 4a, and **n** is its normal vector. We can see that although **S** is a silhouette vertex of the polyhedral mesh, it is not a point in the silhouette of the sphere. For a smooth surface such as a sphere, the silhouette is a curve in the surface such that the normal vector at any point in the curve has zero *z* component [LANE80]. **S** is obviously not such a point so it is not in the 'real' silhouette. Instead, **S'** is such a point, since its normal vector **n'** has zero *z* component. However, **S'** is just an 'imaginary' point[4] since we are indeed looking at the polyhedral mesh, where there is no such point **S'** in reality. See Figure 4c, where an image of the 'real' sphere is superimposed on that of the mesh, and one will be convinced that **S** is in fact not in the real silhouette.

[4] We just mean that the point does not really exist in the polyhedral mesh. We are not referring to anything related to complex numbers.

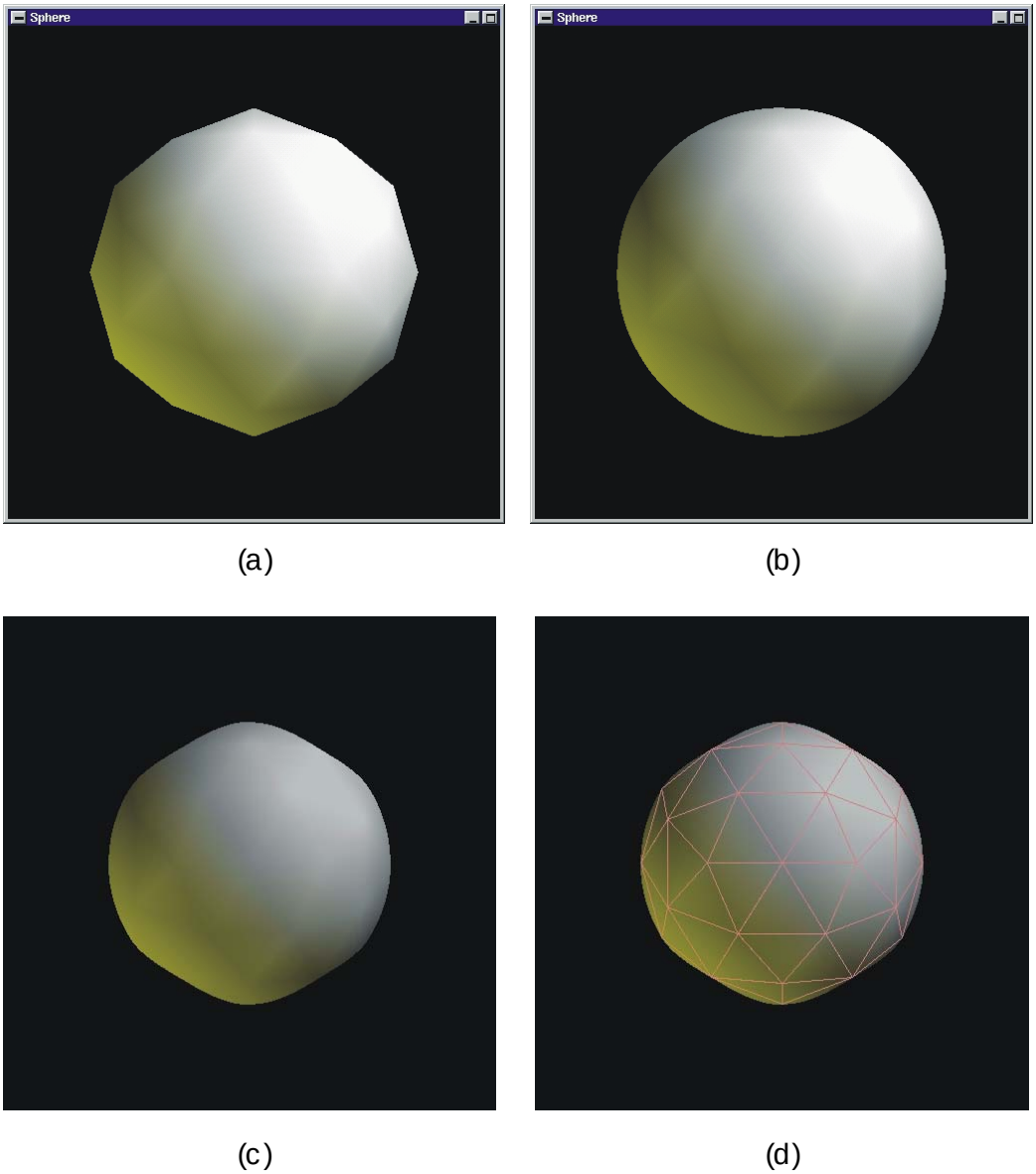


Figure 5

6.2 Unfaithful Static Images

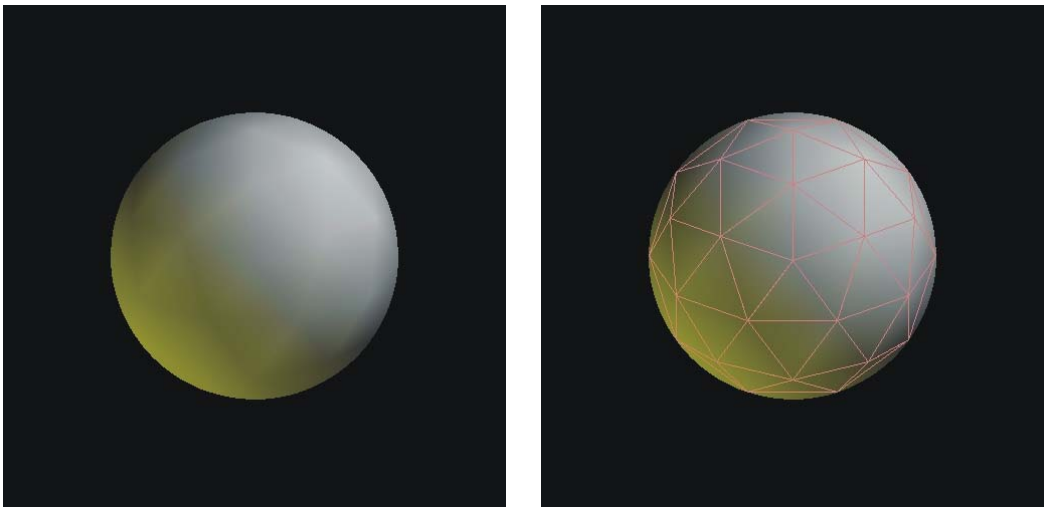
The implication of such a phenomenon is that if we want to have a better illusion that we are looking at a sphere while actually looking at a smoothly shaded polyhedral mesh like the one shown in Figure 5a, when we see a point like **S**, we would actually like to see **S'** in place of it. That is why we have to perturb a silhouette vertex so that it lies in the real silhouette, i.e., we want to move a point like **S** to the position of a corresponding point like **S'**. The necessity of this step will become most obvious if one has seen an image produced by our algorithm with this step (Figure 5b) and

another image produced by our algorithm without this step (Figure 5c). Figure 5d is essentially the same as Figure 5c except that a wireframe of the mesh (not perturbed) is also shown. The object shown in Figure 5c looks like a sphere which is somehow 'pressed' inwards at several places but the one in Figure 5b looks very much like a normal sphere. Recalling Figure 4b, an unrepresentative silhouette vertex like **S** is closer to the center **C** of the sphere than a real silhouette point **S'** does. That is why the object in Figure 5c looks like a 'shrunken' sphere rather than a normal one.

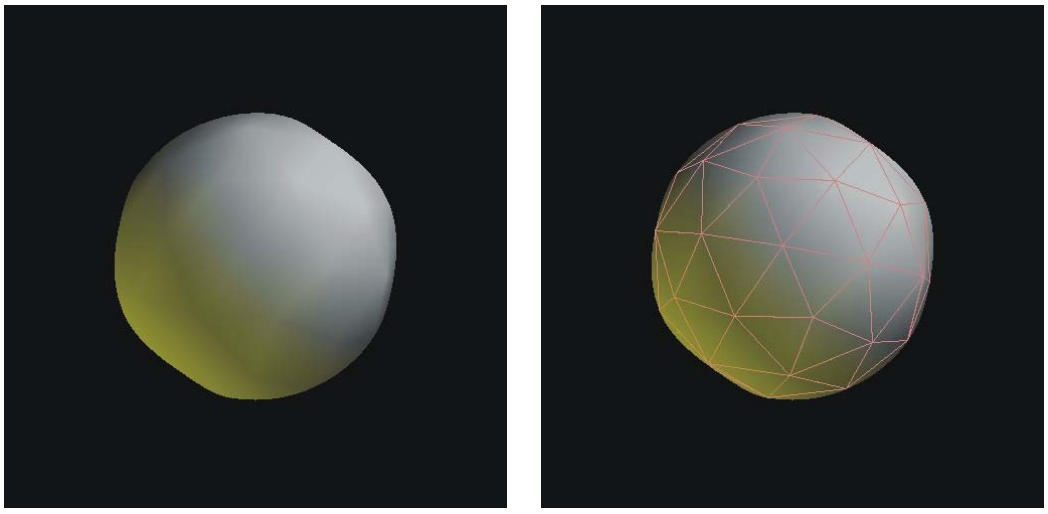
6.3 Lack of Frame-to-Frame Coherence during Animation

In spite of those unrepresentative silhouette vertices, one may be satisfied with an image like Figure 5c: both its silhouette and shading are smooth, and if the viewer does not expect to see a sphere, the image may look perfectly acceptable to him since the object looks like a smooth surface anyway. This may be the case for static scenes, but if the object is meant to be animated, one can hardly accept those unrepresentative silhouette vertices. Translation of the object will not be a problem, but if the object is rotated, we will observe some lack of frame-to-frame coherence. See Figure 6, which shows the same mesh as in Figure 5c but with different orientations. For some orientations (Figure 6a) the silhouette vertices of the mesh happen to be in the real silhouette, and the mesh looks much like a sphere. However, for some other orientations (Figure 6b), it is not the case, and the 'sphere' seems shrunken. If the mesh is continuously being rotated, it will keep changing from one orientation to another, sometimes appearing like a normal sphere, and sometimes appearing like a shrunken sphere. The object will then seem to keep shrinking and enlarging, which is certainly unacceptable.

Recall that our work is concerned with interactive computer applications. In such applications, 3D objects will animate according to user input. Therefore we must cater for the need of animation and perturbation of silhouette vertices is an absolute necessity.



(a)



(b)

Figure 6

6.4 Perturbing Silhouette Vertices for a Sphere

We can get some clues from Figure 4b to how to move the unrepresentative silhouette vertices so that they lie in the real silhouette. The vertex **S** is too close to the center of the sphere **C** so we want to move **S** and other similar vertices a bit away from **C** so

that the object will not have a shrunken appearance. It would be good to move **S** to the position of **S'** in 3D space but it is not necessary to really do so since what a viewer will see is just a 2D image. As long as **S** will be projected on the projection plane to the same point as **S'** would have been, the image will look as if **S** were at the position of **S'** in 3D.

6.4.1 Technicality

For simplicity, we assume the viewer is at infinity, i.e., the projection is an orthographic projection and therefore all points are projected in the same direction, one which is normal to the projection plane. The vector **v** in Figure 4b is the view vector, which is just the reverse of the direction of projection. Under such projection, if **S** is to be projected to the same point as **S'** would have been, it is sufficient to move **S** away from **C** along the direction of **n** by a certain distance d as shown in Figure 4b, where r is the radius of the sphere and θ is the angle between the **v** and **n**. Using simple trigonometry, we know that

$$\sin\theta = \frac{r}{r+d}.$$

By rearranging we get

$$d = r\left(\frac{1}{\sin\theta} - 1\right)$$

Equation 6.1

The new position of **S** is then equal to **S** + d**n**, provided that **n** is a unit vector. If **S** happens to be in the real silhouette, then $\theta = 90°$ and $d = 0$, and therefore **S** is not perturbed.

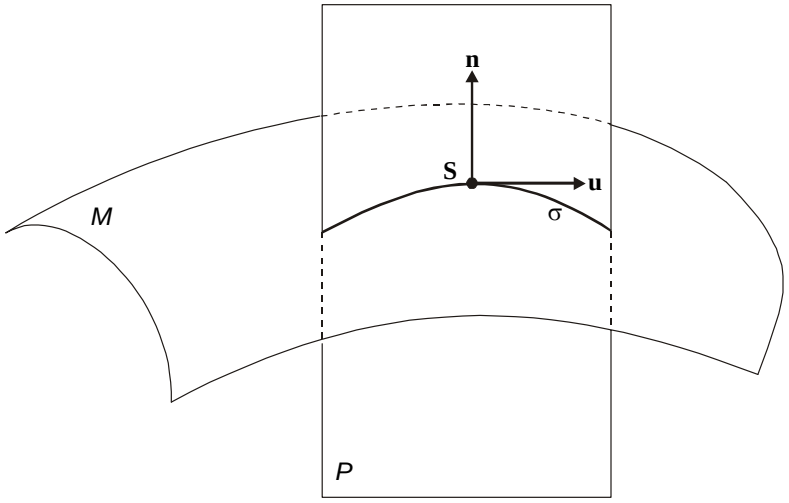
6.4.2 Practical Consideration

There are of course more than one way to move **S** such that it is projected to the same point on the projection plane. We believe that the way described above is the most convenient and fast way to do so basing on two assumptions. First, we assume that r in Equation 6.1 can be computed easily, as described in 6.5 and 8.2. Second, we assume orthographic projections so $\sin\theta$ in Equation 6.1 is just equal to the magnitude of the cross product **n**×**v**. Finally we use a scalar-to-vector multiplication and a vector addition to get **S** + d**n**, the new position of **S**.

6.5 Perturbing Silhouette Vertices for a General Smooth Curved Surface

So far we have just been discussing only the case of a sphere but not for other surfaces in general. As the case of a sphere is simple and intuitive, we use it as an approximation for other curved surface also.

Given any polyhedral mesh, for each silhouette vertex **S,** we must be able to form a polygonal cross-section like the one shown in Figure 4b, except that the cross-section may not represent a circle, but some other curve. We assume that this cross-sectional curve is locally a circular arc in the vicinity of **S**. We further assume that the corresponding real silhouette point **S'** is close enough to **S** such that **S'** is also (approximately) a point of the circular arc just mention. Basing on these assumptions, we can then perturb **S** as if it were a point of a sphere, provided that the radius *r* of this circular arc can be found.



M - underlying curved surface
P - plane containing σ, **u** and **n**

Figure 7

6.5.1 Techniques from Differential Geometry

To find the radius *r*, we have to make use of techniques in differential geometry[5]. In differential geometry, the cross-sectional curve described above is called a *normal section* of the underlying surface. Let us denote this normal section as σ (see Figure 7). Then *r* is the *radius of curvature* of the surface at **S** in the direction **u**, which is a tangent vector to the surface at **S** and parallel to σ. The reciprocal of *r* is the *normal*

[5] For a detailed account of differential geometry, see any book like [ONEI66].

curvature k at **S** in the **u** direction. If we know the exact mathematical description of the underlying curved surface, we can compute the two *principal curvatures* k_1 and k_2, and the corresponding *principal directions* $\mathbf{e}_1$ and $\mathbf{e}_2$ for each vertex **S** of the polyhedral mesh. (Recall that every vertex of the mesh should also lie in the curved surface). We can then compute *k* using Euler's formula: $k = k_1\cos^2\eta + k_2\sin^2\eta$, where η is the angle between **u** and $\mathbf{e}_1$. If we have found *k*, we have found *r*.

