UniQuadrix

Eugene Kenneth Ressler

The Unigrafix Group
Professor Carlo Séquin
Computer Science Division
University of California at Berkeley
Berkeley, California 94720

ABSTRACT

UniQuadrix is a simple graphics modeling program for objects represented as the intersection of quadric and planar half-spaces. It runs under the 4.2 BSD UNIX† operating system on the DEC VAX superminicomputer and the M68000-based Sun Workstation. The program accepts scene descriptions in a language much like that of *Unigrafix*, a polygon-based modeler developed at UC Berkeley, and generates black-and-white, smooth-shaded, smooth-edged images on several output devices. UniQuadrix uses implicit equations to represent the surfaces and boundaries of objects throughout the rendering process. This allows a "scan line" hidden surface algorithm to efficiently identify visible "spans". The algorithm exploits scan line as well as object coherence. An efficient incremental algorithm shades pixels within spans. Images of one hundred half-spaces with one-million shaded pixels commonly require three minutes of VAX time.

† UNIX is a trademark of Bell Laboratories.

Acknowledgements

This work merely focuses the knowledge and caring of those around me. My wife, Freddie, and kids, Daniel and Julia, were supportive beyond reason. Professor Carlo Séquin, my faculty advisor, had the original ideas and pushed me forward when things looked bleak. Fred Obermeier was a friend and doubled as a UNIX expert. The members of the Unigrafix group provided advice and encouragement on countless occasions. In particular, Nachshon Gal's insight on hidden surface algorithms saved many hours. Thanks also to Professor Brian Barsky, who generously agreed to review this paper in the face of many other commitments.

My initial investigations in the world of quadric equations were aided greatly by VAXIMA, the computer algebra system built and maintained by the symbolic computation group headed by Professor Richard Fateman at UC Berkeley.

The final UniQuadrix code for the DEC VAX contains fast versions of the C library square root written by members of the elementary functions group directed by Professor W. Kahan.

Support for this work was provided in part by the Semiconductor Research Cooperative under grant number SRC-82-11-008.

Table of Contents

1. Introduction

Quadric equations and the surfaces they represent lend themselves well to computer graphics applications. Many of the sub-problems that must be solved to model and render them efficiently have usable closed-form solutions. For example, equation coefficients may be transformed with homogeneous matrices to orient a surface in space. A single homogeneous vector-matrix multiplication supplies the quadric surface normal. Even tracing general quadric intersections, in general projective curves of degree four, has been reduced to a second degree problem by Levin [Levin 76, 77]. Certainly, such a powerful facility merits attention.

Quadrics *have* received attention, though their history in computer graphics is a great enigma. On one hand, they are used as primitives in some of the earliest attempts at geometric modeling [Weiss 66]. This is to be expected; a student painter's first exercise is to produce the image of light falling on a sphere or cone exactly because variations on these forms are prevalent in nature. Quadrics seem even more intrinsic to man-made objects; developers of Computer Aided Design systems have found that spheres, cones, and planes are sufficient to model ninety per cent of the parts in a modern American automobile [Hakala 81].

On the other hand, the complexity and potential for numerical instability of general purpose algorithms for manipulating quadrics are well known. Perhaps this is why rendering of curved surfaces is commonly done using planar polygons for approximation [Gouraud 71].

None the less, researchers after Weiss have continued to seek merit in the idea that an intermediate, approximating object representation is not necessary for efficient rendering. The work of Woon [Woon 71] is closely related to the algorithms of UniQuadrix. He demonstrated that an algorithm designed to draw the edges of polygonal objects with hidden lines removed can be used to draw quadric objects with minimal modification.

This work exploits Woon's idea with a hidden *surface* algorithm. *UniQuadrix* is a simple graphics modeler for objects represented as the intersection of quadric and planar half-spaces (*bodies*). It runs under the UNIX 4.2 BSD operating system on the DEC VAX super-minicomputer and the M68000-based Sun Workstation. The program accepts scene descriptions in a language much like that of *Unigrafix* [Séquin 83] and generates black-and-white, smooth-shaded, smooth-edged images on several output devices. These include the Sun's bit-mapped screen. UniQuadrix uses implicit equations to represent the surfaces and boundaries of bodies throughout the rendering process. This allows a 'scan line' hidden surface algorithm to efficiently identify visible 'spans' in a scene. The algorithm exploits scan line as well as object coherence. An efficient, incremental algorithm shades pixels within spans.

The system described here is fully implemented and working at UC Berkeley. However, current UniQuadrix handles only objects with plane-quadric intersections and plane-plane intersections. Extension to include quadric-quadric intersections is a topic for future study.

UniQuadrix is written in about 10,000 lines of the C language divided among forty-six source modules. The input parser was built with the Yet Another Compiler Compiler (*yacc*) facility of UNIX [Johnson 78]. The grammar and actions input to yacc account for another 800 source lines.

Chapter 2 explains how one uses UniQuadrix, for the scene language introduces most clearly what is required of the rendering algorithms. The following Chapters describe the steps of scene processing. Chapter 3 covers the parsing of scene files and construction of

the scene data structure. Chapter 4 describes the rather complicated algorithm used to compute boundaries. Chapter 5, describes the hidden surface algorithm, drawn directly from the "stack" algorithm of Unigrafix [Strauss 82]. Chapter 6 details the shading algorithm of UniQuadrix. Finally, Chapter 7 examines the performance of UniQuadrix and draws conclusions about this work.

2. Using UniQuadrix

2.1. The Language

The UniQuadrix language is modeled on that of Unigrafix with minor exceptions. It is a terse, yet readable way of describing scenes.

2.1.1. Half-Space Bounds

The building block of UniQuadrix scenes is the Half-Space-Bound (HSB). Each HSB is potentially two things — a zero-thickness surface, possibly of infinite extent that is shaded in the final image and a bound on the surfaces of other HSB's.

The language allows two types of HSB's — planar ones:

plane [*PlaneId*] < A_p B_p C_p D_p > [=t][=o];

and quadric ones:

quadric [*QuadricId*] <A_q B_q C_q D_q E_q F_q G_q H_q J_q K_q> [=t][=o];

Id's in UniQuadrix are strings of letters, numbers, sharps (#), and underscores (_), though the first character may not be a number. Only the semicolon (;) is honored as a terminator, so statements may share a line, and spaces, tabs, and newlines (white space) may be used freely to enhance readability.

The coefficients in the HSB statement describe the half-space inequality (HSI) that gives the HSB its surface and bounding characteristics. The HSI's of the statements above are:

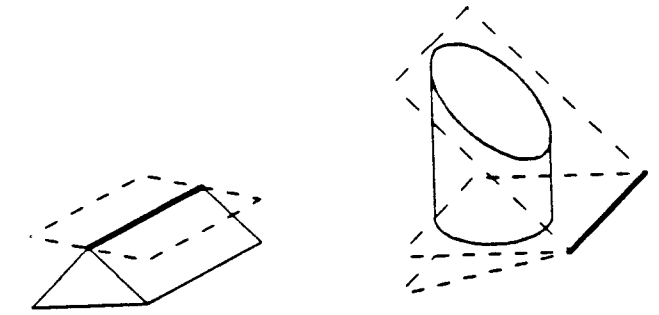
$$P(x, y, z) = A_p x + B_p y + C_p z + D_p \leq 0 \tag{2.1}$$

and

$$\begin{aligned} Q(x, y, z) = {} & A_q x^2 + B_q y^2 + C_q z^2 \\ & + 2D_q xy + 2E_q yz + 2F_q xz \\ & + 2G_q x + 2H_q y + 2J_q z + K_q \leq 0 \end{aligned} \tag{2.2}$$

Option =o makes an HSB opaque and =t, transparent. Opaque is default. The surface of an opaque HSB is shaded, and its HSI potentially bounds other HSB surfaces in the same body. Transparent HSB's have only the latter quality.

UniQuadrix generally assumes that quadric HSIs are defined such that the portion included in any body is convex, though it will work for some cases that do not meet this condition. In particular the algorithm handles hyperboloids of one sheet defined as "necked" cylinders. It is dangerous to include the apex of a cone within a body. In addition, *redundant intersections* are illegal. A redundant intersection results when three or more HSBs of the same body intersect in the same three-space curve (point intersections are allowed). Examples of legal and illegal intersections are shown in Figure 2.1. This rule does not restrict the topology of bodies, because any redundant intersection may be eliminated without changing the appearance of the final image. Hence, the rule is important mainly to writers of programs that generate UniQuadrix input.



Illegal because intersection occurs in final image

Legal if excluded from final image

Figure 2.1 — Redundant Intersections

HSB statements are essentially macro definitions. Nothing is rendered unless an HSB is named in a body statement as discussed in the next section.

2.1.2. Bodies

A body statement lists HSB's:

body [*bodyId*] ([[!] *planeId*] [[!] *quadricId*] . . .);

In the final image, the body appears as the visible portions of its opaque HSB surfaces that satisfy all its HS inequalities. Prefix "!" denotes the complement of the named HSB, including its surface. Thus, a quadric HSB can be a "hole" even though its HSI is convex. For example, a tube with inside radius 1, outside radius 2, and length 4 on the y axis is:

```
quadric outer_tube <1 0 1 0 0 0 0 0 0 -4>;
quadric inner_tube <1 0 1 0 0 0 0 0 0 -1>;
plane top_surf <0 1 0 -2>;
plane bot_surf <0 1 0 2>;
body tube <outer_tube !inner_tube top_surf !bot_surf>;
```

2.1.3. Definitions

Definitions are the basis of hierarchical scene descriptions, exactly as in Unigrafix. They, like HSBs, describe objects that are not rendered unless named elsewhere. Planes, quadrics, and bodies, as well as the instances and arrays to be described may be included in definitions:

```
definition [ defId ];
    [ plane . . . ]
    [ quadric . . . ]
    [ body . . . ]
    [ instance . . . ]
    [ array . . . ]
    .
    .
    .

enddef;
```

The id's of plane and quadric HSB's given within a definition are local to that definition. Likewise, bodies may reference only HSB's given within the same definition.

2.1.4. Instances and Arrays

Instances and arrays are used to render the contents of a definition in one or several copies:

instance [*instId*] (*defId* [*transforms*]);

array [*arrayId*] (*defId* [*initTransforms*])
[*numCopies*] [*deltaTransforms*];

Transforms consist of one or more of the following transformations separated by white space:

=tx	*number*	Translate *number* in x, y, z axis or all axes
=ty	*number*	
=tz	*number*	
=ta	*number*	
=sx	*number*	Scale by *number* on x, y, z axis or all axes
=sy	*number*	
=sz	*number*	
=sa	*number*	
=rx	*number*	Rotate by *number* degrees on given axis
=ry	*number*	(right-hand convention applies)
=rz	*number*	
=vxy	*number*	Shear *number* along the second axis
=vxz	*number*	for each unit along the first axis
=vyx	*number*	
=vyz	*number*	
=vzx	*number*	
=vzy	*number*	

Transformations are concatenated left to right and used to orient the definition named *defId* as it is rendered. For instances, *transforms* give the orientation of a single copy of *defId*. For arrays, *initTransforms* give the orientation of the first of *numCopies* of *defId* and *deltaTransforms* describe the change in orientation between copies. A bundle of six of the tubes described above is:

```
definition tube;
    statements for body tube above
enddef;
array bundle_of_tubes (tube =tx 2) 6 =ry 60;
```

Hierarchies of objects are created by naming instances and arrays within definitions. A stack of three tube bundles is:

```
definition bundle_of_tubes;
    array statement above
enddef;
array stack_of_bundles (bundle_of_tubes =ty -7) 3 =ty 5;
```

2.1.5. Lights

UniQuadrix supports multiple light sources in a manner similar to Unigrafix. Isotropic, ambient lights are specified with:

light [*lightId*] *intensity*;

Parallel lights have the form:

light [*lightId*] *intensity x y z*;

Vector [*x y z*] is the direction *to* a parallel light. *Intensity* of both types of lights is expected to be in the range zero to one. UniQuadrix truncates values outside this range.

Multiple parallel sources may be defined, however, only one, the last by default, is used for rendering. This is a practical restriction; the smooth shading algorithm runs in time proportional to the number of parallel lights.

A very approximate constant shading of curved surfaces is invoked with the -f option in the command line or the **show fast** statement as described below. This shortens computation time while providing an image useful for debugging images and adjusting view geometry.

Uniquadrix automatically creates a light that is used if no other parallel light is specified:

```
light default 1 0 0 -1;
```

The sum of all ambient intensities plus a portion of the parallel intensity given by Lambert's Law is used to shade the surfaces of each body. Areas where this sum is one or more are rendered at maximum device intensity.

