INTRODUCTION TO THE BERKELEY UNIGRAFIX TOOLS Version 3.0

Carlo H. Séquin and Kevin P. Smith

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

This is a brief overview over the current status of the Berkeley *UNIGRAFIX* tools. The various renderers, object generators and modifiers are introduced. The *UNIGRAFIX* object description format that ties these tools together is presented.

1. INTRODUCTION

Berkeley *UNIGRAFIX* provides simple graphics and modeling capabilities for 3-dimensional polyhedral objects within the UNIX operating system. It can render mechanical parts or geometrical manifolds in the style of engineering drawings, with visible edges displayed as solid lines and faces shaded to enhance understandability of the drawing, rather than to make the objects look ''natural''. This kind of output is primarily aimed at postscript printers or high-resolution black-and-white dot-raster plotters. Alternatively the objects can be viewed itercatively on SUN or SGI workstations.

The Berkeley *UNIGRAFIX* system also comprises a set of generator programs that assist in the creation of parameterized object descriptions such as gear wheels or architectural elements (staircases, houses ...). There are programs that modify simple objects by truncating them, by tessellating their faces, or by cutting holes into them; others place pyramids on all faces or replace the edges of the object with thin prismatic solids.

Most of these programs focus on the design, representation, and rendering of mechanical parts and on the creation and visualization of purely geometrical objects in 3 and 4 dimensions. Some of these utilities are useful in the areas of robotics and computer vision. All tools are loosely tied together by a simple descriptive ASCII format for the specification of the geometrical objects. They run under the UNIX 4.2 and 4.3 BSD operating system.

This "system" was put together over the last several years by several Master's Degree candidates and by many students taking a project-oriented graduate course in Geometric Modeling.[1] The original goals were to give us the geometric modeling and rendering tools that were so conspicuously absent from the UNIX environment in the early 1980's, but also to provide an educational experience for the many students interested in computer graphics.

Building such a large and potentially heterogeneous system at an university, where the period during which a student contributes actively to the project ranges from a few months to a few years, has its difficulties. How do you assure that the various pieces built by individuals will hold together and do not become obsolete the moment the student leaves the university ? One approach is to produce a very detailed overall system specification at the beginning of the project, and then fill in the pieces over the years. However, in a field that moves as rapidly as the current evolution in computer graphics and geometric modeling, this is not practical; the final system would be obsolete by the time it is completed.

A better way is to create a modular set of building blocks that can be individually developed at their own paces and that can be replaced by newer and better modules as these become available. In this approach, the only thing that needs to be defined at an early stage is the ''glue'' that holds everything together. In our case this glue consists of the UNIX operating system and of the *UNIGRAFIX* language. The latter is an intermediate descriptive format for the specification of objects and scenes. All modules work to and from this format. Because of its central role, we will discuss this language before we discuss some of the operational modules.

In a more recent attempt to find some "stronger glue" that could provide a tighter coupling between various generators and tools, a set of abstract data structures has been created which should be sufficient for the construction and integration of a variety of new tools and generators. These data abstractions will be described in Section 3.

2. THE UNIGRAFIX LANGUAGE

The *UNIGRAFIX* language is a human readable, yet terse ASCII format for the description of scenes composed of 3-D polyhedral objects in boundary representation, of 2-D planar faces with arbitrary many holes, and of 1-D piecewise linear wire trains. The ASCII format makes possible easy exchange of object descriptions over electronic networks and easy modification with any text editor when an interactive graphics editor is unavailable or impractical to use. It also facilitates the development of programs that generate objects in a procedural manner. A detailed description and discussion of the *UNIGRAFIX* language was presented in 1983.[2] A few minor syntax changes have been made since then to make parsing more efficient and to accommodate some small language extensions. More recently, a more significant extension has been added to allow interpolating curved surfaces.

Syntactically, a *UNIGRAFIX* file consists of statements, starting with a keyword and ending with a semicolon. Statements consist of lexical tokens, separated by commas, blanks, tabs, or newlines. The language is simple and has less than twenty different types of statements:

Table 1. *UNIGRAFIX* Syntax

vertices:	**v**	*ID x y z [colorID]* ;
wires:	**w**	[*ID*] (*v1 v2* ... *vn*) (...) [*colorID*] ;
flat face:	**f**	[*ID*] (*v1 v2* ... *vn*) (...) [*colorID*] ;
definitions:	**def**	*defID* ;
		non-def-statements
	end;	
instances:	**i**	[*ID*] (*defID* [*transforms*]) ;
arrays:	**a**	[*ID*] (*defID* [*transforms*]) *size* [*transforms*] ;
lights:	**l**	[*ID*] *intensity* [*x y z*] ;
color:	**c**	*colorID intensity* [*hue* [*saturation*]] ;
include files:	**include**	*filename* [*transforms*] ;
comments:	**{**	[*anything {nesting is OK} but unmatched { or }*] **}**

2.1. Vertices, Wires, and Faces

Semantically, the vertices are the 'corner stones' of any *UNIGRAFIX* object description. They are described by their absolute locations in 3-D space and are then used as fix-points to define the position of 'wires' and 'faces' (edges). Rather than repeating absolute coordinates for the end-points of edges or for corners of polygons, these latter constructs simply reference previously defined vertices by their identifiers (*ID* in Table 1). A piece-wise linear wire running through 3-D space can be described with a single 'wire' statement that lists all the vertices at subsequent joints.

