INTERACTIVE PROCEDURAL MODEL GENERATION

Carlo H. Séquin

With contributions by:

Alonzo C. Addison,
Laura Downs,
Tom Funkhouser,
Mark Halstead,
Milind M. Joshi,
Raja Kadiyala,
Ashutosh Rege,
Joseph M. Rojas,
Ajay Sreekanth,

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

ABSTRACT

This is a report on the fifth offering of a special graduate course on geometric modeling and computer graphics, CS 285: "Procedural Generation of Geometrical Objects". This document is a collection of the student's course projects with a brief introduction and an overview over this year's course syllabus. The projects described include several interactive editing tools for the Berkeley UniGrafix modeling environment, a couple of utilities useful for the analysis of algebraic surfaces and for the visualization of the stability domain of dynamical systems, as well as a highly prameterized generator for tree models of trees for use in architectural scenes. Most projects have been developed on SGI's personal IRIS workstation.

Table of Contents

CS 285, SPRING 1991: OUTLINE OF TOPICS

WEEK 1
- COURSE OVERVIEW
- POSTSCRIPT LANGUAGE
- PROCEDURAL SHAPE GENERATION: POLYGONS

WEEK 2
- PROCEDURAL SHAPE MODIFICATION: OFFSET PATH
- ANISOTROPIC 3D ETCHING
- POLYGON TESSELLATIONS

WEEK 3
- UNIGRAFIX OBJECT REPRESENTATION
- POLYGON INTERSECTIONS IN 2D
- SWEEP-PLANE ALGORITHMS

WEEK 4
- POLYGON INTERSECTIONS IN 3D
- SATHER LANGUAGE & CLASS LIBRARIES

WEEK 5
- 3D CURVES, SPACE-FILLING CURVES
- MAKEWORM GENERATOR
- MAZES

WEEK 6
- MITRING AND BRANCHES
- KNOTS & TANGLES

WEEK 7
- REGULAR POLYTOPES
- SEMI-REGULAR SOLIDS
- ANIMATION PROGRAM

WEEK 8
- EULER RELATIONS IN POLYHEDRA
- NET DIAGRAMS
- EDGE ROUNDING

WEEK 9
- UNICUBIX
- DEVELOPABLE SURFACES
- RECURSIVE STRUCTURES, FRACTALS

WEEK 10
- TILINGS AND PATTERNS
- SYMMETRY GROUPS IN 2D AND 3D
- SPACE PARTITIONS

WEEK 11
- TALKS: SHORT FORMAL PROJECT PROPOSALS
- CONVEX HULL
- THE CURSE OF HIGH DIMENSIONALITY

WEEK 12
- SPHERE PACKING
- APERIODIC TILINGS
- PUZZLE DESIGN
- FINITE ELEMENT MESHES

WEEK 13
- 3D MESHES
- GEOMETRICAL PROBLEM SOLVING
- POLYHEDRAL TOROIDS, MINIMAL TOROIDS

WEEK 14
- REPRESENTING ASSEMBLIES
- CONCEPTUAL DESIGN
- MECHANISM DESIGN
- DESIGN STRATEGIES

WEEK 15
- TALKS: FINAL FORMAL PROJECT PRESENTATIONS
- PROJECT DEMONSTRATIONS

TABLE 1.

CS 285, COURSE OVERVIEW

In 1983 a new graduate course, "Creative Geometric Modeling" was added to the catalog of our offerings in the area of computer graphics.[1] This course which is being offered every other year has seen significant changes over time. Originally it carried the title CS 292A, ''Creative Geometric Modeling''. In this course the students developed some new generator and modifier programs in the UNIGRAFIX framework.[2] These programs were then used to create artistic displays.

In the second half of the 1980's, the emphasis of the course shifted towards algorithms from the field of computational geometry that are useful in the generation of geometric objects such as might be encounterd in CAD/CAM applications by a mechanical engineer or an architect.[3] The students were exposed to a few important algorithms such as offset surface generation or polygon intersections, and they learned about relevant data structures and coding techniques through actual implementation of these algorithms and by testing them on a range of ever ''nastier'' test examples.

This document is a report on the fifth offering of this course course in Spring 1991, under the title CS 285, ''Procedural Generation of Geometric Objects.'' The course extended over 15 weeks with three 1-hour lectures per week. Table 1 gives an outline of the topics covered in this year's offering roughly in the order in which they were presented. This year, the course had a stronger focus on interactive techniques, and the students were asked to emphasize graphical feedback in their projects. This change in focus was a direct consequence of a donation of a cluster of Personal Iris workstations from Silicon Graphics Corporation. The presence of these workstations certainly triggered extra enthusiasm in most of the students, and led to interactive projects which would not have been possible previously.

The course had formal homeworks during the first half of the term and concentrated on individual course projects during the second half. The homeworks during the first half of the course could be grouped into two classes: design exercises at the conceptual level and actual program implementations for a few select tasks. These are briefly outlined in the following pages.

ACKNOWLEDGEMENTS

The class consisted of a group of hard working and enthusiastic students. Many thanks go to Lon Addison, Laura Downs, Tom Funkhouser, Mark Halstead, Milind Joshi, Raja Kadiyala, Ashu Rege, Maurice Rojas, Ajay Sreekanth, for their active class participation and creative contributions.

I would also like to thank Silicon Graphics Inc., and in particular Jim Clark, for their donation of a cluster of 'Personal Iris' graphics workstations which were used intensively in this course.

---

[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 K.P. Smith, "Introduction to the Berkeley UNIGRAFIX Tools, Version 3.0," Tech. Report (UCB/CSD 90/606), U.C. Berkeley, Nov. 1990.

[3] C.H. Séquin, "Procedural Generation of Geometric Objects," Tech. Report (UCB/CSD 89/518), U.C. Berkeley, June 1989.

THE DESIGN EXERCISES

Every week for the first nine weeks, often from one lecture to the next one two days later, the students had to propose some attack to one or two small open-ended design problems and to outline a possible approach to a solution. No long, detailed write-ups were expected — just enough to make the presented ideas understandable. The various proposed approaches were then discussed and compared in class. These were the design problems posed:

RANDOM POLYGON GENERATORS

Describe in one paragraph your proposal for an algorithm to generate a simple convex polygon that could be used as a test vehicle for various computer graphics algorithms concerning such polygons. Among other criteria (for you to discover), the polygons should be reasonably 'page-filling' and should be representative of all conceivable such polygons.

In a second paragraph describe your proposal for a generator of such random polygons that also produces concave but non-self-intersecting polygons.

POLYGON TESSELLATIONS

Make a proposal for an algorithm to tessellate a general concave polygon with possibly several non-intersecting contours into (a) triangles, (b) into monotone (in the y-direction) polygons {i.e., every horizontal scan-line has **at most one** contiguous cutting segment with a monotone polygon}. Use only existing vertices for these tessellations.

Think about how you might partition such a polygon into a reasonably small number of convex polygons.

POLYGON INTERSECTIONS

Outline a possible algorithm to cut up two general co-planar polygons into mutually exclusive parts. Specifically, assume that you have a file with two polygons A and B with possibly multiple contours in the xy-plane. How would you produce three files poly.dif1, poly.dif2, and poly.intr that contain the partial polygons A-B, B-A, and $A \cap B$? Point out potential sources of difficulties and ways to cope with them. In your approach, give preference to robust, logically correct operations that will do the right thing even in the presence of numerical inaccuracies.

In a second paragraph, describe an algorithm to make all intersections among two polygons in R3 explicit: One or both of the polygons may be cut into separate parts by the other polygon. If they intersect only partially, their contours should show infinitely narrow slits (an edge that is used in both directions). All intersection vertices that become part of more than one contour must be explicitly shared by the various output contours.

3D SPACE-FILLING CURVE

Often people design new things by analogy with familiar things. You will try to experience that approach by creating a 3D analog to the familiar 2D space-filling Hilbert curve. Find a piece-wise linear, self-similar curve in R3 that has some "similar properties." To present your solution, make a sketch or a model (wires, pipes, paper), or create a suitable UniGrafix 'wire' description and produce a ugplot output of an early recursive level. Describe the construction procedure for the higher recursive levels. List the criteria that you used to judge the suitability of potential solutions and by which you selected your submitted solution as the ''best'' among other candidates that you may have considered.

3D MAZE GENERATION

Outline an algorithm to make a challenging 3D maze. Split the task into (1) a topological / geometrical part that defines the paths, and (2) into an embellishment part which makes the maze look interesting.

Suppose you wanted to publish 2D projected pictures of your maze (e.g., in a magazine) — what would be the special considerations in the generation of a maze for that purpose ?

VERTEX TRUNCATION IN 3D

Outline an algorithm to truncate the corners of a 3-dimensional polyhedral object. This is not quite as straight-forward as the truncation of a face in 2D that you did recently. For one thing, there are "saddle vertices" that are neither strictly convex nor strictly concave.

Think of ways to make your approach robust so that legal UniGrafix objects will be produced under practically all possible legal inputs. Think of some reasonable user-settable parameters.

SEMI-REGULAR POLYHEDRA

Make a list of all combinations of three or more regular polygons of less than eleven sides sharing a vertex for which the total angle is less than 360 degrees. Which of these combinations lead to regular or semiregular polyhedra ?

Behind each combination write the name of the polyhedron if you recognize it, or indicate in some other way that you are convinced it will form a closed object when the vertex constellation is suitably repeated through R3. Also mark those combinations that will clearly NOT result in a sensible object, and indicate why. See how many combinations you can mark in some way. Explain in a few sentences how you reasoned about these questions.

EDGE-ROUNDING FOR POLYHEDRAL OBJECTS

Make a conceptual plan for an algorithmic approach that would round the edges of polyhedral objects and output the resulting object as a UniCubix file with a mixture of flat and curved patches. Basically the program should preserve a flat center section for each polyhedral face that is "large enough" and "more or less" preserve the original shape of the face. Edges would be converted into pieces of "cylinders" with either a specified or a "suitable" default radius. Think about the terms in the quotation marks.

The user should have some options for explicitly specifying some parameters such as, for instance, the rounding radius. More complicated specifications may lead to pieces of cones rather than cylinders for each of the edges. Discuss other possible parametric specifications of the rounding process. Then discuss what you consider desirable default values for all the parameters that you specified and what the default result of rounding an arbitrary polyhedron should be. Think about how you would deal with convex and concave corners. Think about potential "problem-geometries" where simple rounding procedures might fail. Give a short outline of the algorithm or of the main program modules that would achieve the above goals.

RECURSIVE SIERPINSKY-INSPIRED SURFACES IN R3

Your assignment is to propose a recursive generation procedure for a volumetric object in R3 bounded by a polyhedral surface with self-similar parts. Look at the Sierpinsky curve for inspiration by considering it the surface of a 2D object.

Your construction need not follow this particular shape; it is just one example how a surface might be constructed in a recursive self-similar manner. Present sketches of an early level of recursion of your object and give a short description that explains the construction. Feel free to use UniGrafix if that makes it easier for you. Mention some of the conceptual difficulties that you encountered when you tried to construct such a shape in 3D.

2D PUZZLE GENERATION

Consider puzzles in 2D consisting of identical or different, marked or unmarked tiles. Try to give answers to the following questions:

- What makes such puzzles hard ?
- What is the hardest puzzle you can construct with only unmarked rectangular tiles ?
- What is the hardest puzzle you can construct with only marked square tiles ?
- What is the hardest puzzle you can construct with only three unmarked pieces of arbitrary shape ?

Make a proposal for one type of puzzle generator.

FIND THE SIMPLEST TOROID

To further develop your 3D visualization, design, and sketching skills, I ask you to contemplate the following polyhedral solid objects of genus 1, so called "toroids."

- Design a toroid with a minimal number of faces.
- Design a toroid with a minimal number of vertices.
- Design a toroid with a minimal number of regular faces, i.e., a mixture of regular n-gons.
- Design a toroidal polyhedron of genus 5, that is as highly symmetrical as possible, i.e., which has the most symmetry operations that put the object back onto itself.

Convey your ideas in some appropriate form: sketch, paper model, UniGrafix description, or procedural outline of construction ...

3-FOLD HIGH-WAY CROSSING

Design a 'clover-leaf' structure to interconnect three major divided high-ways that cross more or less in one spot.

To present your solution: make a sketch or model (wires, paper) of the basic topology of this 'cloverleaf.' Describe in one paragraph the criteria that you used to judge the suitability of potential solutions and by which you selected your submitted solution as the "best" among the candidates that you considered.

MOSTLY UNSTABLE POLYHEDRON

Design a convex polyhedral solid made from material of uniform density that is stable (i.e., that does NOT roll over an edge to an adjacent face) on as FEW of its faces as possible.

To present your solution: make a sketch or model and describe in a couple of paragraphs how you started your search, what kind of bounds you established, and how you might continue your search or refine an initial design.

PROGRAMMING ASSIGNMENTS

On a few occasions, the class was then asked to follow up with an actual implementation of one of the more promising looking approaches. In some instances, algorithms were limited to two dimensions to keep coding complexity and implementation time reasonable. It was suggested that some of these algorithms could be carried to 3 or 4 dimensions as individual projects in the second half of the course.

OFFSET-PATH GENERATION AROUND CONVEX POLYGON

An offset-path is a path that runs at a fixed distance from a given path. It is used to describe the result of growing a polygon by a strip of fixed width or of etching away a certain thickness of material from the contour of the polygon. It is also the path along which a circular milling tool has to be moved to cut a certain shape from a larger piece of material.

Implement suitable algorithms for (a) generating a simple convex polygon and (b) the offset-path at distance of 0.5 inch around such a polygon. (Suitable circular arcs should be constructed around convex corners). Program (b) should output a PostScript file describing the original polygon (filled) as well as the offset path at a distance of 1/2 inch (36 units) around it. Hand in your program listings and pictures of the output of program (b) for two of your own generated polygons. You may want to modularize your efforts and provide the two programs with a "standard" ASCII interface. Program (a) should then produce its result as a file with N lines, each containing an x,y coordinate pair for one of the polygon vertices, listed in CCW-order around the contour. The coordinate pairs should be integer numbers in PostScript units, i.e., 1/72 of an inch; the result should fit into a rectangle of 5 by 8 inches. Program (b) should then read such a file of x,y pairs and do its job.

CONCAVE POLYGON TRIANGULATION

Implement a suitable algorithm for tessellating a simple concave polygon (single non-self-intersecting contour, no holes) into triangles. There will be two test polygons, 'tessel1' and 'tessel2', provided. They will be described as an ordered list of vertices, one integer coordinate pair in PostScript units per line such as:

36 72 {vertex with x=36 points and y=72 points}

Hand in the most relevant parts of your program listings (but not more than 4 pages) plus pictures of the outputs of your program for the tessellated polygons 'tessel1' and 'tessel2'.

POLYGON TRUNCATIONS

This is an exercise to get familiar with and learn to use the UniGrafix ug3-data structures. You will write a program to truncate arbitrary "well-behaved" polygons (non-intersecting contours with nesting with alternating orientations). Construct your program around the ug3 data structures using some of the existing routines (e.g. reading/writing UniGrafix files).

Your program should truncate all inside and outside corners of the given polygons. For each vertex it should move along the two attached edges by some user-definable fractional amount "t" between 0.0 and 0.49 (default: t=0.25) of the edge length, establish two new vertices, connect them by a chord — thus cutting off and eliminating the original vertex. Don't worry about possible intersections.

Your program should also have an option to repeat the truncation process on the established data structure "n" times, thus producing finer and finer truncations and smoother and smoother curves. (As an escape, if this repetition feature is taxing your programming skills or patience, you can also run your program repeatedly over its own ASCII output.)

Apply your program to the test polygons trunc1 and trunc2. For each polygon show the output for a single pass (n=1, and t=0.4) and also for a triple pass (n=3, t=0.25). Hand in the most relevant parts of your program listings, 4 pictures of outputs produced by ugplot, and 2 listings of the UniGrafix ASCII output for the single-pass truncations.

UNIGRAFIX FURNITURE GENERATOR

This is an exercise to learn how to describe 3D objects in UniGrafix format. Your assignment is to write a program that produces a limited parameterized furniture assembly as specified below. The generator program should read the necessary arguments from the command line and produce as output a UniGrafix ASCII file describing that assembly. Do one of these generators:

(a) **Books_on_Shelves**. Create a simple wall of bookshelves, H inches high, D inches deep, and overall L inches long. These shelves should consist of several simple sections with sidewalls and shelves (transformed cubes), loaded with books (also transformed cubes) in a not totally boring manner.

(b) **Terminal_Room**. Create the basic furnishing of a terminal room: a block of depth D and width W, containing (optionally specified) R rows of tables with S workstations on each at regular spacings. For the tables use simple top plates supported by simple legs (could all be transformed cubes). Use a call to a couple of definitions for the actual workstations and for the chairs in front of them. Build yourself a trivial model of each definition for debugging purposes. Later we will substitute fancier models.

(c) **Conference_Table**. Create the basic furnishing of a small seminar or conference room: an area of length L and width W, containing a rectangular, octagonal, hexagonal, or oval conference table with chairs around its perimeter. Construct the table and its supporting structure, but call macro definitions for the chairs for which you should make only a trivial placeholder model.

All these generators need a little decision tree that sets some of the explicit low level parameters based on the overall dimensions or more global parameters specified. For instance, bookshelf sections should be about 2-3 feet wide, and chairs should be spaced about 3-4 feet apart. The overall dimensions thus determine how many sections or chairs will fit; but this calculated number could be overridden by an explicit specification. It is OK to be creative.

Hand in a paragraph from the "manual page" that describes the options and decisions in your generator and pictures of the outputs of your program for two different parameter specifications.

DESIGN A SIMPLE KNOT AND RENDER IT WITH UGWORM

Create a simple, closed, piece-wise linear path through space that forms a knot. Create by hand or with a simple program a corresponding "closed wire" definition describing the axis of the desired prismatic tube through R3. Then run *ugworm* with some suitable radius and with a low number of prism sides. (e.g.: ugworm axis_wire -r 0.2 > Your_knot) Inspect the resulting object with ugiris and render it with *ugplot*.

MODIFICATION OF REGULAR POLYHEDRA

Apply some of the existing object modifiers to some regular or semi-regular polyhedra from the library { ~ug/lib }. Play with several modifier programs for a couple of hours and watch the results with ugiris.

Then fabricate TWO interesting creations and produce hardcopy that you will hand in. The creations do not have to be overly complex. I will be more impressed with original combinations of 2-4 modifiers applied in series to a simple object. On your hand-in, describe the process by which you have created your artwork.

ANIMATE: FISH INTO FOWL

Create a simple, closed surface that represents object A (FISH) and another surface with the same connectivity of its edges that represents object B (FOWL). A simple way to do this is to make a two-dimensional outline of the desired shape and then *ugsweep* it to give it some thickness. Of course it would be nicer if the surface is not simply prismatic. This can be achieved by using a more complicated sweep or by subjecting the prismatic object to *ugstar*.

Now use FISH and FOWL as the endpoints of an animate sequence and watch the transformation occur as you move the slider. Produce hardcopy output of four plots: FISH, FOWL, and two intermediate shapes. Describe in a couple of sentences how you generated the initial objects.

As a second assignment think of a way to transform an icosahedron into a dodecahedron. Produce a plot of the object about half-way in between the two Platonic solids.

UNICUBIX WORKOUT

This exercise is aimed at getting you familiar with the UniCubix program and at finding its limitations. Make a somewhat irregular but pleasing looking object that has a genus of at least 2, which is mostly smooth, but has at least one sharp edge. It is OK to construct the original polyhedral shape by using some of the generator and modifier programs that you have explored in a previous assignment — but keep the object simple !

Hand in plots of your starting shape and of two different versions of the final shape produced with different UniCubix parameters. Also add a short description of how you generated your object and what problems you encountered with UniCubix.

COURSE PROJECTS

In the second half of the course, The students had a chance to choose a project of their own design or to pick one from a list of suggestions. This provided them with an opportunity to take a deeper look at some area of graphics, modeling, or geometry that they were particularly interested in.

The main part of this document is the collection of the final project reports. I did no editing on the text, except for culling out lengthy source code listings or almost redundant figures. I hope the authors will forgive me for taking these editorial liberties.

qm2ug

A Geometry Translator from Alias QuickModel 1.4 to Berkeley UniGrafix3

Alonzo C. Addison

Computer Science 285/299—
Procedural Generation of Geometrical Objects/Independent Study
Professor Carlo Séquin
Spring 1991

Introduction

The Berkeley UniGrafix 3 Tools, developed over the years in the Department of Computer Science at UC Berkeley, provide a powerful set of geometric generation, manipulation, and visualization algorithms. In the wake of Silicon Graphics' generous gift of Personal IRIS classrooms to the UC Berkeley campus, UniGrafix has been adapted to take advantage of the IRIS' legendary 3D hardware and has thus become more accessible and useful. However, despite the power of the individual UniGrafix tools, the absence of a 3D modeler has limited their penetration outside of Computer Science.

The QuickModel to UniGrafix Translator (*qm2ug*) was written to simplify the creation of UniGrafix models and to help bring the UniGrafix tools into the domain of the non-technical user. Students in the Department of Architecture at Berkeley are exploring 3D design and visualization on the IRIS workstations. Although several high-end modeling/rendering/animation packages have been donated (Alias Research's Alias/3, Wavefront's entire Visualizer family of products, and Sigma's ARRIS CAD), many of the sophisticated UniGrafix routines (which are essentially not available in any commercial software) could prove useful to these students in creative projects.

The Department of Architecture currently owns numerous modelers, each with its own proprietary geometry format. These include Alias/3, the Wavefront Visualizers (Personal, Data, and Advanced) and ARRIS CAD on the IRIS; McDonnell Douglas GDS on the MicroVAX; AutoCAD/11 on the IBM PC; and MacArchitrion, Swivel3D, Super3D, ModelShop, and Upfront on the Macintosh. Clearly, attempting to support all of these would be impossible. Since Alias/3 and the IRIS are the current package and machine of choice, this project started out as an Alias/3 'wire' format translator. However, as work got under way it was realized that although an Alias/3 translator might be useful, the high cost and complexity of Alias/3 would limit the usefulness of the tool.

Alias QuickModel, an entry-level modeler provided free of charge with Silicon Graphics workstations, provided a nice compromise. Although less powerful than other modelers in use, it is a straightforward package with basic 3D modeling abilities. Its availability (free with IRIS workstations) and ease of use (documentation is also provided) made it the ideal candidate for a basic modeling front-end to UniGrafix. As the lowest common denominator among modelers on the IRIS, QuickModel is supported by the more powerful packages—Alias provides a QuickModel to Alias/3 translator (qmtoalias) and Wavefront supports the QuickModel format in the Personal Visualizer (also available free of charge with the IRIS).

