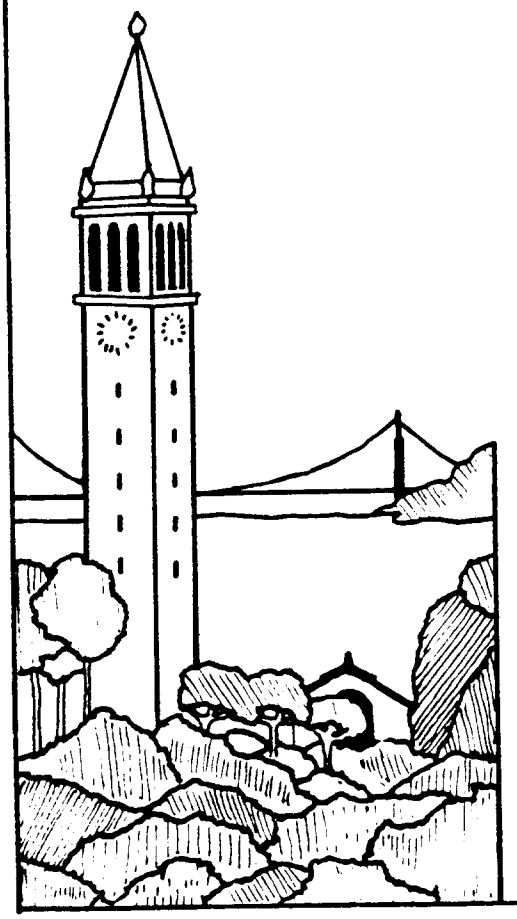
MORE . . .
CREATIVE GEOMETRIC MODELING

Carlo H. Séquin



Report No. UCB/CSD 86/278

December 1985

Computer Science Division (EECS)
University of California
Berkeley, California 94720

MORE ...
CREATIVE GEOMETRIC MODELING

Carlo H. Séquin

With contributions by:

Cecilia Aragon
Rajiv Bhateja
Eric Fan
Dan Filip
Philip D. Flanner
Ziv Gigus
Nachshon Gal
Mark Gerolimatos
Thomas Laidig
Lucia Longhi
Don Marsh
Gene Ressler
Jim Ruppert
Mark Segal
H.B. Siegel
Lun-Shin Yuen

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

ABSTRACT

This is a report on the second offering of the graduate course CS 292A, "Creative Geometric Modeling" first offered in Fall 1983. As part of the course, the students developed some new generator, modifier, and general utility programs for the UNIGRAFIX system. These programs were also used to create artistic displays.

This report gives a brief overview over the course syllabus, introduces some of the new UNIGRAFIX programs, and demonstrates on several examples what these new tools can be used for.

1. CS 292A, CREATIVE GEOMETRIC MODELING

In 1983 a new graduate course, "Creative Geometric Modeling" was added to the catalog of our offerings in the area of computer graphics.[1] The main goals of this course were:

a) To satisfy the increasing demand from students to learn more about the fast-moving and ever more pervasive field of computer graphics.

b) To develop the spatial perception of the participants and to train their skills in geometry.

c) To give the recently developed UNIGRAFIX system a hard work-out to identify bugs in the algorithms and in the user interface.

d) To enhance the UNIGRAFIX environment by adding facilities that make it easier to create complex geometrical objects.

In spring of 1985 this course was offered for the second time. There were 14 formally enrolled students, 10 graduates and 4 undergraduates, some of them from mechanical and civil engineering, as well as a few auditors.

1.1. Syllabus

The course extended over 15 weeks with 1.5 lecture hours per week. The course had formal homeworks during the first half of the term and concentrated on individual course projects during the second half. Below is a rough outline of the lecture and discussion topics and of the homework assignments on a week by week basis:

1) Importance of geometric skills and spatial perception.
Course goals, class format, deliverables, background questionnaire.
Exercises in visualizing 3-D geometric objects and constructions.
Role of creativity; handling problems with ambiguous solutions.
Course-related literature, inspiring books.
Hilbert curve in 2D
Assignment: Find solution for Hilbert curve in 3D.
Design a clover leaf for a 6-way highway intersection.

2) Introduction to UNIGRAFIX, object representation, toolbox.
Discussion of first homework; evaluating the quality of a solution.
Recursive formulation of the 3D Hilbert curve.
Recursive fractal surfaces derived from Sierpinsky curve.
Assignment: Study 'mkworm', Hilbert curve with mkworm.
Design a recursive surface inspired by Sierpinsky curve.

3) Difficulty of creating object descriptions, need for generators.
The generator program 'mkworm'; axfile specifications, worm parameters.
Different approaches for the construction of recursively defined surfaces:
direct specification of surface in boundary representation by local deformations
versus CSG description and postprocessing with 'ugisect'.
Assignment: Recursive path through 3D inspired by C-curve or Dragon curve.
Procedural description of a gear wheel: polygon and swept object.

4) The algorithms and data structures of 'mkworm'.
Mitring at the joints, use of a suitable coordinate system.
Closed loops in 'mkworm'; simple knots: trefoil and cinquefoil knots.

End-to-end axturn: can this be determined by analyzing the individual local joints ?
Assignment: Construct tight tetrafoil knot with 'mkworm'.

5) Knot constructions with 'mkworm'.
The analysis tools in 'mkworm': axspec and axdist.
Tight knots, constraints on ax distances; clover leaf knots.
Matching the edges in prismatic joints; multiple pipe joints.
Assignment: Construct tight cinquefoil knot with minimal axlength.

6) Experimental and constructive approaches to cinquefoil construction.
Knot-tightening by simulated annealing; movements, cost functions.
Other knots; square knots, granny knots, torus knots.
Assignment: Select and outline your course projects.

7) Introduction to the Platonic solids in 3D
Construction of exact vertex coordinates. Symmetry groups.
Archimedean solids and their duals.
The algorithms in 'ugshrink' and 'ugtrunc'.
Assignment: Modification of platonic solids with 'ugtrunc','ugdual','ugfreq'...
Use mkworm and symmetry of platonic solids to create polylink.

8) Euler relations; convex polyhedra, arbitrary polyhedra of any genus.
More on modifier programs for polyhedra.
Quiz: How do you derive the structure of C10-H16 from knowledge of regular polytopes ?
Assignment: Prepare project proposal.

9) Construction of platonic solids in 4 and higher dimensions.
Simplex series, n-cube series (measure polytope), dual to n-cube (cross polytopes)
Enumeration of all regular polytopes in four dimensions.
Due date for: Project proposals.

10) Mental fitness quiz on hyperspace.
Projecting 4-dimensional objects to 3-dimensional space.
Construction of the more complicated 4-dimensional regular polytopes.
Assignment: Prepare formal proposal presentation.

11) Formal 5-minute project proposal presentations.
The purpose is to improve presentation skills.
Conceptual setting: a short interview with a venture capitalist.
You must convince him in 5 minutes to fund your proposal for a new tool.
You need: Viewgraphs, slides, posters ...

12) Discussion of results of "venture capitalist rally."
Discussion of content, form, and impact of presentations.
How to prepare a manual page for your program.
Special problems in your projects: Naming of new vertices in tessellations.
Due date for: Progress reports.

13) More on special problems in your projects.
Useful data structures in 'ugplot' and 'ugdisp'.
Random fractal surface generation, smoothing operations.
Quiz on geometry and spatial perception.
How to finish your project; how to prepare for art show.
Due date for: Formal manual pages.

14) Transformations between Platonic and Archimedean solids.
The translation-rotation operation ("snub" operation).
Toroids, Moebius bands, Klein bottles.
Toroids from regular polygons; minimal toroids.
Top priority: Your project.

15) Special session on paper folding of regular polyhedrons.
Course wrap-up at LaVals.
Due date for: Course projects.

16) Artshow and Reception.

2. THE NEW UNIGRAFIX PROGRAMS

This section gives a brief introduction to the programs developed or completed during this course offering. For an overview over the UNIGRAFIX system see the original CS 292A course report[1] or a short overview over the Berkeley UNIGRAFIX Tools.[2] Some of the larger and more complicated programs are also described in detail in separate reports.[3,4,5,6]

2.1. Generator Programs

We start with the programs that create an object description in UNIGRAFIX format from scratch:

`mkgear' (Laidig)
produces a UNIGRAFIX description of gear wheels or whole gear boxes based on the specification of position and size, of gear wheels and shafts.

`mkrobot' (Fan)
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.

`mkhouse' (Bhateja)
constructs walls, windows, doors, and a roof of a simple parameterized house.

`mkcity' (Gerolimatos)
is a city-sprawler that will generate a "downtown area" of a city in a random fashion. It supports three types of prism-based buildings. Other parameters control the block dimensions, number of blocks in the scene, street width, and the maximum and minimum height of the buildings.

`mklife' (Ruppert)
gives a 3-dimensional rendering of successive generations of cells in John Conway's game of Life.

2.2. Modifier Programs

Other programs act as filters; they start from a UNIGRAFIX description and produce a new object:

`ugtess' (Gigus, Longhi)
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.

`ugfrac' (Yuen)
subdivides all triangular faces into four new faces, where the three new vertices corresponding to the middle of the edges are displaced by a random amount. Recursive application creates wrinkled surfaces suitable to represent landscapes.

`ugext' (Siegel)
extrudes the vertices of one object by ratioing their distance from the origin with a second closed surface. Useful mainly for the creation of unexpected artistic objects.

`ug4Dprism' (Filip)
creates a 4-dimensional prism by sweeping the specified 3-dimensional solid along the fourth coordinate axis.

`ug4Dpyr' (Filip)
creates a 4-dimensional pyramid on top of the specified 3-dimensional solid.

`ug4Drev' (Filip)
creates a 4-dimensional solid of revolution by rotating the specified 3-dimensional solid along the z-w plane.

`ugpipe' (Flanner, Marsh)
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*.

`ugtight' (Aragon)
uses a simulated annealing technique to move the joints of a knot in space with the goal to minimize its string length without creating intersections or changing its topology.

`ugisect' (Segal)
converts a scene description that contains intersecting faces into a proper UNIGRAFIX description, where faces only intersect along joint edges. As such it is a preprocessor for the renderers that rely on the coherence of planar non-intersecting faces in order to minimize depth comparisons in the hidden feature elimination. 'ugisect' can also perform Boolean set operations on two solids and return the union, difference, or intersection of the two objects.

2.3. Renderers and Utilities

`ugi' (Gal)
is an interactive shell for assembling and displaying UNIGRAFIX scenes. Many of the older batch facilities of the UNIGRAFIX system have already been incorporated into

this program and can run directly on the internal data structures of 'ugi' without the need to convert back and forth to the UNIGRAFIX ascii format.[5]

`ugdisp´ (Gal)
is a new renderer that combines some of the best features of 'ugshow' and 'ugplot'. In addition it can provide smooth Gouraud shading for polyhedral objects and it can handle intersecting objects directly.[7]

`uq´ and `UniQuadrix´ (Ressler)
provide an extension of the UNIGRAFIX domain to curved objects. 'UniQuadrix is a language very similar to the UNIGRAFIX format that describes arbitrary quadric half-spaces and planes. 'uq' is the corresponding renderer that will produce smooth-shaded images of objects that can be described as the union of pieces formed by the intersections between one quadric half-space and several planar half-spaces.[8]

`ugman´ (Ressler)
is the general facility to run off one of the manual pages from the UNIGRAFIX library of tools. It corresponds to the UNIX 'man' command.

2.4. Program Documentation

A detailed description of some of the above programs is available in the form of Technical Reports or Master's Theses.

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, "The Berkeley UNIGRAFIX Tools, Version 2.5," Tech. Report (UCB/CSD 85/), U.C. Berkeley, Jan. 1986.
3. C.H. Séquin, M.G. Segal, and P.R. Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial," Tech. Report (UCB/CSD 83/161), U.C. Berkeley, Dec. 1983.
4. P.R. Wensley, "Hidden Feature Elimination and Visible Polygon Return in UNIGRAFIX 2," Master's Report (UCB/CSD 84/172), U.C. Berkeley, May 1984.
5. N. Gal, "The ugi Shell for UNIGRAFIX," Technical Report (in preparation), U.C. Berkeley, Spring 1986.
6. H.B. Siegel, "Jessie: An Interactive Editor for Unigrafix," Master's Report, U.C. Berkeley, Dec. 1985.
7. N. Gal, "Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX," Master's Report, U.C. Berkeley, Jan. 1986.
8. G.K. Ressler, "UniQuadrix," Master's Report (UCB/CSD 85/240), U.C. Berkeley, June 1985.

The following section presents the manual pages for the various new generator and modifier programs and for some other new utilities. In a few cases, these are followed by tutorial examples to show the variety of effects that can be produced with these programs.

NAME

mkcity – Generate a random city

SYNOPSIS

mkcity [*–d <block length, block width>*]
[*–b <# of blocks long, # blocks wide>*]
[*–s <street width>* *–h <height>*] [*–f <command file name>*]

DESCRIPTION

MKCITY is a city-sprawler which will "grow" a downtown area of a city in a random fashion. Currently, mkcity supports three main types of buildings (box, "random" (sort of like the Sears Tower in Chicago), and stepped), although the internal representation does not necessarily prevent non-square geometries from being used.

The constraints on the generator are *maximum height, minimum height, block dimensions, number of blocks in the scene, and street width.* In addition, the user may "seed" the scene with parksUNIGRAFIX and "high rent districts". The park seeds affect the minimum heights of the buildings, while the high rent seeds affect the maximum size. Otherwise, height of the buildings is quite psuedo-random.

COMMAND LINE ARGUMENTS

–d *<block length, blockwidth>*

or **blocksize** *<block length, blockwidth> in command file*

Sets the physical demensions of the blocks: the length and width. Currently, all blocks must be of the same dimension.

–b *<# of blocks long, # of blocks wide>*

or **citysize** *<# of blocks long, # of blocks wide> in command file*

Gives the number of blocks in the city...length and width. Default is 300 x 100.

–s *<width of street>*

or

Specifies the width of the street (seperation between blocks). Default is 50.

–H *<height of building in stories>*

Specifies maximum height of each building. Default is 50 (20ft) stories.

–h *<minimum height of building in stories>*

Specifies minimum height of each building. Default is 2.

-Sc *<posx posy> <severity (0->1)>*

Specifies a high rent seed of severity at <x,y>.

-Sp *<posx posy> <severity (0->1)>*

Specifies a park seed of severity at <x,y>.

–o *<output file name>*

Specifies output to go to an output file. Default is stdout.

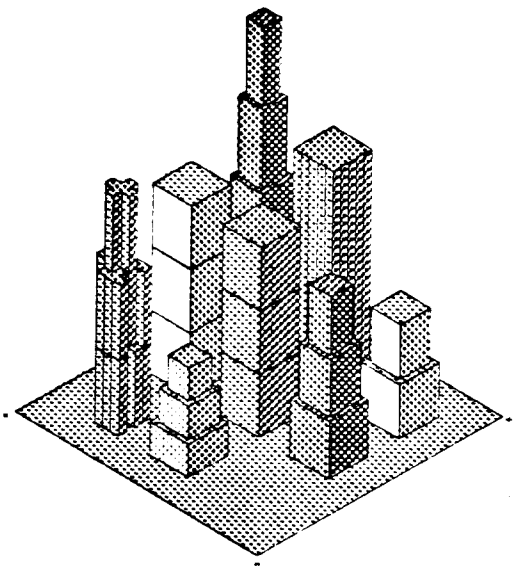
–f *<command file name>*

Take file <command file name> as input

Note that while buildings are given by height, houses are stored in "storied" form, and their height will automatically be adjusted by the sprawler to look in proportion to the block size.

EXAMPLE

mkcity -b 3 3 -d 100 100 -Sp 0 0 .6 -Sc 2 2 1 -s 80 -H 50 -h 3 -o city cat ~ug/lib/illum city | ugplot -ed -1 1 -1 -sa -dv -sy 3



FILES

~ug/bin/mkcity

BUGS

Causes ugdisp to freak out in a massive way. Really can't figure out why. If scenes get too large, ugplot may also have problems. To help remedy this, try "-F <fudge factor>" to seperate pieces of buildings, slightly.

AUTHOR

Mark Gerolimatos

NAME

mkgear — generate graphical descriptions of meshing gears

SYNOPSIS

mkgear [—N] [—a rotate-angle] [—s shaft] [—f facet-curvature]

DESCRIPTION

Mkgear takes a concise definition of shafts and gears from standard input, and produces a *unigrafix* description of them on standard output. Any unrecognized input commands are passed unchanged, so normal *unigrafix* commands included in the input (such as light sources and descriptions of other objects to occupy the scene with the gears) will be passed through without change. Normally, no command line options are needed, but the following can be used to modify *mkgear*'s behavior:

—N Do not generate descriptions of gears. Just pass through the input with comments describing the gears that would be generated.

—a *angle*
Rotate a shaft by *angle* degrees clockwise (as viewed from the first end of the shaft). By default, the first shaft described in the input is rotated, but this can be changed with the —s option.

—s *shaft*
Instead of rotating the first shaft in the input file, use the shaft named *shaft*.

—f *angle*
Set the maximum facet curvature angle to *angle* degrees. *Mkgear* will divide the meshing surface of each gear tooth into facets so that the arc approximated by each facet has less than *angle* degrees of curvature. The default value of 25 degrees is usually adequate; larger values result in coarser gears, but values much smaller can cause *mkgear* to produce unconscionable amounts of output. As a special case, an *angle* of 0 forces *mkgear* to use a single facet for each surface, which is useful for test runs.