6.5.2 Finding the Angle η for Euler's Formula

Finding the angle η is not a trivial task and is worth some detailed explanation. As described in 6.1 and shown in Figure 7, the vector **u** is in the plane *P*, which is defined by the view vector **v** and the normal vector **n** of **S**. For orthographic projections, both **v** and **u** are collinear with the negative *z*-axis (in eye coordinates). Any vector perpendicular to *P* is also perpendicular to both **v** and **u**, and therefore is also perpendicular to the negative z-axis. That means such a vector must have zero *z* component. Let **u'** be a unit vector perpendicular to *P* such that it is also a tangent vector to the underlying surface at **S**. The vector **u'** has zero *z* component so it can be written as (*x*, *y*, 0). Also, **u**, **u'**, $\mathbf{e}_1$ and $\mathbf{e}_2$ are all in the same plane, i.e., the tangent plane at **S**. We can express **u'** as a linear combination of $\mathbf{e}_1$ and $\mathbf{e}_2$ as $a\mathbf{e}_1 + b\mathbf{e}_2$ for some real numbers *a* and *b*. That means

$$\mathbf{u}' = (x, y, 0) = a\mathbf{e}_1 + b\mathbf{e}_2$$

Equation 6.2

If we let $\mathbf{e}_1 = (x_1, y_1, z_1)$ and $\mathbf{e}_2 = (x_2, y_2, z_2)$, from Equation 6.2, we get

$$(x, y, 0) = a(x_1, y_1, z_1) + b(x_2, y_2, z_2) \Rightarrow az_1 + bz_2 = 0$$

That means

$$b = -\frac{az_1}{z_2}$$

Equation 6.3

From Equation 6.2 and Equation 6.3, **u**' can be written as

$$\mathbf{u}' = a\mathbf{e}_1 + b\mathbf{e}_2 = a\mathbf{e}_1 - \frac{az_1}{z_2}\mathbf{e}_2 = a(x_1 - \frac{z_1}{z_2}x_2, y_1 - \frac{z_1}{z_2}y_2, 0)$$

Equation 6.4

Since **u**' is a unit vector,

$$a = \frac{1}{\sqrt{\left(x_1 - \frac{z_1}{z_2}x_2\right)^2 + \left(y_1 - \frac{z_1}{z_2}y_2\right)^2}}$$

Equation 6.5

Let ϕ be the angle between $\mathbf{e}_1$ and **u**'. Then from Equation 6.4 $\cos\phi$ is given by

$$\cos\phi = a\left[x_1\left(x_1 - \frac{z_1}{z_2}x_2\right) + y_1\left(y_1 - \frac{z_1}{z_2}y_2\right)\right]$$

Equation 6.6

Since **u**' is perpendicular to **u**, ϕ and η are complementary and

$$\cos\phi = \sin(90° - \phi) = \sin\eta$$

Equation 6.7

Then, by using Equation 6.5, Equation 6.6 and Equation 6.7, we can evaluate Euler's formula as

$$\begin{aligned} k &= k_1\cos^2\eta + k_2\sin^2\eta \\ &= k_1(1-\sin^2\eta) + k_2\sin^2\eta \\ &= k_1 + (k_2 - k_1)\sin^2\eta \\ &= k_1 + (k_2 - k_1)\cos^2\phi \end{aligned}$$

Equation 6.8

Note that it is not necessary to find η explicitly to evaluate Euler's formula.

6.5.3 Practical Consideration

Since we have to do the computation described in 6.5.2 for each silhouette vertex to perturb, we must be aware of its efficiency. In practice, we have to evaluate a total of four equations: Equation 6.5, Equation 6.6, Equation 6.7 and Equation 6.8. By looking at Equation 6.5 and Equation 6.6, we know that all the 'source' values we need to evaluate the equations are just the coordinates of the two principal vectors $\mathbf{e}_1$

and $\mathbf{e}_2$, which are given as input. Also, since all four equations involves only simple arithmetic operations (Note the square root in Equation 6.5 is not necessary as a is squared eventually.), the computation involved should be quite efficient.

6.6 The Need for Estimation of Curvatures

Unfortunately, we usually only have a polyhedral mesh in hand but not any mathematical description of the underlying curved surface. In this case we have no way to compute the curvatures directly, and have to estimate the principal curvatures and principal directions of the underlying surface using the geometric information provided by the polyhedral mesh. We use the well-known method of least square surface fitting to fit a second order surface to each vertex so that the surface approximately passes through the vertex and all its adjacent vertices, and then computing the principal curvatures and principal directions basing on the surface thus obtained, as described in [KRES98].

6.6.1 Fitting a Surface to Each Vertex

A second order surface used in our curvature estimation scheme is described by the equation

$$z = g(x, y) = ax^2 + bxy + cy^2 + dx + ey + f \ .$$

Equation 6.9

It is the graph of a bivariate function of x and y. This kind of surface is also called a *Monge patch*, as it is in the form $\mathbf{x}(u, v) = (u, v, g(u, v))$ [ONEI66]. The trick to use so that we can use such a simple form for every vertex is to transform the vertex and all of its neighborhood points (i.e., those connected to the vertex by one edge) into a coordinate system with origin at the vertex itself and the positive z-axis along its normal vector. As we are fitting a surface to the vertex, in such a local coordinate system, the surface described by Equation 6.9 is required to pass through the origin, i.e., the vertex. That means the constant term f in Equation 6.9 should vanish. Further, our implementation assumes that the normal vector given for each vertex is a sufficiently accurate one, which therefore is also the normal vector of the surface at the origin, so the coefficients of the linear terms d and e should also vanish. The detailed reason is as follows. The two partial derivatives of the function described in Equation 6.9 are given by

$$\frac{\partial z}{\partial x} = g_x(x, y) = 2ax + by + d \text{ and } \frac{\partial z}{\partial y} = g_y(x, y) = bx + 2cy + e.$$

The *xy*-plane is the tangent plane to the surface at the origin, which means that

$$\left.\frac{\partial z}{\partial x}\right|_{x=0, y=0} = g_x(0,0) = 2a(0) + b(0) + d = 0 \Rightarrow d = 0$$

and

$$\left.\frac{\partial z}{\partial y}\right|_{x=0, y=0} = g_y(0,0) = b(0) + 2c(0) + e = 0 \Rightarrow e = 0.$$

Therefore, Equation 6.9 should reduce to

$$z = g(x, y) = ax^2 + bxy + cy^2.$$

Equation 6.10

Now we have to find out the surface, i.e., the coefficients *a*, *b* and *c* in Equation 6.10. Let us denote each neighborhood point of the vertex as (x_i, y_i, z_i), where i = 1, 2, 3, …, n ($n \geq 3$). By substituting these points into Equation 6.10, we can set up a system of (probably over-determined) system of linear equations:

$$\begin{bmatrix} x_1^2 & x_1 y_1 & y_1^2 \\ x_2^2 & x_2 y_2 & y_2^2 \\ & \vdots & \\ x_n^2 & x_n y_n & y_n^2 \end{bmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}.$$

Equation 6.11

We can then solve for *a*, *b* and *c* by finding the least square solution to this system. Note that each vertex must have at least 3 neighborhood points for this scheme to work.

6.6.2 Computing the Curvature Data

To compute the principal curvatures and principal directions for each vertex, we have to obtain the parameters (*E*, *F*, *G*, *L*, *M*, *N*) of the first and second fundamental form of the surface [KRES98]. As the surface we use is a Monge patch, which is in the form $\mathbf{x}(u, v) = (u, v, f(u, v))$, the parameters can be computed easily as follows [ONEI66]:

$$E = 1 + f_u^2, \qquad L = f_{uu} / W,$$
$$F = f_u f_v, \qquad M = f_{uv} / W,$$
$$G = 1 + f_v^2, \qquad N = f_{vv} / W,$$

where $W = \sqrt{EG - F^2} = (1 + f_u^2 + f_v^2)^{1/2}$.

Then we can solve the following quadratic equations to obtain the two principal curvatures and principal directions [KRES98]:

$$(EG - F^2)k^2 - (EN - 2FM + GL)k + (LN - M^2) = 0$$

Equation 6.12

$$(LF - ME)du^2 + (LG - EN)dudv + (MG - NF)dv^2 = 0.$$

Equation 6.13

Equation 6.13 is a bit subtle. Here, the notation *du*:*dv* denotes a direction, and is called a direction number [LIPS69], which represents the vector $\mathbf{x}_u du + \mathbf{x}_v dv$, where $\mathbf{x}_u$ and $\mathbf{x}_v$ are the velocity vectors along the *u* direction and the *v* direction respectively, i.e., $\mathbf{x}_u(u, v) = (1, 0, f_u(u, v))$ and $\mathbf{x}_v(u, v) = (0, 1, f_v(u, v))$. As the direction number is just a ratio, the actual values of *du* and *dv* are immaterial: only the ratio matters. Thus, without loss of generality, we can just set $dv = 1$ and solve the following equation for the two possible values of *du* and get two directions:

$$(LF - ME)du^2 + (LG - EN)du + (MG - NF) = 0.$$

Equation 6.14

If we denote the two roots of Equation 6.14 by du_1 and du_2, the two directions are $\mathbf{x}_u du_1 + \mathbf{x}_v$ and $\mathbf{x}_u du_2 + \mathbf{x}_v$.

6.6.3 Correspondence of Principal Curvatures to Principal Directions

We do not know which principal curvature is correspond to which principal direction from results obtained in 6.6.2. We solve this problem as follows. A real number *k* is a principal direction in the direction *du*:*dv* if and only if the following condition is satisfied [LIPS69]:

$$\begin{aligned}(L - kE)du + (M - kF)dv = 0 \\ (M - kF)du + (N - kG)dv = 0\end{aligned}.$$

Equation 6.15

Equation 6.15 is again a bit subtle. We denote the two roots of Equation 6.12 by k_1 and k_2. As there are only two ways to pair up the principal curvatures and principal directions, we just assume one of them, i.e., k_1 corresponds to du_1:1, and check if the following is satisfied:

$$\begin{aligned}(L - k_1E)du_1 + (M - k_1F) = 0\\ (M - k_1F)du_1 + (N - kG) = 0\end{aligned}.$$

Equation 6.16

If not, it means that our way of pairing up the principal curvatures and principal directions is not correct so we just swap them.

6.6.4 Back Transformation of Principal Vectors

The final but very important thing to do is to transform the principal vectors back into the object coordinates since all the computation is done in a coordinate system local to the vertex as described in 6.6.1. Also, the principal vectors should be normalized.

6.6.5 Special Case: Principal Parameter Curves

There is a special case where the velocity vectors $\mathbf{x}_u$ and $\mathbf{x}_v$ at a point are already in the principal directions, i.e., the parameter curves at that point are in the principal directions. This is true if and only if $F = M = 0$ [LIPS69]. We have to check this condition beforehand and if it is true, we should not solve Equation 6.14 and the two principal vectors are simply $\mathbf{x}_u$ and $\mathbf{x}_v$. We still have to determine the correspondence between the principal curvatures and principal directions as described in 6.6.3, but instead of Equation 6.16, the condition to check is

$$\begin{aligned}(L - k_1E)du_1 = 0\\ (M - k_1F)du_1 = 0\end{aligned}.$$

Equation 6.17

6.6.6 Umbilic Points

The discussion above generally assumes that a point is non-umbilic[6]. However, this should not concern us since even if a point is umbilic, our scheme still works and it will be just that Equation 6.12 has a repeated root and Equation 6.14 gives two

[6] An umbilic point is one whose normal curvature is constant, i.e., any direction tangent to the underlying surface at the point is a principal direction. Refer to [ONEI66] or any other book on differential geometry for details.

directions which are orthogonal to each other, and must be principal directions (since the point is umbilic anyway).

6.6.7 Estimation of Normal Vectors

Our current implementation assumes that the input normal vectors are accurate enough so that we can just use them directly. However, if one believes that the normal vectors are not accurate enough or if they are simply unavailable, the scheme described in 6.6.1 can be used to estimate the normal vector of a vertex also. In this case the linear terms in Equation 6.9 do not vanish. The constant term is still zero since the surface must pass though the vertex anyway. The only difference is that we do not assume the *z*-axis is normal to the surface. That means the local coordinate system has its origin at the vertex but has no restriction on the choice of the *z*-axis. (The original z-axis of the object coordinates would be a natural choice.) Now instead of solving Equation 6.11, we have to find the least square solution to a system of five or more linear equations, as the two linear terms are included this time. After obtaining the surface, we can find the two velocity vectors $\mathbf{x}_u$ and $\mathbf{x}_v$ at the origin (i.e., the vertex) and then perform a cross product with these two vectors. The normalized cross product is the estimated normal vector for the vertex. Note that for this scheme to work, each vertex must have at least five neighborhood points.

6.7 Practical Consideration

When a vertex is perturbed, all polygons sharing the vertex will be affected. This poses a great practical difficulty for our algorithm. As mentioned earlier, the conventional graphics pipeline treats polygons independent of each other. Not only is it true that the graphics pipeline has no idea about which polygon is adjacent to which, it is also true that it does not know which vertex is shared by which polygons! To facilitate perturbation of silhouette vertices, merely providing polygon neighborhood information with auxiliary vertices is not enough. The graphics sub-system must somehow know which polygons are sharing a particular vertex. More on this issue will be discussed in section 8.

7. Scan-Conversion of Silhouette Polygons

This is a crucial step in our algorithm which makes a rendered polyhedral mesh have a smooth silhouette. In this step, silhouette edges are not rendered as straight line

segments. Rather, they are rendered as curve segments, which join together with each other with G1 continuity, thereby making the silhouette appearing as a smooth curve.

7.1 Obtaining the Curve for Replacement of a Straight Silhouette Edge

Given a silhouette edge, we have two silhouette vertices, each of which has a normal vector for the purpose of shading in the conventional graphics pipeline. Such normal vectors are indeed important clues to how the underlying surface actually looks like, and, if they are projected on the projection plane, also convey information on what curve the silhouette of the underlying surface actually is.

Again for simplicity, we assume orthographic projection. Thus, in eye coordinates, into which normal vectors must be transformed for shading, we can just drop the *z*-component of a normal vector to get its projection. Now for a silhouette edge in window coordinates, we have two vertices, i.e., two endpoints, and each of them is associated with a normal vector, all things being in a 2D plane. If we rotate the normal vectors both in the same direction (clockwise or counter-clockwise) by 90°, we get two tangent vectors. Obviously a curve which interpolates the two endpoints and the two tangent vectors is a perfect candidate for replacing the straight silhouette edge. This situation naturally calls for Hermite curves, or Hermite interpolation. A Hermite curve which interpolates two endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$ with tangent vectors $\mathbf{P}_0'$ and $\mathbf{P}_1'$ respectively is given by the vector equation

$$\mathbf{P}(u) = (2\mathbf{P}_0 - 2\mathbf{P}_1 + \mathbf{P}_0' + \mathbf{P}_1')u^3 - (3\mathbf{P}_0 - 3\mathbf{P}_1 + 2\mathbf{P}_0' + \mathbf{P}_1')u^2 + \mathbf{P}_0'u + \mathbf{P}_0$$

Equation 7.1

where $u \in [0, 1]$. Many books on computer graphics or geometric modeling such as [FOLE90] and [MORT97] have detailed descriptions on Hermite interpolation curves.

7.2 Magnitudes of Tangent Vectors

The two tangent vectors obtained by rotating the two normal vectors have a magnitude of one since normal vectors are unit vectors. That means they only give directions. However, tangent vectors with only directions but no valid magnitudes simply do not give enough information to get a unique Hermite curve since different magnitudes of the tangent vectors give different curves whose appearances can differ

greatly from one another. Therefore we must find some way to assign appropriate values to the magnitudes of the two tangent vectors.

7.2.1 Approximation of Circular Arcs by Hermite Curves

Mortenson describes a way of using a Hermite curve to approximate a circular arc [MORT97]. Our method of assigning tangent vector magnitudes is based on this. Figure 8 shows a situation where we want to approximate a circular arc with a Hermite curve. $\mathbf{Q}_0$ and $\mathbf{Q}_1$ are points in a circular arc of a circle with radius r and center $\mathbf{C}$. θ is half of the angle subtended by the circular arc. The normal vectors of this circular arc at $\mathbf{Q}_0$ and $\mathbf{Q}_1$ are $\mathbf{n}_0$ and $\mathbf{n}_1$ respectively, with corresponding tangent vectors $\mathbf{v}_0$ and $\mathbf{v}_1$. $\mathbf{Q}$ is the intersection of the line passing through $\mathbf{Q}_0$ along the direction of $\mathbf{v}_0$ and the line passing through $\mathbf{Q}_1$ along the direction of $\mathbf{v}_1$. $\mathbf{Q}_0$ and $\mathbf{Q}_1$ are the two endpoints with tangent vectors $\mathbf{v}_0$ and $\mathbf{v}_1$ which a Hermite curve is going to interpolate in a way such that the curve tries to approximate the circular arc. Now $\mathbf{v}_0$ and $\mathbf{v}_1$ are just unit vectors which give directions only. Using different magnitudes for them will give different curves. To approximate the circular arc, the two tangent vectors used in [MORT97] so as to obtain a unique Hermite curve are $4\rho(\mathbf{Q} - \mathbf{Q}_0)$ and $4\rho(\mathbf{Q}_1 - \mathbf{Q})$, where $\rho = \cos\theta/(1 + \cos\theta)$. By finding the two tangent vectors directly, we have essentially found the appropriate magnitudes for $\mathbf{v}_0$ and $\mathbf{v}_1$.