2.1.6. Views

UniQuadrix treats view specifications as objects. A scene may contain any number of views, but only one is used for rendering. Again, this defaults to the last one defined. Views are specified as in the *Simple Graphics Package* of Foley and VanDam [Foley 83]:

view [*viewId*] *viewParams*;

ViewParams consist of one or more of the following:

vrp	*x y z*	View Reference Point
vpn	*x y z* [=v] [=p]	View Plane Normal Vector
vup	*x y z* [=v] [=p]	View 'Up' Vector
dop	*x y z* [=v] [=p]	Direction of Parallel Projection
cop	*x y z* [=a] [=r]	Center of Perspective Projection
whh	*size*	Vertical half-size of view window (parallel to projection of **vup** on view plane)
whw	*size*	Horizontal half-size of view window (perpendicular to projection of **vup** on view plane)
fbp	*d1 d2*	Front/back clipping plane distance (measured from **vrp** along **vpn**)

Multiple *viewParams* are separated by white space. Option =v means that [*x y z*] is a direction vector (default), =p that [*x y z*] is a point at the tip of a vector from **vrp**. Similarly, =a denotes an absolute **cop** (default) and =r, an offset in world coordinates relative to **vrp**. *ViewParams* are parsed and set left to right, so **vrp** should be given first before parameters with =p and =r. Of repeated parameters, only the last is remembered.

UniQuadrix makes inferences based on the presence and absence of view parameters. The view projection is assumed parallel when **dop** is given and perspective when **cop** is given. If **whh** or **whw** is omitted, the image is translated and scaled to fit the device in height or width respectively. Of course when both are missing, auto-scaling is applied to both axes. Front-back clipping is performed only when **fbp** is given. It has no effect on auto-scaling.

When view parameters are missing the following defaults apply:

Missing	*Default*
vrp	Origin
vpn	Opposite **dop** for parallel projection and opposite the direction from **vrp** to **cop** for perspective
vup	Positive *y* axis
dop and **cop**	Parallel projection with **dop** [.3 .3 -1]

UniQuadrix automatically creates a view that is used when no other is given:

view default dop .3 .3 -1;

This is equivalent to a user view specification in every way.

2.1.7. Read

The statement:

read "*pathName*";

causes the named file to become the source of UniQuadrix input until end of file. Thereafter, statements following the read are processed. Reads may be nested to a depth of 16. The quotes (" ") may be omitted if the file name is acceptable as a UniQuadrix id, with no dot (.), slash (/) or other special characters.

2.1.8. Comments

Comments are enclosed in curly braces ({ }). They are valid anywhere in lieu of white space and may be nested to great depths. The latter facilitates "commenting out"

portions of files that themselves contain comments.

2.1.9. Synonyms

The following synonyms are available for language keywords. All keywords and synonyms are reserved from use as id's:

Keyword	*Synonym*	*Keyword*	*Synonym*
array	a	**light**	l
body	b	**plane**	p
definition	def	**quadric**	q
device	dev	**view**	vw
instance	i	**cop**	ep
read	include	**dop**	ed
enddef	end	**vrp**	vc

2.2. Batch Mode

To process an image in batch mode, the command line is

uq [*options*] [*files*] [*options*] [*files*] . . .

Files are UniQuadrix files. *Files* and *options* are processed in order from left to right. If no file or "–" option is given, the standard input is read. Any error more severe than a warning noted during input processing aborts rendering. If such an error occurs while rendering, "soft" devices (those with screens) merely stop drawing. If a "hard" drawing is being generated, raster files are deleted unless the –k option is in effect.

A complete list of *options* is:

–c*Stmts*	Execute the statements *Stmts*. If *Stmts* are missing, Uniquadrix enters interactive mode; on **quit**, the rest of the command line is processed.
–d*DeviceName*	*DeviceName* is used to draw the scene
–e*PathName*	Redirect error messages to *PathName*
–f	Turn on fast constant shading for debugging
–F	Turn of constant shading after –f
–k	Turn on raster file 'keep'; file name is reported
–K	Turn off raster file 'keep' after –k
–n	No warnings; shuts up the error handler unless something serious happens (good for running in background)
–N	Turn warnings on after –n
–sx*Inches*	Force the drawing to be of size *Inches* across.
–sy*Inches*	Force the drawing to be of size *Inches* in height.
–sd*Count*	Do smooth shading calculation at intervals of *Count* pixels instead of the device default (usually eight).
–t*PathName*	Turn on trace; put trace information in stream *PathName* (stderr if *PathName* is missing).
–T	Turn off trace after –t
–v*ViewName*	Use view *ViewName* instead of the last defined.
–x	Do not render an image after the last file/stdin is read
–	Read the standard input as a file

If no –d option is given, UniQuadrix checks for device names in environment variable UQDEV, then TERM. The last causes a Sun workstation to use it's own screen. The

following *DeviceNames* are known:

a, ascii, file	Human-readable file uq.out
cs, colorsun,	Sun work station with color board and monitor
sun	Sun work station
v, va, var, varian	Benson Varian
x, vx, vectrix	Vectrix
w, vp, ver, versatek	Versatek wide-bodied plotter

Sun and **colorsun** are not supported if UniQuadrix is compiled for the VAX. **Colorsun** requires a Sun workstation with color board and monitor.

2.3. Interactive Mode

UniQuadrix supports a primitive interactive mode that makes it easier to edit a scene. The command line to invoke it is:

uq –**i** [*options*] [*files*] [*options*] [*files*] . . .

After *options* and *files* are processed, the user is prompted with:

uq> _

Any UniQuadrix statement is valid in response to the prompt. If an error is noted, all input is ignored and no prompt given until the next semi-colon (;).

The following statements enhance the interactive environment:

redef light *lightId*;
redef light *lightId intensity*;
redef light *lightId intensity x y z*;
redef view *viewId*;
redef view *viewId viewParams*;
device *deviceName* [**keep**] [**nokeep**] [**xsize** *Inches*] [**ysize** *Inches*];
show [*viewId*] [**fast**];
quit;
read;

In interactive mode there is a notion of the *current* view, parallel light, and device — those that were last defined or **redef**'ed. **Redef** can also assign new values to an object already defined. **Device** sets the current device just as the –d batch option. **Keepon** and **keepoff** are similar to –k and –K in the command line, and the **size** keywords set the drawing size in inches just as –sx and –sy. The statement:

show *ViewId*

is shorthand for:

redef view *ViewId*;
show;

The **fast** keyword is identical to –f in the batch command line. **Quit** quits. These statements are also valid within batch files.

Naked **read** enters interactive mode during batch processing, e.g., to specify a view and device. In this case, **quit** returns control to the batch file.

A typical interactive session might be:

```
% uq -i -dcolorsun;                { start up uq }
uq> read scenefile;                { read a scene }
no user errors
uq> light glow .3;                 { ambient light }
uq> light .7 .4 -.4 -1;            { parallel light }
warning: id l#1 created            { uq makes up id }
uq> view top_down dop 0 1 1        { parallel view }
uq> show;                          { render on color Sun }
uq> dev varian keep;               { current device is Varian }
uq> show fast;                     { show with constant shading }
raster file: /tmp/uq123456
uq> redef light glow 0;            { change ambient to zero }
uq> redef light default;           { make default light current }
uq> show default;                  { show with default view }
raster file: /tmp/uq123457         { keep still on }
uq> quit;
warnings:1
parser:0 syntax:0 input:0 command line:0
%
```

3. Manager, Reader, and Flattener

3.1. Manager

Manager is the caller of subprograms in UniQuadrix. Its jobs are twofold: Handle files and options from the command line and handle the *show* operation that results in a picture. Manager's chief underlings are *Reader*, the processor of batch files and interactive statements, and *Flattener*, *Resolver*, *Renderer*, and *Shader*. These process the scene at the end of a batch command line or when Reader recognizes a **show** statement.

3.2. Reader

Reader scans the content of UniQuadrix files and interactive input and builds scene data structures. This may be on call from Manager, or in response to a **read** statement in the input stream. In the latter case, Reader calls itself, so it is implemented as a recursive function. Scenes are stored in hash tables in a conventional manner; the hash function operates on id's. These data structures contain no information beyond what is in the statements that generate them; Reader does no coordinate transformation.

3.3. Flattener

Flattener begins the show process for Manager by doing a depth-first traversal of the trees of definitions, instances, and arrays in a scene hierarchy. Figure 3.1 shows the tree for *stack_of_bundles* in Chapter 2. Each instance and array in a scene corresponds to an internal nodes in such a tree; body statments outside any definition are just the leaves of trivial trees. All leaf nodes are bodies in the final image.

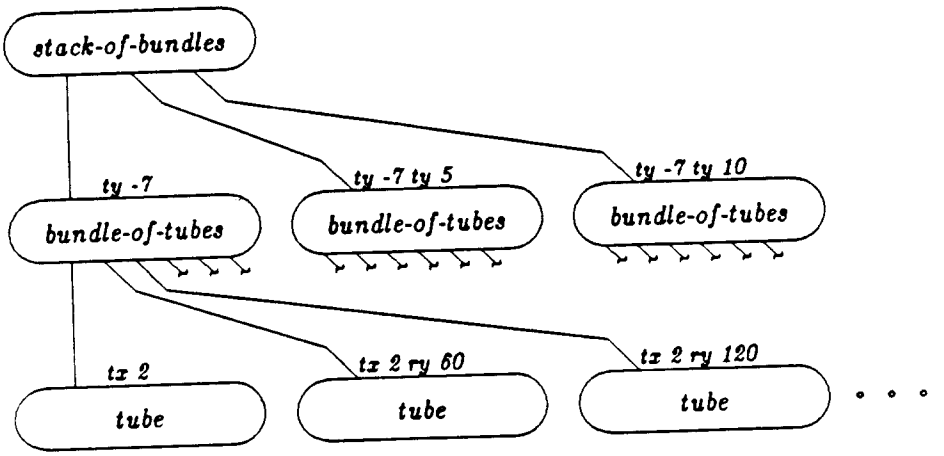


Figure 3.1 — Tree formed by *stack-of-bundles*

As Flattener encounters an internal node, it computes and stores a homogeneous transformation matrix that is the concatenation of all its ancestors' *transforms* plus its own. Figure 3.1 shows the *transforms* stored at the parent as each new node is reached. Note that array *deltaTransforms* accumulate. On reaching a leaf node, Flattener has discovered a body to be rendered. It applies the matrix stored at the parent to the body's HSIs, placing the transformed HSB's in a table that serves as input for Resolver.

Flattener also applies the view transformation matrix. This is computed as part of its initialization and treated as the *transforms* of a pseudo-node at the root of each tree in the scene forrest. The view matrix transforms HSB's in the scene from world coordinates to Resolver Coordinates (RC). It includes a translation, rotation, and potentially a perspective transformation of the scene into a *scene box* as shown in Figure 3.2.

If the user specifies any combination of window width, height, or front-back clipping distance, it is used to fix the x, y, and z size of the box respectively. Clipping is performed by adding planar HSBs that represent appropriate sides of the box to each body. For window axes not given by the user, the image extent gives the box size and no clipping is necessary. Resolver fits the scene box to the device.

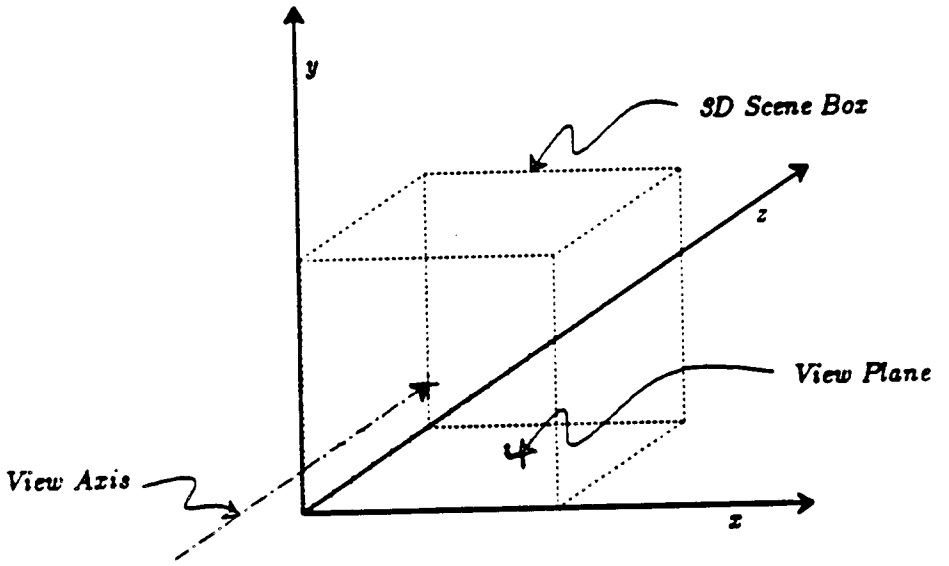


Figure 3.2 — Resolver coordinates

The transformation of HSIs is most easily explained by restating them as matrix equations. The plane equation (1.1) in homogeneous space is:

$$P(x, y, z, w) = A_p x + B_p y + C_p z + D_p w = 0 \qquad (3.1).$$

Let:

$$\mathbf{x} = \begin{bmatrix} x & y & z & w \end{bmatrix} \quad \text{and} \quad \mathbf{P} = \begin{bmatrix} A_p \\ B_p \\ C_p \\ D_p \end{bmatrix} \qquad (3.2).$$

Then, (3.1) becomes:

$$P(\mathbf{x}) = \mathbf{xP} \qquad (3.3).$$

Given a homogeneous transformation matrix T, we wish to obtain $\mathbf{P}' = f(\mathbf{P}, \mathbf{T})$ such that each point x satisfying $\mathbf{xP} = 0$ maps to a point $\mathbf{x}' = \mathbf{xT}$ satisfying $x'\mathbf{P}' = 0$. Straightforward matrix algebra yields:

$$\mathbf{P}' = \mathbf{T}^{-1}\mathbf{P} \qquad (3.4)$$

In a similar manner, the quadric HSI in (1.2) in homogeneous space is:

$$\begin{aligned} Q(x, y, z, w) = {} & A_q x^2 + B_q y^2 + C_q z^2 \\ & + 2D_q xy + 2E_q yz + 2F_q xz \\ & + 2G_q xw + 2H_q yw + 2J_q zw + K_q w^2 = 0 \end{aligned} \tag{3.5}$$

and can be restated:

$$Q(\mathbf{x}) = \mathbf{x}\mathrm{Q}\mathbf{x}^T = 0 \quad \text{where} \quad \mathrm{Q} = \begin{bmatrix} A_q & D_q & F_q & G_q \\ D_q & B_q & E_q & H_q \\ F_q & E_q & C_q & J_q \\ G_q & H_q & J_q & K_q \end{bmatrix} \tag{3.6}.$$

So, the coefficients of the transformed quadric HSI are given by:

$$\mathrm{Q}' = \mathrm{T}^{-1}\,\mathrm{Q}\,(\mathrm{T}^{-1})^T \tag{3.7}.$$

Equations (3.4) and (3.7) influence the design of Flattener. Since they use only T^{-1}, the node transformation matrices discussed above are computed and stored as inverses.