Technical Description

qm2ug translates Alias QuickModel 1.4 geometry to Berkeley UniGrafix 3 format. Its features, structure, and limitations are described below.

If called from the command line, it will translate the file directly and place it into a new file, "output.ug". In addition to the manual page, usage information is available. If called without any arguments or with incorrect arguments, the usage information shown below will appear:

```
-------------------------------------------------------------------------
Usage: qm2ug [-aAngles#] [-sSphere#] [-w] [-v] [-o outfile.ug] infile.qm

Where:  -aAngles#     is the number of facets of a conic section
                        (default 8; valid range 6 to 720)
        -sSphere#     is the sphere complexity
                        (default 1; valid range 1 to 4)
        -w             specifies wireframes instead of faces
                        (default is to use faces) (NOT fully functional)
        -v             specifies verbose mode...
                        (data on progress will appear)
        -o outfile.ug  forces output to the named file instead of 'outfile.ug'
-------------------------------------------------------------------------
```

The program structure is relatively straightforward. It performs its task in the following steps:

- scan command line for arguments
- set flags as appropriate based upon arguments
- read in input file one line at a time
- verify that it is a QuickModel file
- switch to appropriate routine based upon the object type
- generate appropriate UniGrafix equivalents for types as shown in the table below
- formats statements in appropriate order and writes out results

QuickModel	*UniGrafix*	*UniGrafix explanation*
cube	v and f	6 faces arranged to form a cube
cone	v and f	x faces + 1 cap arranged to form a cone
cylinder	v and f	x faces + 2 caps arranged to form a cylinder
sphere	v and f	1 of 4 faceted spheres of increasing complexity
surface	f	appropriate faces
cv	v	vertices for the surface
translation	i -tx -ty -tz	translation portion of an instance statement
rotate	i -rx -ry -rz	rotation portion of an instance statement
scale	i -sx -sy -sz	scale portion of an instance statement
tm	i -M4	homogeneous transformation matrix
light	l	point light (with intensity 1 in all axes)
color	c	color transformed from RGB to HLS format
diffusion	{qm...}	placed in a comment field
specularity	{qm...}	placed in a comment field

qm2ug handles all QuickModel objects types as shown in the table above. These include the primitives (cube, cone, cylinder, and sphere) as well as curves and rotated, patched, and extruded surfaces. It should be noted that there is a model limit of 50 objects in any single QuickModel file. If a user wants to create a larger object, they should split it across multiple files (this will also enhance performance and response in QuickModel.

The first time a new type of primitive is encountered, its vertices are written into the UniGrafix file as a definition. For every primitive the faces are placed into a separate definition file with the appropriate colors tagged onto them. It was necessary to separate them from the vertices (instead of making a single definition of vertices and faces) as the UniGrafix manual indicates that colors can only be set in face, wire, or vertex statements and not it the patch or array statements. If this is

really the only way to handle colors, it is a limitation of UniGrafix which needs to be addressed.

Cubes are defined by the 8 vertices of a unit cube. Cones and cylinders are handled jointly since the vertices of one end of the primitive cone & cylinder are equivalent and it seemed wasteful to write them twice. The number of vertices around the great circle is definable in the -a (angles) command line flag. The default is 8. To be able to use the UniGrafix include command, the vertices of the cube/cone generated are written out to files in the current directory. Spheres are similarly defined via a command line flag. The -s flag may be used to specify one of 4 increasingly detailed spheres. They are similarly called via include statements in the UniGrafix file, so need to be available in a known directory.

qm2ug handles all QuickModel surfaces (which may be generated by patching, rotating, or extruding curves). *qm2ug* currently translates curved surfaces as planar faces. To ensure planarity of faces, the translator divides the faces (even planar ones—it does not stop to check) of surfaces into triangular pieces. Although QuickModel has curved surfaces, a transformation to UniGrafix's UniCubix extension was not straightforward. This is an area for a future addition. If the user wishes to view the surface in QuickModel as it will appear in UniGrafix, the smooth/straight toggle should be set to straight. This will modify the u and v flags for the appropriate vertex of the surface. However, since the translator currently ignores the u and v flags, toggling the surface to "straight" is not required.

The transformations (rotate, scale, translate, and tm) are placed in the UniGrafix instance call. Although I considered multiplying all the transformations out to generate a single matrix, I decided it was better to leave them unaltered. Since QuickModel places them in the incoreect order when generating its file (this appears to be a QuickModel bug/flaw), I read them all in and rearrange them before writing out the instance statement.

Although UniGrafix supports faces with holes, QuickModel is unable to generate them so they did not need to be handled. (QuickModel can create a multi-faceted surface with a hole by patching between two planar curves—this is transformed as a normal patched surface.) It is possible for the user to create self-intersecting polygons in QuickModel. No attempt is made to intercept these in the transformation.

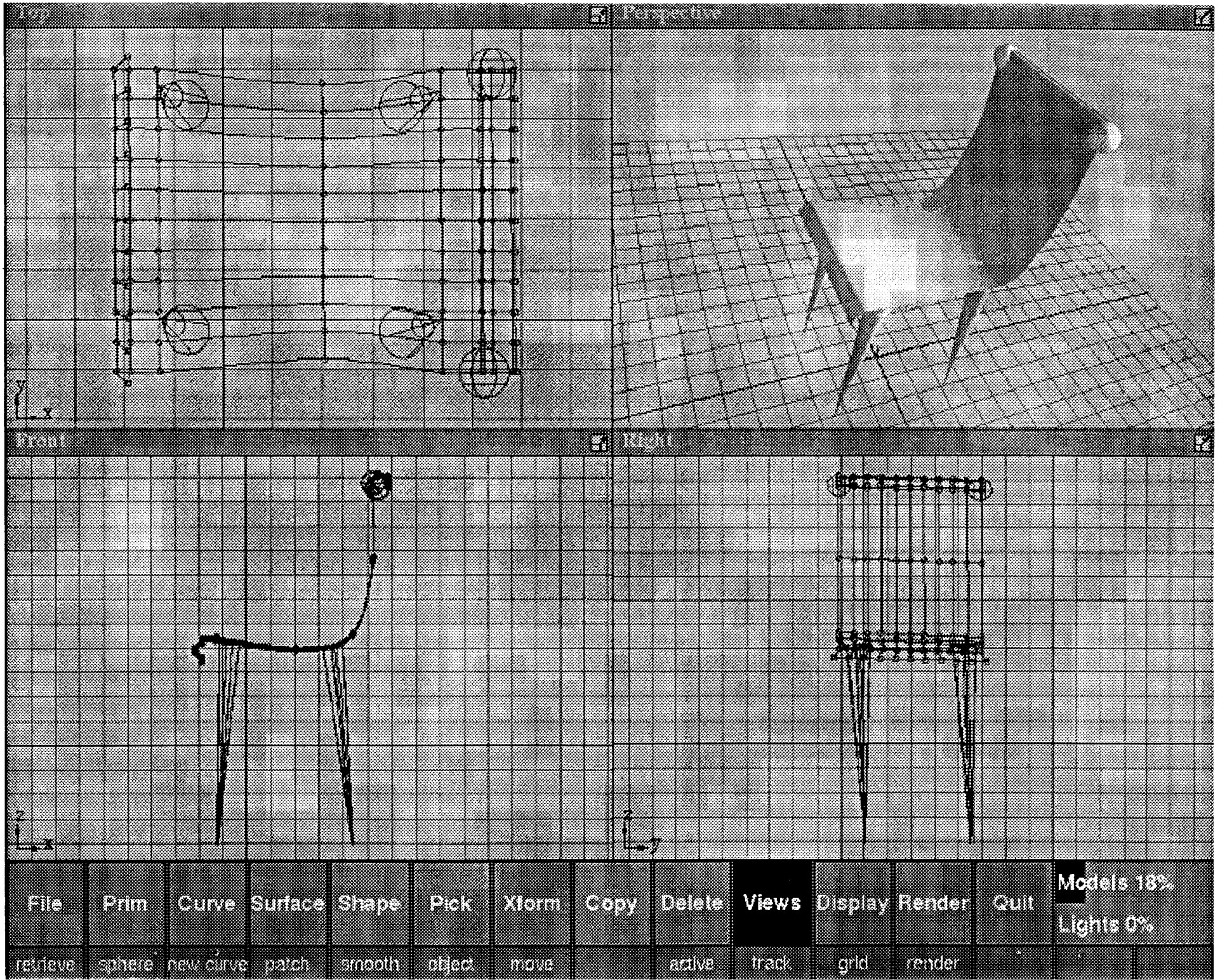
The only form of light source available in QuickModel is a UniGrafix point source of uniform intensity in all axes. It is transformed without problem. Note that there is a limit of 8 light sources in any QuickModel scene.

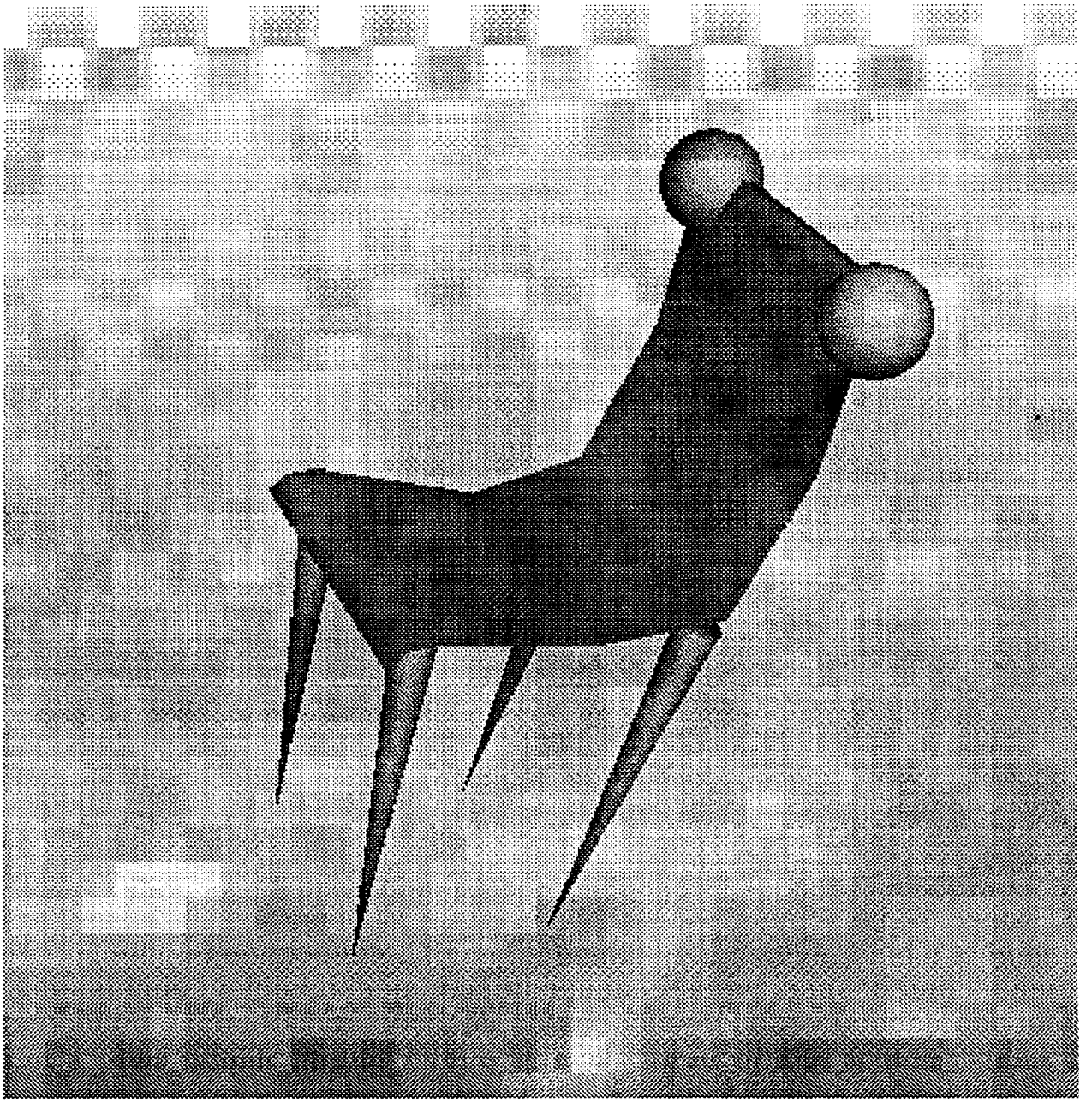
Specularity and diffusion are not handled as the basic UniGrafix3 library does not include them (they are stored in comment statements for future translation back into QuickModel however). Since UniGrafix utilizes the double-coned HLS (Hue, Lightness, Saturation) color model, the translator makes the conversion from QuickModel's RGB (Red, Green, Blue) color definition.

Summary

I have enjoyed writing *qm2ug* and am hopeful that it will prove useful to both technical and non-technical (such as the architecture students in my home department) UniGrafix users. Although perhaps structurally straightforward, qm2ug proved challenging given my limited programming background. Writing it was a very worthwhile educational experience and has given me greater confidence in my ability to program. Perhaps if there is demand I will tackle the task of ug2qm, alias2ug, ug2alias, etc. in the future!

I am hopeful that this effort will help to foster interdisciplinary collaborations between the Departments of Computer Science and Architecture at UC Berkeley, as the members of both groups would, I believe, benefit greatly.


File
Prim
Curve
Surface
Shape
Pick
Xform
Copy
Delete
Views
Display
Render
Quit
Models 18%
Lights 0%
retrieve
sphere
new curve
patch
smooth
object
move
active
track
grid
render



NAME

qm2ug - convert from QuickModel to UniGrafix format

SYNOPSIS

qm2ug [-aAngles#] [-sSphere#] [-w] [-v] [-o outfile.ug] [filename...]

DESCRIPTION

ug2qm (QuickModel to UniGrafix) converts Alias Research QuickModel 1.4 geometry files to Berkeley UniGrafix3 format.

All QuickModel object types (cube, sphere, cylinder, cone, light, color, surface (and cv), diffusion, and specularity) are handled. Note that diffusion and specularity, which are undefined in the basic UniGrafix format are put into comment statements however.

OPTIONS

-aAngles#
is the number of facets of a conic section (default 8; valid range 6 to 720)

-sSphere#
is the sphere complexity (default 1; valid range 1 - 4)

-w specifies wireframes instead of faces (default is to use faces) (NOT fully functional)

-v specifies verbose mode... (data on progress will appear)

-o outfile.ug forces output to the named file instead of 'outfile.ug'

SEE ALSO

qmtoalias - a translator from QuickModel to Alias/3.

DIAGNOSTICS

Exit status is normally 0. If the last file was not correctly translated, an error message will appear and exit status will be 1.

Error Messages -- should be self-explanatory

Usage: qm2ug [-aAngles#] [-sSphere#] ... ...]

Error: trouble opening 'filename'

Error: 'filename' is not an Alias QuickModel 1.4 file

Error: Encountered undefined geometry type in 'filename'

Error: Unmatched brackets in QuickModel file

Error: uh oh... serious internal problems...

BUGS

Quick testing has not revealed any "bugs". Please notify the author if you come across any.

There are several "flaws". If one creates a self-intersecting polygon it will be translated as is, creating potential problems in UniGrafix. Curves and surfaces are approximated as polygonal facets. The "-w" (wire) mode is not yet fully functional. It is recommended that the user simply use the default face mode and use the wireframe toggle in another UniGrafix tool such as 'ugiris' if a wireframe view is desired.

AUTHOR

Alonzo C. Addison (addison@ced.berkeley.edu)
Dept of Architecture, UC Berkeley

(With examples from Kevin Smith and the UniGrafix library authors.)

CS285 Final Report:
An Interactive UNIGRAFIX Editor

Thomas Funkhouser

May 14, 1991

1 Introduction

The UNIGRAFIX environment [2] includes numerous batch tools for making global modifications to polyhedral objects. For instance, there are tools that truncate all the corners, tesselate all the faces, or bevel all the edges of a polyhedron. However, until this semester, there was no universally available interactive UNIGRAFIX tool that allows a user to make small, local changes to a polyhedron. I think such a program would be extremely valuable. It could be used for creating simple irregular polyhedra or making local modifications to existing polyhedra. As a result, I decided to build an interactive UNIGRAFIX editor for my CS285 class project.

In short, the major features of the program are: 1) creating, opening and saving UNIGRAFIX files, 2) translating, scaling and rotating UNIGRAFIX files, 3) running existing UNIGRAFIX tools, 4) creating, deleting, merging and moving vertices, 5) creating, deleting, collapsing and subdividing edges, 6) creating, deleting, collapsing and flipping wires, and 7) creating, deleting, collapsing, flipping, and subdividing faces. All these operations are invoked interactively using a combination of direct-manipulation and command-based interfaces with real-time feedback.

Rather than write a new program from scratch, I decided to enhance the Animator [3] since it already supported most of the file manipulation, viewing and modeling tool features required by my program. For readers not familiar with the Animator, I have included a brief overview of the original version of the Animator in the second section. The third section is the bulk of the report - it describes the new interactive editing features. For each feature, I include a description of the user interface and decisions I made during design and implementation. In the fourth section, I discuss some of my experiences using the program and present several objects I created using the program. Finally, the last two sections contain suggestions for future work and a brief conclusion.

2 Overview of the Animator

The original version of the *Animator* is an interactive modeling tool written by Kevin Smith that allows a user to view polyhedral objects and globally modify them using a suite of UNIGRAFIX tools. In particular, the Animator allows one parameter of a UNIGRAFIX tool to be linked to a slider, in which case the polyhedron is modified in real-time as the value of the slider is changed by the user. The new version of the Animator is similar to the original one, but it also includes an interactive interface for making local changes to UNIGRAFIX objects.

As shown in Figure 1, the Animator has three windows - the main viewing window, the control panel and the slider. The *main viewing window* allows the user to view the polyhedron - rotating,

scaling and translating it in real-time using combinations of the mouse buttons and keyboard keys. The user also may control several aspects of how the polyhedron is displayed in the main viewing window. There are options for toggling on or off face display, wire display, edge display, vertex display, backfacing face display, or transparency of the polyhedron.

The *control panel* is used to enter commands and specify options. At all times, it contains pull-down menus with commands for loading and saving UNIGRAFIX files, running UNIGRAFIX tools, executing editing commands and selecting program options. In addition, it displays controls, such as buttons, checkboxes and edit fields, with which the user can specify settings for tool or program options.

The *slider window* appears whenever the user runs a UNIGRAFIX tool with a dynamic parameter. The value of the slider is linked to the value of the dynamic parameter. When the user manipulates the slider, the value of the dynamic parameter is changed, then the tool is run with the new parameter, the polyhedron is modified accordingly, and the display is updated.

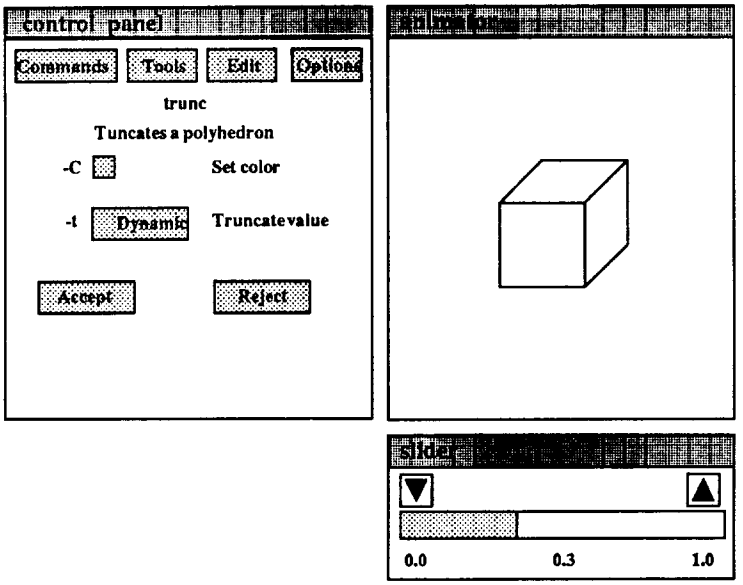


Figure 1: The three windows of the Animator

3 New Interactive Editing Features

This section describes the new interactive editing features of the Animator.

3.1 Creating UNIGRAFIX Files

The original version of the Animator has a pull-down menu interface for loading and saving UNIGRAFIX files. Using this pull-down menu, the user can open any existing UNIGRAFIX file, or save the current polyhedron in a file. The Animator now has an additional file manipulation command on this pull-down menu that allows the user to open a new, empty UNIGRAFIX file. Using this command, the user can create a new UNIGRAFIX file from scratch, rather than having to derive it from another file. In fact, when the Animator starts, an empty UNIGRAFIX file is opened by default. At this point, the user may either begin adding UNIGRAFIX objects to this default file or load another file.

3.2 Creating UNIGRAFIX Vertices

The Animator allows a user to create new vertices in the currently open UNIGRAFIX file. The user can create a new vertex at the current 2D cursor position by double-clicking with the left mouse button. Since the 2D cursor position represents a line through 3D, the actual position of the new vertex is dependent on what type of object lies under the cursor position at the time of the command. If the interior of a face lies under the cursor position, then a new vertex is created positioned at the intersection of the 3D line representing the 2D cursor position and the plane of the underlying face. The new vertex is connected to the other vertices of the face by constructing new edges and contours that tessellate the face, as shown in Figure 2. On the other hand, if an edge lies under the cursor position, then a new vertex is created positioned at the intersection of the 3D line representing the 2D cursor position and the 3D line along the edge. In this case, the underlying edge is split into two new edges that connect its endpoints through the new vertex, as shown in Figure 3. Otherwise, if no face or edge lies under the cursor position, a new vertex is created that is not attached to any other object in the file. The position of the new vertex is set to be the intersection of the 3D line representing the 2D cursor position and the axial plane (XY, XZ or YZ) whose normal vector is directed most toward the viewer, as shown in Figure 4. Using these commands, the user can add any number of vertices to a UNIGRAFIX file.

