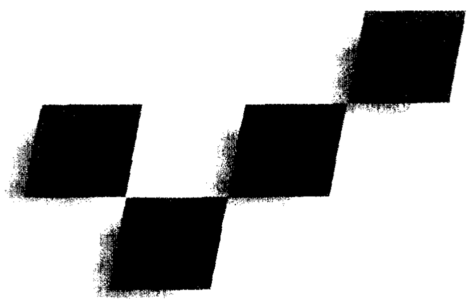


High-Quality Polygon Edging

Russ Herrell
Hewlett-Packard Company

Joe Baldwin and Chris Wilcox
Cyrix Corporation

Applications combining wireframe and shaded primitives often produce unwanted artifacts and thus poor-quality polygon edging. This algorithm uses an edging plane to generate clean edges.

With workstations providing more sophisticated graphics, many applications are progressing from wireframe rendering to shaded images of surfaces. While smooth shading provides increased realism and the ability to visualize more complex models, some of the advantages of the wireframe representation may be discarded in the process. To benefit from the best of both methods, workstation applications often combine wireframe and shaded primitives in the same image.

Wireframe can overlay a shaded surface to provide structural information, such as the original data points that generated the model. When a model is constructed using splines, the wireframe can show either spline patch boundaries or the control points used to calculate the spline surface. Another benefit of wireframe is selective highlighting of a subset of the model, such as a subassembly within a complex mechanical part. This information provides valuable insight to designers.

Applications use several methods to simultaneously display wireframe and shaded primitives. However, unless the underlying hardware actually scan converts polygons and their edges at the same time, most methods rely on vector primitives to draw the wireframe portions and polygon primitives to draw the surfaces. These separate scan conversion operations use different algorithms for each type of primitive.

Although drawing vectors between polygon vertices appears to be an easy way to highlight borders, standard graphics techniques can produce many unwanted artifacts. Scan conversion algorithms often generate significantly different z-buffer values for polygon fill pixels and the associated vector (edge) pixels. Where edge vectors overlap polygon fill, inconsistent z-buffer values can cause edge pixels to be overwritten by fill. The result is poor-quality polygon edging, a pervasive problem in graphics hardware.

We have developed an algorithm that uses an edging plane to generate high-quality polygon edges when using separately drawn vectors to highlight polygon borders. The algorithm maintains the subpixel adjustments necessary for accurate polygonal rendering. The edging plane is simple to incorporate into existing software and hardware scan convertors.

In this article we describe the problem of edging a single polygon, then extend the discussion to multi-polygon surfaces and multiple component assemblies. We introduce our algorithm and compare it to existing methods for improving polygon edges.

The problem

When hidden surface removal is enabled, the z-buffer should correctly combine polygons with their edges. Unfortunately, scan conversion can produce unwanted artifacts during the sequential rendering of a polygonal surface and associated edge vectors. Because the z-buffer values for these primitives usually differ slightly, polygon fill pixels and edge pixels can interfere with each other. When edge pixels are overwritten by fill pixels, the resulting polygon edge will have missing pixels. We call this *polygon edge stitching*. A similar problem occurs when highlighting vectors are sent after the original, edged model is complete. If the edging algorithm has left interfering values from the polygon fill in the

Applications

To help elucidate the applications significance of the algorithm described in "High Quality Polygon Edging," see the Applications sidebar at the end of the article on pages 73 and 74.

0272-17-16/95/$4.00 © 1995 IEEE

z-buffer, then highlighting vectors will again conflict to produce edge stitching.

Why does the scan conversion process produce these inaccuracies? Even though vector and polygon scan conversion algorithms both need subpixel accuracy, they use this precision differently. The fill *z*-buffer values are subpixel adjusted according to the surface gradient of the polygon. However, vector *z*-buffer values are calculated by simply interpolating between two endpoints. The vector generator does not have *z*-slope information for the polygon surface. These inaccuracies result in a mismatch between the *z*-buffer values for the polygon and its edges.

We propose an algorithm that uses multiple rendering passes to draw polygon edges with minimal stitching. We avoid missing edge pixels by ensuring that polygon fill and edge *z*-buffer interactions do not occur. The technique leaves the *z*-values for the vector edge in the *z*-buffer, coinciding with the polygon border visible in the frame buffer. This method has the distinct advantage of allowing the application to re-render vectors on polygon edges with minimal stitching even after the original polygonal rendering of the entire edged model is complete.

