IDA PAPER P-2354

A TOOLBOX FOR SHADING

Michael R. Kappel

February 1990

Prepared for
Computer and Software Engineering Division (CSED)

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 90-035103

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this publication was conducted under IDA's Central Research Program. Its publication does not imply endorsement by the Department of Defense or any other Government Agency, nor should the contents be construed as reflecting the official position of any Government Agency.

This Paper has been reviewed by IDA to assure that it meets the high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1990	Final

4. TITLE AND SUBTITLE
A Toolbox for Shading

5. FUNDING NUMBERS
IDA CRP 9000-502

6. AUTHOR(S)
Michael R. Kappel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2354

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT ***(Maximum 200 words)***

IDA Paper P-2354, A Toolbox for Shading, documents the research performed under an internal research and development program, the Central Research Program (CRP) 9000-502. The research probed the shortcomings of existing shading algorithms. The investigation yielded seven new techniques to render more realistic images at a modest increase in computational costs. Specifically, quadratic interpolation of intensities and adaptive subdivision of surfaces are proposed to generate smoother intensity surfaces and thereby reduce Mach banding. Techniques are introduced to locate highlights and adaptively subdivide surfaces to capture the intensity peaks associated with specular reflection. Ambient cut-off and adaptive subdivision are applied to accurately render diffuse boundaries. Tradeoffs are examined so that a subset of this toolbox of shading techniques can be chosen to most effectively satisfy the processing and realism requirements of a given image-generating application and environment.

14. SUBJECT TERMS
Algorithms; Mach banding; Gouraud shading; Phong shading; computer graphics; images; shading techniques; polygon mesh shading; Cendes-Wong formulae.

15. NUMBER OF PAGES
206

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unciassified	Unclassified	UL

IDA PAPER P-2354

A TOOLBOX FOR SHADING

Michael R. Kappel

February 1990

INSTITUTE FOR DEFENSE ANALYSES

IDA Central Research Project
CRP 9000-502

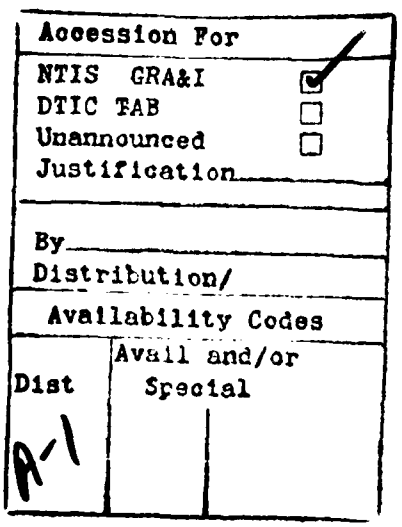
ABSTRACT

Visual realism in computer-generated images is an important goal for many computer applications. However, a trade-off exists between the degree of realism achieved and the time required to generate an image. Thus *efficient* algorithms for generating *realistic* images are being actively investigated.

Shading is one graphical technique for rendering more realistic images of three-dimensional objects. However, the most widely used algorithm, Gouraud shading, suffers from the Mach band effect, a perceptual phenomenon that reduces realism. Gouraud shading also handles specular reflection poorly. The next most popular algorithm, Phong shading, generally reduces Mach banding and captures specular highlights, though at a great computational expense. Bishop and Weimer improved the efficiency of Phong shading, but their algorithm introduces approximation error and is still significantly slower than Gouraud.

This research probes the shortcomings of existing shading algorithms. The investigation yielded seven new techniques to render more realistic images at a modest increase in computational cost. Specifically, quadratic interpolation of intensities and adaptive subdivision of surfaces are proposed to generate smoother intensity surfaces and thereby reduce Mach banding. Techniques are introduced to locate highlights and adaptively subdivide surfaces to capture the intensity peaks associated with specular reflection. Ambient cut-off and adaptive subdivision are applied to accurately render diffuse boundaries.

Trade-offs are examined so that a subset of this toolbox of shading techniques can be chosen to most effectively satisfy the processing and realism requirements of a given image-generating application and environment.



CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Visual Realism

Visual realism in computer-generated images is an important goal in such applications as computer simulation and computer-aided design. The success of a hardware/software package depends upon the realism of the generated images. However, absolute realism is not a requisite for success; a certain threshold exists, above which the viewer has sufficient cues to comprehend an image. Thus visual realism is defined as a subjective quality of an image, which evokes a perceptual response sufficiently similar to that evoked by reality.

The degree of realism to be achieved in a computer-generated image depends broadly on five factors [NEWM79]:

1. the processing required to generate the image
2. the perceptual effects of the image on the observer
3. the application's requirements for realism
4. the capabilities of the display hardware
5. the complexity of the image

Visual realism in a computer-generated image is limited by stringent processing requirements. When objects, light sources, and/or viewpoint move in space and time, a dynamic rendering capability is required. To create the illusion of smooth motion, an image must be regenerated at least ten times per second [FOLE82]. (Do not confuse this rate with the sixty cycle per second refresh rate required by the display processor to preclude flicker.) Thus the entire graphics pipeline must be traversed in one-tenth of a

second. Even advanced display architectures barely satisfy the processing requirements of flicker-free synthesis of dynamic, complex, realistic images in real time.

Several methods have been developed to render more realistic images of three-dimensional objects on a two-dimensional computer screen. Eight of the more popular visualization techniques include [NEWM79] — (1) projections, (2) intensity cues, (3) stereoscopic views, (4) kinetic depth effect, (5) hidden-line elimination, (6) texture mapping, (7) anti-aliasing, and (8) shading. These techniques can be applied individually or in combination. No one technique is best; the choice depends on the trade-off of computational costs, perception, the application's requirement for realism, hardware capability, and image complexity. This research investigates the visualization technique of shading.

1.2 Graphics Pipeline

The process of generating an image on a display device is a sequence of transformations, called the graphics pipeline. The graphics pipeline for raster-scan displays is depicted in Figure 1. The output of the graphics pipeline, intensity (and color) values for every pixel on the screen, is stored in the frame buffer, typically random access memory. The frame buffer is read by the display processor to generate voltage levels to drive the display device.

The first stage of the graphics pipeline traverses a structured display file of application-defined objects. All categories of information — geometric, textual, algorithmic, relational, visual — can be stored in the object database and all can be presented pictorially on the display screen. For our purposes, the structured display file contains an

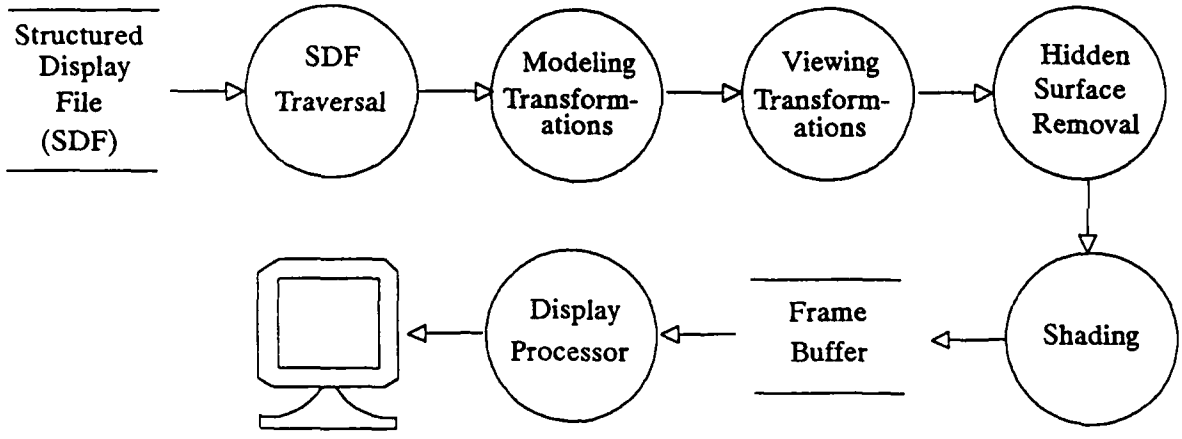


Figure 1. Graphics Pipeline

The graphics pipeline transforms application-defined objects into a raster-scan image. The circles represent data transformations and the parallel lines represent temporary data stores.

ordered list of vertices, completely specifying the polygon mesh approximations to curved three-dimensional objects.

The second stage of the graphics pipeline transforms model coordinates to world coordinates and finally to viewing coordinates. An instance transformation is first applied to build objects from component parts and instantiate them in world space. The software developer chooses a world coordinate system to suit the graphics application. Objects are then defined and manipulated abstractly in terms of this world space. A geometric transformation is then applied to provide dynamic placement of objects in world space.

The third stage of the graphics pipeline converts world coordinates to viewing coordinates by normalizing in three dimensions. The converted output primitives are then clipped

against the canonical view volume. Whole objects or parts of objects may be clipped. Only those points, lines, polygons, text, etc. that are within the boundaries of the current view volume undergo further processing. In this way, extraneous calculations are precluded. The coordinates of the remaining primitives are then converted to 3D normalized device coordinates (NDC) which define a device-independent logical coordinate system such as the unit cube.

The fourth stage of the graphics pipeline removes hidden surfaces. In a three-dimensional scene, objects closer to the viewer may overlap and therefore obscure more distant objects. The obscured surfaces are eliminated from further consideration. Finally, the visible surfaces are projected into two dimensions in 2D NDC space.

The fifth and final stage of the graphics pipeline determines the appropriate shades for every visible surface. Shade is defined as the intensity and color of the light reflected and transmitted towards the observer. A shade is calculated for each picture element, or pixel, on a raster-scan display. Shading is the most computationally expensive stage of the graphics pipeline.

1.3 Shading Models

A shading model is applied to computer-generated objects to determine the shade of each point on the surfaces of the objects. The three simplest models — ambient light, Lambert's diffuse reflection, and Phong's specular reflection — have been implemented widely in graphics systems requiring fast image generation.

Consider the lighting in your immediate environment. There are sources of light and objects that reflect or transmit light. Light may be reflected off of an object towards your

eye or in some other arbitrary direction. In fact, light may be repeatedly reflected until it reaches your eye. The brightness of a point on the surface of an object is thus the sum total of light that is reflected directly from the light sources towards your eye and light that is repeatedly reflected.

Repeated reflections can be modeled accurately via ray tracing or radiosity methods. Individual light rays can be traced from the viewpoint through each pixel back towards the light sources. Alternatively, a set of linear equations can be solved to describe the transfer of light energy between all surfaces in the scene. While affording high realism, both techniques are computationally expensive. Ray tracing and radiosity are currently not suitable for applications requiring fast image generation.

Alternatively, the cumulative intensity of light that has been repeatedly reflected is approximated by a single constant. This low-level background illumination is assumed to be uniform in all directions, and is thereby termed ambient. A measure of a surface's capacity to reflect ambient light is given by a constant called the reflectance coefficient. The amount of ambient light (I_a) which is reflected from any point on the surface of an object is expressed as the product of the intensity of the ambient light (I_b) and the surface's ambient reflectance coefficient (k_a),

$$I_a = I_b \, k_a \tag{1}$$

Light sources are modeled as idealized points which may be at arbitrary distances, in arbitrary orientation, and arbitrary in number. Light that is reflected directly from its source to the viewer is modeled in two ways, diffuse and specular reflections. Diffuse reflection is exhibited by dull surfaces, such as a brick wall, for which incident light is reflected uniformly in all directions. Specular reflection is exhibited by shiny surfaces,

such as glazed pottery, for which incident light is reflected primarily in one direction.

The intensity of a diffuse reflection at a given point on the surface of an object (I_d) is dependent upon the light intensity of the point source (I_p) and its direction ($\vec{L}$), the direction of the surface normal at that point ($\vec{N}$), and the surface's diffuse reflectance coefficient (k_d). This relation is given by Lambert's law (Figure 2):

$$I_d = I_p \, k_d \, \vec{L} \cdot \vec{N} \qquad (2)$$

where I_d is at minimum zero and $\vec{L}$ and $\vec{N}$ are unit vectors.

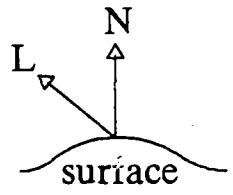


Figure 2. Diffuse Reflection

The intensity of a diffuse reflection at a given point on the surface of an object is dependent upon the light intensity of the point source and its direction ($\vec{L}$), the direction of the surface normal at that point ($\vec{N}$), and the surface's diffuse reflectance coefficient.

A specular highlight is perceived when light is reflected in the direction of the viewer. The intensity of the specular highlight falls off rapidly from its peak intensity. Bui Tuong Phong [BUI75] modeled this fall-off as some power n (depending on the shininess of the surface) of the cosine of the angle between the between the direction of reflection ($\vec{R}$) and the viewpoint ($\vec{V}$) (Figure 3). Phong's model for specular reflection, which is based on empirical data, does a creditable job in capturing highlights. A more accurate, theoretically-based model devised by Torrance and Sparrow is discussed in Section 7.1.1.

In Phong's model, the intensity of a specular reflection (I_s) can be expressed as:

$$I_s = I_p\, k_s\, (\vec{R} \cdot \vec{V})^n \tag{3}$$

where k_s is the surface's specular reflectance coefficient and $\vec{R}$ and $\vec{V}$ are unit vectors.

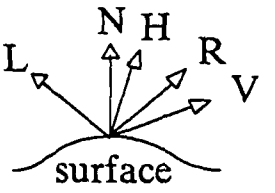


Figure 3. Specular Reflection

The intensity of a specular reflection at a given point on the surface of an object is dependent upon the light intensity of the point source and its direction ($\vec{L}$), the direction of the surface normal at that point ($\vec{N}$), the viewpoint ($\vec{V}$), and the surface's diffuse reflectance coefficient. The direction of reflection ($\vec{R}$) and the direction of maximum highlight ($\vec{H}$) are fully determined.

$\vec{R}$ can be expressed as a function of $\vec{L}$ and $\vec{N}$:

$$\vec{R} = 2\,(\vec{L} \cdot \vec{N})\,\vec{N} - \vec{L} \tag{4}$$

which satisfies the following three conditions:

1. $\vec{R}$ is in the plane formed by $\vec{L}$ and $\vec{N}$.
2. $\vec{R} \cdot \vec{N} = \vec{L} \cdot \vec{N}$.
3. $\vec{R}$ is normalized.

Equation 3 has been alternately expressed as:

$$I_s \cong I_p\, k_s\, (\vec{H} \cdot \vec{N})^n \tag{5}$$

where $\vec{H}$ is a unit vector in the direction of maximum highlight given by:

$$\vec{H} = \frac{\vec{V}+\vec{L}}{|\vec{V}+\vec{L}|}$$

$\vec{H} \cdot \vec{N}$ only approximates $\vec{R} \cdot \vec{V}$; the angle between $\vec{H}$ and $\vec{N}$ is one-half the magnitude of the angle between $\vec{R}$ and $\vec{V}$. As we shall see, Equation 5 has been used in shading algorithms (e.g., surface normal interpolation) for computational convenience, since $\vec{H}$ can be computed efficiently via linear interpolation. However, Equation 3 renders highlights that are smaller than they should be. For those shading algorithms (e.g., intensity interpolation) which calculate intensities exactly at only a few points, the more faithful Equation 3 is used.

The shade of a point on the surface of an object can then be modeled by summing intensities of the ambient, diffuse and specular components given in Equations 1, 2 and 3 (or 5), respectively:

$$I = I_a + I_d + I_s \tag{6}$$

$$= I_b k_a + I_p k_d \vec{L} \cdot \vec{N} + I_p k_s (\vec{R} \cdot \vec{V})^n \tag{7}$$

$$\cong I_b k_a + I_p k_d \vec{L} \cdot \vec{N} + I_p k_s (\vec{H} \cdot \vec{N})^n \tag{8}$$

Equation 7 is directly extensible to multiple light sources:

$$I = I_b k_a + \sum_1^m I_{p_i} k_d \vec{L}_i \cdot \vec{N} + \sum_1^m I_{p_i} k_s (\vec{R}_i \cdot \vec{V})^n \tag{9}$$

where m is the number of light sources.

The color of the light reflected from a surface depends upon the colors of the incident light and of the surface. The color of the ambient and diffuse components is the same color as the surface. The color of the specular component is a function of the colors of the light source and the object's surface. In the simplest model for specular reflection,

Phong assumes that the color of the specular component is the color of the light source. While this assumption is valid for plastics, it is not valid for other materials [COOK81]. For now, we make this simplifying assumption and later, discuss its ramifications (Section 7.1).

Color can be represented by values for its three subtractive primary color components: cyan, magenta and yellow. Subtractive colors are used because reflection is a subtractive process [FOLE82]. The equation for the intensity of the cyan component of reflected light is:

$$I_c = I_{bc}k_{ac} + I_{pc}k_{dc}\vec{L}\cdot\vec{N} + I_{pc}k_{sc}(\vec{R}\cdot\vec{V})^n \qquad (10)$$

The equations for the magenta and yellow components are analogous. The three primary colors combine to give the color of the reflected light.

This treatment of color assumes that a tristimulus model of human vision can be applied to model the interaction of light with objects [FOLE82]. While this assumption produces creditable images, a physically more accurate model was devised by Hall [HALL83]. Hall's model uses wavelength-dependent Fresnel reflectivity terms for better sampling in wavelength space (Section 7.1).

Color complicates our discussion. The psychophysical response to color by humans introduces factors irrelevant to our discussion of algorithmic complexity and image realism. For example, the consideration of our varying sensitivity to different wavelengths opens whole new areas of research without greatly impacting our thesis. Thus the following analyses will be restricted to monochrome images.

Since our primary interest is speed, only the three simplest models and their algorithmic

implementations will be discussed in detail. Three more sophisticated shading models are considered in Section 7.1. As we shall see, intensities are calculated exactly at only a few points when interpolating intensities. Then even the most accurate shading models can be used without incurring great computational expense.

1.4 Polygon Meshes

Three-dimensional surfaces are represented in a variety of ways. Three of the more common techniques are polygon meshes, parametric functions, and quadratic functions. Polygon meshes are the most widely used because they have several distinct advantages. Furthermore, parametric and quadratic surfaces can be and often are reduced to polygon meshes during the display process.

Curved surfaces can be approximated by polygonal facets to any desired degree of precision. Furthermore, polygons facilitate computations. Their planar nature implies simple linear calculations, thereby simplifying the determination of surface normals, intersections, and hidden surfaces. Planes maintain their characteristics in perspective space and their specification with vertices facilitates coordinate transformations.

Consider Newell's teapot [CROW87]. Blinn [BLIN87] provided a compact representation of its structure in terms of sweep curves (for the body and lid) and Bezier surface patches (for the handle and spout). For ease of implementation, we modified the representation of the teapot's body and lid to also be Bezier surface patches. The front surfaces of the body, lid, and handle are each modeled as two surfaces patches, the spout just one. To avoid hidden surface elimination and thereby simplify implementation, only the front surfaces of the teapot are generated and displayed.

The parametric representation of a Bezier surface patch is of the form $x = X(u,v)$, $y = Y(u,v)$, and $z = Z(u,v)$ where x, y, and z are world coordinates, and u and v are the parameters. The bicubic equation for x is as follows:

$$
\begin{aligned}
x &= X(u,v) \\
&= (1-u)^3(1-v)^3 B_{x00} + 3u(1-u)^2(1-v)^3 B_{x10} + 3u^2(1-u)(1-v)^3 B_{x20} + \\
&\quad u^3(1-v)^3 B_{x30} + (1-u)^3 v(1-v)^2 B_{x01} + 3u(1-u)^2 v(1-v)^2 B_{x11} + \\
&\quad 3u^2(1-u)v(1-v)^2 B_{x21} + u^3 v(1-v)^2 B_{x31} + (1-u)^3 v^2(1-v) B_{x02} + \\
&\quad 3u(1-u)^2 v^2(1-v) B_{x12} + 3u^2(1-u)v^2(1-v) B_{x22} + u^3 v^2(1-v) B_{x32} + \\
&\quad (1-u)^3 v^3 B_{x03} + 3u(1-u)^2 v^3 B_{x13} + 3u^2(1-u)v^3 B_{x23} + u^3 v^3 B_{x33}
\end{aligned}
\tag{11}
$$

where B_{xij} are the x-coordinates of the sixteen Bezier control points for each surface patch, with analogous equations for y and z.

World coordinates for points on the teapot's surface are obtained by stepping incrementally in both parameters u and v from 0 to 1. Obviously, a smaller step size yields a greater number of points. Adjacent points are then linked to form triangular facets; a straightforward exercise due to the regularity of the generated mesh.

Surface normals are also calculated at the generated points for the purposes of shading. The tangents to the surface in u,v space are (X_u, Y_u, Z_u) and (X_v, Y_v, Z_v), where:

$$
\begin{aligned}
X_u &= \frac{\partial X(u,v)}{\partial u} \\
&= I - 3(1-u)^2(1-v)^3 B_{x00} + 3(-2u(1-u)+(1-u)^2)(1-v)^3 B_{x10} + \\
&\quad 3(-u^2+2u(1-u))(1-v)^3 B_{x20} + 3u^2(1-v)^3 B_{x30} + -3(1-u)^2 v(1-v)^2 B_{x01} + \\
&\quad 3(-2u(1-u)+(1-u)^2)v(1-v)^2 B_{x11} + 3(-u^2+2u(1-u))v(1-v)^2 B_{x21} +
\end{aligned}
$$

$$3u^2v(1-v)^2B_{x31} + -3(1-u)^2v^2(1-v)B_{x02} +$$

$$3(-2u(1-u)+(1-u)^2)v^2(1-v)B_{x12} + 3(-u^2+2u(1-u))v^2(1-v)B_{x22} +$$

$$3u^2v^2(1-v)B_{x32} + -3(1-u)^2v^3B_{x03} + 3(-2u(1-u)+(1-u)^2)v^3B_{x13} +$$

$$3(-u^2+2u(1-u))v^3B_{x23} + 3u^2v^3B_{x33} \tag{12}$$

with analogous equations for the other five partial derivatives.

The normal (N_x, N_y, N_z) to the surface at a given u,v is the cross product of the tangents:

$$N_x = Y_uZ_v - Z_uY_v \tag{13}$$

$$N_y = Z_uX_v - X_uZ_v \tag{14}$$

$$N_z = X_uY_v - Y_uX_v \tag{15}$$

1.5 Timing Study

The computational costs of shading algorithms will be analyzed in terms of unit costs for floating-point arithmetic operations. The number and type of arithmetic operation are counted for each step of a shading algorithm. The total computational costs are then determined by assuming relative costs for each of the arithmetic operations. While this type of analysis may satisfy the theorist in all of us, computer run times provide some empirical data for comparison.

Timing studies will be presented throughout the remainder of this paper with images generated on a Sun Microsystems® 3/110 color graphics workstation and photographed on a Dunn Instruments' Multicolor™ Image Processing System. The Sun workstation has eight megabytes of main memory and thirty megabytes of swap space. The algorithms were coded in the C programming language and the output generated using the Pixrect

graphics subroutine library. The compiler's floating-point code generation option was used to generate in-line code for the Motorola MC68881 floating-point processor.

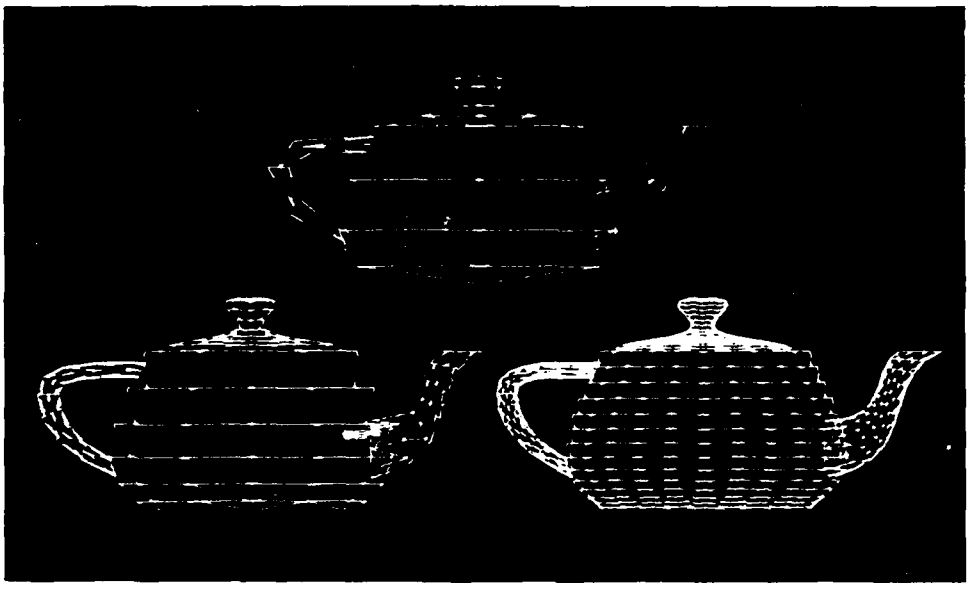
The run times in CPU seconds were obtained using the system-supplied profiler. The profiler tabulates the percentage of time spent executing each subroutine, the number of times it was called, and the number of milliseconds per call. The CPU times obtained by the profiler cannot be assumed to be exact. Though much of the extraneous time required by the operating system has been factored out of the run times for the applications code, according to Heisenberg's Uncertainty Principle, the very act of measuring introduces some error.

Computer-generated images of Newell's teapot are used as the test case. The starting points for comparing computational costs are data files which contain world coordinates and surface normals at the vertices of the polygon mesh, as well as adjacency information for constructing triangular facets.

Figure 4 shows three wire frame images of the teapot. The wire frame represents the original set of triangles which approximates the curved surface of the teapot. As we move from Figure 4a to 4c, smaller steps in parameter space yield a more accurate approximation. Figure 5 shows images of teapots shaded by linearly interpolating intensities, corresponding to the polygon meshes of Figure 4.

Images such as these will appear throughout the rest of the paper along with detailed timing analyses. To give an idea of what we're striving for, Figure 6 shows the best image that can be achieved with the chosen shading model. A surface normal was obtained at each pixel of the teapot and its intensity was calculated by Equation 7.

(a)



(b) (c)

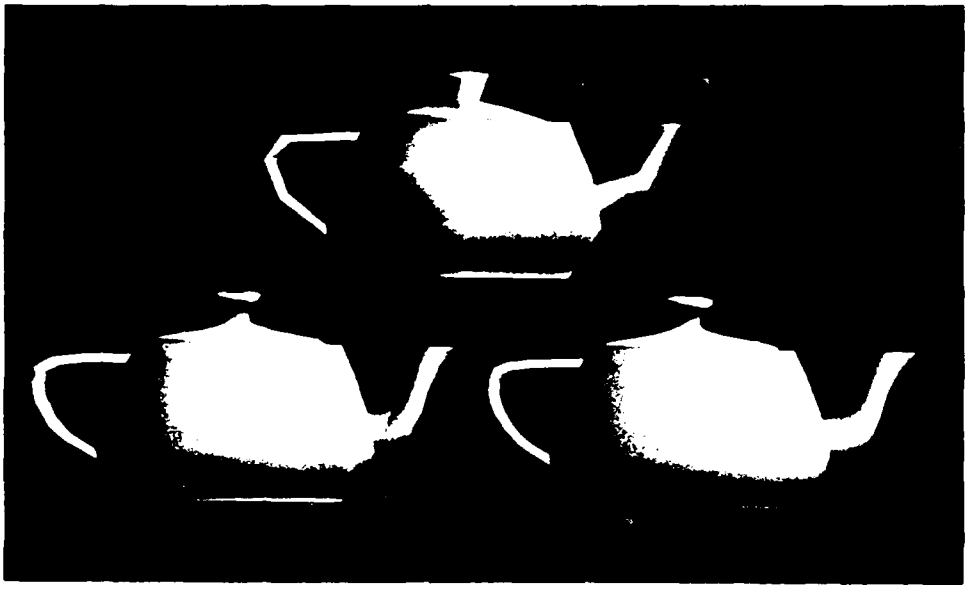
Figure 4. Sample Wire-Frame Images

Three wire frame images of the teapot. As we move from *a* to *c*, smaller steps in parameter space yield a more accurate approximation.

1.6 Objectives

The objectives of this research are to investigate deficiencies in existing shading algorithms and to propose enhancements for their resolution. In particular, the most widely used algorithm, Gouraud shading, suffers from the Mach band effect, a perceptual phenomenon that reduces visual realism. Mach bands are perceived because the linear interpolation scheme generates C^1 discontinuities in the intensity surface. Can we generate a smoother intensity surface to reduce Mach banding? And do so at a modest increase in computational costs, possibly by quadratically interpolating intensities?

(a)



(b) (c)

Figure 5. Sample Shaded Images

Three images of teapots shaded by linearly interpolating intensities, corresponding to the polygon meshes of Figure 4.

Gouraud shading also handles specular reflection poorly. A specular highlight may be entirely missed if it should fall at a point other than at a vertex because its intensity decreases exponentially. Can we locate the peak of the highlight on the intensity surface so that a new vertex can be added to the polygon mesh? Will one new point suffice or will it be necessary to adaptively subdivide the surface to capture the rapid fall-off?

The next most popular algorithm, Phong shading, generally reduces Mach banding and captures specular highlights, though at a great computational expense. Bishop and Weimer improved the efficiency of Phong shading by approximating the dot products of the intensity equation (Equation 8) by a Taylor series. However, their algorithm

Figure 6. Best Shaded Teapot

The best image that can be achieved with the chosen shading model. A surface normal was obtained and the intensity calculated at each pixel of the teapot.

introduces approximation error in regions of high surface curvature. Can we adaptively subdivide the surface to reduce the approximation error? Can we apply a similar technique in Gouraud shading to reduce the curvature of the intensity surface in order to minimize Mach banding?

This research attempts to answer these questions and thereby build a toolbox of cost-effective techniques for shading three-dimensional objects on a raster-scan display.

1.7 Organization

Section 2 details the algorithmic implementations of shading models for polygon meshes and compares computational costs. Section 3 addresses the issues of curvature and continuity of the intensity surface with respect to the perceptual effect of Mach bands. Section 4 proposes a shading algorithm that fits a C^1 intensity surface with quadratic polynomials. Section 5 introduces several enhancements to shading algorithms that capture specular highlights and diffuse boundaries. Section 6 describes several trade-off studies that address issues to be considered when building a shading algorithm suitable for

a given application. Section 7 presents the conclusions of this research and suggestions for future work.

A list of bibliographic references and appendices follow. Appendix A gives the Cendes-Wong formulas to perform quadratic C^1 interpolation over triangular surface patches. Appendix B comprises the C source code implementation of each shading algorithm.

2. POLYGON MESH SHADING

This section details the algorithmic implementations of shading models for polygon meshes. For all implementations, the polygonal facets are assumed to be triangular. Triangular facets can be obtained from the original set of polygons. For example, the polygonalization of a curved surface represented by parametric bicubics yields quadrilaterals. Triangles are simply obtained by adding a diagonal. However, the triangulation of arbitrary polygons is not a trivial matter, but good algorithms have been devised [LAWS77,BARN77, NIEL83].

2.1 Constant

Constant shading is the simplest implementation of the shading model applied to polygon meshes. Constant shading uses a constant average light direction ($\vec{L}$) and a constant average direction of maximum highlight ($\vec{H}$) over each polygonal facet. If the surface normals of the underlying surface are approximated by the surface normal of the facet, the dot products ($\vec{L} \cdot \vec{N}$) and ($\vec{R} \cdot \vec{V}$) are constant for all points. Finally, assuming that the reflectance coefficients are constant over the facet, all terms on the right-hand side of Equation 7 are constant.

Constant shading is the cheapest technique for shading polygon meshes. Since all terms on the right-hand side of Equation 7 are constant for all points on a facet, the intensity of reflected light need only be calculated once per facet. All pixels within the facet are assigned the same shade. Unfortunately, objects rendered with constant shading appear unrealistic. Since the shade of each facet is generally different than that of its neighbor, a C^0 discontinuity (i.e., a discontinuity in magnitude) in intensity arises across facet

boundaries. The underlying polygonal approximation to the curved surface thereby becomes apparent.

2.2 Linear Interpolation of Intensities

Gouraud [GOUR71] proposed a refinement to constant shading to eliminate the C^0 discontinuity in shades across facet boundaries. Shades are linearly interpolated across the face of the facet to achieve C^0 continuity. Assuming that a surface normal is given at each vertex, the algorithm is as follows (Figure 7):

1. Compute the shades at vertices A, B, and C using Equation 7.
2. Calculate constants for interpolating shades.
3. Calculate the shade of each pixel through linear interpolation.

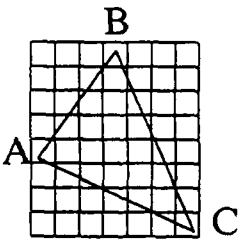


Figure 7. Triangular Facet

A triangular facet with vertices A, B, and C, on a rectangular grid of pixels.

The linear interpolation of shades in two dimensions can be expressed as:

$$I(x,y) = Dx + Ey + F \tag{16}$$

where interpolation constants D, E, and F are given by:

$$D = \frac{(I_C - I_A)(y_B - y_A) - (I_B - I_A)(y_C - y_A)}{(x_C - x_A)(y_B - y_A) - (x_B - x_A)(y_C - y_A)} \tag{17}$$

$$E = \frac{I_B - I_A - D(x_B - x_A)}{y_B - y_A} \tag{18}$$

$$F = I_A - Dx_A - By_A \tag{19}$$

The primary advantage of linearly interpolating intensities is efficiency. The shade of each pixel can be determined in just one addition using forward differencing. Initialize:

$$I(x_0, y_0) = Dx_0 + Ey_0 + F \tag{20}$$

where (x_0, y_0) is the centroid of the triangular facet. The centroid is chosen as the starting point to minimize truncation error. Iterate:

$$I(x_{i+1}, y) = I(x_i, y) + D \tag{21}$$

$$I(x, y_{i+1}) = I(x, y_i) + E \tag{22}$$

Although this algorithm is efficient and achieves C^0 continuity, its linear interpolation scheme gives rise to three drawbacks: anomalous highlights, inconsistent motion, and Mach bands. A specular highlight may be entirely missed if it should fall at a point other than at a vertex because its intensity decreases so rapidly. The shade of a point may change contrary to expectations in a motion sequence if it suddenly depends upon the shades of a different set of vertices. The third drawback, Mach bands, will be discussed in detail in Section 3.

2.3 Linear Interpolation of Surface Normals

Phong [BUI75] proposed a more sophisticated shading technique to better handle specular reflection. Given the most general case in which the viewpoint and point light sources are at finite distances, the surface normal ($\vec{N}$), the light direction ($\vec{L}$), and the

direction of maximum highlight ($\vec{H}$) are linearly interpolated across the face of each facet. Referring to Figure 7, the algorithm is as follows:

1. Calculate interpolation constants for $\vec{N}$, $\vec{L}$, and $\vec{H}$.
2. Calculate vectors $\vec{N}$, $\vec{L}$, and $\vec{H}$ at each pixel through linear interpolation.
3. Normalize vectors $\vec{N}$, $\vec{L}$, and $\vec{H}$.
4. Determine the shade of the pixel using Equation 8.

The linear interpolation of vectors $\vec{N}$, $\vec{L}$, and $\vec{H}$ is similar to intensity interpolation. Here, each coordinate must be interpolated separately. Hence the total computational cost for Step 2 is nine additions for each pixel using forward differencing.

In Step 3, the calculated vectors are normalized to unit vectors by dividing by their magnitudes. The magnitude of a vector is given by the square root of its own dot product. Thus the computational cost of Step 3 for each pixel is six additions, nine multiplications, three divisions, and three square roots.

The computational cost of Step 4 (calculating intensities) is six additions, eight multiplications, and one exponentiation assuming that the Ik factors are pre-calculated. Therefore, the total computational cost of this algorithm at each pixel is twenty-one additions, seventeen multiplications, three divisions, three square roots, and one exponentiation.