In the case of 'face' statements, the edge train defined by the list of vertex-IDs within a single pair of parentheses specifies a closed contour, so that it is not necessary to repeat the first vertex in a contour. Face statements with multiple groups of parentheses can be used to describe faces with several contours. Whether the contour encloses a hole or a separate patch of the face depends on its orientation and on its placement with respect to the first contour. The first contour must be an outer contour. Contours are not allowed to intersect. In all cases all the vertices referenced by a single face must lie in a single plane.

2.2. Curved Edges and Patch Boundaries

Over the last three years, the *UNIGRAFIX* language served its purpose well for objects and scenes composed of linear geometrical primitives. To extend the range of applicability of *UNIGRAFIX* to objects with curved edges and surfaces, the *UniCubix* extension has been added to the language.

We follow an approach introduced by Chiyokura[3] which now forms the foundation of Designbase, the solids modeler developed at Ricoh. In this system objects are entirely defined by their *edges* and by the *borders* between the curved patches, both of which can be straight line segments or cubic space curves. The system automatically fits patches between these borders guaranteeing 0th order continuity along *edges* and first order geometric (G1) continuity across the *borders* between patches. Thus, once a satisfactory and unambiguous method of constructing these patches has been defined, it is sufficient for the language to specify the exact shape of all edge and border curves. For cubic curves this can be done with two additional Bezier points each. This approach can of course be generalized to use more complicated construction rules for the edges. For this reason, *UniCubix* gives the curvature information in explicit edge statements (see below) rather than integrating it into the face statements. This also excludes duplicate, and possibly conflicting, specification of that information in patches sharing the same border, and it keeps the old *UNIGRAFIX* statements unchanged and makes this a straight upward extension.

Table 2. UniCubix Extensions

linear edge:	**el**	[*ID*] (*v1 v2*) ;
linear border:	**bl**	[*ID*] (*v1 v2*) ;
curved edge:	**ec**	[*ID*] (*v1 v2* $b1_x$ $b1_y$ $b1_z$ $b2_x$ $b2_y$ $b2_z$) ;
curved border:	**bc**	[*ID*] (*v1 v2* $b1_x$ $b1_y$ $b1_z$ $b2_x$ $b2_y$ $b2_z$) ;
flat face:	**f**	[*ID*] (*v1 v2* ... *vn*) (...) [*colorID*] ; {unchanged}
curved patch:	**p**	[*ID*] (*v1 v2 v3* [*v4*]) [*colorID*] ;

Since patches are uniquely determined by their borders, there would be no need for a special patch statement. Nevertheless, we are planning to use a separate keyword for a curved patch to distinguish from flat faces. The old face statement '**f ...**' will continue to be treated like a flat polygon that does not necessitate any subdivision for rendering. The control vertices of all the borders of such a patch must of course lie in the plane of the face. The tessellation of the adjacent curved patches determines into how many segments each edge of this flat face will be subdivided.

2.3. Hierarchical Constructs

A building block definition capability exists with the statement pair : '**def...end**'. With this construct, groups of statements can be associated with an identifier (*defID* in Table 1). Copies of these definitions can then be placed at other locations in the scene through the use of 'instance' and 'array' commands. The definitions themselves are not part of the visible scene. Instances and arrays take any homogeneous transformation for the placement of the (first) instance; in addition, the array statement needs the specification of an incremental transformation between subsequent array elements. While the definitions themselves must not be nested, the calling hierarchy can be of arbitrary depth. The permissible *transforms* in the above specified places are of the form:

Table 3. Transformations

–s?	*scale_factor*	for scaling in direction of coordinate axis
–t?	*translation_ amount*	for translation along coordinate axis
–r?	*rotation_ angle*	for rotation around coordinate axis
–m?		for mirroring in direction of coordinate axis
–M3	*3x3 Matrix*	for linear 3-dimensional transformation
–M4	*4x4 Matrix*	for homogeneous 3-D transformation

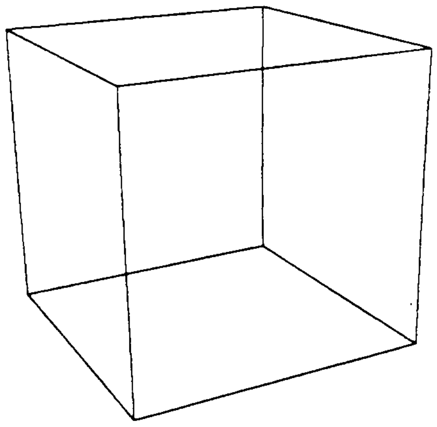
The '?' should be replaced with 'x', 'y', or 'z' to denote in which direction to scale, mirror, or translate, or about which axis to rotate; as a shorthand way of specifying scaling or mirroring in 'all' dimensions, the '?' can be replaced with 'a'. When specifying a transformation in matrix form, from 1 to 9 numbers (for '-M3') or from 1 to 16 numbers (for '-M4') may be specified. The specified numbers replace the entries, by rows, in a unity matrix of degree 3 or 4, respectively. Transformations are applied to the defined object in the order given. Arguments may be integer or floating-point numbers.