Edging algorithms

A simple algorithm for polygon edging requires two passes. First the polygon fill is rendered, then the vectors describe the polygon edge. The same vertices are sent for polygon fill and edge vectors, but the different rendering algorithms used result in the problems mentioned above. Figure 1a shows a teapot rendered with simple edging. Figure 1b magnifies the highlighted area of Figure 1a, where polygon edges intersect fill. This image maps the *z*-values along the *z*-axis. Notice the missing edge pixels that did not pass *z*-compare. The *z*-values are exaggerated in the magnified image to contrast the differences between edging algorithms.

Z-buffer adjustment method

Another often attempted solution to the *z*-buffer intersection problem is to pull the polygon edge *z*-buffer values forward towards the viewer. The implementation is identical to simple edging except that the transformation matrix is slightly modified to pull the *z*-buffer values forward. This transformation is used only for the polygon edges, so the fill pass produces the same *z*-buffer values as before.

While this technique can be made to work for certain models, it has serious drawbacks as a general solution. One difficulty is determining the amount of *z*-buffer adjustment. Apply too much correction and polygon edges will poke out into other objects in the scene. If not enough adjustment is made, polygon edges will still exhibit missing pixels. When the magnitude of the adjustment is computed based on the entire *z*-buffer dynamic range, objects that use only a small portion of the range may be adjusted too much. Moreover, computing the *z*-buffer adjustment for each primitive adds unacceptable overhead to the polygon edging algorithm. Even ignoring performance concerns, we have not discovered any reliable method for calculating the *z*-buffer

Previous work

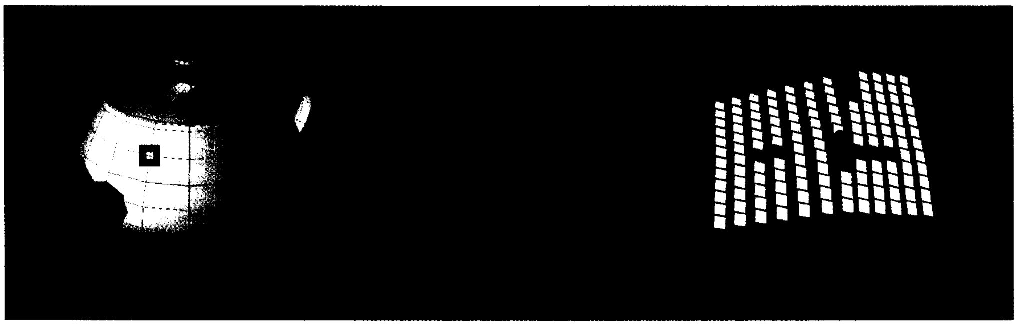
Our previous graphics hardware had few problems with edging. The interpolator described by Roger Swanson and Larry Thayer[1] generated edge pixels that exactly corresponded to the polygon border at the same time it scan converted the polygon. However, there was no subpixel adjustment in any direction, so many advanced features such as antialiasing, alpha blending, and texture mapping were not possible.

The more recent scan conversion hardware described by Tony Barkans[2] supported all of these features but exhibited the polygon edging problems described above because it no longer generated edge pixels at the same time it filled the polygons. We have since implemented several products that minimize polygon edging artifacts by using the methods described in this article.

Other previous algorithms for polygon edging were presented in a paper by Kurt Akeley[3], which describes various *z*-buffer techniques. Akeley showed a five-pass algorithm that creates hollow polygons but requires specialized *z*-buffer hardware. The hollow polygon scheme also allows the edge pixel *z*-values to be replaced by fill pixel *z*-values.

References

1. R. Swanson and L. Thayer, "A Fast Shaded-Polygon Renderer," *Computer Graphics* (Proc. Siggraph), Vol. 20, No. 4, Aug. 1986, pp. 95-101.
2. A. Barkans, "High-Speed, High-Quality Antialiased Vector Generation," *Computer Graphics* (Proc. Siggraph), Vol. 24, No. 4, Aug. 1990, pp. 319-326.
3. K. Akeley, "The Hidden Charms of *Z*-buffer," *Iris Universe*, Vol. 11, Spring 1990, pp. 31-37.