4. Resolver

4.1. A Preview of Renderer

Resolver accepts transformed bodies from Flattener and outputs the data structures used by Renderer. This chapter begins with a partial description of these output data structures.

4.1.1. Show-Faces

To draw planar HSBs, Renderer uses the same structures as would a "scan line" hidden surface algorithm for planar polygons. The *planar show-face* contains a precalculated intensity of reflected light (uniform with our simple lighting model) and a plane equation for depth calculations.

Likewise, Renderer uses *quadric show-faces* to do depth calculations and shading for the curved surfaces of quadric HSBs. Each of two quadric show-faces per HSB corresponds to a portion of surface that is single-valued in z. Thus, each contains the same ten HSI coefficients and a one bit *sheet flag* that labels it as *front* or *back*. Chapter 6 discusses the shading information in quadric show-faces.

The boundary between a quadric HSB's pair of show-faces is called the *limb*. We will show that the limb of a quadric always lies on a plane, the *limb plane*, so that the limb is a conic section. The limb conic may be degenerate. For example, the limb of a sphere is a circle and the limb of a cone is a pair of intersecting lines or degenerates to a point. Figure 4.1(a) highlights the show-faces and limbs of a cylinder and sphere truncated by transparent planar HSB's.

4.1.2. Show-Edges

Show-Edges are used to draw the boundaries between pairs of HSBs in a body. These HSB's are represented by show-faces. Hence, in addition to a description of its own shape, each show-edge contains two pointers to show-faces. Objects in the final image may well have more than two show-faces intersecting in the same three-space line, but such images result from multiple bodies with a common boundary.

Linear show-edges result when two planar HSBs intersect or when a quadric intersects a plane perpendicular to the view plane. End points are sufficient to describe their shape.

Conic show-edges result from general quadric-plane intersections and from limbs. Renderer is able to treat these much like linear show-edges when they satisfy the following rules:

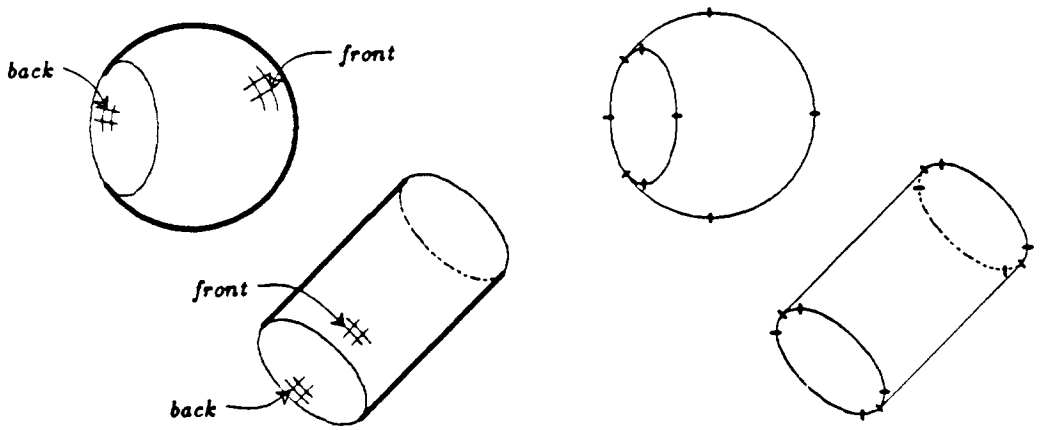
(1) They are single-valued in x.
(2) They are incident to the same show-faces throughout their length.

Figure 4.1(b) marks show-edges for a cylinder and sphere. These satisfy an additional rule imposed as a simplifying design decision:

(3) They are single-valued in y.

Note that (1) and (3) make the conic show-edges of a body dependent on its orientation in the final image.

In addition to its end points, a conic show-edge contains six coefficients to describe its shape. Just as the show-face sheet flag selects one of two values of z for point (x, y),



(a) Show-Faces and Limbs (b) Show-Edges

Figure 4.1 — Renderer structures for quadric HSBs

so a conic show-edge includes a *leg flag* that selects the *left* or *right* value of x for a given y of the edge shape. The value of the leg flag is never ambiguous due to (1).

4.2. Resolver Data Structures

To convert the transformed bodies from Flattener to the show-faces and show-edges of Renderer, Resolver uses private data structures called *i-line tables*, one per body. For a body of n HSBs, HSB_i, $0 \le i \le n-1$, an i-line table is a lower-triangular array of *i-lines* with n rows and columns. $I\text{-}line_{ij}$, $i < j$, $0 \le i \le n-1$, $0 \le j \le n-1$, describes the intersection of HSB_i with HSB_j, and $i\text{-}line_{kk}$, $0 \le k \le n-1$ describes the limb of HSB_k. An i-line table for $n = 6$ is shown in Figure 4.2.

An i-line stores enough information to create show-edges. One field describes the i-line's shape as an array of real numbers. These may be two coordinates of a point, three linear coefficients, or six conic coefficients; a flag field indicates one of these:

POINT	Two planar HSBs intersect and both are perpendicular to the view plane.
LINEAR	Two planar HSBs intersect and at least one of is not perpendicular to the view plane.
CONIC	A quadric intersects a plane that *is not* perpendicular to the view plane. Quadric limb i-lines are also *CONIC*.
EDGE	A quadric intersects a plane that *is* perpendicular to the view plane.
NOEXIST	No intersection curve or limb can be calculated.

I-lines may also have lists of *i-vertices*. An i-vertex is a point on an i-line. I-vertices serve to cut up i-line shapes into pieces that will become show-edges. In this capacity, only their x and y coordinates are of interest. However, since each i-vertex must be tested in its body's HSIs, Resolver also calculates the z coordinate of the point in three-space that projects each i-vertex. The following types of i-vertices are sufficient to delimit show-edges that meet the three criteria above. Figure 4.3 shows examples of each type with labels given in parenthesis below:

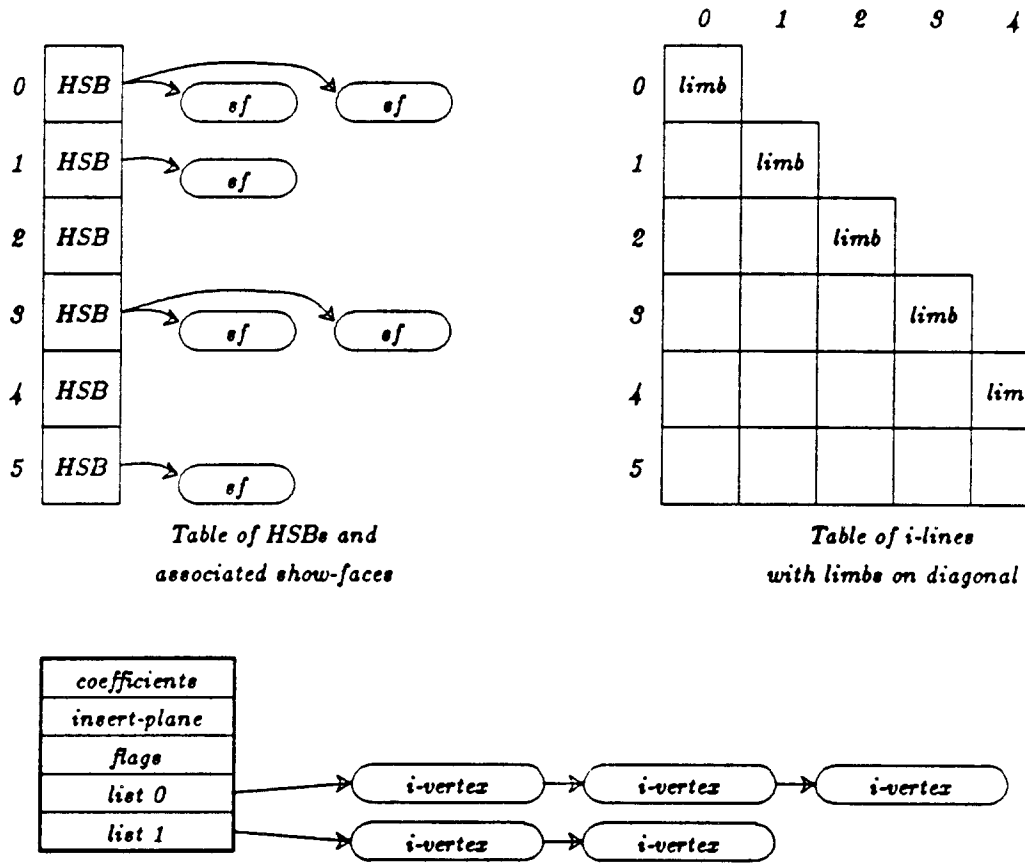


Figure 4.2 — Resolver data structures for a body with six HSBs

HEXTRM (h)	A horizontal extremum of a *CONIC* i-line shape (dx/dy is zero).
VEXTRM (v)	A vertical extremum of a *CONIC* i-line shape (dy/dx is zero).
LIMB (l)	The projection of a planar HSB's intersection with the limb.
ISECT (i)	The projection of the intersection of three HSBs.

Resolver assumes that adjacent pairs of i-vertices on an i-line's list delimit sections of the i-line's shape that should be considered as potential show-edges. To support this assumption, lists are sorted on a key of ascending y, and the i-vertices in a single list fall on sections of i-line shape that are single-valued in x. Hence, *CONIC* i-lines have two lists. These are labeled ***left*** and ***right*** just as the show-edges they will produce. Since an *EDGE* i-line is just a *CONIC* in y and z rather than y and x, *EDGE*s also have two lists corresponding to portions of the shape single valued in z. These are labeled ***front*** and ***back*** just as show-face sheets, and *LIMB* i-vertices are their vertical extrema. *LINEAR* i-lines have just one list. *POINT*s have none. The lists of five simple i-lines are highlighted in Figure 4.3.

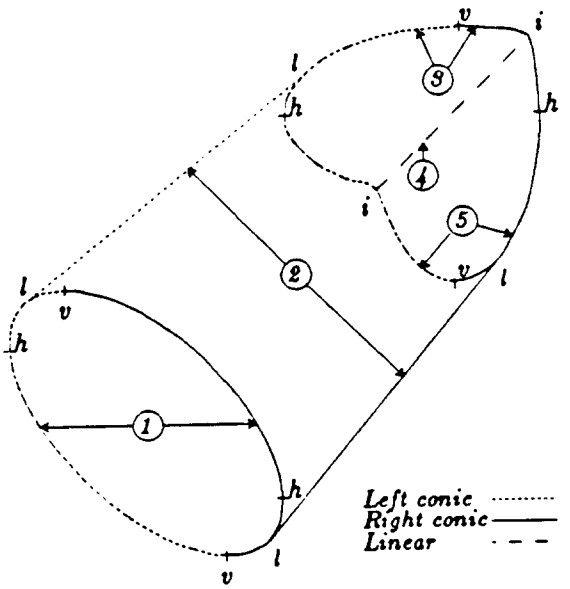


Figure 4.3 — The i-vertex lists of five i-lines

Finally, i-lines contain a planar half-space called the *insert-plane* that helps Resolver put i-vertices in the correct list. An i-vertex that satisfies the insert-plane inequality lies on the left or front leg of a *CONIC* or *EDGE* i-line, respectively.