2.4. Examples

To illustrate the correspondence between the ASCII description and the defined object, we present a few simple cases. The first is the wire-frame of a cube:

Figure 1. Cube file

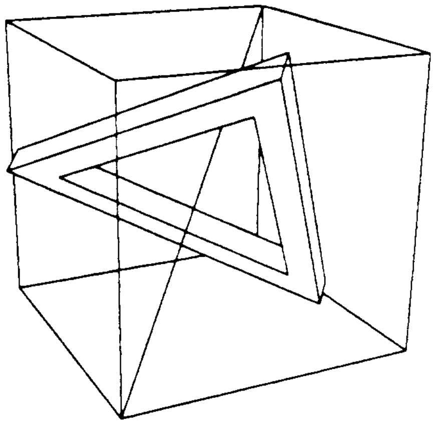
```
v XYZ    1 1 1;
v XY     1 1 -1;
v XZ     1 -1 1;
v X      1 -1 -1;
v YZ     -1 1 1;
v Y      -1 1 -1;
v Z      -1 -1 1;
v N      -1 -1 -1;
w near   ( N Y XY X N );
w far    ( Z XZ XYZ YZ Z );
w sides  ( X XZ )( Y YZ )( XY XYZ )( N Z );  {4 separate wire segments}
```



The second is an equilateral solid triangular frame embedded in a cube frame so that its symmetry axis coincides with one space diagonal of the cube:

Figure 2. Triangle file

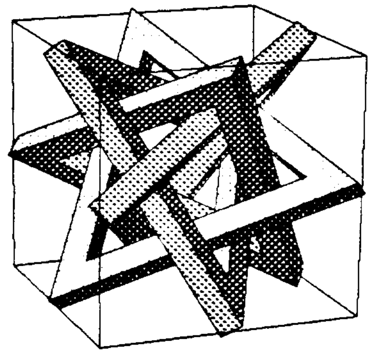
```
v v1A    -5.4082 -0.4082  4.5917 ;
v v1B    -4.5917  0.4082  5.4082
v v1C    -6.5917  0.4082  7.4082 ,
v v1D    -7.4082 -0.4082  6.5917 ;
v v2A     4.5917 -5.4082 -0.4082 ;
v v2B     5.4082 -4.5917  0.4082 ;
v v2C     7.4082 -6.5917  0.4082 ;
v v2D     6.5917 -7.4082 -0.4082 ;
v v3A    -0.4082  4.5917 -5.4082 ;
v v3B     0.4082  5.4082 -4.5917 ;
v v3C     0.4082  7.4082 -6.5917 ;
v v3D    -0.4082  6.5917 -7.4082 ;
f        ( v1A  v1B  v2B  v2A );
f        ( v1C  v1D  v2D  v2C );
f        ( v2A  v2B  v3B  v3A );
f        ( v2C  v2D  v3D  v3C );
f        ( v3A  v3B  v1B  v1A );
f        ( v3C  v3D  v1D  v1C );
f        ( v1C v2C v3C )( v3B v2B v1B );  {face with triangular hole}
f        ( v3D v2D v1D )( v1A v2A v3A );  {face with triangular hole}
```



It is possible to mutually interlock four of these triangles if they are properly oriented. This can be achieved with four separate instance calls or one single array call to the triangle, which is assumed to reside in a file called 'Triangle_file'. The previously defined cube, assumed to be in a file 'Cube_file', has been scaled up by a factor of 7 to match the size of the triangles:

Figure 3. *UNIGRAFIX* Scene

```
def cube;
  include Cube_file;
end;
def triangle;
  include Triangle_file;
end;
i C ( cube  -sa 7.0 );
a T ( triangles ) 4  -rz 90 ;
```



This example also demonstrates that faces with holes can be properly drawn, even if they are interlocking, without the need to cut the faces into smaller pieces. Available rendering styles include: full wire frame (Fig.1), wire frame with backface elimination, hidden lines removed (Fig.2), and shaded faces with hidden parts removed (Fig.3). Faces can be rendered with or without outlines. The latter contribute significantly to the crispness of the display when rendered on a black-and-white device.

2.5. Light Sources and Color

To provide shading on the faces, light sources must be specified. These can be uniform ambient lights or they can be directional. In the first case, all faces regardless of their orientation are equally illuminated. In the latter case, the brightness is determined by the scalar product between illumination vector and face normal.

On appropriate terminals, color renderings can be produced. To specify a color, we use a double-cone model of color space. For each color, its lightness (intensity), its hue, and its saturation must be given; if the last one or two values are omitted, fully saturated colors or neutral gray surfaces are inferred, respectively. As in the case of the vertex coordinates, the lengthy color specification is not repeated for every face using that color; one simply references the corresponding *colorID* (see Table 1.).

3. UG3 DATA STRUCTURES

Recently, in an attempt to both simplify and reorganize *UNIGRAFIX*, a set of libraries was constructed to aid in the writing of *UNIGRAFIX* utilities. These libraries consist of routines for reading *UNIGRAFIX* files into organized data structures, manipulating the data structures in various ways, and writing them back out into an ascii format.

Ug3 data structures store everything in the form of *statements*. There are vertex statements, face statements, and even some abstract statments like an edge statement which is not defined in the language, but is an important part of the general data structure. The structure is referred to as a "winged-edge" data structure; if two faces share an edge, they will point to the same physical edge in memory, and the edge will point to both of the faces.

The central structure is called a *UGfile*. It contains all of the statements that make up the *UNIGRAFIX* object. As there are 17 different types of statements, the UGfile contains pointers to 17 doubly linked lists of statements. There is a list of verticies, a list of edges, a list of faces, and so on.