1 Teapot rendered with (a) simple polygon edging and (b) a magnified view.

2 Teapot rendered with (a) *z*-buffer adjustment method and (b) a magnified view.

adjustment. Figure 2a shows the teapot rendered with the *z*-buffer adjustment method. We have pulled the *z*-buffer values far enough forward to improve most of the model, but some edges are still stitched in areas with a high gradient along the *z*-axis. Figure 2b shows that the edge pixels have moved above the plane of the fill pixels.

Decal-edging method

A more elegant method of edging polygons, which we call the decal-edging method, requires three rendering passes. The algorithm draws the polygon edges before replacing the *z*-buffer values for the polygon fill. This procedure prevents the polygon fill from interfering with the polygon edge. After both the fill and edge pixels are drawn correctly, the fill *z*-buffer values are rendered. The algorithm is as follows:

Pass 1. Render the polygon fill with *z*-buffer comparison enabled, *z*-buffer replacement disabled, and frame-buffer writing enabled.

Pass 2. Render the polygon edges with *z*-buffer comparison enabled, *z*-buffer replacement enabled, and frame-buffer writing enabled.

Pass 3. Render the polygon fill with *z*-buffer comparison enabled, *z*-buffer replacement enabled, but frame-buffer writing disabled.

The decal method is really a painter's algorithm for the polygon fill and edge. The fill is painted before the edge, and *z*-buffer comparisons between the two primitives are avoided. The resulting image in the frame buffer looks good, as in Figure 3a, but the *z*-buffer values do not match what is visible in the frame buffer. Decal edging overwrites some of the edge *z*-values with fill *z*-values in the *z*-buffer during the third pass. These *z*-buffer values might interfere with highlighting vectors rendered after decal edging is completed.

Edging-plane method

Our edging-plane algorithm solves many of these problems. We use this algorithm during software rendering in our graphics libraries, but it can also be implemented in hardware for minimal cost. The algorithm requires three passes, as with decal edging. The basic idea is to use an extra plane to keep track of polygon fill pixels. When edges are drawn on top of polygon fill, this condition is detected, and the *z*-buffer comparison of the edge pixel is forced to pass. The result is that all edge pixels are drawn, and the edge *z*-buffer values are left in place for future rendering. The algorithm starts with the edging plane bits all cleared to zero.

Pass 1. Render the polygon fill with *z*-buffer comparison enabled, *z*-buffer replacement enabled, and frame-buffer writing enabled. The edging plane is set to 1 for every pixel in the polygon fill that passes *z*-compare.

Pass 2. Render the polygon edges with *z*-buffer comparison enabled, *z*-buffer replacement enabled, and frame-buffer writing enabled. If the underlying pixel has an edging plane value of 0, normal *z*-buffer comparison is performed. If an underlying pixel has an edging plane value of 1, the *z*-buffer comparison is forced to pass the edge pixel.

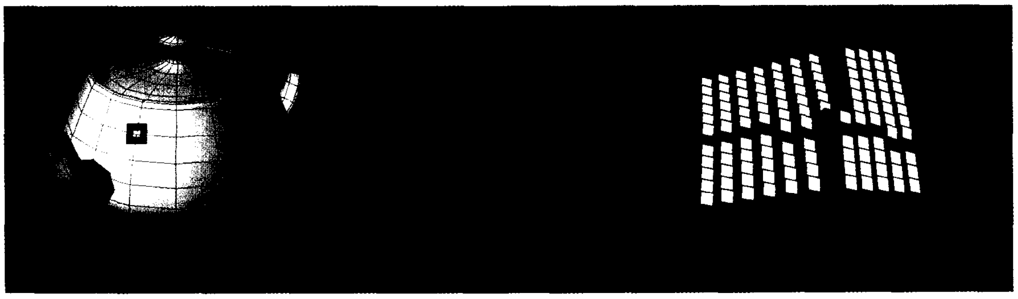
Pass 3. Clear the edging plane underneath the polygon fill by setting the edging plane value to 0. This pass does not require any other *z*-buffer or frame buffer operations.

The first pass of this algorithm is a normal rendering pass of the polygon fill, except that the edging plane bit is set for each pixel of the polygon's fill. The second pass is a normal rendering of the edge vector pixels, except that edge pixels overlapping fill pixels are guaranteed to win the *z*-compare. The third pass simply reinitializes the edging plane for the next polygon.