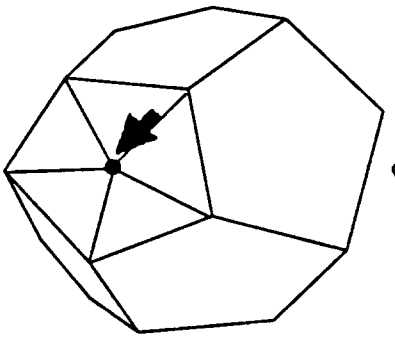


Figure 2: Creating a vertex on a face

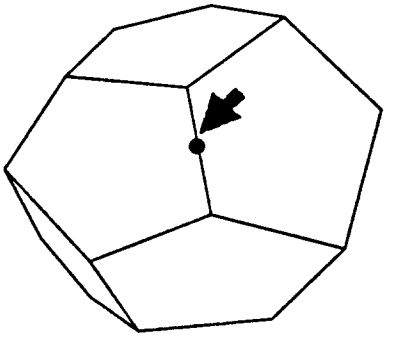


Figure 3: Creating a vertex on an edge

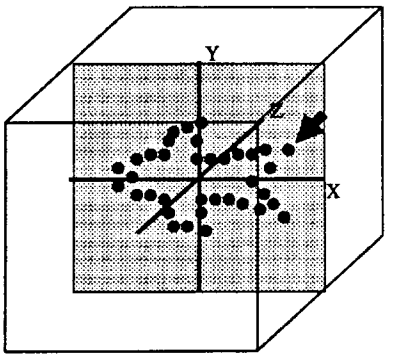


Figure 4: Creating a vertex in space

3.3 Selecting UNIGRAFIX Objects

The user interface for interactively editing polyhedra is based on a *currently selected set* of UNIGRAFIX *objects* (vertices, edges, wires and/or faces). All objects in the currently selected set are drawn in a special color (yellow) and all local editing operations are applied only to these objects. The user replaces the previously selected objects with a new object by simply clicking on the object with the left mouse button. To extend the current selection (i.e. add an object to the currently selected set without replacing the previously selected objects), the user must hold down the shift key when clicking on the object with the left mouse button.

The code for selecting the front-most object with the mouse was one of the most difficult aspects of this project. The GL library provides a function (*pick*) that returns all objects lying under the current cursor position. In order to select the front-most object, I had to find the line in 3D space representing the 2D position of the cursor, and then sort the parametric values of the intersections between this mouse line and all the objects returned by the GL *pick* function, as shown in Figure 5. Furthermore, since many objects can intersect the mouse line at approximately the same parametric value, my code had to resolve conflicts in such situations. I chose to give vertices priority over edges,

and edges priority over faces.

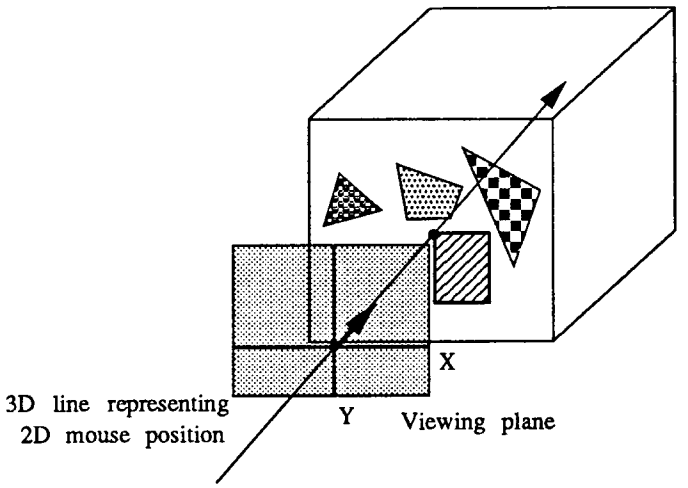


Figure 5: Selecting Objects

3.4 Moving UNIGRAFIX Vertices

The user can move the currently selected set of vertices by moving the mouse with the right mouse button down. When the right mouse button is pushed down, an axial triad cursor appears whose origin corresponds with the first vertex in the currently selected set (the triad is intended to help the user visualize the 3D movement). Then, while the right mouse button is held down, each movement of the mouse in 2D results in a movement of the all the currently selected vertices in 3D, as shown in Figure 6. When the right mouse button is released, the triad disappears and the selected objects keep their new positions.

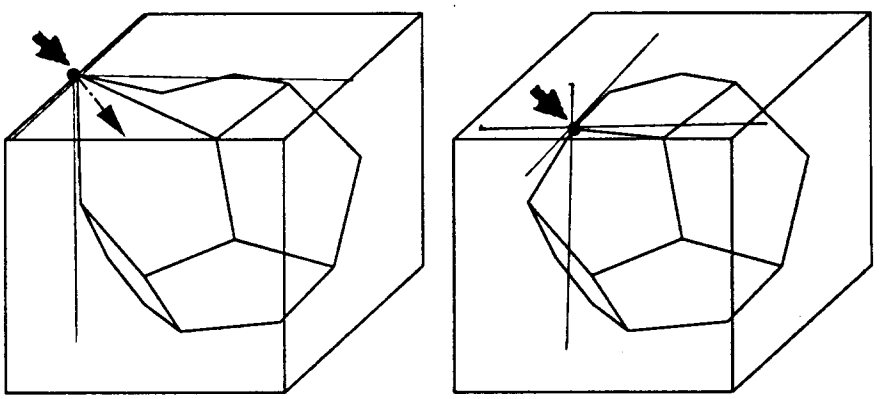


Figure 6: Moving vertices

Mouse movements in 2D are mapped to vertex movements in 3D along the axis whose 2D projection points in the direction most closely resembling the vector of the mouse movement, as proposed by Nielson and Olsen [1]. 3D vertex movement is allowed along only one axis at a time. The net effect is that the 2D space of the mouse is partitioned into six zones corresponding to movement along the positive and negative directions of the three axes, as shown in Figure 7.

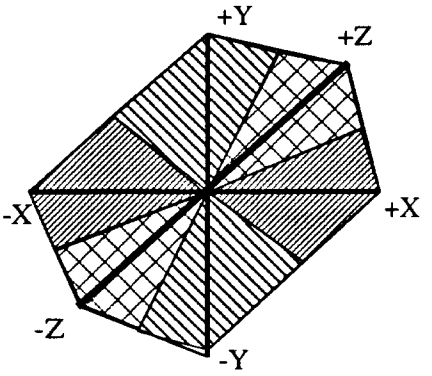


Figure 7: Mapping 2D mouse movements to 3D vertex movements

3.5 Editing UNIGRAFIX Objects

The remaining operations supported by the program are command-based operations that act on the currently selected set of objects. These commands can be invoked by the user with the keyboard or by using the *Edit menu* in the control panel, which is shown in Figure 8. The *Edit menu* currently supports commands to create, delete, collapse, flip, and subdivide the currently selected faces; create, delete, collapse and flip the currently selected wires; create, delete, collapse and subdivide the currently selected edges; and delete and merge the currently selected vertices. The remaining sections describe each of these editing operations in detail.

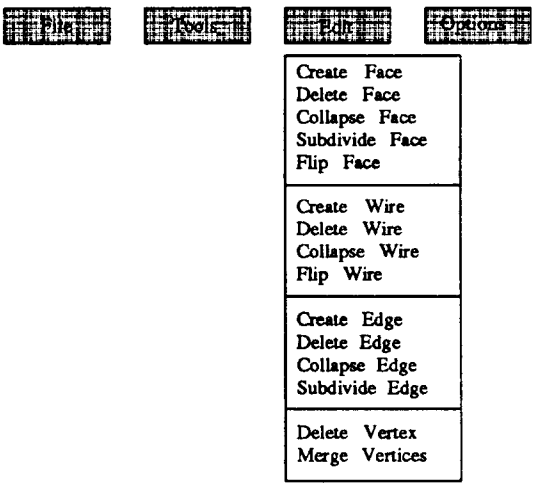


Figure 8: The edit menu

3.6 Creating UNIGRAFIX Faces and Wires

The user creates faces and wires using the *Create Face* and *Create Wire* commands. These commands construct a face or wire from the currently selected vertices, as shown in Figure 9. The order in which the vertices are arranged in the new face or wire corresponds to the order in which they were selected. When a face or wire is created, a default name and color are chosen.

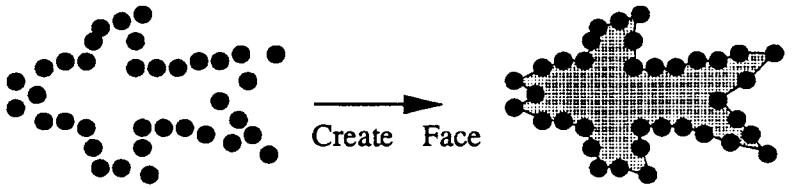


Figure 9: Creating a face

3.7 Deleting UNIGRAFIX Faces and Wires

The user deletes all the currently selected faces or wires using the *Delete Face* or *Delete Wire* commands. These commands are the converse of the *Create Face* and *Create Wire* commands. They delete only the selected faces or wires, and do not delete the attached vertices, as shown in Figure 10.

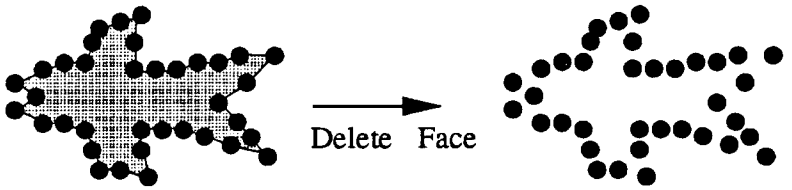


Figure 10: Deleting a face

3.8 Subdividing UNIGRAFIX Faces and Edges

The user subdivides all the currently selected faces or edges using the *Subdivide Face* or *Subdivide Edge* commands. These commands create a new vertex at the centroid of each selected face or edge, connecting this new vertex to all other vertices of the face or edge. More specifically, when a face is subdivided, a new vertex is created at the centroid of the face, the original face is deleted, and then triangular faces are constructed connecting the new vertex to each of the original face's vertices, as shown in Figure 11. The net effect is to triangulate the face, and so this command is very useful for subdividing non-planar faces. The process of subdividing an edge is similar - a new vertex is created at the midpoint of the edge, the original edge is deleted, and new edges are constructed between the new vertex and the original endpoints of the edge. The effect is to add a new vertex to the edge. This is a very useful command for adding detail to a polyhedron.

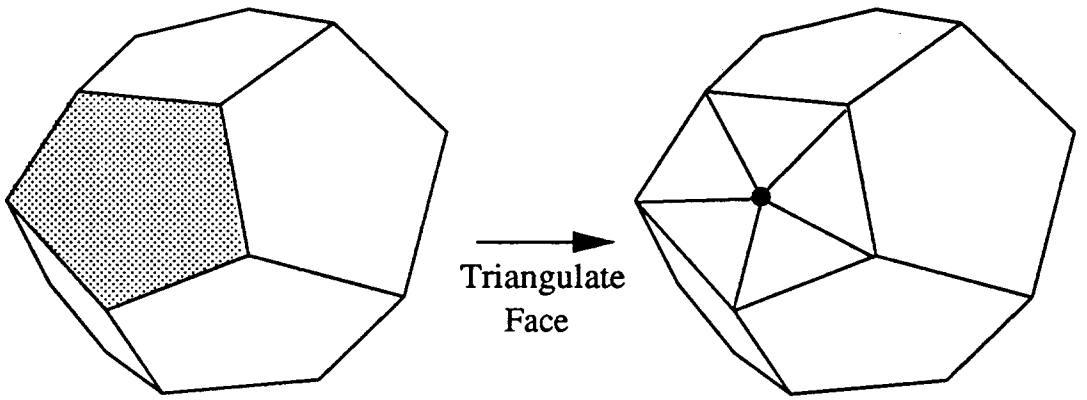


Figure 11: Subdividing a face

3.9 Collapsing UNIGRAFIX Faces, Wires and Edges

The user collapses all the currently selected faces, wires or edges using the *Collapse Face*, *Collapse Wire* or *Collapse Edge* commands. These commands create a new vertex at the centroid of each selected object, and then "collapse" all the boundary vertices of the object into this new vertex. The effect of collapsing a face is shown in Figure 12 - a new vertex is created at the centroid of the face, and then all vertices on the contours of the face are merged into this new vertex, deleting all vertices, edges and contours of the original face. Collapsing an edge is similar - a new vertex is created at the midpoint of the edge, and then the two endpoints of the edge are merged into this new vertex. This is a very useful command for removing detail from a polyhedron.

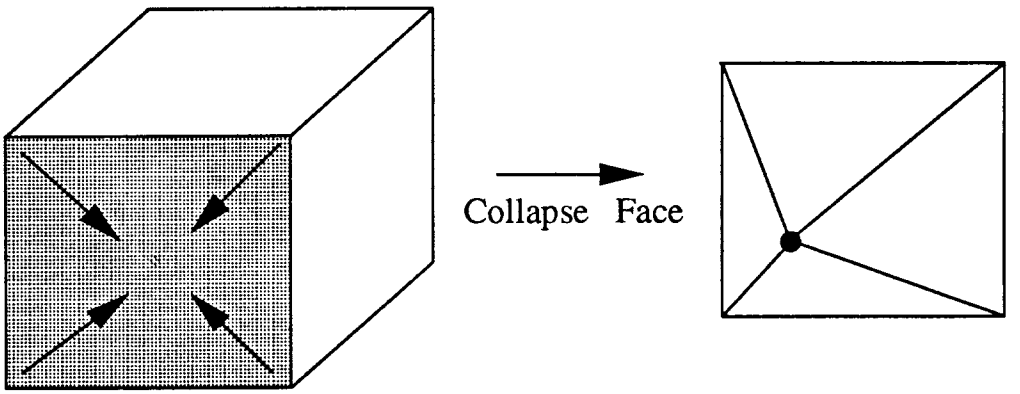


Figure 12: Collapsing a face

3.10 Flipping UNIGRAFIX Faces and Wires

The user flips the orientation of all the currently selected faces or wires using the *Flip Face* or *Flip Wire* commands. These commands reverse the order of edges in the edge lists of all contours of the face or wire. In the case of the *Flip Face* command, the direction of the face normal is reversed, as shown in Figure 13.

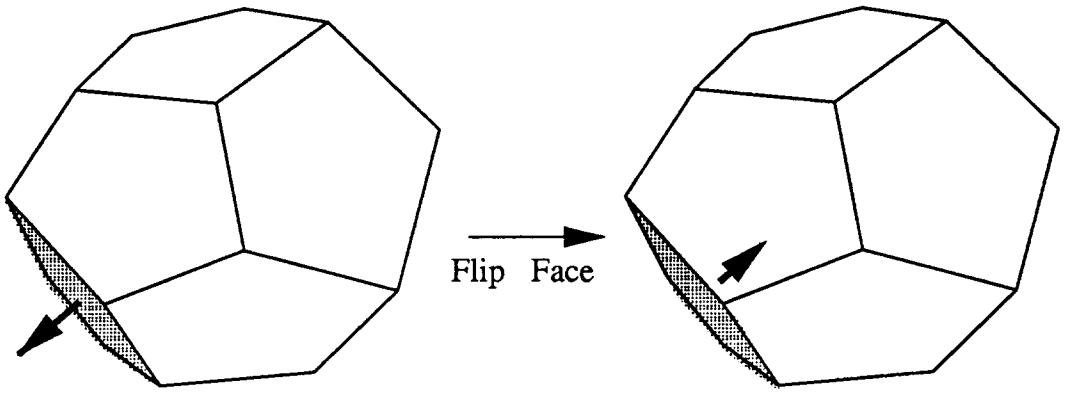


Figure 13: Flipping a face

3.11 Deleting UNIGRAFIX Vertices

The user deletes all the currently selected vertices using the *Delete Vertex* command. When a *Delete Vertex* command is executed, all edges attached to the selected vertices are deleted, then all

contours that end up with less than 3 edges are deleted, then all faces that end up with no contours are deleted, and then finally the selected vertices are deleted. An example is shown in Figure 14.

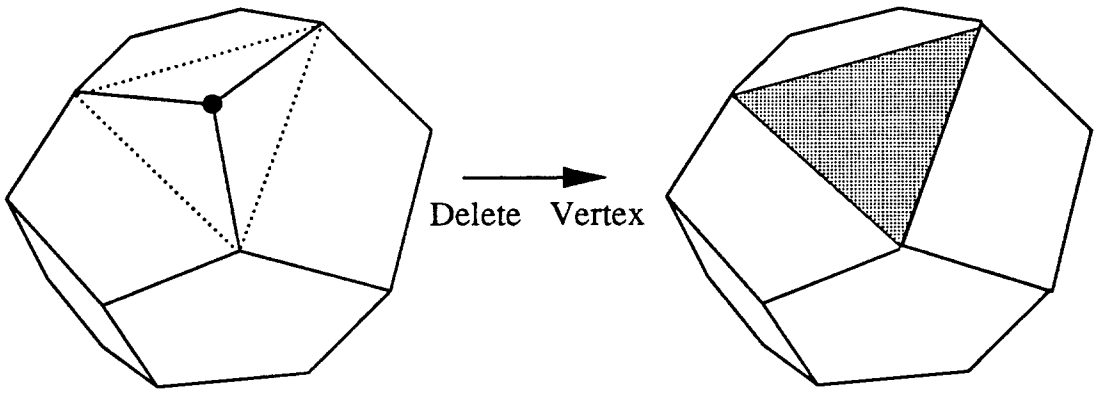


Figure 14: Deleting a vertex

3.12 Merging UNIGRAFIX Vertices

The user merges all the currently selected vertices into one using the *Merge Vertex* command. This command deletes all the selected vertices and then creates a new vertex at the position of the first selected vertex that has a topology that is the union of the topologies of all the selected vertices. The net result is that all the selected vertices are merged into one that has the position of the first selected vertex and the topology of them all. This is a very powerful command for connecting objects. An example is shown in Figure 15.

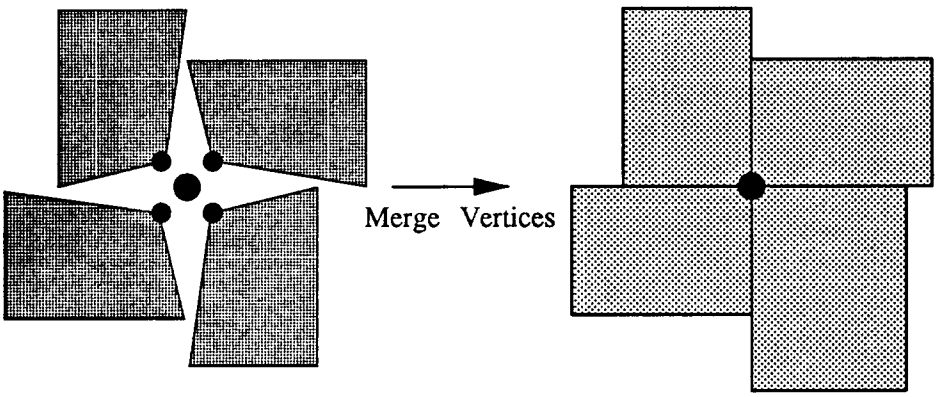


Figure 15: Merging two vertices

4 Discussion

In summary, the Animator supports basic interactive editing operations - creating, deleting and moving UNIGRAFIX objects - along with a few high-level operations that edit the object topology. It is a very simple editor, and therefore is not competitive with commercial 3D modeling systems for constructing complex, irregular polyhedra. On the other hand, it reads and writes UNIGRAFIX files directly and allows execution of UNIGRAFIX tools from within the editor, and so is very useful for creating simple or semi-regular polyhedra, and making small, local changes to existing polyhedra.

During initial experiments using the program, I found that I was able to accomplish several types of tasks quickly that would have been very tedious otherwise. For instance, it was easy to construct simple scenes by interactive creating vertices and then connecting them into faces and wires. A couple examples of this type are shown in Figures 16 and 17. Also, I was able to create semi-regular polyhedra quickly by making local changes to polyhedra constructed using the UNIGRAFIX tools. Some examples of this type are shown in Figures 18 and 19. Finally, it was fairly easy to fix-up and modify UNIGRAFIX files imported from other programs using high-level editing commands such as flip face, merge vertex, delete vertex, etc. The final example, shown in Figure 20, shows a chair (courtesy of Greg Ward) that has been fixed-up and then simplified using these commands. The original chair (in the top left) has 173 faces, while the simplified versions have 83, 22, and 10 faces respectively.

The most disappointing aspect of this project is that I found it very difficult to move a vertex to a precise 3D position using the mouse in my program. The difficulty is not due to the Nielson and Olsen approach, which I found to be quite intuitive, but rather due to inadequate 3D positional feedback during movement of the vertex. The root of the problem is that the 2D point at which the vertex appears on the screen represents a line through 3D space. Even with the axial triad cursor and bounding box feedback, it is rather difficult to tell exactly where along that line the vertex lies in 3D. Often in my experiences, after positioning a vertex, I would rotate the view only to find the vertex in a completely different position than I had imagined. Perhaps a combination of better graphical feedback, textual feedback, and constraining 3D movement can help with this problem.

5 Future Work

This project has just scratched the surface of the features possible in a UNIGRAFIX editing program. The following are some ideas for future extensions:

- Provide textual feedback about the currently selected UNIGRAFIX object(s). For instance, separate panels for each UNIGRAFIX object type could display information such as the name, color, or position of the most recently selected object of its type. Perhaps the user could edit the information in these panels - while the results of the changes are reflected in the display window.

- Allow the user to constrain 3D movement. For instance, 3D movement could be constrained to a user-settable grid or along the normal of the vertex, edge or face that is being moved. Or even better, a 3D snap-dragging approach could be used.

- Support inclusion and interactive positioning of UNIGRAFIX files within other UNIGRAFIX files. For instance, the user might add a cube defined in a UNIGRAFIX file to the currently open UNIGRAFIX file - interactively translating, scaling and rotating it such that it exactly attaches to a certain face in the current UNIGRAFIX file.

- Allow UNIGRAFIX tools to be run on only the currently selected objects. For instance, the user might run ugsphere on only a portion of a polyhedron, rather than the entire polyhedron. Perhaps all local editing commands could be implemented as UNIGRAFIX tools using such a mechanism.

6 Conclusion

This report has described the interactive editing features added to the Animator for my CS285 class project this semester. These features include creating, deleting and moving UNIGRAFIX vertices, edges, wires and faces, as well as several high-level editing operations. All these operations are invoked interactively using a combination direct-manipulation and command-based interface with real-time feedback. They co-exist with the original functionality of the Animator, viewing polyhedral objects and running UNIGRAFIX tools, resulting in a very useful and powerful interactive modeling tool. Hopefully, this program will be used by many future generations of CS285 students.