Every statement in a UGfile has so-called *reference* lists. These are lists that point to all other statements that this statement refers to, and to all other statements that refer to this statement. For example, assuming that edge e is an edge between verticies u and v. Then the edge statement e will have pointers to vertex statements u and v on its *FROM* reference list. And the vertex statements u and v will have edge statement e on each of their *TO* reference lists. These reference lists are maintained in the form of arrays. To find all of the contour statements which refer to a specfic edge, for example, you would search all of the entries in the edge's *TO* reference list (an example of this is in the "refer" document). As a general rule, if statement a makes reference to statement b, then a will be on b's *TO* reference list, and b will be on a's *FROM* reference list.

The *ug3* data structures and libraries have proven useful for quickly writing programs that modify *UNIGRAFIX* objects. Recently, ugshrink was rewritten using ug3 data structures, and the *ug3* version ended up requiring less than one quarter of the code of the original version.

4. RENDERING

One of the original goals of *UNIGRAFIX* was the production of high-resolution black-and-white output of publishable quality.[2] The aim of our rendering routines was not to imitate glossy photographs of real objects, but to render the objects in a clear way in the style of an engineering drawing. In particular, it was found that the presence of outlines around each face greatly enhances the clarity of the drawing and gives it a much crisper look when rendered on black-and-white dot-raster plotters.

The selection of a high-resolution plotter as one of the main output devices of *UNIGRAFIX* has strongly affected the choice of the algorithms for rendering and for hidden feature removal. The widest plotter available to us measures 3 feet; a square plot of that size corresponds to about 50 million pixels. The resolution of this output medium rules out 'ray casting' as a practical rendering technique. Moreover, these plotters need the output information one line at a time in y-sorted order, thus strongly favoring a scan-line algorithm. Therefore, several high-resolution renderers based on scan-line hidden-feature removal algorithms were developed during the early 1980s

More recently, with the emergence of more interactive *UNIGRAFIX* tools, some of the earlier renderers were adapted for this new environment with different constraints. Also, in addition to these renderers that run on most typical graphics raster output devices, we have recently developed a renderer for the Silicon Graphics IRIS that makes use of the special hardware used by this device. In the following the various renderers will be discussed briefly.

4.1. 'ugshow'

This is the original rendering program of *UNIGRAFIX* 1.[2] It is no longer used or supported, though you may find references to it in some old code and reports.

4.2. 'ugplot'

This renderer is an enhanced version of the 'cross' algorithm by Hamlin and Gear.[4] It is an object space algorithm that can also return visible polygons at object resolution.[5] In a single scan-line sweep, the visible edge segments are determined and properly composed into the contours of visible and invisible polygons. The algorithm concentrates on the edges in the scene, analyzes all crossings in the given projection, and relies on the coherence of planar, nonintersecting polygons to minimize the number of depth comparisons. It makes maximal use of object coherence; this makes it very sensitive to small data inconsistencies, such as non-planar polygons or intersecting faces.

4.3. 'ugdisp'

With this renderer we have returned to the more robust approach of *ugshow*, however, without incurring the penalty of the large dynamic data structure resulting from maintaining in parallel all the face lists for each segment on the scan-line. An enhanced version of the 'stack' algorithm by Hamlin and Gear[4] has been used. Special features were added to render edges and wires and to produce outlines around faces. Optional Gouraud shading has been included. The renderer can even be run in a mode where extra depth comparisons are included so that intersecting objects can be handled directly.[6]

4.4. 'uq'

This is a special purpose renderer of the '*UniQuadrix*' modeling and rendering program for objects represented as the Boolean intersection of quadric and planar half-spaces. A language very similar to the *UNIGRAFIX* format is used to define the half-space boundaries by their coefficients. *UniQuadrix* uses implicit equations to represent the surfaces and boundaries of objects throughout the rendering process. This permits a scan-line based algorithm very similar to the one used in ***ugshow*** to quickly identify visible spans. An efficient incremental algorithm shades pixels within spans.[7]

4.5. 'ugray'

This *UNIGRAFIX* ray tracing renderer was completed as a Master's project by Donald M. Marsh in 1987. In order to reduce the computational expense of general ray-tracing, uniform spatial subdivision is used. A fast algorithm for inserting polygons into the spatial subdivision and inexpensive shading and anti-aliasing routines have also been implemented. Rather than displaying an image immediately upon rendering, the image is saved in an intermediary format, ugrim, which can then be displayed later on a variety of devices.

4.6. 'ugiris'

This renderer runs only on the SGI IRIS 4D. It allows you to view and manipulate an object in real-time. It takes advantage of the special hardware on the IRIS, including the full screen z-buffer, 24 bits of color resolution, and hardware Gouraud shading, creating a highly interactive exceptional interface for real-time object viewing.

5. SOME GENERAL UTILITIES

Because of the constraints inherent in some of the renderers discussed above and in some of the filters to be presented in the next section, *UNIGRAFIX* descriptions need sometimes be converted to "simpler" descriptions, using only a subset of the expressibility of the full *UNIGRAFIX* descriptive format. A few "filter" packages provide such services.