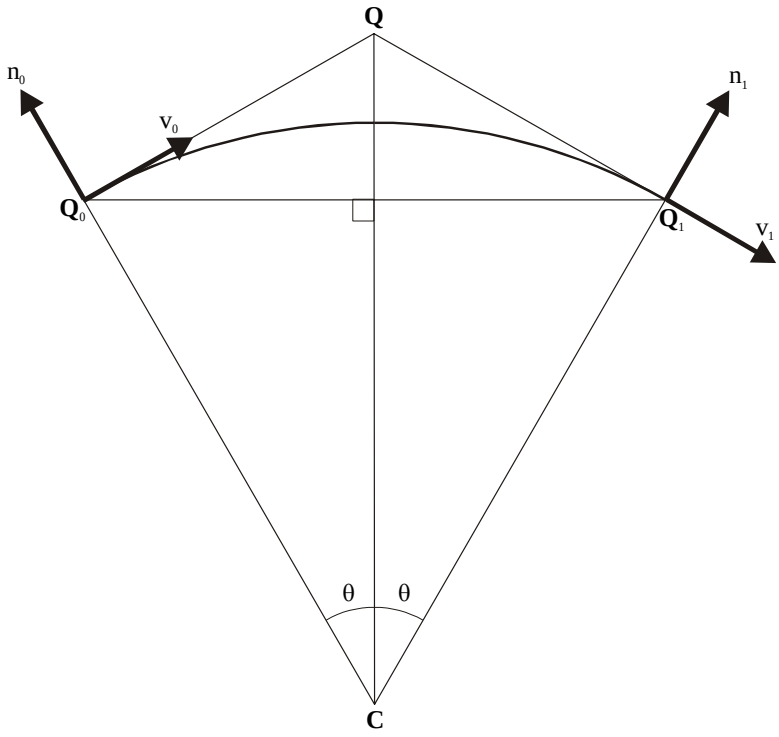


Figure 8

7.2.2 Finding the Tangents in General

If two silhouette vertices and their tangent vectors are actually from a circular arc, we can use the method just mentioned to obtain a Hermite curve. However, in general, we cannot find a circular arc to pass through two arbitrary silhouette vertices and their tangent vectors. Our sole purpose of using the technique of circular arc approximation just mentioned is to obtain some valid values for the magnitudes of the two tangent vectors of the silhouette vertices. We believe that when a Hermite curve tries to approximate a circular arc, the magnitudes of the tangent vectors at the two endpoints have values which make the curve look natural and good. Thus in any case, as long as we can find some (reasonable) circular arc for the Hermite curve to approximate, we can find some good values for the magnitudes of the tangent vectors.

Figure 9 illustrates the technique we use to get a circular arc from two silhouette vertices $\mathbf{Q}_0$ and $\mathbf{Q}_1$ which are not necessarily in a circular arc. $\mathbf{n}_0$ and $\mathbf{n}_1$ are normal vectors of $\mathbf{Q}_0$ and $\mathbf{Q}_1$ as given by the polyhedral mesh, and $\mathbf{v}_0$ and $\mathbf{v}_1$ are their corresponding tangent vectors. We draw a line passing through $\mathbf{Q}_0$ along $\mathbf{n}_0$ and a line passing through $\mathbf{Q}_1$ along $\mathbf{n}_1$ to get an intersection point $\mathbf{C}$, which will be regarded the center of the circle to which the circular arc belongs. Then we move $\mathbf{Q}_0$ in one direction along $\mathbf{n}_0$ and move $\mathbf{Q}_1$ in an opposite direction along $\mathbf{n}_1$ by the same distance t to get two points $\mathbf{Q}_0'$ and $\mathbf{Q}_1'$ respectively such that the line segments $\mathbf{Q}_0'\mathbf{C}$

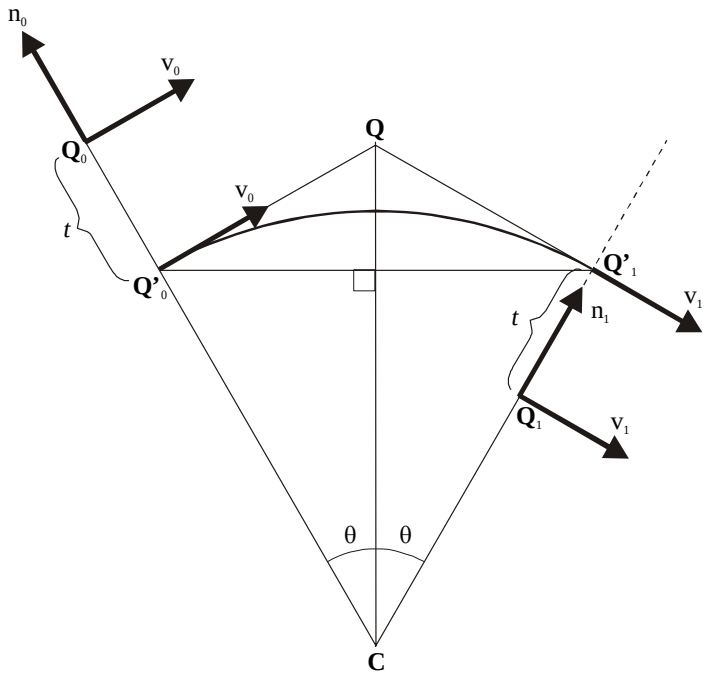


Figure 9

and $\mathbf{Q}_1'\mathbf{C}$ have the same length. Now the triangle $\mathbf{Q}_0'\,\mathbf{Q}_1'\mathbf{C}$ is an isosceles triangle and we have a situation essentially the same as that in Figure 8, with $\mathbf{Q}_0'$ in place of $\mathbf{Q}_0$ and $\mathbf{Q}_1'$ in place of $\mathbf{Q}_1$, and hence there exists a circular arc passing through $\mathbf{Q}_0'$ and $\mathbf{Q}_1'$. With $\mathbf{Q}_0'$ and $\mathbf{Q}_1'$, we use the same way to get a point $\mathbf{Q}$. The tangent vectors we need are $4\rho(\mathbf{Q} - \mathbf{Q}_0')$ and $4\rho(\mathbf{Q}_1' - \mathbf{Q})$, where $\rho = \cos\theta/(1 + \cos\theta)$.

7.2.3 Technicality

Now we have to get into the details of finding t so as to obtain $\mathbf{Q}_0'$ and $\mathbf{Q}_1'$ from $\mathbf{Q}_0$ and $\mathbf{Q}_1$, and finding $\cos\theta$ to get ρ. Using the fact that $\Delta\ \mathbf{Q}_0'\,\mathbf{Q}_1'\mathbf{C}$ is an isosceles

triangle, we know that $\mathbf{Q}'_0\,\mathbf{Q}'_1$ must be perpendicular to the angle bisector of $\angle \mathbf{Q}'_0\,\mathbf{C}\,\mathbf{Q}'_1$. The angle bisector can actually be written as $\mathbf{n}_0 + \mathbf{n}_1$. Therefore, the dot product $(\mathbf{Q}'_1 - \mathbf{Q}'_0)\bullet(\mathbf{n}_0 + \mathbf{n}_1)$ is equal to zero. Since $\mathbf{Q}'_0 = \mathbf{Q}_0 - t\mathbf{n}_0$ and $\mathbf{Q}'_1 = \mathbf{Q}_1 + t\mathbf{n}_1$, $[(\mathbf{Q}_1 + t\mathbf{n}_1) - (\mathbf{Q}_0 - t\mathbf{n}_0)]\bullet(\mathbf{n}_0 + \mathbf{n}_1) = 0$. Rearranging, we get

$$t = -\frac{(\mathbf{Q}_1 - \mathbf{Q}_0)\bullet(\mathbf{n}_0 + \mathbf{n}_1)}{\left\|\mathbf{n}_0 + \mathbf{n}_1\right\|^2}$$

Equation 7.2

However, since $\mathbf{n}_0$ and $\mathbf{n}_1$ are unit vectors, we know

$$\begin{aligned}\left\|\mathbf{n}_0 + \mathbf{n}_1\right\|^2 &= (\mathbf{n}_0 + \mathbf{n}_1)\bullet(\mathbf{n}_0 + \mathbf{n}_1)\\ &= (\mathbf{n}_0 + \mathbf{n}_1)\bullet\mathbf{n}_0 + (\mathbf{n}_0 + \mathbf{n}_1)\bullet\mathbf{n}_1\\ &= \mathbf{n}_0\bullet\mathbf{n}_0 + \mathbf{n}_1\bullet\mathbf{n}_0 + \mathbf{n}_0\bullet\mathbf{n}_1 + \mathbf{n}_1\bullet\mathbf{n}_1\\ &= 2 + 2(\mathbf{n}_0\bullet\mathbf{n}_1).\end{aligned}$$

Substituting this into Equation 7.2, we get

$$t = -\frac{(\mathbf{Q}_1 - \mathbf{Q}_0)\bullet(\mathbf{n}_0 + \mathbf{n}_1)}{2(1 + \mathbf{n}_0\bullet\mathbf{n}_1)}$$

Equation 7.3

The dot product $\mathbf{n}_0\bullet\mathbf{n}_1$ happens to be equal to $\cos 2\theta$ and since

$$\cos\theta = \sqrt{\frac{1+\cos 2\theta}{2}},$$

we get

$$\cos\theta = \sqrt{\frac{1+\mathbf{n}_0\bullet\mathbf{n}_1}{2}}$$

Equation 7.4

7.2.4 Practical Consideration

Since the computational cost of finding the two tangent vectors required is quite significant and we have to do it for every silhouette edge, we must look into the efficiency of some key computation involved carefully. First, we have to evaluate Equation 7.3 to get *t*. The reason why we have to use Equation 7.3 instead of Equation 7.2 is that the computational cost of evaluating the denominator of Equation 7.3 is less than that of Equation 7.2. The other parts of the two equations are just the same and requires only some simple vector and scalar operations. Second, we have to

find $\cos\theta$. Fortunately, as shown in Equation 7.4, we have a value which is computed in Equation 7.3 and can be reused, which is the dot product $\mathbf{n}_0 \bullet \mathbf{n}_1$. However, Equation 7.4 needs a square root operation, which is quite expensive. Finally, as described in 7.2.2, we have to find **Q** to compute the two tangents required. Computing **Q** involves finding the intersection of two lines, i.e., finding the solution to a system of two linear equations.

7.3 Scan-Converting the Curve

After obtaining the curve to replace a straight silhouette edge, we want to scan-convert this curve segment instead of the original straight edge. In our current implementation, we virtually subdivide the curve into many small line segments. We believe that it would be much more efficient to scan-convert the Hermite curve directly, if such a method of scan-conversion does exist. However, for the purpose of fast prototyping we just use the method of subdivision for the time being.

7.3.1 Subdivision into Line Segments

See Figure 10 for an example of a curve segment we want to scan-convert. The figure also shows the original straight silhouette edge and the other two non-silhouette edges which together form a triangle to render. We assume horizontal scan lines and bottom-to-top scanning. Also, all the coordinates we mention here are window coordinates. Conceptually, we subdivide the curve into many small line segments, the number of subdivision being decided by a heuristic we use: it is equal to the difference between the *y*-coordinates of the two endpoints, rounding off to integer. However, for the ease of illustration and discussion, we subdivide the curve in this example into 4 segments only. By doing so, we essentially sample 5 points ($\mathbf{P}_1$, $\mathbf{P}_2$, …, $\mathbf{P}_5$) from the curve.

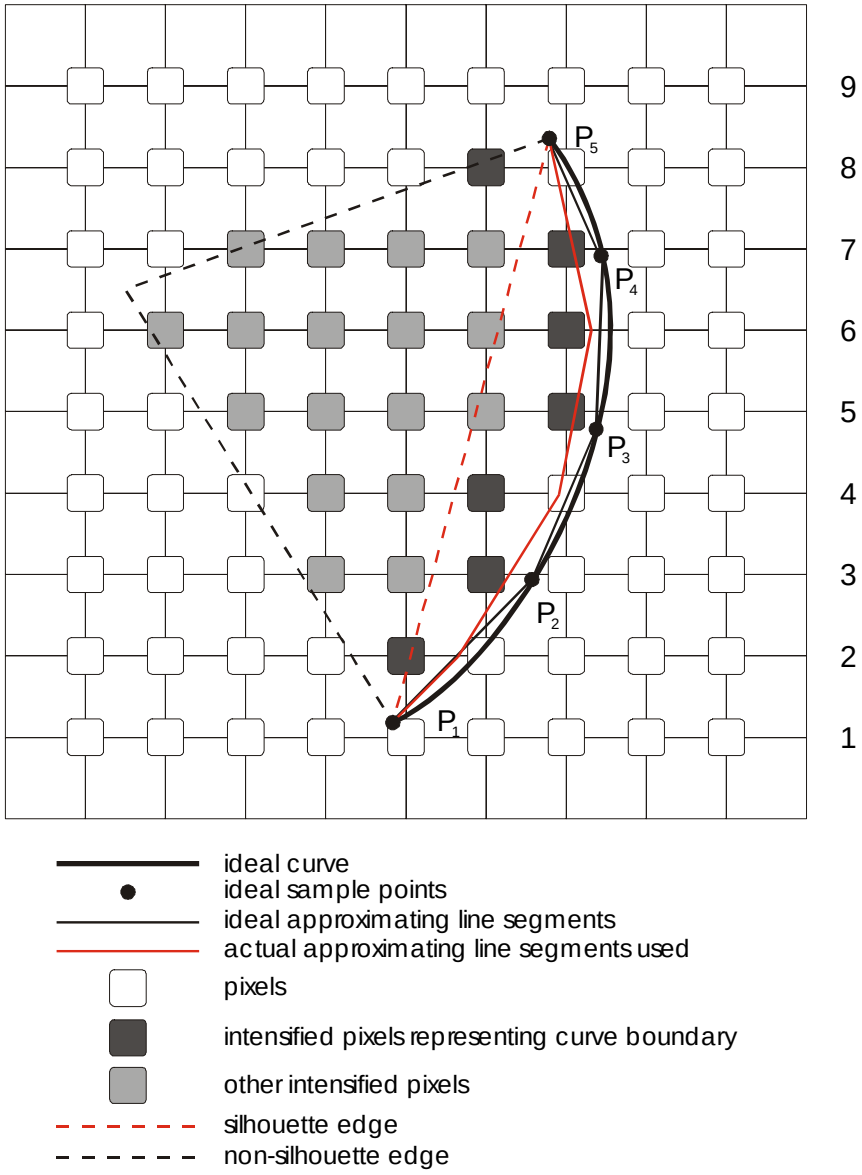


Figure 10

7.3.2 Non-constant 'slope'

Let us first recall the original scan-conversion algorithm. Adjacent pixels on the same scan line which are to be intensified are called a *span*. The two pixels bounding the span are called *span extrema*. When we go from one scan line to the next, we obtain the *x*-coordinate of a span extremum by incrementing the *x*-coordinate of the span extremum of the previous scan line by a certain value, usually called the 'slope',

which is the change in *x*-coordinate with respect to a unit change in *y*-coordinate[7], or *dx*/*dy*. For a given (straight) edge, the slope is constant. However, in our case, the slope is not a constant value. We achieve the effect of rendering a curve by changing the value of the slope regularly while scan-converting along a silhouette edge.

7.3.3 A Step-by-Step Example

We start scan-conversion for the curve in Figure 10 from sample point $\mathbf{P}_1$, which is just an endpoint of the silhouette edge and is given. We call $\mathbf{P}_1$ our *current point*. Before we do any scan-conversion, we have to obtain the coordinates of the next sample point ($\mathbf{P}_2$) by using forward differences [FOLE90]. The forward differences are given by the curve itself and the number of subdivisions. If the *y*-coordinate of this next sample point ($\mathbf{P}_2$) is not greater than that of the current sample point ($\mathbf{P}_1$) by at least 1, we have to find the sample point which is even further ($\mathbf{P}_3$). We would keep on finding further sample points until one with *y*-coordinate greater than that of the current point by at least 1 is found. This sample point is called a *check point*, whose *y*-coordinate is called a *check value*. In this example, $\mathbf{P}_2$ is the check point and we do not have to go any further. Now the slope of our first approximating line segment can be computed easily from the coordinates of the check point and the current point. This approximating line segment is called the *current approximating line segment*.