The input description consists of **vertex**, **shaft**, **gear**, **cluster**, and **box** commands. The **vertex** command is the standard *unigrafix* **vertex** command, and is copied to the output unchanged after being assimilated:

v vname xcoord ycoord zcoord **;**

where *vname* is the name of the vertex, and *xcoord*, *ycoord*, and *zcoord* are the coordinates. All gears must be on shafts, which are defined with the **shaft** command:

shaft sname (vert1 vert2 diam) [—**flat** | —**key** | —**hex** | —**spline** nsplines] [—**inv**] **;**

where *sname* is the name of the shaft, *vert1* and *vert2* are vertices at the two ends of the shaft, and *diam* is the diameter of the shaft. By default, the shaft is round and is welded to its gears (i.e. — there is no indication of what fastens them together), but it can be made flatted, keyed, hexagonal, or splined by specifying the —**flat**, —**key**, —**hex**, or —**spline** option, respectively. For splined shafts, the number of splines is given by *nsplines*. The shaft can also be made invisible, by adding the —**inv** option, allowing gears to be drawn floating in space. Once a shaft is defined, a gear can be put on it with the **gear** command:

gear gname (sname vert nteeth thickness) [—**hub** hubdiam hubthick] [—**inv**] **;**

where *gname* is the name of the gear, *sname* is the name of its shaft, *vert* is a vertex in the same plane as the gear (e.g. — the vertex where a pair of gears meshes), *nteeth* is the number of teeth on the gear, and *thickness* is its thickness. If the —**hub** option is given, the gear is drawn with a hub *hubdiam* in diameter and *hubthick* in thickness. A hubbed gear on a flatted, keyed, or hexagonal shaft is drawn with an Allen setscrew on the hub.

toward the first end of the shaft. The gear can be made invisible with the **—inv** option, which allows other gears to be drawn as if this gear were here. Once the gears are defined, it is still necessary to describe which gears mesh together. This is done with the **cluster** command:

cluster [cname] (gear1 gear2 ...) ;

where *cname* is the optional name for this gear cluster, and *gear1* and *gear2* are the names of two gears that mesh. A cluster can have more than two gears, in which case each gear meshes with the gear named before it and the one named after it. *Mkgear* computes the diameter and bevel angle of each gear based on the positions and directions of the shafts and on the gear ratios involved. If a gear must be two different sizes to mesh with each of two other gears, it is drawn correctly for one of its neighbors, and an error is reported. Note that no gear can be drawn unless there is another (possibly invisible) gear meshing with it.

In addition to gears and shafts, *mkgear* can produce complete or partial boxes, as directed by the **box** command:

box [bname] (vert1 vert2 vert3 ... [**—bearing** shaft1 ...]) ... thickness ;

where *bname* is an optional name for the box, *vert1*, *vert2*, *vert3*, etc. are the vertices that make up one wall of the box (listed in clockwise order as seen from the outside of the box — actually, the choice of outside and inside is arbitrary, but all connecting box walls must agree on which side is the inside and which is the outside), and **—bearing** *shaft1*, etc. list all shafts (if any) that penetrate that wall of the box. The walls of the box are expanded symmetrically around the defined faces to a thickness of *thickness*. For box walls to mate properly, all connecting walls must be defined in a single **box** command.

EXAMPLE

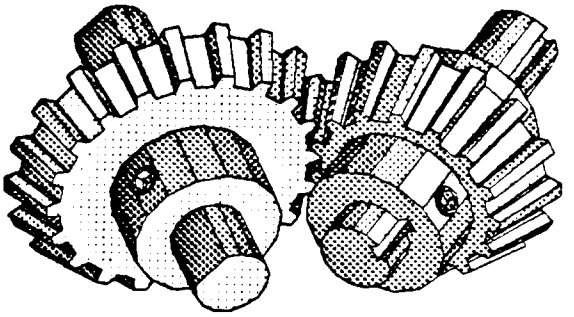
If the file *demo* contains

```
1 .6 -2  2 -2;
1 .6  2  2 -3;
1 .2;

v shaft1start  0  0  5;
v shaft1end  0  0 20;
shaft shaft1 (shaft1start shaft1end 3) -flat;
v shaft2start  5  0  5;
v shaft2end 15  0 15;
shaft shaft2 (shaft2start shaft2end 3) -key;

v meshpoint  6  0 12;
gear gear1 (shaft1 meshpoint 20 2) -hub 6 8;
gear gear2 (shaft2 meshpoint 15 3) -hub 6 8;
cluster (gear1 gear2);
```

then the command

mkgear -s shaft1 -a 135 <demo | ugplot -sa -ed -1 3 -3 -dv -sy 3

produces



FILES

~ ug/bin/mkgear	executable
~ ug/src/mkgear/*	source code
~ ug/lib/geardemo	demonstration

SEE ALSO

ugplot (UG)

DIAGNOSTICS

The diagnostics are intended to be self-explanatory, and include the line number where the error was encountered. If this number is greater than the number of lines in the file, the error was discovered after the input was completely scanned — usually an object name is given to help locate the source of trouble.

Most errors are nonfatal, in that *mkgear* will continue to process input to look for more errors. It will also attempt to produce output correctly describing the part of the input scene it understood.

Mkgear gives a return code of zero unless it encountered a fatal error.

BUGS

The width of each gear tooth is chosen so the gear can mesh with any other gear. This allows considerable lash between two gears with only a few teeth each.

The current algorithm for generating vertex names allows only $26^2=676$ vertices in any one ring. This limits the maximum number of teeth on a gear to 169. If the facet curvature angle is set to less than 11 degrees, some gears with fewer than 169 teeth will not work.

The gears on a keyed shaft also have keyways cut into them, but no key is created. This is not normally a problem, since the key is likely to be hidden from view anyway.

Shafts cannot penetrate box sides at other than right angles.

No more than three box walls can join at a vertex.

In fact, boxes don't work at all yet.

AUTHOR

Tom Laidig

NAME

mkhouse - Create a UNIGRAFIX model of a building

SYNOPSIS

mkhouse [*options*] < input > output

DESCRIPTION

mkhouse reads data defining the floor plan and locations of doors and windows of a building from standard input. It then generates a model of the building in UNIGRAFIX format. Internal walls are not considered at present.

The first line of the input must contain the height of the building and its wall thickness. The floor plan is defined as a polygon in the x-z plane with no branches or overlaps. The top of the building is directed along the positive y axis. (Note that *mkhouse* uses the standard left-handed UNIGRAFIX coordinate system.) The floor plan polygon is defined by specifying a list of nodes in the x-z plane as a two-dimensional grid in clockwise order when viewed from the positive y half-space. A typical nodal specification is given below:

n <node number> <x-coord> <z-coord>

The first and last nodes input for the floor plan are automatically joined to obtain a closed polygon.

Windows and doors may be specified for the walls of the building. For example:

w <node-1> <node-2> <r> <s> <width> <height>

defines a window in the wall spanning node-1 and node-2, where r and s are factors between 0.0 and 1.0 specifying the location of the center of the window on the wall. For each wall, the r-axis extends horizontally to the right from the lower left-hand corner of the wall (viewed from outside), and the s-axis extends vertically upwards from the same point.

Similarly, a door is defined as:

d <node-1> <node-2> <r> <s> <width> <height>

Available options:

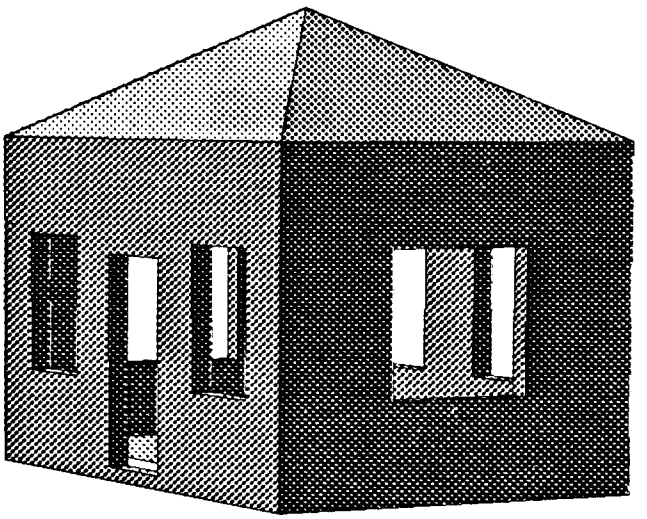
-s *<roof height>* Creates a sloping roof of the specified height instead of the (default) flat roof.

EXAMPLE

mkhouse -s 40 < housespec > house cat house illumF | ugplot -ed 900 100 -500 -sa -dw -sx 4.5 -sy 5

Contents of housespec:

```
90 6
n 1 80 -50
n 2 -80 -50
n 3 -80 50
n 4 80 50
w 1 2  0.2 0.5 30 40
w 1 2  0.8 0.5 30 40
d 1 2  0.5 0.35 30 60
w 2 3  0.5 0.5 40 40
w 4 1  0.5 0.5 40 40
w 3 4  0.3 0.5 40 40
w 3 4  0.7 0.5 40 40
```



FILES

~ug/bin/mkhouse, ~ug/src/generators/pascal/mkhouse.p

SEE ALSO

mktree (UG), mkstairs (UG), ugplot (UG)

BUGS

It's a good idea to make the walls thickness larger than life. Otherwise outer and inner wall faces might not be distinguishable with the available device resolution.

When using the -s option, a space must be left between the letter "s" and the value for the roof height.

Doors and windows that adjoin walls, the floor or the roof result in coplanar faces that are distasteful to ugplot.

AUTHOR

Rajiv Bhateja

NAME

mklife – generate 3-D display of 2-D game of life

SYNOPSIS

mklife [*options*] [< *initialgrid*] [> *ugobject*]

DESCRIPTION

Mklife computes successive generations of an initial grid of binary cells in accordance with the rules of John Conway's game of Life. The successive generations become layers in a 3-dimensional scene suitable for input to UNIGRAFIX. The output can be cubes representing cells, planes representing entire generations, or wires connecting parents with offspring. Cubes are the most informative, but slowest to render. Wires are rendered quickly, but have no good depth cues. Planes are intermediate on both accounts.

Options are:

–ia Specifies that the initial grid will be in ascii format (see below) and will be taken from stdin. This is the default input format.

–ir Specifies that the initial grid is to be randomly generated. See the **–d** and **–s** options (below).

–oc Specifies that the output will be in the format of UNIGRAFIX cube instances. An appropriate cube definition will also be output. This is the default output format.

–op Specifies that the output will be in the form of planes, via UNIGRAFIX faces. No back (bottom) faces will be given.

–ow Specifies that the output will be in the form of UNIGRAFIX wire statements, with wires connecting a cell to its neighbors in the previous generation.

–oa Specifies that the output will be the last generation only, in ascii format (see below). This can be piped back into *mklife*.

–ot Specifies that all generations be output in ascii format for tracing by the user. This is useful in determining what a configuration will become without using a UNIGRAFIX renderer.

–g *generations*
Set the number of generations to compute. **Mklife** will stop before this if the grid becomes empty. Default is 12.

–r *rows*
Set the number of rows in the grid. If the initial grid used with the **–ia** option has a different number of rows, that value is used. Default is 10.

–c *columns*
Set the number of columns in the grid. If the initial grid used with the **–ia** option has a different number of columns, that value is used. Default is 10.

–w Turn on wrap-around at the edges of the grid. The east edge is adjacent to the west edge, and the north edge is adjacent to the south edge. In effect, the grid becomes finite and toroidal. Default is off.

–s *seed*
Use the integer *seed* for creating random initial grid. Ignored if **–ir** option not set.

–d *density*

Use the integer *density* as a percentage density to fill a random initial grid. Default is 24 (percent). Ignored if **–ir** option not set.

–G Turn off the reference grid that is normally included with UNIGRAFIX format output (**–oc**, **–op**, **–ow** options). Default is to output the grid.

–R *rulestring*
Use *rulestring* as the rule for computing successive generations. Format of the rulestring is discussed below. Default is A12D8.

–S *scale*
Use *scale* as the scaling factor for UNIGRAFIX cube instances. Ignored if **–oc** option not used. Value must be between 0 and 1 exclusive. Default is 0.99.

Input file format:

Input files are ascii files containing 0's and 1's (separated by spaces) as values for the cells in the initial grid. Lines in the file correspond to rows in the grid. A small file containing a "glider" might be:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The number of rows and columns in the grid overrides anything specified with the **-r** and **-c** options.

Next Generation Rule format:

The standard game of Life will demonstrate what a rule is, and show how alternate rules are specified to *mklife*. To determine the status of a cell in the next generation, count up the number of cells that are alive amongst its 8 neighbors on the grid. If the current cell is alive, and exactly 2 or 3 of its neighbors are alive, then the cell will be present in the next generation. If the current cell is empty (dead), it will be alive in the next generation only if 3 of its neighbor cells are alive. Otherwise the cell will be empty in the next generation. So we really need two rules, one that tells what to do if the current cell is alive, and one for when the current cell is dead. Representing alive with 1 and dead with 0, we get the following:

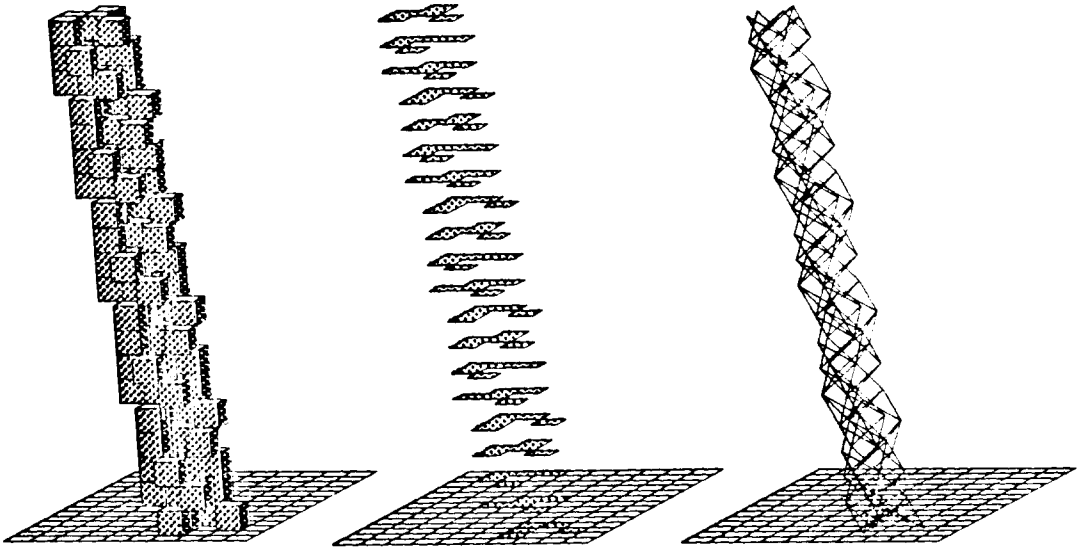
Number of neighbors alive currently	8 7 6 5 4 3 2 1 0
Living cell's status in next generation	0 0 0 0 0 1 1 0 0
Dead cell's status in next generation	0 0 0 0 0 1 0 0 0

If we view the statuses as binary numbers, the Alive rule is 000001100, or 12 decimal. The Dead rule is 000001000, or 8 decimal. Our composite

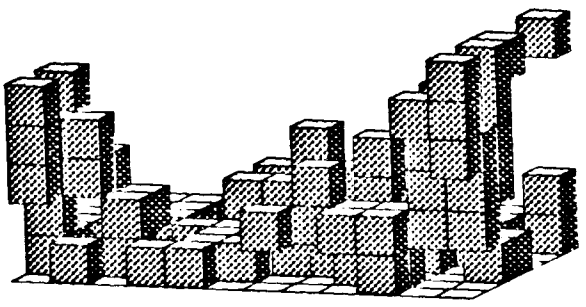
rule is summarized in the string "A12D8". The syntax for a rule is "A<num>D<num>", where <num> is 0-511.

EXAMPLES

cat glider | mklife -g 20 | ugplot -sa -ed 2 1 -5 -sy 3 -dw
cat glider | mklife -g 20 -op | ugplot -sa -ed 2 1 -5 -sy 3 -dw
cat glider | mklife -g 20 -ow | ugplot -sa -ed 2 1 -5 -sy 3 -dw
(the "glider" file from above, in all 3 UNIGRAFIX output styles)



mklife -ir -g 20 -r 12 -c 7 -w -d 50 -op | ugplot -sa -ed 2 1 -5 -sy 3 -dv
(creating a random initial grid)



FILES

~ug/bin/mklife
~ug/src/mklife.c

SEE ALSO

Martin Gardner's Mathematical Games column in Scientific American, early 1970's.
Berlekamp, Conway, Guy, "Winning Ways for Your Mathematical Plays", Academic Press, 1982, Vol. II, Chapter 25. Contains lots of interesting ideas and starting configurations.

DIAGNOSTICS

Echoes the parameters, and may complain and quit if drastically bad input occurs.

BUGS

The **-op** option currently does not handle highly complex configurations. Funny things may happen around the borders of the grid (say, gliders turning into blocks) because interactions that would have occurred out of sight (on an infinite grid) don't occur at all. This effect can be reduced by specifying a larger grid, or by turning on wrap-around.

AUTHOR

Jim Ruppert

NAME

mkrobot – generate hierarchical definitions of a robot arm in UNIGRAFIX format

SYNOPSIS

mkrobot [*options*]

DESCRIPTION

mkrobot is a generator program which reads the predefined parts of a robot arm from the file **~ug/lib/rbparts** in the current directory, takes the values of the various parameters from the command line, and sends a UNIGRAFIX description of the complete arm to the standard output or a named file (-f option). This file contains the parts and the hierarchical definitions of the assembled and moved robot arm. The physical limits imposed by the size and geometry of the arm are checked by *mkrobot* to avoid detached parts or interference.
The available *options* are:

-h <*value*> Default: h=65
This sets the *value* of the tower height, from the ground to the dipping axis of the cylinder. The limits are $65 < h < 105$.

-r <*value*> Default: r=0
This sets the *value* of the arm rotation about a vertical axis.

-d <*value*> Default: d=0
This sets the *value* of the dip angle of the cylinder, measured from a horizontal plane. The limits are $-35 < d < 35$.

-e <*value*> Default: e=0
This sets the *value* of the piston extension, measured from the end of the cylinder to the end of the piston. The limits are $0 < e < 100$.

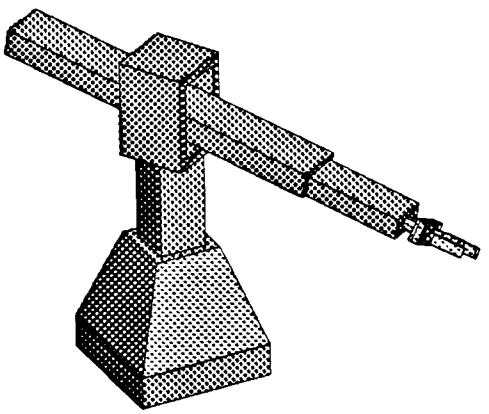
-t <*value*> Default: t=0
This sets the *value* of the twist of the jaw.

-a <*value*> Default: a=0
This sets the *value* of the jaw half-angle, measured as the angle through which a finger rotates from the close position. The limits are $0 < a < 30$.

-f <*output file name*> Default: standard output
This specifies the name of the output file.