References

[1] Nielson G. M. and D. R. Olsen, "Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices," *Proceedings 1986 Workshop on Interactive 3D Graphics,* ACM Press, October, 1986, p175 - 182.

[2] Séquin C. H. and K. P. Smith, *Introduction to the Berkeley UNIGRAFIX Tools,* Report No. UCB/CSD 90/606, Computer Science Division (EECS), University of California, Berkeley, California, November, 1990.

[3] K. P. Smith. *Interactive Modeling Tool,* Unpublished, 1990.

7 Examples

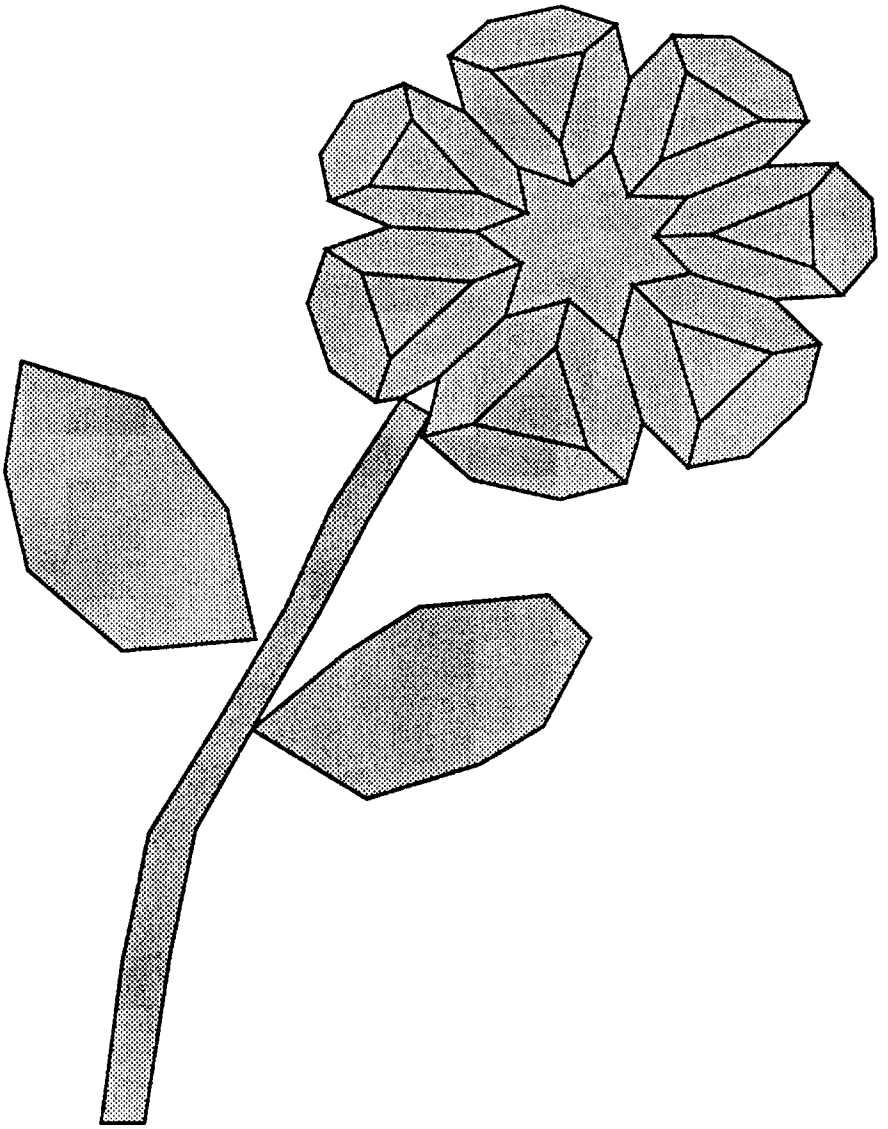


Figure 16: A 2D flower

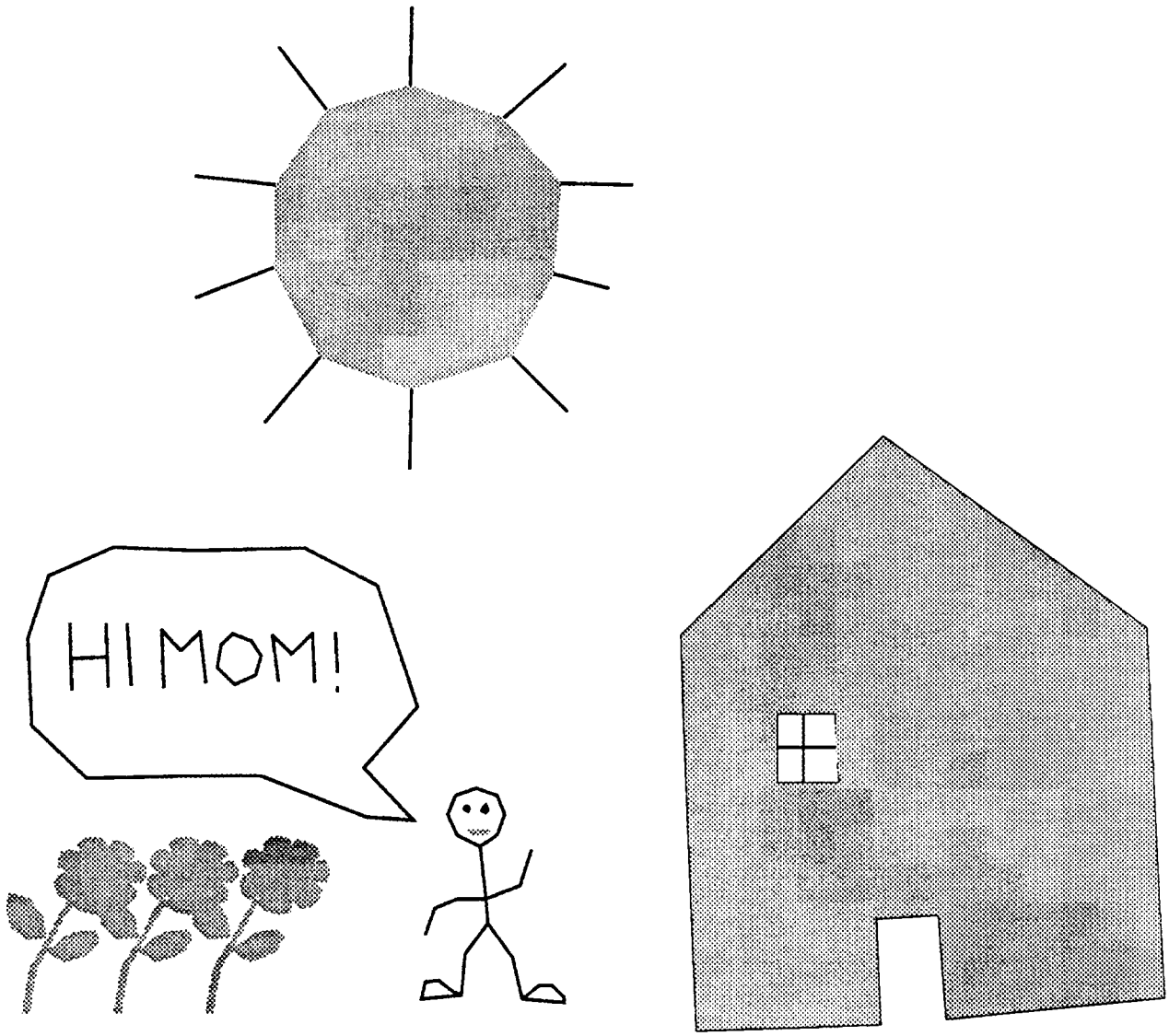


Figure 17: A 2D scene

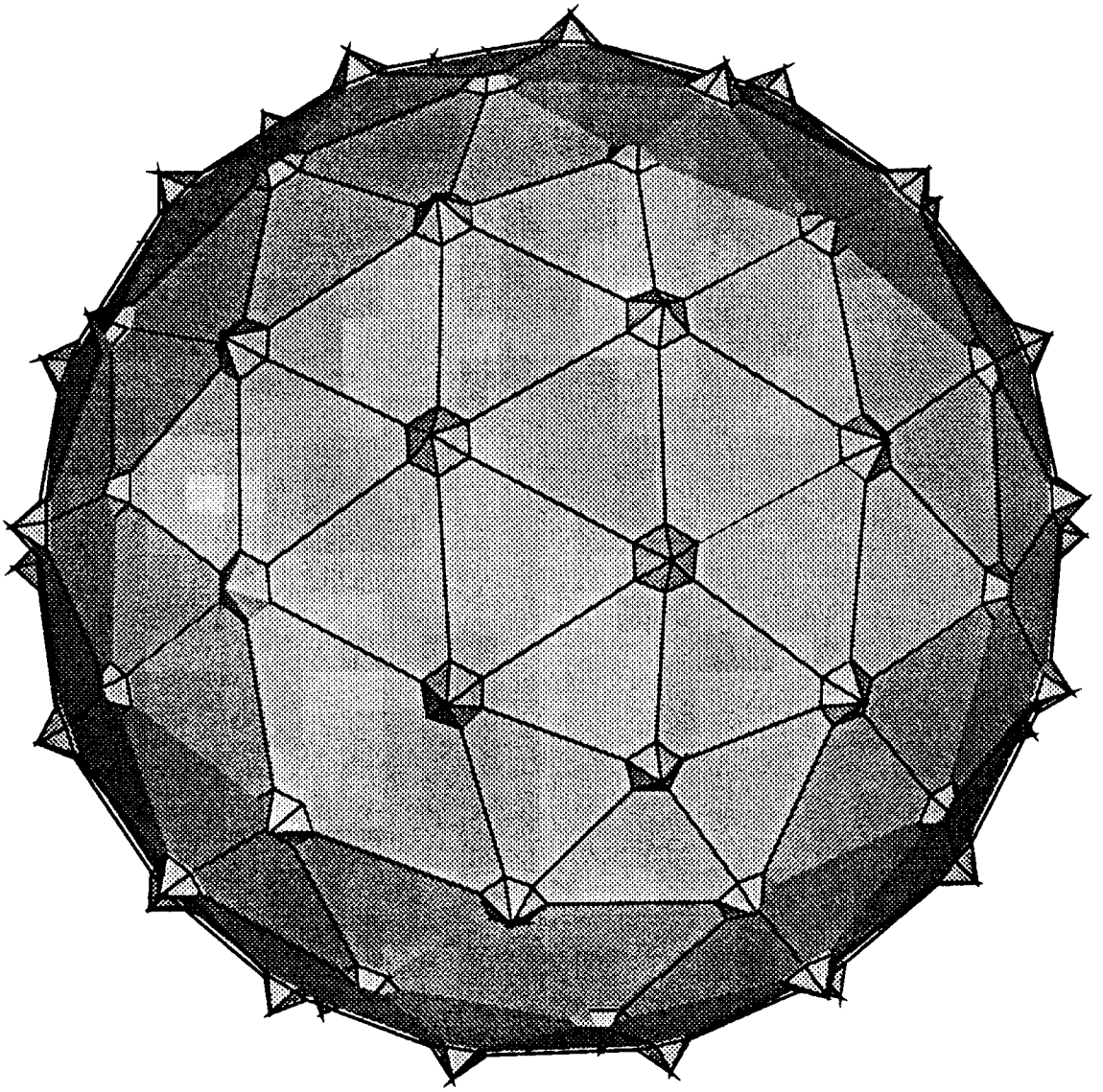


Figure 18: A Semi-Regular Polyhedron

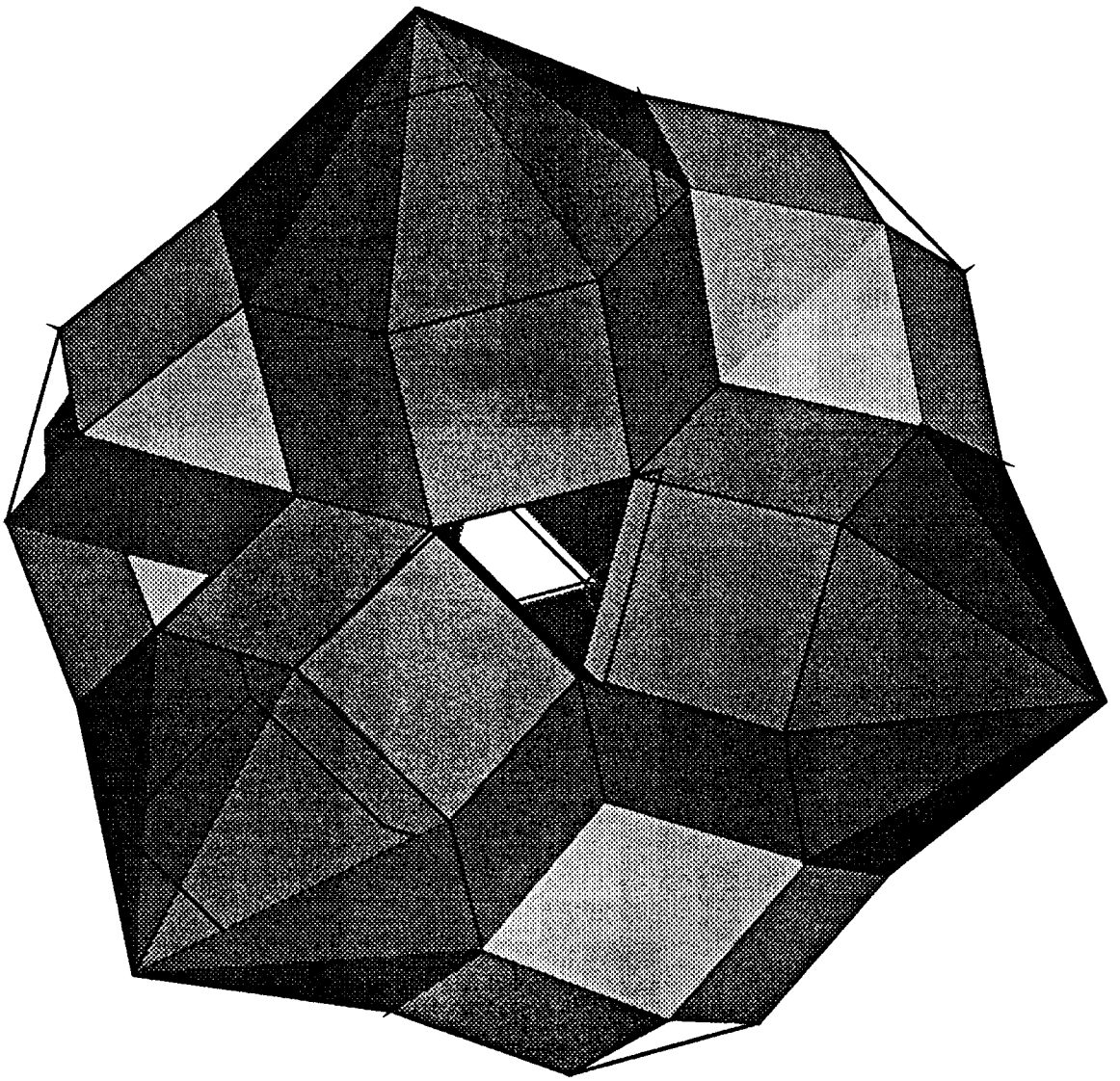


Figure 19: A Toroid of Genus Seven

Figure 20: Simplified Versions of a Chair

NAME

animator – an interactive viewing, editing and modeling tool

SYNOPSIS

animator

DESCRIPTION

The *animator* is an application that allows the user to interactively view, edit and run modeling tools on UNIGRAFIX files. These man pages describe the new interactive editing features of the program.

CREATING UNIGRAFIX FILES

The original version of the Animator has a pull-down menu interface for loading and saving UNIGRAFIX files. Using this pull-down menu, the user can open any existing UNIGRAFIX file, or save the current polyhedron in a file. The Animator now has an additional file manipulation command on this pull-down menu that allows the user to open a new, empty UNIGRAFIX file. Using this command, the user can create a new UNIGRAFIX file from scratch, rather than having to derive it from another file. In fact, when the Animator starts, an empty UNIGRAFIX file is opened by default. At this point, the user may either begin adding UNIGRAFIX objects to this default file or load another file.

CREATING UNIGRAFIX VERTICES

The Animator allows a user to create new vertices in the currently open UNIGRAFIX file. The user can create a new vertex at the current 2D cursor position by double-clicking with the left mouse button. Since the 2D cursor position represents a line through 3D, the actual position of the new vertex is dependent on what type of object lies under the cursor position at the time of the command. If the interior of a face lies under the cursor position, then a new vertex is created positioned at the intersection of the 3D line representing the 2D cursor position and the plane of the underlying face. The new vertex is connected to the other vertices of the face by constructing new edges and contours that tessellate the face. On the other hand, if an edge lies under the cursor position, then a new vertex is created positioned at the intersection of the 3D line representing the 2D cursor position and the 3D line along the edge. In this case, the underlying edge is split into two new edges that connect its endpoints through the new vertex. Otherwise, if no face or edge lies under the cursor position, a new vertex is created that is not attached to any other object in the file. The position of the new vertex is set to be the intersection of the 3D line representing the 2D cursor position and the axial plane (XY, XZ or YZ) whose normal vector is directed most toward the viewer. Using these commands, the user can add any number of vertices to a UNIGRAFIX file.

SELECTING UNIGRAFIX OBJECTS

The user interface for interactively editing polyhedra is based on a *currently selected set* of UNIGRAFIX *objects* (vertices, edges, wires and/or faces). All objects in the currently selected set are drawn in a special color (yellow) and all local editing operations are applied only to these objects. The user replaces the previously selected objects with a new object by simply clicking on the object with the left mouse button. To extend the current selection (i.e. add an object to the currently selected set without replacing the previously selected objects), the user must hold down the shift key when clicking on the object with the left mouse button.

MOVING UNIGRAFIX VERTICES

The user can move the currently selected set of vertices by moving the mouse with the right mouse button down. When the right mouse button is pushed down, an axial triad cursor appears whose origin corresponds with the first vertex in the currently selected set (the triad is intended to help the user visualize the 3D movement). Then, while the right mouse button is held down, each movement of the mouse in 2D results in a movement of the all the currently selected vertices in 3D. When the right mouse button is released, the triad disappears and the selected objects keep their new positions.

Mouse movements in 2D are mapped to vertex movements in 3D along the axis whose 2D projection points in the direction most closely resembling the vector of the mouse movement, as proposed by Nielson and Olsen during the 1986 Workshop on Interactive 3D Graphics in Chapel Hill. 3D vertex movement is allowed along only one axis at a time. The net effect is that the 2D space of the mouse is partitioned into six zones corresponding to movement along the positive and negative directions of the three axes.

EDITING UNIGRAFIX OBJECTS

The remaining operations supported by the program are command-based operations that act on the currently selected set of objects. These commands can be invoked by the user with the keyboard or by using the *Edit menu* in the control panel. The Edit menu currently supports commands to create, delete, collapse, flip, and subdivide the currently selected faces; create, delete, collapse and flip the currently selected wires; create, delete, collapse and subdivide the currently selected edges; and delete and merge the currently selected vertices. The remaining sections describe each of these editing operations in detail.

CREATING UNIGRAFIX FACES AND WIRES

The user creates faces and wires using the *Create Face* and *Create Wire* commands. These commands construct a face or wire from the currently selected vertices. The order in which the vertices are arranged in the new face or wire corresponds to the order in which they were selected. When a face or wire is created, a default name and color are chosen.

DELETING UNIGRAFIX FACES AND WIRES

The user deletes all the currently selected faces or wires using the *Delete Face* or *Delete Wire* commands. These commands are the converse of the *Create Face* and *Create Wire* commands. They delete only the selected faces or wires, and do not delete the attached vertices.

SUBDIVIDING UNIGRAFIX FACES AND EDGES

The user subdivides all the currently selected faces or edges using the *Subdivide Face* or *Subdivide Edge* commands. These commands create a new vertex at the centroid of each selected face or edge, connecting this new vertex to all other vertices of the face or edge. More specifically, when a face is subdivided, a new vertex is created at the centroid of the face, the original face is deleted, and then triangular faces are constructed connecting the new vertex to each of the original face's vertices. The net effect is to triangulate the face, and so this command is very useful for subdividing non-planar faces. The process of subdividing an edge is similar - a new vertex is created at the midpoint of the edge, the original edge is deleted, and new edges are constructed between the new vertex and the original endpoints of the edge. The effect is to add a new vertex to the edge. This is a very useful command for adding detail to a polyhedron.

COLLAPSING UNIGRAFIX FACES, WIRES AND EDGES

The user collapses all the currently selected faces, wires or edges using the *Collapse Face* , *Collapse Wire* or *Collapse Edge* commands. These commands create a new vertex at the centroid of each selected object, and then "collapse" all the boundary vertices of the object into this new vertex. The effect of collapsing a face is that a new vertex is created at the centroid of the face, and then all vertices on the contours of the face are merged into this new vertex, deleting all vertices, edges and contours of the original face. Collapsing an edge is similar - a new vertex is created at the midpoint of the edge, and then the two endpoints of the edge are merged into this new vertex. This is a very useful command for removing detail from a polyhedron.

FLIPPING UNIGRAFIX FACES AND WIRES

The user flips the orientation of all the currently selected faces or wires using the *Flip Face* or *Flip Wire* commands. These commands reverse the order of edges in the edge lists of all contours of the face or wire. In the case of the *Flip Face* command, the direction of the face normal is reversed.

DELETING UNIGRAFIX VERTICES

The user deletes all the currently selected vertices using the *Delete Vertex* command. When a *Delete Vertex* command is executed, all edges attached to the selected vertices are deleted, then all contours that end up with less than 3 edges are deleted, then all faces that end up with no contours are deleted, and then finally the selected vertices are deleted.

MERGING UNIGRAFIX VERTICES

The user merges all the currently selected vertices into one using the *Merge Vertex* command. This command deletes all the selected vertices and then creates a new vertex at the position of the first selected vertex that has a topology that is the union of the topologies of all the selected vertices. The net result is that all the selected vertices are merged into one that has the position of the first selected vertex and the topology of them all. This is a very powerful command for connecting objects.

FILES

~ug/ug3/src/bin/animator/*

SEE ALSO

ugiris (UG)

BUGS

Yet to be reported.

AUTHOR

Tom Funkhouser

The UGCLAY System

Mark Halstead

1. Introduction

This report describes the *ugclay* extensions to the *animator* program. It concentrates mainly on *ugclay* concepts and implementation; for more information on using the system refer to the relevant manual pages.