4.3. Building I-line Tables

Resolver receives an HSB table from Flattener for each body. It begins processing by creating show-faces — one for opaque planar HSBs not perpendicular to the view plane and two for quadric HSBs that are not cylinders perpendicular to the view plane. Each HSB has two pointers to its own show-faces. Where no show-face was created, the pointer is *null*.

An i-line table for the body is completed in two steps. First, the table is created and the coefficients for each non-limb i-line are calculated. For *CONIC* and *EDGE* i-lines, the insert-plane is also determined. When valid coefficients do not exist, as for limbs of planar HSBs and for HSB pairs that do not intersect, the i-line is labeled *NOEXIST*. The following pseudo-C code constructs an i-line table for n HSBs:

```
create an n × n i-line table;
for (i = 1; i < n; ++i)
    for (j = 0; j < i; ++j) {
        determine type of i-line_i,j;
        if (type != NOEXIST)
            calculate coefficients of i-line_i,j;
        if (type == CONIC || type == EDGE)
            calculate insert-plane;
    }
```

Resolver now calculates all possible i-vertices for the body and checks them in each of the body's HSI's. It *throws away* i-vertices that fail in any HSI, then carefully *inserts* any that are not thrown away in the list or lists of appropriate i-lines. Limb i-line coefficients are computed along with limb i-vertices:

```
for (i = 2; i < n; ++i) {     /* one pass for ISECT i-vertices */
    for (j = 1; j < i; ++j) {
        for (k = 0; k < j; ++k) {
            create ISECT i-vertices of HSB_i, HSB_j, and HSB_k;
            for (each i-vertex thus created) {
                for (each HSI of body)
                    if (i-vertex does not satisfy HSI)
                        throw i-vertex away;
                for (each i-vertex not thrown away) {
                    insert in i-line_ij;
                    insert in i-line_jk;
                    insert in i-line_ik;
}   }   }   }   }

for (i = 1; i < n; ++i) {     /* one pass for LIMB i-lines and i-vertices */
    find type and coefficients of limb i-line_ii;
    if (limb i-line_ii is not NOEXIST) {
        for (j = 1; j < n; ++j) {
            if (i == j) continue;
            create LIMB i-vertices where i-line_ii and i-line_ij intersect;
            for (each i-vertex thus created)
                for (each HSI of body)
                    if (i-vertex does not satisfy HSI)
                        throw i-vertex away;
            for (each i-vertex not thrown away) {
                insert in i-line_ii;
                insert in i-line_ij;
}   }   }   }

for (i = 1; i < n; ++i) {     /* one pass for HEXTRM and VEXTRM i-vertices */
    for (j = 0; j <= i; ++j) {
        if (i-line_ij is CONIC) {
            create HEXTRM and VEXTRM i-vertices for i-line_ij;
            for (each HSE of body)
                if (i-vertex does not satisfy HSI)
                    throw i-vertex away;
            for (each i-vertex not thrown away)
                insert in i-line_ij;
}   }   }
```

I-vertices are inserted in an i-line by first picking the correct list, then the correct place within the list on a key of ascending y. Picking a list is trivial for *LINEAR* i-lines — there is only one. There are rules for *CONIC* and *EDGE* i-lines:

(1) If the i-vertex is VEXTRM, insert in both lists
(2) Else if the i-vertex is LIMB and the i-line is EDGE, insert in both lists
(3) Else insert in one list using the list insert-plane

If an i-vertex already occupies the place of a new one, the new one is merged with the old. The merged vertex retains the type of the old one. This is explained in the next section.

4.4. Making Show-Edges

Resolver builds all the i-line tables and saves them so that a final transformation, D, to device coordinates can be applied before creating show-edges. Matrix D cannot be applied nor can show-edges be produced "on the fly" because the size of the scene box itself determines D for auto-scaled images. The size of the box is known only after Resolver is done with its work.

With sorted i-vertex lists, a simple algorithm determines show-edges for an i-line. In the pseudo-code below, *bottom* and *top* are pointers to i-vertices that become end points of show-edges:

```
for (each i-line in the table) {
    for each list in i-line {
        bottom = head i-vertex in list;
        top = bottom → next;
        while (neither bottom nor top point past end of list) {
            generate a show-edge between bottom and top with shape of i-line
            if (top is ISECT or top is LIMB and i-line is a limb)
                bottom = top→next;
            else
                bottom = top;
            top = bottom→next;
}   }   }
```

As mentioned above, coincident i-vertices are merged by the algorithm. This is to prevent the edge creation algorithm above from becoming confused. An explanation by example follows. Consider the five-sided pyramid in Figure 4.4a. If coincident i-vertices were not merged, the apex of the pyramid would appear as two consecutive *ISECT* i-vertices in each of its four incident i-lines. With this as input, the algorithm above would improperly create a single zero-length show-edge for each i-line instead of the correct show-edge. This is a general problem with point intersections of more than three HSBs. Repeated LIMB i-vertices in limb i-lines have the same effect. Merging multilple instances of the same i-vertex is an acceptable solution in all cases but one. Resolver must be careful not to create *any* i-vertex when i-line shapes intersect in a tangent point. As illustrated in Figure 4.4b, this case also produces a double *ISECT*. However, to merge such a pair would cause the edge creation algorithm to omit one show-edge entirely.

As the last processing step, a show-edge is assigned pointers to the show-faces that it bounds. These depend on the type of i-line that produced the show-edge. For *LINEAR* i-lines, processing is trivial — the show-face pointers of both planar HSBs are copied to the show-edge. Show-edges from limb i-lines are similar. The front and back show-face pointers of the quadric HSB are copied.

The lists of *EDGE* i-lines are single valued in z and thus correspond exactly to front and back show-faces. In producing linear show-edges from *EDGE* i-lines, one show-face pointer is given by the list being processed, and the other is null because the associated planar HSB is perpendicular to the view plane.

CONIC i-lines are more demanding. First, the show-face of the i-line's planar HSB_p is copied. For the other pointer, Resolver must choose between front and back show-face pointers of the i-line's quadric HSB_q. Two decision criteria are used:

(a) More than three HSBs intersect in a point

(b) Singular intersection

Figure 4.4 — Multiple i-vertices

(1) The sign of the z component of HSB_q's surface normal at an interior point of the show-edge

(2) The existence of a limb for HSB_q.

The possible outcomes when (1) is non-zero are given in the following table:

	Sign of normal z (1)	
Limb (2)	−	+
yes	front	back
no	back	front

When (1) is zero, the show-edge is coincident with a limb, hence the i-line's planar HSB_p is coincident with the limb plane. In this case, Resolver uses the sign of the z component of HSB_p's surface normal. A positive value implies that HSB_p has cut away the front quadric show-face along the show-edge, so "back" is the answer; *vice versa* for a negative value. A zero value cannot occur; this would make the i-line an *EDGE*.

4.5. Resolver's Tools

It is easy to see that Resolver's algorithms rely heavily on calculations involving planar, quadric, linear, and conic coefficients. This section outlines these calculations without elaboration. Many were derived or checked with the computer algebra system VAXIMA at UC Berkeley.

4.5.1. Conventions

Quadratic Equation:

$$R(x) = A_r x^2 + B_r x + C_r = 0 \tag{4.1}$$

Line Equation:

$$L(x, y) = A_l x + B_l y + C_l = 0; \tag{4.2}$$

Conic Equation:

$$C(x, y) = A_c x^2 + B_c y^2 + C_c xy + D_c x + E_c y + F_c = 0 \tag{4.3}$$

Planar Half-Space:

$$P(x, y, z) = A_p x + B_p y + C_p z + D_p \leq 0 \tag{4.4}$$

Quadric Half-Space, Long Form:

$$\begin{aligned} Q(x, y, z) = {} & A_q x^2 + B_q y^2 + C_q z^2 \\ & + 2D_q xy + 2E_q yz + 2F_q xz \\ & + 2G_q x + 2H_q y + 2J_q z + K_q \leq 0 \end{aligned} \tag{4.5}$$

Quadric Half-Space, Matrix Form:

$$Q(\mathbf{x}) = \mathbf{x}\mathbf{Q}\mathbf{x}^T \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} A_q & D_q & F_q & G_q \\ D_q & B_q & E_q & H_q \\ F_q & E_q & C_q & J_q \\ G_q & H_q & J_q & K_q \end{bmatrix} \tag{4.6}$$

4.5.2. Coefficients of Intersection Line

Eliminate z from two plane equations whose coefficients are denoted by subscripts 1 and 2:

$$\begin{aligned} A_l &= A_{p1}C_{p2} - A_{p2}C_{p1} \\ B_l &= B_{p1}C_{p2} - B_{p2}C_{p1} \\ C_l &= D_{p1}C_{p2} - D_{p2}C_{p1} \end{aligned} \tag{4.7}.$$

4.5.3. Coefficients of Intersection Conic

Eliminate z from plane and quadric equations:

$$\begin{aligned} A_c &= \tfrac{1}{2}(A_p^2 C_q + C_p^2 A_q) - A_p C_p F_q \\ B_c &= \tfrac{1}{2}(B_p^2 C_q + C_p^2 B_q) - B_p C_p E_q \\ C_c &= A_p B_p C_q - A_p C_p E_q - B_p C_p F_q + C_p^2 D_q \\ D_c &= A_p D_p C_q - A_p C_p J_q - D_p C_p F_q + C_p^2 G_q \\ E_c &= B_p D_p C_q - B_p C_p J_q - D_p C_p E_q + C_p^2 H_q \\ F_c &= \tfrac{1}{2}(D_p^2 C_q + C_p^2 K_q) - D_p C_p J_q \end{aligned} \tag{4.8}.$$

4.5.4. Quadric Surface Normal

In matrix form, the quadric surface normal, $\mathbf{n}$, at point $\mathbf{x}$ is given by:

$$\mathbf{n}(\mathbf{x}) = \left[\frac{d\mathbf{Q}}{d\mathbf{x}}\right]_3 \tag{4.9}$$

where subscript three denotes truncation of the homogeneous vector in parenthesis to a three-space vector. This simplifies to:

$$\mathbf{n(x)} = \mathbf{xE} + \mathbf{E^T x^T} = 2\mathbf{xE} \tag{4.10}$$

because E is symmetric. The length of **n** is not important, so the multiplier two is ignored in computation.

4.5.5. Coefficients of Limb Plane

Given a quadric surface that has been transformed to the three-dimensional scene box shown in Figure 3.2, its limb is the locus of points where $\mathbf{n}_z = 0$, or from (4.6) and (4.10) above:

$$\mathbf{n}_z = xF_q + yE_q + zC_q + J_q = 0 \tag{4.11}.$$

Thus, when the quadric limb exists, it lies on a plane, the *limb plane*, and the limb is a conic section.

4.5.6. Coefficients of Limb

Solving a special case of the general plane-quadric intersection yields:

$$\begin{aligned} A_c &= \tfrac{1}{2}(A_q C_q - F_q^2) \\ B_c &= \tfrac{1}{2}(B_q C_q - E_q^2) \\ C_c &= D_q C_q - E_q F_q \\ D_c &= G_q C_q - F_q J_q \\ E_c &= H_q C_q - E_q J_q \\ F_c &= \tfrac{1}{2}(K_q C_q - J_q^2) \end{aligned} \tag{4.12}.$$

When the quadric is a cylinder or cone with a non-degenerate limb, the conic given above will be factorable into lines. UniQuadrix makes no distinction for this case, nor for general intersections that are similarly degenerate.

4.5.7. Existence of Limb

A sufficient condition for a quadric half-space not to have a limb is:

$$\lim_{z \to \infty} Q \le 0 \tag{4.13}.$$

That is, when the half-space (in Resolver Coordinates) includes the z axis at infinity. Since the growth of Q with respect to z is controlled by its z^2 term, (4.13) fails if:

$$C_q > 0 \tag{4.14},$$

UniQuadrix uses this as its test for existence of the limb. This test also gives the desired result, indicating no limb exists for cylinders parallel to the z axis, cones with exactly one ruling line parallel to the z axis, and paraboloids with axis parallel to the z axis, even though these may not satisfy (4.13).

4.5.8. I-vertices of Type *ISECT* (plane-plane-plane intersections)

Solve the system of three plane equations using Gauss elimination.

4.5.9. Intersection of Line and Conic

This is a subroutine for finding i-vertices. If $A_l < B_l$, eliminate y from line and conic equations to obtain a quadratic in x with coefficients:

$$\begin{aligned} A_r &= B_l(B_l A_c - A_l C_c) + A_l^2 B_c \\ B_r &= C_l(2A_l B_c - B_l C_c) + B_l(B_l D_c - A_l E_c) \\ C_r &= B_l(B_l F_c - C_l E_c) + C_l^2 B_c \end{aligned} \tag{4.15}.$$

Otherwise, eliminate x to obtain coefficients of a quadratic in y:

$$\begin{aligned} A_r &= B_l(B_l A_c - A_l C_c) + A_l^2 B_c \\ B_r &= C_l(2B_l A_c - A_l C_c) - A_l(B_l D_c - A_l E_c) \\ C_r &= A_l(A_l F_c - C_l D_c) + C_l^2 A_c \end{aligned} \tag{4.16}.$$

Solve the quadratic that results. The number of real roots corresponds to the number of intersection points. Back-substitute into the line equation to obtain the second coordinate of each intersection.