EXAMPLE

```
mkrobot -h85 -r60 -d10 -e30 -t30 -a20 -f rbconf
cat ~ug/lib/illum rbconf | ugplot -ed -1 2 -3 -sa -dv
```



FILES

~ug/bin/mkrobot
~ug/lib/rbparts
~ug/src/mkrobot.c

SEE ALSO

ugplot (UG)

DIAGNOSTICS

If a *value* exceeds one of its limits, it is set to this limit and the action is announced. If any unknown option, missing value, or missing option type is detected, a pertinent error message is issued and no output will be generated. All diagnostics are sent to the standard error.

BUGS

To be discovered

AUTHOR

Eric S. Fan

NAME

ug4Dpyr – generate a 4-D pyramid from a 3-D solid

SYNOPSIS

ug4Dpyr [<*options*>] < oldobject > newobject

DESCRIPTION

Ug4Dpyr is a filter which produces a 4-D polytope from a 3-D solid by making a 4-D pyramid. The 3-D solid is put into the w = 0 hyperplane and each face of the solid is connected to a vertex outside of the hyperplane by using a 3-D pyramid.

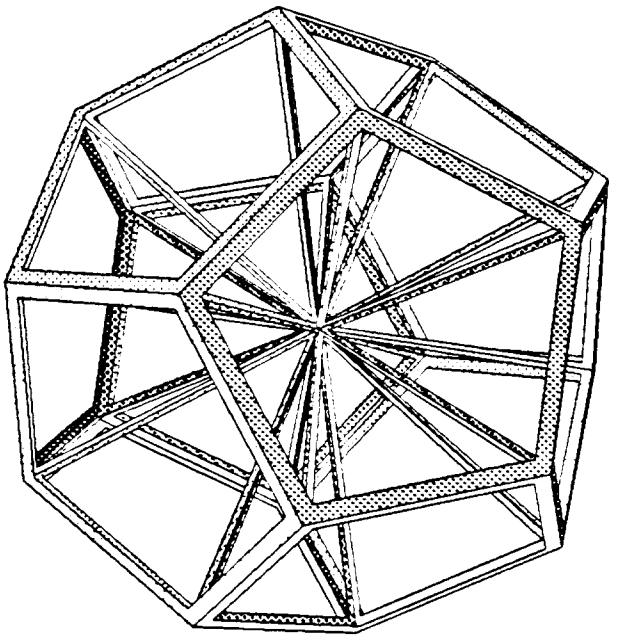
All statements besides the vertex and face statements are passed untouched to the output. The input parameters are:

–v <*x y z w*> *Default: v = 0 0 0 1*

This sets the vertex of the apex of the pyramid. The vertex must lie outside the w = 0 hyperplane (i.e. w component of v must not equal 0).

EXAMPLE

cat ~ug/lib/illum ~ug/lib/dodeca | ug4Dpyr | ug4hole | ug4to3 -ep 0 0 0 -2 | ugplot -sa -ed 1 2 3 -dv



FILES

~ug/bin/ug4Dpyr

SEE ALSO

ug4Dprism (UG), ug4Drev (UG), ug4hole (UG), ugplot (UG)

BUGS

Yet to be reported.

AUTHOR

Dan Filip

NAME

ug4Dprism – generate a 4-D prism (swept solid) from a 3-D solid

SYNOPSIS

ug4Dprism [<*options*>] < oldobject > newobject

DESCRIPTION

Ug4Dprism is a filter which produces a 4-D polytope from a 3-D solid by making a 4-D prism. The 3-D solid is put into the w = 0 hyperplane and swept along the w axis.

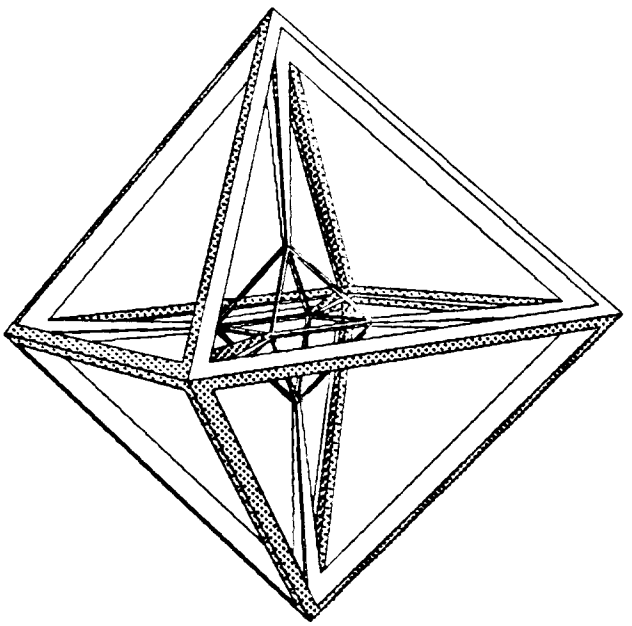
All statements besides the vertex and face statements are passed untouched to the output. The input parameters are:

–d <*real*> *Default: d = 1.0*

This specifies the distance along the w-axis the solid should be swept.

EXAMPLE

cat ~ug/lib/illum ~ug/lib/octa | ug4Dprism | ug4hole | ug4to3 -ep 0 0 0 1.7 | ugplot -sa -ed 3 -2 7 -dv



FILES

~ug/bin/ug4Dprism

SEE ALSO

ug4Dpyr (UG), ug4Drev (UG), ug4hole (UG), ugplot (UG)

BUGS

Yet to be reported.

AUTHOR

Dan Filip

NAME

ug4Drev – generate a 4-D solid of revolution from a 3-D solid

SYNOPSIS

ug4Drev [<*options*>] < oldobject > newobject

DESCRIPTION

Ug4Drev is a filter which produces a 4-D polytope from a 3-D solid by making a 4-D solid of revolution. The 3-D solid is first put in a hyperplane containing the z and w axes and then translated away from the origin. Copies are placed along the z-w plane at intervals of 360/n degrees and corresponding edges and faces are connected.

The input vertex statements must have four components. All statements besides the vertex and face statements are passed untouched to the output. The input parameters are:

–n <*integer*> *Default: n = 4*

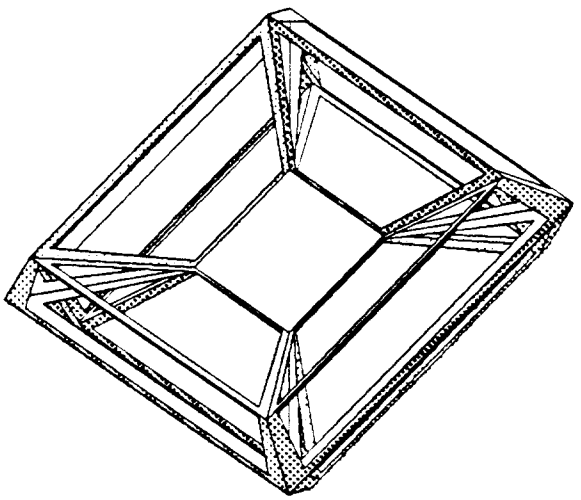
This specifies the number of copies that should be made.

–d <*real*> *Default: d = 4*

This specifies the distance the 3-D object in the 4-D hyperplane should be translated from the origin.

EXAMPLE

cat ~ug/lib/illum ~ug/lib/tetra | ug4Drev | ug4hole | ug4to3 -ep 0 0 2 2 | ugplot -sa -ed 1 2 3 -dv



FILES

~ug/bin/ug4Drev

SEE ALSO

ug4Dpyr (UG), ug4Dprism (UG), ug4hole (UG), ugplot (UG)

BUGS

Yet to be reported.

AUTHOR

Dan Filip

NAME

ugext - Polyhedral Extrusion Filter

SYNOPSIS

ugext [**-fi** fileIn] [**-fi** fileOut] [**-r** radius]
[**-s** scale] [**-g** goalObj] [**-g file** goalObj]
[**-help**] [**-?**] [-]

DESCRIPTION

Ugext is a general-purpose polyhedral extrusion filter. It accepts a previously generated *unigrafix* object as input and "extrudes" the vertices through another goal object, usually a sphere or cube. The resulting object retains the topology of the original object, but with the shape and size of the goal object. For instance, a tesselated cube extruded through as sphere would look like a cube that has had all of its vertices projected onto the surface of a sphere (for a scale factor equal to one).

More interesting shapes make use of the scale option that creates the intermediary objects between the original and goal object. The example above would create a "bloated" cube for scales greater than zero and less than one. A scale of zero implies no change to the original object, and a scale of one implies a full transformation to the goal object.

The most interesting shapes result when the value of the scale parameter is negative or greater than one.

Since extrusions will often generate non-planar faces, it is recommended that the following sequence of steps:

1) Cut all faces into triangles (See *ugtess*), (see BUGS below)

2) Tesselate the object with ugfreq (See *ugfreq*), (see BUGS below)

3) Submit object to *Ugext*.

Ugext accepts the following arguments:

-fi *filename* <*default = stdin*>
Use file *filename* as the *ugext* original object.

-fo *filename* <*default = stdout*>
Use file *filename* as the *ugext* output file.

-r *radius* <*default = 1.0*>
This sets the radius of the goal object. The goal object is always centered around the origin from -1 to 1, but these distances may multiplied by this optional radius.

-s *scale* <*default = 1.0*>
This sets the amount of transformation from the original to the goal object.
scale < 0.0 : strange and interesting objects.
scale = 0.0 : original object.
0.0 > scale > 1.0 : in-between objects.
scale = 1.0 : goal object.
scale > 1.0 : strange and interesting objects.
Scales less than zero and greater than one may create intersecting faces which are mathematically acceptable, but which may confuse the plotting programs. (use *ugisect*)

-g *goalObj* <*default = sphere*>
This option allows you to select a goal object other than a sphere. *Ugext* also allows extrusions through "tetra" and "cube", using its internal formulations of these objects.

-g file *goalfile*
The extrusion goal object may be read in from a previously generated file "goalfile". The

best results are obtained by using polyhedral, regular objects centered about the origin with a maximum unit distance from the origin.

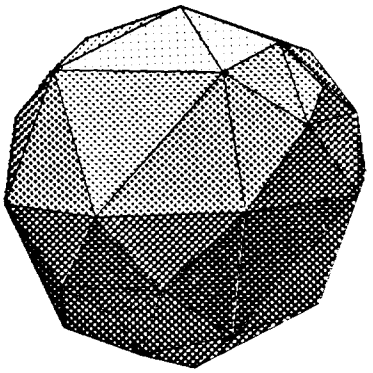
-help Prints out an online usage manual for the program.

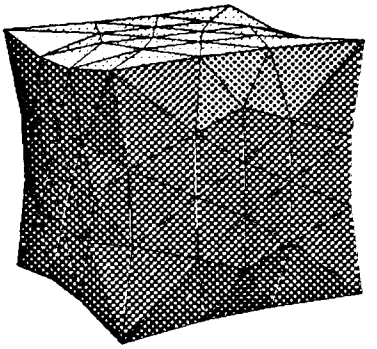
-? Prints out an online usage manual for the program.

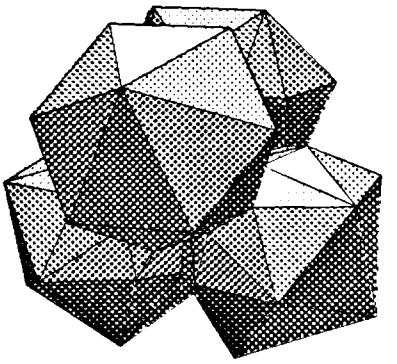
- Prints out an online usage manual for the program.

EXAMPLES

```
cat ~ug/lib/tetra | ugfreq -f4 | ugext > sphereLikeObj
cat ~ug/lib/icosa | ugfreq -f3 | ugext -g cube > cubedIcosahedron
cat ~ug/lib/tetra | ugfreq -f6 | ugext -s 3.0 > electronShell
```







FILES

~ug/bin/ugext ~ug/man/ugext

SEE ALSO

ugfreq (UG), ugplot (UG), ugisect (UG), ugtess (UG)

BUGS

Non-triangular input faces will probably create non-planar faces.

AUTHOR

H.B. Siegel

NAME

Ugfrac - A fractal subdivision filter that works on unigraphics objects composed of triangular faces.

SYNOPSIS

ugfrac [*options, arguments*] < old object > new object

DESCRIPTION

Ugfrac reads in a flat UNIGRAFIX description from standard input, composed of triangular faces. Each face is divided into four subfaces by the addition of 3 new vertices. One corressponding to each edge of the triangular face. The new vertices are placed at a certain displacement from the center of the original edge depending on the following options:

-f <fixed displacement> Default: disp = 0.10
This specifies the fixed displacement that will occur from the center of each edge in the direction of the average of the normals of each adjacent face at that edge. The value can range from -1 to 1 of one-half of the edge length.

-r <maximum random displacement> <seed> Default: maxdisp = 0.10
This specifies the maximum random displacement from the center of each edge. The value can vary from 0 to 1 of one-half the edge length.

-H Default: Show all the faces.
This option leaves out all but the inner triangles at the last subdivision, creating them as double faced triangles.

-n <number of iterations> Default: n = 1.
This specifies the number of times to repeat the fractal subdivision.

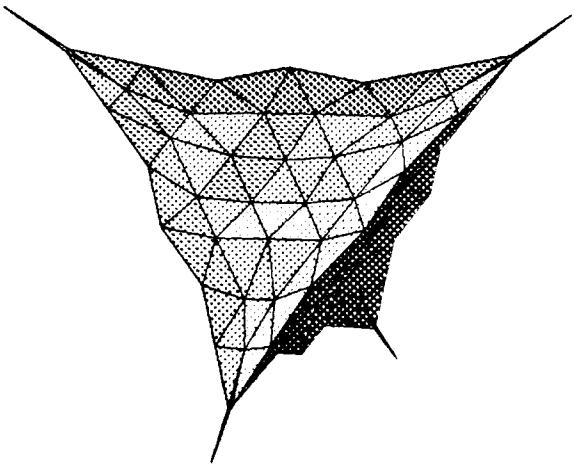
-M If this option is chosen, then all the edges that entirely lie on the X-Z plane will have new points that are also on the X-Z plane, regardless of any of the other options chosen. This option is useful in maintaining a part of an object on the X-Z plane fixed. For example the base of a mountain.

-X -Y -Z Default: All axes are chosen.
Choosing any combination of the axes forces the pertubations to occur in those axes only.

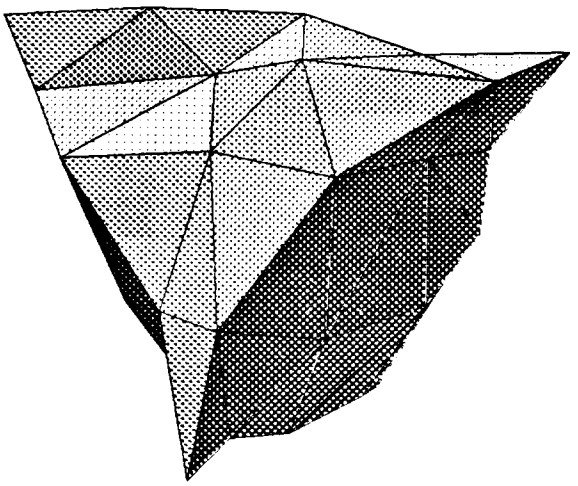
-R If this option is chosen then the displacments from the midpoint will only occur in the radial direction, with the origin as the reference point.

EXAMPLE

cat ~ug/lib/tetra illum | ugfrac -f -0.2 -n 3 | ugplot -ed -5 2 -10 -sa -dw -sy 3



```
cat ~ug/lib/illum ~ug/lib/tetra | ugfrac -r 0.4 999 -n 2 | ugplot -5 2 -10 -sa -dw -sy
```



FILES

~ug/bin/ugfrac

SEE ALSO

ugfreq (UG), ugplot (UG), ugtess (UG)

BUGS

Works only on hierarchically flat objects.

AUTHOR

Lun-Shin Yuen

NAME

ugisect – convert intersecting faces and wires into non-intersecting objects

SYNOPSIS

ugisect [**-I**] [**-D**] [**-U**] [**-A**] [**-vVfF** *eps*] [**-r**] < inputfile > outputfile

DESCRIPTION

Ugisect reads a UNIGRAFIX file and cuts up any intersecting faces to produce a scene description with no intersecting elements. Each existing intersecting element is partitioned into several pieces. The default is to keep all these pieces together in a single statement with multiple contour groups.

Instances of definitions that are intersecting are expanded to the next lower hierarchical level, where all components are again checked for intersection.

Command line flags cause *ugisect* to compute the boundaries of boolean combinations of two solids. To use these options, the input file must consist of exactly two instances at the top level. Faces and wires may occur only in definitions. Each instance must define the boundary of a solid object for the output to be meaningful (*ugisect* does not check that the boundaries are well defined). The instances may contain arbitrary hierarchy.

-I Output the boundary of the intersection of the two solids specified by the two input instances.

-D Output the boundary of the difference of the first specified solid minus the second specified solid.

-U Output the boundary of the union of the two solids specified by the two input instances.

-A <name>
Simultaneously output the intersection, difference and union of the two specified solids. With this option, nothing is placed on standard output; instead, three files "*name*.inter", "*name*.diff" and "*name*.union" are created and the three separate results placed in them. If no name is specified, "inter" is used.

The other flags are:

-v *eps*

-V *eps* Change vertex tolerances. The first form sets the nominal tolerance assigned to vertices as they are read in. The default value is 1e-9. The second form sets the maximum vertex tolerance, which determines how near vertices must lie to be merged into the same vertex. The default is 1e-7.

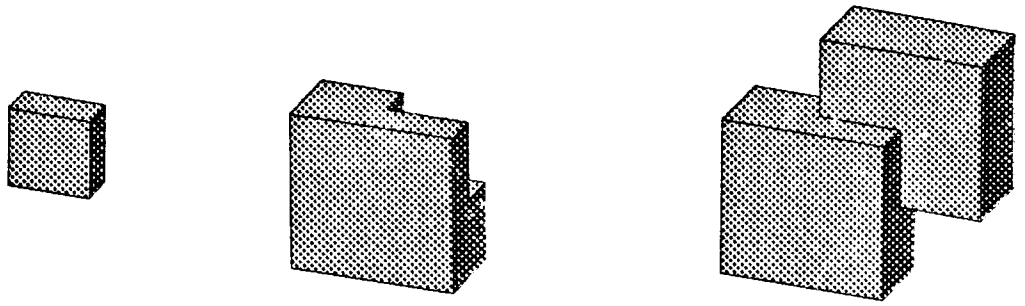
-f *eps*

-F *eps* Change face tolerances. The first form sets the minimum nominal tolerance for input faces; a face may have a computed input tolerance larger than this value if at least one of its vertices lies sufficiently far from its computed plane equation. The default is 1e-6. The second form sets the maximum face tolerance that no face may exceed. It also determines how close together two faces must lie to be considered coplanar. The defualt is 1e-4. If a maximum vertex or face tolerance is exceeded as intersection processing proceeds (as the result of merging), an error message is printed, and the final output may be topologically inconsistent.