For scan line 1, we do not have any pixel to intensify since no pixel there is inside the triangle. Before we go to the next scan line (scan line 2), we have to check whether the *y*-coordinate of the next scan line is greater than the check value or not. If yes, we have to do something else before continuing. Now it is not the case so we can go to the next scan line (scan line 2) and obtain the *x*-coordinate of the right span extremum by incrementing the *x*-coordinate of the previous right span extremum (i.e., the *x*-coordinate of $\mathbf{P}_1$) by the slope we just computed. After intensifying the appropriate pixels, we try to go to the next scan line again (scan line 3). However, this time the *y*-coordinate of the next scan line is greater than that of the check value. That means the next scan line is 'higher' than the check point, and if we use the original slope value to increment the *x*-coordinate, we would step outside the curved boundary and have a span which goes beyond it. Thus the current approximating line

[7] In coordinate geometry, the term slope usually means 'the change in *y*-coordinate with respect to a

segment is not valid any more and we have to find a new one and find its slope. Now we set the current point to be the 'real' point corresponding to the current span extremum, i.e., the point in the current approximating line segment with *y*-coordinate equal to that of the current scan line (scan line 2), and find the new check point by applying forward differences to the previous check point ($\mathbf{P}_2$) to find the next sample point, and keep on using forward differences to find further sample points if necessary. (It is now clear that why the check point have to have its *y*-coordinate greater than that of the current point by at least 1.) The new check point turns out to be $\mathbf{P}_3$. By finding the new current point and check point, we essentially have obtained a new current approximating line segment and its slope can be computed. We repeat the process of finding new approximating line segments (and their slopes) and doing scan-conversion according to them until the check point coincides with the last sample point ($\mathbf{P}_5$) or gets higher than it. In the latter case, we set the check point to be the last sample point. We scan-convert according to this last approximating line segment and finish the scan-conversion of the curve segment, which replaces the original straight silhouette edge.

---

unit change in *x*-coordinate', or *dy*/*dx*, which is the reciprocal of our 'slope'.

7.3.4 The Need for Choosing between Horizontal and Vertical Scan Lines

The set of approximated line segments used in Figure 10 is a rather rough approximation to the original curve. In our implementation the approximation is much more accurate since we use more subdivisions. Recall the heuristic we use: the number of subdivisions is equal to the difference between the *y*-coordinates of the two endpoints, rounding off to integer. However, this scheme may cause problems. See Figure 11 for an example, which uses the same legend as Figure 10 does. The curve in Figure 11 is certainly not rendered satisfactorily, and may be considered unacceptable altogther, as something like Figure 12 is normally expected. There are mainly two problems in Figure 11. First, as the difference between the *y*-coordinates of the two endpoints is too small, the number of subdivisions is also too small. Second, the first candidate check point, $\mathbf{P}_2$, is already higher than the higher endpoint $\mathbf{P}_3$, so, according to our scheme as described in 7.3.3, the check point will simply be set to be the higher endpoint $\mathbf{P}_3$, which causes only one approximating line segment to

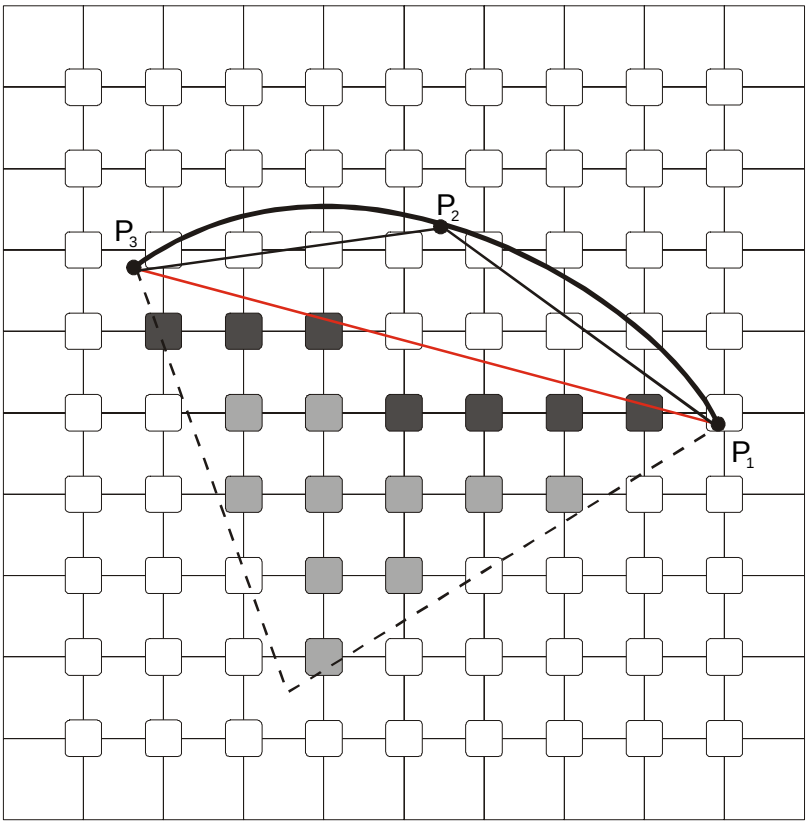


Figure 11

be used and the scan-conversion will go as if the straight silhouette edge were rendered as it is but not replaced by some curve.

In such a case, we should use vertical scan lines and set the number of subdivisions to be the difference between the *x*-coordinates of the two endpoints. Thus, before scan-converting a silhouette edge, we have to check whether the difference between the *x*-coordinates is greater than that between the *y*-coordinates of the two endpoints. If yes, we will use vertical scan lines and the number of

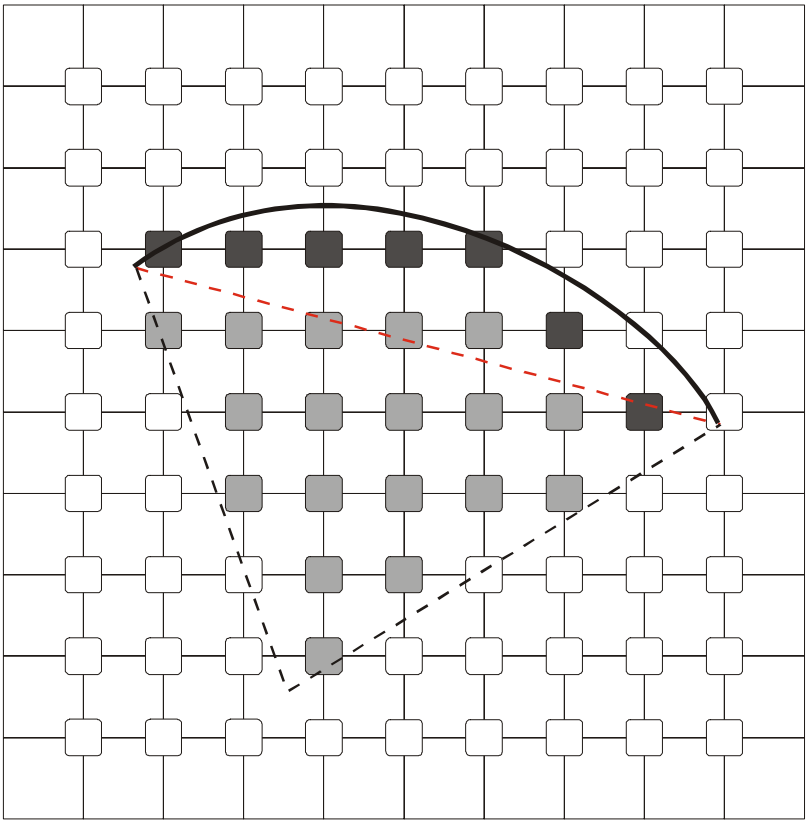


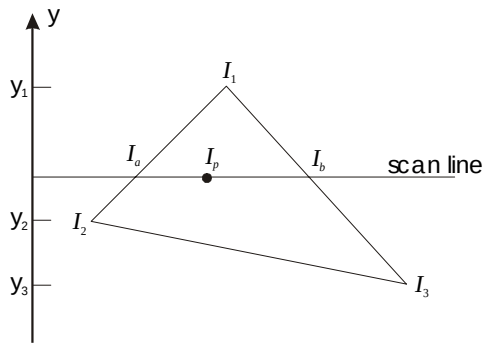
Figure 12

subdivisions is equal to the difference between the *x*-coordinates. Otherwise, we use horizontal scan lines and the number of subdivisions is equal to the difference between the *y*-coordinates.

7.4 Shading

Since we are not rendering an outlined polyhedral mesh but a smoothly shaded one, we must take care of shading also. We assume Gouraud shading, as it is arguably the most commonly used shading method for interactive computer applications. For

simplicity, we also assume that our polyhedral mesh are composed of triangles. In Gouraud shading, a triangle is shaded by linear interpolation of vertex color values along each edge and then between edges along each scan (see Figure 13a). Due to the replacement of straight silhouette edges with curve segments, the silhouette triangles are no longer triangles now but some fan-shaped objects (see Figure 13b). Problems arise from how to shade such objects.



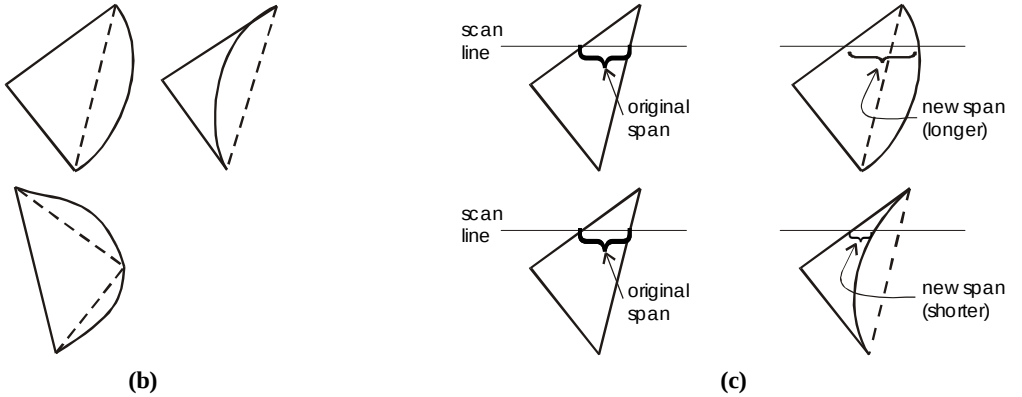
$$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}$$

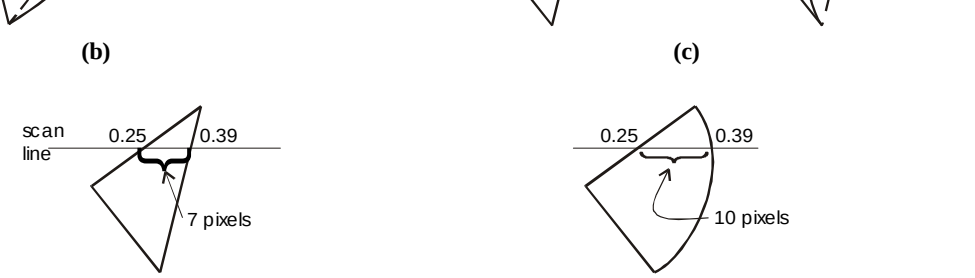
$$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}$$

Intensity interpolation along polygon edges and scan lines

(a)



(b) **(c)**



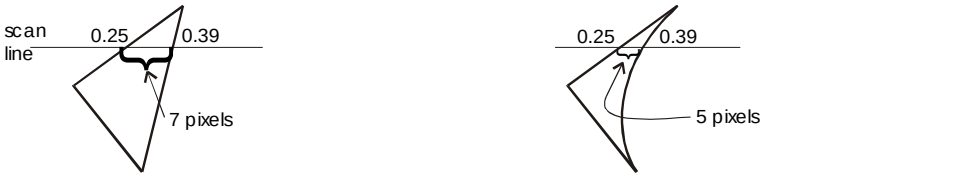
Original intensity values:

0.25	0.27	0.29	0.31	0.33	0.35	0.37	0.39

New intensity values:

0.25	0.264	0.278	0.293	0.306	0.32	0.334	0.348	0.362	0.376	0.39

(d)



Original intensity values:

0.25	0.27	0.29	0.31	0.33	0.35	0.37	0.39

New intensity values:

0.25	0.278	0.306	0.334	0.362	0.39

(e)

Figure 13

We use a straightforward scheme to handle the shading of silhouette triangles. Due to the new shape of a silhouette ‘triangle’, on each scan line the visible span of

the triangle now becomes longer or shorter than the original one (see Figure 13c). In either case, we use the color values of the two boundary points of the original span as the color values of the two boundary points of the new span. Then we do color interpolation as usual to fill the new span (see Figure 13d and Figure 13e). Using this scheme, we have essentially 'stretch' or 'shrink' each span according to the silhouette curve segment. Note that in the figures we just use a single value intensity to refer to a color component for simplicity. In fact, depth values, or z values, are also interpolated using the same scheme.

8. Implementation Issues

Our implementation is based on Mesa [PAUL95], a free 3D graphics library with an API (Application Programming Interface) 'very similar' to that of OpenGL. As a matter of fact, this library is indeed a software implementation of OpenGL. We modified the source code of Mesa and added our own, new API functions so that it can render polyhedral meshes according to our new method.

We believe that basing our implementation on some existing (and reliable) code rather than doing it all from scratch has at least two major advantages. The first one is convenience, since we can save ourselves from implementation not directly related to our idea, such as transformation, lighting calculation, and so on. Also, we can write our testing programs in a way very similar to that of writing ordinary OpenGL programs, except that we also use the new function calls added by ourselves, and use GLUT for building simple user interface. The second advantage, which may be more important, is that we can easily notice what modifications are needed in the conventional 3D graphics pipeline so that it can incorporate our new method of rendering smooth surfaces. Since Mesa is virtually an implementation of OpenGL, which, as a 3D graphics API, is in turn an implementation of the conventional 3D graphics pipeline, we know exactly how the pipeline can be modified to incorporate our method after we have modified the source code of Mesa to implement our algorithm.

8.1 A Polyhedral Mesh as a Whole, Not Separate Polygons

To facilitate perturbation of silhouette vertices, the graphics sub-system must have access to the whole polyhedral mesh throughout the process of rendering. It is

because when a vertex is perturbed, all polygons sharing the vertex essentially become different in terms of geometry, as one of their vertices is changed. The graphics sub-system must be able to make the changes for those polygons before rendering them, and therefore it has no choice but to wait for the process of identifying silhouette polygons and that of perturbing silhouette edges to finish before actually rendering the polygons.

This contrasts with what is being done in the conventional graphics pipeline, in which polygons are treated individually, and they are immediately processed and rendered upon arrival at the pipeline, without the need for waiting for other polygons. Not only polygons, but also vertices are processed independent of each other to some extent. A vertex is transformed, undergoes lighting calculation, etc., as soon as it is sent to the graphics pipeline, without waiting for other vertices, except that before rendering, it must wait for other vertices as all vertices of a polygon must be available before they can be rendered. All operations of the graphics pipeline can indeed be done in a pipeline fashion: while a vertex is undergoing lighting calculation, another vertex which comes after is being transformed, and yet another vertex which came before is waiting for them so that a triangle can be rendered later; with respect to polygons, the situation is similar. Our algorithm does not enjoy this pipeline property, since a polyhedral mesh must be seen as a whole, not separate polygons.

8.2 Specification of Vertices and Associated Data

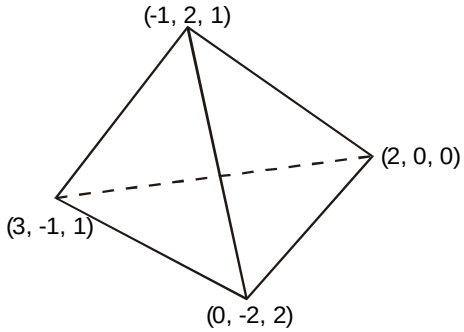
Our algorithm imposes certain restrictions on the format of specifying vertices and their associated data for the graphics subsystem. Unlike the case of the conventional graphics pipeline, where vertices can be specified one by one without relation to each other, our algorithm requires that all vertices of the entire polyhedral mesh to render must be passed to the graphics sub-system as a whole, for reasons mentioned in 8.1.

The best way to do so would be storing the vertex coordinates in an array and then passing the pointer to this array to the graphics sub-system. Let us call this array a *vertex array*. However, the vertex array alone are not enough for specifying a polyhedral mesh. We must also have an array of indices which point to individual elements of the vertex array, to tell how the vertices constitute different faces. This array of indices would be called a *vertex pointer array*. An example is shown in Figure 14a and Figure 14b, where a tetrahedron is specified. The first face is formed

by the vertices with indices 2, 1 and 0, and the second face is formed by the vertices with indices 1, 3 and 0, and so on.

Note that using a format such as that in Figure 14c to specify the tetrahedron is not acceptable. The vertices specified in the vertex array must be unique, i.e., a vertex can be specified in the vertex array once and only once. This restriction is for the convenience of perturbation of silhouette vertices. If a vertex is moved, we just have to change its entry in the vertex array and the polygons sharing the vertex will have been changed accordingly. The requirement for specifying vertices through a vertex array with unique vertices and a vertex pointer array provides the graphics sub-system with enough information to perform perturbation of silhouette vertices conveniently and efficiently.