The appearances of objects shaded by linearly interpolating surface normals are generally more realistic than those shaded by intensity interpolation. Although specular reflection is handled successfully, highlights may still be misplaced. While only C^0 continuity is attained at facet boundaries, the Mach band effect is generally reduced.

However, as before, motion may be inconsistent due to the linear interpolation scheme. While surface normal interpolation is an improvement over intensity interpolation in terms of realism, its computational costs are high.

Three enhancements to Phong's original shading method have been proposed to reduce its computational load — (1) selective application, (2) forward differencing, and (3) Taylor series approximation. They are discussed in the following three subsections.

2.3.1 Selective Application

Phong and Crow [BUI75a] suggested that linear interpolation of surface normals be applied only to the polygonal facets that display specular highlights while all other facets are shaded by linearly interpolating intensities. Highlights are located by first transforming the reflected light vectors in a perspective space relative to the viewpoint. If a transformed vector passes near the viewpoint, its transformed z-component approaches one. The reflected light vector at each vertex is examined to determine whether or not the highlight is near.

Since typically few facets display specular highlights, this combined approach would cost little more than using intensity interpolation exclusively. Some additional processing is required to ensure that C^0 intensity discontinuities do not arise at the boundary between a facet using surface normal interpolation and an adjacent facet using intensity interpolation.

2.3.2 Forward Differencing

Duff [DUFF79] improved the efficiency of the Phong algorithm by forward differencing

the dot products of the intensity function. If the interpolation constants are $\vec{A}$, $\vec{B}$, and $\vec{C}$ for $\vec{L}$ and $\vec{D}$, $\vec{E}$, and $\vec{F}$ for $\vec{N}$, their dot product equals:

$$\vec{L} \cdot \vec{N} = (\vec{A}x + \vec{B}y + \vec{C}) \cdot (\vec{D}x + \vec{E}y + \vec{F}) \tag{23}$$

Initialize:

$$M_x(x_0, y_0) = (\vec{A}\cdot\vec{D})\,x_0^2 + (\vec{B}\cdot\vec{D} + \vec{A}\cdot\vec{E})\,x_0y_0 + (\vec{B}\cdot\vec{E})\,y_0^2 + (\vec{A}\cdot\vec{F} + \vec{C}\cdot\vec{D})\,x_0 + (\vec{B}\cdot\vec{F} + \vec{C}\cdot\vec{E})\,y_0 + \vec{F}\cdot\vec{C} \tag{24}$$

$$\Delta M_x(x_0, y_0) = (\vec{A}\cdot\vec{D})(2x_0 + 1) + (\vec{B}\cdot\vec{D} + \vec{A}\cdot\vec{E})\,y_0 + (\vec{A}\cdot\vec{F} + \vec{C}\cdot\vec{D}) \tag{25}$$

$$\Delta^2 M_x = 2(\vec{A}\cdot\vec{D}) \tag{26}$$

and iterate:

$$M_x(x_{i+1}, y) = M_x(x_i, y) + \Delta M_x(x_i, y) \tag{27}$$

$$\Delta M_x(x_{i+1}, y) = \Delta M_x(x_i, y) + \Delta^2 M_x \tag{28}$$

A similar set of forward differencing equations exists for the y dimension. Each dot product can be calculated in two additions for each pixel. Forward differencing can also be used for the dot products found in the normalization step. Thus ten additions are required at each pixel for the five dot products - $\vec{L} \cdot \vec{N}$, $\vec{H} \cdot \vec{N}$, $\vec{N} \cdot \vec{N}$, $\vec{L} \cdot \vec{L}$, and $\vec{H} \cdot \vec{H}$.

The total iterative cost of Duff's algorithm is twelve additions, one multiplication, two divisions, three square roots, and one exponentiation at each pixel.

2.3.3 Taylor Series Approximation

Bishop and Weimer [BISH86] approximate the diffuse and specular components of the intensity function by Taylor series. If the light vector and the surface normal are linearly interpolated and normalized as before, the diffuse component of Equation 8 can be

rewritten as:

$$I_d(x, y) = I_p k_d \frac{(\vec{A}x + \vec{B}y + \vec{C}) \cdot (\vec{D}x + \vec{E}y + \vec{F})}{\sqrt{(\vec{A}x + \vec{B}y + \vec{C})^2 (\vec{D}x + \vec{E}y + \vec{F})^2}} \tag{29}$$

This function of two variables can be approximated by a Taylor series. Translating the triangular facet so that the centroid is at the origin and expanding in a Taylor series to the second degree:

$$I_d(x, y) = T_5 x^2 + T_4 xy + T_3 y^2 + T_2 x + T_1 y + T_0 \tag{30}$$

with complicated expressions for the T constants.

This diffuse component can be evaluated at each pixel with a computational cost of two additions using forward differencing. The ambient component can be combined with the diffuse component during initialization. The calculation of the specular component at each pixel requires two additions (similar to the diffuse component), then exponentiation, and finally multiplication by $I_p k_s$. The total iterative cost at each pixel is thus five additions, one multiplication, and one exponentiation.

2.4 Comparison of Computational Costs

The total computational cost of these shading algorithms can be partitioned into start-up and iterative costs. Start-up costs are incurred once for each vertex or once for each triangular facet. Iterative costs are incurred for each pixel. Tables 1, 2, and 3 summarize the start-up and iterative costs for each of the shading algorithms under investigation.

The following assumptions were made in deriving computational costs:

1. The image is monochrome.
2. The polygonal facets are triangular.

Table 1. Start-up Costs Per Vertex

Shading Algorithm	**Computational Cost**				
	+	x	/	√	**
Constant	6	7			1
Linear Interpolation of Intensities	6	7			1
Linear Interpolation of Surface Normals					
Original	0				
with Forward Differencing	0				
with Taylor Series Approximation	0				

Table 2. Start-up Costs Per Facet

Shading Algorithm	**Computational Cost**				
	+	x	/	√	**
Constant	0				
Linear Interpolation of Intensities	14	9	2		
Linear Interpolation of Surface Normals					
Original	126	81	18		
with Forward Differencing	303	308	18		
with Taylor Series Approximation	153	332	8	2	

3. Unit surface normals ($\vec{N}$) and unit light vectors ($\vec{L}$ and $\vec{R}$ or $\vec{H}$) are given at all vertices.
4. Only one point light source is present (multiple light sources are considered in Section 4.6).
5. The viewpoint and point light source are at finite distances (the most general case).
6. Partial products have been pre-calculated to minimize cost.

Table 3. Iterative Costs Per Pixel

Shading Algorithm	Computational Cost				
	+	x	/	√	**
Constant	0				
Linear Interpolation of Intensities	1				
Linear Interpolation of Surface Normals					
Original	21	17	3	3	1
with Forward Differencing	12	1	2	3	1
with Taylor Series Approximation	5	1			1

Computational costs can be expressed as a function of the number of vertices in the interior of the triangulation (d), the number of vertices on the boundary (e), and the number of pixels (p) on the computer monitor. The number of facets in the triangulation is then $2d+e-2$ [GRAN86]. Let t_1 through t_5 be the CPU times for the five floating-point operations (addition, multiplication, division, square root, and exponentiation), respectively. The cost of shading by linearly interpolating intensities is:

$$Cost_{LII}(d,e,p) = (6t_1+7t_2+t_5)(d+e) + (14t_1+9t_2+2t_3)(2d+e-2) + t_1p \qquad (31)$$

The cost of shading by linearly interpolating surface normal with Taylor series approximation is:

$$Cost_{LISN}(d,e,p) = (153t_1+332t_2+8t_3+2t_4)(2d+e-2) + (5t_1+t_2+t_5)p \qquad (32)$$

Consider the execution times for double-precision floating-point arithmetic of Motorola's MC68020 CPU and MC68881 coprocessor shown in Table 4 [SUN87]. The computational cost of shading by linearly interpolating intensities is:

$$Cost_{LII}(d,e,p) = 81.84(d+e) + 114.72(2d+e-2) + 3.68p \qquad (33)$$

Similarly, the computational cost of shading by linearly interpolating surface normal with

Taylor series approximation is:

$$Cost_{LISN}(d,e,p) = 2418.72(2d + e - 2) + 46.48p \quad (34)$$

Table 4. Representative CPU Times for Floating-Point Arithmetic [SUN87]

Operation	Microseconds
Addition	3.68
Multiply	5.28
Division	7.84
Square root	20.0
Exponentiation	22.8

With a few assumptions, we can plot computational costs as a function of the number of facets. Assume that the image is to be generated on a 1000-by-1000 pixel monitor, i.e., $p = 1{,}000{,}000$. Also assume that the number of interior points is much larger than the number of boundary points, i.e., $d \gg e$. Since the number of facets equals $2d + e - 2$, the number of vertices is approximately half the number of facets. Using the CPU times for Motorola's chips, Figure 8 plots the theoretical computational costs as a function of the number of facets. The number of facets range from ten thousand to one thousand, so the number of pixels per facet range from one hundred to one thousand. Note that the iterative costs dominate the start-up costs as the number of facets decrease, or alternatively, as the average facet size increases.

In addition, empirical costs are superimposed as individual data points in Figure 8. CPU times were obtained at four levels of resolution for each shading algorithm while rendering a million-pixel image. A 5-to-1 scale factor is used to bring the theoretical and empirical data values into alignment. The existence of the scale factor is due to

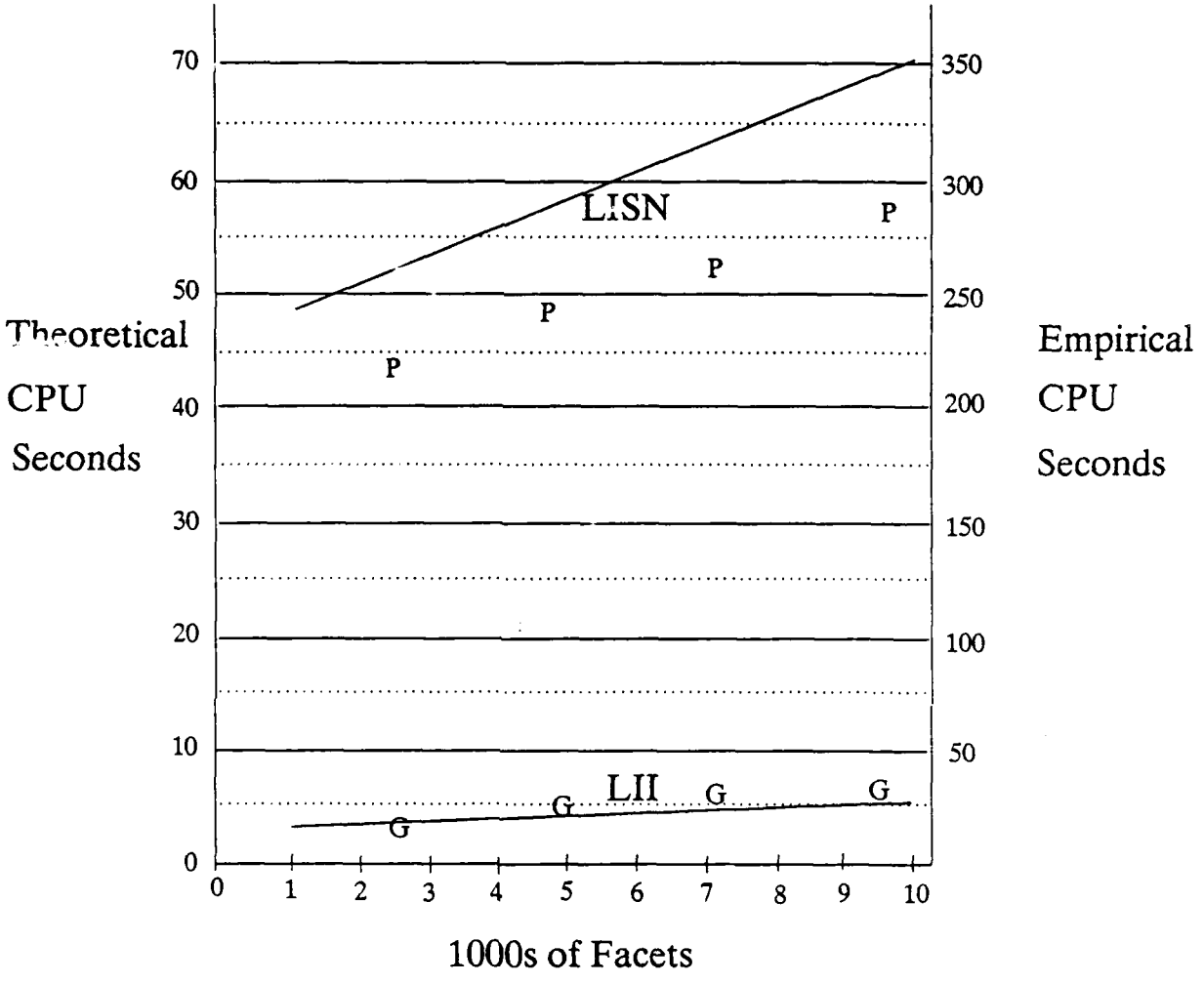


Figure 8. Computational Costs as a Function of the Number of Facets

Theoretical computational costs plotted as a function of the number of facets assuming a 1000-by-1000 pixel monitor and that the number of interior points is much larger than the number of boundary points. The theoretical costs correspond to Equations 33 and 81, respectively. In addition, empirical costs for a million-pixel image are shown as individual data points (G = LII = Linear Interpolation of Intensities, P = LISN = Linear Interpolation of Surface Normals).

operations other than arithmetic that are executed in the real implementation. Such operations include memory referencing, input/output, subroutine calls, etc.

3. INTENSITY SURFACE CONTINUITY

The realism of the images generated by the aforementioned shading algorithms suffer to different degrees from a perceptual effect called Mach bands. The magnitude of the Mach band effect is related to continuity of the intensity surface. This relationship will be expressed in a mathematical model. The theoretical basis of the model will be considered in terms of the psychophysical mechanism underlying the Mach band effect.

An important point to remember throughout this discussion is that Mach bands do arise naturally. They are present when viewing real objects, and it is therefore not our goal in image synthesis to completely eliminate Mach bands. However, those Mach bands which are an artifact of the shading process and are not present in the real object reduce visual realism.

3.1 Lateral Inhibition

The Mach band effect has been attributed to the neural mechanism of lateral inhibition. Receptors in the human eye receive light stimuli from the environment. However, the sensation perceived by a receptor is a combination of overlapping excitatory and inhibitory influences of neighboring receptors. Thus the response of a receptor depends not only on the magnitude of the incoming stimulus, but also on the influences of its neighbors. Lateral inhibition is a physiological process whereby stimulation of a receptor excites local neighbors while inhibiting more distant neighbors. The excitatory and inhibitory influences increase for stimuli of larger magnitude. The local influence of a point stimulus is shown in Figure 9, taken from Ratliff's monograph on Mach bands [RATL65].

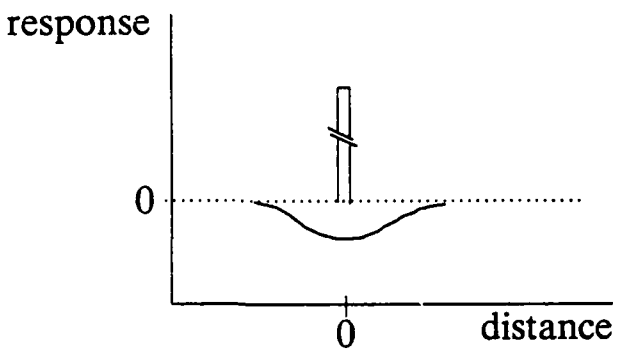


Figure 9. Lateral Inhibition [RATL65]

A point stimulus excites local neighbors while inhibiting more distant neighbors.

Consider the intensity curve in Figure 10. Receptors in the eye are receiving light stimuli in relation to the actual intensities (solid lines). However, those receptors near the discontinuity are being inhibited differently than their neighbors. Receptors in the darker region near the discontinuity are being inhibited to a greater extent than their dark neighbors due to the proximity of the highly inhibitory bright region. Alternatively, receptors in the brighter region near the discontinuity are being inhibited to a lesser extent than their bright neighbors due to the proximity of the darker region. The cumulative effect of the excitatory and inhibitory responses is the intensity of light that is perceived (dotted lines).

3.2 Mathematical Model of the Mach Band Effect

A mathematical model of lateral inhibition provides a theoretical basis for the discussion of Mach bands generated by the various shading algorithms. However, a high-fidelity model is not required because in the subsequent mathematical treatment of Mach bands,

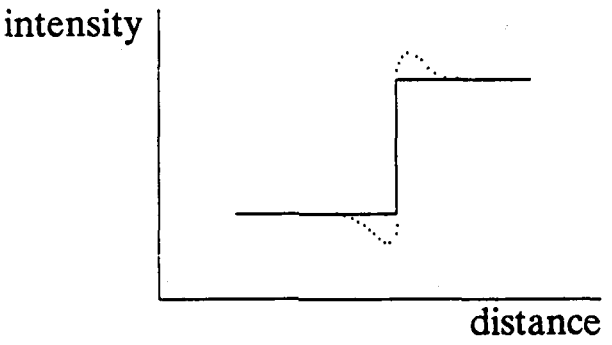


Figure 10. Perceived vs. Actual Intensity [RATL65]

The solid lines represent actual intensities of light stimuli received by receptors in the eye. The dotted lines represent perceived intensities due to the cumulative effect of excitatory and inhibitory responses.

their magnitudes will be discussed in relative terms only. A mathematical model of the gross behavior of neural networks will suffice.

Several mathematical models of lateral inhibition in neural networks have been proposed to explain the Mach band effect. Ratliff's monograph [RATL65] on Mach bands summarizes six such models by Huggins/Licklider, Bekesy, Mach, Fry, Hartline/Ratliff and Taylor. The six models differ in their degree of fidelity. For example, the Hartline/Ratliff and Taylor models consider recurrent inhibition, i.e., the strength of an inhibitory influence is determined by the ultimate response of a receptor rather than the strength of its stimulus. However, as far as the gross behavior of neural networks is concerned, Ratliff argued that all six models are essentially equivalent.

The six models are equivalent in the sense that "the modification of the response appears to be related to the negative of the second derivative" of the intensity curve [RATL65]. Various intensity curves whose second derivative is undefined (i.e., C^2 discontinuous) will be considered in the following discussion. In such cases, the difference between the right

and left first derivatives at the discontinuity will be used in lieu of the second derivative. The magnitude of the Mach band induced at the discontinuity will be proportional to the negative of this difference.

Figure 10 depicted a graph of the perceived Mach bands for an actual intensity curve that is C^0 discontinuous. Analogous graphs can be drawn for curves of higher continuity. Figures 11, 12, and 13 show the Mach bands induced by intensity curves that are C^1 discontinuous, C^2 discontinuous, and C^2 continuous, respectively. The relative magnitudes of the perceived Mach bands (dotted lines) are consistent with the mathematical model of lateral inhibition.

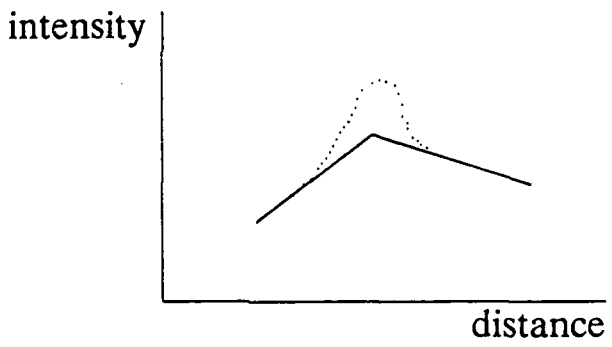


Figure 11. Mach Band and C^1 Discontinuity [RATL65]

The solid lines represent actual intensities of light stimuli, while the dotted line represents perceived intensities. The C^1 discontinuity in the intensity curve gives rise to a Mach band.

3.3 Constant Shading

The constant shading method typically generates different shades for adjacent polygonal facets since the orientation of adjacent facets forming a curved surface are generally different. Thus a C^0 discontinuity in shades arises at the boundary between the two

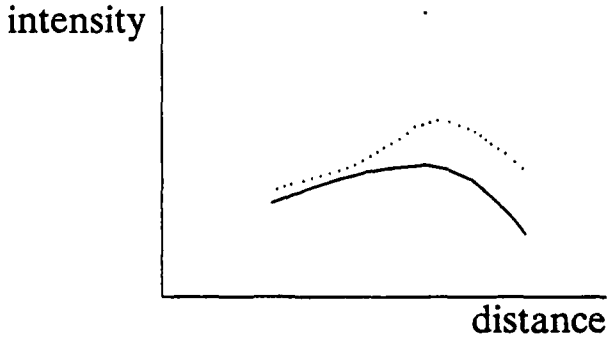


Figure 12. Mach Band and C^2 Discontinuity [RATL65]

The solid lines represent actual intensities of light stimuli, while the dotted line represents perceived intensities. The C^2 discontinuity in the intensity curve gives rise to a smoother, less noticeable, Mach band.

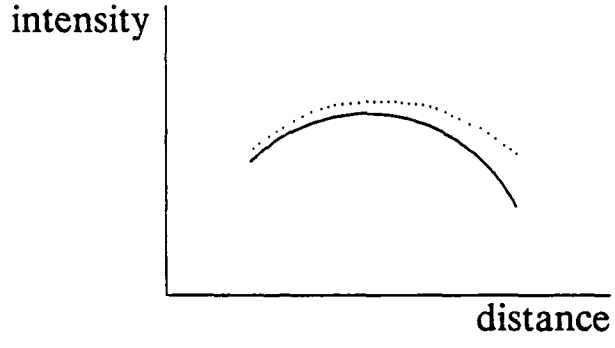


Figure 13. Mach Band and C^2 Continuity (Hypothetical)

The solid lines represent actual intensities of light stimuli, while the dotted line represents perceived intensities. Assuming the validity of the mathematical model for Mach bands, an intensity curve that is C^2 continuous would not give rise to a Mach band.

facets. The C^0 discontinuity induces Mach bands, which further enhance the perception of the boundary between the two polygonal facets. A sample shading intensity curve was presented in Figure 10. The facets that approximate the curved surface are apparent and thus objects rendered with this shading technique appear unrealistic.

3.4 Linear Interpolation of Intensities

Linear interpolation of intensities generates a shading intensity curve which is C^0 continuous, i.e., the magnitudes of the shades across polygon boundaries are equal. However, since shades are linearly interpolated, a C^1 discontinuity in the intensity curve arises at the boundary between two adjacent polygonal facets (see Figure 11). The greater the curvature of the surface being approximated, the greater will be the discontinuity, increasing the magnitudes of the Mach bands. The Mach bands perceived at the facet boundaries accentuate the presence of the polygon artifacts.

3.5 Linear Interpolation of Surface Normals

Linear interpolation of surface normals also induces Mach bands, though generally less severe than linear interpolation of intensities. Similarly, the intensity curve generated is C^0 continuous with C^1 discontinuities at the boundary between two polygonal facets.

Duff [DUFF79] analyzed the severity of Mach bands induced by linear interpolation of intensities and surface normals. Duff used the difference between the right and left derivatives at the facet boundaries as a measure of the Mach band effect. Duff demonstrated that it is possible for linear interpolation of surface normals to induce more severe Mach bands than intensity interpolation in some typical cases with spheres and cylinders.

Furthermore, linear interpolation of surface normals with Taylor series approximation does not even guarantee C^0 continuity for intensities at facet boundaries. The error term associated with their second-degree Taylor series approximation is:

$$R_2(x, y, 0) = \int_0^x \int_0^y \frac{1}{2!} \left((x-s)^2 \frac{\partial^3}{\partial x^3} + (y-t)^2 \frac{\partial^3}{\partial y^3} \right) f(s,t)\, ds\, dt \qquad (35)$$

This error term is large when x or y is large (for a big facet) and/or when the derivative is large (for a surface with high curvature). Then the intensity approximated for the pixel located at the facet boundary differs significantly from its true intensity. A C^0 discontinuity is perceived when the approximated intensity of the pixel on the other side of the boundary differs from its true intensity in the opposite direction.

Bishop and Weimer claim that the error becomes large enough to adversely affect the image when the polygon curvature exceeds 60° [BISH86]. Polygon curvature is defined to be the planar divergence of two adjacent polygonal facets. Figure 14a shows an image of a teapot spout and 14b shows the corresponding underlying polygon mesh. Even when the polygon curvature is as small as 45°, the approximation error is large enough to be noticeable.

(a) (b)

Figure 14. Spouts Shaded with Taylor Series Approximation

(a) An image of a teapot spout and (b) its underlying polygon mesh. Even when polygon curvature is as small as 45°, the approximation error is large enough to be noticeable.

3.6 Adaptive Subdivision - Object Surface Curvature

A polygon mesh can be obtained by repeatedly subdividing the original curved surface. This yields new sets of control vertices defining a succession of smaller and more accurate subsurfaces. Each subsurface is recursively subdivided until some threshold is reached. The threshold criteria can be based on the smallness or flatness of the subsurface, or the closeness of the control vertices to the original curved surface [BART87]. This technique is called adaptive subdivision.

Suppose that we are now given a polygon mesh to render with shading. The accuracy of the approximation, while sufficient in a geometric sense, may not satisfy shading requirements. In particular, consider the shading algorithm that linearly interpolates surface normals via Taylor series approximation. As discussed above, the shading generated by this algorithm breaks down when polygon curvature exceeds 45°. However, this requirement may not have been addressed when the polygon mesh was generated. For example, curvature cannot be uniformly controlled by the step size when the mesh is generated by stepping incrementally in parameter space.

This approximation error is reduced by adaptively subdividing the mesh while rendering. Consider the curvature of the object's surface between two adjacent vertices. Figure 15 depicts two possible cases in two dimensions. The tangent plane through P_1 is defined by the surface normal $\vec{N}$. Calculate the perpendicular distance from P_2 to the tangent plane. If the difference exceeds a given threshold, the edge is subdivided at its midpoint. The magnitude of the threshold depends upon the accuracy requirements of the particular application. The smaller the threshold difference, the more accurate the polygonal approximation will be.

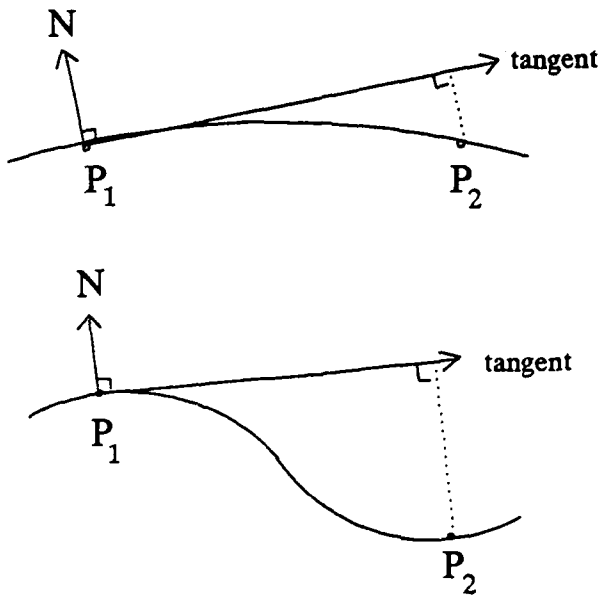


Figure 15. Adaptive Subdivision for Surface Curvature

In two dimensions, project the tangent at one vertex to the other vertex. If the perpendicular distance from P_2 to the tangent exceeds a given threshold, the edge is subdivided at its midpoint.

The adaptive subdivision procedure is performed recursively. On the first pass, each existing edge is considered for subdivision by projecting the tangent planes at the two endpoints. The distance from a point to a plane can be determined easily by rotating the coordinate system so that the plane aligns with two coordinate axes. The distance is then given directly by the third coordinate of the rotated point. Let θ be the angle between $\frac{\partial z}{\partial x}$ and the x-axis and let ϕ be the angle between $\frac{\partial z}{\partial y}$ and the y-axis. If (x_1, y_1, z_1) are the coordinates of an endpoint, the transformation matrix in homogeneous coordinates is given by:

$$\begin{bmatrix} \cos\phi & 0 & -\sin\theta & 0 \\ \sin\theta\sin\phi & \cos\theta & \sin\theta\cos\phi & 0 \\ \cos\theta\sin\phi & -\sin\theta & \cos\theta\cos\phi & 0 \\ -x_1\cos\phi - y_1\sin\theta\sin\phi - z_1\cos\theta\sin\phi & -y_1\cos\theta + z_1\sin\theta & x_1\sin\phi - y_1\sin\theta\cos\phi - z_1\cos\theta\cos\phi & 1 \end{bmatrix}$$

The perpendicular distance of the second endpoint with coordinates (x_2, y_2, z_2) from the tangent plane is then given directly by the rotated z coordinate:

$$z_{perp} = (x_1 - x_2)\sin\phi + (y_1 - y_2)\sin\theta\cos\phi + (z_1 - z_2)\cos\theta\cos\phi \qquad (36)$$

If subdivision is warranted, a new point is computed in screen and/or world coordinates and edges are added to reform triangles. At the next iteration, any new edges are considered for subdivision. A distance threshold can be included to preclude subdividing edges that are too short. This procedure terminates when no new points are added.

For each new point, its screen coordinates and surface normal must be obtained. The screen coordinates are simply the midpoint between two endpoints. The surface normal is obtained in one of three ways depending upon the underlying representation of the original surface and the degree of accuracy required. If only an approximation is called for, the surface normal can easily be obtained by linearly interpolating the normals at the two endpoints. For greater accuracy, the corresponding world coordinates are first determined by applying the inverse of the viewing transformation. If the original surface is represented by a parametric bicubic equation, techniques devised by Catmull [CATM75], Blinn [BLIN86], or Clark [CLAR86] can be used to obtain the corresponding parametric values. These methods including those compiled in [LANE80] are costly due to iterative convergence. Surface normals can then be calculated explicitly.

Alternatively, if the surface of the original object is represented explicitly in world coordinates as such $f(u,v,w) = 0$, then the surface normal is given by:

$$\vec{N} = (f_u, f_v, f_w) \tag{37}$$

After normalizing:

$$\vec{N} = \left(\frac{f_u}{\sqrt{f_u^2 + f_v^2 + f_w^2}}, \frac{f_v}{\sqrt{f_u^2 + f_v^2 + f_w^2}}, \frac{f_w}{\sqrt{f_u^2 + f_v^2 + f_w^2}}\right) \tag{38}$$

Figure 16a shows an image of a teapot shaded by linearly interpolating surface normals via Taylor series approximations. Notice the shading anomalies near the vertices of triangular facets, particularly on the spout. Figure 16b shows the corresponding underlying polygon mesh. Since the spout curves 180° from one side to the other, each facet takes on about 45° of curvature. This curvature is large enough to introduce significant approximation error in the Taylor series, and thereby C^0 discontinuities in the intensity surface.

(a) (b)

Figure 16. Teapot without Adaptive Subdivision for Object Surface Curvature

(a) An image of a teapot, and (b) its underlying polygon mesh, shaded by linearly interpolating surface normals via Taylor series approximations. Shading anomalies are apparent near the vertices of triangular facets, particularly on the spout. Curvature of about 45° is large enough to introduce approximation error, and thereby C^0 discontinuities in the intensity surface.

Figure 17a shows an image of the same teapot (identical lighting and surface characteristics) except that the object's surface has been adaptively subdivided. Note

that the aforementioned shading anomalies are not apparent. Figure 17b shows the corresponding polygon mesh which has been adaptively subdivided in regions of high curvature.

(a) (b)

Figure 17. Teapot with Adaptive Subdivision for Object Surface Curvature

(a) An image of the teapot, and (b) its underlying polygon mesh, whose surface has been adaptively subdivided in regions of high curvature. Shading anomalies are not apparent.

3.7 Adaptive Subdivision - Intensity Surface Curvature

As discussed in the previous subsection, an object's surface can be adaptively subdivided to reduce the approximation error introduced by linearly interpolating surface normals via Taylor series. A similar procedure can be applied to the *intensity surface* when its curvature is sufficient to induce Mach bands when linearly interpolating intensities.

The curvature of the intensity surface at a particular vertex is estimated by the tangent plane (tangent to the *intensity surface*) at that point. Calculate the perpendicular distance from an adjacent vertex to the tangent plane in intensity space. If the difference exceeds a given threshold, the edge is subdivided at its midpoint. The magnitude of the threshold depends upon the accuracy requirements of the particular application. The

smaller the threshold difference, the smoother the generated intensity surface will be.

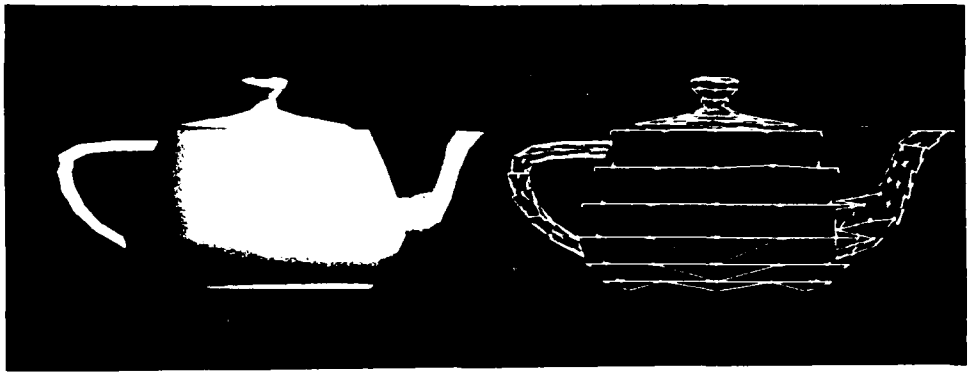
The calculation of the perpendicular distance from an intensity point to an intensity tangent plane is similar to that performed for object surface curvature. The distance is determined by rotating the coordinate system so that the tangent plane aligns with the x and y coordinate axes. The perpendicular distance is then given directly by the rotated intensity point:

$$I_{perp} = (x_1 - x_2)\sin\phi + (y_1 - y_2)\sin\theta\cos\phi + (I_1 - I_2)\cos\theta\cos\phi \qquad (39)$$

where the intensity point has coordinates (x_2, y_2, I_2) and the tangent plane at point (x_1, y_1, I_1) is given by θ, the angle between $\frac{\partial I}{\partial x}$ and the x-axis and ϕ, the angle between $\frac{\partial I}{\partial y}$ and the y-axis.

As before, the surface normal at the midpoint is obtained in one of three ways depending upon the underlying representation of the original surface and the degree of accuracy required. If only an approximation is called for, the surface normal can easily be obtained by linearly interpolating the normals at the two endpoints. For greater accuracy, the surface normal can be calculated explicitly if the original surface is represented by a parametric bicubic equation or given explicitly in world coordinates. For any new points, new edges are then added to reform triangles.

Figure 18a shows an image of a teapot shaded by linearly interpolating intensities without adaptively subdividing its intensity surface. The specular highlighting is turned off to enhance diffuse reflections. Two Mach bands are perceived near the right side and lower portion of the teapot body where the curvature of the intensity surface is great. Figure 18b shows the corresponding underlying polygon mesh.



(a) (b)

Figure 18. Teapot without Adaptive Subdivision for Intensity Surface Curvature

(a) An image of a teapot, and (b) its underlying polygon mesh, shaded by linearly interpolating intensities without adaptively subdividing its intensity surface. Two Mach bands are perceived near the right side and lower portion of the teapot body where the curvature of the intensity surface is great.

Figure 19a shows an image of the same teapot (identical lighting and surface characteristics) except that the intensity surface has been adaptively subdivided. The polygon curvature of the intensity surface is constrained to be no greater than 10°. The Mach bands that were present in Figure 18a are noticeably reduced. Figure 19b shows the corresponding polygon mesh. Note the addition of vertices in regions of high curvature of the intensity surface.