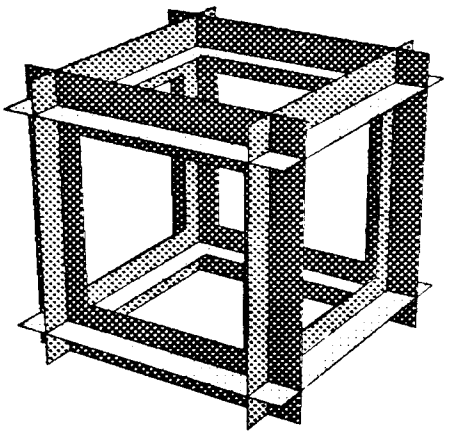
-r Recover vertices. Normally, in the interests of eliminating redundant calculation, computed intersection points are saved even if not immediately incorporated into a cut face. Such a point may arise again in the consideration of other face pairs. However, for large scenes, the memory cost may be prohibitive. The **-r** option prevents intermediate intersection points from being saved, considerably reducing storage requirements.

EXAMPLES

```
ugisect -A slabs < ~ug/lib/slabs
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.inter
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.diff
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.union
```



```
ugshrink -f 1.3 < ~ug/lib/cube | ugshrink -H -f 0.6 | ugisect
   | ugplot -ep -6 5 -10 -ab -sa -dw -sy 2.5 -sx 2.5
```



FILES

~ug/bin/ugisect, ~ug/src/ugc
name.inter, name.diff, name.union

SEE ALSO

ugexpand (UG), ugxform (UG), ugshow (UG), ugplot (UG) ugdisp, (UG)

DIAGNOSTICS

Upon termination *ugisect* will print out statistics on the number of intersecting elements.

BUGS

Does nothing about wires; currently they always pass through uncut.

Will produce a warning and possibly incorrect results if coplanar faces are detected, unless each member of the coplanar pair belongs to a distinct solid boundary.

Instances are expanded whenever their bounding boxes intersect; if it turns out that the instances do not actually intersect the instances are left expanded.

Occasionally a hole is output as the first contour of a face when using boolean operations.

AUTHOR

Mark Segal

NAME

ugman – format and print a unigrafix manual page

SYNOPSIS

ugman [-t] [-r] *title*
ugman -k *keyword*

DESCRIPTION

Ugman is a man facility for unigrafix. Given *title*, ugman searches the directory ~ug/man for a manual text file by that title. When found and no -t option is specified, it checks ~ug/man/nman for an nroff-formatted version of the title file. If none is found, it creates one. Finally, it executes colcrt(1) and more(1) on the formatted file. Colcrt(1) provides a pleasing rendition of italicized and tbl/eqn-formatted text, however, it adds lines for underscores. Use the -r (raw) option to get unmodified nroff output. With -t, the manual text file is typeset and sent to the imagen printer.

The -k option causes the first 20 lines of each formatted file to be searched for *keyword*. The message 'see: *title*' is printed for each occurence.

To install a new man page, put the text file in ~ug/man. After making changes to a text file, be sure to remove the old formatted version so it will be updated by the next user to read it.

FILES

~ug/man/xxx
~ug/man/nman/xxx

DIAGNOSTICS

Complains if requested title does not exist.

BUGS

The command 'ugman -r | lpr' works correctly only if the lp can handle nroff output, in particular, tbl and eqn results. Though wasteful, manual text files are processed through tbl and eqn to cover all bases.

AUTHOR

Gene Ressler

NAME

ugmerge – merge close vertices and edges

SYNOPSIS

ugmerge [– **eps** *tolerance*] < oldobject > newobject

DESCRIPTION

Ugmerge is a filter that merges all vertices within *tolerance* distance of one another. *Tolerance* by default is 1e-6, but can be changed with the **–eps** option.

EXAMPLE

cat ~ug/lib/cube | ugshrink -f 0.9999 | ugmerge > sixSquares

cat ~ug/lib/cube | ugshrink -f 0.9999 | ugmerge -eps 0.001 > sameOldCube

FILES

~ug/new/ugmerge

SEE ALSO

~ug/new/ugcoin [old internal name]

BUGS

It does not merges vertices that lie close to an edge by merging such vertices into the edge itself.

AUTHOR

Paul Wensley created the old ugcoin.

NAME

ugpipe – generalized ball and cylinder filter for UniQuadrix

SYNOPSIS

ugpipe [*options*] **< ugobject > ugobject**

DESCRIPTION

Ugpipe is a filter which produces ball and cylinder descriptions for the *UniQuadrix* graphics rendering system. *Ugpipe* accepts standard *UNIGRAFIX* scene descriptions. It 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*.

Options:

–b		Subtract balls from final scene output (render with cylinders only, truncated where they would have intersected the balls, had they been included).
–c		Subtract cylinders from final scene output (render with balls only, truncated where they would have intersected the cylinders).
–m		Generate a mitred joint at each vertex. Overrides all ball specifications.
–t		Generate transparent truncation planes and mirror planes. Truncation planes define the intersection between balls and cylinders; mirror planes define the intersection between cylinders (or cones) incident on the same vertex.
–rb	*radius*	Specify global radius for all balls that do not have a local radius specification (see below).
–rc	*radius*	Specify global radius for all cylinders that do not have a local radius specification (see below).

Input description format

In addition to standard *UNIGRAFIX* description files (including definitions, instances, and arrays), *Ugpipe* accepts various extensions to allow maximal usage of the capabilities of *UniQuadrix*. For example, the presence of *UniQuadrix* objects will not confuse the *Ugpipe* parser, so the user can imbed his own *UniQuadrix* objects in the ball and cylinder output produced by *Ugpipe*.

The radius of each ball and the ends of each cylinder can be specified locally in the description file. Such local specifications override the global radii specifications given on the command line. Note that differing radii specifications at the ends of the same cylinder will result in a tapered or cone-shaped "cylinder", which is perfectly legal.

Although cylinders are only allowed to intersect balls or other cylinders at vertices, balls are allowed to intersect other balls. This is handy for producing ball-models of molecules, for example.

Specification of local radii

Ugpipe accepts an optional fourth argument for each vertex which allows the user to specify a radius for the ball surrounding the vertex. This radius will override any global ball radius specification. The format is:

v [*id*] *x y z* [*radius*]

Radii can be specified for the cylinders generated from face edges in a similar fashion:

f [*id*] (*id id id* { *id* }) [*radius*] { (*more ids...*) [*radius*] } [*radius*]

Although it may be of limited utility, a local radius can be specified for each set of contour edges individually. The final radius specification is applied to all contour edges that don't have an individual radius specification. If none of these local radii specifications are present, the global cylinder radius value is used for all cylinders associated with the edges of this face.

The specification of cylinder radii along a wire is more complicated. The global cylinder radius can be overridden by specifying one radius for the entire wire, which can then be overridden by specifying a radius for certain segments within the wire, which can be overridden by radius specifications at particular joints. (Don't worry, we'll do an example in a minute...)

The general format of the wire statement is:

w [*id*] (*id* [*radius*] *id* [*radius*] { *id* [*radius*] }) [*radius*]
{ (*more wire segments...*) [*radius*] } [*radius*]

Here's an example of a complicated wire statement:

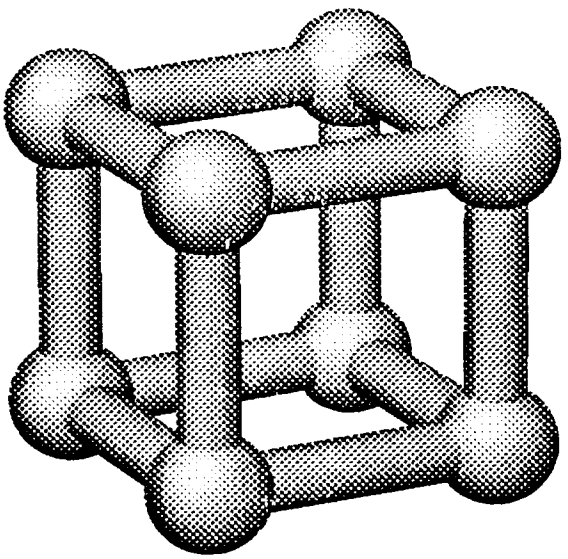
w (a1 a2 a3 1 a4) 2 (b1 b2 b3) (c1 c2 3 c3) 4 5;

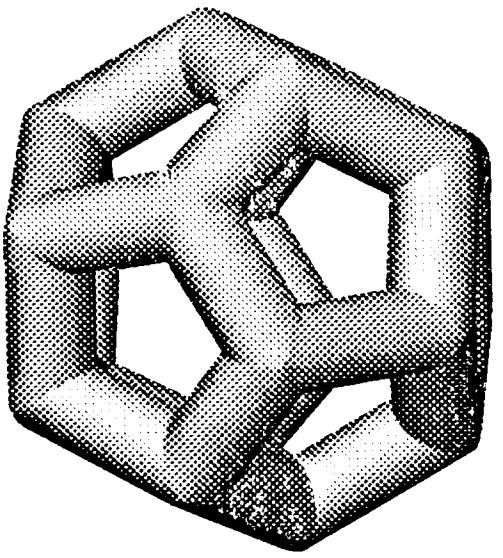
The radius of 5 specified at the end of this statement indicates that all cylinders in this wire will have radius five (overriding any global radius specification) unless specified otherwise by other local definitions. The first wire segment (the one going through vertices a1, a2, a3, and a4) actually overrides this specification immediately, declaring that the radius of all cylinders in it will be 2. Inside this segment, however, we have specified a radius of 1 at vertex a3. This means that we will get a cone-shaped cylinder whose starting radius is 2 at vertex a2, tapering to a radius of 1 at a3. The cylinder from a3 to a4 begins with a radius of 1 and ends with a radius of 2.

In the second segment, since there are no overriding specifications, the cylinders will have radius 5. In the third segment, all cylinders have radius 4 except for the cones incident on vertex c2, which will have a radius of 3 at that point.

EXAMPLES

```
cat ~ug/lib/cube viewF | ugpipe -rb 0.4 -rc 0.2 | uq
cat ~ug/lib/dodeca viewF | ugpipe -rc 0.2 | uq
```





FILES

~ug/bin/ugpipe

BUGS

Cylinders must not intersect other cylinders or balls except at vertices.

Uq currently complains profusely (thousands of warnings about coincident vertices!) when fed *Ugpipe* output. This is *Ugpipe's* fault for generating unnecessary mirror planes. These warnings are bothersome, but they will not adversely impact the rendering.

Ugpipe does not understand include or color statements. Comments are removed.

Cones don't intersect in a plane, so don't expect their edges to match up exactly. Cones of differing radii meeting at a vertex can generate very large or sometimes infinite quadrics if a mitred joint is specified.

Although ball/ball intersections are calculated and displayed correctly, no such calculations are made for balls belonging to different instances or array definitions.

AUTHOR

Philip Flanner, Don Marsh

NAME

ugtess – tessellate faces into convex polygons or triangles

SYNOPSIS

ugtess [-t] < oldobject > newobject

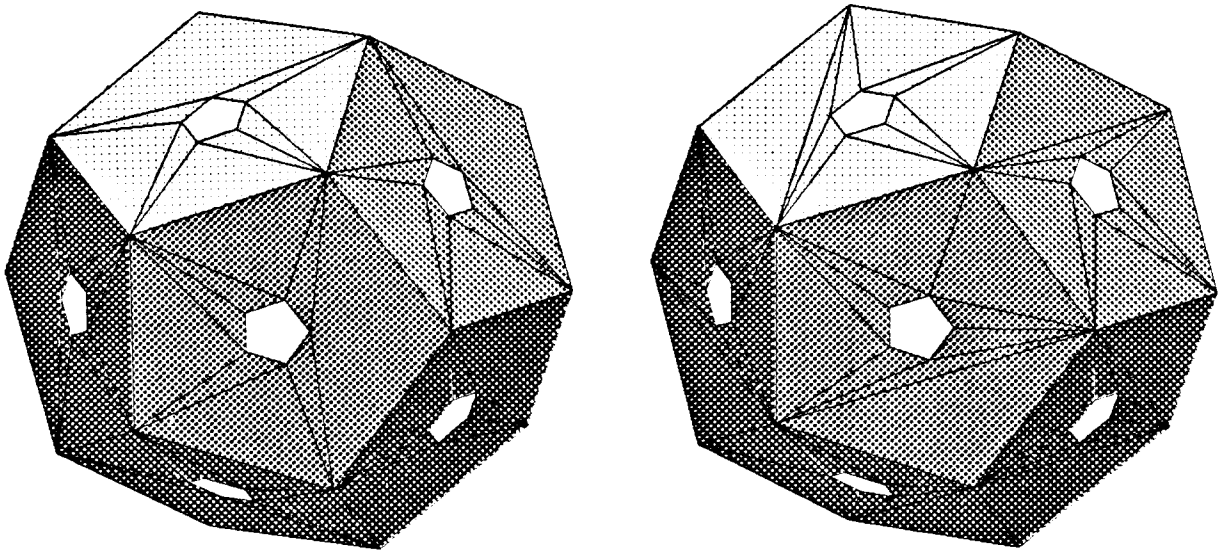
DESCRIPTION

is a filter that *tessellates* the faces of an arbitrary unigrafix object into convex polygons without creating any new vertices.

–t The faces are *triangulated*.

EXAMPLE

cat $(LIB)/dodeca illumF | ugshrink -H -f 0.2 | ugtess | ugplot -ed -7 2 -5 -sa -dw -sy 3 -sx 3
cat $(LIB)/dodeca illumF | ugshrink -H -f 0.2 | ugtess -t | ugplot -ed -7 2 -5 -sa -dw -sy 3 -sx 3



FILES

~ug/bin/ugtess

SEE ALSO

ugext (UG), ugfrac (UG), ugfreq (UG)

BUGS

Yet to be found.

AUTHORS

Ziv Gigus, Lucia Longhi

NAME

ugtight - tighten knot using simulated annealing algorithm

SYNOPSIS

ugtight [-i]

DESCRIPTION

Ugtight reads from a file "ax" in the current directory that contains a single loop of ax-joints specified in the format for axfiles for *mkworm* and tightens the loop by minimizing total axlength using the simulated annealing algorithm. The program outputs an axfile "ax0" that contains joints of the tightened knot and is readable by *mkworm* .

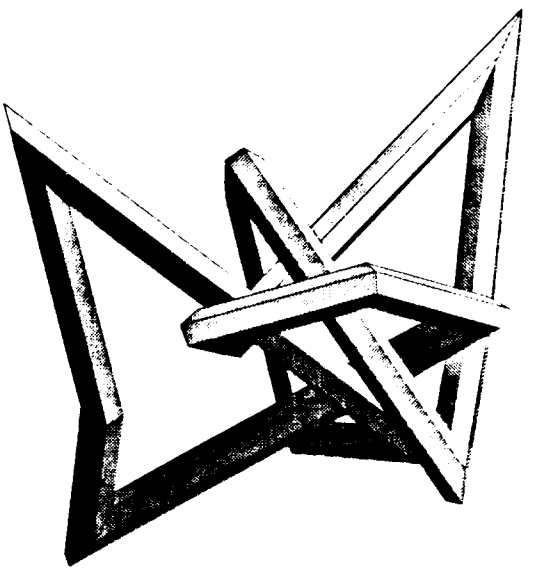
The file "ax" must begin with LOOP or L and end with RETURN or R. Lines specifying joints must begin with "j" and each such line must contain 4 real numeric fields. The 4th field in the first line is then used globally as the ax radius. The file cannot contain any comment lines. The program writes status information about the simulated annealing algorithm and the initial and final states of the knot into a log file "log".

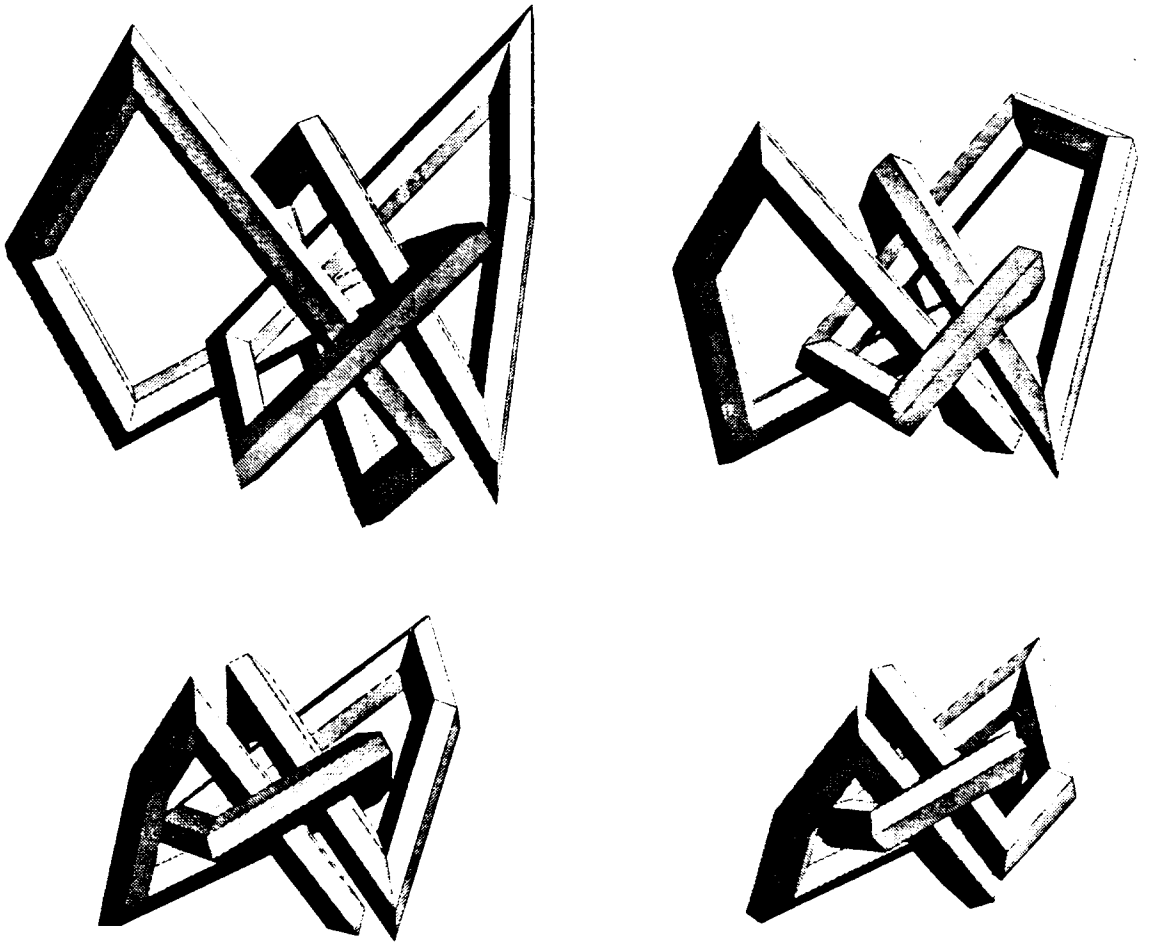
The -i option puts the user in interactive mode. This option is intended for a user with some knowledge of the simulated annealing algorithm and provides her/him with the ability to fine-tune the algorithm, print out intermediate results, and read from a file other than "ax". The program will print out a menu of options that include changing the temperature manually, reinitializing the random number generator, changing the number of moves per temperature, keeping some of the joints fixed, and having every other move be in the direction of an ax-segment (the default) or all moves random.