Other than catering for perturbation of silhouette vertices, the format of vertex data specification must also facilitate identification of silhouette polygons. That means there must be some way to specify auxiliary vertices. Using the tetrahedron in Figure 14a as an example, we can specify auxiliary vertices using an *auxiliary vertex pointer array* as shown in Figure 14d. Note that the face consisting of the vertices pointed to by entries in the vertex pointer array with indices i, i+1 and i+2 have auxiliary vertices pointed to by entries in the auxiliary vertex pointer array with indices i, i+1 and i+2. This corresponds to V_i, V_{i+1}, V_{i+2} and A_i, A_{i+1}, A_{i+2} in the definition of auxiliary vertices. The auxiliary vertex pointer array has the same number of elements as the vertex pointer array.



(a)

vertex array

	(-1, 2, 1)	(2, 0, 0)	(0, -2, 2)	(3, -1, 1)
array index	0	1	2	3

vertex pointer array

	2	1	0	1	3	0	2	0	3	1	2	3
array index	0	1	2	3	4	5	6	7	8	9	10	11

(b)

vertex array

	(-1, 2, 1)	(2, 0, 0)	(0, -2, 2)	(2, 0, 0)	(3, -1, 1)	(-1, 2, 1)	(0, -2, 2)	(-1, 2, 1)	(3, -1, 1)	(2, 0, 0)	(0, -2, 2)	(3, -1, 1)
array index	0	1	2	3	4	5	6	7	8	9	10	11

vertex pointer array

	0	1	2	3	4	5	6	7	8	9	10	11
array index	0	1	2	3	4	5	6	7	8	9	10	11

(c)

auxiliary vertex pointer array

	3	3	3	2	2	2	1	1	1	0	0	0
array index	0	1	2	3	4	5	6	7	8	9	10	11

(d)

Figure 14

Besides the above, our algorithm also requires the graphics sub-system to know the normal vector, the two principal curvatures and the two principal directions associated with each vertex of the polyhedral mesh to render. These can be specified with a *normal vector array*, a k_1 *array* and a k_2 *array*, together with an $\mathbf{e}_1$ *array* and an $\mathbf{e}_2$ *array* respectively. All these arrays have the same number of elements as the vertex array, since they are data associated with the vertices on a per-vertex basis. Therefore, to specify a polyhedral mesh for our algorithm, one has to provide the graphics sub-system with a total of eight arrays of data, with the restrictions mentioned above.

8.3 Preprocessing

Ordinary polyhedral mesh data are certainly not in the format just described above. They are usually presented in the form of individual polygons, since it is exactly what the conventional graphics pipeline needs to render them. Having such a set of mesh data in hand, we have to derive adjacency information from them so as to present them in the form of a vertex array, a vertex pointer array and an auxiliary vertex pointer array. For the normal vector array, if different occurrences of a unique vertex has the same normal vector throughout the original mesh data, the task is trivial. For the curvature information, i.e., the principal curvatures and the principal directions, we have to estimate them once we obtain the adjacency information, using the technique of least-squares surface fitting as mentioned earlier.

Given a set of individual polygons as input, we currently use a rather brute force approach to find out the adjacency information of the underlying mesh, and we assume that the input are triangles. Our aim is to obtain an adjacency list of the graph as presented by the mesh and a list of triangles which links their vertices back to the nodes of the graph. For each input triangle, we check whether each of its edges is already present in the adjacency list or not. If not, insert the edge, and in any case add the triangle to the triangle list and establish the links between the edges and the triangle. After we have exhausted the triangles of the original input data, we will have the adjacency information we need to prepare the input format of mesh data required by our rendering algorithm.

Note that the above processing only involves information which remains invariant during interactive display of the polyhedral mesh. That means the

information is view-independent and therefore is obtained before interactive display. Hence, this is a preprocessing step, which does not affect the efficiency of interactive rendering by our algorithm.

8.4 Assumptions and Limitations of the Current Implementation

Our current implementation of the algorithm is not a really general one. It relies on certain assumptions and have some limitations. First of all, we assume the input polygons are all triangles. Also, the polyhedral mesh is supposed to represent a **closed**, smooth curved surface, with no sharp corner or edges. Further, the implementation is currently limited to orthographic projections only. All these are just for the purpose of fast prototyping and we believe that our algorithm can exceed these limits after some modifications. A more general implementation will be under way.

8.5 After Thought: An Alternative Implementation

There is an alternative way to implement our algorithm which introduces fewer changes to the traditional graphics pipeline: we can move most of the extra processing introduced by our algorithm away from the pipeline and consider it some kind of preprocessing. Recall the four steps depicted in section 4. For Step 1, i.e., identification of silhouette polygons, our current implementation already does it quite deep inside the 'pipeline', i.e., after the transformation of all vertices into window coordinates. The reason is that traditional visible-face determination is done in window coordinates, and we can make use of the result of such processing, which must be done anyway. Step 2, perturbation of silhouette vertices, is done in eye coordinates, but since Step 1 already requires transformation into window coordinates, which is done after transformation into eye coordinates, it does not make any difference. Step 3 is almost in the lowest level of the 'pipeline' as it involves scan-conversion. Step 4 is the same old thing which does not concern us.

Step 1 involves visible-face determination, but it is not a must to do it in windows coordinates: it can also be done in object coordinates. The only difference is that we use another way to determine whether a face is back-facing or not, i.e., to perform a dot product with the face normal and the direction of projection and check whether it is negative or not. Step 2 can also be done in object coordinates, though the computation thus involved will be a bit not as simple as that in eye coordinates.

Since these two Steps can be done in object coordinates, it is possible to move them out of the graphics pipeline altogether. That means we identify silhouette polygons, and perform the necessary perturbation of vertices in our own data structure, which is external to the graphics pipeline, before sending the vertex data to the pipeline. After these two Steps are done, the vertex data can be passed to the graphics pipeline as individual vertices in the same old fashion! The graphics pipeline still needs modification, but only for the scan-conversion part. It is even better if the graphics pipeline can reuse the result obtained in visible-face determination done in Step 1.

This way of implementation sounds really tempting. The reason for not doing so can be considered 'historical'. There is no reason why we would not consider doing the implementation all over again in this alternative way.

9. Quality

For a static image of a smoothly shaded polyhedral mesh rendered by our algorithms, the quality of the silhouette should be very visually pleasing since the silhouette is guaranteed smooth, as curve segments replacing straight silhouette edges are joint together with each other with G1 continuity. For example, see the Color Plates, which show images of some simple surfaces rendered by our algorithm. Using our algorithm, the faithfulness of a polyhedral mesh to the underlying curved surface depends mainly on the accuracy of the polyhedral model, including that of the vertices, normal vectors and computed or estimated curvatures. Quality of frame-to-frame coherence during animation (i.e., rotation) depends on how accurate our assumptions made in the process of perturbation of silhouette vertices are, and of course depends on how accurate the curvature information is.

So far we have only tried rendering models which represent simple surfaces and have an exact mathematical description. In fact, we generated the polyhedral meshes by ourselves using their mathematical descriptions. Our algorithm is meant to be applicable to arbitrary polyhedral meshes. However, due to the lack of suitable models (those which are smooth every where and do not have sharp corners/edges), we have been unable to test our algorithm thoroughly. A more extensive testing will be done in future.

10. Performance

It is difficult to make an objective measurement of the performance of our algorithm compared to the conventional method for several reasons. We will see some possible ways of comparing the performance of the two algorithms one by one.

10.1 Comparison to the Conventional Rendering Algorithm

If both algorithms render the same mesh, i.e., the numbers of polygons to render are equal, our algorithm must be slower than the conventional algorithm since extra computation is needed to handle the silhouette. This would not be a fair comparison, but it is still worthwhile to do such a comparison since we can know how slow our algorithm is when compared to the conventional method.

Alternatively, we can compare the rendering speeds of the two algorithms such that they produce images with silhouettes of the same or comparable quality. However, this almost implies that the conventional algorithm must use (much) more polygons than ours does to achieve the same silhouette quality. What remained to see is virtually whether the extra computation for handling silhouette edges of our algorithm compensate for the otherwise increase in number of polygons to render or not. A major difficulty here is that judging the quality or smoothness of silhouettes may be subjective. Also, if the two methods produce silhouettes of the same or comparable quality, the quality of shading produced by the conventional method is probably better since more polygons are used. This would neither be a fair comparison.

An ideal comparison might be made in the case where the two methods produce images of the same image quality in terms of both the silhouette and shading. Again, we have to resort to subjective judgement.

10.2 Obvious Performance Penalties and Some Counter Arguments

Nevertheless, there are arguably some obvious performance penalties of our algorithm in comparison with the conventional one. First, vertices and polygons can no longer be processed in a pipeline fashion as they cannot be treated as independent of each other. Efficiency gained by pipeline processing will be lost in our algorithm. Also, the graphics sub-system must have a notion of a polyhedral model as a whole. Owing to this, the graphics sub-system may have to have access to external memory

since it is simply impractical to imagine a graphics sub-system implemented in hardware has enough internal memory to house a polyhedral mesh as a whole. Since accessing external memory is usually slower than accessing internal one, the rendering speed would be slower. Moreover, more data per vertex is needed to pass to the graphics sub-system, namely auxiliary vertices and curvature data.

However, the features mentioned above may not be totally undesirable. As mentioned earlier, our algorithm requires a programmer to provide vertex data using arrays, and in a way such that each shared vertex must be specified only once since perturbation of vertices are needed. For the conventional graphics pipeline, if the primitive type is triangles, the same vertex would be specified and therefore undergo transformation, lighting calculation, etc., more than once since a vertex is usually shared by three or more triangles. Our requirement can be considered desirable since each vertex is guaranteed to undergo all processing once. This may compensate for the performance penalty of the inability of pipeline processing. Further, the requirement of the graphics sub-system to access external memory may not be much a penalty, as our way of specifying vertex data is very similar to that of using vertex arrays in OpenGL, which is an extension added to the more recent versions for increased performance, since it was found that specifying vertices one by one consumes many procedures and is therefore inefficient. After all, the real obvious performance penalty of our algorithm may be just the extra computation for detecting and handling silhouette polygons, for which there is no desirable characteristics to compensate in terms of efficiency. One more important point to note is that it would be an entirely different story if we implement our algorithm in the way described in 8.5.

10.3 Some Timing Information

Here we present a comparison of the frame rates achieved by our algorithm to those by the conventional one (Figure 15)[8]. Polyhedral meshes with different number of triangles are rendered (Figure 16)[9]. The meshes are supposed to represent spheres and are produced by subdivision of an icosahedron (a twenty-face Platonic solid) as described at the end of Chapter 2 in [WOO96]. The machine used is a Pentium Pro

[8] Note that the *x*-axis of Figure 15 is not in scale.
[9] Note that all meshes rendered by our algorithm exhibits smooth silhouettes regardless of the number of faces.

200 running Linux with 96MB of main memory. Our rendering is done with the implementation of our algorithm based on Mesa [PAUL95] while the conventional rendering is done with the original Mesa, both of which do not have any hardware acceleration for 3D graphics. The data are obtained by continuously rotating the meshes along *x*-axis and measuring the time needed to render each frame. We have rendered 100 frames for each mesh and the orientations of two consecutive frames differ by 1 degree. See the Appendix for the timing data obtained.

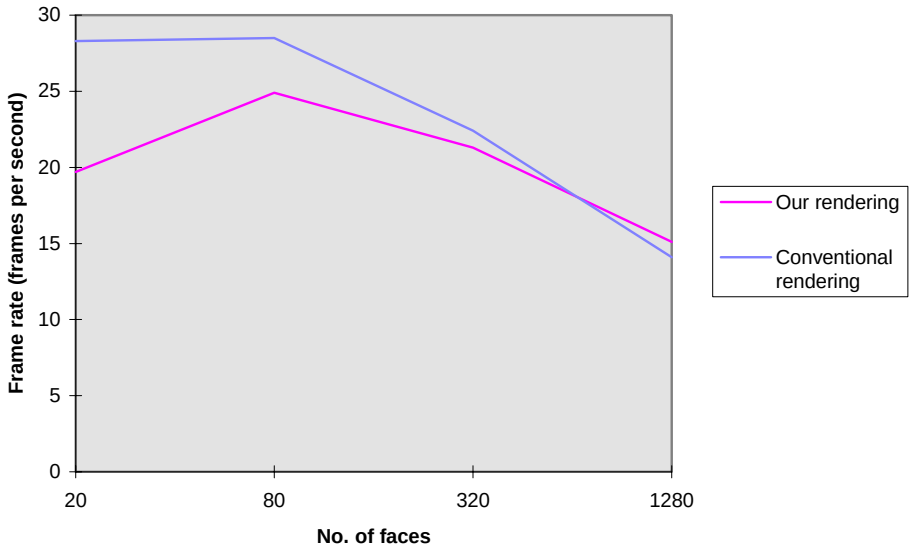
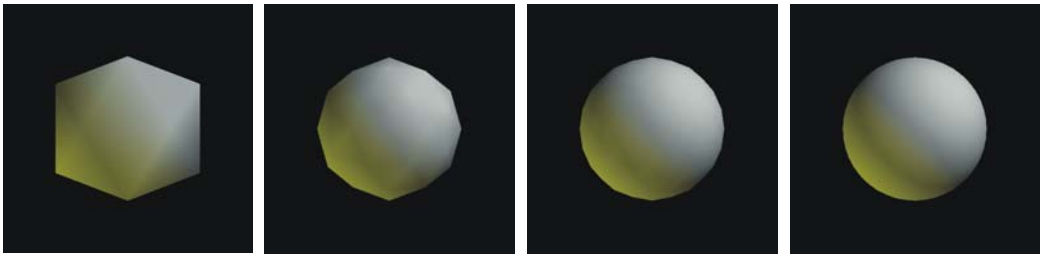


Figure 15

We can see some surprising results. Using our algorithm, the frame rate of rendering a mesh with only 20 triangles is lower than that with 80 triangles. This is probably due to the fact that using our algorithm, processing pixel spans for a silhouette triangle is computationally more expensive than that for an ordinary (interior) triangle. With only 20 triangles, most of the triangles are silhouette triangles and therefore almost all pixel spans are processed in the slow fashion, making the rendering time very long.

Another strange result is that the frame rate of rendering a mesh with 1290 triangles using our algorithm is faster than that using the conventional algorithm! The reason is that our mesh has been preprocessed, i.e., each shared vertex is specified and processed only once, but the mesh for conventional rendering are presented as individual triangles, so a shared vertex may undergo processing many times. As there are many triangles, the saving of computation of our algorithm

becomes significant. Also, pixel spans that are processed in the (slow) fashion for silhouette triangles are much fewer than those which are processed in the original (fast) fashion.



Conventional rendering

Our rendering

Figure 16

11. Future Work

The top-priority thing to do would be an extensive testing of our algorithm on arbitrary polyhedral meshes, as mentioned in section 9. Besides, our work as described here leaves a number of places for future enhancements and extensions. Some of these will improve our existing algorithm both in terms of efficiency and quality. Others will increase the applicability of the algorithm.

11.1 Better Curvature Estimation

It is possible to find if there are better ways for curvature estimation. Recall that given an arbitrary polyhedral mesh for which we do not know what the underlying curved surface is, we have to estimate the principal curvatures and principal directions for each vertex to facilitate perturbation of silhouette vertices, as described in Section 6. We currently use the method of least square surface fitting. So far we have not done a serious survey on different methods of curvature estimation. It

should be worthwhile to see if we can do curvature estimation more accurately, since curvature information is crucial to perturbation of silhouette vertices, which in turn determines the quality of frame-to-frame coherence during animation.

11.2 Better Way of Scan-Converting the Silhouette Curve Segment

Our way of scan-converting the curve segment which is used to replace a straight silhouette edge is a rather ad-hoc and inefficient one. As described in 7.3, we have to change the value of the 'slope' regularly. Computation of a slope requires a floating-point division, which is considered quite expensive in the low level. In the original scan-conversion algorithm, every edge is a straight edge and the slope is computed only once for each edge, since the slope of a straight line is a constant value. In our case, as we have to change the value of the slope, we have to perform several floating-point divisions while scan-converting a silhouette edge. If we can scan-convert the curve directly using forward differences, i.e., without breaking the curve into many small line segments (like scan-converting a circular arc), we can replace those extra floating-point divisions by some floating-point additions. It would be another interesting topic to see if we can scan-convert a Hermite curve directly.