(a) (b)

Figure 19. Teapot with Adaptive Subdivision for Intensity Surface Curvature

(a) An image of the teapot, and (b) its underlying polygon mesh, whose intensity surface has been adaptively subdivided. The polygon curvature of the intensity surface is constrained to be no greater than 10°. The Mach bands that were present in Figure 18a are noticeably reduced.

4. INTENSITY SURFACE FITTING

Mach's mathematical model of lateral inhibition stated that the magnitude of a Mach band is proportional to the negative of the second derivative of the intensity curve (Section 3.2). Thus a smoother intensity curve would reduce the Mach band effect. The polygon artifacts would be less apparent and the rendered objects would thereby be more realistic.

In the previous section, adaptive subdivision was introduced to generate smoother intensity surfaces. When linearly interpolating surface normals via Taylor series, an object's surface is adaptively subdivided to reduce the C^0 discontinuities of approximation error. When linearly interpolating intensities, the intensity surface is adaptively subdivided to reduce C^1 discontinuities in the intensity surface.

We now take one logical step forward and generate an intensity surface that is C^1 continuous. When linearly interpolating intensities, first-order piecewise polynomials are obtained to fit a C^0 intensity surface. As a natural extension, can we find higher-order piecewise polynomials to fit a C^1 intensity surface?

This question has been answered affirmatively for geometric surfaces. Fortunately, an intensity surface is like any other geometric surface whose third dimension can be expressed as a function of the other two. So, whereas the third dimension is depth or height for geometric surfaces, our third dimension is intensity. We, therefore, explore geometric surface-fitting techniques and choose one to generate smoother intensity surfaces.

4.1 Surface Fitting

Many surface-fitting algorithms exist; a survey of such methods is found in [BOEH87]. The number of such algorithms that can be applied to shading is limited by the nature of the problem. Foremost, the problem is one of fitting an intensity surface that is smoother than those generated by existing shading algorithms, i.e., the surface must be at least C^1 continuous. Second, the surface-fitting technique must be computationally efficient. This trade-off in smoothness and speed suggests using a technique that achieves only C^1 continuity.

The third consideration in the choice of surface-fitting algorithms is interpolation vs. approximation. An interpolated surface passes through the data points, while an approximated surface need only pass near. Since intensity values are calculated exactly (for the given shading model) at the data points, we will require the surface to pass through those points. Thus an interpolating technique is sought.

Fourth, the problem is one of determining shades of pixels on a two-dimensional screen. Since intensity is a function of two variables (i.e., the two screen dimensions), a bivariate interpolant is sought.

The fifth consideration is points vs. patches. Shephard [SHEP65] devised a class of methods that determine point values based on the values of and distances from all given points. Such schemes are computationally expensive. Alternatively, while it is generally not possible to interpolate all data points with one low-order polynomial, a surface can be constructed from patches which are each represented by a different low-order polynomial. Although many low-order polynomials need to be determined, each can be evaluated rapidly. Thus a piecewise polynomial approach is sought.

The sixth restriction is in the shape of the surface patches. The data points, i.e the vertices of the polygonal facets which approximate the original geometric surface, fall at arbitrary screen locations. However, techniques for polygonal patches with four or more sides require a regular array of data points. For example, Sibson and Thompson [SIBS81] proposed a quadratic C^1 interpolant requiring a rectangular array of data points. The only known methods for interpolation over scattered data are based on triangulations. As stated in Section 2, triangular facets can be easily obtained from the original polygon mesh.

In summary to this point, a bivariate C^1 piecewise polynomial interpolant over irregular triangular patches is sought.

The final consideration in the choice of surface-fitting technique is the degree of the polynomial interpolant. Section 4.2 addresses the cubic interpolation techniques of (1) the nine parameter cubic, (2) global approaches, (3) the Clough-Tocher scheme, and (4) transfinite methods. Section 4.3 addresses quadratic interpolation with subdivision to increase the number of degrees of freedom. In particular, the six-subtriangle Powell-Sabin quadratic interpolant is discussed in detail.

To achieve C^1 continuity, the polynomial must interpolate the intensity value as well as the two tangent vectors at each of the three vertices of a triangular facet. Thus the polynomial interpolant must afford at least nine degrees of freedom. A bicubic polynomial affords ten degrees of freedom, and is thereby the lowest-order polynomial that can interpolate the given data. For this reason, bicubics have received the greatest attention.

Due to its second-order terms, a biquadratic is the lowest-order polynomial that can achieve C^1 continuity. However, a biquadratic polynomial affords only six degrees of freedom, insufficient to interpolate the given data. Fortunately, the number of degrees of freedom can be increased by subdividing the facet. Although cubic interpolants have received more attention, quadratics are less costly to evaluate, an important consideration when fitting an intensity surface.

4.2 Cubic Interpolation

The local surface-fitting methods described below use barycentric coordinates which provide a unique representation of points in the x,y plane. Let P be an arbitrary point in the x,y plane and let A, B and C be vertices of a triangle (Figure 20). Then,

$$x_P = x_A r + x_B s + x_C t \tag{40}$$

and

$$y_P = y_A r + y_B s + y_C t \tag{41}$$

such that

$$r + s + t = 1. \tag{42}$$

Furthermore, r, s, and t are all in the range 0 to 1 for a point within or on the boundary of the triangle.

4.2.1 Nine Parameter Cubic

The most straightforward cubic interpolant is the nine paramcter cubic. Although only C^0 continuity is guaranteed between surface patches, we study the nine parameter cubic because it is the basis of other C^1 methods.

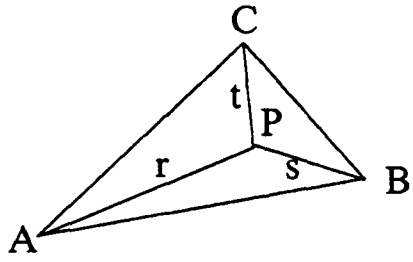


Figure 20. Barycentric Coordinates

Barycentric coordinates r, s, and t provide a unique representation of points in the x,y plane with respect to the vertices of a triangle ABC.

The nine parameter cubic uses Bernstein polynomials for Bezier surfaces in local barycentric coordinates. Bernstein polynomials of degree 3 are defined by:

$$B^3_{i,j,k}(r,s,t) = \frac{3!}{i!j!k!} r^i s^j t^k \tag{43}$$

where $i+j+k=3$ for $i,\ j,\ k \geq 0$.

A Bezier surface over a triangular patch can be expressed in terms of Bernstein polynomials using barycentric coordinates. The height of the surface (I) at point (x,y) is a nine parameter cubic defined as follows:

$$\begin{aligned} I(x,y) = I(r,s,t) &= \sum_{\Delta} b_{i,j,k} B^3_{i,j,k}(r,s,t) \\ &= b_{3,0,0}r^3 + b_{0,3,0}s^3 + b_{0,0,3}t^3 + 3b_{2,1,0}r^2 s + 3b_{2,0,1}r^2 t + 3b_{1,2,0}s^2 r + \\ &\quad 3b_{0,2,1}s^2 t + 3b_{1,0,2}t^2 r + 3b_{0,1,2}t^2 s + 6b_{1,1,1}rst \end{aligned} \tag{44}$$

where the b's are the Bezier points of the control polygon.

For each triangular patch, nine Bezier control points are completely specified by the given

data (Figure 21). $b_{3,0,0}$, $b_{0,3,0}$, and $b_{0,0,3}$ are specified by the intensity values at the vertices. $b_{2,1,0}$, $b_{2,0,1}$, $b_{1,2,0}$, $b_{0,2,1}$, $b_{1,0,2}$, and $b_{0,1,2}$ are specified by the tangents at the vertices. The center Bezier control point $b_{1,1,1}$ is undetermined, but a standard value is used to capture quadratic precision:

$$b_{1,1,1} = \frac{3}{2}c - \frac{1}{2}d \tag{45}$$

where c is the centroid of the tangent Bezier points and d is the centroid of the corner Bezier points.

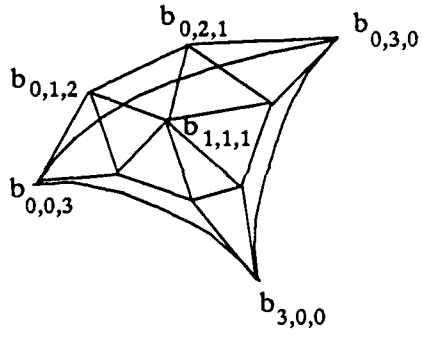


Figure 21. Cubic Bezier Surface

Nine Bezier control points are completely specified by the given data for each triangular patch.

4.2.2 Global Approaches

To achieve C^1 continuity, the center Bezier control points must be chosen such that the quadrilateral *abcd* formed by two adjacent subtriangles is planar (Figure 22). This planar relation can be expressed by a linear equation for each adjacent pair. Let the three points of subtriangle *abc* describe a plane with coefficients e, f, g, and h. Then point d with the homogeneous coordinates (x,y,z,w) is in the plane if and only if:

$$ex + fy + gz + hw = 0 \tag{46}$$

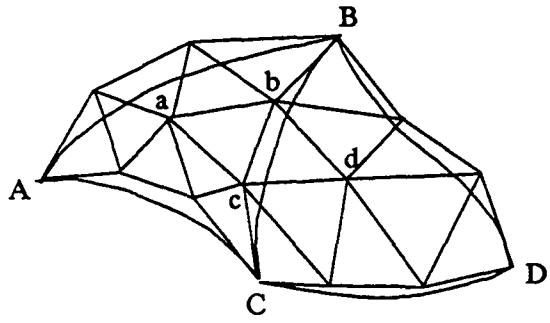


Figure 22. C^1 Condition for Triangular Patches

To achieve C^1 continuity, the center Bezier control points *a* and *d* must be chosen such that the quadrilateral *abcd* formed by two adjacent subtriangles is planar.

This equality can be obtained by either global or local techniques. Global techniques collect the linear relations for all pairs of adjacent triangles and solve the large system of equations. Although the matrices are sparse, the solutions of such large linear systems are computationally expensive. Schmidt [SCHM82] solves this system directly while Grandine [GRAN86] makes small iterative corrections.

4.2.3 Clough-Tocher

Local techniques to modify the inner Bezier control point are based on the Clough-Tocher scheme [STRA73]. Variations have been devised by Lawson [LAWS77] and Farin [FARI83]. Although local methods have been shown to be less accurate [NIEL83], the trade-off in efficiency makes them preferable in this application.

The Clough-Tocher scheme achieves C^1 continuity by adding cross-boundary derivatives at the midpoint of each edge. These three additional constraints brings the total to

twelve, too many for one bicubic polynomial. To add degrees of freedom, each original triangle is subdivided at its centroid into three subtriangles.

In barycentric coordinates local to subtriangle *abc* of Figure 22, point *d* can be expressed as:

$$d = ar + bs + ct \tag{47}$$

The small quadrilateral *becf* of Figure 23 is similar (in the geometric sense) to the larger quadrilateral *abcd* of Figure 22. For quadrilateral *becf* to be planar, the equality:

$$f = er + bs + ct \tag{48}$$

must also hold for the same values of r, s, and t. Using the nine parameter cubic to obtain initial values e_0 and f_0, small corrections are added to satisfy Equation 48:

$$f_0 + \epsilon_2 = (e_0 + \epsilon_1)r + bs + ct \tag{49}$$

Farin [FARI83] uses a least squares method to obtain:

$$\epsilon_1 = \frac{-gr}{(1 + r^2)} \qquad \epsilon_2 = \frac{g}{(1 + r^2)} \tag{50}$$

where

$$g = e_0 r + bs + ct - f_0. \tag{51}$$

This procedure achieves C^1 continuity across the edge of the two original triangles. To achieve continuity across the edges of the subtriangles of the original triangle, the h's are chosen so that quadrilateral *gehe'* is planar (Figure 24). The value of the center Bezier control point $b_{1,1,1}$ at point a is then given by the centroid of the h's.

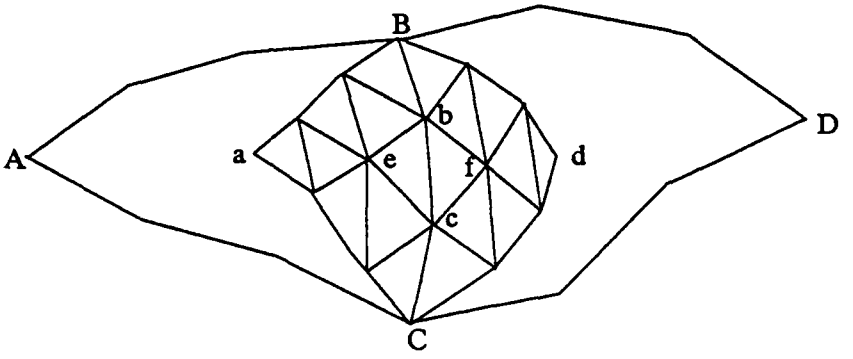


Figure 23. Clough-Tocher Interpolation

The small quadrilateral *becf* is geometrically similar to the larger quadrilateral *abcd* of Figure 22.

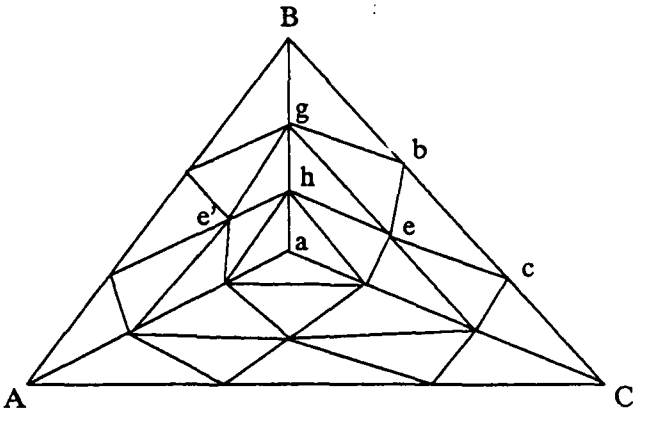


Figure 24. Clough-Tocher Correction

To achieve continuity across the edges of the subtriangles of the original triangle, the *h*'s are chosen so that quadrilateral *gehe'* is planar.

4.2.4 Transfinite Methods

Another class of surface-fitting algorithms known as transfinite methods blend curves defined at patch boundaries. Coons [COON67] derived a bicubic blending function for four boundary curves to achieve C^1 continuity across the boundaries of a quadrilateral

patch. Referring to Figure 25, Coons defined two operators *P1* and *P2* by:

$$P_1 f = f(0,v)\,H_0^3(u) + f(1,v)\,H_3^3(u) \tag{52}$$

$$P_2 f = f(u,0)\,H_0^3(v) + f(u,1)\,H_3^3(v) \tag{53}$$

where the H's are cubic Hermite polynomials. These two surfaces are superimposed by forming the Boolean sum:

$$Pf = (P_1 \oplus P_2) f = (P_1 + P_2 - P_1 P_2) f \tag{54}$$

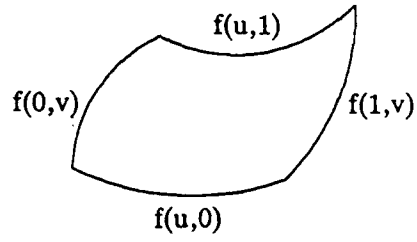


Figure 25. Coons Patch

Coons derived a bicubic blending function for four boundary curves to achieve C^1 continuity across the boundaries of a quadrilateral patch.

Gordon [GORD69] generalized the Coons patch by interpolating a family of curves defined over a patch. Using univariate Lagrange polynomials $L_i^n(u)$ and $J_k^m(v)$, the Boolean sum $(P_1 f \oplus P_2 f)$ interpolates the family of curves $f(u_i, v)$ and $f(u, v_k)$ where

$$P_1 f = \sum_{i=0}^{n} f(u_i, v)\, L_i^n(u) \tag{55}$$

$$P_2 f = \sum_{k=0}^{m} f(u, v_k)\, J_k^m(v) \tag{56}$$

Barnhill, Birkhoff, and Gordon [BARN73] applied the Coons-Gordon technique to triangular patches. Variations on the transfinite theme were introduced by Nielson

[NIEL80] with minimum norm networks and Little [LITT83] with convex combinations.

4.3 Quadratic Interpolation

All of the above surface-fitting algorithms use cubic polynomial interpolants. Although cubics have received more attention, quadratics are less costly to evaluate. Quadratics are the lowest-order polynomials which can achieve C^1 continuity using piecewise interpolants. Although one biquadratic polynomial does not afford enough degrees of freedom (six coefficients) for an entire triangular patch (nine constraints), the number of degrees of freedom can be increased by subdividing the original triangles.

4.3.1 Powell-Sabin

Powell and Sabin [POWE77] argued that it is theoretically possible to interpolate a C^1 surface using quadratics. Suppose a triangle is subdivided into four subtriangles as in Figure 26. The equation of the boundary separating subtriangles PQR and ARQ is given by:

$$lx + my + n = 0 \tag{57}$$

Let $q_1(x, y)$ and $q_2(x, y)$ be the quadratic functions for subtriangles PQR and ARQ, respectively. Then these two subtriangles are C^1 continuous if and only if:

$$q_2(x, y) = q_1(x, y) + k(lx + my + n)^2 \tag{58}$$

holds for some k [POWE77].

The quadratic $q_1(x, y)$ affords six degrees of freedom. Equation 58 introduces three more degrees of freedom, one for each crossing from subtriangle PQR to each of the other three subtriangles. Thus the subdivision into four subtriangles provides a total of nine

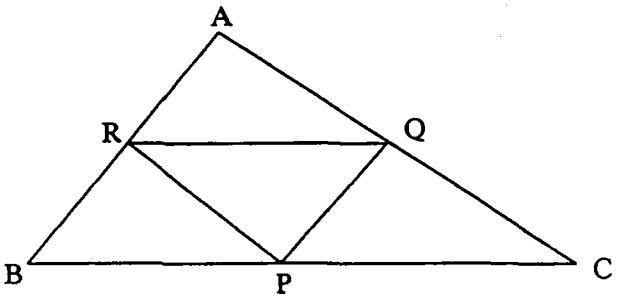


Figure 26. Four-Subtriangle Subdivision

It is theoretically possible to interpolate a C^1 surface using quadratics over a triangle that is subdivided into four subtriangles.

degrees of freedom, enough to satisfy the nine patch constraints. The coefficients of the four quadratics are determined by solving a 9-by-9 system of linear equations. Although this subdivision achieves C^1 continuity over each patch, it does not provide C^1 continuity between adjacent patches.

Powell and Sabin then analyzed a six-subtriangle subdivision (Figure 27). They showed that at least nine degrees of freedom are afforded and the six quadratic interpolants are uniquely determined. This subdivision achieves C^1 continuity over each patch as well as between adjacent triangular patches.

Let $q_1(x, y)$ be the quadratic interpolant for subtriangle AOQ. This quadratic affords six degrees of freedom. Equation 58 introduces six more degrees of freedom, one for each crossing to an adjacent subtriangle, for a total of twelve. In addition to the nine original constraints, three more are introduced by the fact that when the center point is encircled, $q_1(x, y)$ must be obtained again. The coefficients of the six quadratics can be obtained by solving a 12-by-12 system of linear equations (Section 4.3.2). A less expensive way of

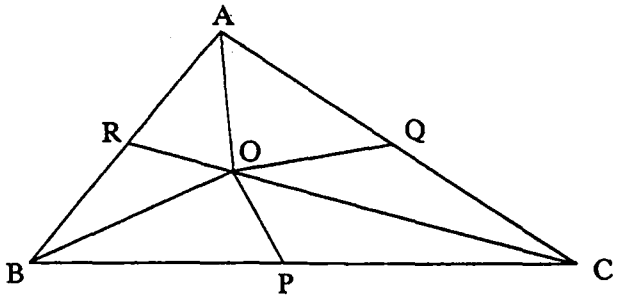


Figure 27. Six-Subtriangle Subdivision

A six-subtriangle subdivision achieves C^1 continuity over each patch as well as between adjacent triangular patches.

obtaining the quadratic interpolants using barycentric coordinates was established by Cendes and Wong (Section 4.3.3).

For the six-subtriangle case, Powell and Sabin considered the placement of the interior point. If point O is the centroid of triangle ABC, difficulties arise when triangles are obtuse. The line joining the centroids of two adjacent triangles does not intersect the common edge. They suggest that the incenter, i.e., the center of an inscribed circle, always provides a successful partitioning.

Powell and Sabin then turned their attention to twelve subtriangles. This subdivision affords twelve degrees of freedom and is always acceptable but is not uniquely determined.

4.3.2 Simultaneous Equations

Powell and Sabin showed theoretically that the coefficients of the six quadratics that fit a C^1 surface over a triangular facet with nine constraints can be obtained by solving a

12-by-12 system of linear equations. Though the solution to these equations is more computationally expensive than the Bernstein-Bezier representation in barycentric coordinates developed by Cendes and Wong (Section 4.3.3), we now develop this system of equations for a potential parallel hardware implementation (Section 6.5).

Let $q_1(x, y)$ be the quadratic interpolant for subtriangle AOQ (Figure 27). q_1 introduces six coefficients — a, b, c, d, e and f — and satisfies three constraints, as follows:

$$I_A = ax_A^2 + bx_Ay_A + cy_A^2 + dx_A + ey_A + f \tag{59}$$

$$\frac{\partial I_A}{\partial x} = 2ax_A + by_A + d \tag{60}$$

$$\frac{\partial I_A}{\partial y} = bx_A + 2cy_A + e \tag{61}$$

where I_A is the intensity and $\frac{\partial I_A}{\partial x}$ and $\frac{\partial I_A}{\partial y}$ are the partial derivatives at vertex A.

Let $q_2(x, y)$ be the quadratic interpolant for subtriangle AOR and let $l_1x + m_1y + n_1$ be the equation of AO, the boundary between subtriangles AOQ and AOR. q_2 satisfies the same three constraints as q_1 does if:

$$q_2(x, y) = q_1(x, y) + k_1(l_1x + m_1y + n_1)^2 \tag{62}$$

q_2 introduces one more degree of freedom, k_1.

Let $q_3(x, y)$ be the quadratic interpolant for subtriangle ROB and let $l_2x + m_2y + n_2$ be the equation of RO. The quadratic surface given by q_3 is C^1 continuous with q_2 if:

$$\begin{aligned} q_3(x, y) &= q_2(x, y) + k_2(l_2x + m_2y + n_2)^2 \\ &= q_1(x, y) + k_1(l_1x + m_1y + n_1)^2 + k_2(l_2x + m_2y + n_2)^2 \end{aligned} \tag{63}$$

q_3 introduces one more degree of freedom, k_2, and satisfies three additional constraints:

$$I_B = ax_B^2 + bx_By_B + cy_B^2 + dx_B + ey_B + f + k_1(l_1x + m_1y + n_1)^2 + k_2(l_2x + m_2y + n_2)^2 \quad (64)$$

$$\frac{\partial I_B}{\partial x} = 2ax_B + by_B + d + k_1(2l_1x_B + 2l_1m_1y_B + 2l_1n_1) + k_2(2l_2x_B + 2l_2m_2y_3 + 2l_2n_2) \quad (65)$$

$$\frac{\partial I_B}{\partial y} = bx_B + 2cy_B + e + k_1(2m_1y_B + 2l_1m_1x_B + 2m_1n_1) + k_2(2m_2y_B + 2l_2m_2x_B + 2m_2n_2) \quad (66)$$

where I_B is the intensity and $\frac{\partial I_B}{\partial x}$ and $\frac{\partial I_B}{\partial y}$ are the partial derivatives at vertex B.

By traversing lines BO and PO, the three equations that satisfy the three constraints at vertex C are similarly developed while introducing two more degrees of freedom, k_3 and k_4. Finally, lines CO and QO are traversed to completely encircle center point O, giving three more equations that must again satisfy the three constraints at vertex A and introducing two more degrees of freedom, k_5 and k_6.

The twelve equations are solved simultaneously to obtain solutions for the twelve unknowns — $a, b, c, d, e, f, k_1, k_2, k_3, k_4, k_5, k_6$. The six quadratic surfaces can then be computed directly.

4.3.3 Cendes and Wong

Cendes and Wong [CEND87] established a set of formulas (Appendix A) for C^1 quadratic interpolation based upon the theoretical results of Powell and Sabin. The same quadratic surface resulting from the solution of a 12-by-12 system of linear equations can be obtained with a Bernstein-Bezier representation in barycentric coordinates. Consider

the six-subtriangle subdivision of triangle *ABC* in Figure 28. A C^1 quadratic surface can be represented as:

$$I(x, y) = I(r,s,t) = \sum_{\Delta} b_{i,j,k} B^2_{i,j,k}(r,s,t)$$

$$= b_{2,0,0}r^2 + b_{0,2,0}s^2 + b_{0,0,2}t^2 + 2b_{1,1,0}rs + 2b_{1,0,1}rt + 2b_{0,1,1}st \qquad (67)$$

where the six b's are the control points and r, s, and t are barycentric coordinates local to the subtriangle.

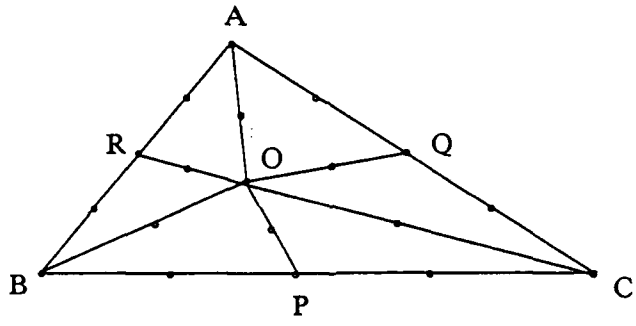


Figure 28. Quadratic Bezier Control Points

Nineteen bezier control points for quadratic interpolation with six subtriangles.

Any of the aforementioned surface-fitting techniques can be applied to shading. Whereas the third dimension is depth or height for geometric surfaces, our third dimension is intensity. Since computational costs are of major concern, we choose quadratic interpolation over cubic, yielding a savings of one addition per pixel using forward differencing. Of the quadratic interpolation techniques, we choose the Cendes-Wong formulas. The Bernstein-Bezier representation in barycentric coordinates developed by Cendes and Wong generates the same quadratic surface as the solution of a 12-by-12

system of linear equations with smaller computational costs.

4.4 Implementation

The formulas for C^1 quadratic interpolation established by Cendes and Wong can be applied to shading. Like the other shading algorithms, it is assumed that the original curved surface is approximated by triangles and that the surface's reflectance coefficients are given. The light sources and viewpoint are specified and unit vectors $\vec{N}$, $\vec{L}$, and $\vec{R}$ are given at each vertex. The intensity of reflected light for each vertex that is visible from the viewpoint can then be immediately obtained from Equation 7. In terms of surface fitting, these intensities are viewed as the scattered data values to be interpolated over x,y screen coordinates.

In addition to intensity values, tangent vectors must be determined at each vertex. The tangent vectors represent the change in intensity for a change in each screen dimension. Nielson and Franke [NIEL83], in their survey of techniques for estimating tangents, state that the tangent vectors have a significant effect on the overall quality of the generated surface. The two basic approaches include (1) a weighted average of tangents of the triangles adjacent to the vertex and (2) a weighted least squares fit to local or global data. One of these tecl.niques can be used to estimate tangents.

In this application, however, we may have additional information so that tangent vectors can be determined exactly. If the surface of the original object is represented explicitly in world coordinates (u,v,w) by:

$$f(u,v,w) = 0 \tag{68}$$

then the surface normal is given by:

$$\vec{N} = (f_u, f_v, f_w) \tag{69}$$

after normalizing:

$$\vec{N} = (\frac{f_u}{\sqrt{f_u^2 + f_v^2 + f_w^2}}, \frac{f_v}{\sqrt{f_u^2 + f_v^2 + f_w^2}}, \frac{f_w}{\sqrt{f_u^2 + f_v^2 + f_w^2}}) \tag{70}$$

Let the screen direction x be represented in world coordinates as:

$$x = u\cos\alpha + v\cos\beta + w\cos\gamma \tag{71}$$

Then the directional derivative of the surface normal in world coordinates gives the partial derivative of the surface normal with respect to screen dimension x:

$$\frac{\partial \vec{N}}{\partial x} = \vec{N}_u \cos\alpha + \vec{N}_v \cos\beta + \vec{N}_w \cos\gamma \tag{72}$$

The partial derivatives for $\vec{L}$ and $\vec{R}$ can be obtained similarly. The partial derivative of the intensity function (Equation 7) with respect to the screen dimension x is:

$$\frac{\partial I}{\partial x} = I_p k_d \frac{\partial(\vec{N}\cdot\vec{L})}{\partial x} + I_p k_s \frac{\partial(\vec{R}\cdot\vec{V})^n}{\partial x} \tag{73}$$

with a similar equation for y. For a given vertex, the partial derivatives of $\vec{N}$, $\vec{L}$, and $\vec{R}$ can be calculated and plugged into Equation 73 to determine the tangent vector exactly.

Once intensities and tangent vectors have be computed, the x,y screen coordinates of incenter O and edge points P, Q, and R are determined (see Figure 28). The nineteen Bezier control points are then calculated with the Cendes-Wong formulas.

The intensity surface given by the quadratic polynomial (Equation 67) for each of the six subtriangles is evaluated using forward differencing. Since screen coordinate x (and y) is a linear relation of barycentric coordinates, a step of one unit in the x direction is equivalent to some constant step ($\Delta r, \Delta s,$ and Δt) in the r, s, and t directions, respectively.

The deltas are calculated as follows:

In the barycentric coordinate system local to triangle *ABC* (Figure 29), *r* varies from 1 to 0 between points *A* and *P*. The *x* coordinate of point *P* is given by:

$$x_P = x_B + \frac{y_B - y_A}{y_C - y_B}(x_B - x_C) \tag{74}$$

Thus, for a unit step in the *x* direction:

$$\Delta r = \frac{1}{x_A - x_P} \tag{75}$$

Δs and Δt for unit steps in the *x* direction are computed similarly as well as for unit steps in the *y* direction.

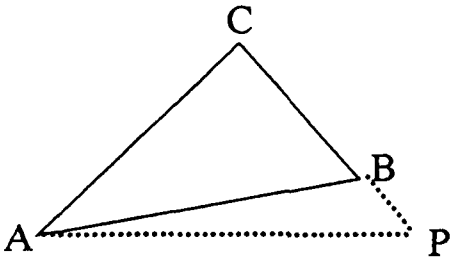


Figure 29. Constant Step in Barycentric Coordinates

In the barycentric coordinate system local to triangle *ABC*, *r* varies from 1 to 0 between points *A* and *P*.

The intensity at point $(x+1, y)$ can then be written as:

$$I(x+1, y) = I(r+\Delta r, s+\Delta s, t+\Delta t) = \sum_{\Delta} b_{i,j,k} B^2_{i,j,k}(r+\Delta r, s+\Delta s, t+\Delta t)$$

$$= b_{2,0,0}(r+\Delta r)^2 + b_{0,2,0}(s+\Delta s)^2 + b_{0,0,2}(t+\Delta t)^2 + 2b_{1,1,0}(r+\Delta r)(s+\Delta s) +$$

$$2b_{1,0,1}(r+\Delta r)(t+\Delta t) + 2b_{0,1,1}(s+\Delta s)(t+\Delta t) \tag{76}$$

Using forward differencing, initialize at (x_0, y_0):

$$I(x_0, y_0) = I(r_0, s_0, t_0) \tag{77}$$

$$\begin{aligned}\Delta I(x_0, y_0) = {} & b_{2,0,0}(2r_0\Delta r + \Delta r^2) + b_{0,2,0}(2s_0\Delta s + \Delta s^2) + b_{0,0,2}(2t_0\Delta t + \Delta t^2) + \\ & 2b_{1,1,0}(r_0\Delta s + s_0\Delta r + \Delta r\Delta s) + 2b_{1,0,1}(r_0\Delta t + t_0\Delta r + \Delta r\Delta t) + \\ & 2b_{0,1,1}(s_0\Delta t + t_0\Delta s + \Delta s\Delta t)\end{aligned} \tag{78}$$

$$\begin{aligned}\Delta^2 I = {} & b_{2,0,0}(2\Delta r^2) + b_{0,2,0}(2\Delta s^2) + b_{0,0,2}(2\Delta t^2) + \\ & 2b_{1,1,0}(2\Delta r\Delta s) + 2b_{1,0,1}(2\Delta r\Delta t) + 2b_{0,1,1}(2\Delta s\Delta t)\end{aligned} \tag{79}$$

Since we are iterating over subtriangles, we need not be as concerned with truncation error. Hence we may start at a vertex rather than at the centroid of the subtriangle. The above set of initialization equations simplifies since, at a vertex, two barycentric coordinates are zero and the other is equal to one.

We can then step in the x direction, evaluating intensities in just two additions at each iteration:

$$I(x_i+1, y) = I(x_i, y) + \Delta I(x_i, y) \tag{80}$$

$$\Delta I(x_i+1, y) = \Delta I(x_i, y) + \Delta^2 I \tag{81}$$

A similar set of equations exist for stepping in the y direction.

4.5 Computational Costs

The C^1 intensity surface generated by the new algorithm is given by a piecewise quadratic interpolant using the Bernstein-Bezier representation in barycentric coordinates. It was previously shown that the quadratic polynomials can be evaluated with just two additions

at each iteration. Tables 5 and 6 summarizes the number of floating-point operations required for each of the four steps in initializing one facet, applying the same assumptions as were made for the other shading algorithms.

Table 5. Start-up Costs Per Vertex — Quadratic Interpolation of Intensities

Step	+	x	/	√	**
Intensities	6	7			1
Tangents	24	13	9	2	
Total	30	20	9	2	1

Table 6. Start-up Costs Per Facet — Quadratic Interpolation of Intensities

Step	+	x	/	√	**
Bezier Points	46	49	4	1	
Interpolation Constants	131	225	7		
Total	177	274	11	1	

Intensities and tangents are calculated for each vertex in the polygon mesh. Bezier points and interpolation constants are determined for each triangular facet. Thus the cost of shading by quadratically interpolating intensities is:

$$Cost_{QI}(d,e,p) = (30t_1+20t_2+9t_3+2t_4+t_5)(d+e) + (177t_1+274t_2+11t_3+t_4)(2d+e-2) + 2t_1p \quad (82)$$

where d is the number of vertices in the interior of the triangulation, e is the number of vertices on the boundary, p is the total number of pixels on the computer monitor, and t_1 through t_5 are the CPU times for the five floating-point operations, as before.

Using the CPU times of the Sun computers (see Table 4), the computational cost is:

$$Cost_{QII}(d,e,p) = 349.36(d+e) + 2204.32(2d+e-2) + 7.36p \quad (83)$$

Making the same assumptions about the number of pixels and the relative magnitudes of d and e as in Section 2, we plot computational costs as a function of the number of facets (Figure 30). The costs for linearly interpolating intensities (LII) and linearly interpolating surface normals (LISN) from Figure 8 are superimposed. In addition, empirical costs for a million-pixel image are shown as individual data points on the graph. The empirical costs track the theoretical costs well for quadratic interpolation of intensities (QII), using the same 5-to-1 scale factor as in Figure 8. The costs for QII are less than LISN but more than LII; they are closer to LII due to relatively small iterative costs, but rise as rapidly as LISN due to relatively large start-up costs.

4.6 Multiple Light Sources

So far we have analyzed the computational costs in the presence of one light source. To generalize our results, we now consider multiple light sources. When interpolating intensities, whether linearly or quadratically, computational costs increase only marginally. To compute the intensity at each vertex, the contribution of each light source must be calculated by Equation 7 and then summed. The interpolation of intensity for each pixel then proceeds like the one-light-source case, i.e., the iterative costs are unchanged.

However, linear interpolation of surface normals must deal with each light source separately. Each light vector $\vec{L}$ and vector of maximum highlight $\vec{H}$ are interpolated separately. Iterative, and therefore total, costs increase by a factor equal to the number of light sources.