4.5.10. I-vertices of Type *ISECT* (plane-plane-quadric intersections)

Given the conic of a plane-quadric and the line of a plane-plane intersection, find their intersections in the view plane using (4.15) or (4.16). If the quadratic has double or imaginary roots, create *no* i-vertex; otherwise, create two. Finally, obtain their z coordinates by back-substitution into one of the plane equations.

4.5.11. I-vertices of Type *LIMB*

Given a quadric HSI_q and a planar HSI_p that truncates HSI_q, *LIMB* i-vertices are at the intersections of the HSI_q, its limb plane, and HSI_p.

Therefore, if the i-line formed by HSI_q and HSI_p is *CONIC*, find the line of intersection for HSI_p and the limb plane using (4.7), then find this line's intersections in the view plane with the limb conic using (4.15) or (4.16). If the quadratic has double or imaginary roots, create *no* i-vertex; otherwise, create two. Finally, obtain z coordinates for intersections by back-substitution into the limb plane equation.

If HSI_q and HSI_p form an *EDGE*, find the intersections in the view plane of its line equation and the limb conic, then proceed as with the *CONIC* above.

4.5.12. Extreme I-vertices and Insert-Planes

As discussed in Section 4.2, the i-line insert-plane is Resolver's test to determine the correct list of an i-vertex. This and the vertical and horizontal extrema of a conic i-line are calulated with the same relationships. Make (4.3) homogeneous in x. The result is a quadratic in x with:

$$\begin{aligned} A_r &= A_c \\ B_r &= C_c y + D_c \\ C_r &= B_c y^2 + E_c y + F_c \end{aligned} \tag{4.17}.$$

Now, make (4.3) homogeneous in y, then:

$$\begin{aligned} A_r &= B_c \\ B_r &= C_c x + E_c \\ C_r &= A_c x^2 + D_c x + F_c \end{aligned} \tag{4.18}.$$

Call:

$$B_r^2 - 4A_rC_r \tag{4.19}$$

the *discriminant* of these quadratics. *VEXTRM* i-vertices occur where the discriminant of the quadratic given by (4.17) is zero. The discriminant is, itself, a quadratic in y with coefficients:

$$\begin{aligned} A_r &= C_c^2 - 4A_cB_c \\ B_r &= 2C_cD_c - 4A_cE_c \\ C_r &= D_c^2 - 4AcF_c \end{aligned} \tag{4.20}.$$

*HEXTRM*s occur at the zeros of the discriminant given by (4.18). This is a quadratic in x:

$$\begin{aligned} A_r &= C_c^2 - 4A_cB_c \\ B_r &= 2C_cE_c - 4B_cD_c \\ C_r &= E_c^2 - 4BcF_c \end{aligned} \tag{4.21}.$$

Furthermore, the line:

$$2A_rx + B_r = 0 \tag{4.22}$$

with A_r and B_r from (4.17) divides the conic into legs single-valued in x and passes through any *VEXTRM* i-vertices. Equation (4.22) restated as a plane inequality with C_p set to zero serves as the insert-plane for *CONIC* i-lines. A similar inequality in y and z (A_p zero) serves as the insert-plane for *EDGES*. Similarly,

$$2A_ry + B_r = 0 \tag{4.23}$$

with coefficients from (4.18) divides the conic into parts single-valued in y and passes through *HEXTRM* i-vertices. Lines (4.22) and (4.23) are used to find x and y corresponding to the y and x solutions of (4.20) and (4.21). Figure 4.5 shows the relationship between these i-vertices and lines.

4.5.13. Zero Tests

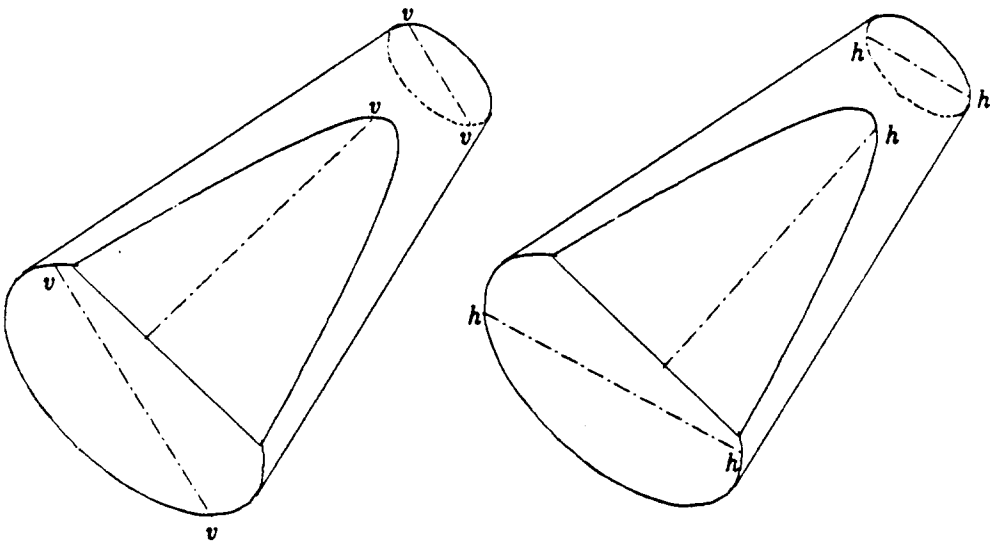
The following tests for zeros support Resolver algorithms:

(1) Is a plane perpendicular to the view plane?
(2) Is the axis of a cylinder perpendicular to the view plane?
(3) Does a quadratic have double roots?
(4) Does a point satisfy an HSI?
(5) Do two i-vertices on a list coincide?

For each, UniQuadrix computes E, the estimated worst case error. If the tested number is within E of zero, the test returns true, otherwise, false. The effectiveness of these tests has not been studied in detail, but they have been reliable in practice.

4.6. Implementation

It was more difficult to describe the algorithms of Resolver than it was to implement them. The Resolver code of UniQuadrix is contained in five modules totaling about 1200 lines. This includes all the coefficient formulae above and routines to generate show-faces and edges.



(4.22) *(4.23)*

Figure 4.5 — Lines given by (4.22) and (4.23)

5. Renderer

Renderer receives show-edges from Resolver and outputs visible spans to Shader in the form:

$$\left[x_{left},\ x_{right},\ y_{scanline},\ show\text{-}face \right]$$

The hidden surface algorithm of Renderer is taken almost directly from that of Unigrafix 1's program **ugshow** by Paul Strauss [Strauss 82]. This chapter briefly outlines the algorithm then details changes that make UniQuadrix simpler.

5.1. Overview

Renderer places show-edges from Resolver in an array of buckets with one bucket per scan line. This is the *Edge Start List* (ESL). An edge is assigned to a bucket by the y coordinate of its bottom vertex. Within a bucket, edges are sorted on a key of ascending x and then dx/dy at their bottom vertices.

Scan lines are processed one at a time from minimum to maximum device y. Renderer maintains an *Active Edge List* (AEL) of edges that intersect the current scan line (CSL). The AEL is maintained in ascending x order. Each scan line requires three passes through the active list. During pass one, edges whose tops are below the CSL, called *finishing* edges, are deleted from AEL and edges in the ESL bucket for CSL, *starting* edges, are inserted. Pass two is an incrementing step that readies show-edges in AEL for the next scan line. Pass three is used to inspect AEL to determine visible spans.

Renderer does not currently implement the enhanced edge option of **ugshow**. UniQuadrix also gains some space and speed advantage by using show-edges that point to two adjacent show-faces. (**Ugshow** created coincident show-edges, each with one show-face pointer.) Of course, the support routines for Renderer must distinguish between planar and quadric show-faces for depth comparisons and between conic and linear show-edges when calculating points.

5.2. More On Show-Faces and Show-Edges

Show-faces and show-edges contain information other than the end points, shape, and show-face pointers discussed in Chapter 4. Show-edges have a *current* x that is initialized to x of the bottom vertex and updated during pass two. Show-edges also have a stack of pointers. Stack elements point to show-faces that are in front of or behind the show-edge on the CSL. In addition, the stack contains a pointer to any face that its show-edge bounds on the left. Renderer maintains stacks in order of increasing depth (z) so that the head of a show-edge's face stack points to the face visible immediately to its right.

5.3. Passes Two and Three

Passes two and three of Renderer are exactly as in Unigrafix. Pass two updates the *current* x of each edge on AEL then bubble sorts the list to maintain ascending x order. Swaps performed during the sort carefully adjust the face stacks of swapped edges.

Pass three begins with the show-edge at the head of AEL. Renderer remembers the show-face at the head of this edge's face stack and chains through AEL until it finds an edge without the same face at the head of its stack. The pixels between these edges are a span. The new stack head is remembered and the process repeats. Spans are discovered until the end of AEL.

5.4. Pass One

Pass one uses a recursive procedure to make the Renderer algorithm simpler than ugshow. Like Unigrafix, Renderer's pass one consists of a loop that is executed until all edges in AEL have been touched and all starting edges for CSL have been inserted in AEL. Let CE be the current edge pointer initialized to the the head of AEL. Then the following pseudo-code describes the loop:

```
while (still edges in AEL or still starting edges) {
    while (CE is finishing)
        process the top vertex of CE;
    if (CE→current_x > head_starting_edge→current_x)
        process the bottom vertex of ESL_bucket_head;
    else {
        touch CE by updating flags;
        CE = CE→next;
}   }
```

A vertex is processed in three steps. First, any finishing edges that are incident are removed from AEL. If the edge to be deleted is not CE, it is first moved to the position adjacent to CE by successive swaps, just as the sort in pass two. These swaps make the face stacks of intervening edges consistent. If the deleted edge is CE, the CE pointer is moved to the next in AEL. Second, each incident starting edge that lies completely within CSL (nominally both starting and finishing) is removed from ESL and its opposite vertex is processed recursively. Finally, remaining starting edges are removed from ESL and inserted in AEL.

In this manner, we eliminate the need to treat horizontal edges as a special case as in Unigrafix. The *insertFL* and *removeFL* lists of **ugshow** are no longer required.

5.5. Renderer's Tools

5.5.1. Depth Comparisons

Only one low-level routine of Renderer is aware that some show-faces are curved and others are flat — the one that inserts faces in stacks in order of increasing depth. An insertion of face F into the stack of edge E requires successive comparisons with faces G already on the stack. The comparison answers the question:

Is F in front of G at the point $P = (\ E\text{→}current\_x,\ y_{CSL}\ )$?

In many cases, it suffices to compute $z(P)$ of faces F and G. The lesser z certainly belongs to the face in front. There are still three cases, however, where the depths may be equal at P while faces F and G are not coincident. These are shown in Figure 5.1. The heavy line in each case represents E. Renderer resolves case (a) by recomputing depth at point Q where E intersects the next scan line. When the depths are again equal as in (b), it computes slope dz/dx or dz/dy at Q depending on whether edge E projected on the view plane is more or less than forty-five degrees from the horizontal at Q. The face with lesser slope is in front. Finally, when the slopes are equal as for the tangent faces in (c), d^2z/dx^2 or d^2z/dy^2 gives the rate of change in slope. The face with the smaller rate of change is in front. When two quadric show-faces are tangent and perpendicular to the view plane at Q (planar show-faces perpendicular to the view plane are never created), rates of change are evaluated as $\pm\,|\,d^2x/dz^2\,|$ or $\pm\,|\,d^2y/dz^2\,|$. The sign is + if the show-face is of type *front* and *vice versa*. Again, the face with smaller rate of

change is in front. If these rates are equal, Renderer assumes that the faces are coincident and that their order on a face stack is not important.

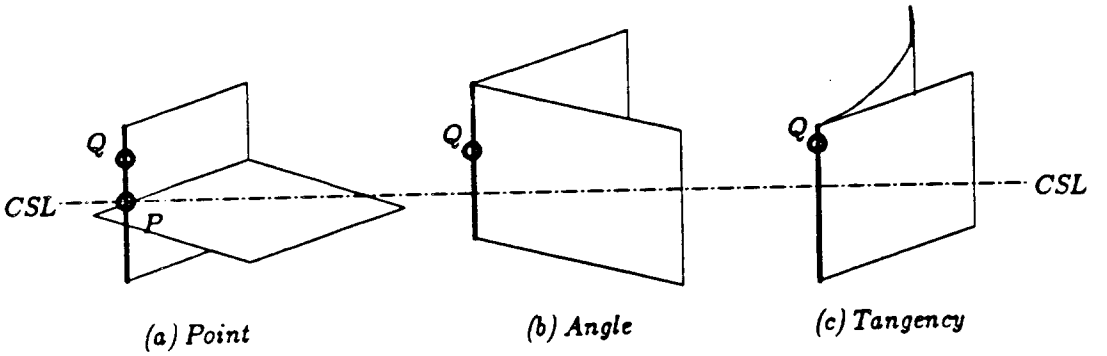


Figure 5.1 — Cases of equal depth

To compute depth for planar show-faces, the plane equation is solved for z:

$$z(x, y) = -\frac{Ax + By + D}{C} \qquad (5.1).$$

For quadric show-faces, x and y are fixed, and the resulting quadratic is solved with the quadratic formula. One root is picked using the show-face sheet flag.