The original *animator* program provided an environment in which the user could view and modify existing Unigraphix objects. The current Unigraphix object was displayed in a window and this view could be rotated, scaled and translated. To modify an object, the user could select from a variety of tools which performed such operations as truncation, beveling and tesselation, either on the current object or Unigraphix definitions contained in files. The result could be displayed and "animated" by changing a single degree of freedom in the operation with a slider.

Extensions to the *animator* have been developed which allow the user to interactively edit the current Unigraphix object (refer to Tom Funkhouser's report). Vertices, edges and faces may be created, deleted and moved. To move vertices, the user first points and clicks with the mouse to select a set of vertices. These are then dragged to new positions in 3D space (keeping the same relative positions within the set). Only the selected vertices are dragged, resulting in the elongation of some edges and the stretching of some faces.

The idea for *ugclay* arose from the experience of constructing real objects in clay. The natural qualities of clay – its softness and reshapeability – make it an ideal medium for modeling solid objects. It would be nice if we could imitate these qualities within the framework of a Unigraphix object editor. This is the aim of the *ugclay* system.

2. Concepts

One observation that can be made of clay models is that forces applied to the surface – for example pushing or pinching – result in a non-local change in the object. In other words a push may result in a bend or squash over an area larger than the area of contact. Translating this into the context of Unigraphix model editing we realize that dragging a set of vertices should somehow affect those vertices which are not part of the dragged set. The job of *ugclay* then is to move these vertices in such a way that the user feels as though they are manipulating a piece of clay.

In fact the *ugclay* system is general enough to model substances with a variety properties, such as cloth or jello!

3. Approach

The interface between the object editor and *ugclay* is simple. As a set of vertices is dragged *ugclay* is provided with a list of those vertices and an offset vector in 3-space representing the latest drag increment to be applied to the vertices. *Ugclay* decides on the basis of the current *movement model* and user controlled settings how this change will affect every vertex in the model.

Two movement models are provided.

4. Movement Models

4.1 Drag mode

4.1.1 Description

This is the simplest mode available – it basically consists of adding some fraction of the offset vector $\tilde{v}$ to each vertex. Two effects are simulated: *translate* and *squash*.

Translate: each vertex in the drag set is moved by the offset vector $\tilde{v}$. Each vertex in the "fringe" of vertices connected by an edge to a vertex in the drag set is moved by an offset $f\tilde{v}$ where f is the translate factor $0 \leq f \leq 1$. Each vertex in the fringe connected by two vertices is moved by $f^2\tilde{v}$ and so on.

Squash: each vertex in the drag set is moved by the offset vector $\tilde{v}$. Each vertex p in the "one edge fringe" is shifted by a vector $\tilde{w}$ where $\mid \tilde{w} \mid = f \mid \tilde{v} \mid$, f is the squash factor and $0 \leq f \leq 1$. The vector $\tilde{w}$ is the (uniformly) weighted average of the vectors $\vec{pq}$ where q is a vertex of the initial drag set. For vertices in the "two edge fringe" $\mid \tilde{w} \mid = f^2 \mid \tilde{v} \mid$ etc.

4.1.2 Motivation

The motivation behind the drag mode effects is simple: dragging on a few vertices causes the other vertices to move some amount in the same direction, in other words the object is stretched. This basic effect is provided by translation. The movement will also tend to have some component in the direction of the vertices being dragged, an effect provided by squash.

Depending on the properties of the object, vertices close to those being dragged move more than those further away, due of course to the stretching of the intervening material. The translate and squash factors control this property.

With the correct settings of the translate and squash factors some quite realistic looking effects can be generated. Low factor settings give the feel of a soft material where changes are very local; high factors cause a widespread stretch. Setting the translate factor to 1 and the squash factor to 0 allows the user to drag an object around without changing its shape. Setting both factors to 0 causes only the vertices in the initial drag set to move.

High settings of the translate factor have the effect of both stretching *and* moving the object. If only stretching is required the user is given the option of pinning the object. In this case the vertices furthest (in terms of number of edges) from the drag set of vertices remain fixed in space and only stretch effects are simulated.

It would of course be possible to develop other effects based on a similar approach. For example one could take into account the directions of the edges at a vertex and allow more general pinning. However the aim of the drag mode was to provide the simplest possible movement model. The more complex *spring mode* has these abilities.

4.1.3 Implementation

To implement the drag mode it is necessary to perform a breadth first search of the Unigraphix object, beginning with the vertices in the initial drag set. This is accomplished by maintaining a queue of vertices; we repeatedly remove a vertex from the head of the queue, update its position by the appropriate vector and add those as yet unvisited neighbours to the end of the queue. The neighbours of a vertex are found by iterating over the incident edges which are easily accessed through the Unigraphix statement structure.

To aid the search each vertex is assigned a *visit number* as it is added to the queue. This helps to identify those vertices which have already been visited and to identify where in the queue a new fringe level begins (each successive fringe has a higher visit number assigned). This visit number must be stored with the vertex; to allow this each vertex has attached to it an *extra* structure pointed to by the Unigraphix statement extra pointer. This extra structure also contains additional information pertaining to the spring mode.

As the breadth first search proceeds the current translate vector and drag vector length are maintained. As each new fringe of vertices is encountered this translate vector and drag length are scaled by the appropriate factors. As the last fringe is processed it is possible that a nonzero translate vector is being added, which implies that the object is translated by this much as well as stretched. If the object is pinned then it is necessary to go back and subtract this amount from each vertex to ensure that only stretch effects are produced (in other words the last fringe of vertices remains fixed in space).

4.2 Spring Mode

4.2.1 Description

Spring mode is more complicated than drag mode and thus provides a wider range of interesting effects. It is not, however, able to imitate all the effects of drag mode and so the two complement each other in utility.

The idea behind spring mode is to model *elastic* materials. The materials of drag mode are inelastic – once stretched they remain in that state. Materials in spring mode on the other hand have a natural tendency to return to their original shape.

This behaviour is accomplished by treating each vertex as a point mass and each edge as a damped viscoelastic spring unit. Thus a Unigraphix object naturally defines an arrangement of point masses connected by springs.

Briefly, the theory of spring elasticity is as follows. A spring has a natural rest length x_0. If it is stretched or compressed to a length x the restoring force generated by the spring is:

$$\vec{F} = k(x - x_0)$$

where $\vec{F}$ is a force vector lying along the line of the spring and k is the spring factor.

An arrangement of point masses and springs is in a state of equilibrium (rest) when the force vectors

acting on each mass sum to zero. If the sum is nonzero the mass is accelerated according to the formula:

$$\tilde{F} = m\tilde{a}$$

where m is the mass and $\tilde{a}$ is the acceleration vector lying in the same direction as $\tilde{F}$.

Therefore, for a given mass/spring system we can determine whether equilibrium has been reached by summing the forces acting on each mass. These forces are calculated by comparing the extension of each spring to its rest length. If the system is not in equilibrium the masses can be accelerated. In fact this is exactly how *ugclay* works. It repeatedly performs this operation until the vertices move into a state of equilibrium.

Before this process is described further there are a number of details that require dicussion. First of all there must be a method by which the user controls the shape of the object. By necessity this involves changing the state of equilibrium. The method employed is the *pinning*, *unpinning* and *dragging* of vertices. A pinned vertex maintains a fixed position in space. The user is able to select vertices to be pinned and unpinned and can drag pinned vertices around.

Depending on the requirements of the user the unpinned vertices can be either completely free or *restrained*. This restrainment is provided in terms of springs connecting each unpinned vertex to a fixed point in space. This fixed point is usually the initial position of the vertex when the user began editing. A method is provided for relocating the fixed point to the current vertex location. These restraining springs are very useful as they provide a means of maintaining a solid's general shape. Without them a solid would be able to collapse inwards.

Another fact to note is that the springs used are not simply elastic. They also have a *damping* ability, which means that the spring generates a force proportional to the rate of change of extension in a direction that will resist this change. By controlling the constant of proportionality the user can affect how quickly the system will reach its equilibrium point after vertices are dragged. That is, it will control the oscillations inherent in non-damped systems.

Of more importance is the *viscoelastic* property of the springs. This is the ability to stretch (increase or decrease rest length) with a rate proportional to the amount of extension. This is important because it models real materials where an applied force causes extension *and* stretching.

The above properties are easily incorporated into the general method for finding the equilibrium state and yet add greatly to the functionality of the system. Two additional effects are also taken into account: *gravitational acceleration* and *medium damping*. Gravitational acceleration is simply an acceleration vector acting on each mass. It can either be directed along the negative y-axis in object space coordinates or in a direction which appears to be "vertically downwards" on the screen (and so changes with respect to the object coordinates after each rotation). Medium damping is an effect which models the motion of the masses through a viscous medium such as water. It generates an additional force vector on each mass which acts in a direction opposite to and in proportion to the velocity of the mass in order to slow its motion.

It should be noted that *ugclay* provides the user with the means to change all the constants of

proportionality mentioned above along with the amount of gravitational acceleration. By varying these properties the user can simulate a range of different materials, from clay through to cloth. To find the "right" settings requires a certain amount of experimentation.

After stretching an object around until it is in the required shape the user may "freeze" the model. This unpins all vertices. sets all spring rest lengths to the current edge lengths and moves the fixed points of each restraining spring to lie on the corresponding vertex. In other words the model is put into a stable unpinned state. This model may then be saved as new Unigraphix object description or modified using a tool or drag mode.

4.2.2 Implementation

The implementation of spring mode is conceptually straightforward. As mentioned previously each vertex has an extra structure attached. For use in spring mode this extra structure stores:

- a flag to indicate whether the vertex is pinned
- the current velocity of the vertex
- the position of the fixed end of the restraining spring (if the vertex is not pinned)
- the rest length of the restraining spring

In addition, each edge has an attached extra structure containing the rest length of the spring represented by the edge.

The iterative solution process to find the equilibrium position is carried out as follows. The animator contains a main event loop where it waits for input events generated by the user. Each time through the loop a procedure is called to update the positions of all vertices. Assume that the time between calls is Δt. Then in pseudocode this update procedure is:

```
for each vertex stmt do
    add vertex velocity.Δt to vertex position
    force_so_far = 0
    if vertex is restrained
        calculate spring extension
        add resulting force to force_so_far
    end
    add force due to medium damping to force_so_far
end
for each edge
    calculate edge extension and resultant force
```

```
        calculate edge damping force
        add forces to force_so_far of incident vertices
        adjust rest length of spring according to acting forces
end
for each vertex
        divide force_so_far by mass to get acceleration
        add gravitational acceleration
        add acceleration.Δt to vertex velocity
end
```

This scheme is essentially an explicit Euler method for solving in discrete time steps a set of first order differential equations. It does not require a matrix approach as each vertex has a known small number of attached edges. An implicit method such as Runge-Kutta could have been implemented but in general this is not necessary as we are dealing with highly damped systems where the vertex velocities are small. We are more interested in the equilibrium state than the motion of the system. There is a problem sometimes with the system oscillating wildly but this can usually be remedied by reducing Δt. Hence the user is provided with the ability to control this value.

5. Controls

5.1 Overview

Throughout the previous sections it has been mentioned that the user has the ability to change a number of factors and constants of proportionality to control the *ugclay* mode of operation. The user is given this control by means of the *clay control panel.*

The *animator* program itself includes facilities for toggle buttons in its control panel. However *ugclay* requires extensive use of sliders in addition to a variety of buttons. Thus the decision was made to use an available panel library and editor to construct the clay control panel. The drawback of this approach is that the user interface now consists of two different styles. The best way to resolve this problem would be to reimplement the original *animator* controls in the new panel library. This has not been done as integration with the object editing code (using the old interface) was only possible at the end of the project.

The operation of the clay control panel is fairly self-explanatory. It involve setting toggle switches and changing sliders to control various functions. Refer to the *ugclay* manual pages for further information.

5.2 Implementation

Using the panel library was not as simple as it was hoped. Theoretically the only requirement was to include a call to the panel code within the main event loop. In practice however the panel code wreaked havoc with the event queue operation, as it would consume important events required

for the correct operation of the *animator*. Fixing this satisfactorily required changes to the panel library. The fixes made are satisfactory for the current application but in the long term, if the panel library is to be useful, extensive changes will be required within the panel code to correctly handle the event queue. Specifically, it will be necessary for the user to read events and pass these to the panel code rather than let the panel code read from the queue itself.

6. Interface Issues

The program interface between the *ugclay* system and the *animator* is kept as simple as possible. Only five calls are necessary to implement the major functions:

- pin a set of selected vertices
- unpin a set of selected vertices
- drag a set of vetices by a given amount
- solve for the spring forces and vertex movements
- handle the clay control panel

All other calls are necessary only to:

- initialize the *ugclay* system
- allocate/free the extra structures attached to vertices/edges when these statements are created or deleted

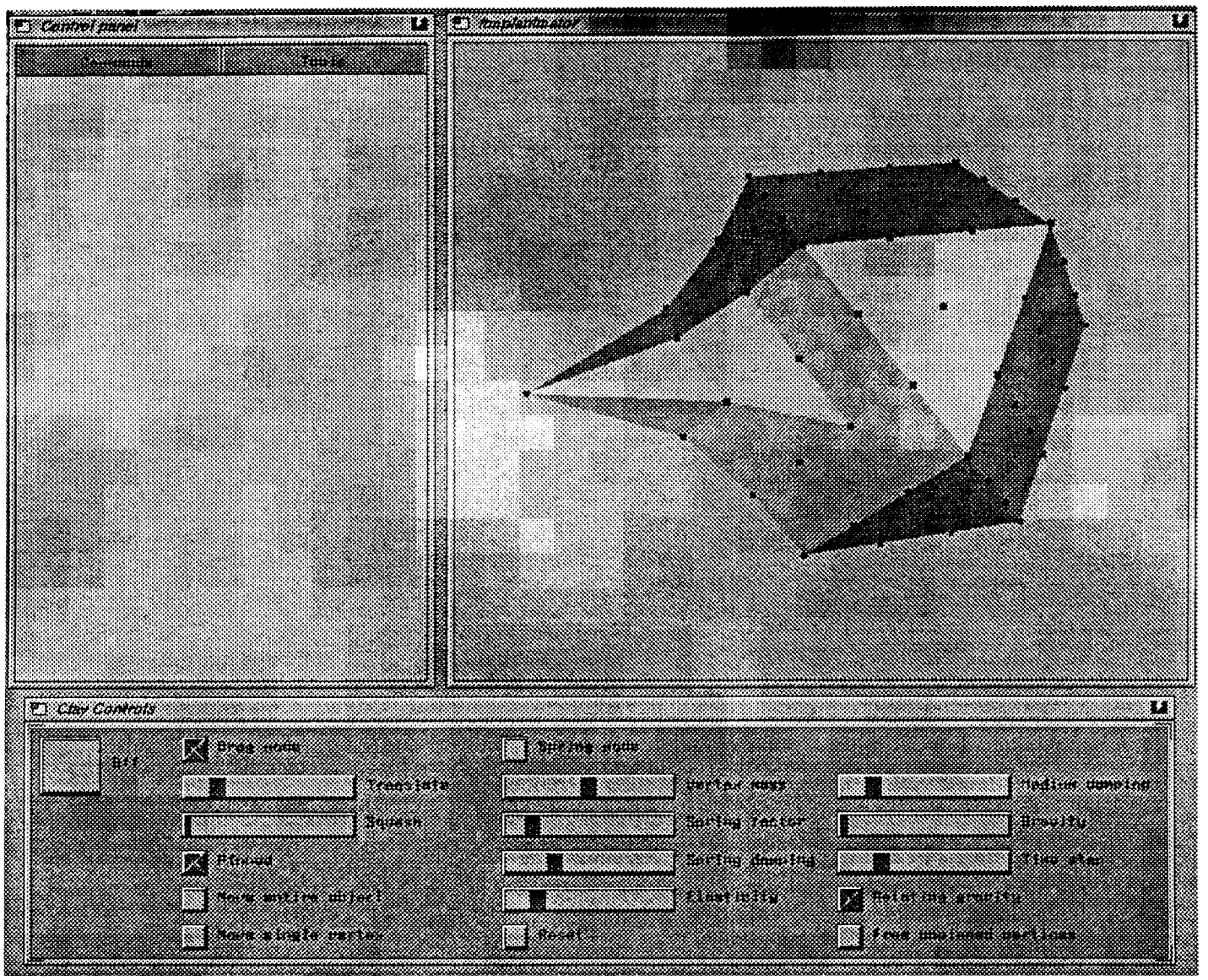
Ensuring that *ugclay* continued to work correctly as faces were collapsed and subdivided required some effort.

7. Summary

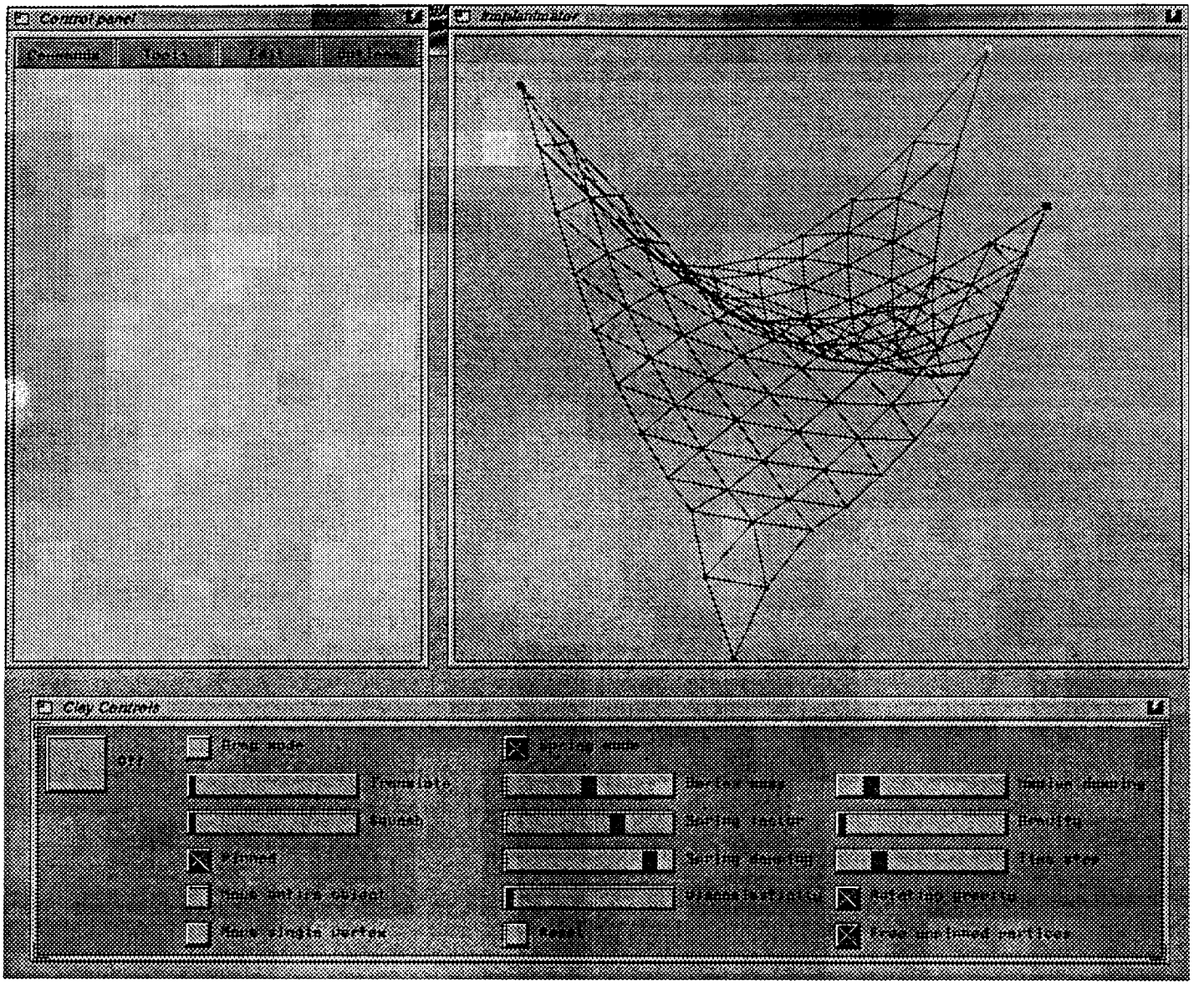
This report has described the motivation behind *ugclay* and its implementation. Overall the project has been quite successful in its attempt to provide a more useful object editing environment. It has also provided an interesting environment for animation – it is possibly more fun to watch objects assume their rest positions than it is to mould them!

Of course there are always a number of improvements which could be made to make the system more useable. These fall outside the bounds of what was possible within the limited time available, especially given the constraint that we were working with Unigraphix models. It would be interesting to look further into ways of providing a more realistic "claylike" environment.