Figure 3a shows the teapot rendered with the edging plane method. The magnified image in Figure 3b shows that the *z*-buffer values for the edge pixels remain, even where the edge is farther away than the fill. The *z*-buffer values are again exaggerated to show the *z*-values left by the edging-plane algorithm.

Edging-plane software. We can easily integrate the edging plane with existing software scan-conversion algorithms. Our implementation allocates the edging plane in virtual memory as a packed array, with a single bit per pixel. The edging plane can be created separately for each window or shared for the entire display area. The edging plane could also be implemented as an additional bit of the software *z*-buffer array.

When polygon edging is enabled, software multipass-scan-conversion routines are substituted into the rendering pipeline. These routines compute the bounding box for the polygon and then render the first pass of the edging-plane algorithm. The polygon fill is drawn into the frame buffer. Hidden surface removal is performed with a software *z*-buffer, and the edging plane values are set whenever a fill pixel is drawn. The second pass



3 Teapot rendered with (a) edging-plane method and (b) a magnified view.

draws polygon edges in the frame buffer using a modified z- buffer comparison rule. The final pass in the software implementation clears all the edging plane values underneath the bounding box of the polygon. This is considerably faster than rendering the polygon again.

Edging plane hardware. We could obtain a hardware edging plane by adding one more z-buffer plane or by sacrificing one bit of z-buffer resolution. We didn't like the costs of either option. Our implementation of edging planes shares the bit plane in the z-buffer already used as a virgin plane to provide fast z-buffer clears. To illustrate how we can use the same bit plane for two different functions requires an explanation of our scheme for fast z-buffer clears.

The fast z-buffer clear sets all virgin bits within the area being cleared. This method is faster than clearing the entire z-buffer, since only a small amount of memory needs to be accessed. During rendering, z-buffer comparisons look at the virgin bit for each pixel. If the bit is set, the remaining z-buffer value is ignored and z-buffer replacement is forced to pass. If the bit is clear, a normal z-buffer comparison occurs.

Our edging-plane algorithm uses this virgin plane hardware. When polygon edging is active, it selects a hardware multipass-scan-conversion routine. This routine differs slightly from the software edging plane.

Pass 1. Render the polygon fill with z-buffer comparison enabled, z-buffer replacement enabled, and frame-buffer writing enabled. The hardware sets the virgin bit to 1 for every pixel in the polygon fill that passes z-compare.

Pass 2. Render the polygon fill with z-buffer comparison enabled, z-buffer replacement enabled, and frame-buffer writing enabled. This pass uses our normal z-buffer comparison which forces the z-buffer replacement if the virgin bit is set.

Pass 3. Render the polygon fill with z-buffer comparison, z-buffer replacement, and frame-buffer writes all disabled. The hardware simply clears the virgin bit to 0 for every pixel in the polygon fill.

This algorithm gives us clean polygon edges, leaves the vector z-values in the z-buffer, and uses existing z-buffer virgin bits. Here's how it works:

While rendering the first pass of polygon fill, the z-compare hardware can encounter both virgin and nonvirgin pixels. The fill pixels will always win the z-buffer compare against virgin pixels and will compare normally against nonvirgin pixels. So regardless of the outcome of the z-compares, there will be no virgin pixels in the fill region after this pass completes. Hence, no information is stored in the virgin bit underneath the fill, and we can borrow the virgin bits temporarily to use as an edging plane. The only requirement is to perform all three passes on a polygon before proceeding to the next polygon. This is a better solution than reducing the z-buffer resolution.

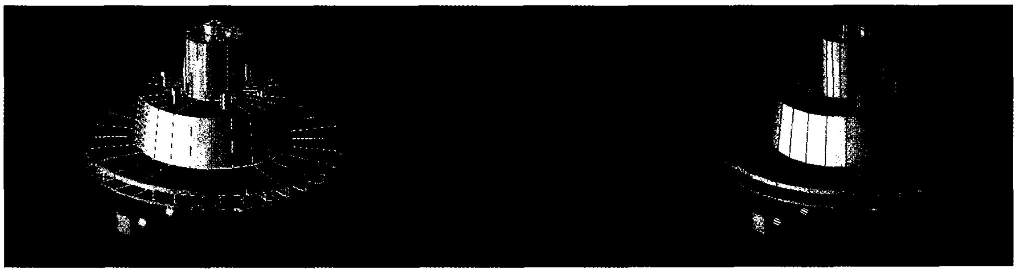
While rendering the polygon edge during the second pass, the hardware can still encounter virgin and nonvirgin pixels. If nonvirgin pixels are left from another primitive, the edge pixels will z-buffer compare against them in the normal manner. Real virgin pixels left from the fast z-buffer clear are found only in regions where the edge does not overlap the fill. The edge pixel will always win the z-compare against these pixels, which is the desired result. Finally, the pixels that overlap the fill from the current polygon will have the virgin bit set, since we borrowed the virgin plane inside the polygon fill to use as an edging plane. In this case also, the edge pixels will win the z-compare, thereby avoiding interference from the fill and leaving the edge z-buffer values along the polygon border.