The chain rule for partial derivatives provides formulae to compute show-face slopes and rates as well as the slope of a show-edge at a point. Equations (5.2) give these formulae for z dependent on x in the implicit plane or quadric equation $F(x, y, z) = 0$. Formulae for other variables and for functions in the view plane follow by simple substitution:

$$\frac{dz}{dx} = -\frac{\frac{\partial F}{\partial x}}{\frac{\partial F}{\partial z}} \qquad \frac{d^2z}{dx^2} = -\frac{\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial z \partial x}\frac{dz}{dx}}{\frac{\partial F}{\partial z}} \qquad (5.2)$$

Naturally, slopes are constant and rates are zero for planes and lines. Even for quadrics and conics, these formulae are straightforward to calculate.

5.5.2. Tracing show-edges

There is also only one procedure in Renderer that knows some edges are straight lines and others are curved — the one that updates current x at each scan line.

The current x of linear show-edges is updated by merely adding dx/dy. For conic show-edges, y is fixed for CSL and the resulting quadratic (5.3) solved for x. One root is selected by the leg flag.

$$\begin{aligned} A_r &= A_c \\ B_r &= C_c y + D_c \\ C_r &= B_c y^2 + E_c y + F_c \end{aligned} \qquad (5.3)$$

Coefficients (5.3) are not computed at every scan line. Rather, their first and second differences are used to update them incrementally. This requires five additions.

6. Shader

Shader accepts visible spans from Renderer and cuts them up into *device spans* These are sets of pixels on a scan line that have the same integer intensity. Shader is called up to 10^6 times to draw an image of moderate complexity on the large plotters. Its algorithm must be efficient.

6.1. The Lighting Model

Current UniQuadrix uses a very simple monochrome lighting model based on Lambert's Law:

$$i = \begin{cases} K(i_{amb} + i_{par}\cos\theta) & , \theta < \pi/2 \\ K i_{amb} & , \text{otherwise} \end{cases} \qquad (6.1),$$

where i is an integer device intensity in $[0 \ldots i_{max}]$, i_{amb} is total incident ambient intensity, and i_{par} is the incident intensity of a single parallel source. Angle θ is between a vector to the parallel source, $\mathbf{i}_{par}$, and the reflecting surface normal, $\mathbf{n}$, as shown in Figure 6.1.

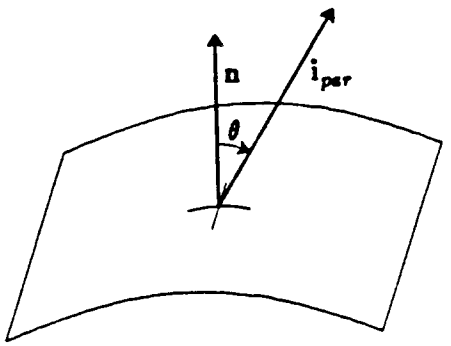


Figure 6.1 — Lighting Model

The factor K scales diffusely reflected world intensity units to the device. Since the language calls for unit world intensity to produce saturated white in the final image, K is just i_{max}.

6.2. The Model Applied to Show-Faces

Referring again to Figure 6.1, assume $\mathbf{i}_{par}$ has length $K\,i_{par}$ and $i'_{amb} = K\,i_{amb}$. Now, (6.1) becomes:

$$i = \begin{cases} i'_{amb} + \dfrac{\mathbf{n}\,\mathbf{i}^{T}_{par}}{|\mathbf{n}|} & , \mathbf{n}\,\mathbf{i}^{T}_{par} > 0 \\ i'_{amb} & , \text{otherwise} \end{cases} \qquad (6.2).$$

Vector $\mathbf{i}_{par}$ and constant i'_{amb} are calculated only once per show operation. Hence, Shader is concerned mostly with computation of $\mathbf{n}\,\mathbf{i}_{par}^{T}/\,|\,\mathbf{n}\,|$. For a plane:

$$\mathbf{n} = \left[A_p\ B_p\ C_p\right] \tag{6.3},$$

so i is constant for planar show-faces. Its value is computed and stored once as each planar show-face is created. Shader need only look this up and output a single device span.

For a quadric, $\mathbf{n}$ is given by (4.10) and, of course, varies over the surface. After the quadric is subject to the view and device transformations, T, we obtain:

$$\frac{\mathbf{n}\,\mathbf{i}_{par}^{T}}{|\,\mathbf{n}\,|} = \frac{(\mathbf{x}\mathrm{T}^{-1}\mathrm{Q})_3\,\mathbf{i}_{par}^{T}}{\sqrt{(\mathbf{x}\mathrm{T}^{-1}\mathrm{Q})_3\,(\mathbf{x}\mathrm{T}^{-1}\mathrm{Q})_3^{T}}} \tag{6.4}.$$

But, $\mathbf{x}$ is a three-space vector. Before Shader can use (6.4) to find intensity at a given pixel (x, y), it must calculate $z(x, y)$ on the surface of:

$$\mathbf{x}\mathrm{T}^{-1}\mathrm{Q}(\mathrm{T}^{-1})^{T}\mathbf{x}^{T} = 0 \tag{6.5}.$$

Since these calculations may occur 10^7 times in a large drawing, it pays to do them efficiently. We have sought a way to shade quadrics without calculating z. The interesting albeit disheartening results are outlined in Chapter 7.

6.3. Quadric *Z*

Shader makes use of the following characteristics of pixels in a span to compute z efficiently:

(1) Pixel y coordinates are all the same
(2) Neighboring pixels differ by one in their x coordinates

Another way of stating (1) is that pixels in a span are all projected from points where a *scan plane* perpendicular to the y axis intersects the quadric (6.5) in device space. The intersection of the quadric on the scan plane is a conic in x and z whose coefficients need be determined only once per span. Successive values $z(x)$ on this conic are evaluated by the same procedure used for successive $x(y)$ on conic show-edges in Renderer. The show-face sheet flag serves in place of the show-edge leg flag.

6.4. Piecewise Matrix Multiplication

Of (6.4), two terms are constant and may be calculated and stored once as each show-face is created — the 4×3 matrix:

$$\mathrm{N} = (\mathrm{T}^{-1}\mathrm{Q})_3 \tag{6.6}.$$

and the 4×1 matrix:

$$\mathrm{L} = \mathrm{N}\,\mathbf{i}_{par}^{T} \tag{6.7}$$

The arithmetic at each pixel after z is calculated can now be shown as:

$$\frac{\mathbf{n}\,\mathbf{i}_{par}^{T}}{|\,\mathbf{n}\,|} = \sqrt{\frac{(\mathbf{x}\mathrm{L})^2}{(\mathbf{x}\mathrm{N})(\mathbf{x}\mathrm{N})^{T}}} \tag{6.8}$$

Where the square root is a fast integer calculation that assumes the result to be in $[0\ldots i_{max}]$. But Shader does not carry out complete matrix multiplications xL and xN at each pixel. Rather, by (1) it precomputes terms in y once per span, and by (2) it

computes each successive value of terms in x by addition. Only terms in z are computed at every pixel and only when xL > 0.

6.5. Implementation

6.5.1. Non-Unit Shading Increments

The Shader algorithm is easy to implement so that intensity in a span is calculated at fixed integral increments instead of at each pixel. We correctly judged that UniQuadrix would spend a large share of its running time executing Shader. Thus, a simple heuristic gives the user some control over the trade-off between image quality and running time. Each span is treated separately, with intensity calculated at its left and right ends, x_l and x_r, and at intermediate points separated by Δ. In C integer arithmetic:

$$\Delta = \min \left\lfloor \frac{(x_r - x_l)}{k} + 1,\ \Delta_{max} \right\rfloor \qquad (6.9).$$

Constant k is a compile-time parameter, and Δ_{max} is a user parameter that forces evaluation at least every Δ_{max} pixels. Within each Δ interval, Shader renders the integer average of intensities at the ends. For efficiency, the device driver is called only when intensity changes as Shader moves across the span. The "+ 1" is a computationally cheap way of keeping Δ greater than zero when there are fewer than k pixels in the span. A value of sixteen for k produces images almost identical to those with intensity calculated at each pixel.

6.5.2. The Device Interface

All parts of UniQuadrix see the same device interface regardless of the device used for rendering. A C structure defines this interface. It includes device characteristics such as Δ_{max}, the number of intensities it can render, the number of pixels on each axis, and pixel aspect ratio. It also contains pointers to routines to open and close the device, output a span, and apply controls. Manager maintains a hash table of such structures. One is picked as the *current device* by the last **device** statement, by the -d command line option, or by UNIX environment variables as described in Chapter 2.

The current device structure has proven to be a powerful and clean abstraction. Among many other functions, it provides information to calculate the device matrix (D) to Resolver, identifies bottom and top scan lines of the image for Renderer, and passes the routine to output a span to Shader. The commands and options to change image size and Δ_{max} merely modify this data structure that represents the current device.

6.5.3. Driver Programs

UniQuadrix programs that interact with physical devices are very much the same as Unigrafix. The Sun's monochrome screen and the Varian and Versatek plotters are bi-level devices. These drivers use *stipple patterns* to emulate the dots of lithographic halftones. Device span intensity provides an index into a stipple table that contains bit-images of the patterns. For the plotters, appropriate runs of stipple pattern are masked and ored to a raster buffer that represents a number of scan lines in the final image. Complete buffers are written to a temporary file that is finally printed with a system call to the UNIX lpr program. The Sunwindows interface copies stipple patterns to the Sun's bit-mapped screen.

Color Sun and Vectrix drivers merely load the color map with an even gray scale as the device is opened and use device span intensity as a color map index. No gamma correction [Foley 83] is performed.

7. Evaluation and Conclusions

7.1. Some Run-Time Statistics

UniQuadrix includes code that counts interesting events and prints the CPU time required for portions of scene processing. This chapter presents these statistics for some images given in the Appendix. Table 7.1 reflects Resolver input and output. Table 7.2 counts the computationally expensive operations that Renderer performs. Finally, Table 7.3 gives run times. These were collected on a VAX 750 with two megabytes of memory. UniQuadrix runs faster on a larger VAX 780 by a factor of 1.3 to 1.7. All VAX machines used for testing have floating point accelerators. Images measure eight and one half by eleven inches (some were photographically reduced to fit the Appendix pages) and were produced on the Varian plotter. This plotter has a resolution of 200 dots per inch.

Table 7.1 — Resolver statistics

Image	Bodies	HSBs(1)		Ivertices(2)		Show-Edges	
		planar	*quadric*	created(3)	inserted(4)	*linear*	*conic*
clover knot	24	72	24	622	292	36	328
gear	23	106	53	1253	324	80	350
name	85	270	85	2267	1050	131	1002
triangles	30	60	30	408	360	0	419

Notes

(1) HSBs of all bodies in the flattened scene hierarchy

(2) I-vertices inserted in more than one i-line are counted only once

(3) All i-vertices that could be calculated

(4) *Created* i-vertices that satisfy their bodies' HSIs; inserted in at least one i-line

Table 7.2 — Renderer statistics

Image	Depth Calc's (5)	Face Inserts (6)	Edge Swaps (7)	Spans (8)	Intensities (9)	Pixels (×1000) (10)
clover knot	2325	614	500	8991	143808	1683
gear	3509	841	286	13238	59297	1453
name	638	1941	1541	9624	118157	773
triangles	10633	2475	1118	17048	228490	902

Notes

(5) Finding $z(x, y)$ on a planar or quadric show-face
(6) Adding a face to a show-edge stack on a key of ascending z
(7) Exchanging two edges on AEL during sorts and deletions
(8) Input to Shader (not device spans)
(9) Intensity calculations on quadric show-faces
(10) Total shaded pixels in image

Table 7.3 — Run time statistics

Image	Reader *(sec)* (11)	Flattener-Resolver *(sec)* (12)	Renderer *(sec)* (13)	Shader *(sec)* (14)	Total *(sec)* (15)
clover knot	1.72	3.76	34.72	138.9	179.16
gear	.45	9.60	15.86	42.89	68.87
name	5.06	14.66	23.79	116.86	159.78
triangles	1.80	4.18	36.52	178.30	219.93

Notes

(11) Includes UniQuadrix initialization and reading scene file(s).
(12) Includes creation of show-faces and insertion of show-edges in ESL
(13) The portion of Renderer-Shader time spent in Renderer (determined using the UNIX **gprof** facility)
(14) The portion of Renderer-Shader time spent in Shader and in the device driver writing spans (determined using the UNIX **gprof** facility)
(15) Gross CPU time at exit; includes Manager functions

It is easy to see that UniQuadrix is compute-bound by the shading algorithm for images with even small quadric surfaces. Clearly, the pains taken to minimize calculation in Shader and the fast shading option for image debugging were worthwhile.