5.1. 'ugisect'

Most of the renderers described in the previous section rely in their hidden feature elimination algorithm on the fact that the faces are planar and do not intersect one another. Scenes that contain intersecting objects need to be preprocessed once with *ugisect*[8] to convert them to a *UNIGRAFIX* description with no intersecting faces before they can be rendered. *Ugisect* can handle arbitrary collections of polygons without generating spurious edges that would subdivide unnecessarily contiguous parts of planar faces. In addition, when true polyhedral solids are involved, *ugisect* can also form set-theoretic operations, i.e., union, intersection, or difference of two objects.

5.2. 'ugxform'

This "filter" program makes a global transformation on a *UNIGRAFIX* file by simply transforming all vertices and instances at the top level of the scene hierarchy with the transformation specified on the command line. All other information is passed through unaltered. This utility can be used to transform the scene so that the default viewing option produces an optimal display. It can also be used to produce anisotropic (differential) scaling of vertex groups for use in other objects.

5.3. 'ugexpand'

This batch program expands instances and arrays recursively into their individual constituent parts. It produces a hierarchically flat description of vertex, wire, and face statements. This form is needed by some of the modifier programs that cannot cope with a hierarchical description. Optionally, the long and cumbersome hierarchical vertex names that may result in this process can be replaced with new terse (and meaningless) identifiers. Another important option is to merge all vertices within a distance *epsilon* of one another. This helps to avoid problems resulting from near intersections that may be created by numerical inaccuracies.

5.4. 'ugvmerge'

This is another clean-up filter. It merges coinciding vertices that may cause trouble for programs such as *ugplot*. All vertex pairs that are separated by less than a specifyable tolerance *eps* are merged into a single vertex. Unlike ugexpand, this filter uses the newer *ug3* library.

6. THE UNIGRAFIX LIBRARY

A rendering system alone is probably unsatisfactory for most users when not complemented by some tools to aid in the generation of interesting objects. Simple *UNIGRAFIX* objects can be readily created with a text editor. Large, but regular objects can be generated by writing small programs (in your favorite language) that produce the required ASCII files. The simple and terse format of the *UNIGRAFIX* language as well as its hierarchical nature make both these approaches quite practical.

6.1. Basic Polyhedron

The *UNIGRAFIX* library contains simple geometric primitives that are frequently used as the starting point for the generation of objects. They comprise the Platonic solids in 3-D and 4-D space. In addition, this library contains many of the archimedean solids and some of the objects that are depicted in this manual.

6.2. Generator Programs

In the following we present some examples of programs that create *UNIGRAFIX* object descriptions from scratch based on some user-supplied parameters or data files. These programs have typically been developed by students as course projects in graduate courses on computer graphics and solids modeling.

'ugsweep' sweeps a polygon through space with an arbitrary incremental transform between steps and produces the surface of the swept-out volume.

'mkworm' creates properly mitred prismatic tube sections around piece-wise linear paths through 3-D space. These paths are read in from an 'ax'-file and can form closed loops but must not include branches.

'mktree' outputs, in the above mentioned 'ax-file' format, the joint-coordinates of a tree-like object based on the growth algorithm of Kawaguchi.[9] The shape of the generated structure can be altered by a set of command-line options to make them resemble trees, shells, or corals.

'mkstairs' creates helical staircases or ramps according to a set of parameters. Each step is an instance of a definition describing a single step or ramp segment with the proper geometry for a smooth fit.

'mkgear' produces a *UNIGRAFIX* description of gear boxes based on the specification of position and size, of gear wheels and shafts.

'mkrobot' is a generator program that reads the predefined parts of a robot arm from the file ~ug/lib/rbparts, takes the values of the various position parameters from the command line, performs some checking on the size and ranges specified, and produces a *UNIGRAFIX* description of the complete manipulator arm.

'mkquasi' generates two or three dimensional quasiperiodic tilings based upon methods described in a 1985 paper by J. E. S. Socolar.[10] The user may specify some or all parameters using a special input file format.

'ugsubd' is a generic object generator. It creates a *UNIGRAFIX* representation of a parametric surface (x(u,v), y(u,v), z(u,v)) defined by the user. The surface is approximated by a collection of triangular facets which are generated through adaptive subdivision. This process produces good representations of smooth surfaces.

6.3. Modifier Programs

Other programs start from an existing *UNIGRAFIX* description to produce a new object, either by such processes as projection or truncation, or by modifying each individual face of the polyhedral object in some specified way:

'ugshrink' separates the faces of a polyhedron and shrinks them individually by a specified factor with respect to the face center. It can also be used to cut similarly shaped holes into faces or to produce concentric rings.

'ugfreq' subdivides triangular faces into a tessellation of similar, but smaller facets. The degree of subdivision is specified by the 'frequency' parameter.

'ugtess' is a filter that tessellates the faces of an arbitrary unigrafix object into convex polygons without creating any new vertices. An option exists to triangulate the faces instead.

'ugstar' constructs pyramid-shaped extrusions or intrusions on all faces of a polyhedron. The tip of the pyramid lies at a parameterized distance on the face-normal through the face-center.

'ugtrunc' truncates the corners of a polyhedron. New vertices are formed either in the middle of every edge or at a parameterized distance from the ends. These new vertices are then linked in a circular manner around every old vertex to form the new faces. To guarantee planar truncation faces, an approximate plane is first placed through all the vertices determined in the above manner on the edges emerging from a particular vertex; then the truncation plane is moved through the new vertex that minimizes the distance of the plane from the old vertex. Vertices with emerging edges that occupy more than a half-space (saddle points) will not get truncated.