While re-rendering the fill pixels on the third pass, none of the normal z-buffer or frame buffer writes are enabled. This pass simply clears all the borrowed virgin bits, leaving the correct z-values in the z-buffer. This pass again relies on the fact that no real virgin bits exist within the fill area after the first pass.

Edging multiple polygons

Of course, many polygons are required to render a typical surface, object, or assembly. To avoid gaps in the rendering of an object, databases are designed so that the polygons of a surface share adjoining edges. Since fill pixels do not extend past the true boundary of a polygon,[1] fill pixels from adjacent polygons will not interfere with one another. However, the associated edge pixels often do lie outside the polygon boundary. Thus, edge pixels from a polygon might need to overlay the fill pixels of an adjacent polygon.

As long as both adjacent polygons are edged, and as long as the shared edge is drawn in the same direction each time, the edging-plane algorithm will produce clean edges and the associated z-buffer values will still match the frame buffer results. The first polygon drawn will leave the z-values for the edge in its own fill region. When the second polygon is drawn, these edge pixels



4 Brake assembly rendered with (a) decal edging and highlighting and (b) edging-plane highlighting.

will either not be disturbed, or will be redrawn with the same values.

A problem occurs when one assembly is edged and another is not. The intersection edge may exhibit stitching as the fill pixels from the non-edged polygons can now interfere with the edge pixels from the edged polygons.

Also, edges of polygons that are behind other polygons and not usually visible might poke through. This is really a z-resolution problem, aggravated by edging and not unique to our algorithm.

Results

Leaving the vector z-buffer values along the polygon edges is one of the major benefits of this algorithm because vectors can now be re-rendered along the polygon edge at a later time. Some applications use this to highlight substructures after rendering the original, edged model. These applications could redraw the entire model with different colored edges, but the performance is much better when redrawing only the highlight vectors. Figure 4a shows a brake assembly originally rendered using decal edging. The white polygon edges were subsequently overlaid with red vectors using the same vertices. The decal method edged the original polygons satisfactorily, but failed when the highlight vectors were later drawn. Figure 4b shows the same image rendered with the edging-plane algorithm, with highlight vectors added on a later pass.

Real applications typically highlight sub-assemblies and individual components. Redrawing only the highlighted edges will produce good quality edges on the highlighted assembly itself. If the entire image was originally drawn with edged polygons, which is often the case, the highlighted vectors along the intersection edges will also be free of stitching. Problems occur mainly when unedged polygons meet edged polygons.

Performance of the edging-plane algorithm is similar to the performance of the decal algorithm — about one third of the scan conversion rate for filled polygons without edges. The performance degradation is caused by rendering the polygon fill twice, plus drawing the edges and changing pipeline control between the passes.

Future work

The method presented here works well for polygon edging because it guarantees that the edge z-values win over the fill z-values. This is a desirable result in most cases, but not all. A better solution would be a scan conversion algorithm that produces exactly identical z-buffer values for edge and fill pixels. This accuracy can be achieved by developing algorithms that generate the same z-values for vectors and polygons that share vertices. This perfection may be difficult to accomplish while still correctly rendering polygons with subpixel accuracy. Another proposed solution is to edge polygons with thin polygons that share the same z-buffer gradient. Of course, when edge pixels stray outside the boundaries of the polygon, other artifacts will still be created.

Finally, it is possible to incorporate drawing edge pixels directly into the scan conversion hardware, as described previously. This would add considerable complexity to future products at a time when our scan conversion hardware is evolving toward less-complicated primitives with higher performance. If the scan converter generates edge pixels, then decomposition of more complex polygons would require an expensive burden of edge information to be carried along with each primitive in the pipeline.