7.2. Conclusions

7.2.1. Wins

This project began with the idea that by generalizing the definitions of edges and faces in an image, quadric surfaces can be rendered using the same algorithm used in **ugshow** to render planar polygons, which exploits object coherence as much as possible. UniQuadrix is a working system which demonstrates the viability of this idea.

Response to UniQuadrix from a user community accustomed to the vertex-face world of Unigrafix has been favorable with regard to the quality of UniQuadrix images. Of course, the rigors of producing half-space input are a shock for those who regularly enter imagined object descriptions directly into a Unigrafix file. As a partial solution to this problem, programs have been built that produce UniQuadrix files from descriptions with far more intuitive meaning than coefficients. Some of the images in the Appendix were produced with these programs. In particular, "clover knot" and "triangles" used a new version of **mkworm** that produces automatically the specifications for all cylinders and the terminating planes or spheres at every joint from a piecewise linear specification of the axes of the used tubular elements. "Name" was produced with a new program **ugpipe** which permits the proper mitering of more than two elements and also accepts the specification of a spherical ball at each joint.

Turn-around time is altogether reasonable for the type of images produced by UniQuadrix. A new Unigrafix rendering program called **ugdisp** under development by Nachshon Gal performs Gouraud shading [Gouraud 71] in an efficient manner. It operates in time comparable to UniQuadrix for objects with similar appearance, but it cannot avoid the ragged piecewise linear outlines produced of the underlying polyhedral object definition. The run time of any polygon renderer using a polygonal mesh fine enough to present conic edges accurate to one pixel in the final image would far exceed the run time for equivalent UniQuadrix images.

7.2.2. Losses

Early in the development of UniQuadrix, we sought a very simple and fast incremental algorithm to shade pixels across whole spans on visible quadric show-faces. We calculated an exact expression for the reflected intensity of a parallel light:

$$f(i, x, y) = 0 \tag{7.1}$$

where f is the result when z is eliminated from the equation of the quadric surface, $Q(x, y, z) = 0$, and the implicit form of (6.8), $I(i, x, y, z) = 0$. The approach was to fix y in f for the current scan line, then evaluate points on the resulting function in i and x using the method described by Jordan, *et. al.* [Jordan 73]. Computation in this algorithm consists entirely of integer additions and comparisons and is bounded in number of steps for a span by the number of pixels plus twice the number of device intensities. Unfortunately, Jordan's algorithm is practical only for conics. The fully general expression of (7.1) is of the form:

$$f = A(x, y)i^4 + B(x, y)i^2 + C(x, y) = 0 \tag{7.2},$$

where A, B, and C are fully general expressions of degree four in x and y, each with fifteen terms. Moreover, the coefficient of each term is a matrix expression in the coefficients of Q and $\mathbf{i}_{per}$. Even if there were a method to evaluate (7.2) efficiently for i as x is incremented, the calculation required to find A, B, and C and the storage for once-per-image and once-per-span intermediate terms are likely to be very large.

In the special cases of cylinders and spheres viewed with orthographic projections, (7.2) simplifies to a conic in i, x, and y. Small tests evaluating this conic using Jordan's method indicated that such a special purpose shader would run at least ten times faster than the UniQuadrix Shader in average cases.

Finally, current UniQuadrix does not fit well in the interactive window environment of the Sun workstations at Berkeley. These workstations lack hardware for floating point operations. Thus, an image rendered with 2000×2000 pixels in three minutes on a VAX takes a similar amount of time with 300×300 pixels on the Sun with no other processes competing. Even the -f fast constant shading option does not lead to what we feel is an acceptable interactive turn-around. Sun does offer a hardware option for floating point and we will certainly try UniQuadrix on such a system if the opportunity arises.

7.2.3. Work Still To Be Done

UniQuadrix is a no-frills implementation of a rendering algorithm. It is ripe with opportunities for enhancement and expansion. Here are some proposals for a continuation of this work.

The performance measurements indicate that UniQuadrix would become far more useful on the Sun workstation and even on the VAX if the wire-frame rendering options of Unigrafix were available. By drawing show-edges with no consideration for hidden surface removal, the user could envision quickly how a final, smooth-shaded image would be oriented. Even a re-implementation of the **ugshow** enhanced edge option in UniQuadrix would cut run time by a factor of almost three, producing wire-frame images with hidden edges removed.

Using the "traced edge" technique of Woon [Woon 71], it is possible to extend Resolver to produce the quartic show-edges of general quadric-quadric intersections and to give Renderer the tools to trace them. I-lines for general intersections need four i-vertex lists. A new type of i-line is needed to represent cylinders that are perpendicular to the view plane, similar to the function of *EDGE* i-lines for planes. The main problem to be solved is giving a topological order to the four roots of a quartic; show-edge leg flags would have four possible values instead of two, and we must map this value to a continuous segment between two extrema of the quartic i-line shape.

The algorithm of Resolver should be re-evaluated in the context of numerical stability. The key areas were briefly mentioned in Section 4.5.13. Indeed, a formal analysis of the entire algorithm is certain to provide fresh insights.

The primitive interactive mode of UniQuadrix was incorporated mainly to test the rest of the program in repeated runs. A much richer interactive environment could be easily implemented over the existing code.

A front end should be provided to UniQuadrix that makes it easier for the user to specify spheres, cylinders, and other quadric surfaces than through the use of the coefficients of the quadric equation; the use of half-axes, center coordinates, and orientation angles is more easily understood by the user.

It would be easy and desirable to incorporate in UniQuadrix a simple color model such as the one in Unigrafix.

7.3. Summary

UniQuadrix is a system for rendering objects composed of quadric and planar half-spaces. It uses implicit equations to represent the scene throughout the rendering process. The program reads scene descriptions in a terse language like that of Unigrafix. It flattens the scene hierarchy and computes intermediate data structures that represent the boundaries between half-spaces. These structures are used to produce edges and faces for a "scan line" hidden surface algorithm that exploits scan line as well as object coherence. Curved edges and faces are rendered exactly and not approximated with polygons and straight lines. Output is to any of several bi-level and gray scale devices. Compute time is rather short considering the quality of image produced.

References

[Appel 67] Appel, A., "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proceedings of the ACM National Conference,* Thompson Books, Washington D. C., 1967, pp. 37-45.

[Bayer 73] Bayer,B.E. "An Optimum Method For Two-Level Rendition of Continuous Pictures," *International Conference on Communications, Conference Record,* 1973, pp. (26-11) - (26-15).

[Boyse 81] "Data Structure for a Solid Modeler," SIGGRAPH '81, Seminar Notes: Solid Modeling.

[CRC 74] CRC Standard Mathematical Tables, Edited by Samuel M. Selby, CRC Press, 1974.

[Foley 83] Foley, J.D. and A.Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

[Gouraud 71] Gouraud, H., "Continuous Shading of Curved Surfaces, " *IEEE Transactions on Computers,* C-20(6), June 1971, pp. 623-628.

[Hakala 81] Hakala, D.G., R.C. Hillyard, P.J. Malraison, B.E. Nourse, "Natural Quadrics in Mechanical Design," *SIGGRAPH '81 Seminar Notes: Solid Modeling,* published in Autofact West, Vol. 1 (CAD/CAM VIII), pp 363-378, Society of Manufacturing Engineers, Dearborn, Michigan, November 1980.

[Jarvis A76] Continuous Tone Images on a Bilevel Display," *IEEE Transactions on Computers,* C-24(8), August 1976, pp. 891-898.

[Jarvis B76] Jarvice, J.F, C.N. Judice, W.H. Ninke. "A Survey of Techniques for the Image Display of Continuous Tone Pictures on Bilevel Displays," *Computer Graphics and Image Processing,* 5(1), March, 1976, pp. 13-40.

[Jordan 73] Jordan, B.W., W.J. Lennon, B.C. Holmes. "An Improved Algorithm for the Generation of Non-parametric Curves," *IEEE Transactions on Computers,* C-22(12), December 1973, pp. 1052-1060.

[Knowlton 77] Knowlton, K. and Cherry, L., "ATOMS - A Three-D Opaque Molecule System for Color Pictures of Space-Filling or Ball-and-Stick Models," *Computers and Chemistry,* 1, 1977, pp. 161-166.

[Levine A76] Levin, J.Z., "A Parametric algorithm for drawing pictures of solid objects composed of quadric surfaces," *Communications of the ACM,* 19, 1976, pp. 555-563.

[Levine B76] Levin, J.Z., "Mathematical Models For Determining the Intersections of Quadric Surfaces," *Computer Graphics and Image Processing,* 11, 1979, pp. 73-87.

[Mahl 72] Mahl, R., "Visible Surface Algorithm for Quadric Patches," *IEEE Transactions on Computers,* C-21, January 1972, pp. 1-4.

[MAGI 68] Mathematical Applications Group, Inc., "3-D Simulated Graphics, " *Datamation,* February 1968.

[Max 79] Max, N., "ATOMLLL: - ATOMS with Shading and Highlights," *SIGGRAPH '79 Proceedings,* published as *Computer Graphics* 13(2),

August 1979, pp. 300-307.

[Porter 78] Porter, T., "Spherical Shading," *Siggraph '78 Proceedings*, published as *Computer Graphics*, 12(3), August 1978, pp. 282-285.

[Porter 79] Porter, T. "The Shaded Surface Display of Large Molecules," *SIGGRAPH '79 Proceedings*, published as *Computer Graphics*, 13(2), August 1978, pp. 282-285.

[Sarraga 83] Sarraga, R.F. "Algebraic Methods for Intersections of Quadric Surfaces in GMSOLID," *General Motors Research Publication GMR-3944R*, July 1, 1983, published in *Computer Vision*, Graphics and Image Processing, 22, May 1983, pp.222-238.

[Séquin 83] Séquin, C.H. and P.S. Strauss. "UNIGRAFIX," *IEEE 1983 Proceedings of the 20th Design Automation Conference," pp 374-381.*

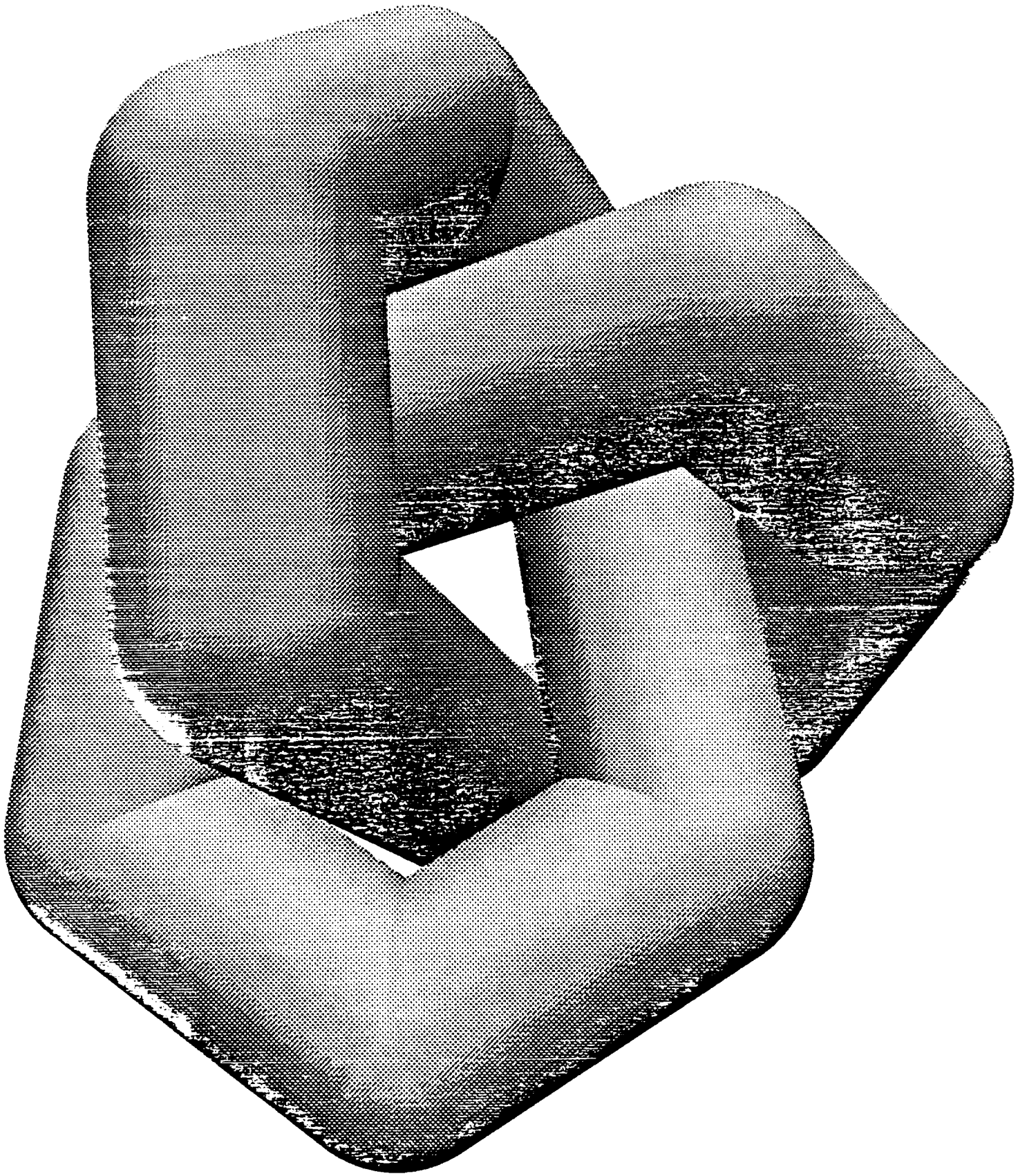
[Staud 78] Staudhammer, J., "On Display of Space Filling Atomic Models in Real-Time," *SIGGRAPH '78 Proceedings*, published in *Computer Graphics*, 12(3), August 1978, pp. 167-172.