The program takes a long time to run, especially on larger knots, so it might be best to run it in the background or overnight using *at*.

EXAMPLE

vi ax; mkworm -n 6 -r 1; mv worm worm1; ugplot -sa < worm1;
ugtight; cp ax0 ax; mkworm -n 6 -r 1; mv worm worm2; ugplot -sa < worm2;
The figures show several stages in the tightening sequence.





FILES

~ug/new/ugtight ~ug/src/ugtight, ax, ax0, log

SEE ALSO

mkworm (UG), ugplot (UG)

BUGS

Sometimes does not find exact global minimum--i.e. knot is tight but not as tight as it could possibly be. Only takes one loop as input. Needs more error checking. Holding one or more joints fixed seems to adversely affect the performance of the algorithm.

AUTHOR

Cecilia Aragon

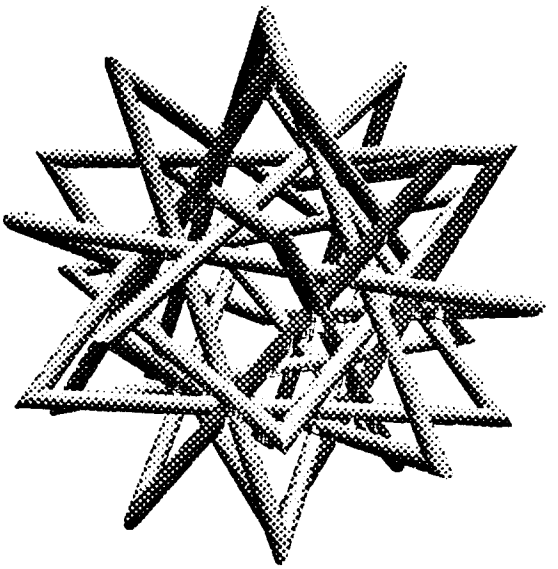
3. END-OF-COURSE ART-SHOW

Using a combination of the new programs, some crafty hand-editing of the UNIGRAFIX description at some stage of the transformation process, as well as clever arrangement of objects generated with different runs, the students managed to create some rather artistic displays. Somewhat more elaborate versions, typical ranging two to three feet in size, were exhibited at the second UNIGRAFIX Art-show in the CS Lounge on May 10, 1985 (see poster below). Some simplified versions more suitable of the limited size of these pages are recreated and displayed on the following pages. In some cases, a short explanatory paragraph gives some information about the concept and the steps that were used to produce the artwork. In other cases, the artist only provided some tantalizing remarks to stimulate the imagination of the reader. We hope that the presentation of these examples will entice the reader to try on his own to make creative use of the Berkeley UNIGRAFIX tools.

UNIGRAFIX

ART-SHOW & RECEPTION

Friday, May 10, 3-6pm, CS Lounge



The class CS292A 'Creative Geometric Modeling'
and the UNIGRAFIX group
invite students, staff and faculty
to stop by the CS Lounge that afternoon
to admire the results of four months of course work,
and to celebrate the end of the Spring Term 1985.

(As with every decent artshow,
there will also be some hardware and software
that cater to a different kind of taste.)

Capturing Geometry

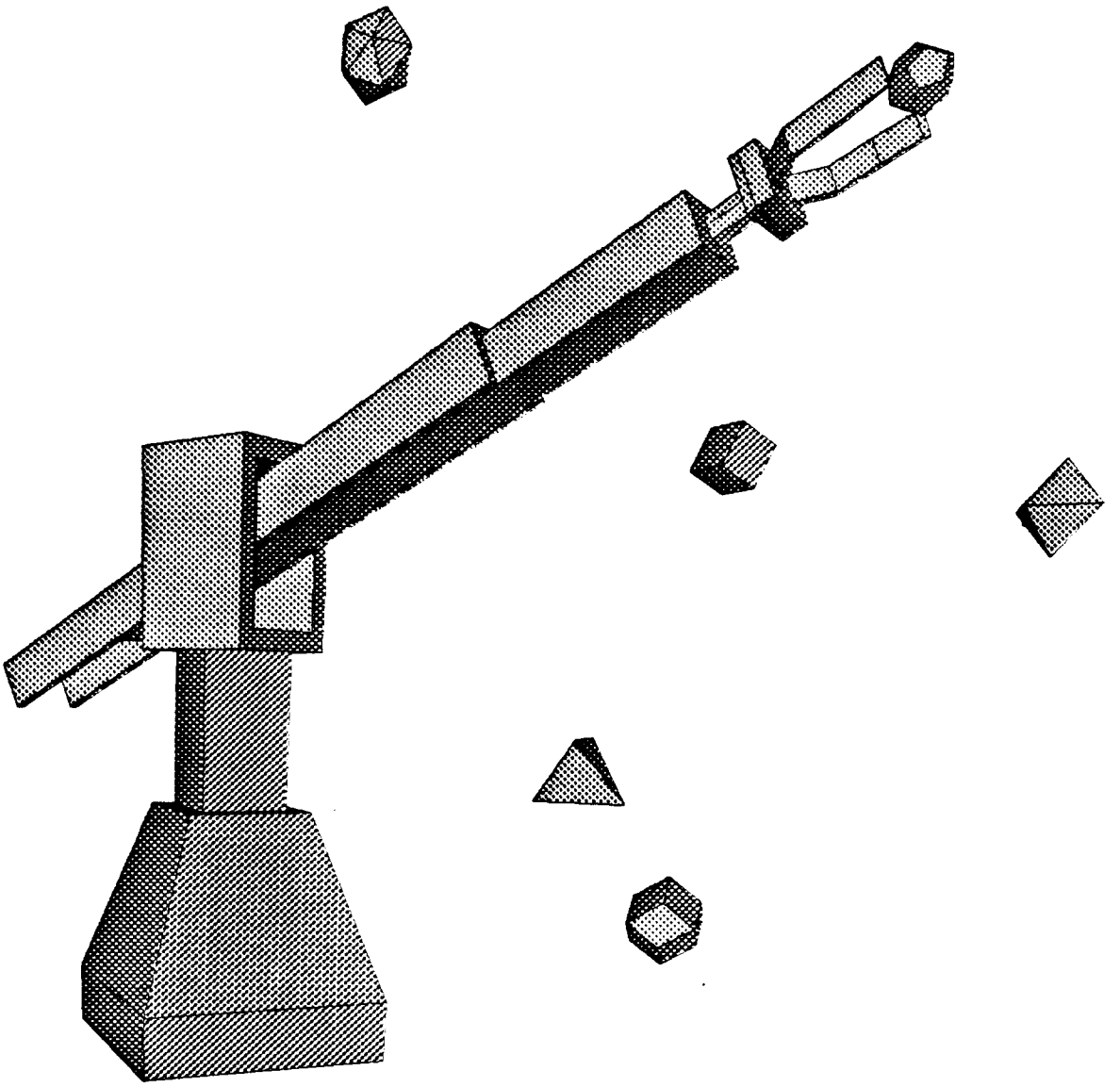
Eric S. Fan

Division of Structural Engineering and Structural Mechanics
Department of Civil Engineering
University of California, Berkeley, CA 94720

ABSTRACT

The robot arm is reaching out to the wonderful world of geometry to capture its share.

The robot arm has six degrees of freedom: for the tower - vertical motion, for the arm - horizontal sweep, dip or raise, and extension, for the jaw - twist and opening. The simulation of movement is accomplished by adjusting the transformations within the hierarchical framework of the arm definition. The floating objects are icosahedron, dodecahedron, cube, octahedron, tetrahedron, and rhomboedric dodecadron.



CAPTURING GEOMETRY

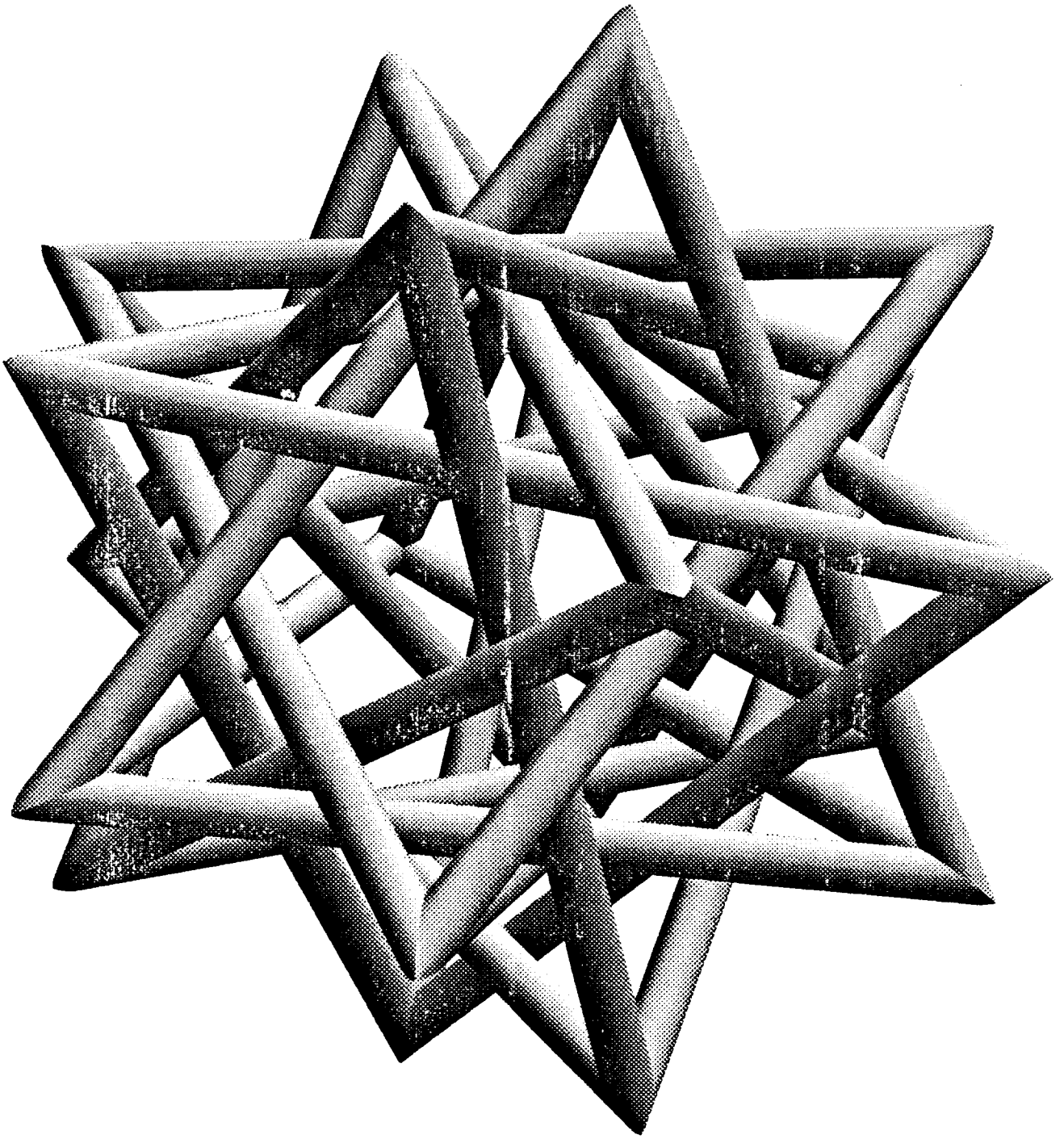
Eric Fan

" TETRA - TANGLE "

Carlo H. Séquin
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Five tetrahedra can intersect each other in a symmetrical manner so that their corner vertices coincide with the vertices of an icosahedron. Here the edges of the tetrahedra are replaced with properly mitred tubes of a suitable diameter so that the tubes just touch and give rigid stability to the whole construction.

The position of the tetrahedra was specified manually; *Ugpipe* produced the specifications for the mitred cylindrical sections; and *UniQuadrix* rendered the final image.



TETRA - TANGLE

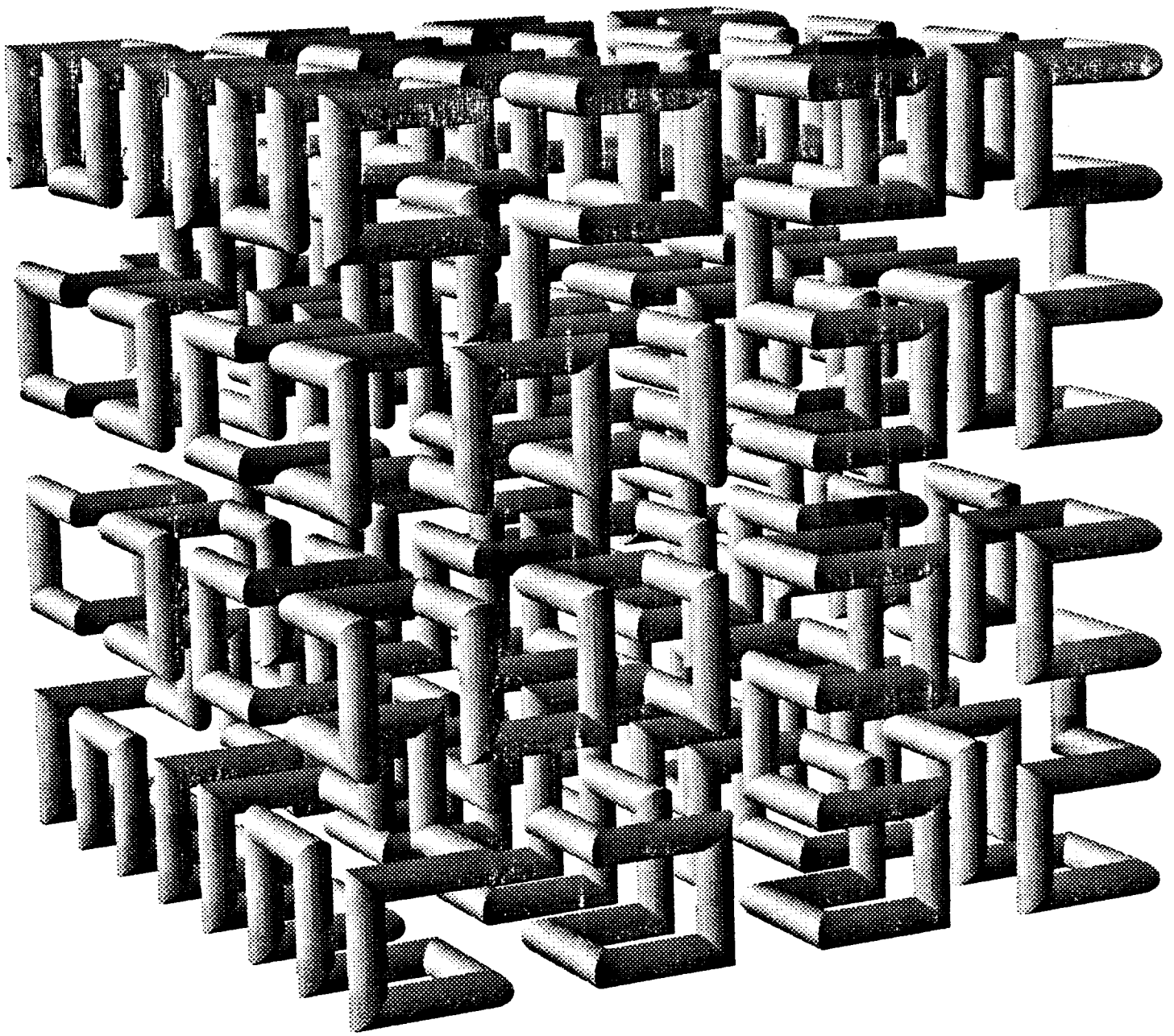
Carlo H. Séquin

" HILBERT PIPE "

Carlo H. Séquin
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

This was inspired by Hilbert's famous curve in two dimensions. The challenge was to get a similar space-filling curve in three dimensions. Additional design constraints were to make the curve closed, to make it bilaterally symmetrical with respect to more than one plane, to have no two subsequent pipe segments collinear and not more than three subsequent segments coplanar. - Well, a solution was found ...

The renderings of the second and third generation curves were produced with the help of *mkworm*, *ugpipe*, and *UniQuadrix*.