11.3 Preservation of Sharp Corners and Edges

Our algorithm assumes that an input polyhedral mesh represents a surface which is smooth everywhere, i.e., it has no sharp corners and edges of any kind. This assumption is somewhat unrealistic. Although objects in reality do have many parts which can be represented by smooth curved surface patches, those parts nevertheless may join together with each other at sharp corners or edges. Just look around and you would be amazed by how few objects there are which do not have at least one or two sharp corners and/or edges. In order to make our algorithm a practical one, we must modify it so that it can handle sharp corners and edges.

For handling a sharp edge, one way to do so is to regard a shared vertex which is on a sharp edge as two separate vertices. Using this convention, two faces jointed at a sharp edge will not be counted as neighbors. If we use a further convention that any edge which is not shared is not a silhouette edge, we should be able to handle the problem of sharp edges. However, handling sharp corners like apices of cones will be much more difficult.

Another way is to provide a way of specifying vertices for users such that they can simply label the vertices at sharp corners and/or edges and therefore the algorithm can know where these sharp corners and/or edges are. The process of determining which vertex should be labeled is supposed to be done in the modeling phase.

Yet another way is to preprocess the mesh so as to check whether the angle between a pair of faces is smaller than a threshold value called the *crease angle*. If the angle is smaller than the crease angle, the edge shared by the two faces is considered an sharp edge. The crease angle should be a value which can be modified by users.

11.4 Textures

Textures are now an almost indispensable part of interactive computer graphics applications. Virtual reality, computer animation, video games and so on all involve extensive use of textures, as it is a rather computationally inexpensive way which can increase the realistic effect of rendered objects tremendously. Our algorithm currently does not take textures into account. Though we believe that texture coordinates are just like RGB color components and depth values in a sense such that they can all be interpolated in the way described in 7.4, we cannot be sure about the visual effect thus obtained. We have to modify the current implementation to test whether this idea is viable or not.

11.5 Incorporation into Traditional Graphics Sub-systems

Although our way of rendering polyhedral meshes is different from that of traditional graphics sub-systems in a number of ways, the former is based on the traditional scan-line algorithm integrated with Gouraud shading, which essentially forms the core of the latter. In other words, the two are significantly similar. From the experience of our implementation, it is clear that our algorithm can be incorporated into a traditional graphics sub-system implemented in software without major technical difficulties, since we modified the source code of Mesa, a de-facto software implementation of OpenGL, which in turn is an implementation of the traditional graphics pipeline, to implement our algorithm instead of writing all our own code from scratch.

However, when it comes to graphics sub-system implemented in hardware, which is nowadays a commonplace, problems arise. The original scan-line algorithm integrated with Gouraud shading is really a simple, elegant and efficient algorithm,

which is suitable for cost-effective hardware implementation. As our algorithm takes various measures to identify and render silhouette polygons, which is not done in the original algorithm, it can hardly be expected that our algorithm enjoys the same suitability for hardware implementation as the original one. We are somehow placed in a dilemma: 3D computer graphics are made commonplace because of efficient and inexpensive graphics hardware but our algorithm, which is meant to improve the quality of 3D computer graphics in a relatively computationally inexpensive way, may not be that suitable for hardware implementation. A serious and thorough investigation into the possibility of efficient and cost-effective hardware implementation of our algorithm is certainly out of the scope of this work. We, however, believe that it would be worthwhile to do such an investigation if resources allow.

12. Conclusion

We have introduced a new way of directly tackling the well-known problem of polygonal silhouettes of polyhedral meshes rendered by interpolated shading. The algorithm we propose does not involve any radically new theories or techniques which are unknown to the computer graphics community. Rather, our algorithm is based on the original, already widely used scan-conversion algorithm integrated with Gouraud shading. We have borrowed theories and techniques from differential geometry and curve interpolation, while adding several workarounds to various places to construct our algorithm.

The conventional way of rendering polyhedral meshes is modified such that those originally straight silhouette edges which would otherwise join together with each other at sharp corners are now replaced by appropriately curved edges which join together with each other with G1 continuity, thereby yielding a rendered mesh with a silhouette which is guaranteed smooth. Our method does not increase the number of polygons to render and has decoupled improving the smoothness of the silhouettes from improving the smoothness of shading, in contrast to what is commonly being done. Although extra computation and special requirements on the format of input data are needed to identify silhouette polygons, to appropriately process them and finally render them, and our algorithm does not enjoy the efficiency of pipeline processing like the conventional algorithm does, all this extra overhead

can hopefully be offset by the performance gained by rendering a much smaller number of polygons than that would be needed by the conventional rendering algorithm to obtain a silhouette which is guaranteed smooth. This performance gain cannot be overlooked since if a polyhedral mesh rendered by the conventional way has to have a silhouette which is guaranteed smooth, the mesh has to be subdivided into many tiny polygons, each of which must be smaller in size than a pixel, which means that if an object occupies a significant portion of the computer screen, the number of polygons to render can by very large.

It cannot be claimed that our new algorithm is better and faster than the conventional one in rendering smoothly shaded polyhedral meshes with smooth silhouettes by the results obtained so far. It is even hard to predict the practicality of our algorithm. Nevertheless, we believe that this work has at least proposed a new perspective of looking at the problem of polygonal silhouettes of smoothly shaded polyhedral meshes, and hopefully can lead to some new research directions in this area.

14. Appendix

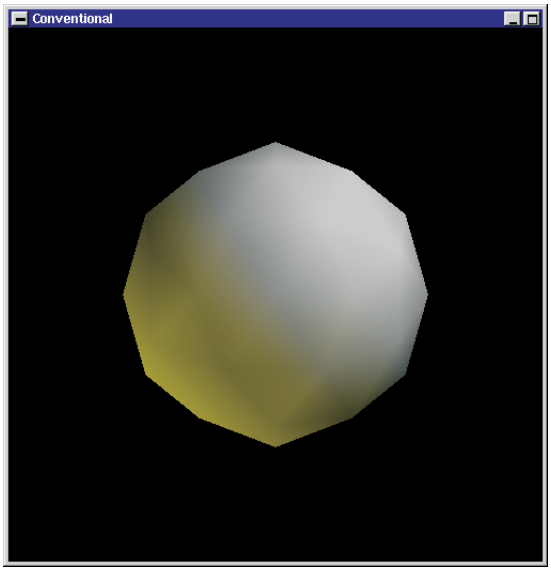
	20 face		80 faces		320 faces		1290 faces	
	Frame rendering time (microsecond)		Frame rendering time (microsecond)		Frame rendering time (microsecond)		Frame rendering time (microsecond)	
	Our rendering	Conventional rendering	Our rendering	Conventional rendering	Our rendering	Conventional rendering	Our rendering	Conventional rendering
	53113	34796	40076	34764	47016	46547	65678	70564
	53163	35233	40070	36688	47053	46558	65676	70495
	53159	35184	40125	34915	46923	48167	65622	70579
	53154	35372	40038	35018	46696	49718	67398	78894
	53131	37445	40066	35080	46723	44667	65444	72951
	53140	35678	40211	35189	46974	44767	67717	77214
	51633	35839	40254	35164	46652	44678	70634	70561
	51601	35956	40248	35163	46687	44584	70725	70602
	51569	36673	40253	35199	46815	44570	68673	70432
	55043	36198	40196	35229	46812	44554	65736	70466
	51518	36324	40216	35421	46807	44597	65696	71027
	51491	36488	40206	35156	46866	44643	65637	70462
	51454	38410	40189	35136	46537	44572	65763	70495
	51432	36662	40169	35162	46558	44544	65588	70447
	51400	36712	40157	35066	46474	44533	65416	70421
	51406	36795	40164	35012	46410	44436	65392	70386
	51375	36770	40145	37339	46566	44970	65404	70374
	51325	36785	39788	36911	46537	44434	65481	70286
	51303	36811	39554	38513	46397	44490	65438	70267
	51292	36811	39776	36831	46465	44364	65362	70185
	51288	36765	39872	36832	46471	44356	65422	70123
	51255	36745	39839	38425	46389	44218	65390	70094
	51786	36711	39787	34959	46417	44289	65383	70036
	51265	36679	39774	34888	46448	44505	65216	69894
	51273	36551	41581	34817	46480	44296	65124	69890
	51253	36469	41536	34684	46400	44276	65021	69972
	51302	36370	39692	36229	46511	44431	64953	70437
	51245	36289	41518	34580	46559	44209	64683	69780
	51269	36159	41482	34529	46475	44113	64774	69594
	51329	36056	39655	34531	46391	44116	65000	69734
	51302	36007	39529	34493	46356	44137	64995	69766
	51362	35842	39539	34453	46357	44236	65230	69933
	51503	35727	39502	34436	46308	44275	65199	69814
	51498	35581	39550	34398	46435	44343	65252	69905
	51646	35486	39583	34402	46536	44291	65650	69899
	51533	35291	39619	34334	46372	44377	65198	69955
	51830	35089	40243	34252	62326	44326	67254	69927
	51922	34938	39672	34335	46517	44391	68817	69962
	51689	36055	39719	34359	46558	45724	65424	69995
	51769	34542	39745	34437	46560	44321	65426	70045
	51849	37746	39725	34474	46421	44916	65422	70133
	51906	34061	40051	34473	46391	44328	65488	70111
	51817	33935	39848	34425	46411	44325	65549	70886
	52014	33954	39827	34436	46553	44353	65456	70328
	52151	34096	39846	35065	46460	44378	65605	70479
	52259	34217	39897	34630	46474	44385	65804	70469

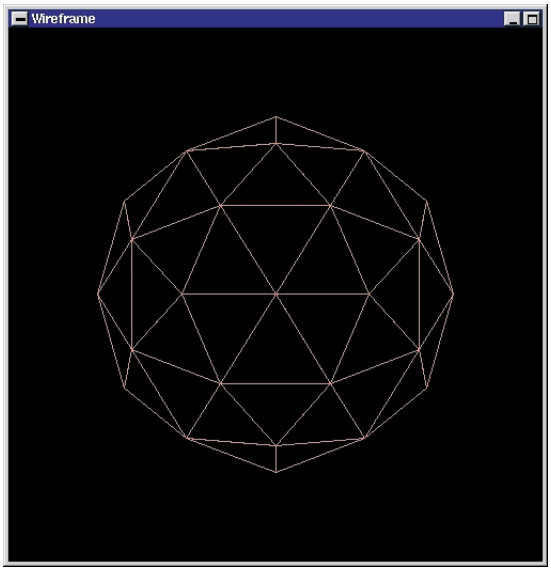
	52342	34371	39860	34713	46458	44371	65916	70522
	52400	34514	39889	34774	46446	44420	66235	70519
	52554	36335	39922	34814	46630	44462	66206	70490
	49132	34604	39974	34906	46852	44555	65919	70608
	49106	38095	40445	34991	46791	44606	65819	70621
	49147	36519	40454	35042	46882	44736	65221	70707
	49096	34768	40507	35062	47488	44786	66044	70623
	49100	34762	40383	35104	46795	44794	65679	70720
	49100	34819	40366	35112	46588	44834	66084	70669
	49073	34737	40361	35146	46824	44773	66148	70721
	49106	34717	40379	35123	46769	44773	66178	70655
	49079	34725	40348	35108	46727	44808	66060	70595
	49070	34730	40338	35111	46696	44664	65950	71381
	50315	34617	40930	35098	47016	44638	65955	70721
	50305	34541	40213	35066	46583	44775	65898	70648
	50186	34451	40197	34992	46580	44725	67338	70654
	50149	34381	40182	34911	46511	44718	66144	70688
	50196	34316	40235	34825	46482	44682	66199	70730
	49981	34208	40184	34677	46457	45284	66071	70749
	50068	34096	40171	34704	46616	44615	65847	70795
	49925	34151	40186	34727	46637	44636	65932	70811
	49841	34263	40239	34787	46915	44724	65994	70746
	49162	34381	40176	34799	46953	44698	66272	70804
	49997	34479	40160	34835	46655	44736	66316	70709
	49915	34567	39848	34845	46797	44804	66810	70780
	49908	34639	40016	34836	46768	44734	66340	70759
	49809	34666	40022	35482	46798	44680	66076	70753
	49773	34712	40025	34863	46842	44719	66140	70809
	49723	34768	40084	34817	47297	44666	66094	71320
	49685	34839	40004	34769	46798	44672	66512	70726
	49769	34836	40008	34830	46731	44666	66467	70503
	49704	34820	39761	34881	46658	44715	66343	70460
	49668	34815	39819	35128	46834	44631	66100	70511
	49660	34822	39761	34877	46780	44610	65818	70433
	49630	34765	39768	35079	46840	44636	65658	70045
	49625	34731	39914	34796	47056	44551	65910	70413
	50162	34677	39897	34754	47120	44798	66249	70533
	49613	34618	39929	34736	47013	44493	66444	70505
	49572	34501	39922	34668	47006	44250	66572	70543
	49612	34344	39914	34560	46923	44488	66347	70644
	49598	33929	39926	34233	47028	44520	66291	70730
	49629	35818	39950	34541	47076	44593	66090	70697
	49603	34548	40006	34658	47099	44583	66649	70755
	49857	34600	39937	34778	47106	44506	66273	70746
	49627	34667	39953	34817	46807	44611	66386	71401
	49689	34711	39799	34828	46776	44717	66458	70703
	49655	34741	39853	34843	46845	44989	66471	70829
	49603	34765	39807	34847	46640	44674	66281	70987
	49682	34781	39801	34853	46716	44639	65936	71065
	49701	34847	40022	34849	46749	44638	66051	70758
	49716	34765	40081	35585	46739	44649	66127	70707
	49776	34733	43410	34777	46826	44716	66258	70740
	49742	34671	43498	34805	46873	44791	66313	70720

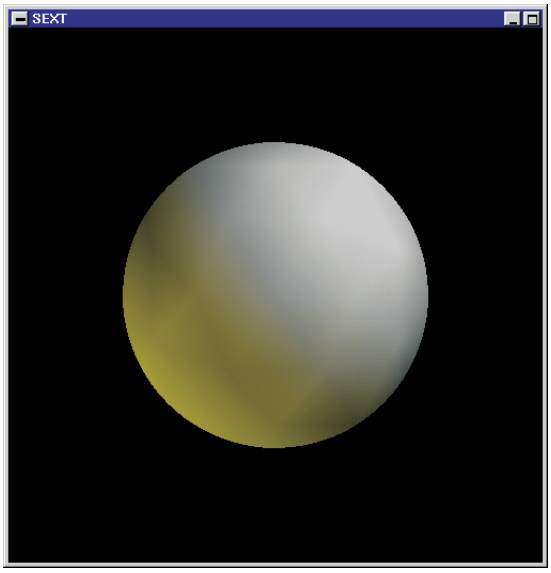
	49801	34695	40026	35439	46854	44727	66102	70731
Avg.	50739.54	35312.74	40133.92	35030.33	46852.9	44705.87	66042.06	70664.61
Frame rate (frames per second)	19.7085	28.31839161	24.91658	28.5466908	21.3434	22.3684272	15.14187	14.15135525

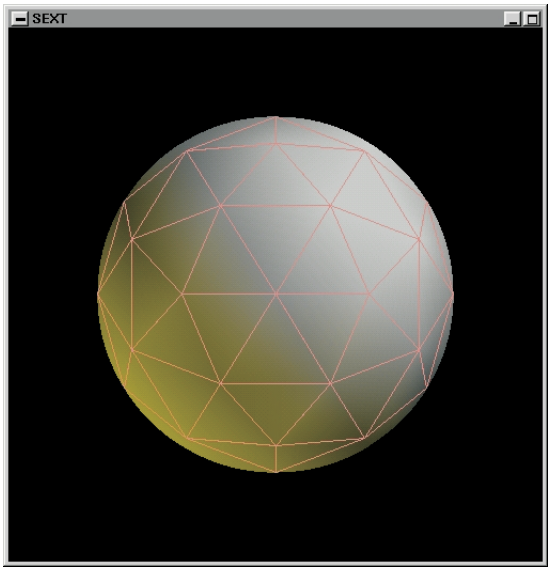
15. Color Plates

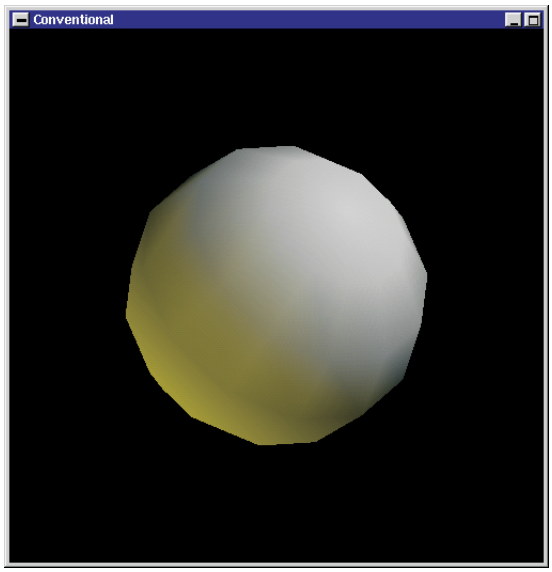
The Color Plates are such that each page shows a simple surface in an arbitrary orientation rendered by our algorithm. For each page, the top left image is rendered by the conventional rendering algorithm, and the top right image shows the wireframe. The bottom left one is rendered by our algorithm, and the bottom right one shows the same thing but with a wireframe (not perturbed) superimposed on it.