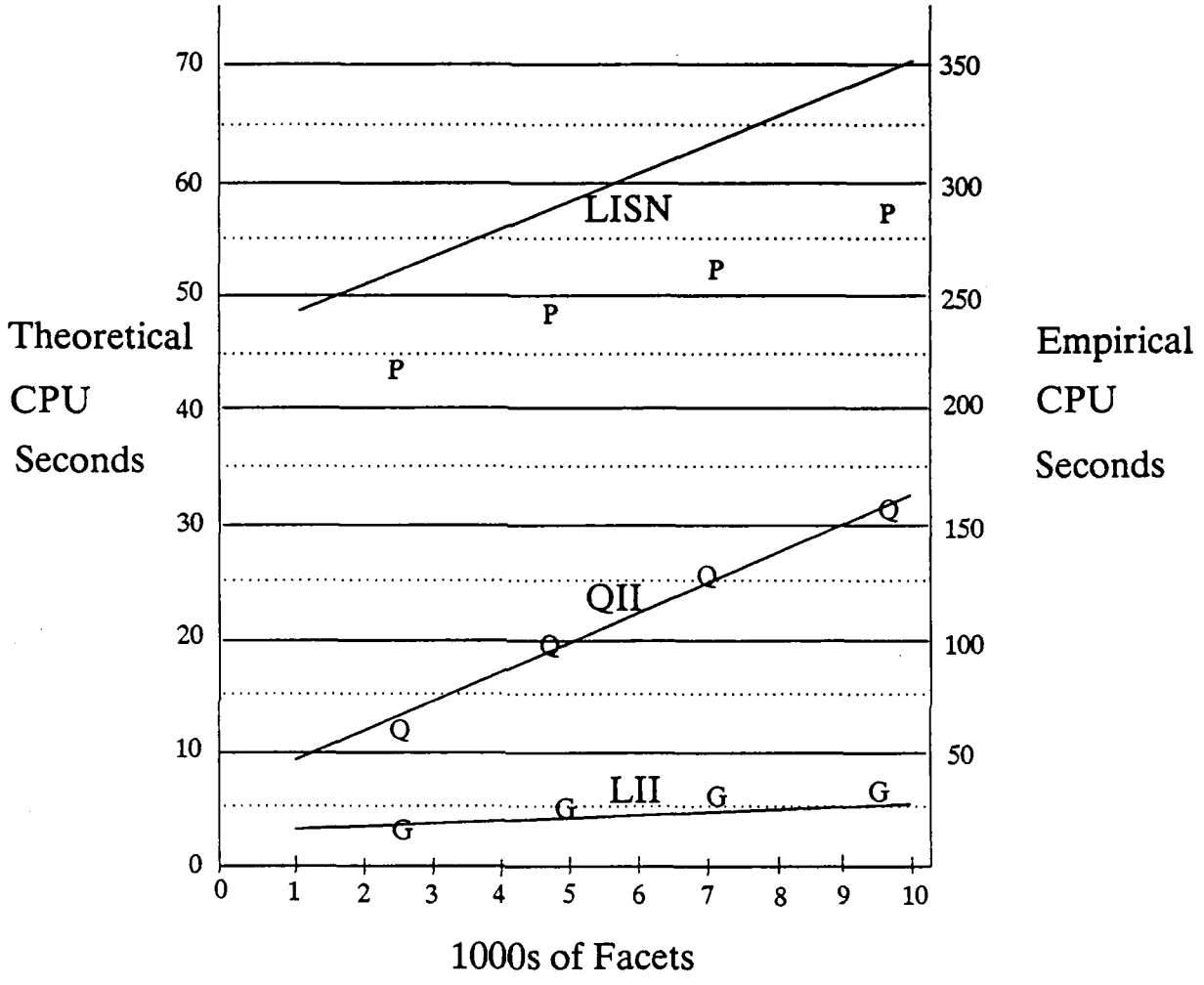


Figure 30. Computational Costs as a Function of the Number of Facets

Theoretical computational costs plotted as a function of the number of facets assuming a 1000-by-1000 pixel monitor and that the number of interior points is much larger than the number of boundary points. The theoretical costs correspond to Equations 33, 34, and 83, respectively. In addition, empirical costs for a million-pixel image are shown as individual data points (G = LII = Linear Interpolation of Intensities, Q = QII = Quadratic Interpolation of Intensities, P = LISN = Linear Interpolation of Surface Normals).

Tables 7, 8, and 9 break down the computational costs per vertex, per facet, and per pixel, respectively, for each shading algorithm in the presence of two point light sources.

Table 7. Start-up Costs Per Vertex — Two Light Sources

Shading Algorithm	Computational Cost				
	+	x	/	√	**
Linear Interpolation of Intensities	12	14			2
Linear Interpolation of Surface Normals	0				
Quadratic Interpolation of Intensities	36	27	9	2	2

Table 8. Start-up Costs Per Facet - Two Light Sources

Shading Algorithm	Computational Cost				
	+	x	/	√	**
Linear Interpolation of Intensities	14	9	2		
Linear Interpolation of Surface Normals	306	664	16	4	
Quadratic Interpolation of Intensities	177	274	11	1	

Table 9. Iterative Costs Per Pixel — Two Light Sources

Shading Algorithm	Computational Cost				
	+	x	/	√	**
Linear Interpolation of Intensities	1				
Linear Interpolation of Surface Normals	10	2			2
Quadratic Interpolation of Intensities	2				

The following assumptions were made in deriving computational costs:

1. The image is monochrome.
2. The polygonal facets are triangular.
3. Unit surface normals ($\vec{N}$) and unit light vectors ($\vec{L}$ and $\vec{R}$ or $\vec{H}$) are given at all vertices.

4. Two point light source are present
5. The viewpoint and point light source are at finite distances (the most general case).
6. Partial products have been pre-calculated to minimize cost.

Computational costs can be expressed as a function of the number of vertices in the interior of the triangulation (d), the number of vertices on the boundary (e), and the number of pixels (p) on the computer monitor. The number of facets in the triangulation is $2d+e-2$. Let t_1 through t_5 be the CPU times for the five floating-point operations (addition, multiplication, division, square root, and exponentiation), respectively. The cost of shading in the presence of two light sources is:

$$Cost_{LII}(d,e,p) = (12t_1+14t_2+2t_5)(d+e) + (14t_1+9t_2+2t_3)(2d+e-2) + t_1p \quad (84)$$

$$Cost_{LISN}(d,e,p) = (306t_1+664t_2+16t_3+4t_4)(2d+e-2) + (10t_1+2t_2+2t_5)p \quad (85)$$

$$Cost_{QII}(d,e,p) = (36t_1+27t_2+9t_3+2t_4+2t_5)(d+e) + (177t_1+274t_2+11t_3+t_4)(2d+e-2) + 2t_1p \quad (86)$$

Using the CPU times of Motorola's chips (see Table 4), the computational cost as a function of the number of vertices in the interior of the triangulation (d), the number of vertices on the boundary (e), and the total number of pixels on the monitor (p) is:

$$Cost_{LII}(d,e,p) = 163.68(d+e) + 114.72(2d+e-2) + 3.68p \quad (87)$$

$$Cost_{LISN}(d,e,p) = 4837.44(2d+e-2) + 92.96p \quad (88)$$

$$Cost_{QII}(d,e,p) = 431.20(d+e) + 2204.32(2d+e-2) + 7.36p \quad (89)$$

Making the same assumptions about the number of pixels and the relative magnitudes of d and e as in Section 2, we plot computational costs as a function of the number of facets in the presence of two, three, or four point light sources (Figures 31, 32, and 33 for each

of the three shading algorithms). Note the marginal increase in costs for LII and QII as the number of light sources increase while LISN costs increase arithmetically.

In addition, empirical costs are superimposed as individual data points in Figures 31, 32, and 33. CPU times were obtained at four levels of resolution for each shading algorithm while rendering a million-pixel image in the presence of two (2) and four (4) light sources. The same 5-to-1 scale factor between theoretical and empirical costs is used as in Figures 8 and 30.

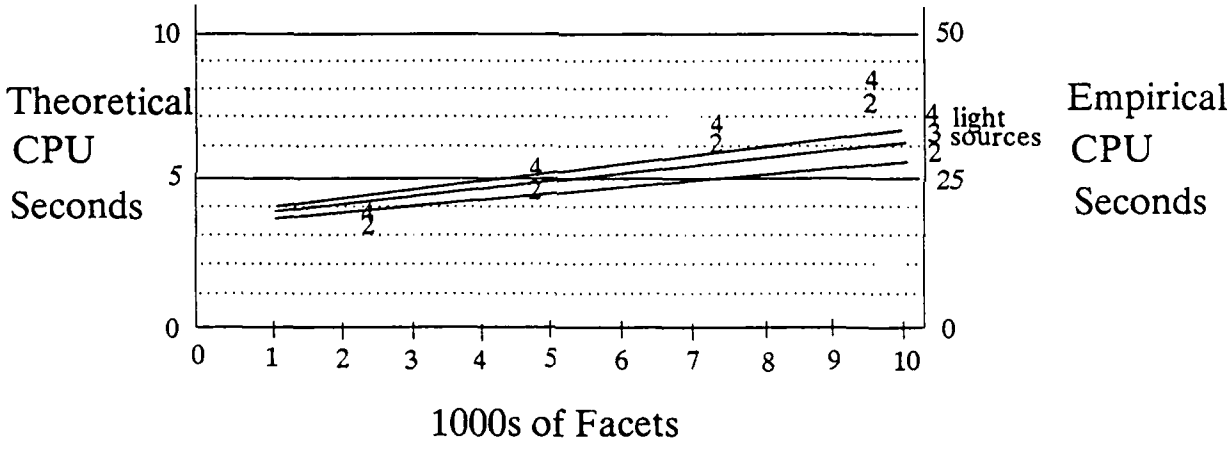


Figure 31. LII Computational Costs - Multiple Light Sources

Computational costs for linear interpolation of intensities plotted as a function of the number of facets in the presence of two, three, or four light sources, assuming a 1000-by-1000 pixel monitor and that the number of interior points is much larger than the number of boundary points. Theoretical costs correspond to Equation 87. Empirical costs are superimposed as individual data points for rendering a million-pixel image in the presence of two (2) and four (4) light sources.

4.7 Summary

The innovation of this new approach is to fit a smoother intensity surface to scattered

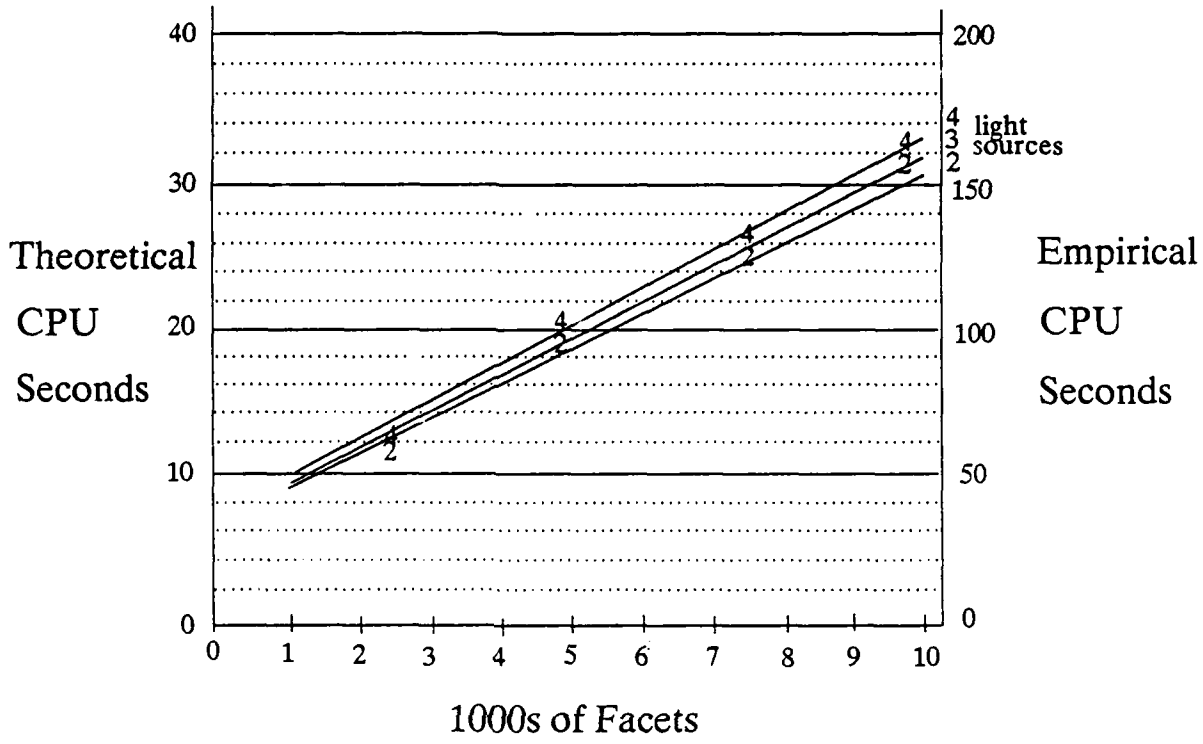


Figure 32. QII Computational Costs - Multiple Light Sources

Computational costs for quadratic interpolation of intensities plotted as a function of the number of facets in the presence of two, three, or four light sources, assuming a 1000-by-1000 pixel monitor and that the number of interior points is much larger than the number of boundary points. Theoretical costs correspond to Equation 89. Empirical costs are superimposed as individual data points for rendering a million-pixel image in the presence of two (2) and four (4) light sources.

intensity data using a geometric surface-fitting technique. It can be viewed as a natural extension to linearly interpolating intensities which uses first-order piecewise polynomials to fit a C^0 intensity surface. The new approach uses quadratic (second-order) piecewise polynomials to fit a C^1 intensity surface.

Quadratic interpolation of intensities virtually eliminates the Mach band problem plaguing linear interpolation of intensities. Figure 34a shows an image of a teapot shaded by linearly interpolating intensities. Mach bands are apparent near the right side and

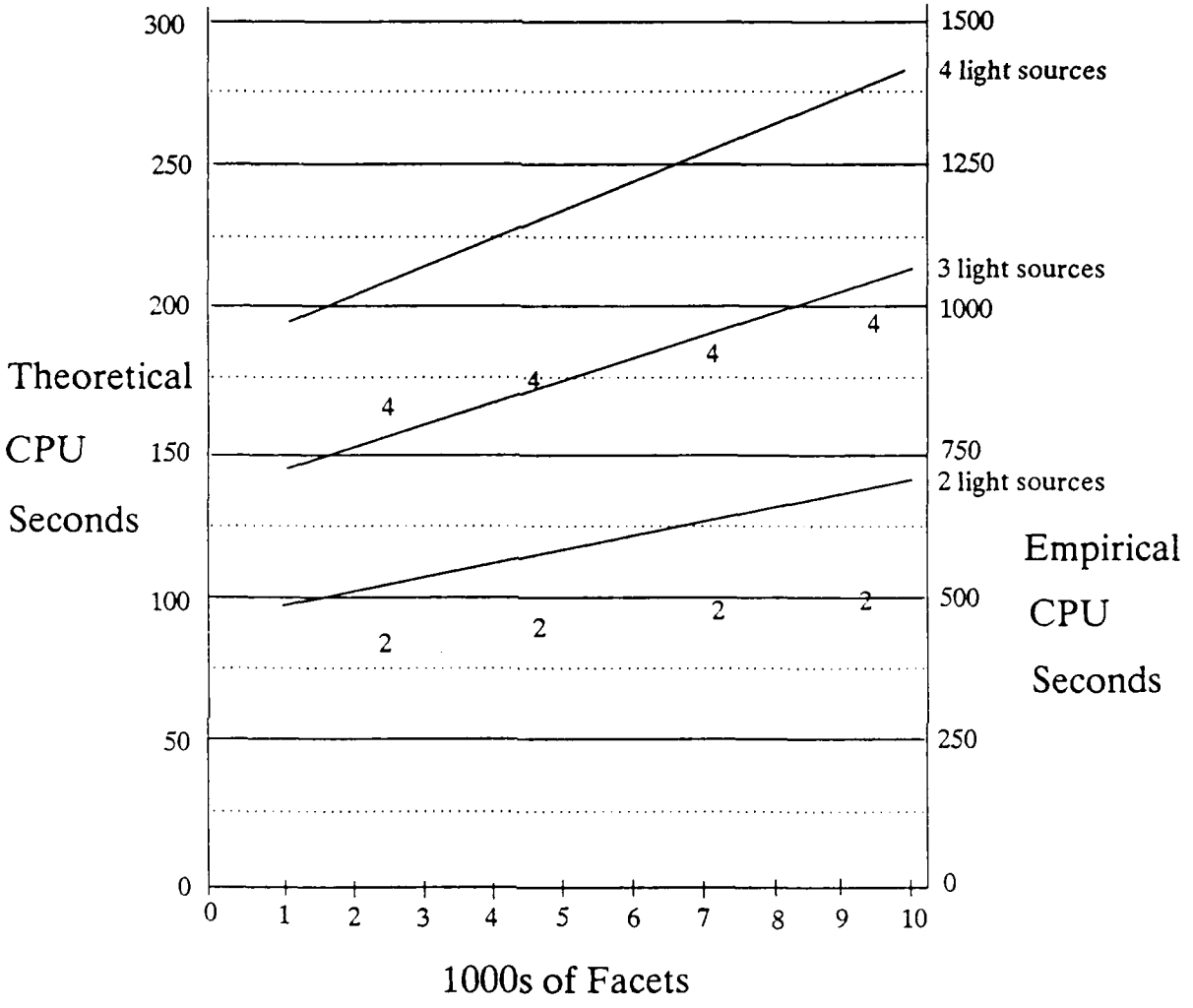


Figure 33. LISN Computational Costs - Multiple Light Sources

Computational costs for linear interpolation of surface normals plotted as a function of the number of facets in the presence of two, three, or four light sources, assuming a 1000-by-1000 pixel monitor and that the number of interior points is much larger than the number of boundary points. Theoretical costs correspond to Equation 88. Empirical costs are superimposed as individual data points for rendering a million-pixel image in the presence of two (2) and four (4) light sources.

lower portion of the teapot body. Figure 34b shows the same teapot (identical lighting and surface characteristics) shaded with quadratic interpolation. It appears more realistic

(a) (b)

Figure 34. Linear vs. Quadratic Interpolation of Intensities

Quadratic interpolation of intensities virtually eliminates the Mach band problem plaguing linear interpolation of intensities. Mach bands are apparent near the right side and lower portion of the teapot body in the first image shaded by linearly interpolating intensities. These Mach bands are absent in the second image shaded by quadratic interpolation.

due to the absence of the Mach bands.

The linear shading of Figure 34a did not take advantage of the adaptive subdivision scheme proposed in Section 3.7. Both quadratic shading and adaptive subdivision address the Mach banding problem, though with differing costs. The trade-off in their realism and speed will be presented in Section 6.2.

Quadratic interpolation of intensities ameliorates the problem of inconsistent motion which afflicts both linear interpolation of intensities and linear interpolation of surface normals. For the linear schemes, the shade determined for a pixel depends upon the values of intensity or surface normal at each of the three vertices of the triangular facet that it's in. Thus its shade may change contrary to expectations in a motion sequence if it suddenly depends upon a different set of vertices. On the other hand, for quadratic interpolation, the shade of a pixel depends upon intensity values at the three vertices as well as those at vertices of adjacent facets. Therefore, it is less likely that motion would

cause inconsistent results.

However, shading inconsistencies due to motion may potentially arise at the silhouette edges. The tangents to the intensity surface at vertices on the silhouette do not depend on the intensity values at adjacent but hidden vertices. If the object is then rotated or the viewpoint changes, these tangents become dependent on previously hidden vertices. The resulting shading change may be contrary to expectations.

Quadratic interpolation achieves these improvements at a modest increase in computational cost, two additions per pixel vs. one for linear interpolation of intensities. However, linear and quadratic interpolation of intensities share a certain disadvantage: both handle specular reflection poorly. The next section addresses this problem.

5. ADDITIONAL TOOLS

5.1 Specular Reflection

Specular reflection is currently handled poorly when interpolating intensities, whether linearly or quadratically. If the peak of a highlight happens not to fall on a vertex of the polygon mesh, the highlight may be entirely missed. Because the intensity of a specular highlight falls off exponentially, the intensity surface will be interpolated somewhere through the base of the highlight and inadvertently chop off its peak.

Specular reflection can be rendered realistically when interpolating intensities if a finer mesh of data points is generated in the region of the highlights To accomplish this, highlights must first be located on the intensity surface. Then the surface may be adaptively subdivided in regions with large intensity gradients.

Phong and Crow [BUI75a] presented a technique for locating specular highlights to address the high cost of linearly interpolating surface normals. They argued that surface normals can be linearly interpolated in facets near highlights, while the less expensive technique of linearly interpolating intensities can be applied in all other facets. Highlights are located by first transforming the reflected light vectors in a perspective space relative to the viewpoint. If a transformed vector passes near the viewpoint, its transformed z-component approaches one. The reflected light vector at each vertex is examined to determine whether or not the highlight is near.

This technique by Phong and Crow for locating highlights is not exact enough for our purposes, since only the "nearness" of a highlight can be determined. We now present a technique for locating a specular highlight precisely.

5.1.1 Locating Specular Highlights

The first step in capturing specular highlights is determining their positions. The peak of a highlight is located at the point where the surface normal and the vector of maximum highlight ($\vec{H}$) are coincident. If the surface normal can be expressed as a function of world coordinates as in Equation 69, the world-coordinate position of the highlight on the surface can be determined explicitly. A new data point is added there. The surface normal is set equal to $\vec{H}$ and the intensity of reflected light is given by Equation 7.

Alternatively, if the surface normal is not a function of world coordinates, the vector of maximum highlight is represented in the barycentric coordinate system defined by the surface normals at the three vertices of a triangular facet. Let points A, B, and C be the vertices of a facet with surface normals $\vec{N}_A$, $\vec{N}_B$, and $\vec{N}_C$, respectively. The unit surface normals can be mapped to the unit sphere (Figure 35). A specular highlight will fall in triangle ABC if some interpolated value between normals $\vec{N}_A$, $\vec{N}_B$, and $\vec{N}_C$ equals $\vec{H}$.

This approach to locating a specular highlight works only if $\vec{H}$ is constant, i.e., when light sources are infinitely far. Remember that the peak of a highlight is also located at the point where the reflected light vector ($\vec{R}$) and eye direction ($\vec{V}$) are coincident. Thus when light sources are at finite distances, we simply use $\vec{R}$ and $\vec{V}$ in lieu of $\vec{H}$ and $\vec{N}$. The constant $\vec{V}$ is represented in the barycentric coordinate system defined by the reflected light vectors at the three vertices. A specular highlight will fall in triangle ABC if some interpolated value between reflected light vectors $\vec{R}_A$, $\vec{R}_B$, and $\vec{R}_C$ equals $\vec{V}$.

We compute the barycentric coordinates (r_V, s_V, t_V) of $\vec{V}$ with respect to the polar coordinates of the surface normals:

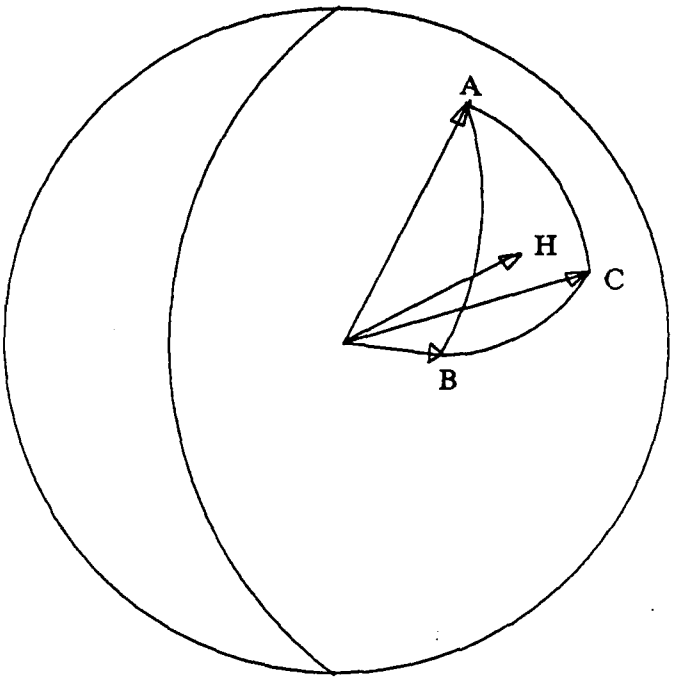


Figure 35. Surface Normals on Unit Sphere

A specular highlight will fall in triangle ABC if some interpolated value between normals $\vec{N}_A$, $\vec{N}_B$, and $\vec{N}_C$ equals the vector of maximum highlight $\vec{H}$.

$$t_V = \frac{V_y - R_{1y} - \dfrac{(R_{3y}-R_{1y})(V_x-R_{1x})}{R_{2x}-R_{1x}}}{\dfrac{(R_{2y}-R_{1y})(R_{1x}-R_{3x})}{R_{2x}-R_{1x}} - R_{y1}+R_{y3}} \tag{90}$$

$$s_V = \frac{V_x - R_{1x} + (R_{1x} - R_{3x})\, t_V}{R_{2x}} - R_{1x} \tag{91}$$

$$r_V = 1 - s_V - t_V \tag{92}$$

If r_V, s_V, and t_V are all in the range 0 to 1, the highlight falls within the triangular facet. In fact, the position of the highlight in screen coordinates is given by:

$$x_H = x_A r_V + x_B s_V + x_C t_V \tag{93}$$

$$y_H = y_A r_V + y_B s_V + y_C t_V \tag{94}$$

where (x_A, y_A), (x_B, y_B), and (x_C, y_C) are the screen coordinates of vertices A, B, and C, respectively.

If two of the surface normals are coincident, an alternate procedure is followed. For a highlight to fall within the facet, the vector of maximum highlight must fall between the two different normals. First, the dot products between the two normals and the vector of maximum highlight are calculated. Referring to Figure 36, *dot* must be less than both *dot* 1 and *dot* 2. Next, to check if $\vec{H}$ is in the plane formed by $\vec{N}_1$ and $\vec{N}_2$, the cross product of $\vec{N}_1$ and $\vec{N}_2$ is taken. If the dot product of this vector and $\vec{H}$ is zero, $\vec{H}$ is in the plane. The relative position of the highlight is given by arc cos(*dot* 1) / arc cos(*dot*). A new point is then added along each of two edges of the triangular facet.

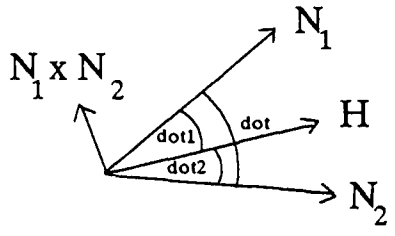


Figure 36. Highlight Between Two Normals

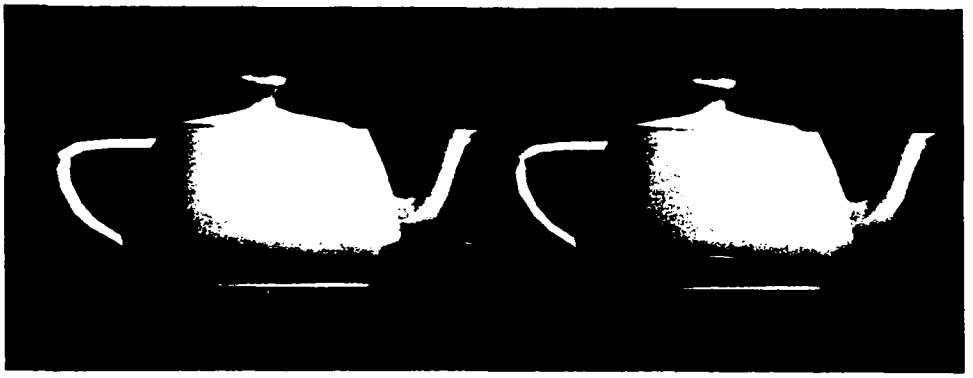
For a highlight to fall within the facet, the vector of maximum highlight must fall between the two different normals. *dot* must be less than both *dot* 1 and *dot* 2 and $\vec{H}$ must fall in the plane formed by $\vec{N}_1$ and $\vec{N}_2$.

5.1.2 Adaptive Subdivision - Specular Component

Although the peak of the specular highlight is captured, the intensity may fall off too rapidly to be accurately interpolated by a quadratic polynomial. The solution is to add more data points. If the change in specular component between two adjacent points is greater than a certain threshold, a new point is added at the screen position midway between the two. The corresponding world coordinate position, if needed, can be obtained by applying the inverse of the viewing transform matrix. A surface normal can be calculated explicitly in world coordinates or be linearly interpolated between the two endpoints. The intensity of reflected light is then obtained with Equation 7.

Figure 37a shows an image of a teapot shaded by linearly interpolating intensities without locating specular highlights nor adaptively subdividing, while the image in Figure 37b has been enhanced by these techniques. Crisper and more realistic highlights are apparent in the second image. Figures 38a and b give the corresponding polygon meshes. Note the additional subdivisions on the surface of the teapot in areas of specular reflection.

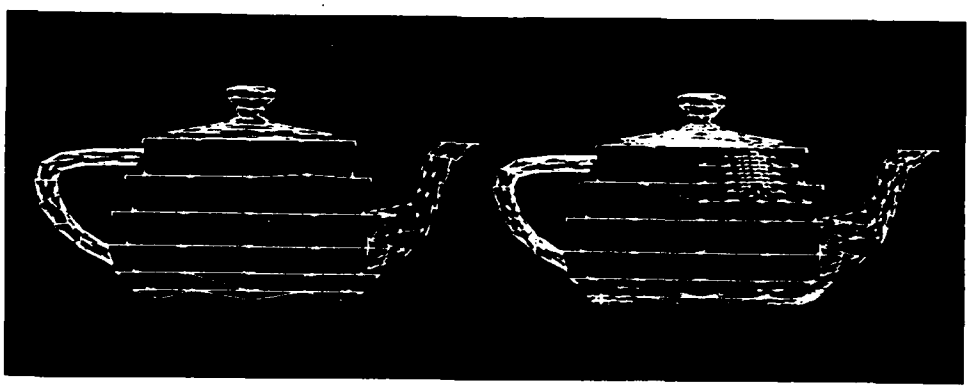
The costs of locating specular highlights and adaptively subdividing based on the specular component can be greatly reduced when dynamically rendering a sequence of images by taking advantage of frame-to-frame coherence. When objects, light sources, and/or viewpoint move in space and time, the specular intensity surface changes. However, it need not be completely recomputed each frame since changes are relatively minor from one frame to the next. If each frame represents a one-tenth of a second time step, it is expected that a highlight appearing in a given facet will next appear in that same facet or one of its near neighbors. The costs of locating the highlight can then be limited to only a few facets. Furthermore, the data points added through adaptive subdivision need not be



(a) (b)

Figure 37. Adaptive Subdivision - Specular Component

(a) An image of a teapot shaded by linearly interpolating intensities and (b) an image enhanced by locating specular highlights and adaptively subdividing. Highlights are crisper and more realistic in the second image.



(a) (b)

Figure 38. Adaptive Subdivision - Specular Component - Polygon Meshes

The polygon meshes corresponding to Figure 37. Areas of specular reflection are adaptively subdivided in the second teapot.

discarded after each frame. These added vertices can be reused and discarded only when the highlight has migrated far enough away, i.e., when the specular component has dropped below a certain threshold.

5.2 Diffuse Boundary

A C^1 intensity surface is generated when quadratically interpolating intensities. However, the real intensity surface may have areas that are C^1 discontinuous. Other than the obvious discontinuities at real edges (i.e., not facet edges), C^1 discontinuities naturally arise at the perceptual boundary where the diffuse component becomes zero. This occurs when the dot product of the surface normal and a light vector is zero, i.e., the vectors are orthogonal. Without correction, quadratic interpolation would smooth this discontinuity, hindering realism (Figure 39).

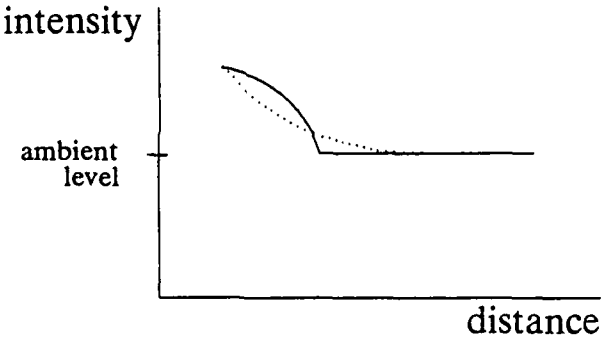


Figure 39. Quadratic Smoothing of the Diffuse Boundary

Quadratic interpolation (dotted line) smooths the natural C^1 discontinuity arising at the diffuse boundary.

Furthermore, when linearly interpolating intensities, the diffuse boundary is misplaced. Consider two adjacent vertices whose surface normals are greater and less than 90° from a light vector, respectively. As shown in Figure 40, the diffuse boundary would fall on the former vertex rather than at some point in between the two.

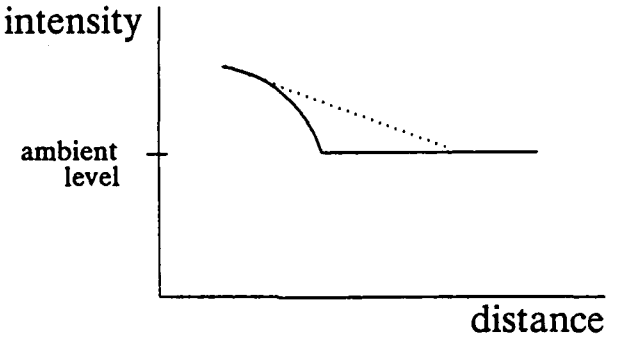


Figure 40. Linear Misplacement of the Diffuse Boundary

When linearly interpolating intensities (dotted line), the diffuse boundary is misplaced.

5.2.1 Ambient Cut-Off

For the case of one light source, the C^1 discontinuity can be captured at the diffuse boundaries by allowing the diffuse component, given by Equation 2, to become negative, i.e., when the angle between the light vector and the surface normal exceeds 90°. We interpolate as usual, but then compare the interpolated intensity with the ambient light level. The intensity of the pixel is set to the greater of these two values (Figure 41). Note that the specular component must be zero when the angle between the light vector and surface normal exceeds 90°.

Figure 42a shows an image of the teapot without correcting for the diffuse boundary, while ambient cut-off has been used for the teapot of Figure 42b. Note the better-defined diffuse boundary on the body of the second teapot.

This correction requires one comparison at each pixel. Assume that the computational cost of a comparison is equivalent to that of an addition operation. Thus one unit of computation is added to the iterative costs of the shading algorithms when capturing

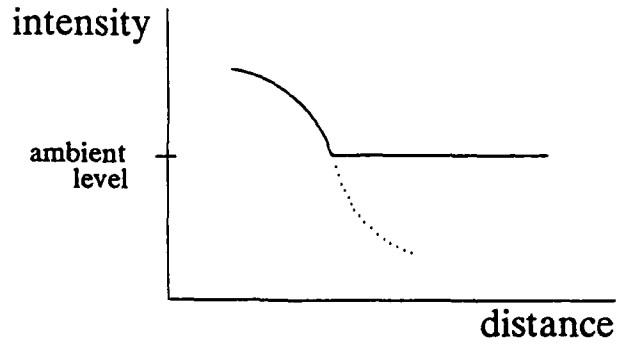


Figure 41. C^1 Discontinuity at Diffuse Boundary

The C^1 discontinuity can be captured at the diffuse boundaries by allowing the diffuse component to become negative (dotted line). The calculated intensity is then cut off at the ambient light level.