[Strauss 82] Strauss, Paul S., "The Unigrafix System: Implementation Guide," Master's Project Report, UC Berkeley Computer Science Division, August 1982.

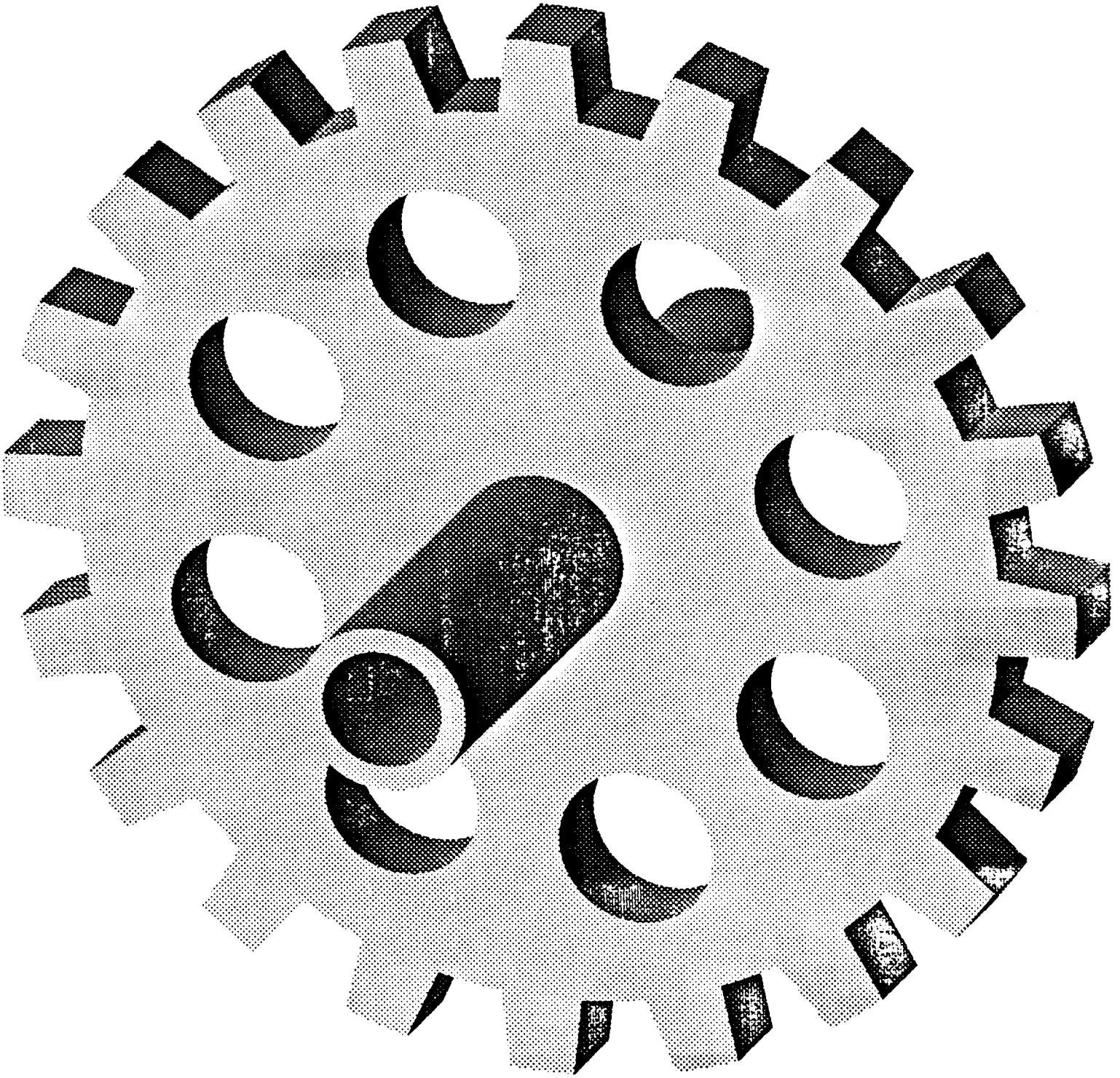
[Weiss 66] Weiss, R.A., "BeVision, a Package of IBM 7090 Fortran Programs to Draw Orthographic Views of Combinations of Planes and Quadric Surfaces," *Journal of the ACM*, 13(2), April 1966, pp. 194.

[Woon 71] Woon, P.Y. and H. Freeman, "A Procedure for Generating Visible-Line Projections of Solids Bounded by Quadric Surfaces," *Proceedings 1971 IFIP Congress*, North-Holland Pub. Co., Amsterdam, 1971, pp. 1120-1125.

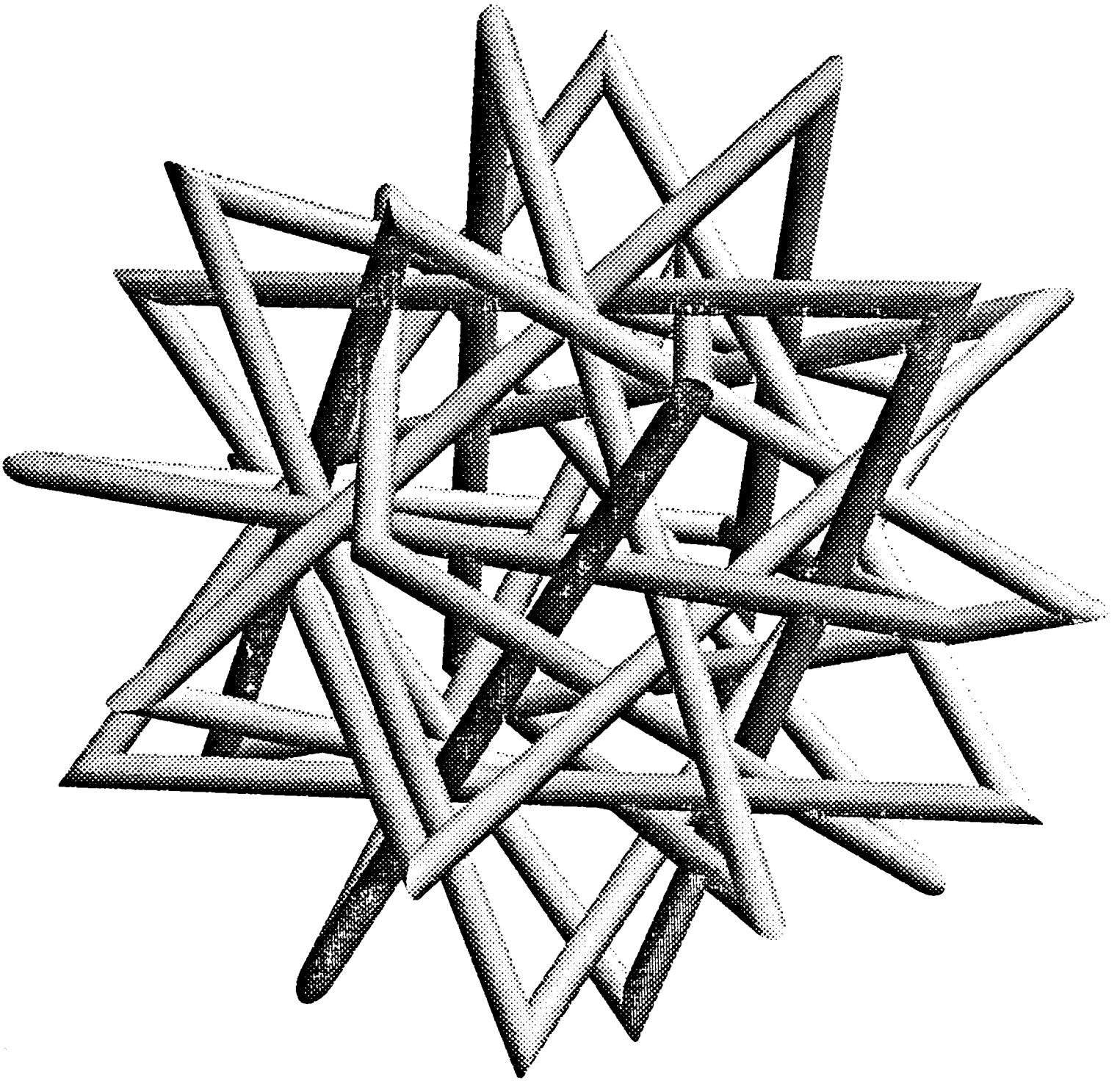
Most of the images on the following pages were produced by students in a course on geometric modelling at UC Berkeley. They are rendered on the Benson Varian plotter.



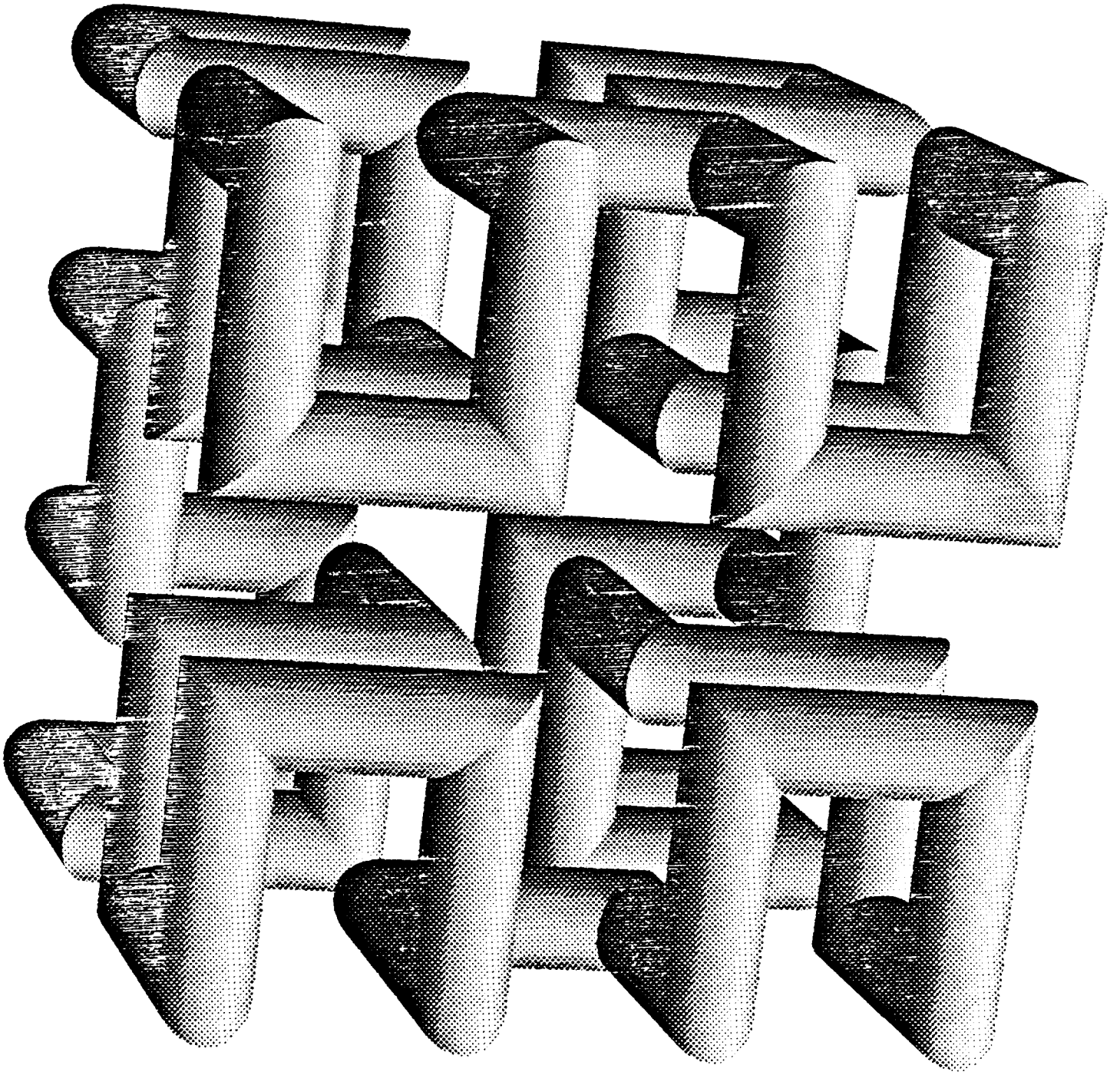
Clover Knot



Gear



Triangles



Hilbert Curve