'ugsphere' projects all vertices radially from the origin onto a sphere of a given radius around a specified center point. This is useful to construct geodesic domes.

'ugpipe' produces ball and cylinder descriptions in the *UniQuadrix* descriptive format. It starts from standard *UNIGRAFIX* scene descriptions and converts all vertices into balls and all wire segments and face edges into cylinders. The output contains all of the quadric and planar descriptions necessary to render the object with *UniQuadrix.*

'ug4to3' projects 4-dimensional vertex coordinates into 3-dimensional space. It applies to each vertex the specified transformation. The default transformation is a parallel projection along the w-axis, i.e., simply a removal of the w-component. Face, wire, color, and light statements are passed unaltered to the output.

'ugunwrap' takes a polyhedral object and unwraps it into a flat plane. After unwrapping the object, it is then possible to print it, cut it out, and fold it up so as to create the actual object.

'ugfold' is the reverse of ugunwrap. Given a file describing a net of polygons, it will generate a *UNIGRAFIX* file describing the resulting polyhedron.

'ugcontour' takes an object described with triangular faces, and generates topological contours along any specified axis.

'ugcull' is a program which reduces the number of polygons in a *UNIGRAFIX* model, making the model cruder, but simpler.

'uginterp' generates quadratic interpolating patches over a polyhedron. The patches are designed to fit so that they meet with G1 continuity.

'uginbetween' reads two or more *UNIGRAFIX* scenes, and interpolates between them, producing a series of output scenes which may then be rendered to form a movie.

'ugsoap' generates minimal surfaces. Given an object with unpaired edges, it will deform the existing surfaces in the object to generate a minimal area surface between all unpaired edges.

'ugworm' generates mitered prismatic tubes for a wire frame. This is like the mkworm generator program, except that it can handle intersections of more than one edge.

'bump' stands for Berkeley *UNIGRAFIX* Movie Package. Bump makes extensions to the language having to do with time and motion, and can generate a sequence of movie frames.

6.4. Modifier Pipes

In typical UNIX style, the described filter and generator programs can be piped into one-another to form powerful scripts. The examples below show how the objects were generated as well as how they were rendered. Note the variety of objects that can be generated by starting from some of the same very simple primitives.

Figure 4.

```
cat ~ug/lib/dodeca | ugstar -h 3 | ugtrunc -t 0.9 | ugtrunc -C -t 0.7 > f4
cat f4 illum | ugplot -ep -25 60 -100  -sa -dw -sy 3 -sx 2.75
```

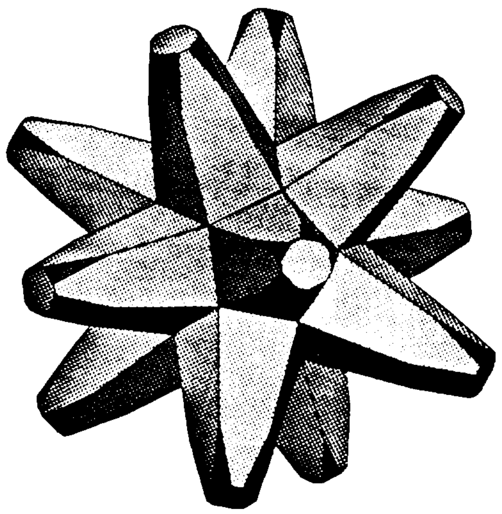


Figure 5.

```
cat wire | ugsweep -n 9 -tx 6 -rz 30 -tx -6 -n 9 -tx -6 -rz -30 -tx 6 | ugxform -tx 6 -ty 6 | ugshrink -f 0.8 | ugsweep -tz 10 > f5
cat f5 illum | ugplot -ep -50 30 -100  -sa -dw -sy 3 -sx 2.75
```

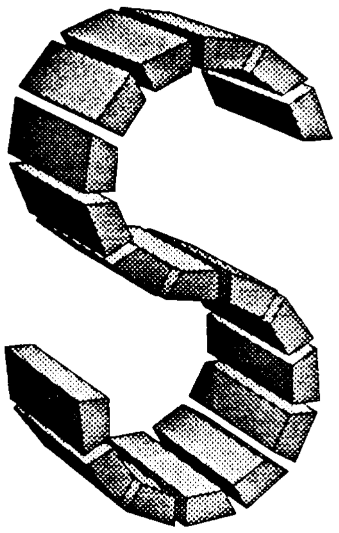


Figure 6.

```
cat ~ug/lib/icosa | ugfreq -f2 | ugsphere | ugshrink -f0.8 -H > f6
cat f6 illum7 | ugdisp -ep -60 30 -100  -ab -sa -sg -dw -sy 3 -sx 2.75
```

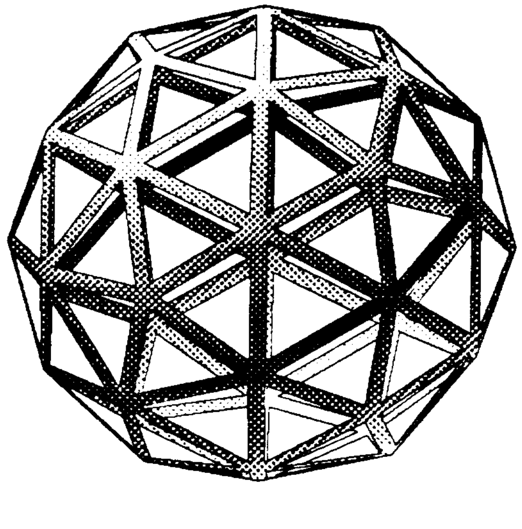


Figure 7.

```
cat ~ug/lib/D4cube | ug4to3 -ep 0 0 0 3 | ugpipe -rb 0.3 -rc 0.15 > f7
cat f7 illum8 view8 | uq
```