Figure 42. Ambient Cut-Off

(a) An image of the teapot and (b) an enhanced image correcting for the diffuse boundary. Ambient cut-off captures the diffuse boundary.

diffuse boundaries with one light source.

5.2.2 Locating Diffuse Boundaries

For the case of two or more light sources, a different approach is required. Consider the diffuse contributions of two light sources in Figure 43. To capture accurate perceptual

boundaries, new data points are added along their lengths. A new point is added between two adjacent screen-coordinate points (x_1, y_1) and (x_2, y_2) if their diffuse components $diffuse_1$ and $diffuse_2$, respectively, are of different sign (again allowing Equation 2 to become negative). We linearly interpolate the diffuse component between these two points to determine the zero crossing. The screen coordinates of the new point are then given by:

$$x_{new} = x_1 factor + x_2(1-factor) \tag{95}$$

$$y_{new} = y_1 factor + y_2(1-factor) \tag{96}$$

where

$$factor = \frac{diffuse_2}{diffuse_2 - diffuse_1} \tag{97}$$

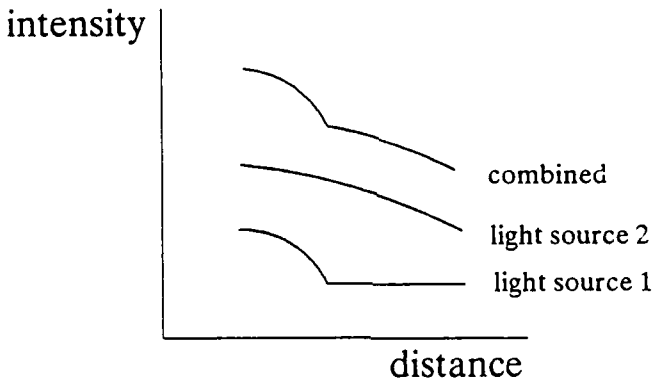


Figure 43. Diffuse Boundary with Two Light Sources

For two or more light sources, ambient cut-off cannot be used to capture the diffuse boundary. New points are added on edges where the diffuse component becomes zero.

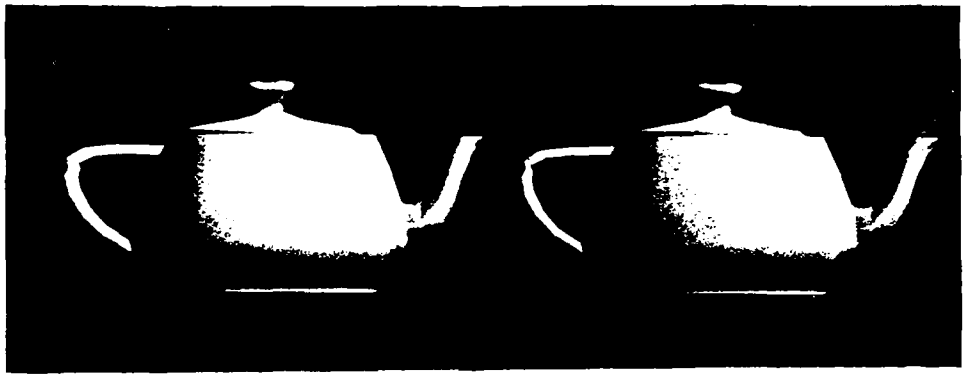
The corresponding world coordinate position is obtained by applying the inverse of the viewing transform matrix. The surface normal is determined explicitly if it can be

expressed as a function of world coordinates as in Equation 69. Alternatively, the surface normal is obtained by linearly interpolating the surface normals between the two endpoints of the edge. The intensity of reflected light is then obtained with Equation 7. However, tangents require special treatment.

The tangent vectors differ on either side of the diffuse boundary due to the C^1 discontinuity. Using tangent estimation, tangents on each side should only be based upon neighboring points that lie on that same side. Alternatively, tangents can be calculated explicitly. On the lighter side, tangents are based upon contributions from both light sources, while on the darker side, tangents are based only upon the contribution of the light source with the positive diffuse component.

Figure 44a shows an image of the teapot without correction for diffuse boundaries, while they have been located for the teapot of Figure 44b. Note the better-defined diffuse boundary on the body of the second teapot. Figures 45a and b show the corresponding polygon meshes. Note the points added along the diffuse boundaries.

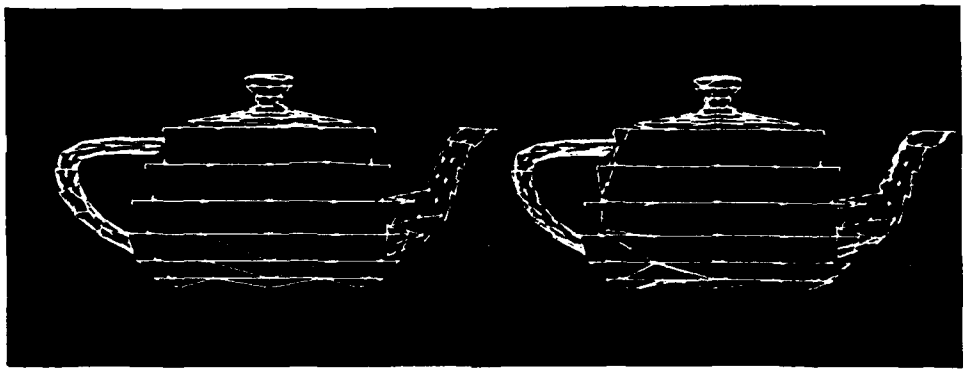
The costs of locating diffuse boundaries can be greatly reduced when dynamically rendering a sequence of images by taking advantage of frame-to-frame coherence. When objects, light sources, and/or viewpoint move in space and time, the diffuse intensity surface changes. If each frame represents a one-tenth of a second time step, the diffuse boundary will, however, not migrate far from one frame to the next. The costs of locating the diffuse boundary can then be limited to the same set of facets through which the boundary passes as well as their near neighbors.



(a) (b)

Figure 44. Locating Diffuse Boundaries

(a) An image of the teapot and (b) an enhanced image in which the diffuse boundary is located.



(a) (b)

Figure 45. Locating Diffuse Boundaries - Polygon Meshes

The polygon meshes corresponding to Figure 44. New points were added at the diffuse boundary in the second teapot.

6. TRADE-OFFS

We now have a complete toolbox for shading three-dimensional objects on a raster-scan display. When linearly interpolating intensities, the toolbox comprises:

- adaptively subdividing the diffuse intensity surface to reduce Mach banding
- locating specular highlights
- adaptively subdividing specular intensity surfaces
- locating diffuse boundary
- using ambient cut-off for diffuse boundary in presence of only one light source

When quadratically interpolating intensities, the toolbox comprises:

- locating specular highlights
- adaptively subdividing specular intensity surfaces
- locating diffuse boundary
- using ambient cut-off for diffuse boundary in presence of only one light source

When linearly interpolating surface normals, the toolbox comprises:

- adaptively subdividing the object's surface to reduce approximation error

We now consider how to build a shading algorithm which is most appropriate for a given application and hardware environment. More specifically, "appropriateness" depends on realism requirements, computational costs, lighting and surface characteristics, image complexity, and perception. To enable the reader to build a shading algorithm with appropriate tools, a trade-off study, accompanied by timing analyses, follows.

6.1 Original Triangulation vs. Adaptive Subdivision

The shading algorithms under consideration start with a triangular mesh that approximates the curved surfaces of an object to be rendered. We refer to this mesh as the original triangulation. It may or may not be sufficient for shading purposes, that is, satisfy the realism requirements of a particular application. When insufficient, adaptive subdivision can be applied while shading. The question remains, what is the trade-off between the original triangulation and adaptive subdivision to satisfy realism requirements?

In cases where it is more cost effective to adaptively subdivide, one may take the argument to the extreme: let the original triangulation be just one facet and adaptively subdivide while shading to meet shading requirements. While this will work in theory, the silhouette edge will be too coarse. To smooth the silhouette edge, a technique developed by Phong and Crow [BUI75a] can be applied independently.

Figure 46 shows two images of a teapot that have been shaded by linearly interpolating surface normals. Figure 46a started with 56 facets and was adaptively subdivided to yield 250 facets. Figure 46b started with 248 facets and was not adaptively subdivided. The first image has a sharper highlight and smaller approximation error. The second image has a more accurately placed highlight and a more faithful silhouette edge. While their run times are equivalent (25.14 vs. 25.16 seconds, respectively), the second image is clearly superior. Thus up to a point, it is more cost effective to start with a finer polygon mesh.

Typically, the triangular approximation of curved surfaces is performed once. Sometimes we have no choice; the triangular mesh may have been previously generated and the

(a) (b)

Figure 46. Silhouette Edge vs. Adaptive Subdivision

The first teapot started with 56 facets and was adaptively subdivided to yield 250 facets. The second teapot comprises 248 facets and was not adaptively subdivided. The first image has a sharper highlight and smaller approximation error. The second image has a more accurately placed highlight and a more faithful silhouette edge. While their run times are equivalent, the second image is clearly superior.

specifications of the original surfaces are now unavailable. Or the world coordinates on the surface of a real object may have been obtained with a 3D digitizer. Even if the original surface is available, the determination of world coordinates, surface normals, and adjacency for all surfaces is computationally expensive. There is not enough time in the graphics pipeline to re-triangulate each time surface shading changes.

One-time triangulation can be performed to satisfy static geometric requirements. On the other hand, the triangulation needed to satisfy shading requirements varies from one rendering to the next. As lighting, surface characteristics, and/or orientation change, the intensity surface changes, possibly requiring re-triangulation. Therefore, the original triangulation cannot be guaranteeed to satisfy shading requirements for all possible shaded renderings.

6.1.1 Object Surface Curvature

Consider the relationship between surface geometry and shading. When linearly interpolating surface normals via Taylor series (Section 2.3.3), geometry and shading are directly related. Suppose that the original triangulation is obtained by adaptively subdividing the original curved surface. If the geometric requirement is for each approximating subsurface to be sufficiently flat, the shading requirement is satisfied. Each triangular facet will then be flat enough to preclude significant approximation error in the Taylor series.

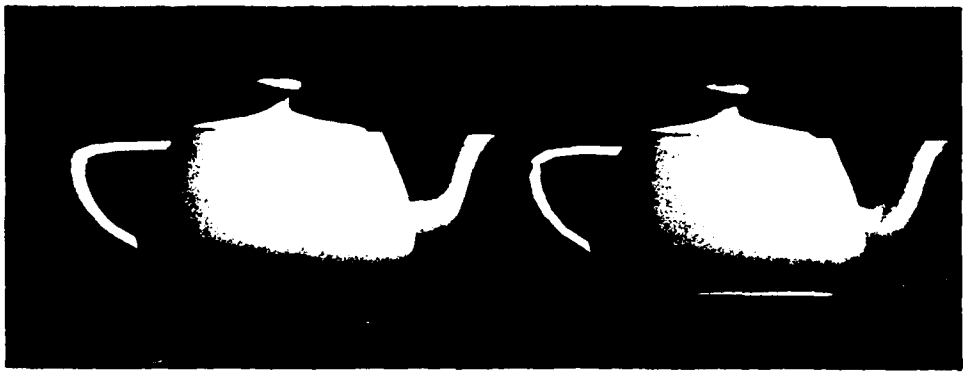
However, if the original curved surface is represented parametrically, curvature cannot be uniformly controlled. The triangular mesh is obtained by stepping incrementally in parameter space, but step size is not directly related to curvature. Since each facet is not guaranteed to be flat enough, adaptive subdivision based on surface normals (Section 3.6) is our only recourse. Object surface curvature is dependent only on the surface geometry, not on the lighting and surface reflectance characteristics. Adaptive subdivision of the polygon mesh can therefore be performed once before shading or for the visible surfaces while shading.

6.1.2 Intensity Surface Curvature

When linearly interpolating intensities and adaptively subdividing to reduce Mach banding (Section 3.7), geometry and shading are somewhat less related. The diffuse intensity surface is related to the geometric surface by the dot product(s) of the surface normal and light vector(s). Assume that the geometric surface was triangulated so that curvature does not exceed 10° for any facet. When the surface normal and light vector

are near coincident, the maximum change in their dot product over a facet is $\cos 0° - \cos 10° = .015$. This difference is small enough to preclude Mach banding. However, when the surface normal and light vector are nearly orthogonal, the maximum change in their dot product over a facet is $\cos 80° - \cos 90° = .174$. This difference may be sufficient to produce Mach bands, given the appropriate light intensity and surface reflectance characteristics.

Figure 47a shows an image of a teapot comprising 944 facets and shaded without adaptive subdivision. Figure 47b started with 248 facets and was adaptively subdivided to yield 632 facets. The first image was generated in 9.22 seconds, while the second took 9.28. Though these CPU run times are comparable, Mach banding appears to be less severe when starting with a finer polygon mesh, not to mention the more faithful silhouette edge.



(a) (b)

Figure 47. Original Triangulation vs. Adaptive Subdivision - Intensity Surface Curvature

The first teapot comprises 944 facets and was shaded without adaptive subdivision. The second teapot started with 248 facets and was adaptively subdivided to yield 632 facets. Though their run times are comparable, Mach banding appears to be less severe when starting with a finer polygon mesh.

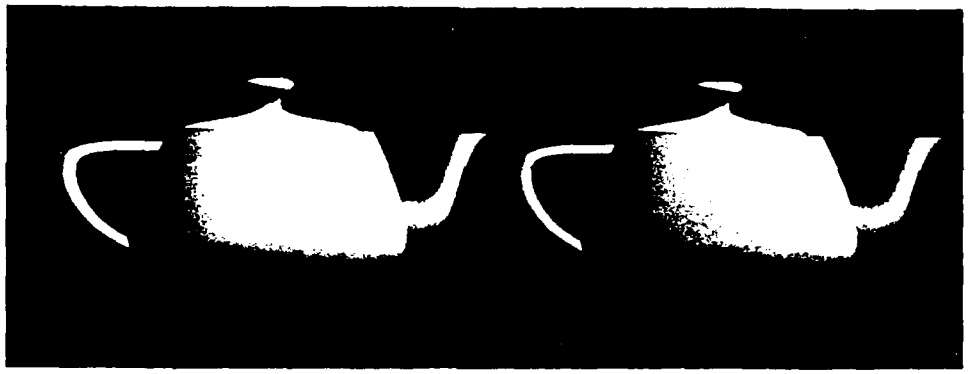
6.1.3 Specular Component

When adaptively subdividing to capture specular highlights (Section 5.1.2), geometry and shading are virtually unrelated. The specular intensity surface is related to the geometric surface by a power function of the dot product(s) between the viewpoint and reflected light vector(s). Let the power be an average value, say 50, and again assume that the geometric surface was triangulated so that curvature does not exceed 10° for any facet. Let a specular peak fall on one facet vertex, i.e., the viewpoint and reflected light vector are coincident, so that the specular component equals 1. An adjacent vertex whose surface normal differs by 10° would then have a specular component equal to $\cos 20^{\circ\ 50} = .04$. This difference of 0.96 in the specular component reflects a large intensity change over one facet. Thus a 10° difference in surface curvature cannot be guaranteed to capture specular highlights.

However, it is not cost effective to start with a finer polygon mesh because, in most areas, the smaller facets are not needed. It is better to capture the large intensity gradients associated with highlights by adaptively subdividing based on the specular component. Figure 48a shows a teapot comprising 944 facets and shaded without adaptive subdivision. Figure 48b started with 248 facets and was adaptively subdivided to yield 595 facets. The specular highlight of Figure 48b is not only superior to that of Figure 48a, the second image using adaptive subdivision was generated faster (24.24 vs. 16.98 seconds, respectively).

6.1.4 Diffuse Boundary

The previous subsections addressed adaptive subdivision with respect to the object



(a) (b)

Figure 48. Original Triangulation vs. Adaptive Subdivision - Specular Component

The first teapot comprises 944 facets and was shaded without adaptive subdivision. The second teapot started with 248 facets and was adaptively subdivided to yield 595 facets. The specular highlight of the latter is superior and was generated faster.

surface, the intensity surface, and the specular component. The relationship between geometry and shading varied from direct to cosine to power function, respectively. However, when applying adaptive subdivision to capture the diffuse boundary(ies) (Section 5.2.2), there is no corresponding functional relationship.

By definition, a diffuse boundary falls on an edge when the surface normal is orthogonal to a light vector. Let the surface normals at two adjacent vertices be greater and less than 90° from a light vector, respectively. Without adaptive subdivision, the diffuse boundary would fall on the former vertex rather than at some point in between the two. The degree of misplacement depends mostly on the length of the edge. If we again assume that the geometric surface was triangulated so that curvature does not exceed 10° for any facet, no bounds can be placed on edge lengths.

Therefore, the geometrical requirement of surface curvature cannot be guaranteed to satisfy the shading requirement for diffuse boundaries. Since the original triangulation

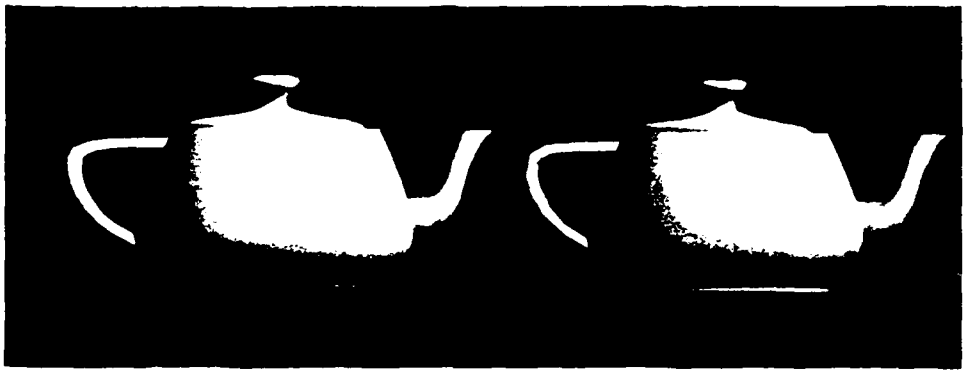
will probably not be sufficient to capture diffuse boundaries, we must resort to adaptive subdivision.

6.2 Linear vs. Quadratic Interpolation of Intensities

The previous section addressed trade-offs within a given shading algorithm. This section explores the trade-off in visual realism vs. computational costs between two algorithms. Specifically, quadratic interpolation of intensities attacks the problem of Mach banding by fitting a C^1 intensity surface. Similarly, a finer original triangulation when linearly interpolating intensities reduces Mach banding by decreasing the magnitudes of C^1 discontinuities.

Figure 49a shows an image of a teapot with 1168 facets shaded by linearly interpolating intensities. The teapot of Figure 49b has 248 facets and was shaded by quadratic interpolation. Their CPU run times are 10.10 and 10.84 seconds, respectively. Though their run times are comparable, linear interpolation does a superior job in shading with only about five times as many facets. Though the Mach bands are not completely eliminated, Figure 49a has a smoother appearance and a more faithful silhouette edge.

The conclusion that linear interpolation of intensities does a better job than quadratic interpolation is based on the assumption that a fine enough polygon mesh is given. However, if we are given a very coarse polygon mesh, quadratic interpolation of intensities has a clear advantage. While the quadratically interpolated intensity surface may not be very accurate, at least we are guaranteed C^1 continuity. Furthermore, with very few vertices and facets, the start-up costs for quadratic interpolation are reasonably small, while iterative costs are fixed at just twice that of linear interpolation.



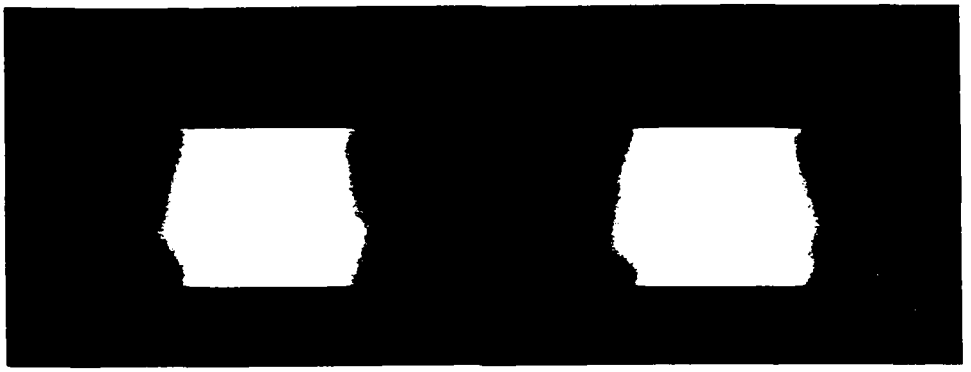
(a) (b)

Figure 49. Linear vs. Quadratic Interpolation

The first teapot comprising 1168 facets was shaded by linearly interpolating intensities. The second teapot has 248 facets and was shaded by quadratic interpolation. Though their run times are comparable, linear interpolation generates a smoother appearance and more faithful silhouette edge.

In the extreme, consider a teapot whose body comprises only twelve triangles. Figure 50a shows its image rendered by linearly interpolating intensities and Figure 50b by quadratic interpolation. With only twelve triangles, quadratic interpolation still does a reasonable job in smooth shading, while linear interpolation is clearly deficient.

With very coarse meshes, nonlinear foreshortening, a problem not previously addressed, may appear. Each of the shading algorithms under investigation interpolates in screen space. None take into account the foreshortening associated with the prospective projection from three dimensions to two. While the errors introduced are typically small [HECK86], they may become noticeable when the polygonal facets are large. Though we will not consider the problem of nonlinear foreshortening further, we mention in passing that the correct solution is obtained by an extra division at each pixel [HECK86].



(a) (b)

Figure 50. Linear vs. Quadratic Interpolation - Coarse Triangulation

With only twelve triangles, the quadratic interpolation of the first image still does a reasonable job in smooth shading, while the linear interpolation of the second image is clearly deficient.

6.3 Linear Interpolation of Surface Normals vs. Adaptive Subdivision

Unassisted, linear interpolation of surface normals does the best job of capturing specular reflections among the three basic shading methods. However, this algorithm is significantly slower than the other two. By locating specular peaks and adaptively subdividing based on the specular component, linear and quadratic interpolation of intensities render realistic highlights at some expense.

One can argue that quadratic interpolation of intensities should be able to capture specular reflections with fewer subdivisions because of its higher-degree polynomial. However, remember that we are fitting an intensity surface to a region that behaves as a power cosine function; the power typically in the range of 1 to 200 [FOLE82]. A quadratic surface is not much better than a linear surface at fitting a high-degree surface. Thus to achieve the same degree of realism in rendering a specular highlight, quadratic interpolation has to subdivide about as much as linear. Therefore, linear interpolation of intensities has the clear advantage in the realism vs. cost trade-off for specular reflection.

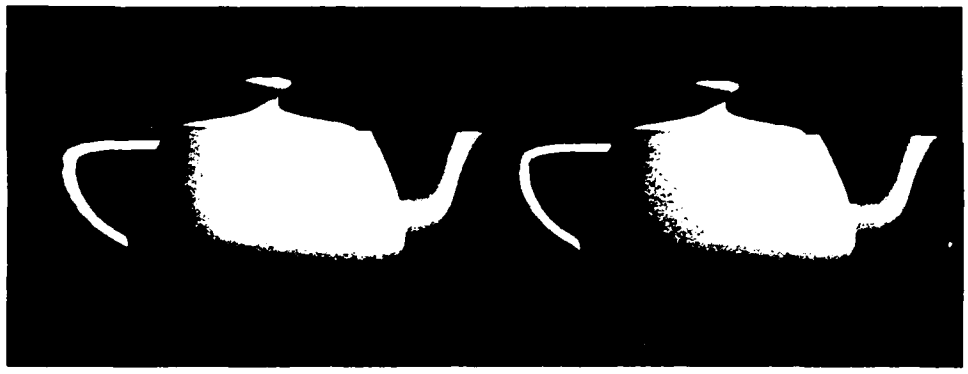
The number of subdivisions at the equal-cost point is actually much greater due to adaptability. Since specular highlights are rare, most facets do not require subdivision; more effort can be applied to those that do. The following timing study with images of the teapot demonstrates this point.

Figure 51a shows an image of a teapot with 944 facets shaded by linearly interpolating surface normals. The teapot of Figure 51b was shaded by linearly interpolating intensities; it started with 944 facets and adaptively subdivided based on the specular component to yield 1287 facets. Their CPU run times are 31.44 and 13.54 seconds, respectively. The images are comparable in terms of realism (the highlight on the body of the second teapot is actually more faithful due to use of $\vec{R} \cdot \vec{V}$ in Equation 7 vs. $\vec{H} \cdot \vec{N}$ in Equation 8). However, linear interpolation of intensities with adaptive subdivision based on the specular component far outdistances linear interpolation of surface normals in terms of speed.

6.4 Ambient Cut-Off vs. Locating Diffuse Boundaries

Both ambient cut-off and locating diffuse boundaries have been proposed as techniques to capture the C^0 boundaries at which the diffuse component becomes zero. Since ambient cut-off is only viable in the presence of one light source, we will compare these two techniques under this constraint.

While both techniques do a creditable job in capturing diffuse boundaries, different computational costs are associated with each. As derived in Section 5.2.1, ambient cut-off has no additional start-up costs but requires one additional comparison (costing one unit) at each pixel. Locating diffuse boundaries, on the other hand, incurs higher start-up



(a) (b)

Figure 51. Linear Interpolation of Surface Normals vs. Adaptive Subdivision

The first teapot comprising 944 facets was shaded by linearly interpolating surface normals. The second teapot was shaded by linearly interpolating intensities; it started with 944 facets and adaptively subdivided based on the specular component to yield 1287 facets. The images are comparable in terms of realism, even though the run time of the latter is much faster.

costs but has no additional iterative costs; the intersections of the diffuse boundary and the facet edges must be located and triangles reformed for the newly added points.

Figure 52a shows an image of a teapot whose diffuse boundaries were captured by ambient cut-off. Points were added along the diffuse boundaries of the intensity surfaces of Figure 52b. The diffuse boundaries in both images are well-defined. In addition, their CPU run times are comparable (6.66 vs. 6.94 seconds, respectively). While ambient cut-off is simpler to implement, it is not as general a technique as locating the diffuse boundary.

6.5 Software vs. Hardware Implementation

The Pixel-planes project [FUCH81] took a radically different approach to computer architecture design. Pixel-planes is organized into a binary tree of 64 one-bit arithmetic logic units and two multiplier trees. Linear expressions of the form, $Ax+By+C$, can be

(a) (b)

Figure 52. Ambient Cut-Off vs. Locating the Diffuse Boundary

Ambient cut-off was used to capture the diffuse boundaries in the first teapot. Points were added along the diffuse boundaries of the second teapot. The diffuse boundaries in both images are well-defined and their run times are comparable.

computed in parallel. Thus, coordinate transformations, clipping, projections, and shading can be computed for all pixels in one pass.

Goldfeather and Fuchs [GOLD86] recently designed a VLSI-based system for rapidly rendering quadratic surfaces. The system, called Pixel-powers, is a direct extension of Pixel-Planes, a linear expression evaluator described above. Pixel-powers is capable of evaluating quadratic expressions for every pixel in parallel. Their goal was to generate images of curved surfaces represented by quadratic expressions in real time.

The Pixel-powers system can be innovatively used to implement the quadratic shading algorithm. Though Pixel-powers was designed to render geometric surfaces, its quadratic expression evaluator (QEE) can be used to generate intensity surfaces as well. However, the shading algorithm must be re-expressed in a form suitable for the QEE.

The shading algorithm employs the Cendes-Wong formulas to generate a Bernstein-Bezier surface in barycentric coordinates. However, the QEE requires a quadratic

expression of the form:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F \tag{98}$$

The shading algorithm can be altered to fit this form by solving a 12-by-12 system of linear equations. The six coefficients, A, B, C, D, E, and F, are obtained for each C^1 piecewise quadratic intensity surface facet, as derived in Section 4.3.2. Although the computational costs of this solution are higher than the initialization costs for Cendes-Wong, the coefficients can then be fed to the QEE to evaluate the shade of each pixel in parallel.

Goldfeather and Fuchs claim that Pixel-powers is capable of rendering 30,000 curved polygonal facets per second. Based on this analysis, such claims can now be extended to curved polygonal facets that are shaded with a C^1 quadratic shading technique.

6.6 Summary

Table 10 summarizes the results of the trade-off study between the original triangulation (OT) and adaptive subdivision (AS). Table 11 summarizes the results of the trade-off study between computational costs and visual realism.

Table 10. Original Triangulation vs. Adaptive Subdivision Trade-off

	LII	QII	LISN
Object surface curvature			OT
Intensity surface curvature	OT		
Specular component	AS	AS	
Diffuse boundary	AS	AS	

The more cost-effective method, original triangulation (OT) or adaptive subdivision (AS), appears for each tool for each shading algorithm (LII = Linear Interpolation of Intensities, QII = Quadratic Interpolation of Intensities, LISN = Linear Interpolation of Surface Normals). A blank cell means that the tool does not apply for the shading algorithm.

Table 11. Computational Cost vs. Visual Realism Trade-off

	LII		QII		LISN	
	cost	**realism**	**cost**	**realism**	**cost**	**realism**
Fine polygon mesh	M	M	H	M	VH	H
Coarse polygon mesh	L	L	L	M	H	L
Diffuse only	L	M	M	H	VH	H
Diffuse and specular	M	H	H	H	VH	H

Computational costs and visual realism are tabulated qualitatively for a set of application-defined requirements for each shading algorithm (LII = Linear Interpolation of Intensities, QII = Quadratic Interpolation of Intensities, LISN = Linear Interpolation of Surface Normals). The scale of measures is L = low, M= medium, H = high, and VH = very high.

7. CONCLUSIONS

We have presented a toolbox for shading three-dimensional objects on a raster-scan display whose surfaces are represented as polygon meshes. For each shading technique, theoretical and mathematical foundations were documented. Algorithms were developed and analyzed in terms of theoretical and empirical computational costs. Trade-offs in realism requirements, computational costs, lighting and surface characteristics, image complexity, and perception were investigated. Timing analyses accompanied the discussion to enable the reader to identify which shading tools are most appropriate for his or her application and hardware environment.

The toolbox comprises three basic shading methods — linear interpolation of intensities, linear interpolation of surface normals, and quadratic interpolation of intensities. The first shading algorithm suffers from the Mach band effect, while the second has a heavy computational load. The third algorithm, newly proposed in this paper, takes an innovative approach to shading by applying a known geometric surface-fitting technique. A smoother intensity surface is fit through scattered intensity data using piecewise quadratic polynomials. The C^1 intensity surface reduces Mach banding and can be calculated in just two additions per pixel using forward differencing.

In addition, several tools have been developed to handle specific deficiencies in the above methods. Two tools improve surface continuity — adaptive subdivision of the object's surface and adaptive subdivision of the intensity surface. The first technique reduces approximation error when linearly interpolating surface normals via Taylor series. The latter method is an intensity-space extension to the former geometric technique to reduce Mach banding when linearly interpolating intensities.

When interpolating in intensity space, it was not heretofore possible to realistically capture specular reflections. A combination of techniques was introduced to locate specular highlights and adaptively subdivide based on the specular component. These techniques provide realistic renderings of specular reflections when quadratically or linearly interpolating intensities.

In images rendered with an existing shading algorithm, the diffuse boundaries, i.e., the perceptual boundaries where the diffuse component becomes zero, were typically misplaced. The magnitude of the error was dependent upon the coarseness of the polygon mesh. Two tools were introduced to give sharp, properly placed diffuse boundaries. The intersections between the diffuse boundaries and the facet edges are located and new vertices added. Or, in the presence of only one light source, an extra comparison is performed at each pixel to clamp intensities to the ambient light level.

7.1 Future Work

Certain areas of research presented in this paper can be more fully explored. Eight such areas include (1) specular reflection, (2) color, (3) intensity, (4) non-point light sources, (5) adaptive subdivision to reduce Mach banding, (6) dynamic images, (7) hardware implementation, and (8) images of other objects.

7.1.1 Specular Reflection

Phong's model for specular reflection, which is based on empirical data, does a creditable job in capturing highlights. A more sophisticated model based on physical studies was devised by Torrance and Sparrow [TORR66, TORR67]. A surface is assumed to

comprise microscopic facets, each a perfect reflector. The collective distribution of their orientation around the surface normal determines the reflective characteristics of the surface.

When linearly or quadratically interpolating intensities, the Torrance-Sparrow model can easily be incorporated into our toolbox for shading. Intensities are calculated using this model at the vertices of the polygon mesh. These more accurate intensities are then interpolated across the polygon facets. The added computational costs are nominal since it is incurred only at start-up. On the other hand, when linearly interpolating surface normals, the added expense is incurred at each pixel.

7.1.2 Color of Specular Reflection

Cook and Torrance [COOK81] devised a model based on geometrical optics that accounts for the spectral energy distribution of light. The color of the specular reflection is a function of the colors of the light source and the object's surface. This enhancement to the lighting model can be easily incorporated into our toolbox. An appropriate triple of specular reflectance coefficients can be chosen for the three primary colors based upon the colors of the light source and the object's surface.

Cook and Torrance [COOK81] further apply the Fresnel equation to express the color of the specular reflection as a function of the angle of incidence. As the angle of incidence decreases, the color of the object's surface has greater influence on the color of the specular reflection. When linearly or quadratically interpolating intensities, this effect can be realized by calculating intensities at the vertices of the polygon mesh based the angle of incidence. These more accurate intensities are then interpolated across the

polygon facets. This model can also be applied for linear interpolation of surface normals with added computational expense at each pixel.

7.1.3 Intensity and Color

The cost comparisons, timing studies, and realism trade-offs were all based on monochrome images. It was stated that the analyses of monochrome images resulted in no loss of generality since intensity equations for each of the three subtractive primary colors can be computed separately. This assumes that a tristimulus model of human vision can be applied to the interaction of light with objects [FOLE82]. While this is easy to implement, a more accurate model may be desired to generate more realistic images for a given application.

Instead of only three color components, Hall [HALL83] devised a lighting model which samples ten wavelengths in the color spectrum. Hall incorporates wavelength-dependent Fresnel reflectivity terms to accurately scale the intensity contribution at each wavelength. In addition to reflected light, Hall's mathematical model for intensity includes terms for transmitted light, as well as the transmittance through the medium between the surface and viewer. As before, these more accurate intensities can be calculated once upon start-up when interpolating intensities, or iteratively at each pixel when interpolating surface normals.

7.1.4 Non-Point Light Sources

In this paper, we modeled light sources as idealized points that radiate equally in all directions. Such light sources, while simplifying the mathematics, cast sharp shadows. A

real light source has a non-zero cross-sectional area and may radiate directionally. Such distributed light sources cast soft shadows with umbras and penumbras. A more accurate lighting model may be desired to generate more realistic images for a given application.

Warn [WARN83] devised a lighting model to capture the size and directionality of light sources. Light from a point light source is specularly reflected off of an imaginary reflector. The size, angle, and specular reflectance properties of the reflector model the characteristics of the distributed light source. The apparent intensity of the light source striking the surface of an object is easily computed. The modified intensity of reflected light is then computed by our Equation 7 as before. This enhancement can be easily incorporated for linear and quadratic interpolation of intensity.

7.1.5 Adaptive Subdivision to Reduce Mach Banding

One technique was proposed in this paper to reduce Mach banding by adaptively subdividing when linearly interpolating intensities. This technique lost the realism-cost trade-off to a finer original triangulation for two reasons: the basic shading algorithm of linear interpolation of intensities is very efficient, and the criterion for adaptive subdivision was not particularly effective. The potential for improvement lies in the second point.