The edging-plane algorithm can also be applied to the more general z-buffer decal problem described by Akeley. Since the basic surface and the decal are not necessarily the same primitives, true coplanar scan conversion can be even more difficult to achieve. Again, the edging plane's main advantage is that it leaves the depth values of the decal primitive in the z-buffer so that this same decal can be reapplied later without re-rendering the background pieces. This feature is useful when texture mapping various images onto a larger background surface, such as when rendering a cockpit control panel. Individual gauges, for example, could be texture mapped onto the panel's surface, and multiple textures can be applied as the instruments' indicated values change without disturbing the rest of the panel.

Conclusions

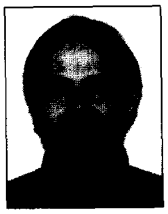
Even as applications migrate from wireframe to shaded polygons, the merging of wireframe information into shaded images still benefits many applications. The edging-plane algorithm generates high-quality edges in this heterogeneous environment without forcing substantial penalties on performance or complexity, and without creating inconsistencies between the z-buffer and frame buffer contents. ■

Acknowledgments

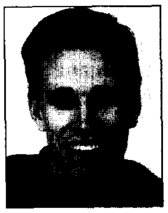
Many engineers at Hewlett-Packard have worked on polygon edging solutions. We would especially like to thank Alan Krech, Noel Scott, and Ted Rossin for their ideas and reviews of this article. Thanks also to John Fujii for help with the pictures.

References

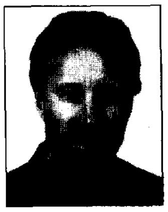
1. F. Norrod and L. Thayer, "An Advanced VLSI Chip Set for Very High-Speed Graphics Rendering," *Proc. NCGA 91*, NCGA, Fairfax, Va., 1991, pp. 1-10.



Russ Herrell *is a research and development engineer with Hewlett-Packard's Graphics Hardware Lab in Fort Collins, Colorado. He received BS and MS degrees in electrical engineering from Montana State University in 1980 and 1982.*



Joe Baldwin *is a senior software engineer with Cyrix Corporation in Longmont, Colorado. He contributed to the edging algorithm development while working in Hewlett-Packard's graphics software development organization in Fort Collins, Colorado. Baldwin graduated from Montana State University with a BSEE in 1982.*



Chris Wilcox *is a senior software engineer for Cyrix Corporation. He worked on this article while employed by Hewlett-Packard in Fort Collins, Colorado. Wilcox received a BS and MS in computer science from the University of Utah.*

Readers may contact Herrell at Hewlett Packard, Mailstop 73, 3404 E. Harmony Rd., Ft. Collins, CO 80525, e-mail russ@fc.hp.com.

Applications

The following applications elucidate the significance of the edging-plane algorithm described in "High Quality Polygon Edging" by presenting two implementations of it—one independent of Hewlett-Packard hardware and software libraries, the other within a current HP product.

An OpenGL implementation

Ken Tidwell, *HP Graphics Software Lab*

The edging-plane algorithm can be implemented using OpenGL. The OpenGL implementation described below uses a stencil plane as the edging plane. However, because edging-plane functionality is not completely available in an OpenGL stencil, the second pass of the edging-plane algorithm takes an extra step. The entire process comprises four steps.

1. Step 1 implements pass 1 of the edging-plane algorithm. This step assumes that the stencil is cleared to 0s and that depth and stencil testing is enabled. The following function calls are made:

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

// Draw polygon
```

These calls create a mask of the depth-buffered polygon in the stencil plane.

2. Step 2 implements the first condition of pass 2 of the algorithm. If a pixel has a value of 0 in the stencil plane, it will have to pass the depth test in order to be drawn. The stencil is left untouched in this step.

```
glStencilFunc(GL_NOTEQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// Draw polygon edge as a line
```

The edge pixels that do not overlap the polygon are drawn if they pass the depth test. The depth buffer is updated with the depth values of pixels that pass.

3. Step 3 implements the second condition of pass 2. If an edge pixel has a stencil value of 1, the pixel is drawn and the depth buffer is updated with the depth value of the pixel. Again, the stencil is left untouched.