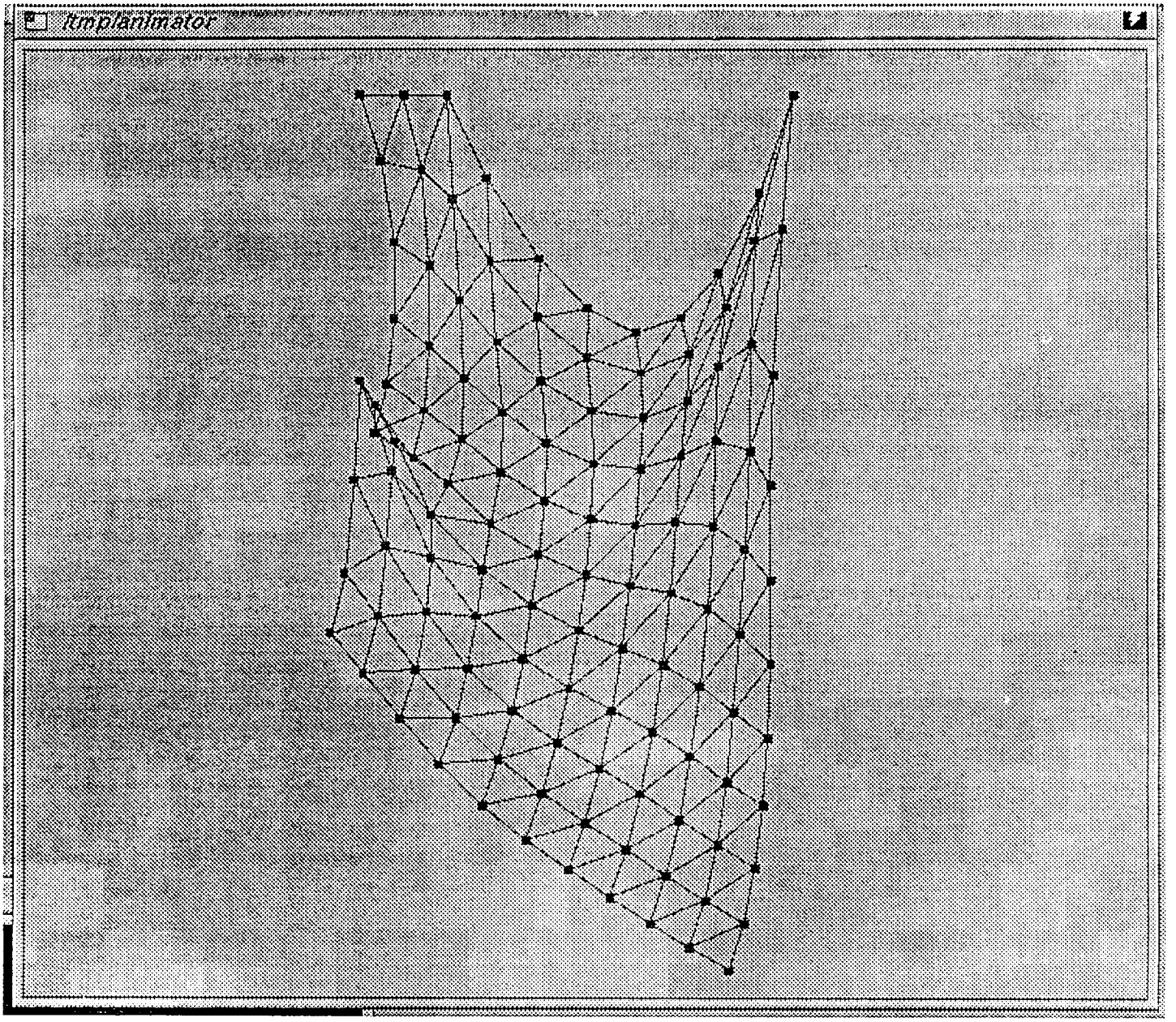
On the following pages are some example illustrations from *ugclay*. They are taken directly from the screen during a *ugclay* session and hopefully demonstrate some of the utility of the system.



Stretching a tessellated cuboctahedron in drag mode



A "suspended net" in spring mode



Another suspended net showing more pinned vertices

DESCRIPTION

This manual describes the *ugclay* additions to the *animator* program. These additions allow the user to interactively manipulate the currently displayed object by moving vertices.

Ugclay has been integrated with object editing extensions to the *animator* which allow the creation, deletion and movement of faces, wires and vertices. Most important to the *ugclay* user is the ability to select a set of vertices which can be dragged in three dimensions. As the user does so, *ugclay* attempts to aid the user by deforming the entire object as if it were made of a stretchy substance such as clay or jello. This is done by moving each vertex in the object according to: (i) the current movement mode selected by the user, (ii) various user definable parameters and (iii) the initial drag given to the set of vertices.

The original functions of the *animator* system have been retained.

OPERATION

The *ugclay* system is controled via the clay control panel which appears in the lower half of the screen. This panel contains buttons and sliders which define the current mode of operation.

The *On* button in the top left corner of the panel switches *ugclay* on and off. In the *off* position ugclay will not affect the operation of the *animator* and objects may be edited in the normal way. In particular, dragging a set of vertices will not invoke the *ugclay* module and only local changes to the set of vertices itself will occur.

Object Editing

The user should consult documentation on the *animator* program and its editor extensions for information on loading objects and the selection of sets of vertices. Two functions which pertain directly to *ugclay* are provided in the *animator* control panel. To pin the currently selected set of vertices select the *Pin* option from the edit menu. Similarly to unpin a set of vertices select the *Unpin* option. The uses of pinned vertices will be discussed in a later section. Pinned vertices appear as blue squares in the object display.

If *ugclay* is turned on and the user drags a set of selected vertices then all the vertices in the object will be shifted according the current movement mode.

Movement Modes

Two movement modes are provided in the *ugclay* system. Select one by pushing the appropriate mode button on the clay control panel.

Drag Mode

This is the simplest mode. The new position of each vertex depends on: (i) the change in position of the dragged vertex (*translate*) and (ii) the new relative position of the dragged vertex (*squash*).

Translate

The change of position vector v of the dragged vertex is calculated. A fraction f of v is added to the position of each vertex connected by a single edge to the dragged vertex. A fraction $f*f$ of v is added to each vertex two edges away from the dragged vertex and so on. The fraction f may vary between 0 and 1 and is controled by the setting of the translate slider in the clay control panel. This factor controls the amount of "stretch" applied to the object.

For high translate factors the vertices furthest from the dragged vertex (in terms of number of edges) may shift. In other words the object is moved as well as stretched. If the user pushes the toggle button marked *Pinned*, the furthest vertex (or vertices) will remain pinned in space and only stretching will be observed.

Squash

The length l of the vector v is calculated. For each vertex one edge away from the dragged vertex a vector w of length $f*l$ which points towards the dragged vertex is added to vertex position. For vertices two edges away the length of w is $f*f*l$ and so on. The squash factor f is determined by setting the squash slider on the control panel.

Both translate and squash effects may be applied simultaneously. The translate and squash factors can be used to vary the response of the object to vertex movements. Finding the desired settings will require experimentation. To help the user, two frequently used settings have been provided with control panel buttons. Push the required button to establish the settings. The choices are:

Move entire object The translate factor is set to 1, the squash factor to 0 and the *Pinned* option is turned off. The result is that dragging one vertex drags all connected vertices (usually the entire object).

Move single vertex The translate and squash factors are set to 0. Only the dragged vertex will move.

Spring Mode

In drag mode the user should feel as though they are constructing an object out of clay. That is, once the model has been deformed it retains its new shape. In contrast, spring mode allows the user to model objects out of a "rubbery" substance such as jello. The objects in this mode have a natural tendency to rebound into their original shape.

This is accomplished by setting up a mass/spring system in the object. Each vertex is treated as a point mass and each edge as a spring unit with damping. Each vertex may be either (i) *pinned* (its position in space is fixed) (ii) *free* (the vertex is free to move anywhere) or (iii) *restrained* (the vertex is connected by a spring to its original position).

As mentioned previously a set of vertices is pinned or unpinned by selecting the appropriate option from the edit menu. Vertices which are dragged by the user are automatically pinned. The *Free unpinned vertices* button on the clay control panel indicates whether vertices which are unpinned are free or restrained.

The shape of the object being edited is determined by balancing the spring forces acting on each vertex to find an equilibrium position. This shape can be changed by dragging and pinning vertices.

A number of other parameters also affect the equilibrium point of the system: the *mass* of the vertices, the *spring constant* of the springs, the value of *gravity* and the *rest length* of the springs. The first three parameters are controllable by setting sliders on the control panel. The rest length of the springs is set equal to the current edge length when either spring mode is selected or the *reset* button on the control panel is pressed. In other words this reset state will be an equilibrium point as long as the current gravity factor is zero.

The rest length of the springs can also be changed by setting a non-zero factor on the *Viscoelasticity* slider. This factor controls the *stretching* of the springs over time. Any spring that is not at its rest length will stretch by the given viscoelasticity factor (between *0* and *1*) at each time step (see below).

To determine the equilibrium point of the object an iterative solution process is used. The intermediate steps of this solution process are displayed and so the user is presented with an animation of the object bouncing around until it settles into its new shape. To control this solution process three sliders are provided: the *damping factor* sliders control how quickly the vertex motion will subside, the *time step* slider controls the length of the time steps taken in the solution process. Note that if damping is set to zero the inaccuracies of the solution method may make the object oscillate wildly! Pressing the reset button should freeze the solution as it is currently displayed. This action also sets the gravity slider value to zero (otherwise the unpinned object would fall off the screen!)

Note that pressing the reset button changes the rest lengths of the springs *and* unpins all vertices.

Note also that you should pin some vertices before turning up the gravity so that the object will be suspended by some points.

SUMMARY

A brief summary of the pertinent controls follows.

On the edit menu:

Pin: the currently selected vertices are pinned

Unpin: the currently selected vertices are unpinned

Buttons on the clay control panel:

On/Off: turns the *ugclay* module on and off

Drag mode: selects the drag movement mode

Spring mode: selects the spring movement mode

Pinned: prevents the object from moving in drag mode

Move entire object: sets slider values in drag mode so the user can drag an object around (without stretching)

Move single vertex: sets slider values in drag mode so the user can move a single vertex

Reset: in spring mode freezes the object in its current shape and unpins all vertices

Rotating gravity: if this toggle button is on then gravity acts in the vertical *screen* direction, otherwise it acts in the *y-axis* direction of object coordinates

Free unpinned vertices: if this toggle button is set then any unpinned vertices will be free to move, otherwise they are restrained

Sliders on the clay control panel:

Translate: controls the fraction of translate effect in drag mode

Squash: controls the fraction of squash effect in drag mode

Vertex mass: controls the mass of vertices in spring mode

Spring factor: controls the "springiness" of the springs in spring mode

Spring damping: controls the amount of damping in vertex motion along the direction of attached springs

Viscoelasticity: controls the amount of spring stretching over time

Medium damping: controls the amount of general motion damping due to the medium in which the object is moving

Gravity: controls the acceleration of gravity

Time step: controls the time step taken in the iterative solution process

SEE ALSO

animator(UG)

AUTHOR

Mark Halstead

CS 285 Final Project Report
Ugiris4d - An Interactive Viewer for 4-Dimensional UniGrafix Objects

Ajay Sreekanth

May 14, 1991

1 Introduction

Ugiris4d is an interactive viewer for 4-dimensional polyhedral objects. It allows a user to interactively transform the object in 4-dimensional hyperspace, in real time. It is believed that such a viewer could serve as an invaluable educational tool in trying to understand higher dimensional spaces, and that was the motivating factor behind its development.

Ugiris4d is modeled on *ugiris* , the already existing interactive viewer for 3-dimensional UniGrafix objects. The power of *ugiris* lies in its ability to display 2-dimensional projections of 3-dimensional objects, and vary the projection at interactive speeds depending on how the user transforms the object. This temporal variation of projected views greatly aids the process of conceptually reconstructing the third dimension, or the true 3-dimensional shape of the object. It seems natural to expect that the understanding of 4-dimensional structure would similarly also be greatly aided by the temporal variation of its projections, and *ugiris4d* seeks to accomplish that.

2 Viewing a 4-D Object

The two popular ways of viewing a higher dimensional object in a lower dimensional space are by using lower dimensional *slices* or *shadows* (projections).

Ugiris4d first calculates a perspective projection of a 4-dimensional object, resulting a 3-dimensional object. This 3-dimensional object is then rendered using the SGI Iris hardware, and this performs another perspective transformation resulting in a final 2-dimensional image.

3 Object Transformations in 4-D

3.1 Zoom/scale

A combination of keys pressed, and mouse movement allows the user to zoom in and out to view the object. Basically, this moves the eyepoint along the line that connects it to the origin in 4-space. The scaling option simply scales the final image on the screen and is useful when certain features want to be examined closely.

3.2 Rotate

For a 4-dimensional object, specifying an axis of rotation and an angle of rotation (as in 3 dimensions) does not uniquely define a rotation. This is due to the fact that there are two distinct and nonequivalent axes perpendicular to a given plane of rotation. However, it remains true in 4-dimensions as well, that a rotation can affect only two dimensions at a time.

It can be proved, that any 4-dimensional rotation can be achieved by a sequence of rotations that are limited to rotations in the six planes defined by the coordinate axes. Combinations of keys pressed and mouse movement allow the user to rotate the object in each of the coordinate planes individually. A sequence of these rotations can be used to achieve any given rotation.

3.3 Translate

Again, a combination of key strokes and mouse movement allows the user to translate the object along each coordinate axis independently. Obviously, a combination of these can be used to achieve any desired translation of the object.

4 Lighting and Shading

Color information can be used as a visual cue to represent additional information. For example, the 3-dimensional projection of a 4-dimensional object could be colored so that the color of any point depends on the value of its fourth coordinate (which has been lost in the projection). This is essentially identical to the idea of depth cueing used in displays of 3-dimensional objects. This approach has been adopted, and these color cues certainly help in trying to comprehend 4-dimensional structure.

In this shading model, the colors of all the vertices are calculated from their 4th coordinates, and then the faces (or wires) are Gouraud shaded, so as to represent the variation of the coordinate value. Initially interpolation was performed across the visible color spectrum (from violet to red), but the visual cues obtained from this were not intuitive, so color is now interpolated between red and yellow to represent the variation of the 4th coordinate value. The colors are contrasting enough to allow easy detection of the color difference, and also the eye intuitively perceives shades of orange as between the colors red and yellow, so

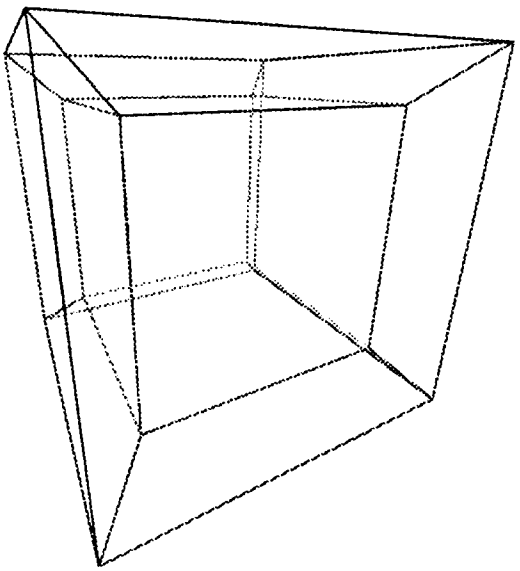


Figure 1: Projection of a Hypercube

this visual cue is effective.

With such a shading model, or even in the case of normal rendering where adjacent faces have the same color, it is extremely difficult to detect edges without a proper lighting model. It would have been necessary to use a lighting model in 4 dimensions to render these objects, as first projecting to 3 dimensions and then using a standard 3-dimensional lighting model will not produce correct results. An accurate lighting model in 4 dimensions is not a simple extension since now volume elements are shaded, rather than surfaces as in 3 dimensions. It was beyond the scope of this project to study and/or develop a 4-dimensional lighting model, so currently no lighting model is used.

In order to help distinguish adjoining faces, a wireframe option is provided. This basically draws a wireframe along the edges of all defined faces.

When drawing only wireframes, the lack of thickness of a wireframe drawn might make it difficult to see. An option is provided whereby the thickness of the wireframes drawn can either be increased or decreased.

5 Revealing the Inner structure of Objects

The 3-dimensional projection of a 4-dimensional object often contains faces that are completely enclosed by other faces, and are thus not visible from the outside. It is tough to fully comprehend the entire spatial structure without these faces being visible.

One way of revealing the inner structure of such objects is to model each face as being translucent, so that faces enclosed by others are still partially visible. Some SGI Irises have Alpha planes that can be used to render such faces with interpolated transparency. However, using these planes requires that the faces of the object be drawn in an order that is sorted according to decreasing depth. Since the list of faces would have to be sorted for

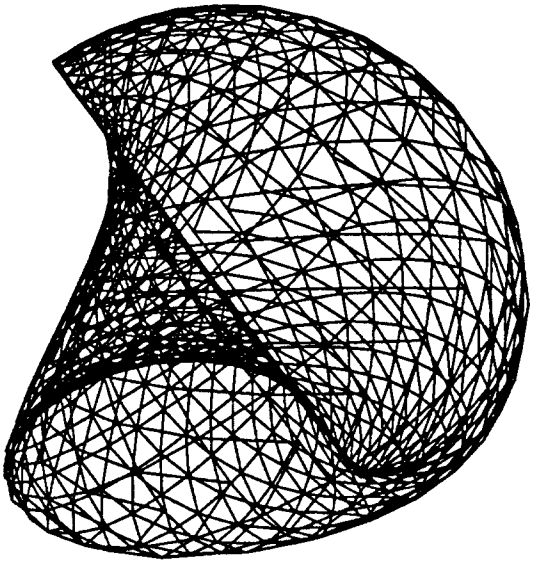


Figure 2: Projection of a 4-D Klein Bottle

each view refresh, this would have drastically slowed down the refresh, and so this method is not used.

Another way of revealing internal structure is to consider successive slices through the object. The user interface allows the user to vary the location of the near clipping plane, and moving this while it intersects the object displays successive slices through the object that is formed by the projection.

The third method of displaying internal structure is to consider a *hole* cut out of each face in 4-dimensional space. The inside of the object is now visible through these holes, and since parts of the original faces are preserved, it helps in visualizing the entire structure. The program *ug4hole* cuts such holes in the faces of an input 4-dimensional polyhedron, and the output of this program can be viewed with *ugiris4d* to better view the structure.

6 Implementation Details

Ugiris makes use of the specialized hardware on the SGI Iris machines to perform a lot of the rendering. It retains the static UniGrafix object structure, and all the lighting calculations and viewing transformations are performed using the Iris graphics pipeline.

Ugiris4d also retains the static UniGrafix object structure, but cannot make use of the Iris hardware for the viewing transformations since GL allows only 3-dimensional transformations and not 4-dimensional transformations. So for each view refresh (when the object is transformed), the projection into 3 dimensions has to be recalculated in software, and this is then rendered in a manner similar to *ugiris* . Having to recompute the projection for each view refresh significantly slows down the viewer, but it still is fast enough to achieve interactive speed.

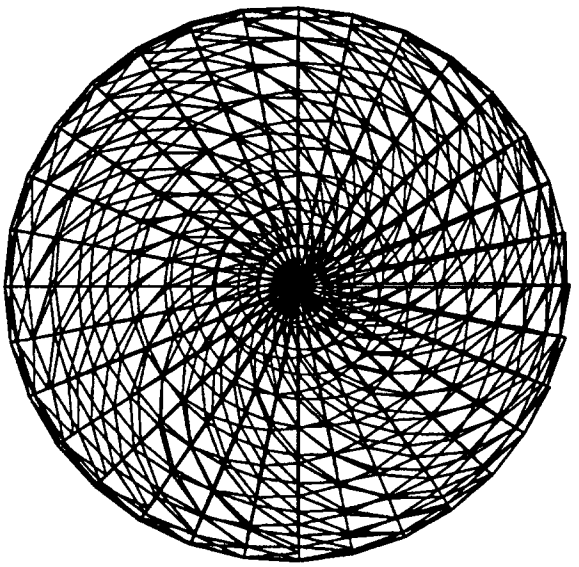


Figure 3: Another Projection of a 4-D Klein Bottle

6.1 Projecting From 4-D to 3-D

A perspective projection is performed, and it is done analogously to a perspective projection from 3 dimensions to 2 dimensions. This is done by first calculating the transformation so that the viewing direction is along the w axis. This transformation can be calculated by first calculating the rotation needed in the $X - W$ plane to make the x coordinate of the view vector zero, and then repeating this procedure to make zero the y and z coordinates of the eye vector, by rotating it in the corresponding planes. The object has also to be transformed by this transformation. Now the perspective projection can be computed by the perspective division of the x, y, and z coordinates by the w coordinate.

6.2 Transformations

Rather than apply each transformation to all the vertices in the UniGrafix file structure, the effective transformation is achieved by applying the inverse of the desired transformation to the eyepoint and leaving the original vertices intact. Now, the projection has to be calculated again with the new eye point, but this is done anyway when the view is refreshed.

7 Writing a 3-D projection to a file

At times it can be instructive to closely examine the object that is created by the 3-dimensional projection of a 4-dimensional object. The structure of this 3-dimensional projection could ideally be viewed with *ugiris* .

Ugiris4d allows the 3-dimensional projection to be saved to a file, at any stage. This file can now be viewed using *ugiris* .

8 Syntax Changes

The syntax of UniGrafix is changed only very slightly to allow the definition of 4-dimensional polyhedral objects. The only statement altered is the vertex statement, and its new syntax is:

v id x y z w;

The previously optional color id specification is ignored.

9 Observations

The most important observation is that on using *ugiris4d* , it is immediately obvious that 4-dimensional transformations are not at all intuitive. This makes it very difficult for a user to perform a series of transformations to obtain a desired effect. This is probably not a reflection on the effectiveness of the interface, but rather an illustration of the difficulties encountered in trying to visualize the 4th dimension.

An important issue seems to be the necessity of being able to interactively view a particular 3-dimensional projection of an object.

10 Future Work

Some of the extensions (as already discussed above) could be:

- Integrate *ugiris4d* with the Unigrafix *animator* so that a given 3-dimensional projection can be viewed in 3-dimensions alone.
- Implement efficient 4-dimensional clipping.
- Implement a 4-dimensional lighting model.
- Explore efficient methods of performing all the above, and all the transformations on the Iris graphics pipeline.

NAME

Ugiris4d – interactive unigrafix viewer for 4-dimensional objects

SYNOPSIS

ugiris4d [-e x y z w] < object

DESCRIPTION

ugiris4d is an interactive viewer for 4-dimensional extended UniGrafix objects that is essentially similar to *ugiris* in functionality. The input vertex statements must have 4 coordinates each. The other UniGrafix statements are unchanged. *ugiris4d* forms a 3-dimensional projection of the 4-dimensional object that is then rendered in a manner similar to *ugiris*.

Available options:

–e <x y z w>

Initially sets the eyepoint at the specified location in 4-dimensional space. The default eyepoint is (0 0 3 3).

After the object is displayed initially, it can be interactively manipulated in real-time as follows.

Zooming: The image can be resized by using the left mouse button. While the button is depressed, moving the mouse to the right will move the eyepoint closer towards the origin, and moving it to the left will do the opposite - move the eyepoint further away from the origin.

Scaling: Holding down the left shift key and depressing the middle mouse button causes the image to be scaled. When the mouse is moved to the left, the image is scaled down, and it is scaled up when the mouse is moved to the right. This does not change the eye point or the object position - only the final size of the displayed image is affected.

Rotation: The object can be rotated by using the middle mouse button in combination with certain keys on the keyboard. Even in 4-dimensional hyperspace, rotation can affect only 2 dimensions at a time. Consequently, there are 6 types of rotations that are sufficient to represent an arbitrary rotation. There are the rotations in the 6 coordinate planes. When the middle button is depressed, moving the mouse to the left/right will rotate the object in the X-Z plane, and moving it up/down will rotate in the Y-Z plane. When the left shift key is held down and the middle mouse button is depressed, moving left/right will rotate in the X-Y plane, and moving up/down will rotate in the Z-W plane. When the right shift key is held down and the middle button is pressed, moving left/right will rotate in the X-W plane, and moving up/down will rotate in the Y-W plane.