The criterion for adaptive subdivision was based on intensity surface curvature for two reasons. First, psychophysical experiments showed a relation between the perceived magnitude of Mach bands and intensity surface curvature. Second, the technique was a direct extension of adaptive subdivision based on object surface curvature. However, since this criterion proved lacking, other criteria should be explored to efficiently reduce

Mach banding by adaptive subdivision.

7.1.6 Dynamic Images

Several assertions were made concerning dynamic images which were not supported through implementation. Quadratic interpolation of intensities may generally improve shading consistency from one frame to the next, while adversely affecting dynamic shading near silhouette edges (Section 4.7). The costs of locating specular highlights and adaptively subdividing based on the specular component can be greatly reduced when dynamically rendering a sequence of images by taking advantage of frame-to-frame coherence (Section 5.1). Similarly, the costs of locating diffuse boundaries can be greatly reduced by taking advantage of frame-to-frame coherence (Section 5.2). These results should be examined via implementation on computer hardware with real-time graphics capability.

7.1.7 Hardware Implementation

The potential exists for implementing parts of this shading toolbox in hardware. The basic Phong and Gouraud shading algorithms have been implemented in hardware extensively. Custom hardware can be built to implement any other part of the toolbox. Alternatively, the existing Pixel-powers system can be put to the task of parallelizing quadratic interpolation of intensities.

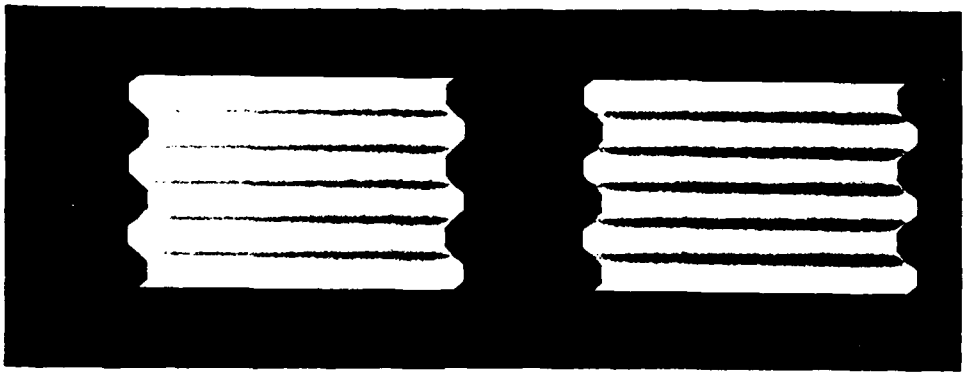
7.1.8 Images of Other Objects

Throughout this paper, an image of a teapot was rendered to compare and analyze realism and computational costs. Because of its varied surface curvature, it has proven

to be a good image for our study, and we hope representative enough to generalize our conclusions. To validate our results, realism and cost trade-offs should be analyzed for a variety of images in different applications on several computers.

We briefly investigated another class of object, a corrugated roof. The surface of the roof differs from the teapot by exhibiting substantial concavity and regularity. Surface normals, as well as intensity surface normals, at adjacent vertices may bend towards one another or be coincident. This impacts most of the algorithms in the shading toolbox because adaptive subdivision is based on surface normals.

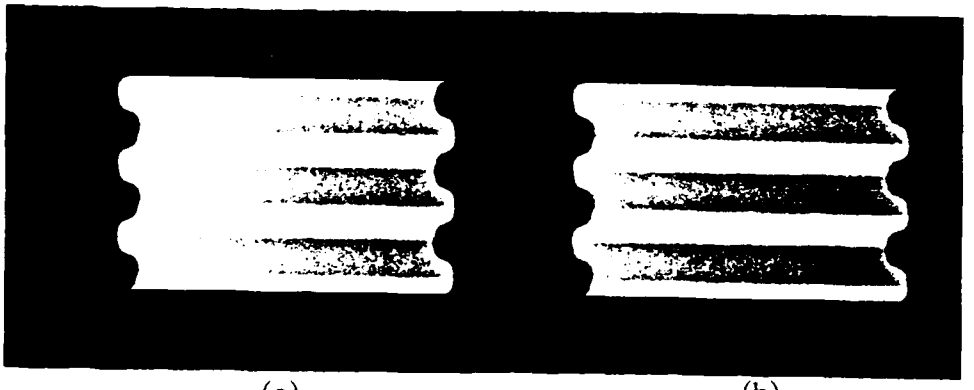
The shading algorithms took the topology of this new object in stride. Figures 53a and b present two images of the corrugated roof, shaded by linearly and quadratically interpolating intensities, respectively. Techniques to locate highlights and adaptively subdivide based on the specular component were employed. Note the linear highlights which span the roof's entire length. Figure 54 shows the roof at a somewhat higher resolution and in different lighting.



(a) (b)

Figure 53. Corrugated Roof 1

The first image is shaded by linearly interpolating intensities and the second by quadratically interpolating intensities. Techniques to locate highlights and adaptively subdivide based on the specular component capture the linear highlights which span the roof's entire length.



(a) (b)

Figure 54. Corrugated Roof 2

The first image is shaded by linearly interpolating intensities and the second by quadratically interpolating intensities. These images are at a somewhat higher resolution and lighted differently than Figure 53.

8. REFERENCES

[ARMI78] Armington, J. C., J. Krauskopf, and B. R. Wooten, editors, *Visual Psychophysics and Physiology*, Academic Press, New York, 1978.

[BARN73] Barnhill, R. E., G. Birkhoff, and W. J. Gordon, "Smooth Interpolation in Triangles", *Journal of Approximation Theory*, Vol. 8, 1973, pp. 114-128.

[BARN77] Barnhill, R. E., "Representation and Approximation of Surfaces", *Mathematical Software III* (J. R. Rice, ed.), Academic Press, New York, 1977.

[BART87] Bartels, R. H., J. C. Beatty and B. A. Barsky, *Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, Los Altos, California, 1987.

[BISH86] Bishop, G. and D. M. Weimer, "Fast Phong Shading", *Computer Graphics* (Proceedings SIGGRAPH '86), Vol. 20, No. 4, August 1986, pp. 103-106.

[BLIN86] Blinn, J. F., "A Scan Line Algorithm for Displaying Parametrically Defined Surfaces", *Image Rendering Tricks*, Association for Computing Machinery SIGGRAPH '86, Course Notes #20, August 1986.

[BLIN87] Blinn, J. F., "What, Teapots Again?", *Computer Graphics and Applications*, Institute of Electrical and Electronics Engineers, Vol. 7, No. 9, September 1987, pp. 61-63.

[BOEH87] Boehm, W., "A Survey of Curve and Surface Methods in CAGD", *The Geometry of Curve and Surface Design*, Association for Computing Machinery SIGGRAPH '87 Course #20, July 1987.

[BUI75] Bui Tuong Phong, "Illumination for Computer Generated Pictures", *Communications of the ACM*, Vol. 18, No. 6, June 1975, pp. 311-317.

[BUI75a] Bui Tuong Phong and F. C. Crow, "Improved Rendition of Polygonal Models of Curved Surfaces", *Proceedings of the Second USA-Japan Computer Conference*, 1975.

[CATM75] Catmull, E. E., "Computer Display of Curved Surfaces", *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, Institute of Electrical and Electronics Engineers, May 1975, pp. 309-315.

[CEND87] Cendes, Z. J. and S. H. Wong, "C^1 Quadratic Interpolation over Arbitrary Point Sets", *Computer Graphics and Applications*, Institute of Electrical and Electronics Engineers, Vol. 7, No. 11, November 1987, pp. 8-16.

[CLAR86] Clark, J. H., "A Fast Algorithm for Rendering Parametric Surfaces", *Image Rendering Tricks*, Association for Computing Machinery SIGGRAPH '86, Course Notes #20, August 1986.

[COOK81] Cook, R. L. and K. E. Torrance, "A Reflectance Model for Computer Graphics", *Computer Graphics* (Proceedings SIGGRAPH '81), Vol. 15, No. 3, August 1981, pp. 307-316.

[COON67] Coons, S. A., "Surfaces for Computer-Aided Design of Space Forms", Technical Report MAC-TR-41, Massachusetts Institute of Technology, Cambridge, MA., 1967.

[CROW87] Crow, F., "The Origins of the Teapot", *Computer Graphics and Applications*, Institute of Electrical and Electronics Engineers, Vol. 7, No. 1, January 1987, pp. 8-19.

[DUFF79] Duff, T., "Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays", *Computer Graphics* (Proceedings SIGGRAPH '79), Vol. 13, No. 2, Aug 1979, pp. 270-275.

[FARI83] Farin, G. E., "Smooth Interpolation to Scattered 3D Data", *Surfaces in Computer Aided Geometric Design* (R. E. Barnhill and W. Boehm, eds.), North-Holland, Amsterdam, 1983.

[FOLE82] Foley, James D. and Andries van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, Reading, MA, 1982.

[FUCH81] Fuchs, H. and J. Poulton, "Pixel-planes: A VLSI-Oriented Design for a Raster Graphics Engine", *VLSI Design*, Vol.2 No.3, 3rd Quarter, 1981, pp.20-28.

[GOLD86] Goldfeather, J. and H. Fuchs, "Quadratic Surface Rendering on a Logic-Enhanced Frame-Buffer Memory", *Computer Graphics and Applications*, Institute of Electrical and Electronics Engineers, Vol. 6 No. 1, January 1986, pp.48-59.

[GORD69] Gordon, W. J., "Distributive Lattices and the Approximation of Multivariate Functions", *Proceedings of the Symposium on Approximation with Special Emphasis on Splines* (I. J. Schoenberg, ed.), Univ. of Wisconsin Press, Madison, Wisconsin, 1969.

[GOUR71] Gouraud, H., "Continuous Shading of Curved Surfaces", *IEEE Transactions on Computers*, Vol. C-20, No. 6, June 1971, pp. 623-628.

[GRAN86] Grandine, T. A., "An Iterative Method for Computing Bivariate C^1 Piecewise Cubic Polynomial Interpolants", MRC Technical Summary Report #2937, University of Wisconsin, June 1986.

[HALL83] Hall, R. A. and D. P. Greenberg, "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics and Applications*, Vol. 3, No. 8, August 1983, pp. 10-20.

[HECK86] Heckbert, P. S., "Survey of Texture Mapping", *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, November 1986, pp. 56-67.

[LANE80] Lane, J. M., L. C. Carpenter, T. Whitted, J. F. Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces", *Communications of the ACM*, Vol. 23, No. 1, January 1980, pp. 23-34.

[LAWS77] Lawson, C. L., "Software for C^1 Surface Interpolation", *Mathematical Software III* (J. R. Rice, ed.), Academic Press, New York, 1977.

[LITT83] Little, F., "Convex Combination Surfaces", *Surfaces in Computer Aided Geometric Design* (R. E. Barnhill and W. Boehm, eds.), North-Holland, Amsterdam, 1983.

[NIEL80] Nielson, G. M., "Minimum Norm Interpolation in Triangles", *SIAM Journal of Numerical Analysis*, Vol. 17, 1980, pp. 44-62.

[NIEL83] Nielson, G. M. and R. Franke, "Surface Construction based upon Triangulations", *Surfaces in Computer Aided Geometric Design* (R. E. Barnhill and W. Boehm, eds.), North-Holland, Amsterdam, 1983.

[NEWM79] Newman, W. M. and R. F. Sproull, *Principles of Interactive Computer Graphics*, 2nd Ed., McGraw-Hill Book Company, N. Y., 1979.

[POWE77] Powell, M. J. D. and M. A. Sabin, "Piecewise Quadratic Approximations on Triangles", *ACM Transactions on Mathematical Software*, Vol. 3, December 1977, pp. 316-325.

[RATL65] Ratliff, F., *Mach Bands: Quantitative Studies on Neural Networks in the Retina*, Holden-Day, San Francisco, 1965.

[SCHM82] Schmidt, R., "Eine Methode zur Konstruktion von C^1-Flachen zur Interpolation unregelmassig verteiler Daten", *Multivariate Approximation II* (Schempp and Zeller, eds.), Birkhauser, Basel, 1982, pp. 343-361.

[SHEP65] Shephard, D., "A Two-Dimensional Interpolation Function for Irregularly Spaced Data", *Proceedings of the ACM National Conference*, 1965, pp. 517-524.

[SIBS81] Sibson, R. and G. Thomson, "A Seamed Quadratic Element for Contouring", *The Computer Journal*, November 1981, pp. 378-382.

[STRA73] Strang, G. and G. Fix, *An Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[SUN87] "Sun Floating Point Accelerator", Sales Brochure, Sun Microsystems, Inc., Mountain View, CA., 1987.

[TORR66] Torrance, K. E. and E. M. Sparrow, "Polarization, Direction Distribution, and Off-Specular Peak Phenomena in Light Reflected from Roughened Surfaces", *Journal of the Optical Society of America*, Vol. 56, No. 7, July 1966, pp. 916-925.

[TORR67] Torrance, K. E. and E. M. Sparrow, "Theory for Off-Specular Reflection from Roughened Surfaces", *Journal of the Optical Society of America*, Vol. 57, No. 9, September 1967, pp. 1105-1114.

[WARN83] Warn, D. R., "Lighting Controls for Synthetic Images", *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 13-21.

RELATED REFERENCES

[ARMI78] Armington, J. C., J. Krauskopf, and B. R. Wooten, editors, *Visual Psychophysics and Physiology*, Academic Press, New York, 1978.

[BARN74] Barnhill, R. E. and R. F. Riesenfeld, *Computer Aided Geometric Design*, Academic Press, New York, 1974.

[BART86] Bartels, R. H., J. C. Beatty and B. A. Barsky, *Introduction to the Use of Splines in Freeform Curve and Surface Design*, Association for Computing Machinery SIGGRAPH '86 Course #4, University of Waterloo and University of California, 1986.

[BASI68] *Basic Principles of Sensory Evaluation,* American Society for Testing and Materials, Philadelphia, PA, 1968.

[BEKE67] Bekesy, G. von, *Sensory Inhibition*, Princeton University Press, New Jersey, 1967.

[BEKE68] Bekesy, G. von, "Brightness Distribution Across Mach Bands Measured with Flicker Photometry and the Linearity of Sensory Nervous Interaction", *Journal of the Optical Society of America*, Vol. 58, 1968, pp. 1-8.

[BROD80] Brodlie, K. W., *Mathematical Methods in Computer Graphics and Design*, Academic Press, London, 1980.

[CROW77] Crow, F. C., "Shadow Algorithm for Computer Graphics", *Computer Graphics* (Proceedings SIGGRAPH '77), Vol. 11, No. 2, July 1977, pp.

242-248.

[FAUX80] Faux, I. D. and M. J. Pratt, *Computational Geometry for Design and Manufacture*, Ellis Horwood Ltd., Chichester, England, 1980.

[FECH66] Fechner, G., *Elements of Psychphysics*, Holt, Rinehart and Winston, New York, 1966.

[FIOR72] Fiorentini, "Mach Band Phenomena", in *Handbook of Sensory Physiology*, Vol. 7, No. 4: Visual Psychophysics (Jameson and Hurvich, eds.) Springer-Verlag, Chap. 8, 1972, pp 188-203.

[FORS77] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, N.J., 1977.

[FRY48] Fry, G. A., "Mechanism subserving simultaneous brightness contrast", *American Journal of Optometry and Archives of American Academy of Optometry*, Vol. 25, 1948, pp. 162-178.

[GREG86] Gregory, J. A. (editor), *The Mathematics of Surface*, Clarendon Press, Oxford, England, 1986.

[HALL89] Hall, R., *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, 1989.

[LOWR61] Lowry, E. M. and J. J. DePalma, "Sine-wave Response of the Visual System, I. The Mach Band Phenomenon", *Journal of the Optical Society of America*, Vol. 51, 1961, pp. 740-746.

[MAX88] Max, N., "Smooth Appearance for Polygonal Surfaces", Technical Report, Lawrence Livermore National Laboratory and the University of California at Davis, 1988.

[MCCO55] McCollough, C. "The Variation in Width and Position of Mach Bands as a Function of Luminance", *Journal of Experimental Psychology*, Vol. 49, 1955, pp. 141-152.

[PAVL82] Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.

[RATL74] Ratliff, F. and H. K. Hartline, *Studies on Excitation and Inhibition in the Retina*, The Rockefeller University Press, New York, 1974.

[ROBI72] Robinson, J. O., *The Psychology of Visual Illusion*, Hutchinson & Co., London, 1972.

APPENDIX A - CENDES-WONG FORMULAS

Appendix A supplies the set of formulas established by Cendes and Wong [CEND87] to perform quadratic C^1 interpolation over triangular patches. Figure 55 ascribes a number to each of the nineteen Bezier control points.

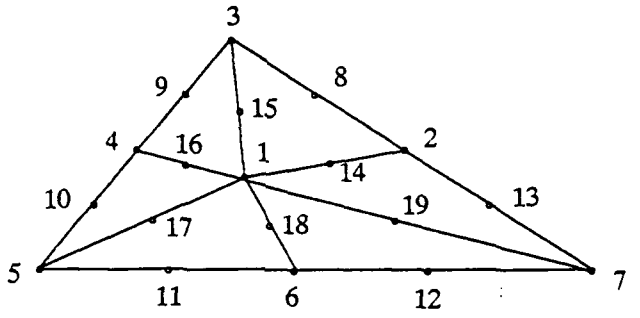


Figure 55. *Cendes-Wong Quadratic Interpolation*

Nineteen bezier control points for quadratic interpolation with six subtriangles.

Table 12 gives the formulas for the control points b. Let (r,s,t) be the barycentric coordinates of the triangle incenter at node 1. Let α, β, and γ represent distance ratios as follows:

$$\alpha = \frac{d_{7,6}}{d_{7,5}} \qquad \beta = \frac{d_{3,2}}{d_{3,7}} \qquad \gamma = \frac{d_{5,4}}{d_{5,3}}$$

where $d_{i,j}$ is the distance between nodes i and j. Coefficients c_1 through c_9 are defined as follows:

$$c_1 = \frac{1}{2}(x_1 - x_3)$$

$$c_2 = \frac{1}{2}(x_1 - x_5)$$

$$c_3 = \frac{1}{2}(x_1 - x_7)$$

$$c_4 = \frac{\beta}{2}(x_7 - x_3)$$

$$c_5 = \frac{1-\gamma}{2}(x_5 - x_3)$$

$$c_6 = \frac{\gamma}{2}(x_3 - x_5)$$

$$c_7 = \frac{1-\alpha}{2}(x_7 - x_5)$$

$$c_8 = \frac{\alpha}{2}(x_5 - x_7)$$

$$c_9 = \frac{1-\beta}{2}(x_3 - x_7)$$

The d coefficients are defined similarly by replacing y for x in the above equations.

Table 12. Cendes-Wong Formulas

	I_3	$\partial I_3 / \partial x$	$\partial I_3 / \partial y$	I_5	$\partial I_5 / \partial x$	$\partial I_5 / \partial y$
b_1	r	rc_1	rd_1	s	sc_2	sd_2
b_2	$(1-\beta)$	$(1-\beta)c_4$	$(1-\beta)d_4$			
b_3	1					
b_4	γ	γc_5	γd_5	$(1-\gamma)$	$(1-\gamma)c_6$	$(1-\gamma)d_6$
b_5				1		
b_6				α	αc_7	αd_7
b_7						
b_8	1	c_4	d_4			
b_9	1	c_5	d_5			
b_{10}				1	c_6	d_6
b_{11}				1	c_7	d_7
b_{12}						
b_{13}						
b_{14}	$(1-\beta)$	$(1-\beta)c_1$	$(1-\beta)d_1$			
b_{15}	1	c_1	d_1			
b_{16}	γ	γc_1	γd_1	$(1-\gamma)$	$(1-\gamma)c_2$	$(1-\gamma)d_2$
b_{17}				1	c_2	d_2
b_{18}				α	αc_2	αd_2
b_{19}						

Table 12 (Continued)

	I_7	$\partial I_7 / \partial x$	$\partial I_7 / \partial y$
b_1	t	tc_3	td_3
b_2	β	βc_9	βd_9
b_3			
b_4			
b_5			
b_6	$(1-\alpha)$	$(1-\alpha)c_8$	$(1-\alpha)d_8$
b_7	1		
b_8			
b_9			
b_{10}			
b_{11}			
b_{12}	1	c_8	d_8
b_{13}	1	c_9	d_9
b_{14}	β	βc_3	βd_3
b_{15}			
b_{16}			
b_{17}			
b_{18}	$(1-\alpha)$	$(1-\alpha)c_3$	$(1-\alpha)d_3$
b_{19}	1	c_3	d_3

APPENDIX B - IMPLEMENTING THE SHADING TOOLBOX

Appendix B supplies the C source code implementation of the shading toolbox. Source code is given for the three basic shading algorithms:

- linear interpolation of intensities
- linear interpolation of surface normals
- quadratic interpolation of intensities

In addition, source code is given for six tools to enhance shading:

- adaptive subdivision of the object's surface to reduce Taylor series approximation error
- adaptive subdivision of the intensity surface to reduce Mach banding
- locating specular highlights
- adaptive subdivision based on the specular component
- locating diffuse boundary and facet edge crossings
- ambient cut-off for diffuse boundaries

Finally, the program and data to generate the four parts of the teapot — body, lid, spout, and handle — are supplied.

Main Routine

```
/****************************** Globals ******************************/

#include <pixrect/pixrect_hs.h>
#include <stdio.h>
#include <ctype.h>
#include <math.h>

struct pixrect *screen, *region;
double xnormal[1000],ynormal[1000],znormal[1000],xworld[1000],yworld[1000],
    intens[1000],specular[1000],xtang[1000],ytang[1000],xlight,ylight,zlight,
    xeye,yeye,zeye,xhigh,yhigh,zhigh,x[23],y[23],z[23],
    ambient,kd,ks,spec_power,max_dist,max_spec,
    xmid[1000][40],ymid[1000][40],xincenter[1000][40],yincenter[1000][40],
    qq1[1000][40],qq2[1000][40],qq3[1000][40];
int  adj[1000][40],nvertex[1000],insert_list[1000][3],connect_list[1000][5],
    nconnect[1000],boundary[1000],iamb,npoints;


/****************************** Main ******************************/

#include "globals.h"

main (argc, argv)
 char **argv;
 int  argc;
{
 unsigned char  shade[256];
 FILE       *stream;
 int        i;

 read_light_data ();

 /* open frame buffer */

 screen = pr_open ("/dev/fb");
 region = pr_region (screen, 0, 0, 567, 270);

 /* set color map to gray scale */

 for (i=0; i<256; i++) {
  shade[i] = (unsigned char) i;
 }
 prs_putcolormap (region, 0, 255, shade, shade, shade);

 /* shade individual teapot part only */

 if (argc > 1 && strcmp(argv[1], "w") != 0 && strcmp(argv[1], "d") != 0) {
```

```
 if (strcmp(argv[1], "l") == 0) {
  read_object_data ("lid.data");
  calculate_shades (argc, argv);
 }
 else if (strcmp(argv[1], "s") == 0) {
  read_object_data ("spout.data");
  calculate_shades (argc, argv);
 }
 else if (strcmp(argv[1], "b") == 0) {
  read_object_data ("body.data");
  calculate_shades (argc, argv);
 }
 else if (strcmp(argv[1], "h") == 0) {
  read_object_data ("handle.data");
  calculate_shades (argc, argv);
 }
}

/* shade entire teapot */

else {
 read_object_data ("lid.data");
 calculate_shades (argc, argv);

 read_object_data ("spout.data");
 calculate_shades (argc, argv);

 read_object_data ("body.data");
 calculate_shades (argc, argv);

 read_object_data ("handle.data");
 calculate_shades (argc, argv);
}

/* save to raster file */

if ((argc > 1 && strcmp(argv[1], "d") == 0) ||
   (argc > 2 && strcmp(argv[2], "d") == 0)) {
 stream = fopen ("new.pixrect", "w");
 pr_dump (region, stream, NULL, 2, 1);
 }
}


/****************************** Calculate Shades ****************************/

calculate_shades (argc, argv)
```

Main Routine

```
 char **argv;
 int   argc;
{
 int    j, p1, p2, p3, nvert;

 /* locate specular highlights */

 if (ks > 0.0) locate_specular ();

 /* calculate intensity for each vertex */

 for (p1=0; p1<npoints; p1++) {
  calc_intensity (p1);
 }

 /* adaptively subdivide for specular component */

 if (ks > 0.0 && max_dist < 99999.) subdivide_specular ();

 /* calculate tangents to intensity surface */

 for (p1=0; p1<npoints; p1++) {
  calc_slope (p1);
 }

 /* create wire frame image */

 if (argc > 1 && strcmp(argv[1], "w") == 0) {
  for (p1=0; p1<npoints; p1++) {
   for (j=0; j<nvertex[p1]; j++) {
    p2 = adj[p1][j];
    if (p2 > p1) {
     pr_vector (region, (int)xworld[p1], (int)yworld[p1],
           (int)xworld[p2], (int)yworld[p2], 0, 0);
    }
   }
  }
 }

 /* shade teapot */

 else {

  /* find incenter for each facet */

  for (p1=0; p1<npoints; p1++) {
   nvert = adjacent(p1);
```

```
    for (j=0; j<nvert; j++) {
      p2 = adj[p1][j];
      p3 = adj[p1][(j+1) % nvertex[p1]];
      if (p2 > p1 && p3 > p1) {
        incenter (p1, p2, p3, j);
      }
    }
  }

  /* shade each facet */

  for (p1=0; p1<npoints; p1++) {
    nvert = adjacent(p1);
    for (j=0; j<nvert; j++) {
      p2 = adj[p1][j];
      p3 = adj[p1][(j+1) % nvertex[p1]];
      if (p2 > p1 && p3 > p1) {
        triangle (p1, p2, p3, j);
      }
    }
  }
 }
}
```

Linear Interpolation of Intensities

```
#include "globals.h"

triangle (p1,p2,p3)
  int p1,p2,p3;
{
  double       mid,yh_yl,xinter,intmax;
  int          shade,ylow,yhigh,other,ixmin,ixmax,ymin,ymax;
  register int   ix,iy;
  register double intensity,intensity00,intx1,inty1,xmin,xmax,
           dxmin,dxmax;

  /* find smallest and largest y coordinate */

  ylow = p1;
  if (yworld[p2]<yworld[p1]) ylow = p2;
  if (yworld[p3]<yworld[ylow]) ylow = p3;
  yhigh = p1;
  if (yworld[p2]>yworld[p1]) yhigh = p2;
  if (yworld[p3]>yworld[yhigh]) yhigh = p3;
  other = p1;
  if (ylow!=p2 && yhigh!=p2) other = p2;
  if (ylow!=p3 && yhigh!=p3) other = p3;

  yh_yl = 1. / (yworld[yhigh] - yworld[ylow]);

  /* calculate interpolation coefficients for forward differencing */

  intensity00 = intens[ylow];
  inty1 = (intens[yhigh] - intens[ylow]) * yh_yl;

  mid = (yworld[other] - yworld[ylow]) * yh_yl;
  xinter = xworld[ylow] + mid * (xworld[yhigh] - xworld[ylow]);
  intmax = intens[ylow] + mid * (intens[yhigh] - intens[ylow]);
  intx1 = (intens[other] - intmax) / fabs(xworld[other] - xinter);

  /* prepare for iteration from low y to mid y */

  ymin = (int)(yworld[ylow]);
  ymax = (int)(yworld[other]);

  xmin = xworld[ylow];
  xmax = xworld[ylow];

  dxmin  = (xworld[yhigh] - xworld[ylow]) * yh_yl;
  dxmax  = (xworld[other] - xworld[ylow] ) / (yworld[other] - yworld[ylow]);

  /* for each y */
```

```
for (iy=ymin+1; iy<=ymax; iy++) {
  intensity = intensity00;
  ixmin = (int)xmin;
  ixmax = (int)xmax;

  if (xmax>xmin) {

    /* for each x */

    for (ix=ixmin; ix<=ixmax; ix++) {
      shade = (int)(intensity);
      ambient_cutoff (&shade);

      pr_put(region,ix,iy,shade);

      /* increment for step in x */

      intensity = intensity + intx1;
    }
  }
  else {

    /* for each x */

    for (ix=ixmin; ix>=ixmax; ix--) {
      shade = (int)(intensity);
      ambient_cutoff (&shade);

      pr_put(region,ix,iy,shade);

      /* increment for step in x */

      intensity = intensity + intx1;
    }
  }

  /* increment for step in y */

  intensity00 = intensity00 + inty1;
  xmin = xmin + dxmin;
  xmax = xmax + dxmax;
}

/* prepare for iteration from mid y to high y */

ymin = (int)(yworld[other]);
ymax = (int)(yworld[yhigh]);
```

```
  xmax = xworld[other];
  dxmax = (xworld[yhigh] - xworld[other]) / (yworld[yhigh] - yworld[other]);

  for (iy=ymin+1; iy<=ymax; iy++) {
    intensity = intensity00;
    ixmin = (int)xmin;
    ixmax = (int)xmax;

    if (xmax>xmin) {

      /* for each x */

      for (ix=ixmin; ix<=ixmax; ix++) {
        shade = (int)(intensity);
        ambient_cutoff (&shade);

        pr_put(region,ix,iy,shade);

        /* increment for step in x */

        intensity = intensity + intx1;
      }
    }
    else {

      /* for each x */

      for (ix=ixmin; ix>=ixmax; ix--) {
        shade = (int)(intensity);
        ambient_cutoff (&shade);

        pr_put(region,ix,iy,shade);

        /* increment for step in x */

        intensity = intensity + intx1;
      }
    }

    /* increment for step in y */

    intensity00 = intensity00 + inty1;
    xmin = xmin + dxmin;
    xmax = xmax + dxmax;
  }
}
```

Linear Interpolation of Surface Normal

```
#include "globals.h"

triangle (p1, p2, p3)
 int   p1, p2, p3;
{
 double xs, ys, mid, xdel, ydel, xcenter, ycenter,
     T5, T4, T3, T2, T1, T0, S5, S4, S3, S2, S1, S0,
     Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz,
     a, b, c, d, e, f, g, h, i, x1, x2, x3, yy1, y2, y3,
     xydel, xh_xl, xh_xo, xo_xl, yh_yl, xmin, xmax, dxmin, dxmax;
 int   shade, ylow, yhigh, other, ixmin, ixmax, ymin, ymax;
 register int   ix, iy;
 register double diffuse, specular, dx1, dx2, dy1, dy2, sx1, sx2, sy1, sy2,
          diffuse0, dx10, dxy, specular0, sx10, sxy;

 /* find centroid of triangular facet */

 centroid(p1, p2, p3, &xcenter, &ycenter);
 x1 = xworld[p1] - xcenter;
 x2 = xworld[p2] - xcenter;
 x3 = xworld[p3] - xcenter;
 yy1 = yworld[p1] - ycenter;
 y2 = yworld[p2] - ycenter;
 y3 = yworld[p3] - ycenter;

  /* calculate interpolation coefficients for surface normal */

 if (fabs(y2-yy1) > .0001) {
  Ax = ((xnormal[p3] - xnormal[p1]) * (y2 - yy1) - (xnormal[p2] - xnormal[p1]) * (y3 - yy1)) /
   ((x3 - x1) * (y2 - yy1) - (x2 - x1) * (y3 - yy1));
  Bx = (xnormal[p2] - xnormal[p1] - Ax * (x2 - x1)) / (y2 - yy1);
  Cx = xnormal[p1] - Ax * x1 - Bx * yy1;

  Ay = ((ynormal[p3] - ynormal[p1]) * (y2 - yy1) - (ynormal[p2] - ynormal[p1]) * (y3 - yy1)) /
   ((x3 - x1) * (y2 - yy1) - (x2 - x1) * (y3 - yy1));
  By = (ynormal[p2] - ynormal[p1] - Ay * (x2 - x1)) / (y2 - yy1);
  Cy = ynormal[p1] - Ay * x1 - By * yy1;

  Az = ((znormal[p3] - znormal[p1]) * (y2 - yy1) - (znormal[p2] - znormal[p1]) * (y3 - yy1)) /
   ((x3 - x1) * (y2 - yy1) - (x2 - x1) * (y3 - yy1));
  Bz = (znormal[p2] - znormal[p1] - Az * (x2 - x1)) / (y2 - yy1);
  Cz = znormal[p1] - Az * x1 - Bz * yy1;
 }
 else {
  Ax = ((xnormal[p2] - xnormal[p1]) * (y3 - yy1) - (xnormal[p3] - xnormal[p1]) * (y2 - yy1)) /
   ((x2 - x1) * (y3 - yy1) - (x3 - x1) * (y2 - yy1));
  Bx = (xnormal[p3] - xnormal[p1] - Ax * (x3 - x1)) / (y3 - yy1);
```

```
  Cx = xnormal[p1] - Ax * x1 - Bx * yy1;

  Ay = ((ynormal[p2] - ynormal[p1]) * (y3 - yy1) - (ynormal[p3] - ynormal[p1]) * (y2 - yy1)) /
   ((x2 - x1) * (y3 - yy1) - (x3 - x1) * (y2 - yy1));
  By = (ynormal[p3] - ynormal[p1] - Ay * (x3 - x1)) / (y3 - yy1);
  Cy = ynormal[p1] - Ay * x1 - By * yy1;

  Az = ((znormal[p2] - znormal[p1]) * (y3 - yy1) - (znormal[p3] - znormal[p1]) * (y2 - yy1)) /
   ((x2 - x1) * (y3 - yy1) - (x3 - x1) * (y2 - yy1));
  Bz = (znormal[p3] - znormal[p1] - Az * (x3 - x1)) / (y3 - yy1);
  Cz = znormal[p1] - Az * x1 - Bz * yy1;
}

/* precalculate factors */

a = (xlight*Ax + ylight*Ay + zlight*Az);
b = (xlight*Bx + ylight*By + zlight*Bz);
c = (xlight*Cx + ylight*Cy + zlight*Cz);
d = Ax*Ax + Ay*Ay + Az*Az;
e = 2.*(Ax*Bx + Ay*By + Az*Bz);
f = Bx*Bx + By*By + Bz*Bz;
g = 2.*(Ax*Cx + Ay*Cy + Az*Cz);
h = 2.*(Bx*Cx + By*Cy + Bz*Cz);
i = Cx*Cx + Cy*Cy + Cz*Cz;

/* calculate diffuse component for forward differencing */

T5 = (3.*c*g*g - 4.*c*d*i - 4.*a*g*i) / (8.*i*i*sqrt(i));
T4 = (3.*c*g*h - 2.*c*e*i - 2.*b*g*i - 2.*a*h*i) / (4.*i*i*sqrt(i));
T3 = (3.*c*h*h - 4.*c*f*i - 4.*b*h*i) / (8.*i*i*sqrt(i));
T2 = (2.*a*i - c*g) / (2.*i*sqrt(i));
T1 = (2.*b*i - c*h) / (2.*i*sqrt(i));
T0 = c / sqrt(i);

/* precalculate factors */

a = (xeye*Ax + yeye*Ay + zeye*Az);
b = (xeye*Bx + yeye*By + zeye*Bz);
c = (xeye*Cx + yeye*Cy + zeye*Cz);

/* calculate specular component for forward differencing */

S5 = (3.*c*g*g - 4.*c*d*i - 4.*a*g*i) / (8.*i*i*sqrt(i));
S4 = (3.*c*g*h - 2.*c*e*i - 2.*b*g*i - 2.*a*h*i) / (4.*i*i*sqrt(i));
S3 = (3.*c*h*h - 4.*c*f*i - 4.*b*h*i) / (8.*i*i*sqrt(i));
S2 = (2.*a*i - c*g) / (2.*i*sqrt(i));
S1 = (2.*b*i - c*h) / (2.*i*sqrt(i));
```

```
S0 = c / sqrt(i);

/* find smallest and largest y coordinate */

ylow = p1;
if (yworld[p2]<yworld[p1]) ylow = p2;
if (yworld[p3]<yworld[ylow]) ylow = p3;
yhigh = p1;
if (yworld[p2]>yworld[p1]) yhigh = p2;
if (yworld[p3]>yworld[yhigh]) yhigh = p3;
other = p1;
if (ylow!=p2 && yhigh!=p2) other = p2;
if (ylow!=p3 && yhigh!=p3) other = p3;

yh_yl = 1. / (yworld[yhigh] - yworld[ylow]);

xh_xl = xworld[yhigh] - xworld[ylow];
xh_xo = xworld[yhigh] - xworld[other];
xo_xl = xworld[other] - xworld[ylow];

mid = (yworld[other]-yworld[ylow]) * yh_yl;
xs = xworld[ylow] + mid * xh_xl;
if (xs < xworld[other]) xdel = 1.0;
else xdel = -1.0;
ydel = 1.0;
xydel = xh_xl * yh_yl;

xs = xworld[ylow] - xcenter;
ys = yworld[ylow] - ycenter;

/* calculate zeroth, first, and second differences
  for diffuse component */

diffuse0 = T5*xs*xs + T4*xs*ys + T3*ys*ys + T2*xs + T1*ys + T0;
dx10 = T5*(2.*xs*xdel+xdel*xdel) + T4*ys*xdel + T2*xdel;
dx2 = T5*2.*xdel*xdel;
dy1 = T5*(2.*xs*xydel+xydel*xydel) + T4*(ys*xydel+xs*ydel+xydel*ydel) +
 T3*(2.*ys*ydel+ydel*ydel) + T2*xydel + T1*ydel;
dy2 = T5*2.*xydel*xydel + T4*2.*ydel*xydel + T3*2.*ydel*ydel;
dxy = T5*2.*xydel*xdel  + T4*ydel*xdel;

/* calculate zeroth, first, and second differences
  for specular component */

specular0 = S5*xs*xs + S4*xs*ys + S3*ys*ys + S2*xs + S1*ys + S0;
sx10 = S5*(2.*xs*xdel+xdel*xdel) + S4*ys*xdel + S2*xdel;
sx2 = S5*2.*xdel*xdel;
```

```
sy1 = S5*(2.*xs*xydel+xydel*xydel) + S4*(ys*xydel+xs*ydel+xydel*ydel) +
  S3*(2.*ys*ydel+ydel*ydel) + S2*xydel + S1*ydel;
sy2 = S5*2.*xydel*xydel + S4*2.*ydel*xydel + S3*2.*ydel*ydel;
sxy = S5*2.*xydel*xdel  + S4*ydel*xdel;

/* prepare for iteration from low y to mid y */

ymin = (int)(yworld[ylow]);
ymax = (int)(yworld[other]);

xmin = xworld[ylow];
xmax = xworld[ylow];

dxmin  = (xworld[yhigh] - xworld[ylow]) * yh_yl;
dxmax  = (xworld[other] - xworld[ylow] ) / (yworld[other] - yworld[ylow]);

/* for each y */

for (iy=ymin+1; iy<=ymax; iy++) {
  diffuse = diffuse0;
  dx1 = dx10;
  specular = specular0;
  sx1 = sx10;
  ixmin = (int)xmin;
  ixmax = (int)xmax;

  if (xmax > xmin) {

    /* for each x */

    for (ix=ixmin+1; ix<=ixmax; ix++) {
      shade = (int) ((ambient + kd*diffuse +
          ks*pow(specular, spec_power)) * 255.);

      ambient_cutoff (&shade);

      pr_put (region, ix, iy, shade);

      /* increment for step in x */

      diffuse = diffuse + dx1;
      dx1 = dx1 + dx2;
      specular = specular + sx1;
      sx1 = sx1 + sx2;
    }
  }
  else {
```

```
    /* for each x */

    for (ix=ixmin; ix>ixmax; ix--) {
     shade = (int) ((ambient + kd*diffuse +
                ks*pow(specular, spec_power)) * 255.);
     ambient_cutoff (&shade);

     pr_put (region, ix, iy, shade);

     /* increment for step in x */

     diffuse = diffuse + dx1;
     dx1 = dx1 + dx2;
     specular = specular + sx1;
     sx1 = sx1 + sx2;
    }
   }

   /* increment for step in y */

   diffuse0 = diffuse0 + dy1;
   dy1 = dy1 + dy2;
   dx10 = dx10 + dxy;

   specular0 = specular0 + sy1;
   sy1 = sy1 + sy2;
   sx10 = sx10 + sxy;

   xmin = xmin + dxmin;
   xmax = xmax + dxmax;
  }

  /* prepare for iteration from mid y to high y */

  ymin = (int)(yworld[other]);
  ymax = (int)(yworld[yhigh]);

  xmax = xworld[other];
  dxmax = (xworld[yhigh] - xworld[other]) / (yworld[yhigh] - yworld[other]);

  /* for each y */

  for (iy=ymin+1; iy<=ymax; iy++) {
   diffuse = diffuse0;
   dx1 = dx10;
   specular = specular0;
   sx1 = sx10;
```

```
ixmin = (int)xmin;
ixmax = (int)xmax;

if (xmax > xmin) {

 /* for each x */

 for (ix=ixmin+1; ix<=ixmax; ix++) {
  shade = (int) ((ambient + kd*diffuse +
            ks*pow(specular, spec_power)) * 255.);
  ambient_cutoff (&shade);

  pr_put (region, ix, iy, shade);

  /* increment for step in x */

  diffuse = diffuse + dx1;
  dx1 = dx1 + dx2;
  specular = specular + sx1;
  sx1 = sx1 + sx2;
 }
}
else {

 /* for each x */

 for (ix=ixmin; ix>ixmax; ix--) {
  shade = (int) ((ambient + kd*diffuse +
            ks*pow(specular, spec_power)) * 255.);
  ambient_cutoff (&shade);

  pr_put (region, ix, iy, shade);

  /* increment for step in x */

  diffuse = diffuse + dx1;
  dx1 = dx1 + dx2;
  specular = specular + sx1;
  sx1 = sx1 + sx2;
 }
}

/* increment for step in y */

diffuse0 = diffuse0 + dy1;
dy1 = dy1 + dy2;
dx10 = dx10 + dxy;
```

```
    specular0 = specular0 + sy1;
    sy1 = sy1 + sy2;
    sx10 = sx10 + sxy;

    xmin = xmin + dxmin;
    xmax = xmax + dxmax;
  }
}
```