HILBERT PIPE

Carlo H. Séquin

"Ugpipe Object"

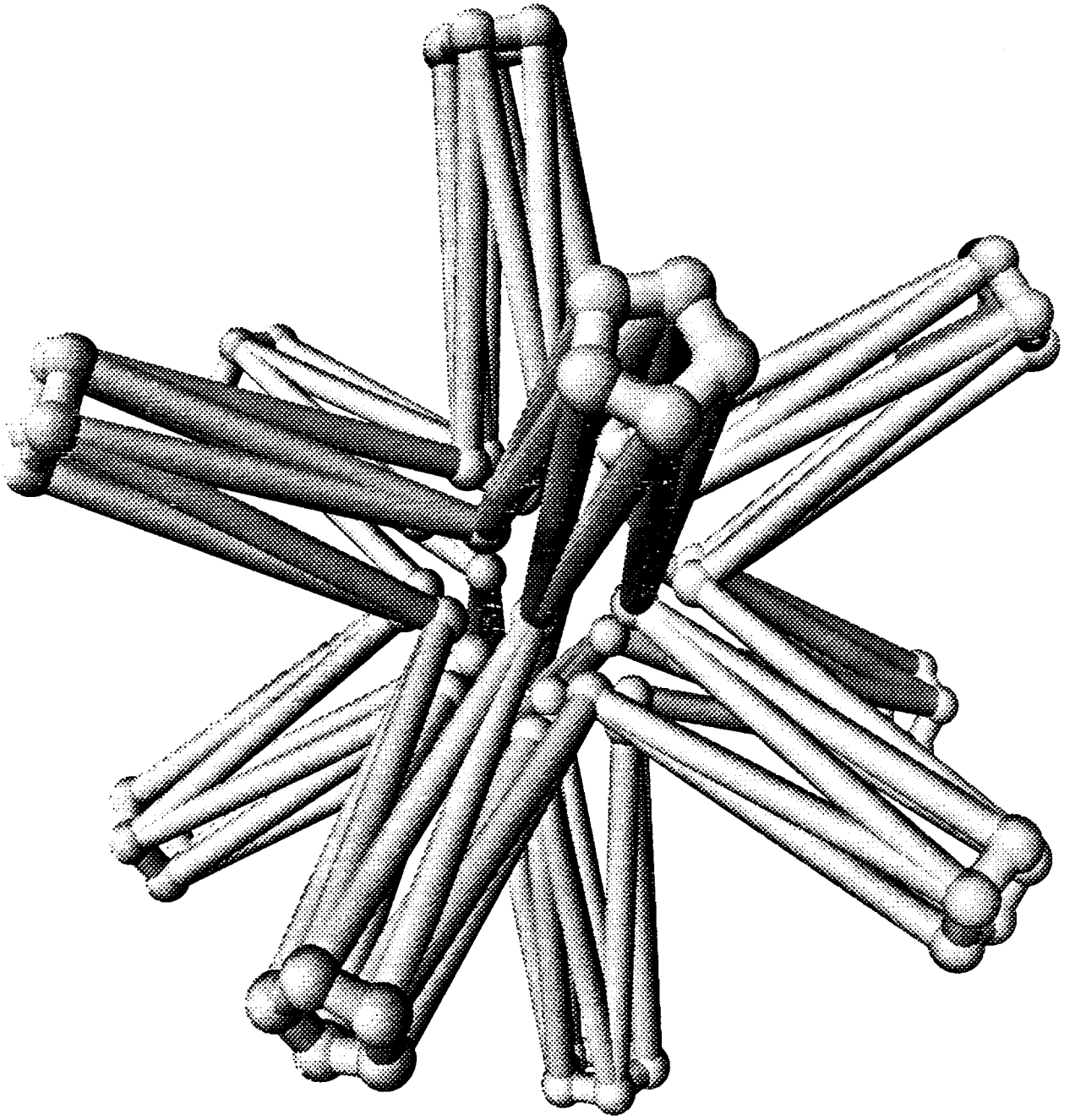
Don Marsh and Philip Flanner

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

ABSTRACT

Using standard Unigrafix filters, we began with a dodecahedron (that's a regular object whose faces are pentagons), and attached a 5-sided pyramid to each face. We then truncated the top part of each pyramid, and ran the result through our "ugpipe" filter which replaces edges with cylinders and vertices with balls. The entire command is:

ugstar <~ug/lib/dodeca | ugtrunc | ugpipe -rc 0.1 -rb 0.15 | uq



UGPIPE OBJECT

Don Marsh and Philip Flanner

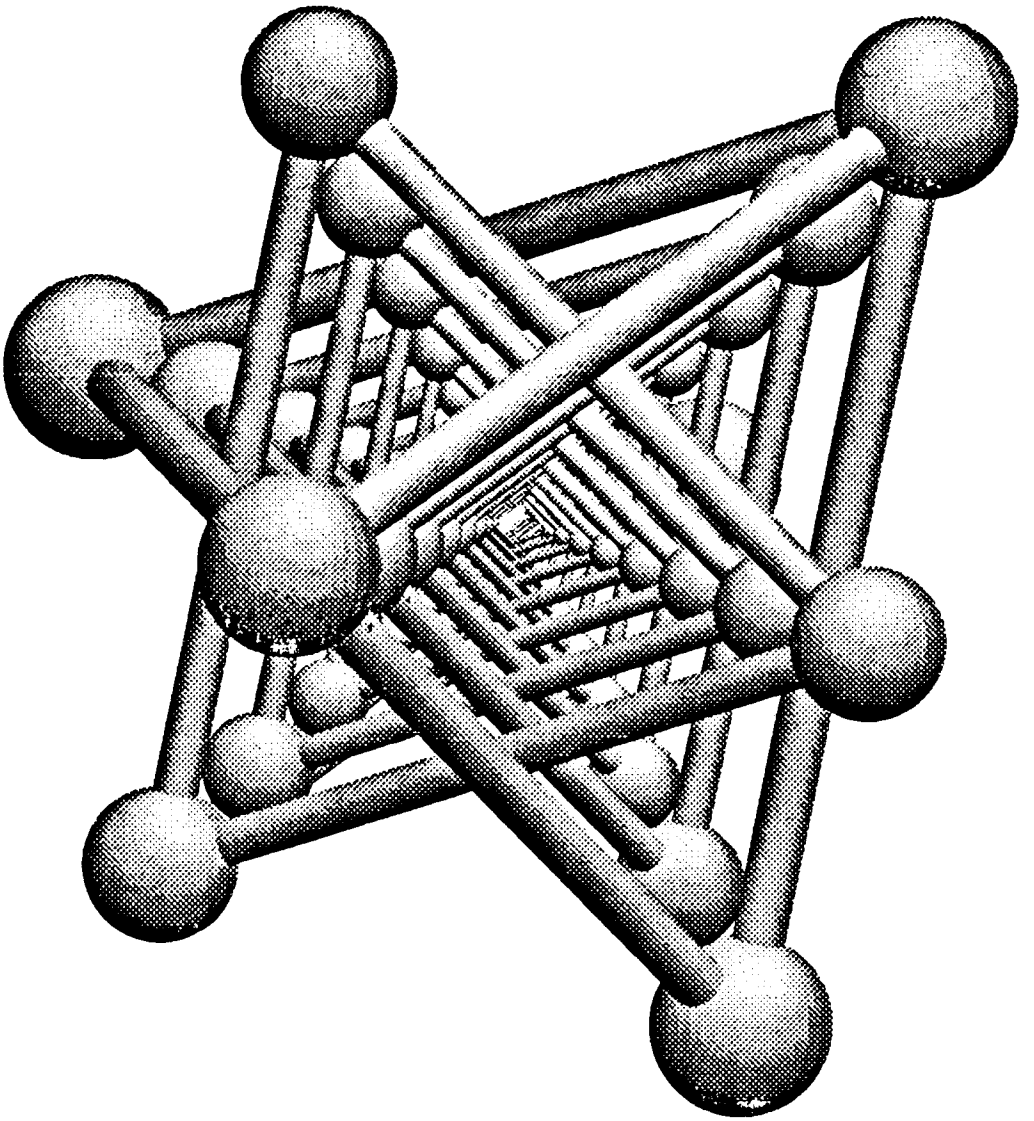
"Dizzy Tetrahedrons"

Don Marsh and Philip Flanner

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

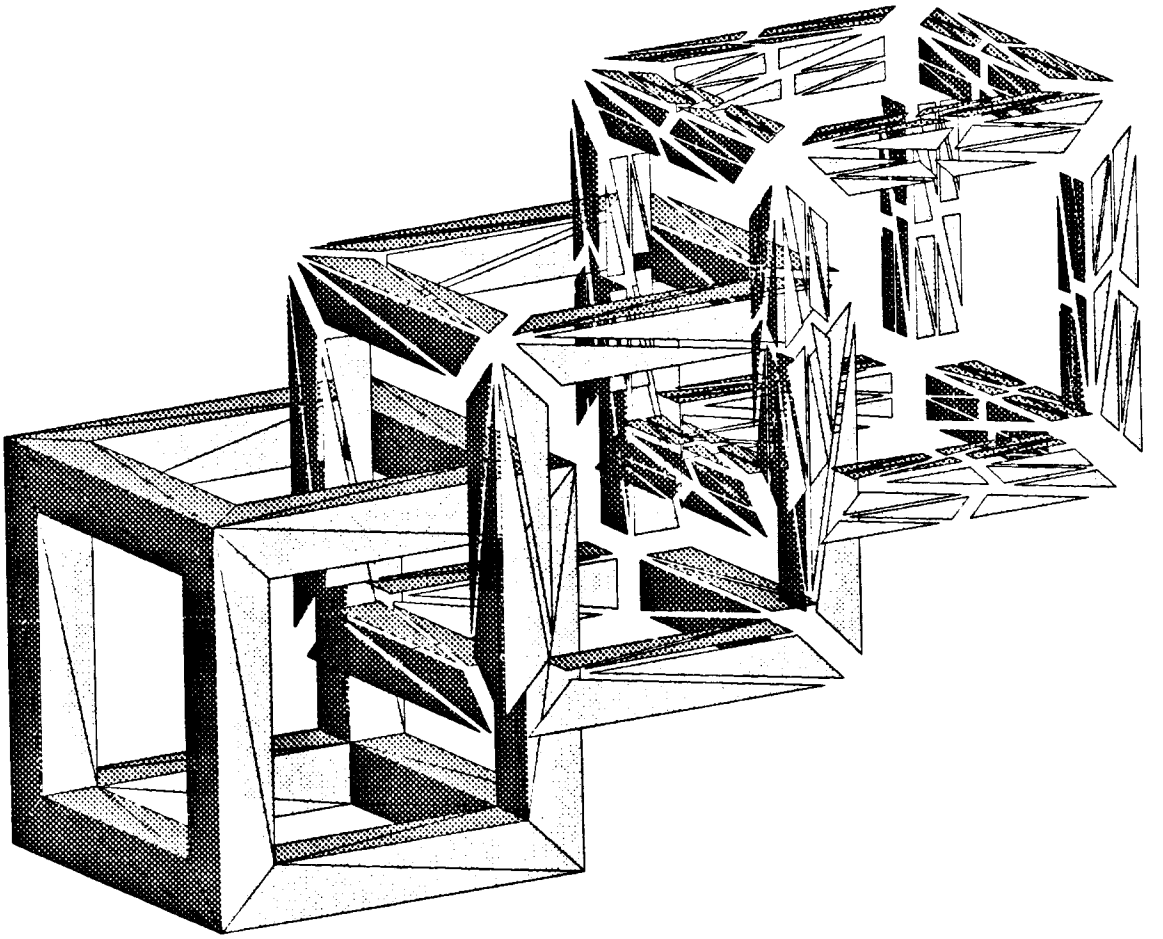
ABSTRACT

These nested tetrahedrons were generated from a single definition of a tetrahedron using "ugpipe." This definition was then incrementally scaled to produce smaller instances inside the original. Using a negative scaling factor, each tetrahedron flips orientation with respect to its immediate parent. This was a particularly fun image to design, because the equations that describe how to generate a smaller tetrahedron that nests exactly inside the last one turn out to be surprisingly simple and beautiful.



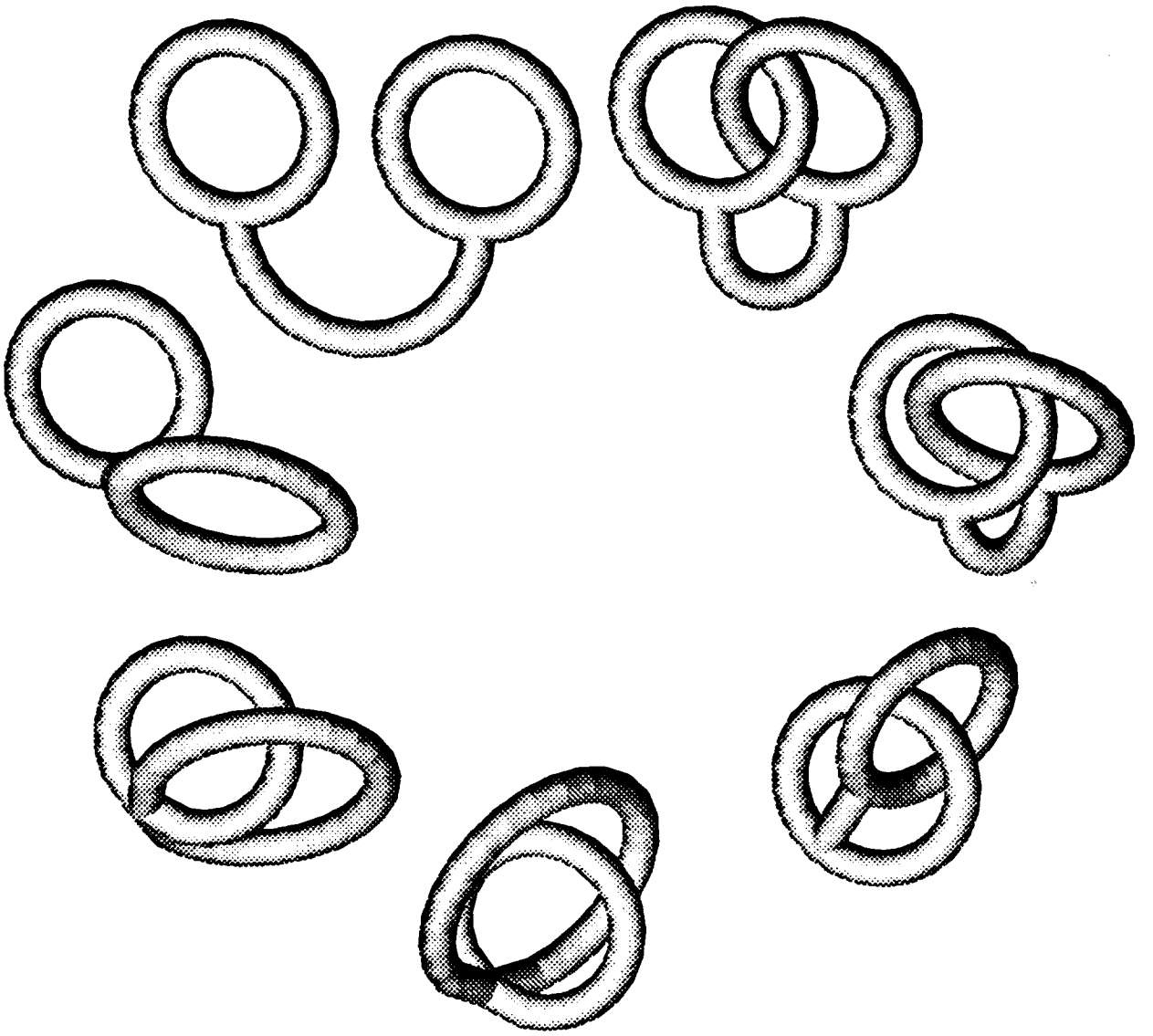
DIZZY TETRAHEDRONS

Don Marsh and Philip Flanner



DISINTEGRATING CUBE

Lucia Longhi



MATHEMATICAL DONUTS: TOPOLOGY WITHOUT WORDS

Jim Ruppert

"Mr. Snowman"

Don Marsh and Philip Flanner

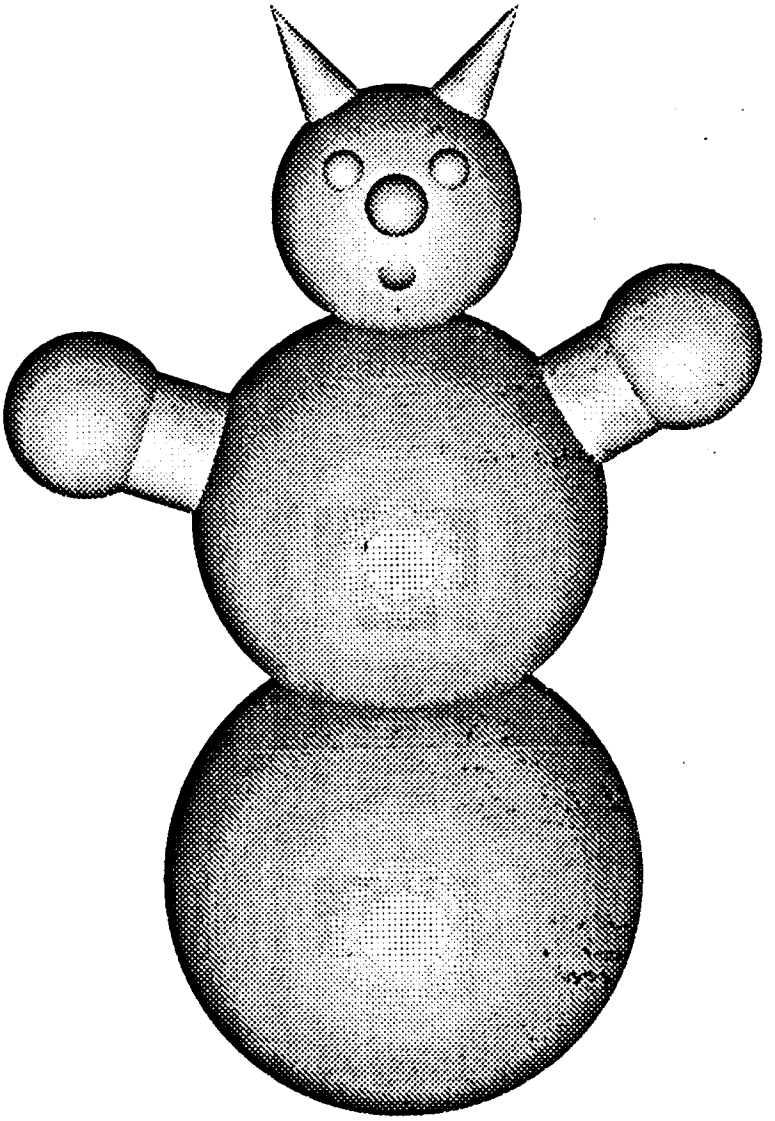
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

Before the advent of Uniquadrix and Ugpipe, Mr. Snowman remained an elusive and somewhat mythical creature. Although reports of occasional sightings continued, many doubted his existence. But with these new tools at our disposal, we are able to offer this conclusive proof that Mr. Snowman is not a figment of the demented imagination.