```
glDepthFunc(GL_ALWAYS);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// Draw polygon edge as a line
```

These calls draw the edge pixels that overlap the depth-buffered polygon.

4. Step 4 implements pass 3 of the edging-plane algorithm, which clears the edging-plane values underneath the polygon to 0.

```
glDepthFunc(GL_NEVER);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_KEEP, GL_REPLACE,
            GL_KEEP);

// Draw polygon
```

There is another way to implement this step:

```
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);
```

To get the correct results when using this algorithm, take care to specify the vertices of adjacent polygon edges in the same direction. ■

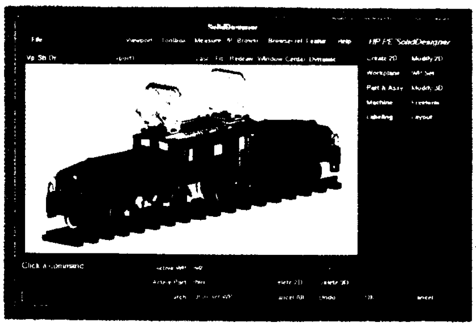
Application of polygon edging in Hewlett-Packard's PE/SolidDesigner

Claus Brod and Michael Metzger, *HP Mechanical Design Division*

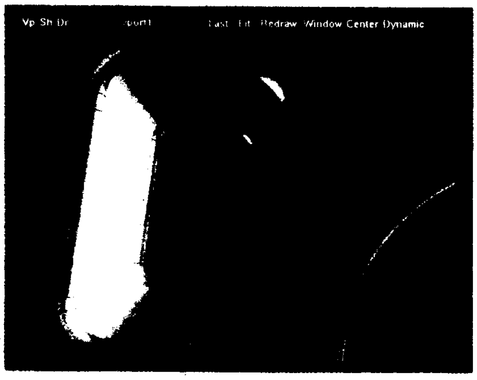
HP PE/SolidDesigner is a 3D volume modeler that makes heavy use of features in the supporting graphics libraries. The designer always sees the shaded model, as shown in Figure A. In addition, the default display mode shows the model edges (that is, edges that can be picked) "on top" of the shaded model. Figure B shows an example of this "edge mode," which is by far the most useful for modeling.

PE/SolidDesigner lets the user control hidden-line removal, shading, and edge display (no edges, model edges only, and all polygon edges). To support this flexiblity, it depends on a graphics library and hardware that address the issues of displaying polygon edges and filled polygon interiors at the same time. The edging-plane algorithm helps to achieve a clear and pleasant visualization. Moreover, the algorithm improves performance by giving users immediate visual feedback through edge highlighting. For example, Figure C illustrates where the edging-plane algorithm really shines. Model edges selected for modification are drawn in a special highlighting color on top of the model in the display. There is no need to redraw the whole model, which—for large models—would be a major nuisance.

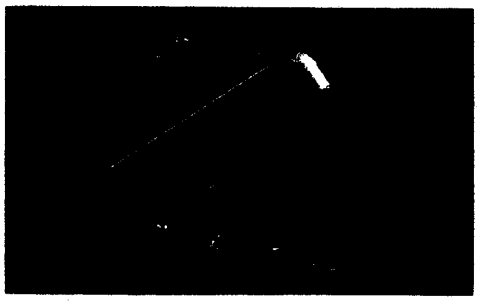
In daily modeling, the edging-plane algorithm avoids stitching, and it does so very efficiently. However, there are still some situations where the visual result is not completely satisfactory (typically, when neighboring parts are displayed independent of each other). Figure D illustrates the problem that occurs when one assembly is edged and the other is not, as discussed in the main text under "Edging multiple polygons." ■



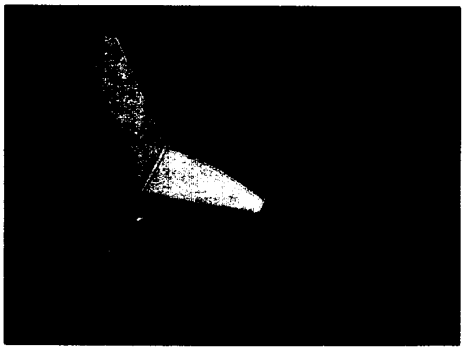
A User interface for HP PE/SolidDesigner, displaying in the shaded-only mode.



B Default display mode ("edge mode").



C Some parts are selected for modification, and their edges are highlighted.



D Stitched edges with independent polygons.