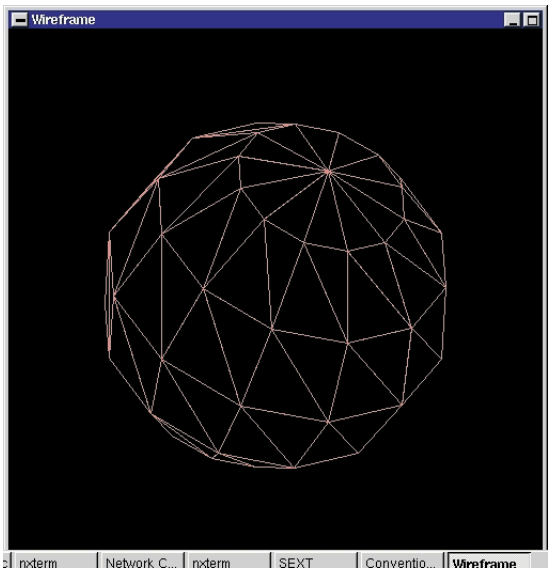

Conventional

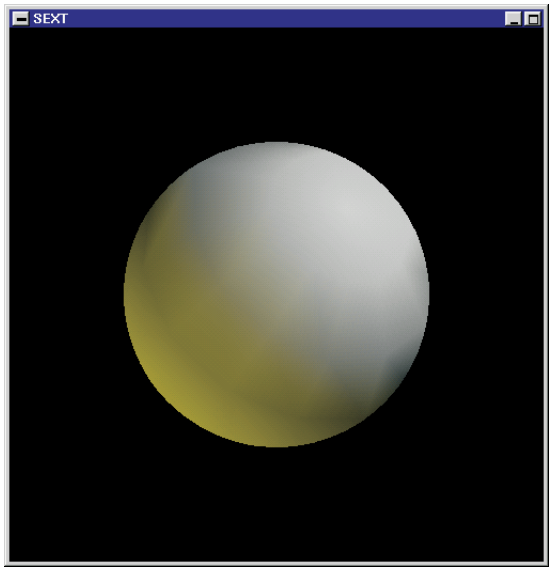

Wireframe

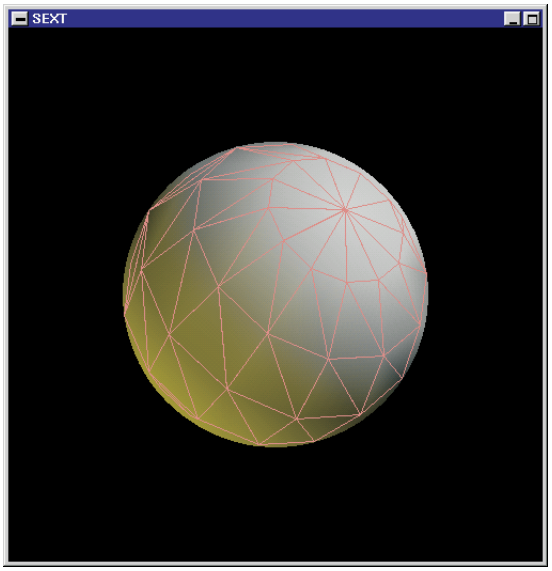

SEXT

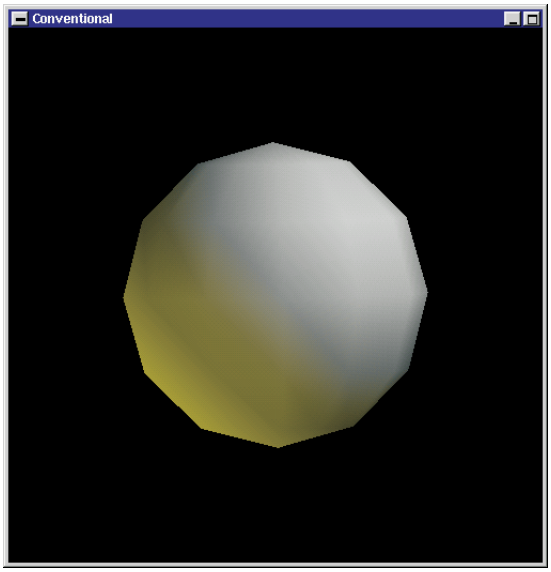

SEXT

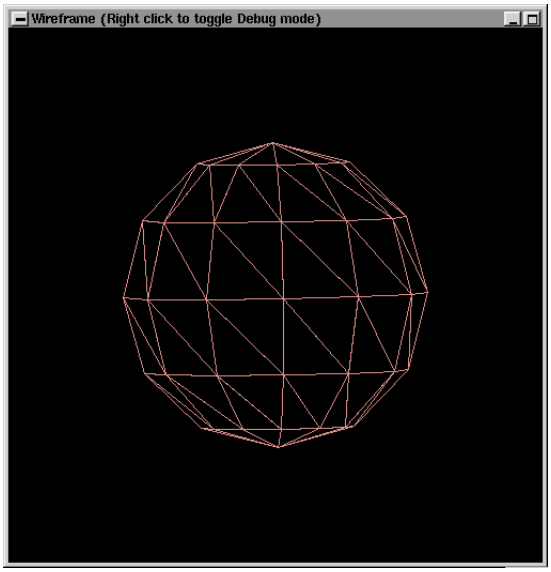

Conventional

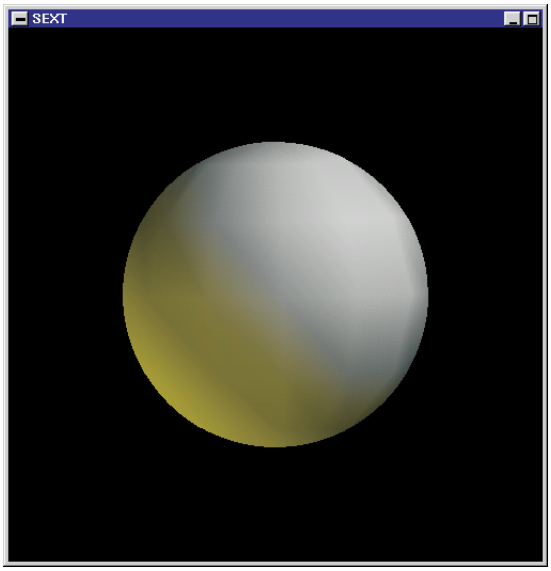

Wireframe
nxterm
Network C...
nxterm
SEXT
Conventio...
Wireframe

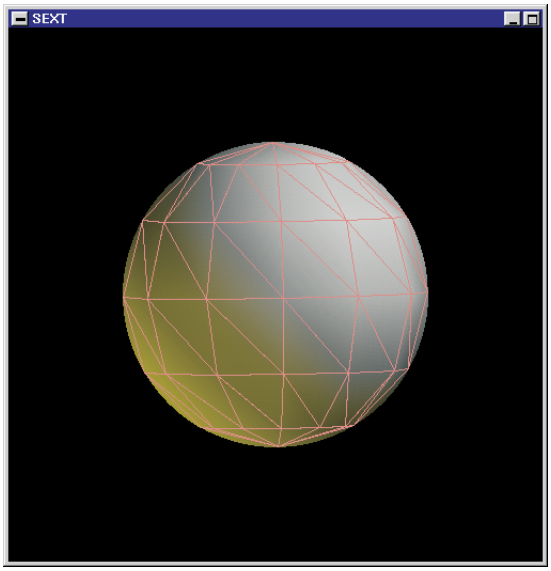

SEXT

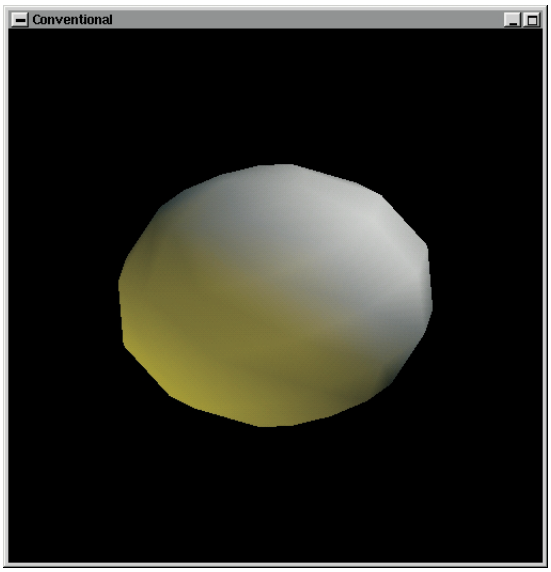

SEXT

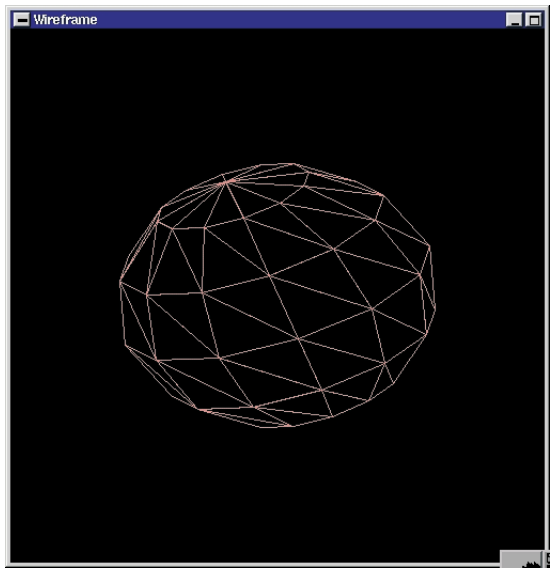

Conventional

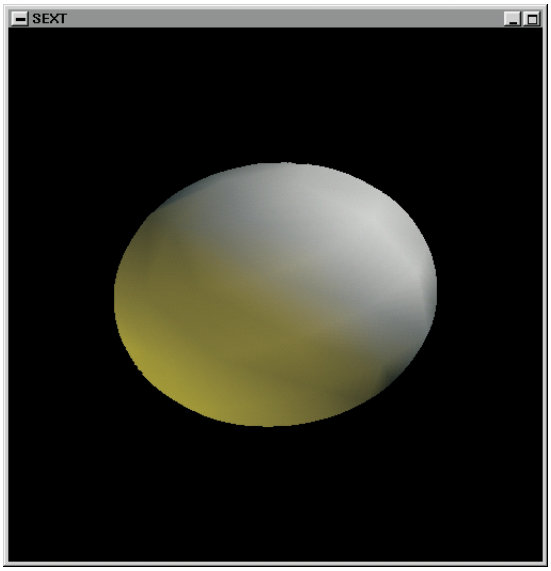

Wireframe (Right click to toggle Debug mode)