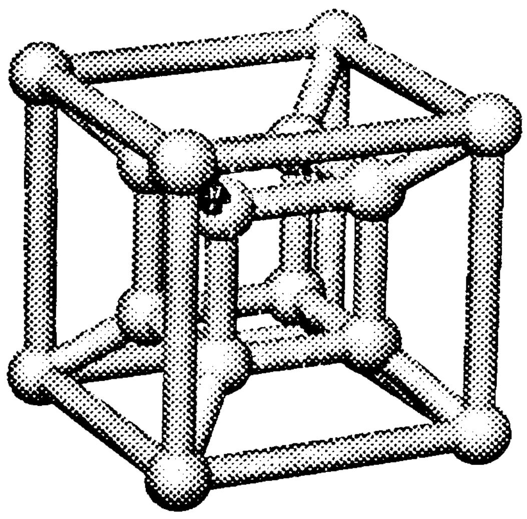


Figure 8.

```
cat dodeca | ugworm -r 0.15 -n 6 > f8
cat f8 | ugplot -sa -ep -20 -2 -20 -sx 3 -sy 3
-dp
```

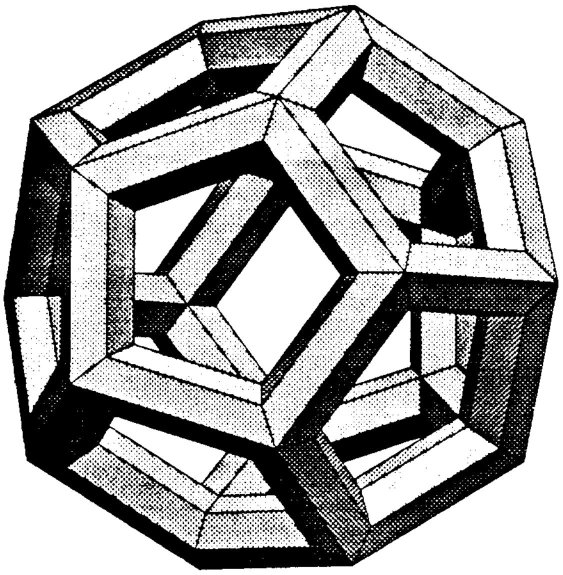


Figure 9.

```
cat ~ug/lib/cube | ugshrink -f 1.3 | ugshrink
-H -f 0.6 | ugisect > f9
cat f9 illumc | ugplot -ep -6.5 5 -10 -ab -sa
-dw -sy 3 -sx 3
```

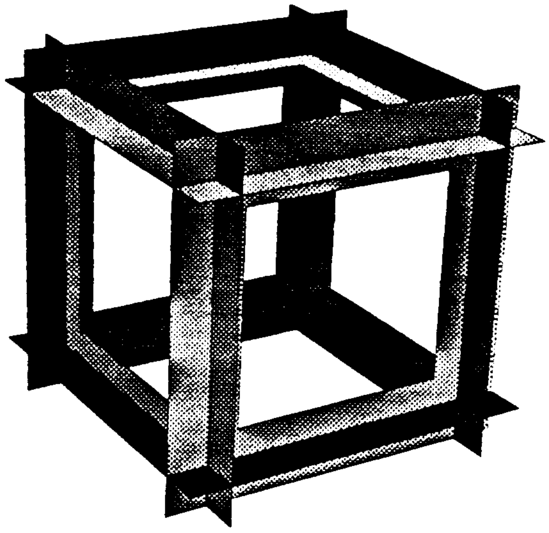


Figure 10.

```
cat icosa | ugtrunc -t0.666 | ugunwrap | ug-
star -h 1.5 -N > f10
cat f10 | ugplot -ep 0 50 -30 -sa -sx 3 -sy 3
-dp
```

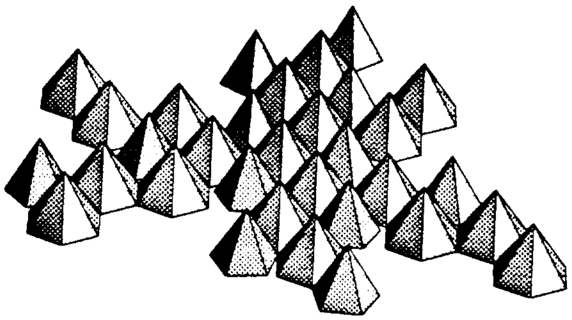
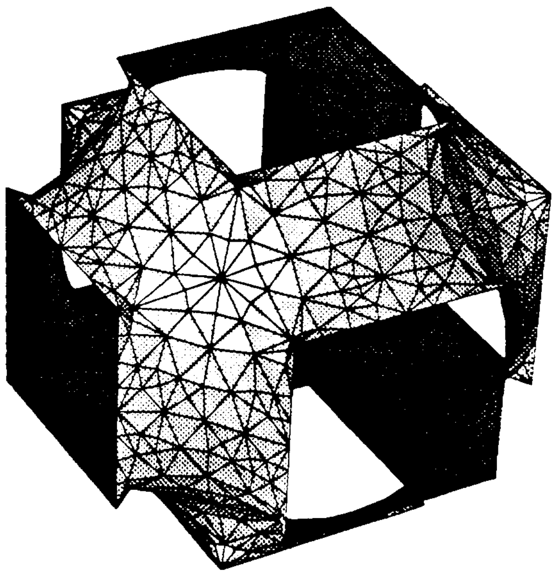