When the projection is frozen in 3-D, the resulting 3-dimensional object can be rotated as follows: When the middle mouse button is depressed, moving the mouse to the left/right will rotate the object about the Y-axis, and moving it up/down will rotate the object about the X-axis. When the left shift key is held down and the middle mouse button is depressed, moving the mouse to the left/right will rotate the object about the Z-axis.

Translation: The object can be translated along the 4 coordinate directions using the right mouse button. When the right mouse button is depressed, moving the mouse to the left/right will translate the object along the X axis, and moving up/down translates along the Y axis. When the left shift key is held down and the right mouse button is pressed, moving the mouse to the left/right translates along the Z axis, and moving up/down translates along the W axis.

When the projection is frozen in 3-D, the resulting 3-dimensional object can be translated as follows: When the right mouse button is depressed, moving the mouse to the left/right will translate the object along the X-axis, and moving it up/down will translate the object along the Y-axis. When the left shift key is held down and the right mouse button is depressed, moving the mouse to the left/right will translate the object along the Z-axis.

Slicing: The inner structure of the 3-dimensional projection can be revealed by slices through it. Holding down the right shift key, and then depressing the left mouse button helps control the position of the near clipping plane. Moving the mouse to the left moves the near clipping plane closer

to the eye, and moving it to the right moves the plane farther away from the eye.

Pressing the space bar prints a menu that lists the available key stroke options.

S : Writes the current 3-D projection to file ugiris.out

G : Toggles whether or not to Gouraud shade the object with the colors at the vertices indicating the values of their 4th coordinate

W : Toggle whether or not to display a wireframe along the edges around all the faces.

R : Raise thickness of wireframes.

L : Lower thickness of wireframes.

E : Print out current position of the eye point.

D : Toggle whether or not to use depth cueing.

B : Toggle whether or not to display back faces.

3 : Toggles whether or not to freeze the projection in 3-D. When the projection is frozen, then the resulting 3-dimensional projection can be manipulated using an interface similar to that of *ugiris*.

EXAMPLE

cat ~ug/lib/Hcube | ugiris4d -e 1.0 2.0 3.0 4.0

SEE ALSO

ugiris (UG), ug4to3 (UG), ug4hole (UG)

BUGS

I haven't implemented clipping in 4-dimensions, as that would slow things down tremendously since this clipping would have to be performed in software each time the image is refreshed. This leads to incorrect images when the eyepoint enters the object, but as long as the eyepoint is constrained to lie outside the object, there are no problems.

Also, I haven't implemented lighting in 4-dimensions. This leads to problems in detecting edges if adjacent faces are colored the same.

AUTHOR

Ajay Sreekanth

Mechanism Editor

Milind M Joshi

1.0 Introduction

The design of mechanisms usually involves two phases. The designer has to first decide the sizes of the links and the positions of the joints. Once he has put together a mechanism he has to check if it functions properly. The function of any mechanism is the motion of a subset of its links relative to the ground. In case the function is unsatisfactory, the designer has to go back and change the links and/or joints and try again.

Our program is designed keeping these two phases in mind. It is thus made up of two loosely coupled modules viz.

- Mechanism Editor
- Mechanism Animator

To speed up the animation process, we have another module which precomputes the invariants for the animation.

In the following sections we described the data-structures and implementation of each of these modules. We then describe how these three modules can be put together to build a tool for interactive design of mechanisms.

2.0 The Editor

The editor is a very simple graphical editor which allows you to add/delete links, wheels and joints. The exact user interface is described in the manual pages. In this report we discuss the implementation issues.

2.1 Data Structures

The editing of a mechanism involves adding and deleting components. The inserts and deletes have to satisfy some constraints e.g. whenever a link is deleted, all joints involving that link have to be deleted.

We use a list of links and a list of joints as our data structures for storing the mechanism during the edit phase. With each link we also keep a set of pointers pointing to the joints connected to it and with each joint we keep pointers to the pair of links it joins. The set of pointers gives us a quick way of finding the interconnection between links.

2.2 Implementation

The editor is implemented as a finite state machine which waits for mouse events and takes appropriate action depending on the current state and the mouse event.

As new links and joints get added, corresponding entries are added to the data structure. Whenever user finishes drawing a link or a pair of wheels a new *gl object* is created for that link or pair of wheels.

3.0 The Planner

The job of the planner is to precompute the invariants so that the animator can be speeded up. The planner generates a linked list of pseudo-instructions which are executed by the animator.

3.1 Data Structures

Any mechanism has a natural graph representations with joints as nodes and links as edges. But for the purposes of solving a mechanism, it is better to consider the dual of this representation i.e. a graph with links as nodes and joints as edges.

e.g. consider a simple mechanism

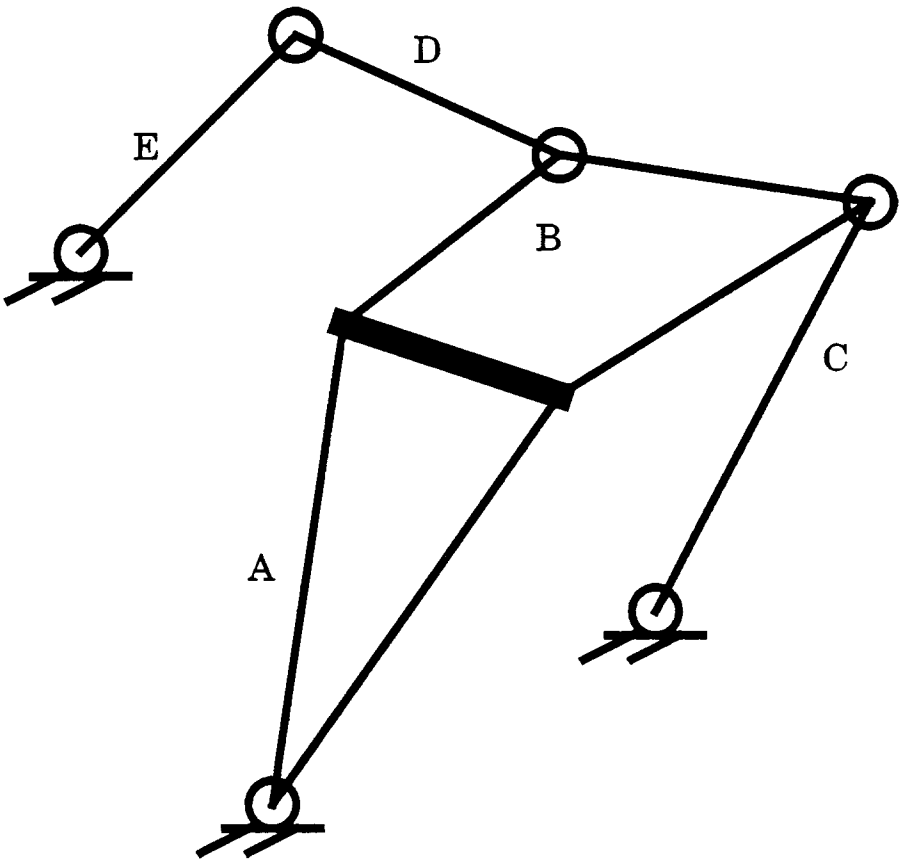


FIGURE 1. a simple mechanism

Where the revolute joints are represented as circles. and the prismatic joints are represented as thick lines.

The mechanism shown above can be represented as

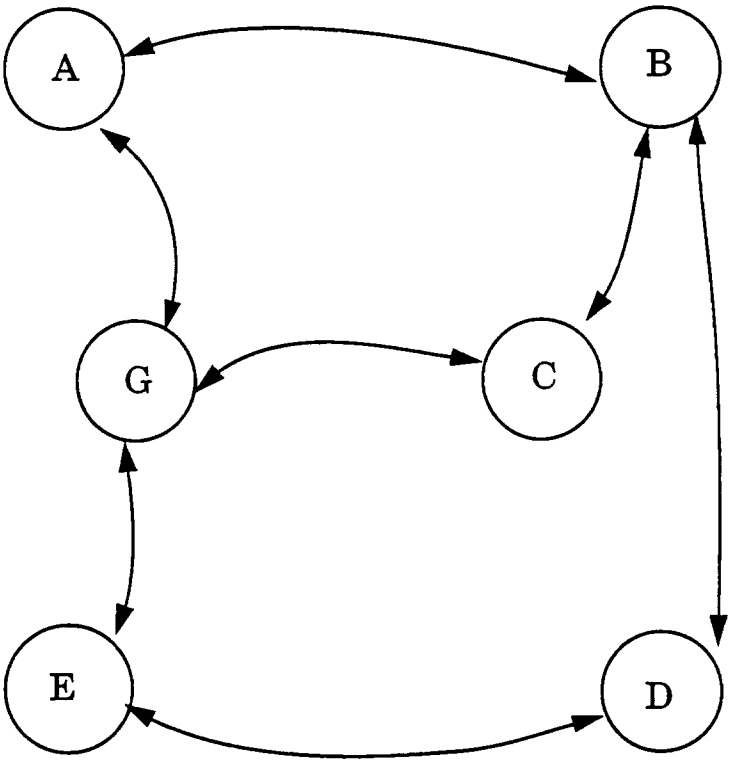


FIGURE 2. Graph

where G is the ground link.

The planner needs to find solvable subgraphs of the mechanism graph as described in the next section. Thus we store the graph in its matrix representation in the planning stage. There are two reasons for choosing this representation -

1. It is very easy to find the solvable subgraphs and rigid subgraphs using the matrix representation.
2. The number of links does not change between the planning stage and the animation stage and thus use of matrix representation does not cause dynamic allocation problems.

Thus we store the above graph as a matrix shown below

X	G	A	B	C	D	E
G	0	1	0	1	0	1
A	1	0	2	0	0	0
B	0	2	0	1	1	0
C	1	0	1	0	0	0
D	0	0	1	0	0	1
E	1	0	0	0	1	0

FIGURE 3. The matrix representation.

Note the different numbers appearing in the graph. Here 0 means no joint, 1 means revolute joint and 2 means prismatic joint. This information helps planner to decide which *subgraph-solver* function to use.

The other important data structure is the output of the planner. The planer generates a list of pseudo instructions which gets executed by the animator. It will be easier to understand the structure of these pseudo-instructions once we have seen the implementation of the planner.

3.2 Implementation

The purpose of the planner is to plan out the steps in animating a particular mechanism. Let us use the above mechanism to show how a plan can be generated to animate it.

Before we can make a plan we need to know the link that will be rotated. In our program, we allow only the grounded links with revolute joints to rotate. One such link is C. Let us build a plan to animate the mechanism when C is rotated uniformly.

If we know the amount of rotation of C, we can find the position of the joint BC. Since A is connected to ground, it can only rotate about the joint AG. Thus given the position of the joint BC the position of both B and A can be determined since those are rigid links. To put this in more abstract terms, we are trying to find a path with four nodes in the mechanism graph where the position of the two *end* nodes in the path is known. The path we discovered was GABC with position of G and C known.

The reason for emphasizing the word *end* in the above description is as follows. If the two known links are not the end nodes, and since we have a four node path, two things are possible.

1. One of the unknown links may be connected to two known links. But when this happens, it may not be possible to find a position for the unknown link. e.g. when B,G are known and A is unknown it may not be possible to have a possible position for A which preserves its length. We use this condition to eliminate rigid submechanisms.

2. The other possibility is that, one of the unknown links is not connected to any known link. To solve a link we need at least one connection to a known link and thus we have to wait till that happens.

Following the path algorithm described above we can see that after solving A and B we get a new path GEDB in our example which is solvable. Once GEDB is solved we know the positions of all the links and thus we have a complete plan.

To solve any 4 node path we also need to know the structure of the edges i.e. the joints connecting them, since the joints determine the possible motion of the links relative to each other.

Thus our planner essentially looks like

1. output the name of the link being rotated
2. while (there exist unknown nodes){
3. look for nodes with unknown position connected to two known nodes. If such nodes exist, generate an error message and return.
4. find a path with 4 nodes where the position of the end nodes is known and the other two nodes is unknown. If no such paths exist, generate an error message and stop.
5. output the names of the nodes and the structure of the edges between them.
6. }

Another important point in the implementation of the planner is the representation of wheels. We allow wheels only at grounded joints and in pairs. Thus wheels are always connected to two links determining their relative rotation. Thus a pair of wheels acts like a normal link connected to two other links and we store the pair as a link ignoring [1] its connection to the ground. The only difference is that we assume that there are *pseudo-wheel-joints* which connect the pair of wheels to other links. The wheel links can be solved as usual.

4.0 The Animator

4.1 Data structure

The data structure used by the animator is the output of the planner i.e. a linked list describing the sequence in which positions of nodes can be found.

1. or rather implicitly assuming.

Thus the structure for the above example will look something like

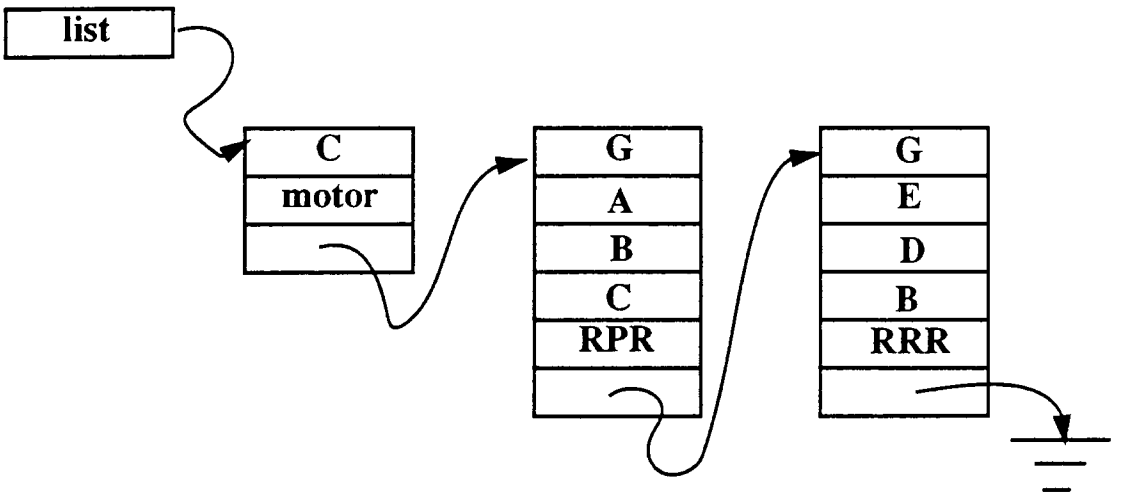


FIGURE 4. The list of instructions generated by the planner

The job of the animator is now very simple. It essentially does the following -

```
1. while (animation is going on){
2.      for (current_instruction = list; current_instruction!= NULL; cur-
            rent_instruction = current_instruction->next){
3.          call the appropriate solver with the known and unknown links to find new posi-
            tion of the links.
4.      }
5.      render the links;
6. }
```

5.0 Conclusion

We can see that a powerful design tool for designing mechanisms quickly and easily can be build by putting together three simple components an editor, a planner and an animator. The key component of the tool is the planner which gets closed form solutions for the mechanism so that it can be animated easily.

We hope that animating mechanism will speed up the mechanism design process by helping the designer in visualizing the function of a mechanism.

6.0 Acknowledgment

It would have been impossible to implement this project in 4 weeks without a lot of help from Eric Enderton's Master's thesis. The key idea of planning out the steps for solving the mechanism is taken directly from his thesis.

7.0 The program

The program can be found in the directory ~c285-af/project. The executable is called MeAn. The man page in TeX form is stored in the file man_ani.tex

8.0 Comments

The project was made simple by the theoretical work done by Eric. All I essentially did was hacked up a program. But there was an important thing to be learned from this. The job of writing a program becomes very easy once you have understood what needs to be done and planned out how it can be done. In my case all the designing was already done by Eric and all I had to do was understand his design and implement it.

9.0 Adding other subgraph solvers

There are three things that need to be done by anyone adding other subgraph solvers -

1. Modify the loop in the *plan* function so that the function *next_step* gets called in case of the subgraph solver being added.
2. Add another *else if* to the *next_step* function so that it builds an instruction for the solver being added.
3. write a solver which takes the instruction as an argument and modifies the translation and rotation parameters for the unknown links.

NAME

ugmech - a mechanism editor and animator

SYNOPSIS

ugmech

DESCRIPTION

ugmech has two loosely coupled modules viz. mechanism editor and mechanism animator.

The mechanism editor can be used to design a *planar* mechanism consisting of polygonal links, friction wheels, revolute joints and prismatic joints.

The animator allows one to check the function of a mechanism by moving a link connected to ground. The animation is continuous and can be frozen in between. To use the animator the mechanism should have only one degree of freedom.

USAGE

ugmech is structured as a hierarchical set of states. Each state has a corresponding menu with options to go up and down the hierarchy. The right mouse button can be used to pop-up the menu corresponding to the current state. There is a set of functions which can be executed from each set. The left and middle mouse button along with some of the menu options help in the execution of these functions.

The following paragraphs describe the states along with the menus and functions associated with them.

Top

This is the top level state and ugmech always starts in this state.

Menu

Edit

Changes the state to Edit.

Animate

Changes the state to Animate.

Save

Allows one to save the mechanism in a text file. It pops up a sub-window to input the name of the file in which the mechanism is to be saved.

Load

Loads a new mechanism into the editor. The filename is accepted via a sub-window. Note that this deletes the old mechanism from the editor.

Quit

Allows one to quit ugmech.

Debug

Dumps the current state of the mechanism.

Mouse Buttons

The left and middle mouse button have no function in this state

Edit

This is the mechanism editor module.

Menu

Add

Link

Lets you add a link to the mechanism. To begin a link, move the cursor to the desired point and click the left mouse button. Additional points can be added by moving the cursor and clicking the left mouse button. To complete the link press the middle mouse button. Whenever the cursor moves near a point (vertex of the polygon corresponding to a link) of a preexisting link, it automatically snaps to that point. Adding a new point there will create a *revolute* joint between the two links. Whenever two links have a pair of revolute joints connecting them, the pair of joints gets converted to a prismatic joint between those points. During the creation of the link, rubber banding is used to show the current shape.

Wheel

Wheels can only be added between two preexisting grounded points. The two points can be chosen with left mouse clicks. Once the two points are chosen, two wheels are displayed with a radii ratio dependent on the current position of the mouse. Whenever the desired ratio is obtained, click the middle mouse button to add the wheels.

Ground

To ground a point of a link, move to that point and click the left mouse button.

Delete

Allows one to delete a link or a pair of wheels. To delete just move the cursor over the link to be deleted and click the left mouse button.

Pop

Takes you back to the top level.

Mouse Buttons

The function of left and right mouse button depends on the function chosen.

Animate

This is the mechanism animator module. To animate a mechanism select a grounded link by clicking the left mouse button over it. To freeze the animation press the middle mouse button. To go back to the top level use the pop menu option.

Menu

Speed

very slow

rotate the chosen link 0.002 radians per step.

slow

rotate the chosen link 0.004 radians per step.

medium

rotate the chosen link 0.01 radians per step.

fast

rotate the chosen link 0.02 radians per step.

very fast

rotate the chosen link 0.04 radians per step.

Pop

Go back to the top level.

OPTIONS

None!

FORMAT OF THE MECHANISM DESCRIPTION FILE

The format for a mechanism description file is as follows.

description of the links

description of the wheels

edscription of the joints

Each link is described as

l *i*

x1 y1

x2 y2

.

.

x*n* y*n*

l *i+1*

.

.

To describe a wheel use the following format

w *i+2*

ratio of the radii

The centers of the wheel are automatically defined by the joints. To describe a joint.

j

from from's point to to's point

.

.

BUGS

Load and save options don't do anything! Use *DEBUG* to dump the mechanism.

Only RRR, RPR, RWR solvers are implemented.

The mechanism specification language is too terse. We are assuming that nobody will manually enter it.

AUTHOR

Milind M Joshi

A Graphical Tool for Algebraic Curves

Ashutosh Rege

Abstract

This project deals with solving various problems involving planar algebraic curves. These include plotting algebraic curves, determining critical points, intersections, and solving planar decomposition problems. A software framework has been created which should make development of further tools, such as those for three and higher dimensions, an easier task. The framework provides an environment for solving various algebraic problems which arise in connection with algebraic curves such as computing pseudo-remainders, greatest common divisors, resultants, Sturm sequences etc.

1 Introduction

Algebraic curves arise in a variety of situations. The following examples give an idea of the applications in which manipulating algebraic curves and solving problems concerning them play an important role.

- *Piano Mover's Problem* One of the classical approaches to solving robot motion planning problems is to formulate it as a problem of finding a path for a point object through configuration space avoiding the obstacle space contained therein. The obstacles are modelled as semi-algebraic sets, i.e. sets whose boundaries are given by algebraic surfaces. In the two-dimensional case, as shown in Figure 1, the boundaries are given by algebraic curves. The decision problem then is to determine if the start point and end point for the object lie in the same connected component. In most algorithms for the Piano Movers' Problem, e.g. the Roadmap algorithm [Ca], the basic idea is to project the higher dimensional surfaces down one level and ultimately to the plane determine the critical points there and fold back to the higher dimensions. Thus determining intersections and critical points in the plane forms an important component of such algorithms.

- *Geometric Reasoning* A typical problem in geometric reasoning is shown in Figure 2. Here the problem has to be solved in an algorithmic manner. One approach to solving such problems is to formulate them as a decision problem in elementary algebra. Algorithms for solving such decision problems use quantifier elimination which in turn can be done by solving systems of polynomial equations and inequalities. A typical approach is provided by Collins [Col] and is called a cylindrical algebraic decomposition or CAD. The basic idea behind the approach is to obtain a decomposition of higher dimensional space by projecting down on to lower dimensions. Details can be found in [Col,Ar].