Quadratic Interpolation of Intensity

```
#include "globals.h"

triangle (p1, p2, p3, ipoint)
  int p1, p2, p3, ipoint;
{
  int   k, adj1, adj2, adj3, nvp1, nvp2;
  double q1, q2, q3, dxdz4, dydz4, dxdz7, dydz7, dxdz9, dydz9, alph, beta, gamm,
      a1, a2, a3, a4, a5, a6, a7, a8, a9, b1, b2, b3, b4, b5, b6, b7, b8, b9;

  /* determine adjacency */

  nvp1 = nvertex[p1];
  if (boundary[p1] && boundary[p2]) adj1 = -1;
  else adj1 = adj[p1][(ipoint-1+nvp1) % nvp1];
  if (boundary[p1] && boundary[p3]) adj3 = -1;
  else adj3 = adj[p1][(ipoint+2) % nvp1];
  if (boundary[p3] && boundary[p2]) adj2 = -1;
  else {
    nvp2 = nvertex[p2];
    for (k=0; k<nvp2; k++) {
      if (adj[p2][k] == p3) {
        if (adj[p2][(k-1+nvp2) % nvp2] == p1) adj2 = adj[p2][(k+1) % nvp2];
        else adj2 = adj[p2][(k-1+nvp2) % nvp2];
      }
    }
  }

  /* start with given coordinates and calculated tangents */

  x[7] = xworld[p1];
  y[7] = yworld[p1];
  x[9] = xworld[p2];
  y[9] = yworld[p2];
  x[4] = xworld[p3];
  y[4] = yworld[p3];

  dxdz7 = xtang[p1];
  dydz7 = ytang[p1];
  dxdz9 = xtang[p2];
  dydz9 = ytang[p2];
  dxdz4 = xtang[p3];
  dydz4 = ytang[p3];

  /* determine incenter */

  x[13] = xincenter[p1][ipoint];
  y[13] = yincenter[p1][ipoint];
```

```
q1   = qq1[p1][ipoint];
q2   = qq2[p1][ipoint];
q3   = qq3[p1][ipoint];

/* compute ratio at midpoint of first edge */

if (adj1 >=0) midpoint3 (p1, p2, p3, adj1, ipoint);
else {
  x[3] = x[7] + .5 * (x[9]-x[7]);
  y[3] = y[7] + .5 * (y[9]-y[7]);
}
if (fabs(x[7]-x[9]) < 0.0001) alph = (y[9]-y[3])/(y[9]-y[7]);
else                  alph = (x[9]-x[3])/(x[9]-x[7]);

/* compute ratio at midpoint of second edge */

if (adj2 >= 0) midpoint2 (p1, p2, p3, adj2, ipoint);
else {
  x[2] = x[4] + .5 * (x[9]-x[4]);
  y[2] = y[4] + .5 * (y[9]-y[4]);
}
if (fabs(x[4]-x[9]) < 0.0001) beta = (y[2]-y[4])/(y[9]-y[4]);
else                  beta = (x[2]-x[4])/(x[9]-x[4]);

/* compute ratio at midpoint of third edge */

if (adj3 >= 0) midpoint1 (p1, p2, p3, adj3, ipoint);
else {
  x[1] = x[4] + .5 * (x[7]-x[4]);
  y[1] = y[4] + .5 * (y[7]-y[4]);
}
if (fabs(x[7]-x[4]) < 0.0001) gamm = (y[1]-y[7])/(y[4]-y[7]);
else                  gamm = (x[1]-x[7])/(x[4]-x[7]);

/* precalculate factors */

a1 = .5*((q1-1.)*x[4] + q2*x[7] + q3*x[9]);
a2 = .5*(q1*x[4] + (q2-1.)*x[7] + q3*x[9]);
a3 = .5*(q1*x[4] + q2*x[7] + (q3-1.)*x[9]);
a4 = .5*beta*(x[9]-x[4]);
a5 = .5*(1.-gamm)*(x[7]-x[4]);
a6 = .5*gamm*(x[4]-x[7]);
a7 = .5*(1.-alph)*(x[9]-x[7]);
a8 = .5*alph*(x[7]-x[9]);
a9 = .5*(1.-beta)*(x[4]-x[9]);

b1 = .5*((q1-1.)*y[4] + q2*y[7] + q3*y[9]);
```

```
  b2 = .5*(q1*y[4] + (q2-1.)*y[7] + q3*y[9]);
  b3 = .5*(q1*y[4] + q2*y[7] + (q3-1.)*y[9]);
  b4 = .5*beta*(y[9]-y[4]);
  b5 = .5*(1.-gamm)*(y[7]-y[4]);
  b6 = .5*gamm*(y[4]-y[7]);
  b7 = .5*(1.-alph)*(y[9]-y[7]);
  b8 = .5*alph*(y[7]-y[9]);
  b9 = .5*(1.-beta)*(y[4]-y[9]);

  /* compute Bezier control points */

  z[4] = intens[p3];
  z[7] = intens[p1];
  z[9] = intens[p2];
  z[6]  = z[4] + a4*dxdz4 + b4*dydz4;
  z[5]  = z[4] + a5*dxdz4 + b5*dydz4;
  z[8]  = z[7] + a6*dxdz7 + b6*dydz7;
  z[10] = z[7] + a7*dxdz7 + b7*dydz7;
  z[12] = z[9] + a8*dxdz9 + b8*dydz9;
  z[11] = z[9] + a9*dxdz9 + b9*dydz9;
  z[17] = z[4] + a1*dxdz4 + b1*dydz4;
  z[20] = z[7] + a2*dxdz7 + b2*dydz7;
  z[22] = z[9] + a3*dxdz9 + b3*dydz9;
  z[13] = q1*z[17] + q2*z[20] + q3*z[22];
  z[2] = (1.-beta)*z[6] + beta*z[11];
  z[1] = gamm*z[5] + (1.-gamm)*z[8];
  z[3] = alph*z[10] + (1.-alph)*z[12];
  z[15] = (1.-beta)*z[17] + beta*z[22];
  z[14] = gamm*z[17] + (1.-gamm)*z[20];
  z[16] = alph*z[20] + (1.-alph)*z[22];

  /* shade each of the six subtriangular facets */

  draw (7, 3);
  draw (3, 9);
  draw (9, 2);
  draw (2, 4);
  draw (4, 1);
  draw (1, 7);
}

/******************************** Draw ******************************/

draw (a, b)
  int a, b;
{
  double dr, ds, dt, mid, dry, dsy, dty, bez00, bez11, bez22, bez01, bez02, bez12,
```

```
    yh_yl, yh_yo, yo_yl, xh_xl, xh_xo, xo_xl;
int   shade, ylow, yhigh, other, ixmin, ixmax, ymin, ymax;
register int   ix, iy;
register double intensity, intensity00, intens100, intens1, intens2,
        inty1, inty2, delxy, xmin, xmax, dxmin, dxmax;

/* find smallest and largest y coordinate */

ylow = a;
if (y[b]<y[a]) ylow = b;
if (y[13]<y[ylow]) ylow = 13;
yhigh = a;
if (y[b]>y[a]) yhigh = b;
if (y[13]>y[yhigh]) yhigh = 13;
other = a;
if (ylow!=b && yhigh!=b) other = b;
if (ylow!=13 && yhigh!=13) other = 13;

/* precalculate factors */

xh_xl = x[yhigh]-x[ylow];
xh_xo = x[yhigh]-x[other];
xo_xl = x[other]-x[ylow];

yh_yl = 1. / (y[yhigh]-y[ylow]);
yh_yo = 1. / (y[yhigh]-y[other]);
yo_yl = 1. / (y[other]-y[ylow]);

bez00 = z[other];
bez11 = z[yhigh];
bez22 = z[ylow];
bez01 = z[other+yhigh];
bez02 = z[other+ylow];
bez12 = z[yhigh+ylow];

/* covert from barycentric to screen coordinates */

if (fabs(y[yhigh] - y[ylow]) < 0.0001) dr = 0.;
else {
  mid = (y[other]-y[ylow]) * yh_yl;
  mid = x[ylow] + mid * xh_xl;
  dr = 1. / (x[other]-mid);
}
if (fabs(y[other] - y[ylow]) < 0.0001) ds = 0.;
else {
  mid = (y[yhigh]-y[ylow]) * yo_yl;
  mid = x[ylow] + mid * xo_xl;
```

```
  ds = 1. / (x[yhigh]-mid);
}
if (fabs(y[yhigh] - y[other]) < 0.0001) dt = 0.;
else {
  mid = (y[ylow]-y[other]) * yh_yo;
  mid = x[other] + mid * xh_xo;
  dt = 1. / (x[ylow]-mid);
}
if (dr < 0.0) {
  dr = -dr;
  ds = -ds;
  dt = -dt;
}

/* calculate zeroth, first, and second differences */

intensity00 = bez22;
intens100  = bez00*dr*dr + bez11*ds*ds + bez22*dt*(2. + dt) +
        2.*bez01*dr*ds + 2.*bez02*dr*(1. + dt) + 2.*bez12*ds*(1. + dt);
intens2    = 2.*(bez00*dr*dr + bez11*ds*ds + bez22*dt*dt) +
        4.*(bez01*dr*ds + bez02*dr*dt + bez12*ds*dt);

dry = 0.0;
dsy = yh_yl;
dty = -dsy;

inty1 = bez11*dsy*dsy + bez22*dty*(2. + dty) + 2.*bez12*dsy*(1. + dty);
inty2 = 2.*(bez11*dsy*dsy + bez22*dty*dty) + 4.*bez12*dsy*dty;
delxy = bez11*2.*dsy*ds + bez22*2.*dty*dt +
     2.*(bez01*dr*dsy + bez02*dr*dty + bez12*(ds*dty + dt*dsy));

/* prepare for iteration from low y to mid y */

ymin = (int)y[ylow];
ymax = (int)y[other];

xmin = x[ylow];
xmax = x[ylow];

dxmin  = (x[yhigh] - x[ylow]) * yh_yl;
dxmax  = (x[other] - x[ylow]) * yo_yl;

/* for each y */

for (iy=ymin+1; iy<=ymax; iy++) {
  intensity = intensity00;
  intens1   = intens100;
```

```
    ixmin = (int)xmin;
    ixmax = (int)xmax;

    if (xmax > xmin) {

      /* for each x */

      for (ix=ixmin+1; ix<=ixmax; ix++) {
        shade = (int)(intensity);
        ambient_cutoff (&shade);

        pr_put (region, ix, iy, shade);

        /* increment for step in x */

        intensity = intensity + intens1;
        intens1   = intens1   + intens2;
      }
    }
    else {

      /* for each x */

      for (ix=ixmin; ix>ixmax; ix--) {
        shade = (int)(intensity);
        ambient_cutoff (&shade);

        pr_put (region, ix, iy, shade);

        /* increment for step in x */

        intensity = intensity + intens1;
        intens1   = intens1   + intens2;
      }
    }

    /* increment for step in y */

    intensity00 = intensity00 + inty1;
    inty1       = inty1       + inty2;
    intens100   = intens100   + delxy;
    xmin = xmin + dxmin;
    xmax = xmax + dxmax;
  }

  /* prepare for iteration from mid y to high y */
```

Quadratic Interpolation of Intensity

```
ymin = (int)(y[other]);
ymax = (int)(y[yhigh]);

xmax = x[other];
dxmax = (x[yhigh] - x[other]) * yh_yo;

/* for each y */

for (iy=ymin+1; iy<=ymax; iy++) {
  intensity = intensity00;
  intens1   = intens100;
  ixmin = (int)xmin;
  ixmax = (int)xmax;

  if (xmax > xmin) {

    /* for each x */

    for (ix=ixmin+1; ix<=ixmax; ix++) {
      shade = (int)(intensity);
      ambient_cutoff (&shade);

      pr_put (region, ix, iy, shade);

      /* increment for step in x */

      intensity = intensity + intens1;
      intens1   = intens1   + intens2;
    }
  }
  else {

    /* for each x */

    for (ix=ixmin; ix>ixmax; ix--) {
      shade = (int)(intensity);
      ambient_cutoff (&shade);

      pr_put (region, ix, iy, shade);

      /* increment for step in x */

      intensity = intensity + intens1;
      intens1   = intens1   + intens2;
    }
  }
```

```
    /* increment for step in y */

    intensity00 = intensity00 + inty1;
    inty1       = inty1       + inty2;
    intens100   = intens100   + delxy;
    xmin = xmin + dxmin;
    xmax = xmax + dxmax;
  }
}
```

Adaptive Subdivision - Object Surface

```
#include "globals.h"

/****************************** Subdivide Object ******************************/

subdivide_object ()
{
 int    j, p1, p2, p3, opoints, start_point, nvert;
 double  xdiff, ydiff, zdiff, deltaI, sin1, cos1, sin2, cos2;

 initialize ();
 opoints = 0;

 /* loop until no further subdivisions */

 while (1) {
  start_point = opoints;
  opoints = npoints;

  /* for each vertex */

  for (p1 = start_point; p1 < opoints; p1++) {

   /* for each edge */

   for (j = 0; j < nvertex[p1]; j++) {
    p2 = adj[p1][j];
    if (p1 > p2) {

     /* make sure edge is long enough */

     xdiff = xworld[p1] - xworld[p2];
     ydiff = yworld[p1] - yworld[p2];
     if (xdiff*xdiff + ydiff*ydiff > max_dist) {
      zdiff = zworld[p1] - zworld[p2];

      /* calculate rotation angles */

      sin1 = xnormal[p1] / sqrt (xnormal[p1]*xnormal[p1] + 1.);
      cos1 = 1. / sqrt (xnormal[p1]*xnormal[p1] + 1.);
      sin2 = ynormal[p1] / sqrt (ynormal[p1]*ynormal[p1] + 1.);
      cos2 = 1. / sqrt (ynormal[p1]*ynormal[p1] + 1.);

      /* determine distance to tangent plane in rotated coordinate system */

      deltaI = (xdiff * sin2) + (ydiff * sin1 * cos2) +
           (zdiff * cos1 * cos2);
```

```
        /* add point if exceeds threshold */

        if (fabs(deltaI) > max_norm)
          add_point (p1,p2,j);

        /* make same checks for other vertex */

        else {

         /* calculate rotation angles */

         sin1 = xnormal[p2] / sqrt (xnormal[p2]*xnormal[p2] + 1.);
         cos1 = 1. / sqrt (xnormal[p2]*xnormal[p2] + 1.);
         sin2 = ynormal[p2] / sqrt (ynormal[p2]*ynormal[p2] + 1.);
         cos2 = 1. / sqrt (ynormal[p2]*ynormal[p2] + 1.);

         /* determine distance to tangent plane in rotated coordinate system */

         deltaI = (xdiff * sin2) + (ydiff * sin1 * cos2) +
               (zdiff * cos1 * cos2);

         /* add point if exceeds threshold */

         if (fabs(deltaI) > max_norm)
           add_point (p1,p2,j);
        }
       }
      }
     }
    }
    if (opoints == npoints) break;

    /* reform triangles */

    connect_points (opoints);
   }
  }
```

Adaptive Subdivision - Intensity Surface

```
#include "globals.h"

/****************************** Subdivide Intensity ******************************/

subdivide_intensity ()
{
 int opoints, start_point, p1, p2, j;

 opoints = 0;

 /* loop until no further subdivisions */

 while (1) {
  start_point = opoints;
  opoints = npoints;

  /* for each vertex */

  for (p1 = start_point; p1 < opoints; p1++) {

   /* for each edge */

   for (j = 0; j < nvertex[p1]; j++) {
    p2 = adj[p1][j];
    if (p1 > p2) {

     /* make sure edge is long enough */

     xdiff = xworld[p1] - xworld[p2];
     ydiff = yworld[p1] - yworld[p2];
     if (xdiff*xdiff + ydiff*ydiff > max_dist) {
      Idiff = intens[p1] - intens[p2];

      /* calculate rotation angles */

      sin1 = xtang[p1] / sqrt (xtang[p1]*xtang[p1] + 1.);
      cos1 = 1. / sqrt (xtang[p1]*xtang[p1] + 1.);
      sin2 = ytang[p1] / sqrt (ytang[p1]*ytang[p1] + 1.);
      cos2 = 1. / sqrt (ytang[p1]*ytang[p1] + 1.);

      /* determine distance to tangent plane in rotated coordinate system */

      deltaI = (xdiff * sin2) + (ydiff * sin1 * cos2) +
           (Idiff * cos1 * cos2);

      /* add point if exceeds threshold */
```

```
        if (fabs(deltaI) > max_norm)
          add_point (p1,p2,j);

        /* make same checks for other vertex */

        else {

         /* calculate rotation angles */

         sin1 = xtang[p2] / sqrt (xtang[p2]*xtang[p2] + 1.);
         cos1 = 1. / sqrt (xtang[p2]*xtang[p2] + 1.);
         sin2 = ytang[p2] / sqrt (ytang[p2]*ytang[p2] + 1.);
         cos2 = 1. / sqrt (ytang[p2]*ytang[p2] + 1.);

         /* determine distance to tangent plane in rotated coordinate system */

         deltaI = (xdiff * sin2) + (ydiff * sin1 * cos2) +
              (Idiff * cos1 * cos2);

         /* add point if exceeds threshold */

         if (fabs(deltaI) > max_norm)
           add_point (p1,p2,j);
        }
       }
      }
     }
    }
    if (opoints == npoints) break;

    /* reform triangles */

    connect_points (opoints);

   }
  }
```

Locate Specular Highlights

```
#include "globals.h"

/****************************** Locate Specular ******************************/

locate_specular ()
{
 int    j, u, p1, p2, p3, opoints, nvert;
 double  r, s, t;

 initialize ();
 opoints = npoints;

 /* for each vertex */

 for (p1=0; p1<opoints; p1++) {
  nvert = adjacent(p1);

  /* for each facet */

  for (j=0; j<nvert; j++) {
   p2 = adj[p1][j];
   p3 = adj[p1][(j+1) % nvertex[p1]];

   if (p2 > p1 && p3 > p1) {

    /* determine barycentric coordinates of vector of maximum
      highlight in terms of local intensity surface normals */

    bary (xnormal[p1], ynormal[p1], xnormal[p2], ynormal[p2],
       xnormal[p3], ynormal[p3], xhigh, yhigh, &r, &s, &t);

    /* highlight falls with facet if all three barycentric
      coordinates are in the range of 0 to 1 */

    if (r > 0. && r < 1. && s > 0. && s < 1. && t > 0. && t < 1.) {
     if (!boundary[p1] || !boundary[p2] || !boundary[p3]) {

      /* move existing vertex to position of maximum highlight */

      u = p1;
      if (s * (1-boundary[p2]) > r * (1-boundary[p1])) u = p2;
      if (t * (1-boundary[p3]) > r * (1-boundary[p1]) &&
        t * (1-boundary[p3]) > s * (1-boundary[p2])) u = p3;

      xworld[u] = r*xworld[p1] + s*xworld[p2] + t*xworld[p3];
      yworld[u] = r*yworld[p1] + s*yworld[p2] + t*yworld[p3];
```

```
            xnormal[u] = xhigh;
            ynormal[u] = yhigh;
            znormal[u] = zhigh;
          }
        }
      }
    }
  }
}
```

Adaptive Subdivision - Specular Component

```
#include "globals.h"

subdivide_specular ()
{
 int  opoints, start_point, p1, p2, j;

 opoints = 0;

 /* loop until no further subdivisions */

 while (1) {
  start_point = opoints;
  opoints = npoints;

  /* for each vertex */

  for (p1=start_point; p1<opoints; p1++) {

   /* for each edge */

   for (j=0; j<nvertex[p1]; j++) {
    p2 = adj[p1][j];

    /* add point if difference in specular components
      exceeds given threshold */

    if (p1 > p2 && fabs(specular[p1]-specular[p2]) > max_spec) {

     /* make sure edge is long enough */

     xdiff = xworld[p1] - xworld[p2];
     ydiff = yworld[p1] - yworld[p2];
     if (xdiff*xdiff + ydiff*ydiff > max_dist) {

      add_point (p1,p2,j);
     }
    }
   }
  }
  if (opoints == npoints) break;

  /* reform triangles */

  connect_points (opoints);
 }
}
```

Locate Diffuse Boundary

```
#include "globals.h"

/****************************** Locate Diffuse ******************************/

locate_diffuse ()
{
 int    ilight, opoints, p1, p2, p3;
 double  xdiff, ydiff, lbound, ubound, factor, diffuse;

 opoints = npoints;

 /* for each light source */

 for (ilight=0; ilight<nlights; ilight++) {

  /* for each vertex */

  for (p1=0; p1<=opoints; p1++) {

   /* for each edge */

   for (j=0; j<nvertex[p1]; j++) {

    /* if intensity of point 1 negative */

    if (intens2[ilight][p1] < -1.0 && !dboundary[ilight][p1]) {
     p2 = adj[p1][j];
     if (p2 <= opoints) {

      /* and intensity of point 2 positive */

      if (intens2[ilight][p2] > 1.0 && !dboundary[ilight][p2]) {

       /* find point where diffuse component is zero iteratively */

       xdiff = xworld[p2] - xworld[p1];
       ydiff = yworld[p2] - yworld[p1];
       lbound = 0.;
       factor = 1.;
       diffuse = 99.;

       /* loop until close enough to zero diffuse */

       while (fabs(diffuse) > 1.) {
        if (diffuse > 0.0) ubound = factor;
        else          lbound = factor;
```

```
            /* try new midpoint */

            factor = .5 * (ubound + lbound);
            xnew = xworld[p1] + factor*xdiff;
            ynew = yworld[p1] + factor*ydiff;
            znew = sqrt(1. - xnew*xnew - ynew*ynew);

            /* calculate new diffuse component */

            diffuse = kd[ilight] * 255. * (xnew*xlight[ilight] +
                  ynew*ylight[ilight] + znew*zlight[ilight]);
          }
          p3 = add_point(p1,p2,factor,j,0);
          dboundary[ilight][p3] = 1;
        }
      }
    }
  }
 }

 /* reform triangles */

 connect_points(opoints+1);
}
```