Figure 11.

```
cat cube | ugshrink -f 0.5 -H | ugsoap > f11
cat f11 | ugplot -ep -6 8 -10 -sa -ab -dp -sx 3
-sy 3
```



7. INTERACTIVE VIEWING

In some sense, the *UNIGRAFIX* system construction has been started at the back end, providing the rendering programs first. Ugi, a prototype for an *UNIGRAFIX* interactive shell was written in 1985. More recently the development of "hug", a more advanced shell has been started.

7.1. The Original Shell 'ugi'

This interactive environment for the display of *UNIGRAFIX* scenes provided most of the old *UNIGRAFIX* batch capabilities for scene manipulation, view specification, and display style within a homogeneous interactive framework. It greatly enhanced the speed and ease with which *UNIGRAFIX* objects can be designed, viewed, and modified.

7.2. A Newer Interactive Shell, 'hug'

Hug is an emerging new interactive shell based on the ug3 data structures discussed in section 3. It has been designed with two primary goals in mind. First, the *UNIGRAFIX* data structure representing the current scene is passed between the various tools and the renderers without ever being converted into an ASCII format. The second goal has to do with the ease of adding new tools to the shell. Any tool written using the ug3 data structures, and following a few simple conventions can easily be added to the shell. Note that tools written prior to the ug3 data structures can still be used in hug by converting the scene to and from ASCII.

7.3. A Shell for Curved Surfaces, 'uci'

Uci is an editing, viewing, and designing tool for *UniCubix* objects. It provides most of the *ugi* capabilities for objects with curved edges and curved surface patches.

7.4. Animator

Animator is a graphical tool that runs only on an SGI IRIS. It allows users to interactively view *UNIGRAFIX* objects and to dynamically apply certain *UNIGRAFIX* tools to them. The user can interactively change the parameter value of a modifier tool and study its effect on the resulting object.

8. CONCLUSION

UNIGRAFIX privides an enhancement of the UNIX environment; it makes three main contributions: First, it presents a terse ASCII-based language for the description of geometrical scenes at the object database level.

Second, it provides an efficient rendering system for high-resolution views of polyhedral objects. It produces hardcopy output in the style of engineering drawings, rather than refined displays simulating photographic renderings of real objects.

Third, it offers a collection of generator and modifier programs and an interactive viewing shells that make it easy for the user to create rather complex objects with a command-line pipe or with a small shell script. The currently available utilities are primary aimed toward geometric objects such as semi-regular polyhedra and lattices in three and four dimensions.

UNIGRAFIX is being made available to people for their own use and at their own risk. These programs form in no way a "turn-key system." We believe they are a basis from which others can start their own experiments, and perhaps a guide how such a system could look, once all the parts have been honed to perfection.

ACKNOWLEDGMENTS

The *UNIGRAFIX* system would never have come into existence without the dedicated effort of many graduate students at U.C.Berkeley. Special pioneering contributions have been made by Paul Strauss, Paul Wensley, Mark Segal, Nachshon Gal, Michael Natkin, and many students who took the course CS285, "Procedural Object Generation" in the last few years. This development has been supported at various times by Tektronix, Inc., Siemens Corporation, and Silicon Graphics Inc..

References

1. C.H. Séquin, ''Creative Geometric Modeling with UNIGRAFIX,'' Tech. Report (UCB/CSD 83/162), U.C. Berkeley, Dec. 1983.
2. C.H. Séquin and P.S. Strauss, ''UNIGRAFIX,'' *Proc. 20th Design Automation Conf.*, pp. 374-381, Miami Beach, FL, June 1983.
3. H. Chiyokura, *Solid Modeling System DESIGNBASE - Design and Implementation*, Addison-Wesley, Singapore, Spring 1988.
4. G. Hamlin and C.W. Gear, ''Raster-Scan Hidden Surface Algorithm Techniques,'' *Computer Graphics*, vol. 11, no. 2, pp. 206-213, Summer 1977.
5. C.H. Séquin and P.R. Wensley, ''Visible Feature Return at Object Resolution,'' *Computer Graphics and Appl.*, vol. 5, no. 5, pp. 37-50, May 1985.
6. N. Gal, ''Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX,'' Master's Report, U.C. Berkeley, Jan. 1986.
7. G.K. Ressler, ''UniQuadrix,'' Master's Report (UCB/CSD 85/240), U.C. Berkeley, June 1985.
8. M.G. Segal, ''Partitioning Polyhedral Objects into Non-Intersecting Parts,'' Master's Report (UCB/CSD), U.C. Berkeley, Spring 1986.
9. Y. Kawaguchi, ''A Morphological Study of the Form of Nature,'' *Computer Graphics (Siggraph'82 Conf. Proc.)*, vol. 16, no. 3, pp. 223-232, 1982.
10. Socolar, J.E.S., et al., *Quasicrystals with arbitrary orientational symmetry*, 1985. Phys Rev B, 32:8. p 5547