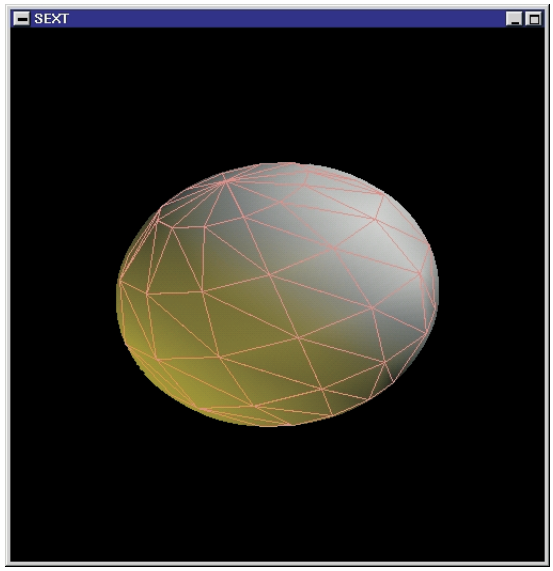

SEXT

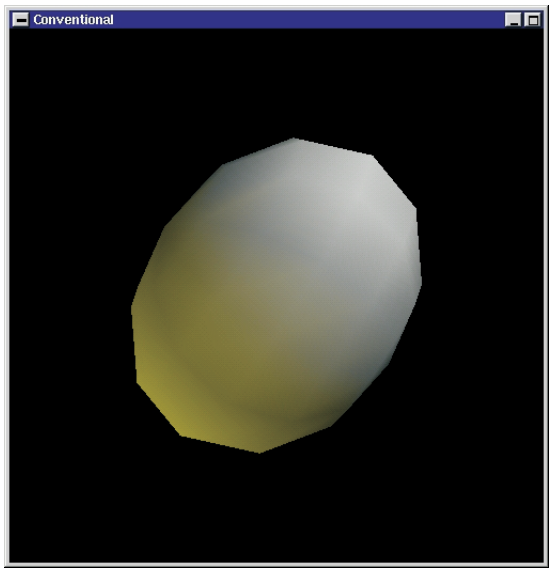

SEXT

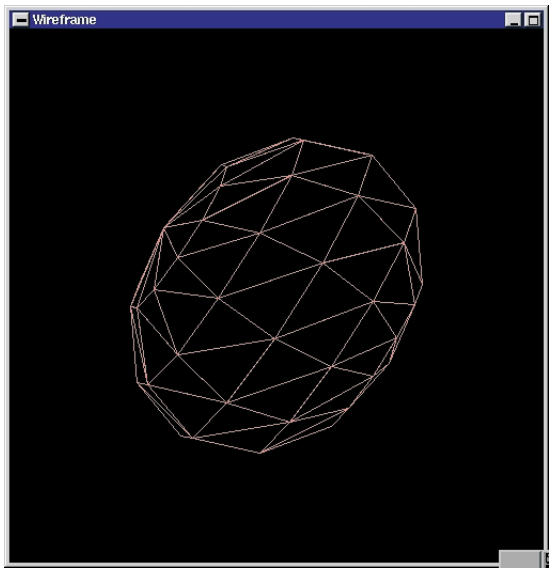

Conventional

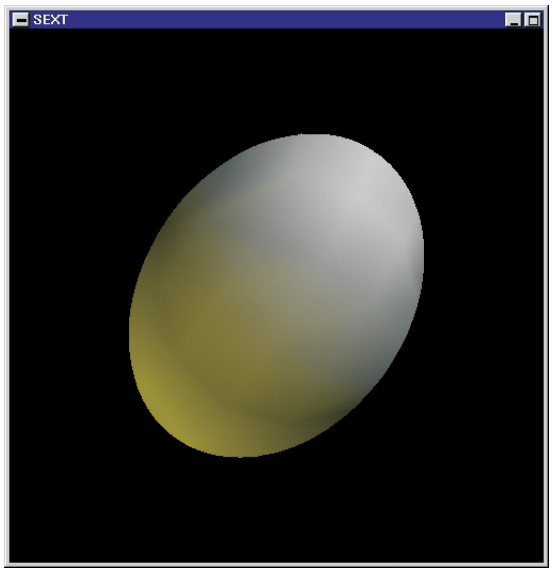

Wireframe

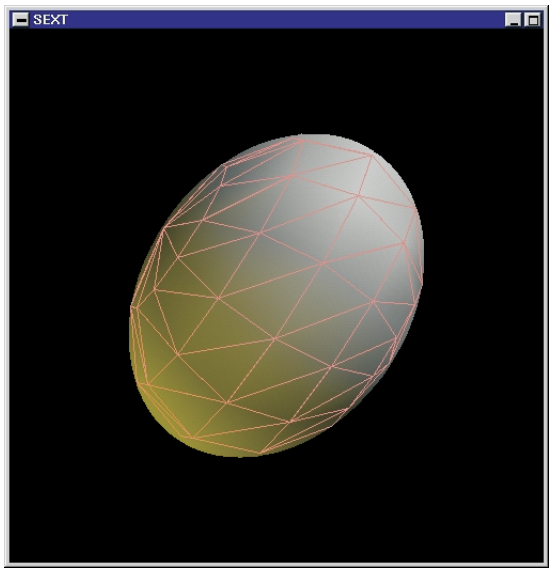

SEXT

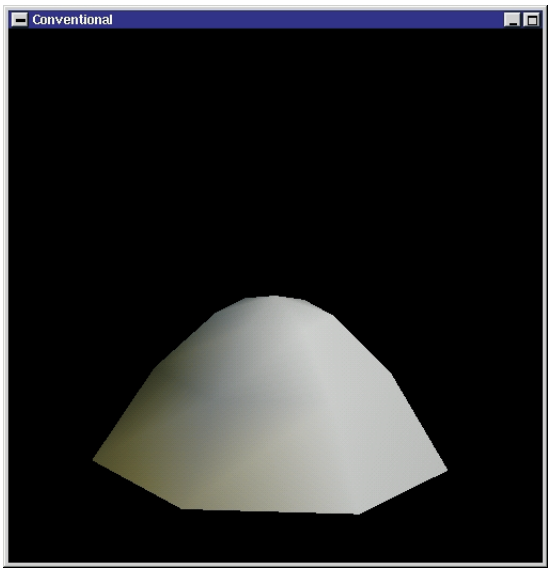

SEXT

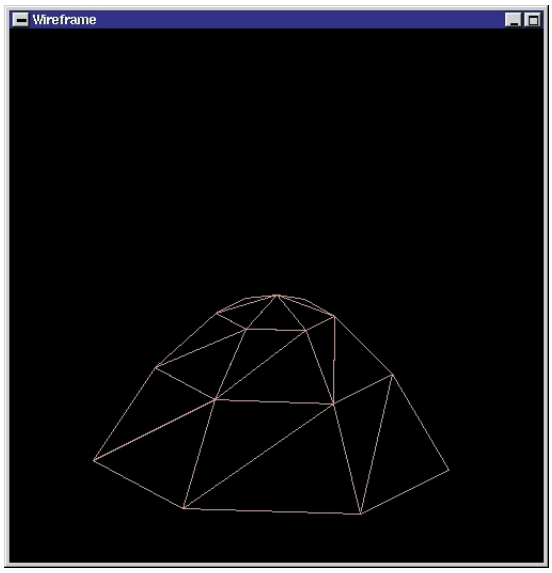

Conventional

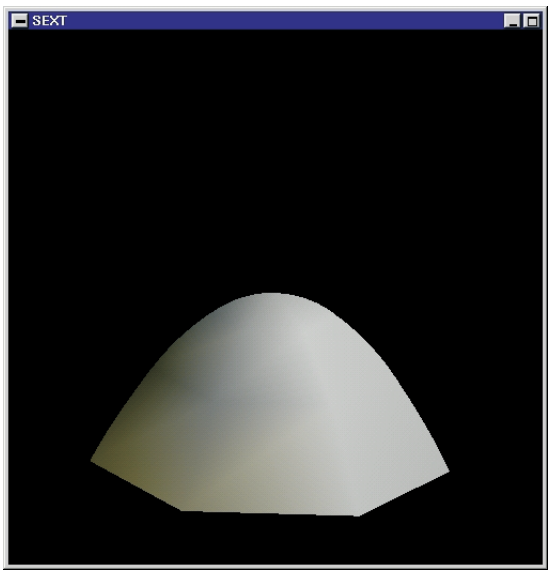

Wireframe

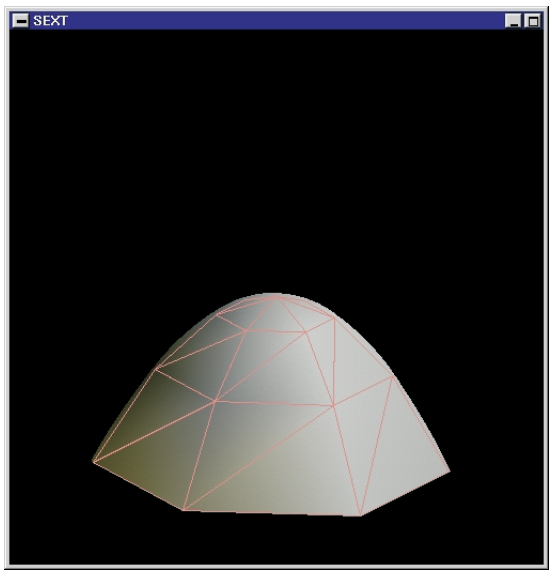

SEXT

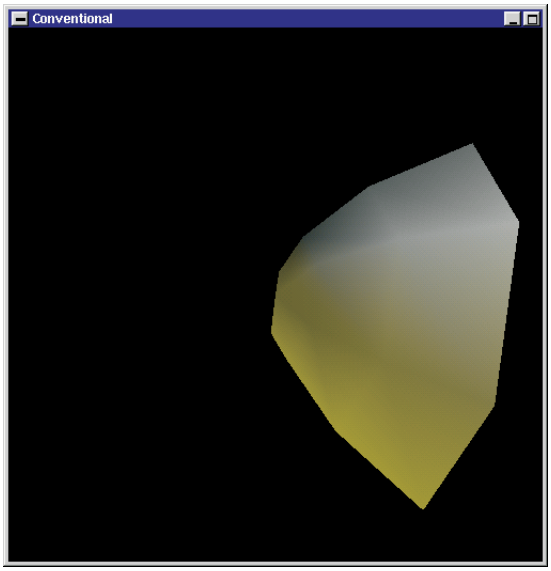

SEXT

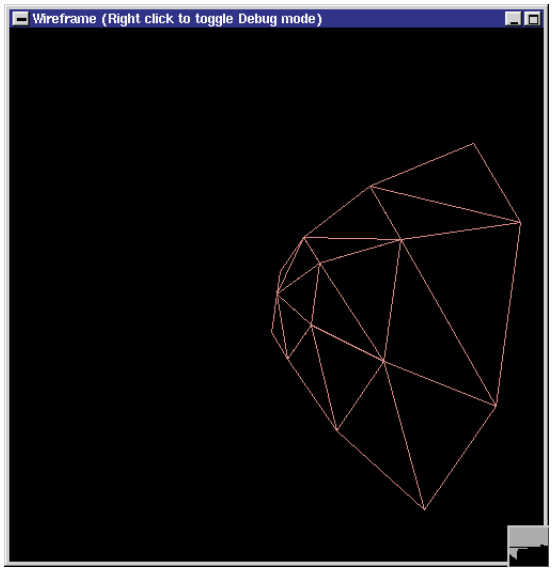

Conventional

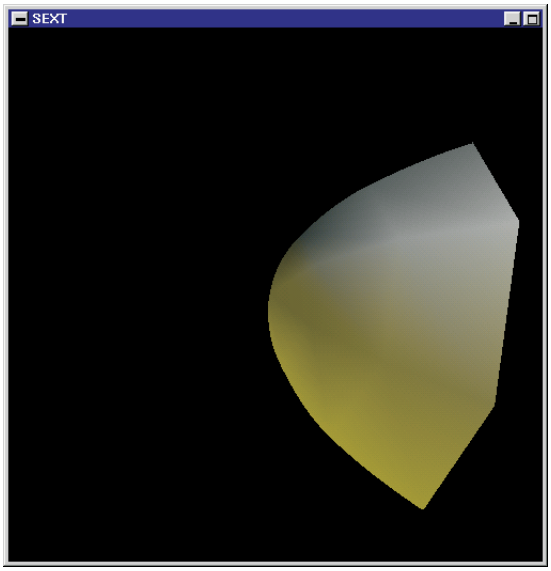

Wireframe

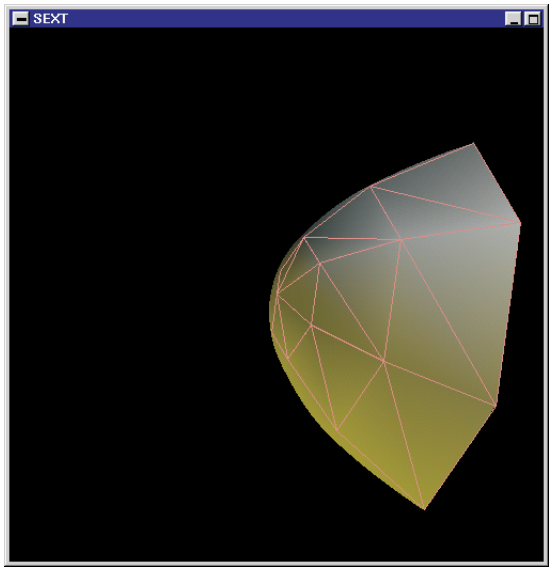

SEXT

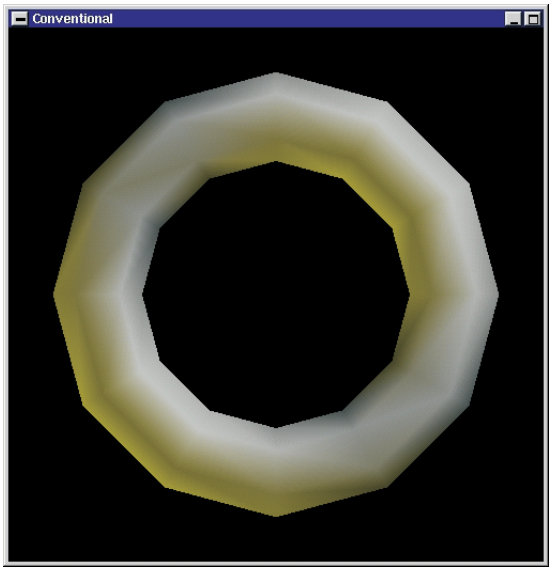

SEXT

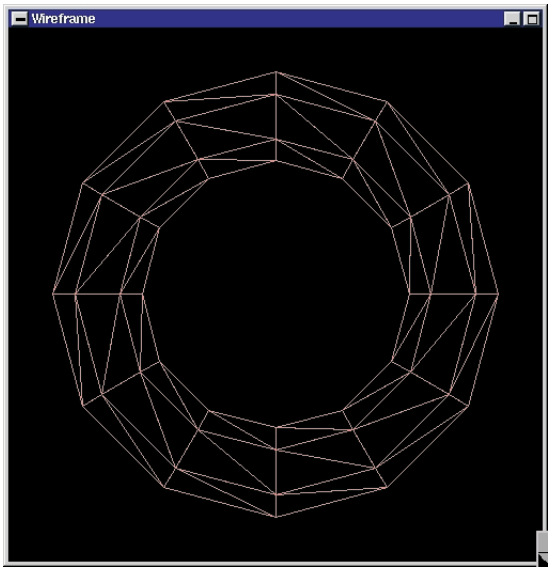

Conventional

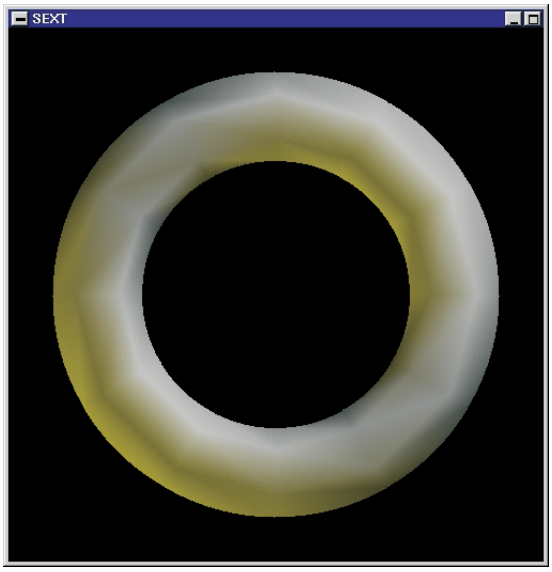

Wireframe (Right click to toggle Debug mode)

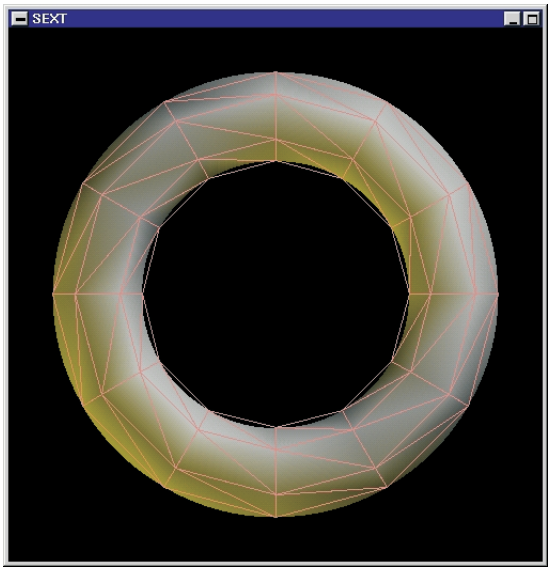

SEXT

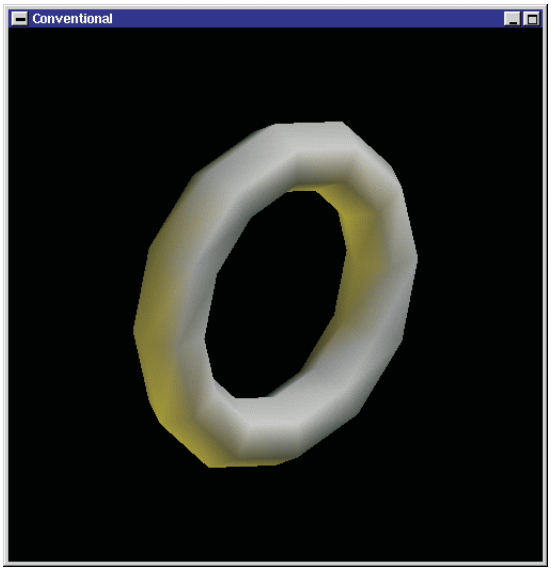

SEXT

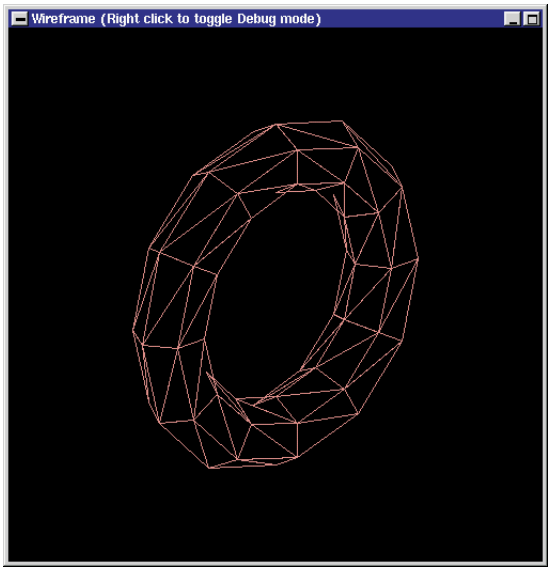

Conventional

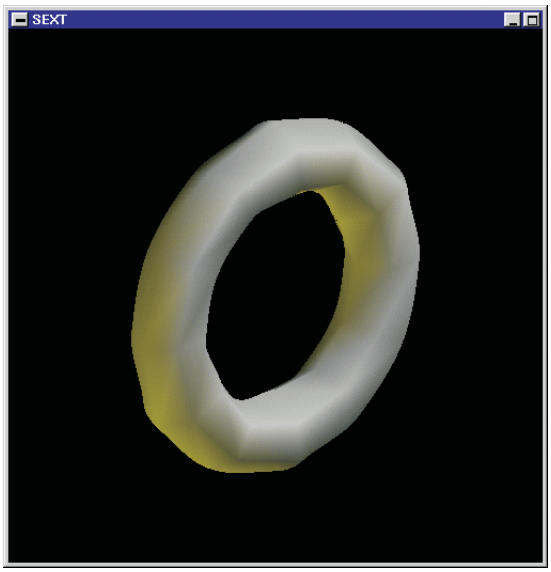

Wireframe

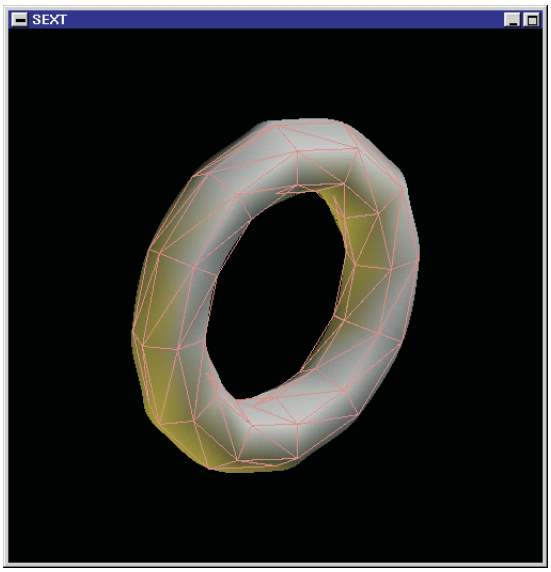

SEXT


SEXT


Conventional


Wireframe (Right click to toggle Debug mode)


SEXT


SEXT