Mr. Snowman was crafted using intersecting balls fed through our "ugpipe" filter. The ears and arms were created by threading wires through Mr. Snowman's head and body. Below is the file which created him.

```
v lowerbody 0 -3.6 0 2.3;  v upperbody 0 0 0 2;
v head 0 3 0 1.2;  v nose 0 3.3 -1.1 .3;
v lfeye -.5 3.6 -.9 .2;  v rteye .5 3.6 -.9 .2;
v mouth 0 2.7 -0.9 0.3;

v leftear -1.2 5 .2 0;  v rightear 1.2 5 .2 0;
w (leftear head .6 rightear) 0;

v leftarm -3 1 0 .8;  v rightarm 2.7 1.6 0 .8;
w (leftarm upperbody rightarm) .6;

view vrp 0 0 0 dop 0 .5 -2;
```



MR. SNOWMAN

Don Marsh and Philip Flanner

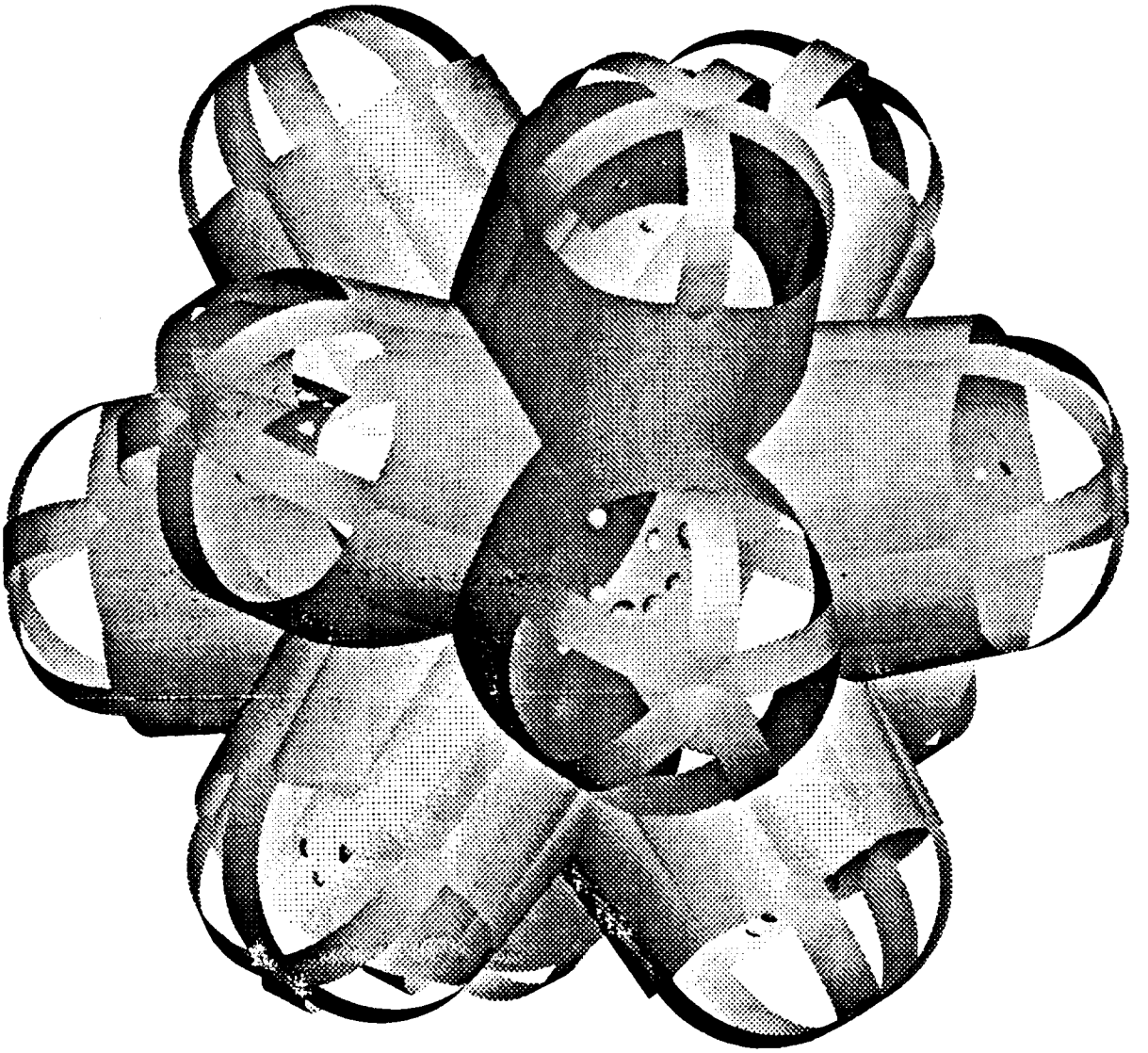
"High-tech Potato-chip Basket"

Don Marsh and Philip Flanner

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

ABSTRACT

The High-tech Potato-chip Basket could easily be the invention of the decade. Using a sophisticated new "anti-magnetic" field, the potato chips (and potato chip fragments) are gently suspended in mid-air, keeping them fresh and beyond the reach of scavenging insects. A harmless electric field will also repel the hands of young children attempting to snack between meals.



HIGH-TECH POTATO-CHIP BASKET

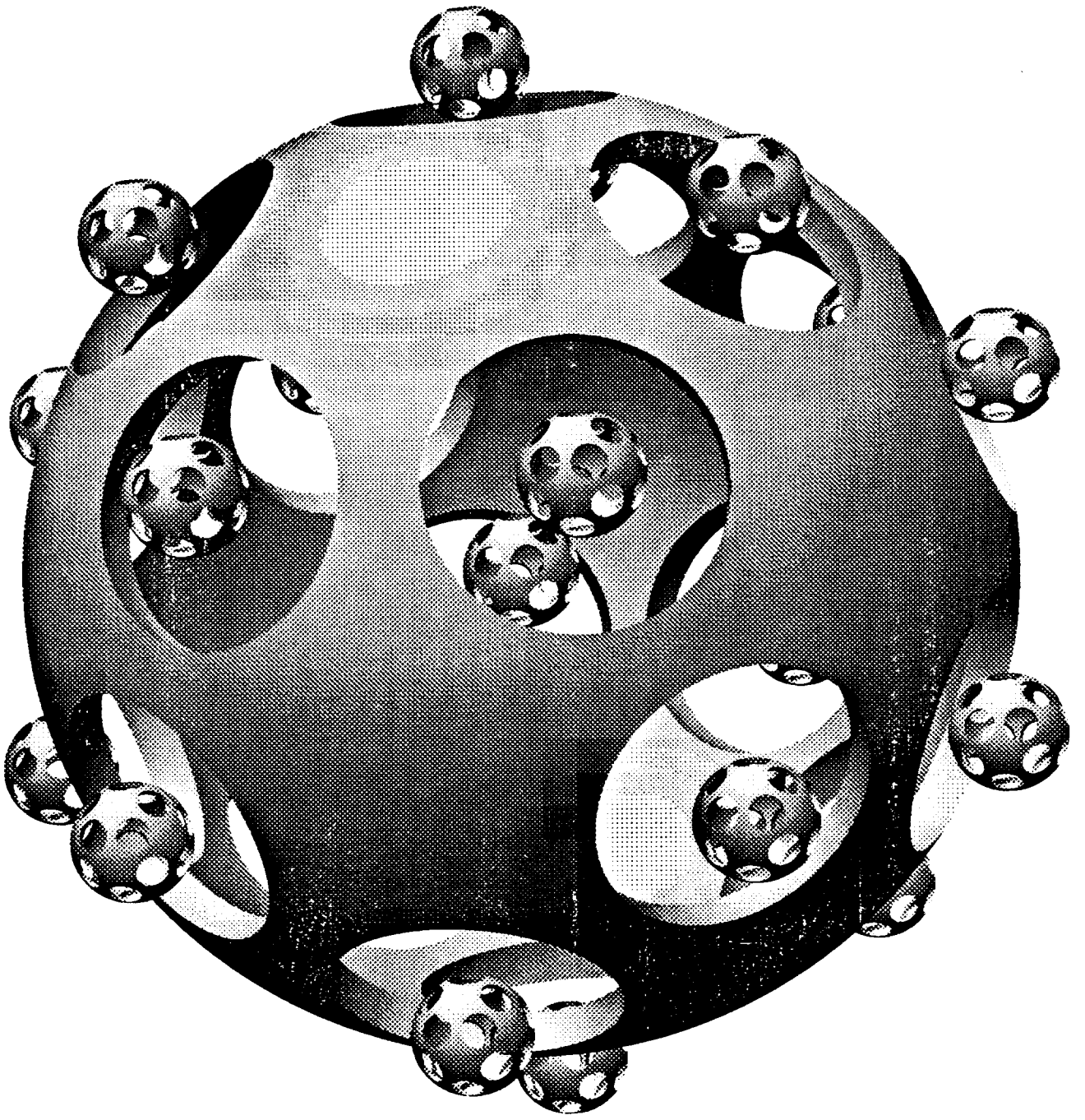
Don Marsh and Philip Flanner

" FREE - FLOATERS "

Carlo H. Séquin
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

The twenty corners of a dodecahedron are the centers of spheres that cut out holes from the large sphere. The resulting object is then reduced in size and replicated in the center and in each one of the cut-out holes.

The wire frame of the dodecahedron and the specification of the desired sizes of the spheres were fed to the new program *ugpipe*, which produced the the proper specification of the intersecting planes between the spheres. These specifications were then converted into a smooth-shaded plot by *UniQuadrix*.



FREE - FLOATERS

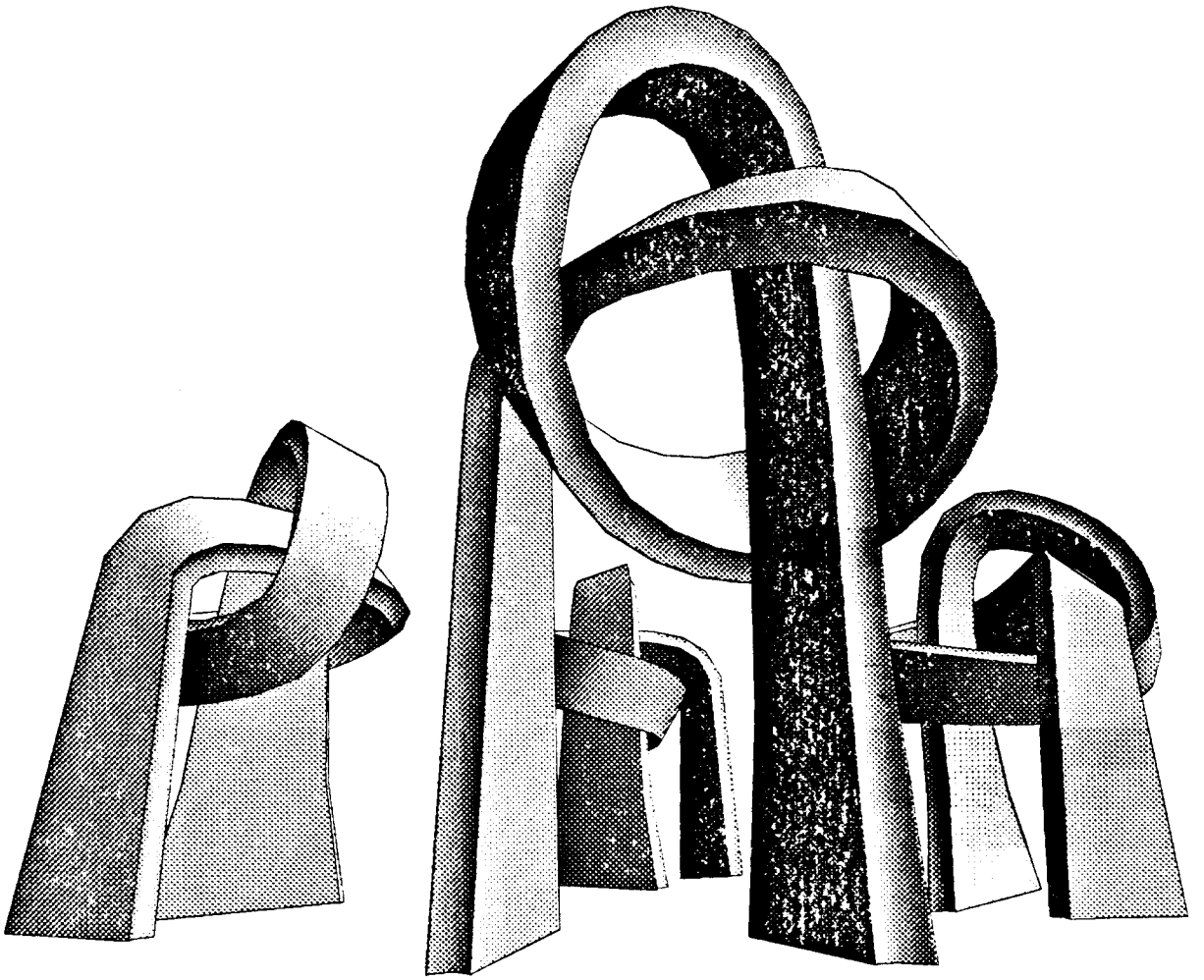
Carlo H. Séquin

" STONEHENGE - 2000 "

Carlo H. Séquin
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

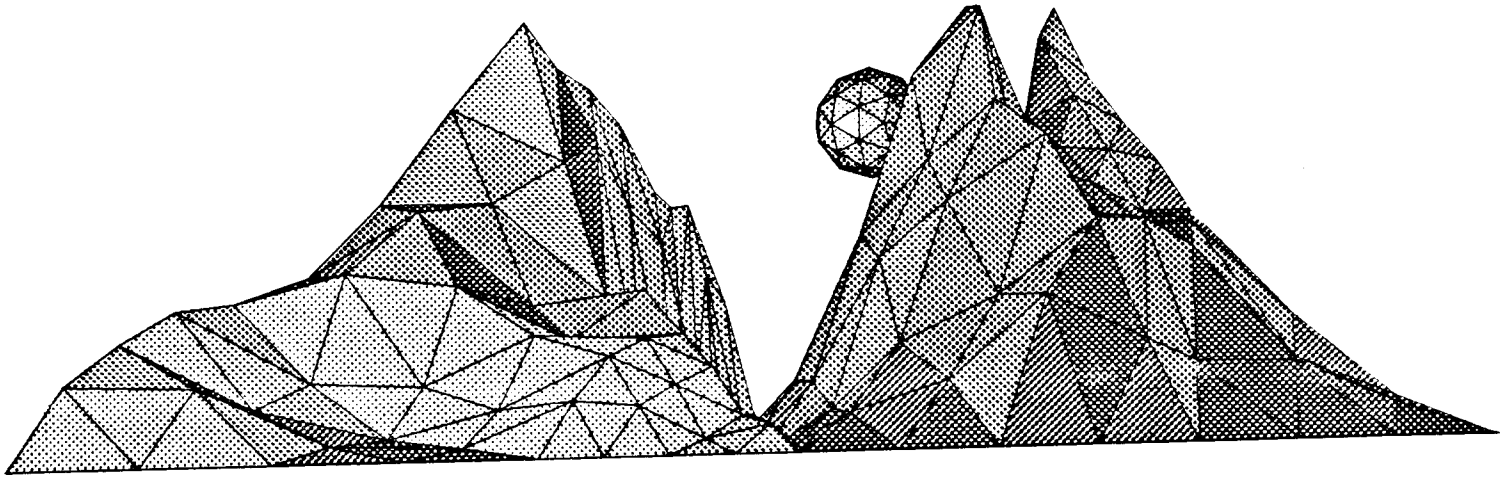
This mysterious group of gigantic knotted arches, standing in a special, not yet fully understood configuration on an imaginary plane in Northern California, is believed to have originated from a tribe known as the *Unigrafists*.

The currently accepted theory assumes that these artifacts are the cooperative result of several computer tools. First a *spline* program was used to create the smooth curve of the knot axis from only a few points. An enhanced version of *mkworm* was then used to sweep the cross section of the beam along this axis; this program can also scale and rotate the swept cross section independently with three degrees of freedom. The resulting object was then rendered with *ugdisp*, a new renderer with smooth Gouraud shading. There are no clues, however, as to the purpose of this creation.