Ambient Cut-off

```
#include "globals.h"

/****************************** Ambient Cut-off ******************************/

ambient_cutoff (shade)
 int *shade;
{
 /* if calculated shade is less than the ambient light level */

 if (*shade < iamb) *shade = iamb;
}
```

```
#include "globals.h"

/****************************** Calc Intensity ******************************/

calc_intensity (p1)
 int p1;
{
 double  diffuse, xreflect, yreflect, zreflect;

 /* calculate diffuse component */

 diffuse = xnormal[p1]*xlight + ynormal[p1]*ylight + znormal[p1]*zlight;

 /* determine reflected light vector */

 xreflect = 2.*diffuse*xnormal[p1] - xlight;
 yreflect = 2.*diffuse*ynormal[p1] - ylight;
 zreflect = 2.*diffuse*znormal[p1] - zlight;
 normalize (&xreflect, &yreflect, &zreflect);

 /* calculate specular component */

 specular[p1] = pow ((xreflect*xeye + yreflect*yeye + zreflect*zeye), spec_power);
 if (specular[p1] < 0.0) specular[p1] = 0.0;

 /* compute intensity */

 intens[p1] = (ambient + kd*diffuse + ks*specular[p1]) * 255.;
}

/****************************** Calc Slope ******************************/

calc_slope (p1)
 int p1;
{
 /* compute tangents to intensity surface */

 double xslope=0.0, yslope=0.0, area, xarea=0.0, yarea=0.0, aroot, broot, cosine,
      x, y, zx, zy, xa, xb, xc, ya, yb, yc, a, b, c;
 int   j, k, p2, p3, nvert;

 /* for each facet adjoining point 1 */

 nvert = adjacent(p1);
 for (j=0; j<nvert; j++) {
  k = (j+1) % nvertex[p1];
  p2 = adj[p1][j];
```

```
p3 = adj[p1][k];

xa = xworld[p1] - xworld[p2];
ya = yworld[p1] - yworld[p2];
xb = xworld[p3] - xworld[p1];
yb = yworld[p3] - yworld[p1];
xc = xworld[p3] - xworld[p2];
yc = yworld[p3] - yworld[p2];

a = xa*xa + ya*ya;
b = xb*xb + yb*yb;
c = xc*xc + yc*yc;

aroot = sqrt (a);
broot = sqrt (b);
cosine = (a+b-c) / (2.*aroot*broot);
if (fabs(cosine) > 1.0) cosine = 1.0;

/* calculate area of facet */

area = aroot * broot * sqrt (1. - cosine*cosine);

/* compute intensity slope in x direction */

if (fabs(yc) < 0.0001) {
 xslope = xslope + area * (intens[p3] - intens[p2]) / xc;
 xarea = xarea + area;
}
else {
 x = ya * xc / yc + xworld[p2];
 if (fabs(x - xworld[p1]) > 0.0001) {
  zx = ya * (intens[p3]-intens[p2]) / yc + intens[p2];
  xslope = xslope + area * (zx - intens[p1]) / (x - xworld[p1]);
  xarea = xarea + area;
 }
}

/* compute intensity slope in y direction */

if (fabs(xc) < 0.0001) {
 yslope = yslope + area * (intens[p3] - intens[p2]) / yc;
 yarea = yarea + area;
}
else {
 y = xa * yc / xc + yworld[p2];
 if (fabs(y - yworld[p1]) > 0.0001) {
  zy = xa * (intens[p3]-intens[p2]) / xc + intens[p2];
```

```
      yslope = yslope + area * (zy - intens[p1]) / (y - yworld[p1]);
      yarea = yarea + area;
     }
    }
   }

  /* compute tangent as weighted average based on area */

  xtang[p1] = xslope / xarea;
  ytang[p1] = yslope / yarea;
}

/****************************** Normalize ******************************/

normalize (xvector, yvector, zvector)
 double *xvector,  *yvector,  *zvector;
{
 double tot;

 /* normalize to unit vector in three dimensions */

 tot = sqrt (*xvector*(*xvector)+*yvector*(*yvector)+*zvector*(*zvector));
 *xvector = *xvector/tot;
 *yvector = *yvector/tot;
 *zvector = *zvector/tot;
}

/****************************** Norm2 ******************************/

norm2 (xvector, yvector)
 double *xvector,  *yvector;
{
 double tot;

 /* normalize to unit vector in two dimensions */

 tot = sqrt (*xvector*(*xvector)+*yvector*(*yvector));
 *xvector = *xvector/tot;
 *yvector = *yvector/tot;
}

/****************************** Initialize ******************************/

initialize ()
{
 int i;
```

```
 for (i=0; i<1000; i++) {
  nconnect[i] = 0;
 }
}

/******************************** Read Light Data ******************************/

read_light_data ()
{
 FILE   *stream;

 /* read light vector */

 stream = fopen ("light.data", "r");
 fscanf (stream, "%lf%lf%lf", &xlight, &ylight, &zlight);
 ylight = -ylight;
 normalize (&xlight, &ylight, &zlight);

 /* read eye direction */

 fscanf (stream, "%lf%lf%lf", &xeye, &yeye, &zeye);
 yeye = -yeye;
 normalize (&xeye, &yeye, &zeye);

 /* calculate vector of maximum highlight */

 xhigh = xeye + xlight;
 yhigh = yeye + ylight;
 zhigh = zeye + zlight;
 normalize (&xhigh, &yhigh, &zhigh);

 /* read ambient light level and reflectance coefficients */

 fscanf (stream, "%lf", &ambient);
 iamb = ambient * 255.;
 fscanf (stream, "%lf", &kd);
 fscanf (stream, "%lf", &ks);
 fscanf (stream, "%lf", &spec_power);

 /* read threshold criteria */

 fscanf (stream, "%lf", &max_spec);
 fscanf (stream, "%lf", &max_dist);
 max_dist = max_dist*max_dist;
}

/******************************** Read Object Data ******************************/
```

```
read_object_data (filename)
 char *filename[];
{
 int   i, j, ipoint;
 FILE  *stream;

 /* read total number of vertices */

 stream = fopen (filename, "r");
 fscanf (stream, "%d", &npoints);

 for (i=0; i<npoints; i++) {

  /* read world coordinates */

  fscanf (stream, "%d", &ipoint);
  fscanf (stream, "%lf", &xworld[ipoint]);
  fscanf (stream, "%lf", &yworld[ipoint]);

  /* read surface normal */

  fscanf (stream, "%d", &boundary[ipoint]);
  fscanf (stream, "%lf", &xnormal[ipoint]);
  fscanf (stream, "%lf", &ynormal[ipoint]);
  ynormal[ipoint] = -ynormal[ipoint];
  fscanf (stream, "%lf", &znormal[ipoint]);

  /* read adjacency info */

  fscanf (stream, "%d", &nvertex[ipoint]);
  for (j=0; j<nvertex[ipoint]; j++) {
   fscanf (stream, "%d", &adj[ipoint][j]);
  }
 }
}

/****************************** Bary ******************************/

bary (x1, yy1, x2, y2, x3, y3, xs1, ys1, r, s, t)
 double x1, x2, x3, yy1, y2, y3, xs1, ys1, *r, *s, *t;
{
 /* compute barycentric coordinates */

 if (fabs(x2-x1) > .0000001) {
  *t = (ys1 - yy1 - (y2-yy1)*(xs1-x1)/(x2-x1)) /
   ((y2-yy1)*(x1-x3)/(x2-x1) - yy1 + y3);
  *s = (xs1-x1+(x1-x3)*(*t)) / (x2-x1);
```

```
  *r = 1. - *s - *t;
 }
 else {
  *s = (ys1 - yy1 - (y3-yy1)*(xs1-x1)/(x3-x1)) /
   ((y3-yy1)*(x1-x2)/(x3-x1) - yy1 + y2);
  *t = (xs1-x1+(x1-x2)*(*s)) / (x3-x1);
  *r = 1. - *s - *t;
 }
}

/****************************** Centroid ******************************/

centroid(p1, p2, p3, xcen, ycen)
 int    p1, p2, p3;
 double *xcen,  *ycen;
{
 /* compute centroid of triangle */

 *xcen = (xworld[p1]+xworld[p2]+xworld[p3])/3.;
 *ycen = (yworld[p1]+yworld[p2]+yworld[p3])/3.;
}

/****************************** Incenter ******************************/

incenter (p1, p2, p3, ipoint)
 int    p1, p2, p3, ipoint;
{
 double delx, dely, d1, d2, d3, dtot, q1, q2, q3;

 /* compute incenter of triangle */

 delx = xworld[p1]-xworld[p2];
 dely = yworld[p1]-yworld[p2];
 d1 = sqrt (delx*delx+dely*dely);

 delx = xworld[p3]-xworld[p2];
 dely = yworld[p3]-yworld[p2];
 d2 = sqrt (delx*delx+dely*dely);

 delx = xworld[p1]-xworld[p3];
 dely = yworld[p1]-yworld[p3];
 d3 = sqrt (delx*delx+dely*dely);

 dtot = 1. / (d1 + d2 + d3);
 q1 = d1 * dtot;
 q2 = d2 * dtot;
 q3 = d3 * dtot;
```

```
 xincenter[p1][ipoint] = q1*xworld[p3] + q2*xworld[p1] + q3*xworld[p2];
 yincenter[p1][ipoint] = q1*yworld[p3] + q2*yworld[p1] + q3*yworld[p2];
 qq1[p1][ipoint] = q1;
 qq2[p1][ipoint] = q2;
 qq3[p1][ipoint] = q3;
}

/****************************** Intersection ******************************/

intersection (x1, yy1, x2, y2, x3, y3, x4, y4, vertex)
 double  x1, x2, x3, x4, yy1, y2, y3, y4;
 int    vertex;
{
 double  l1, l2, m1, m2, n1, n2;

 /* find intersection of two lines */

 l1 = yy1 - y2;
 m1 = x2 - x1;
 n1 = -m1*yy1 - l1*x1;
 l2 = y3 - y4;
 m2 = x4 - x3;
 n2 = -m2*y3 - l2*x3;

 x[vertex] = (m1*n2-m2*n1) / (l1*m2-m1*l2);
 if (fabs(m1) > 0.0001) y[vertex] = (-(l1*x[vertex]) - n1) / m1;
 else                   y[vertex] = (-(l2*x[vertex]) - n2) / m2;
}

/****************************** Adjacent ******************************/

int adjacent (p1)
 int p1;
{
 if (boundary[p1]) return (nvertex[p1]-1);
 else          return nvertex[p1];
}

/****************************** Add Point ******************************/

add_point (p1, p2, j)
 int   p1, p2, j;
{
 /* add new point at midpoint between points 1 and 2 */

 xworld[npoints] = xworld[p1] + .5 * (xworld[p2] - xworld[p1]);
```

```
 yworld[npoints] = yworld[p1] + .5 * (yworld[p2] - yworld[p1]);

 /* calculate surface normal via linear interpolation */

 xnormal[npoints] = xnormal[p1] + .5 * (xnormal[p2] - xnormal[p1]);
 ynormal[npoints] = ynormal[p1] + .5 * (ynormal[p2] - ynormal[p1]);
 znormal[npoints] = znormal[p1] + .5 * (znormal[p2] - znormal[p1]);
 normalize (&xnormal[npoints], &ynormal[npoints], &znormal[npoints]);

 /* calculate new intensity */

 calc_intensity (npoints);

 /* update adjacency info */

 insert_list[npoints][1] = p1;
 insert_list[npoints][2] = p2;
 boundary[npoints] = boundary[p1] && boundary[p2] &&
  (adj[p1][0] == p2 || adj[p1][nvertex[p1]-1] == p2);
 nvertex[npoints] = 2;
 adj[npoints][0] = p1;
 adj[npoints][1] = p2;
 replace (p2, p1, npoints);
 adj[p1][j] = npoints;

 npoints++;
}

/****************************** Midpoint1 ******************************/

midpoint1 (p1, p2, p3, adj, ipoint)
 int p1, p2, p3, adj, ipoint;
{
 /* find middle Bezier control point for edge 1 */

 int index, pmin;

 /* previously computed */

 if (ipoint == nvertex[p1]-1 || adj < p1) {
  x[1] = xmid[p1][(ipoint+1) % nvertex[p1]];
  y[1] = ymid[p1][(ipoint+1) % nvertex[p1]];
 }

 /* determine intersection between edge and line connecting incenters */

 else {
```

```
  intersection (x[4], y[4], x[7], y[7], x[13], y[13],
          xincenter[p1][ipoint+1], yincenter[p1][ipoint+1], 1);
  xmid[p1][ipoint+1] = x[1];
  ymid[p1][ipoint+1] = y[1];
 }
}

/****************************** Midpoint2 ******************************/

midpoint2 (p1, p2, p3, adj, ipoint)
 int p1, p2, p3, adj, ipoint;
{
 /* find middle Bezier control point for edge 2 */

 int index, pmin;

 /* previously computed */

 if (adj < p1) {
  find_mid2 (p2, p3, &pmin, &index);
  x[2] = xmid[pmin][index];
  y[2] = ymid[pmin][index];
 }

 /* determine intersection between edge and line connecting incenters */

 else {
  find_mid (p2, p3, adj, &pmin, &index);
  intersection (x[4], y[4], x[9], y[9], x[13], y[13],
          xincenter[pmin][index], yincenter[pmin][index], 2);

  find_mid2 (p2, p3, &pmin, &index);
  xmid[pmin][index] = x[2];
  ymid[pmin][index] = y[2];
 }
}

/****************************** Midpoint3 ******************************/

midpoint3 (p1, p2, p3, adj, ipoint)
 int p1, p2, p3, adj, ipoint;
{
 /* find middle Bezier control point for edge 3 */

 int index, pmin;

 /* previously computed */
```

```
 if (ipoint > 0 || adj < p1) {
  x[3] = xmid[p1][ipoint];
  y[3] = ymid[p1][ipoint];
 }

 /* determine intersection between edge and line connecting incenters */

 else {
  intersection (x[7], y[7], x[9], y[9], x[13], y[13],
          xincenter[p1][nvertex[p1]-1], yincenter[p1][nvertex[p1]-1], 3);
  xmid[p1][ipoint] = x[3];
  ymid[p1][ipoint] = y[3];
 }
}

/****************************** Find Mid ******************************/

find_mid (p1, p2, p3, pmin, index)
 int   p1, p2, p3,  *pmin,  *index;
{
 if (p2 < p3) {
  if (p2 < p1) {
   *pmin = p2;
   *index = find_index (p2, p3, p1);
  }
  else {
   *pmin = p1;
   *index = find_index (p1, p2, p3);
  }
 }
 else {
  if (p3 < p1) {
   *pmin = p3;
   *index = find_index (p3, p2, p1);
  }
  else {
   *pmin = p1;
   *index = find_index (p1, p2, p3);
  }
 }
}

/****************************** Find Mid2 ******************************/

find_mid2 (p1, p2, pmin, index)
 int   p1, p2,  *pmin,  *index;
{
```

```
 if (p1 < p2) {
  *pmin = p1;
  *index = find_index2 (p1, p2);
 }
 else {
  *pmin = p2;
  *index = find_index2 (p2, p1);
 }
}

/****************************** Find Index ******************************/

int find_index (p1, p2, p3)
 int p1, p2, p3;
{
 int i, nvert;

 nvert = nvertex[p1];

 for (i=0; i<nvert; i++) {
  if (adj[p1][i] == p2  && adj[p1][(i+1) % nvert] == p3) return i;
  if (adj[p1][i] == p3  && adj[p1][(i+1) % nvert] == p2) return i;
 }
}

/****************************** Find Index2 ******************************/

int find_index2 (p1, p2)
 int p1, p2;
{
 int i, nvert;

 nvert = nvertex[p1];

 for (i=0; i<nvert; i++) {
  if (adj[p1][i] == p2) return i;
 }
}

/****************************** Connect Points ******************************/

connect_points (start_point)
 int start_point;
{
 /* reform triangles */

 int j, k, l, m, n, nv, u, p1, p2, p22, p3, p4, p5, p6, side, new_adds, nvert;
```

```
/* loop until no new points added */

new_adds = 1;
while (new_adds) {
 new_adds = 0;

 /* for each vertex */

 for (p1=0; p1<start_point; p1++) {
  nvert = adjacent(p1);

  /* for each facet */

  for (j=0; j<nvert; j++) {
   p2 = adj[p1][j];
   p3 = adj[p1][(j+1) % nvertex[p1]];

   if (p2 >= start_point && p3 >= start_point) {
    for (k=1; k<=2; k++) {
     for (l=1; l<=2; l++) {

      /* if on adjacent edges */

      if (insert_list[p2][k] == insert_list[p3][l]) {
       p4 = insert_list[p2][(k%2)+1];
       p5 = insert_list[p3][(l%2)+1];
       for (m=0; m<nvertex[p4]; m++) {

        /* if two points added, add new point on third and last edge */

        if (adj[p4][m] == p5) {
         add_connect (p4, p5);
         new_adds = 1;
         goto next_tri;
        }
       }
      }
     }
    }
   }
   next_tri:;
  }
 }
}

/* for each new point */
```

```
for (p1=start_point; p1<npoints-1; p1++) {
 for (p2=p1+1; p2<npoints; p2++) {
  for (k=1; k<=2; k++) {
   for (l=1; l<=2; l++) {

    /* if on adjacent edges */

    if (insert_list[p1][k] == insert_list[p2][l]) {
     m = insert_list[p1][k];
     nv = nvertex[m];
     if (boundary[p1] && boundary[p2] && nv > 2) goto next13;

     /* count the connections */

     for (n=0; n<nv; n++) {
      if (adj[m][n] == p1) {
       if (adj[m][(n-1+nv) % nv] == p2 || adj[m][(n+1) % nv] == p2) {
        nconnect[p1]++;
        nconnect[p2]++;
        connect_list[p1][nconnect[p1]] = p2;
        connect_list[p2][nconnect[p2]] = p1;
        find_connect (p1, nconnect[p1], &p22, &p3, &p4, &p6);
        for (u=1; u<nconnect[p1]; u++) {
         find_connect (p1, u, &p22, &p3, &p4, &p5);
         if (p5 == p6) {
          insert (p1, m, connect_list[p1][u], p2);
          goto next3;
         }
        }
        insert (p1, insert_list[p1][1], insert_list[p1][2], p2);
       next3:
        find_connect (p2, nconnect[p2], &p22, &p3, &p4, &p6);
        for (u=1; u<nconnect[p2]; u++) {
         find_connect (p2, u, &p22, &p3, &p4, &p5);
         if (p5 == p6) {
          insert (p2, m, connect_list[p2][u], p1);
          goto next4;
         }
        }
        insert (p2, insert_list[p2][1], insert_list[p2][2], p1);
       next4: goto next13;
       }
      }
     }
    }
   }
  }
 }
```

```
    }
  next13: ;
  }
}

/* for each new point */

for (p1=start_point; p1<npoints; p1++) {

  /* if not connected to any other new point,
     connect to two vertices */

  if (nconnect[p1] == 0) {
   connect2 (p1);
  }

  /* if connected to one other new point,
     connect to it and to one vertex */

  else if (nconnect[p1] == 1) {
   connect1 (p1);
   choice (p1, 1);
  }

  /* if connected to two other new points,
     connect to both and to one vertex */

  else if (nconnect[p1] == 2) {
   find_connect (p1, 1, &p2, &p3, &p4, &p5);
   find_connect (p1, 2, &p2, &p3, &p4, &p6);
   if (p5 == p6) {
    connect1 (p1);
   }
   else {
    choice (p1, 1);
    choice (p1, 2);
   }
  }

  /* if connected to three other new points,
     connect to one vertex */

  else if (nconnect[p1] == 3) {
   side = odd_man (p1);
   choice (p1, side);
  }
}
```

```
}

/****************************** Add Connect ******************************/

add_connect (p1, p2)
 int p1, p2;
{
 int i;

 for (i=0; i<nvertex[p1]; i++) {
  if (adj[p1][i] == p2) {
   add_point (p1, p2, i);
   return;
  }
 }
}

/***************************** Find Connect ******************************/

find_connect (p1, side, p2, p3, p4, p5)
 int p1, side, *p2, *p3, *p4, *p5;
{
 int i, j;
 *p2 = connect_list[p1][side];
 for (i=1; i<=2; i++) {
  for (j=1; j<=2; j++) {
   if (insert_list[*p2][j] == insert_list[p1][i]) {
    *p3 = insert_list[p1][i];
    *p4 = insert_list[p1][(i % 2) + 1];
    *p5 = insert_list[*p2][(j % 2) + 1];
    return;
   }
  }
 }
}

/****************************** Connect1 *******************************/

connect1 (p1)
 int p1;
{
 int p2, p3, p4, p5, nvp3, pj;

 /* connect new point to one existing vertex
    and update adjacency info */

 if (boundary[p1]) return;
```

```
 find_connect (p1, 1, &p2, &p3, &p4, &p5);
 nvp3 = nvertex[p3];
 for (pj=0; pj<nvp3; pj++) {
  if (adj[p3][pj] == p1) {
   if (adj[p3][(pj+1) % nvp3] == p2) {
    insert (adj[p3][(pj-1+nvp3) % nvp3], p4, p3, p1);
    insert (p1, p4, p3, adj[p3][(pj-1+nvp3) % nvp3]);
   }
   else {
    insert (adj[p3][(pj+1) % nvp3], p4, p3, p1);
    insert (p1, p4, p3, adj[p3][(pj+1) % nvp3]);
   }
  }
 }
}

/****************************** Connect2 ******************************/

connect2 (p1)
 int p1;
{
 int p2, p3, pj, nvp2;

 /* connect new point to two existing vertices
   and update adjacency info */

 p2 = insert_list[p1][1];
 p3 = insert_list[p1][2];
 nvp2 = nvertex[p2];
 for (pj=0; pj<nvp2; pj++) {
  if (adj[p2][pj] == p1) {
   if (!boundary[p1] || pj>0) {
    insert (adj[p2][(pj-1+nvp2) % nvp2], p2, p3, p1);
    insert (p1, p2, p3, adj[p2][(pj-1+nvp2) % nvp2]);
   }
   if (!boundary[p1] || pj<nvp2-1) {
    insert (adj[p2][(pj+1) % nvp2], p2, p3, p1);
    insert (p1, p2, p3, adj[p2][(pj+1) % nvp2]);
   }
  }
 }
}

/****************************** Choice ******************************/

choice (p1, side)
 int p1, side;
```

```
{
 int p2, p3, p4, p5, smallest;

 /* choice of two existing vertices to connect to,
   pick vertex that creates largest angle */

 find_connect (p1, side, &p2, &p3, &p4, &p5);
 if (p2 > p1) {
  smallest = select1 (p1, p2, p5, p4);
  if (smallest) {
   insert (p1, p2, p4, p5);
   insert (p5, p2, p4, p1);
  }
  else {
   insert (p2, p1, p5, p4);
   insert (p4, p1, p5, p2);
  }
 }
}

/******************************** Odd Man ********************************/

int odd_man (p1)
 int p1;
{
 int p2[4], p3, p4, p5[4], pi;

 for (pi=1; pi<=3; pi++) {
  find_connect (p1, pi, &p2[pi], &p3, &p4, &p5[pi]);
 }
 if (p5[1] == p5[2]) return 3;
 else if (p5[1] == p5[3]) return 2;
 else if (p5[3] == p5[2]) return 1;
}

/******************************** Dot Product ********************************/

double dot_product (p1, p2, p3)
 int  p1, p2, p3;
{
 double  xdiff1, xdiff2, ydiff1, ydiff2;

 /* compute two-dimensional dot product */

 xdiff1 = xworld[p1] - xworld[p2];
 ydiff1 = yworld[p1] - yworld[p2];
 norm2 (&xdiff1, &ydiff1);
```

```
 xdiff2 = xworld[p1] - xworld[p3];
 ydiff2 = yworld[p1] - yworld[p3];
 norm2 (&xdiff2, &ydiff2);

 return (xdiff1*xdiff2 + ydiff1*ydiff2);
}

/****************************** Select1 ******************************/

int select1 (p1, p2, p3, p4)
 int p1, p2, p3, p4;
{
 double small, cosine;

 /* select largest angle, smallest dot product */

 small = dot_product (p1, p4, p3);

 cosine = dot_product (p1, p2, p3);
 if (cosine > small) small = cosine;

 cosine = dot_product (p3, p2, p1);
 if (cosine > small) small = cosine;

 cosine = dot_product (p3, p4, p1);
 if (cosine > small) small = cosine;

 cosine = dot_product (p2, p4, p1);
 if (cosine > small) return 1;

 cosine = dot_product (p2, p4, p3);
 if (cosine > small) return 1;

 cosine = dot_product (p4, p2, p3);
 if (cosine > small) return 1;

 cosine = dot_product (p4, p2, p1);
 if (cosine > small) return 1;

 return 0;
}

/****************************** Insert ******************************/

insert (p1, p2, p3, p4)
 int p1, p2, p3, p4;
```

```
{
 int l, m, nv;

 /* insert a new vertex between two others
   in the adjacency list */

 nv = nvertex[p1];
 for (l=0; l<nv; l++) {
  if (adj[p1][l] == p2) {
   if (boundary[p1]) {
    if (l > 0 && adj[p1][(l-1)] == p3) l--;
   }
   else {
    if (adj[p1][(l-1+nv) % nv] == p3) l--;
   }
   for (m=nv; m>l+1; m--) {
    adj[p1][m] = adj[p1][m-1];
   }
   adj[p1][l+1] = p4;
   nvertex[p1]++;
   return;
  }
 }
}

/****************************** Replace *******************************/

replace (p1, p2, p3)
 int p1, p2, p3;
{
 int pj;

 /* replace one vertex by another
   in the adjacency list */

 for (pj=0; pj<nvertex[p1]; pj++) {
  if (adj[p1][pj] == p2) {
   adj[p1][pj] = p3;
   return;
  }
 }
}
```

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <string.h>

#define screen_scale 10.
#define screen_size 900.

extern double xndc();
extern double yndc();

int    n, i, j, k, ipoint, total_points, ssteps, tsteps, even;
double Bez[3][20][4], coord, ssize, tsize, min,
       xnormal, ynormal, znormal, s_deriv[3], t_deriv[3],
       screen_factor = screen_size/screen_scale;

FILE   *stream;

/****************************** Main ******************************/

main (argc, argv)
 int   argc;
 char **argv;
{
  /* triangulate individual teapot parts */

  if (argc > 1) {
   if (strcmp(argv[1], "b") == 0) {
    stream = fopen("body.sweep", "r");
    input ();
    stream = fopen("body.data", "w");
    output ();
   }

   if (strcmp(argv[1], "l") == 0) {
    stream = fopen("lid.sweep", "r");
    input ();
    stream = fopen("lid.data", "w");
    output ();
   }

   if (strcmp(argv[1], "h") == 0) {
    stream = fopen("handle.sweep", "r");
    input ();
    stream = fopen("handle.data", "w");
    output ();
   }
```

```
  if (strcmp(argv[1], "s") == 0) {
   stream = fopen("spout.sweep", "r");
   input ();
   stream = fopen("spout.data", "w");
   output ();
  }
 }

 /* triangulate the whole teapot */

 else {
  stream = fopen("body.sweep", "r");
  input ();
  stream = fopen("body.data", "w");
  output ();

  stream = fopen("lid.sweep", "r");
  input ();
  stream = fopen("lid.data", "w");
  output ();

  stream = fopen("handle.sweep", "r");
  input ();
  stream = fopen("handle.data", "w");
  output ();

  stream = fopen("spout.sweep", "r");
  input ();
  stream = fopen("spout.data", "w");
  output ();
 }
}

/****************************** Input *******************************/

input ()
{
 /* read step sizes */

 fscanf (stream, "%d%d", &ssteps, &tsteps);
 ssize = 1.0 / (double)(ssteps-1);
 tsize = 1.0 / (double)(tsteps-1);

 /* read number of patches */

 fscanf (stream, "%d", &n);
```

```
 /* read Bezier control points */

 for (i=0; i<n; i++) {
  for (j=0; j<4; j++) {
   for (k=0; k<3; k++) {
    fscanf (stream, "%lf", &Bez[k][i][j]);
   }
  }
 }
}

/****************************** Output ******************************/

output ()
{
 double s, t, skip, sing;

 /* output total number of points */

 total_points = (n-1)/3 * ssteps * tsteps - (n-4)/3 * tsteps;
 fprintf (stream, "%d0, total_points);

 ipoint = 0;
 min = 0.0;

 /* for each patch */

 for (i=0;i<n-3;i=i+3) {
  if (i > 0) min = ssize;

  /* step in s direction */

  for (s=min; s<1.01; s=s+ssize) {
   fprintf (stream, "0);
   j = 0;

   /* singular point at top of lid */

   if (Bez[0][i][0] == 0.0 && s == 0.0) {

    skip = 2. * ssize / (double)(tsteps + 1);
    for (sing=ssize-skip; sing>-ssize+0.0001; sing=sing-skip) {
     if (sing>0.0) point ( sing, 0.0);
     else       point (-sing, 1.0);
    }
   }
```

```
    else {

     /* step in t direction */

     for (t=0.0; t<1.01; t=t+tsize) {
      point (s, t);
     }
    }
   }
  }
 }
}

/****************************** Point ******************************/

point (s, t)
   double s, t;
{
 fprintf (stream, "%d", ipoint);
 even = (j++ + 1) % 2;

 /* determine coordinates */

 for (k=0; k<2; k++) {
  coord =   (1.-s)*(1.-s)*(1.-s)*(1.-t)*(1.-t)*(1.-t)*Bez[k][i  ][0] +
   3.*s    *(1.-s)*(1.-s)*(1.-t)*(1.-t)*(1.-t)*Bez[k][i+1][0] +
    3.*s    *s    *(1.-s)*(1.-t)*(1.-t)*(1.-t)*Bez[k][i+2][0] +
     s    *s    *s    *(1.-t)*(1.-t)*(1.-t)*Bez[k][i+3][0] +

      3.*(1.-s)*(1.-s)*(1.-s)*t    *(1.-t)*(1.-t)*Bez[k][i  ][1] +
       9.*s    *(1.-s)*(1.-s)*t    *(1.-t)*(1.-t)*Bez[k][i+1][1] +
        9.*s    *s    *(1.-s)*t    *(1.-t)*(1.-t)*Bez[k][i+2][1] +
         3.*s    *s    *s    *t    *(1.-t)*(1.-t)*Bez[k][i+3][1] +

          3.*(1.-s)*(1.-s)*(1.-s)*t    *t    *(1.-t)*Bez[k][i  ][2] +
           9.*s    *(1.-s)*(1.-s)*t    *t    *(1.-t)*Bez[k][i+1][2] +
            9.*s    *s    *(1.-s)*t    *t    *(1.-t)*Bez[k][i+2][2] +
             3.*s    *s    *s    *t    *t    *(1.-t)*Bez[k][i+3][2] +

              (1.-s)*(1.-s)*(1.-s)*t    *t    *t    *Bez[k][i  ][3] +
               3.*s    *(1.-s)*(1.-s)*t    *t    *t    *Bez[k][i+1][3] +
                3.*s    *s    *(1.-s)*t    *t    *t    *Bez[k][i+2][3] +
                 s    *s    *s    *t    *t    *t    *Bez[k][i+3][3];

  if (k == 0) fprintf (stream, " %lf", xndc(coord));
  else        fprintf (stream, " %lf", yndc(coord));

 }
```

```
if (t < 0.01 || t > 0.99 || (i == 0 && s < 0.01) || (i == n-4 && s > 0.99))
 fprintf (stream, " 1");
else fprintf (stream, " 0");

/* calculate partial derivatives */

for (k=0; k<3; k++) {
 s_deriv[k] = -3.*(1.-s)*(1.-s)          *(1.-t)*(1.-t)*(1.-t)*Bez[k][i  ][0] +
  3.*(-2.*s*(1.-s) + (1.-s)*(1.-s))*(1.-t)*(1.-t)*(1.-t)*Bez[k][i+1][0] +
   3.*(-s*s + 2.*s*(1.-s))       *(1.-t)*(1.-t)*(1.-t)*Bez[k][i+2][0] +
    3.*s*s                  *(1.-t)*(1.-t)*(1.-t)*Bez[k][i+3][0] +

     3.* -3.*(1.-s)*(1.-s)        *t   *(1.-t)*(1.-t)*Bez[k][i  ][1] +
      9.*(-2.*s*(1.-s) + (1.-s)*(1.-s))*t   *(1.-t)*(1.-t)*Bez[k][i+1][1] +
       9.*(-s*s + 2.*s*(1.-s))       *t   *(1.-t)*(1.-t)*Bez[k][i+2][1] +
        3.*3.*s*s                *t   *(1.-t)*(1.-t)*Bez[k][i+3][1] +

         3.* -3.*(1.-s)*(1.-s)        *t   *t   *(1.-t)*Bez[k][i  ][2] +
          9.*(-2.*s*(1.-s) + (1.-s)*(1.-s))*t   *t   *(1.-t)*Bez[k][i+1][2] +
           9.*(-s*s + 2.*s*(1.-s))       *t   *t   *(1.-t)*Bez[k][i+2][2] +
            3.*3.*s*s                *t   *t   *(1.-t)*Bez[k][i+3][2] +

             -3.*(1.-s)*(1.-s)        *t   *t   *t   *Bez[k][i  ][3] +
              3.*(-2.*s*(1.-s) + (1.-s)*(1.-s))*t   *t   *t   *Bez[k][i+1][3] +
               3.*(-s*s + 2.*s*(1.-s))       *t   *t   *t   *Bez[k][i+2][3] +
                3.*s*s                *t   *t   *t   *Bez[k][i+3][3];

 t_deriv[k] =  (1.-s)*(1.-s)*(1.-s)* -3.*(1.-t)*(1.-t)*Bez[k][i  ][0] +
  3.*s   *(1.-s)*(1.-s)* -3.*(1.-t)*(1.-t)*Bez[k][i+1][0] +
   3.*s   *s   *(1.-s)* -3.*(1.-t)*(1.-t)*Bez[k][i+2][0] +
    s   *s   *s   * -3.*(1.-t)*(1.-t)*Bez[k][i+3][0] +

     3.*(1.-s)*(1.-s)*(1.-s)*(-2.*t*(1.-t) + (1.-t)*(1.-t))*Bez[k][i  ][1] +
      9.*s   *(1.-s)*(1.-s)*(-2.*t*(1.-t) + (1.-t)*(1.-t))*Bez[k][i+1][1] +
       9.*s   *s   *(1.-s)*(-2.*t*(1.-t) + (1.-t)*(1.-t))*Bez[k][i+2][1] +
        3.*s   *s   *s   *(-2.*t*(1.-t) + (1.-t)*(1.-t))*Bez[k][i+3][1] +

         3.*(1.-s)*(1.-s)*(1.-s)*(-t*t + 2.*t*(1.-t))*Bez[k][i  ][2] +
          9.*s   *(1.-s)*(1.-s)*(-t*t + 2.*t*(1.-t))*Bez[k][i+1][2] +
           9.*s   *s   *(1.-s)*(-t*t + 2.*t*(1.-t))*Bez[k][i+2][2] +
            3.*s   *s   *s   *(-t*t + 2.*t*(1.-t))*Bez[k][i+3][2] +

             (1.-s)*(1.-s)*(1.-s)* 3.*t*t   *Bez[k][i  ][3] +
              3.*s   *(1.-s)*(1.-s)* 3.*t*t   *Bez[k][i+1][3] +
               3.*s   *s   *(1.-s)* 3.*t*t   *Bez[k][i+2][3] +
                s   *s   *s   * 3.*t*t   *Bez[k][i+3][3];
```

```
}

/* calculate surface normal */

xnormal = -s_deriv[1]*t_deriv[2] + s_deriv[2]*t_deriv[1];
ynormal =  s_deriv[0]*t_deriv[2] - s_deriv[2]*t_deriv[0];
znormal = -s_deriv[0]*t_deriv[1] + s_deriv[1]*t_deriv[0];
normalize (&xnormal, &ynormal, &znormal);

/* singular point at top of lid */

if (Bez[0][i][0] == 0.0 && s == 0.0)
 fprintf (stream, " %lf %lf %lf", 0.0, 1.0, 0.0);
else
 fprintf (stream, " %lf %lf %lf", xnormal, ynormal, znormal);

/* determine adjacencies */

/* top right corner */

if (ipoint == 0) fprintf (stream, " 3 1 %d %d0, tsteps+1, tsteps);

/* top left corner */

else if (ipoint == total_points-1) {
 if (even) fprintf (stream, " 2 %d %d0, ipoint-1, ipoint-tsteps);
 else     fprintf (stream, " 3 %d %d %d0, ipoint-1, ipoint-tsteps-1, ipoint-tsteps);
}

/* bottom left corner */

else if (ipoint == tsteps-1) {
 if (even) fprintf (stream, " 3 %d %d %d0, ipoint-1, 2*tsteps-2, 2*tsteps-1);
 else     fprintf (stream, " 2 %d %d0, ipoint-1, 2*tsteps-1);
}

/* bottom right corner */

else if (ipoint == total_points-tsteps) fprintf (stream, " 2 %d %d0, ipoint+1, ipoint-tsteps);

/* top edge */

else if (s < ssize - 0.0001) {
 if (even) fprintf (stream, " 5 %d %d %d %d %d0, ipoint-1, ipoint+tsteps-1,
            ipoint+tsteps, ipoint+tsteps+1, ipoint+1);
 else     fprintf (stream, " 3 %d %d %d0, ipoint-1, ipoint+tsteps, ipoint+1);
}
```

```
 /* bottom edge */

 else if (i == n-4 && s > 0.99) {
  if (even) fprintf (stream, " 3 %d %d %d0, ipoint-1, ipoint-tsteps, ipoint+1);
  else     fprintf (stream, " 5 %d %d %d %d %d0, ipoint-1, ipoint-tsteps-1,
               ipoint-tsteps, ipoint-tsteps+1, ipoint+1);
 }

 /* right edge */

 else if (t < 0.01) fprintf (stream, " 4 %d %d %d %d0, ipoint-tsteps, ipoint+1,
                   ipoint+tsteps+1, ipoint+tsteps);

 /* left edge */

 else if (t > 0.99) {
  if (even) fprintf (stream, " 4 %d %d %d %d0, ipoint-tsteps, ipoint-1,
               ipoint+tsteps-1, ipoint+tsteps);
  else     fprintf (stream, " 4 %d %d %d %d0, ipoint-tsteps, ipoint-tsteps-1,
               ipoint-1, ipoint+tsteps);
 }

 /* interior point */

 else {
  if (even) fprintf (stream, " 6 %d %d %d %d %d %d0, ipoint-1, ipoint-tsteps,
               ipoint+1, ipoint+tsteps+1, ipoint+tsteps, ipoint+tsteps-1);
  else     fprintf (stream, " 6 %d %d %d %d %d %d0, ipoint-1, ipoint-tsteps-1,
               ipoint-tsteps, ipoint-tsteps+1, ipoint+1, ipoint+tsteps);
 }

 ipoint++;
}

/****************************** Normalize ******************************/

normalize(xvector, yvector, zvector)
   double *xvector,  *yvector,  *zvector;
{
 double tot;

 /* normalize to unit vector */

 tot = sqrt(*xvector*(*xvector)+*yvector*(*yvector)+*zvector*(*zvector));
 *xvector = *xvector/tot;
 *yvector = *yvector/tot;
 *zvector = *zvector/tot;
```

```
}

/****************************** XNDC ******************************/

double xndc (x)
   double x;
{
 return ((x + 3.) * screen_factor);
}

/****************************** YNDC ******************************/

double yndc (y)
   double y;
{
 return ((-y + 3.) * screen_factor);
}
```

Teapot Data

```
/****************************** Body ******************************/

2 4

7

1.5000 2.25000 0.0
1.5000 2.25000 1.5000
-1.5000 2.25000 1.5000
-1.5000 2.25000 0.0

1.7500 1.72500 0.0
1.7500 1.72500 1.7500
-1.7500 1.72500 1.7500
-1.7500 1.72500 0.0

2.0000 1.20000 0.0
2.0000 1.20000 2.0000
-2.0000 1.20000 2.0000
-2.0000 1.20000 0.0

2.0000 0.75000 0.0
2.0000 0.75000 2.0000
-2.0000 0.75000 2.0000
-2.0000 0.75000 0.0

2.0000 0.30000 0.0
2.0000 0.30000 2.0000
-2.0000 0.30000 2.0000
-2.0000 0.30000 0.0

1.5000 0.07500 0.0
1.5000 0.07500 1.5000
-1.5000 0.07500 1.5000
-1.5000 0.07500 0.0

1.5000 0.00000 0.0
1.5000 0.00000 1.5000
-1.5000 0.00000 1.5000
-1.5000 0.00000 0.0

/****************************** Handle ******************************/

7 9

7
```

```
-1.60 1.8750 0.0
-1.60 1.8750 0.3
-1.50 2.1000 0.3
-1.50 2.1000 0.0

-2.30 1.8750 0.0
-2.30 1.8750 0.3
-2.50 2.1000 0.3
-2.50 2.1000 0.0

-2.70 1.8750 0.0
-2.70 1.8750 0.3
-3.00 2.1000 0.3
-3.00 2.1000 0.0

-2.70 1.6500 0.0
-2.70 1.6500 0.3
-3.00 1.6500 0.3
-3.00 1.6500 0.0

-2.70 1.4250 0.0
-2.70 1.4250 0.3
-3.00 1.2000 0.3
-3.00 1.2000 0.0

-2.50 0.9750 0.0
-2.50 0.9750 0.3
-2.65 0.7875 0.3
-2.65 0.7875 0.0

-2.00 0.7500 0.0
-2.00 0.7500 0.3
-1.90 0.4500 0.3
-1.90 0.4500 0.0

/****************************** Lid ******************************/

9 9

7

0.0 3.00 0.0
0.0 3.00 0.0
0.0 3.00 0.0
0.0 3.00 0.0

0.8 3.00 0.0
```

Teapot Data

```
0.8 3.00 0.8
-0.8 3.00 0.8
-0.8 3.00 0.0

0.0 2.70 0.0
0.0 2.70 0.0
0.0 2.70 0.0
0.0 2.70 0.0

0.2 2.55 0.0
0.2 2.55 0.2
-0.2 2.55 0.2
-0.2 2.55 0.0

0.4 2.40 0.0
0.4 2.40 0.4
-0.4 2.40 0.4
-0.4 2.40 0.0

1.3 2.40 0.0
1.3 2.40 1.3
-1.3 2.40 1.3
-1.3 2.40 0.0

1.3 2.25 0.0
1.3 2.25 1.3
-1.3 2.25 1.3
-1.3 2.25 0.0

/****************************** Spout ******************************/

13 11

4

1.700 1.27500 0.0
1.700 1.27500 0.66
1.700 0.45000 0.66
1.700 0.45000 0.0

2.600 1.27500 0.0
2.600 1.27500 0.66
3.100 0.67500 0.66
3.100 0.67500 0.0

2.300 1.95000 0.0
2.300 1.95000 0.25
```

Teapot Data

2.400 1.87500 0.25
2.400 1.87500 0.0

2.700 2.25000 0.0
2.700 2.25000 0.25
3.300 2.25000 0.25
3.300 2.25000 0.0

Distribution List for IDA Paper P-2354

NAME AND ADDRESS	NUMBER OF COPIES
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
IDA	
General W.Y. Smith, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Dr. Michael R. Kappel. CSED	2
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	2
Dr. Richard Wexelblat, CSED	1
IDA Control & Distribution Vault	2