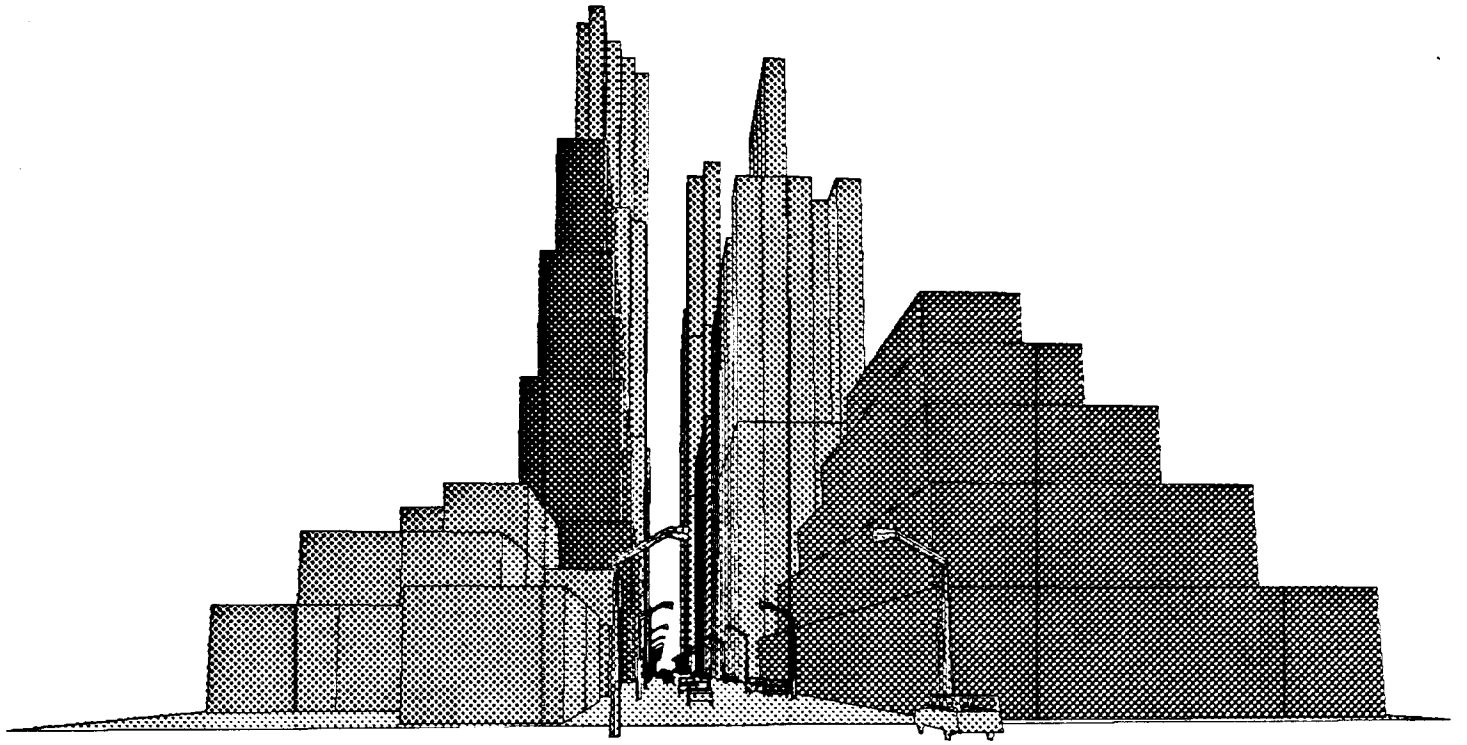
STONEHENGE - 2000

Carlo H. Séquin



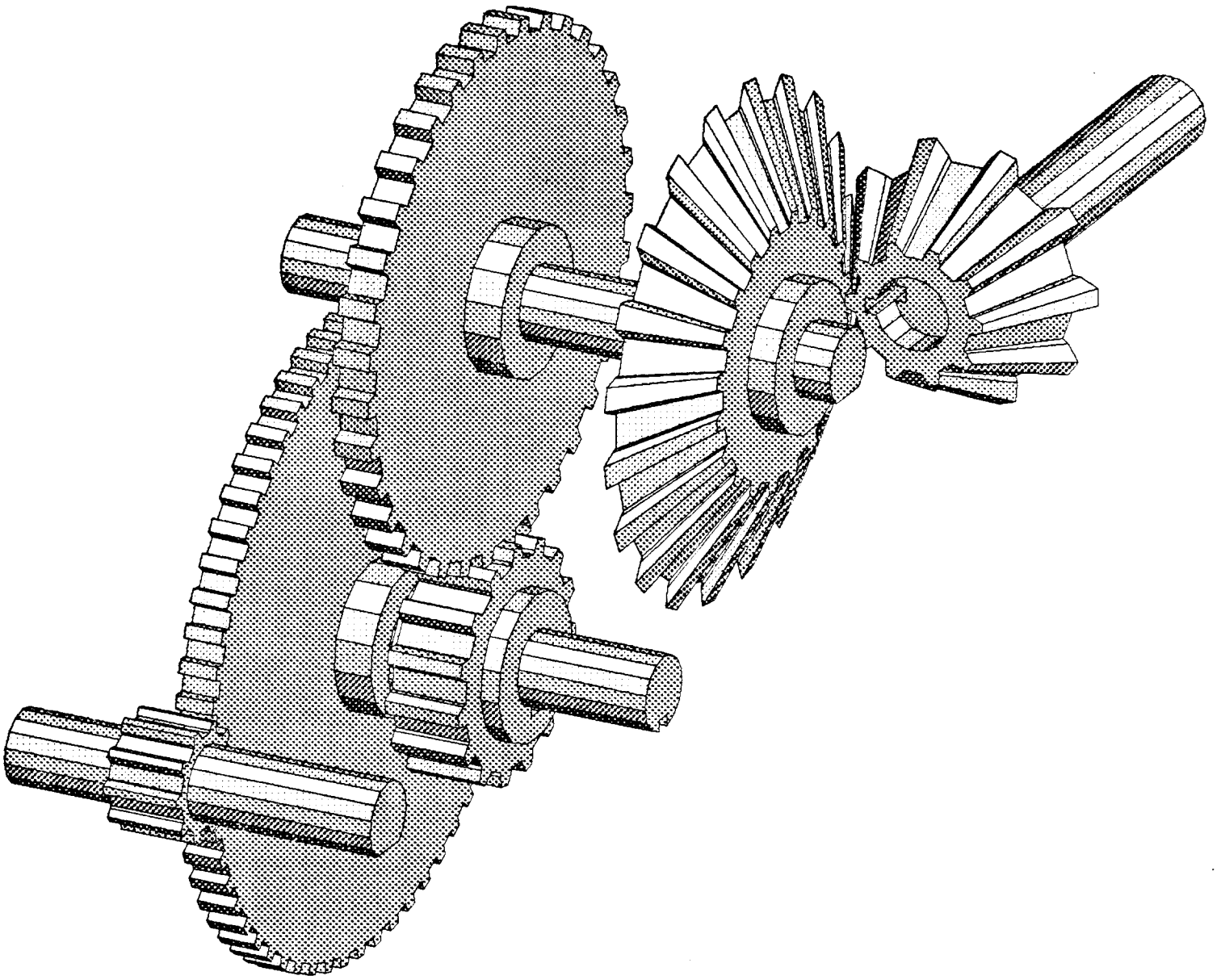
FRACTAL MOUNTAINS

Lun-Shin Yuen



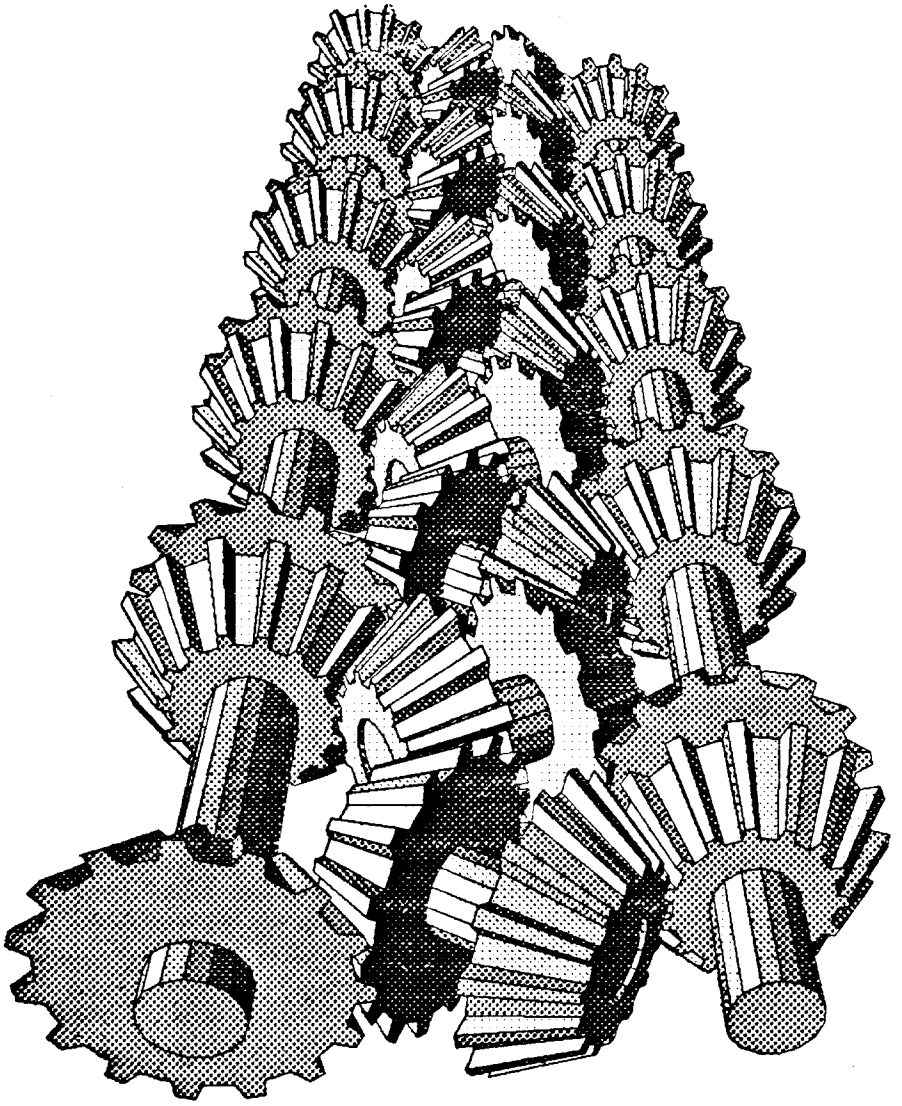
DOWNTOWN TOLEDO, 1997

Mark Gerolimatos



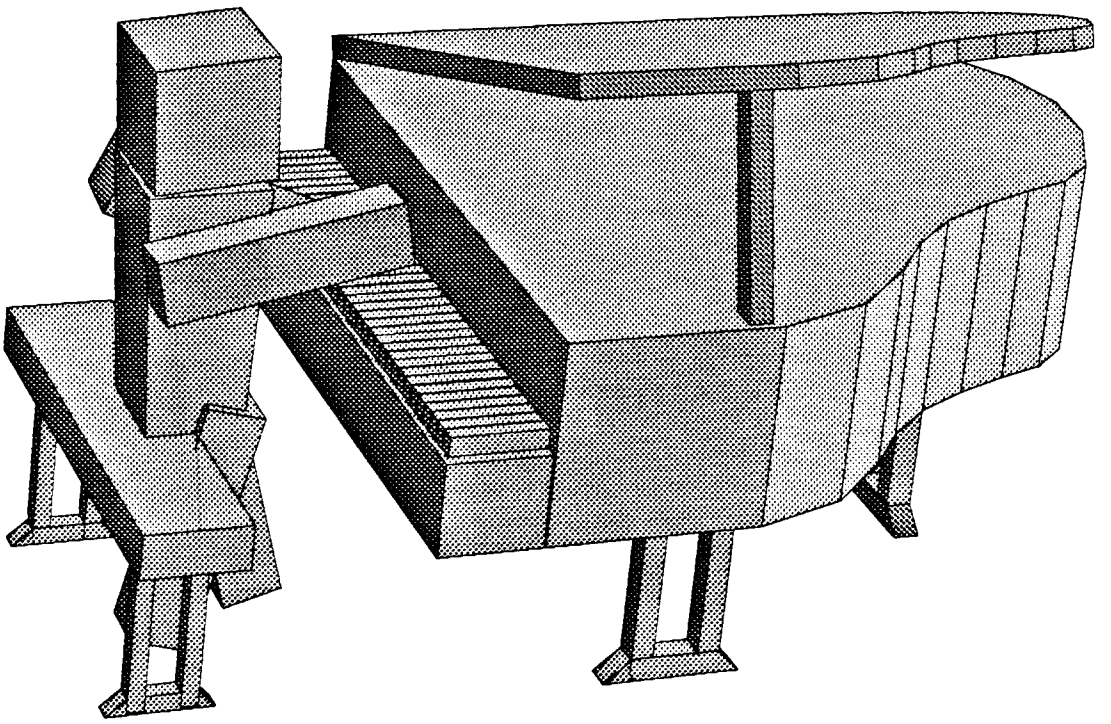
GEAR SYMPHONY

Thomas Laidig



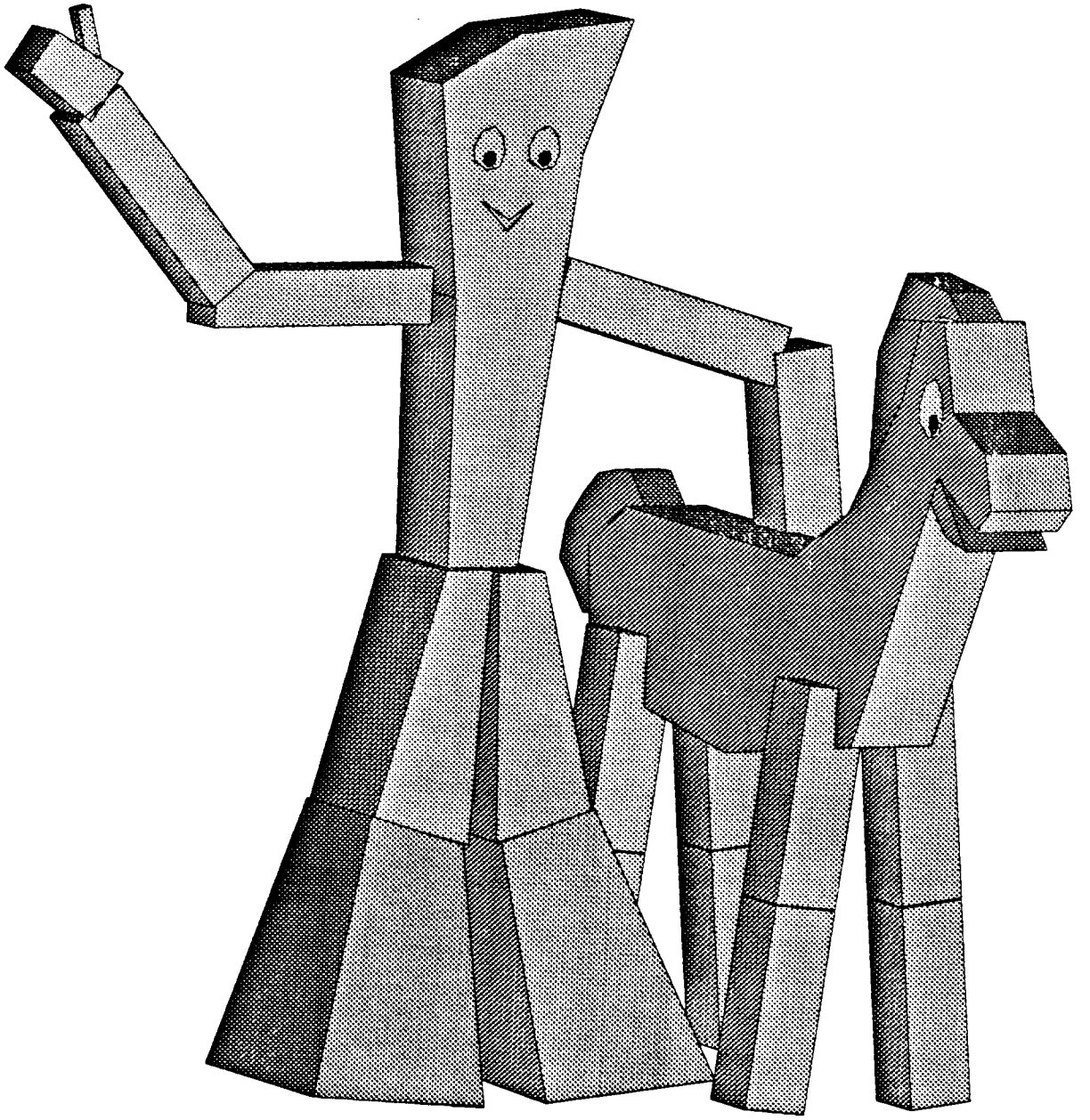
ROBOT DNA

Thomas Laidig



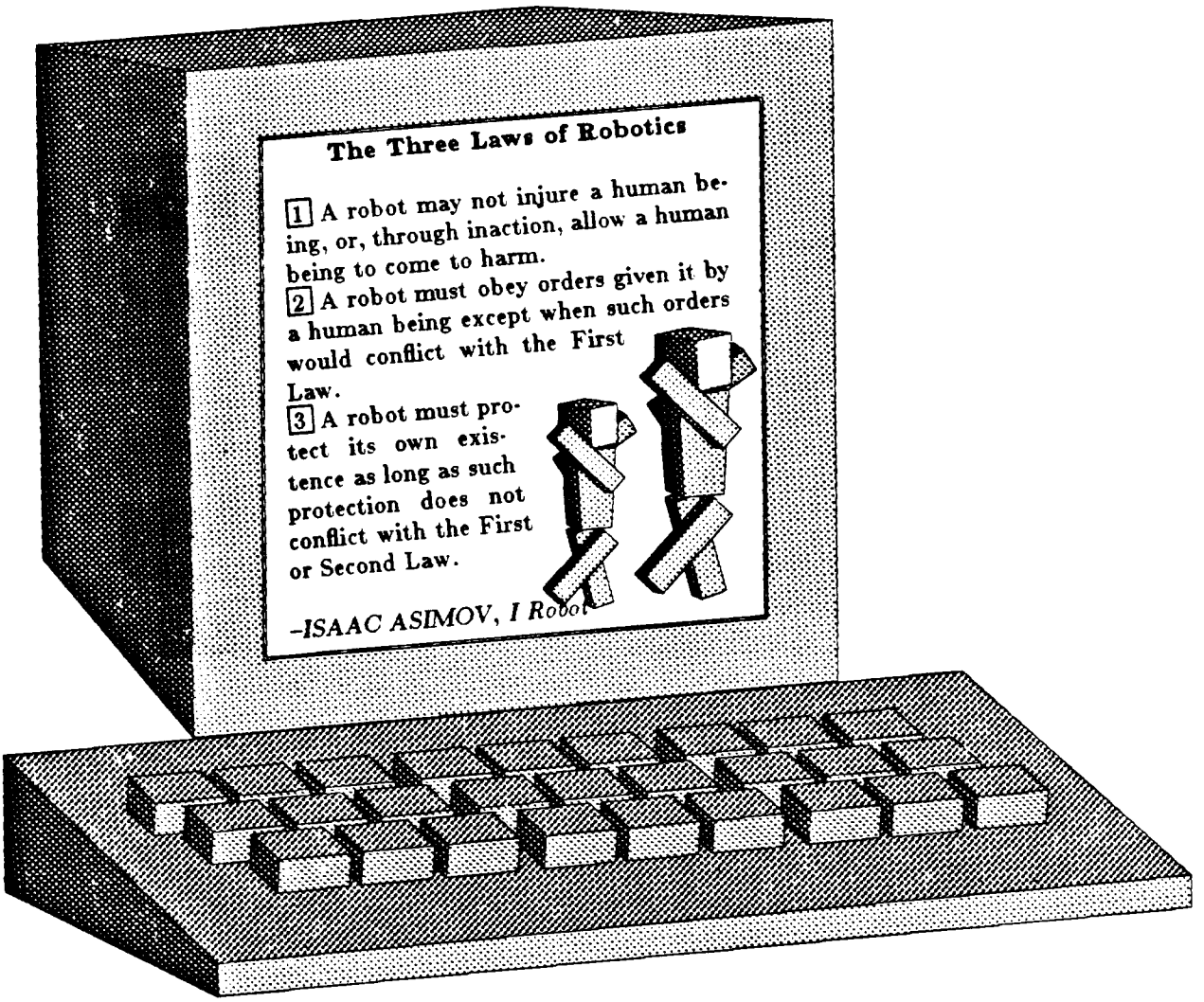
TOBOR AT THE PIANO

H.B. Siegel



GUMBY & POKEY

H.B. Siegel



MARCH OF THE ROBOTS

Nachshon Gal and H.B. Siegel