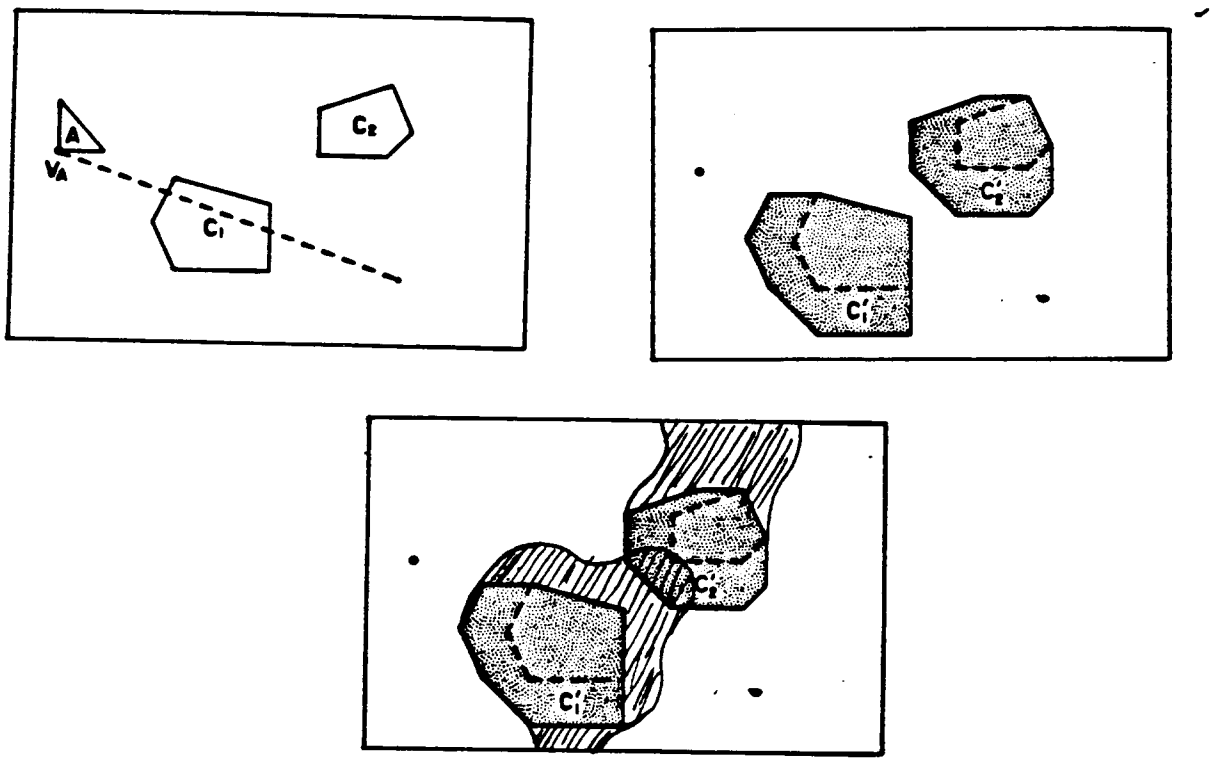


Figure 1: The Piano Movers' Problem

- *Grasping* One formulation of the grasping problem for smooth objects is to parametrize the location of the fingers over the curve defining the object. A grasp is stable iff the line joining the points of contact is contained within the friction cones of both fingers. This condition gives rise to set of algebraic equations in the two parameters corresponding to the location of the two fingers. Thus, we obtain algebraic curves as shown in Figure 3, corresponding to the stable regions. A grasp is stable if and only if it lies within the regions bounded by the curves. Thus the solution to the problem of determining whether a given grasp is stable involves solving the point-location problem in a cellular decomposition of the algebraic curves; the inverse synthesis problem of developing a grasp for the object involves determining the cellular decomposition, i.e. the regions bounded by the curves.

- *Offsets and blending* ([HH]) Mechanical parts tend to have secondary surfaces whose purpose is to connect the primary surfaces which provide the functionality of the part. Further in

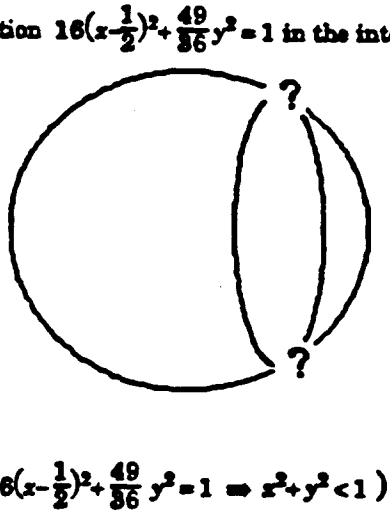


Figure 2: Geometric Reasoning

some applications such surfaces and curves provide functionality of their own such as fillets for stress reductions, transition surfaces for enhancing fluid flow etc. Algebraic curves also arise in the computations of offsets to planar shapes such as mechanical parts.

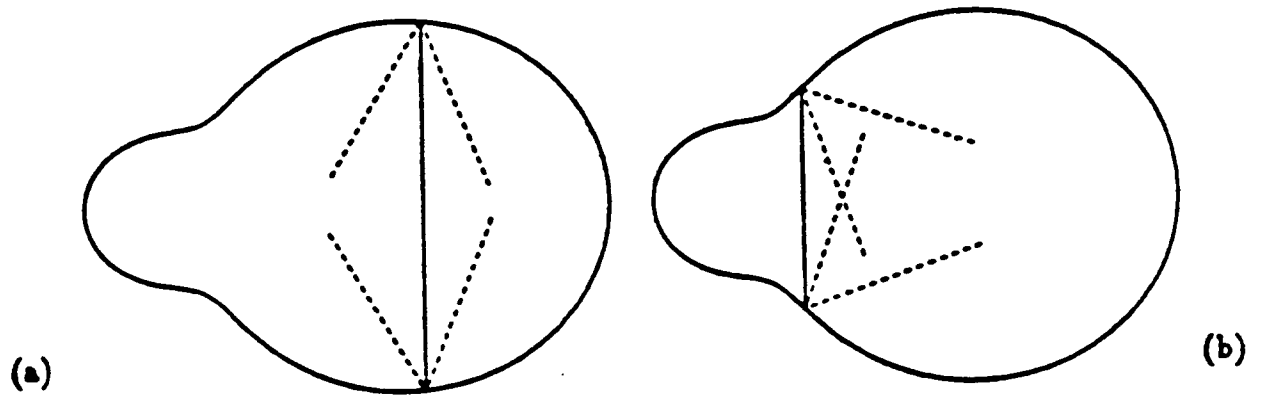
In this report we describe a graphical tool for solving various problems involving planar algebraic curves. The immediate goal of this project is to implement algorithms for algebraic curves' problems such as intersections, critical points, decomposition etc. More generally, though, the project has involved developing software tools to deal with planar algebraic curves and provides a framework for further development of tools for manipulating planar and higher dimensional algebraic curves.

1.1 Overview of the report

In subsequent sections, we provide a description of various aspects of the project. In section 2, we describe various methods for plotting algebraic curves. We show how Sturm sequences combined with an intelligently directed planar sweep can provide fast plotting of curves. Section 3 deals with computing intersections and critical points and Section 4 describes some of the computational algebra issues that arose during the course of the project.

2 Plotting Algebraic Curves

In this section we describe the various approaches considered for plotting algebraic curves. More specifically, we are given some planar algebraic curve, i.e. a polynomial equation $f(x,y) = 0$ and ranges for x and y, $x \in [x_{min}, x_{max}]$, $y \in [y_{min}, y_{max}]$ and a value of the desired resolution ϵ. The output should be a plot of the curve within the ranges given and up to the desired resolution (up



(a) A stable grasp. (b) An unstable grasp.

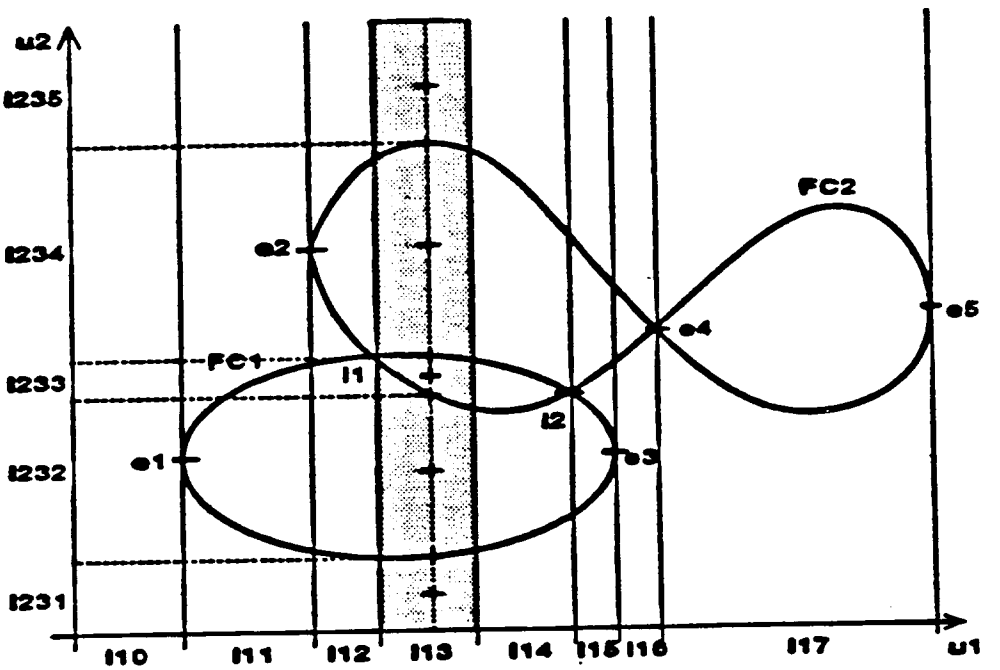


Figure 3: Grasping

to the inherent resolution of the monitor/window). The figures at the end of the report show the output of the plotter for various algebraic curves.

2.1 Marching Boxes

The idea here is to determine the sign of the polynomial at the 4 corners of a box of a certain size, in the planar region under consideration, preferably of the order of a few pixels. Assuming we start out with a box which contains some section of the curve, the algorithm proceeds by examining the signs of the function (in this case a polynomial) at the four corners of the box. If any two adjacent corners of the box are of opposing signs we can conclude that the function has a zero in between the two corners and therefore a piece of the corresponding curve passes between them. We next examine the boxes adjacent to the corners the curve passed through and so on. Proceeding in this fashion, we can plot the curve. The advantage is that we will consider a number of boxes proportional to the size of the curve. There are two problems with this approach : the first is that the algorithm needs a start point for each connected component of the curve. This can be handled without much ado. One could, for example, use some logarithmic search scheme to determine the initial points. The second problem is more serious. The algorithm suffers from the flawed assumption that between any two corners of the curve there is only one zero of the function. In general there can be an arbitrary number of zeros (depending on the degree of the polynomial) and the approach as described above would not locate them - in fact it would located none if the number of zeros was even. This is true even if the smallest possible pixel-box was used. The problem lies in the fact that the algorithm has no way of determining the multiplicity of zeros. Of course we could fix the algorithm so that at each box it determines the number of zeros in between any two corners. We can do this using some of the techniques outlined below. However, it seems *a priori* that the added computations may well make this approach slower than some of the others below.

2.2 Sturm sequences

The basic idea behind this approach is to exploit the fact that we are dealing with curves which are zeros of polynomials. If we could determine the zeros within the ranges given we could plot the curve. The first approach one would consider is to instantiate the curve for various values of x (depending on the resolution required) and then solve for the zeros of the resulting polynomial in y. This brings us to the problem of determining roots for a polynomial :

Let $f(x)$ be some univariate polynomial. Let $g(x)$ be a polynomial such that the greatest common divisor, $(f, g) = 1$. A *Sturm sequence* is a negative remainder sequence of polynomials $\{p_i\}$ i.e.

$$p_{i+1} = -remainder(p_{i-1}, p_i) \quad or \quad p_{i-1} = q_i p_i - p_{i+1}$$

with the Sturm property,

$$sign(p_{i-1}) = -sign(p_{i+1})$$

Now define $SA(f, g, x_0)$ to be the number of sign alternations in the Sturm sequence evaluated at x_0 i.e. the number of sign alternations in the sequence $p_0 = f(x_0)$, $p_1 = g(x_0)$, $p_2(x_0)$, ... We then have

Lemma 1 *(Sturm) The number of real roots in the interval* $[a, b]$ *is given by* $SA(f, g, a) - SA(f, g, b)$.

The above lemma implies an obvious sub-division algorithm for finding all the real roots of f : start with the interval under consideration. At each step subdivide the interval(s) which contain a zero into two intervals of equal size. Repeat till the intervals are refined to the desired accuracy or resolution.

We therefore have the basic approach for plotting planar algebraic curves :

1. Instantiate $i = 0$, $x_i = x_{min} + i * \epsilon$. If $x_i > x_{max}$ terminate.
2. Let $F(y) = f(x_i, y)$. Let F' be the derivative of F. If F and F' have a non-trivial gcd, divide each one by the gcd.
3. Determine the roots of F by computing the Sturm sequence for F and F'.
4. Set $i = i + 1$. Go to 1.

Here ϵ is the desired resolution. The number of steps required is given by

$$\frac{(x_{max} - x_{min})}{\epsilon} E(m) \log \frac{(y_{max} - y_{min})}{\epsilon} + \frac{(x_{max} - x_{min})}{\epsilon}(S(m) + E'(n))$$

where

- E(m) is the time required to determine sign alternations for a polynomial remainder sequence with first polynomial of degree m.
- S(m) is the time required to compute the Sturm sequence of a univariate polynomial of degree m
- E'(n) is the time required to instantiate in the first variable a bivariate polynomial of degree n in the first variable.

2.3 Improved algorithms

The algorithm in the previous subsection could do much better if it didn't have to compute the Sturm sequence at each value of x. Instead the Sturm sequence could be computed for the bivariate polynomial $f(x, y)$ by treating it as a polynomial in x with coefficients which are polynomials in y. Then at each step computing the Sturm sequence would simply mean instantiating the Sturm sequence already computed with the value of x under consideration. This would speed up the time to

$$\frac{(x_{max} - x_{min})}{\epsilon} E(m) \log \frac{(y_{max} - y_{min})}{\epsilon} + \frac{(x_{max} - x_{min})}{\epsilon}(nE'(n)) + S'(n, m)$$

where $S'(n, m)$ is the time required to compute the Sturm sequence for a bivariate polynomial with the first variable having a degree n and the maximum degree of the coefficient polynomials (in the second variable) being m.

We can further improve the running time of this algorithm if we could provide the algorithm with a better guess for the intervals in which the roots lie. This would substitute the logarithmic search with a constant time search. We can use the Chain Rule

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy$$

Along the curve f(x,y) = 0, so the change in f corresponding to the steps (dx, dy) is zero, i.e. $df = 0$ giving

$$\frac{dy}{dx} = \frac{-\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial y}}$$

Given a current value (x, y) and a change h in x, , we can use this identity to predict k in the next value $(x + h, y + k)$. Thus we will give the algorithm a better guess for the range of the roots instead of $[y_{min}, y_{max}]$. This will reduce the search to a very small interval.

3 Determining Intersections and Critical Points

Determining the intersection points of two curves is obviously equivalent to determining the points at which the polynomial equations defining them disappear simultaneously. In the univariate case we are given two polynomials $F = \sum_{i=0}^{i=n} f_i x^i$ and $G = \sum_{j=0}^{j=m} g_j x^j$. The two polynomials have no common root iff (f,g) = 1. That is, iff the following determinant is non-zero.

$$\begin{vmatrix} f_n & f_{n-1} & \cdots & f_0 & 0 & \cdots & 0 \\ 0 & f_n & \cdots & f_1 & f_0 & \cdots & 0 \\ \vdots & & & & & & \vdots \\ 0 & \cdots & 0 & f_n & f_{n-1} & \cdots & f_0 \\ g_m & g_{m-1} & \cdots & g_0 & 0 & \cdots & 0 \\ 0 & g_m & \cdots & g_1 & g_0 & \cdots & 0 \\ \vdots & & & & & & \vdots \\ 0 & \cdots & 0 & g_m & g_{m-1} & \cdots & g_0 \end{vmatrix}$$

The determinant is called the *Sylvester resultant* of F and G.

The above result can be generalized to multi-variate polynomials. In the bivariate case we can write the polynomial as one in say x with coefficients which are polynomials in y. We can then form the determinant as above to get a polynomial in y - this polynomial is denoted $Res_x(f, g)$. However since the coefficients are polynomials we get additional conditions as to when the two polynomials share a common root.

Lemma 2 *$Res_x(f, g) = 0$ iff there is a common solution of f and g, or the leading coefficients of f and g vanish simultaneously or the coefficient polynomials of f or of g have a common root.*

Thus in order to determine the intersection points of the two curves we first compute their resultant, determine its roots and at each root check if both polynomials vanish identically.

A point on a curve is said to be *critical* if the curve and its derivative intersect. Thus determining critical points is the equivalent to the previous problem of determining intersections, the only additional requirement is the computation of the derivative of the polynomial which defines the curve.

4 Algebraic computations

All of the various operations on algebraic curves outlined above require the computation of various algebraic entities such as Sturm sequences, greatest common divisors, resultants etc. In this section we sketch briefly or provide pointers to algorithms for such computations.

Pseudo-remainders : Computing the Sturm sequence or the GCD of polynomials requires the computation of polynomial remainder sequences as described earlier. In computing Sturm sequences we are interested only in the signs of the polynomials involved. Thus we are not restricted to using the Euclidean remainder sequence sketched in the previous section. The same holds true for gcd's : we need only determined the gcd up to similarity (i.e. multiplication by a constant). In fact, computations over the rationals as required by the Euclidean algorithm are slower than computing over integers since rational arithmetic involves many more evaluations of gcd's of integers at each step. Thus a faster way to compute remainder sequences and GCD's is to use pseudo-division to generate pseudo-remainders [Bu,Kn]. The basic idea is to multiply part of the dividend polynomial by the leading coefficient of the divisor polynomial in order to generate all integer values. Details can be found in [Kn,BT].

Subresultant polynomial remainder sequence algorithm : The resultant as defined earlier would seem to require the computation of the determinant of a sparse matrix. However a faster approach to computing the resultant is to use the subresultant algorithm outlined in [BT,Bu]. This algorithm is particularly suited for sparse multivariate polynomials.

5 Program Output

The figures at the end of the report show the output of the program including features such as critical point computation and intersection computation. The first figure shows the plot of a degree 4 curve, the conchoid along with its critical points. The next figure shows the intersection points of another curve, the deltoid, with the conchoid. Finally we have the intersection of a third curve with both previous curves.

6 References

[ACM] Arnon et al., "Cylinder Algebraic Decomposition", Siam J. Computing, 1984.

[BKR] Ben-Or et al. "The Complexity of Elementary Algebra and Geometry", JCSS, 1986.

[Bu] Buchberger et al., *Computer Algebra*, Springer-Verlag, 1983.

[BT] Brown, W and J. Traub, "On Euclid's algorithm and the theory of subresultants", JACM 18, 1971.

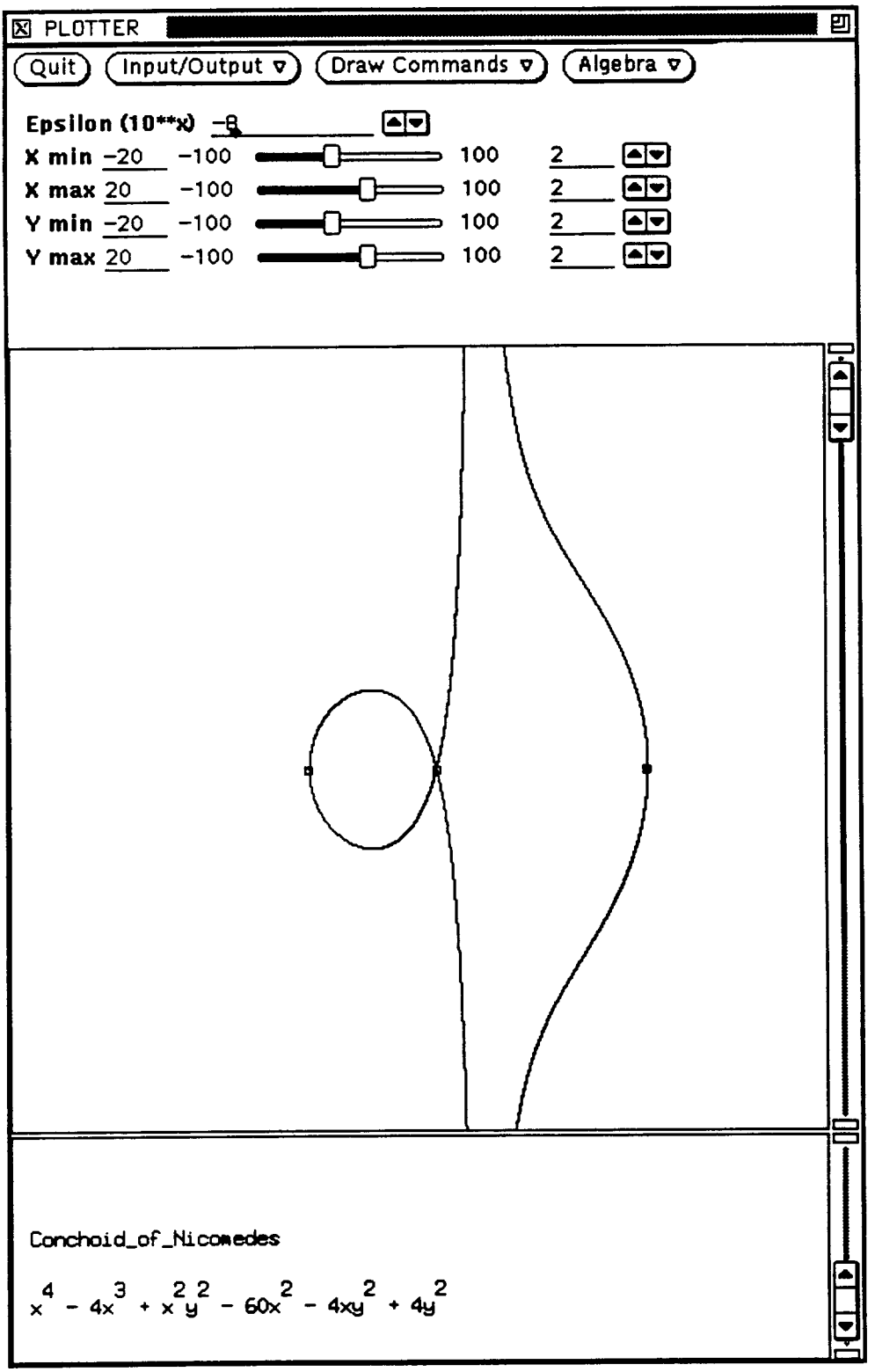
[Ca] J. Canny, "The Complexity of Robot Motion Planning", Ph.D. Thesis, MIT, 1987.

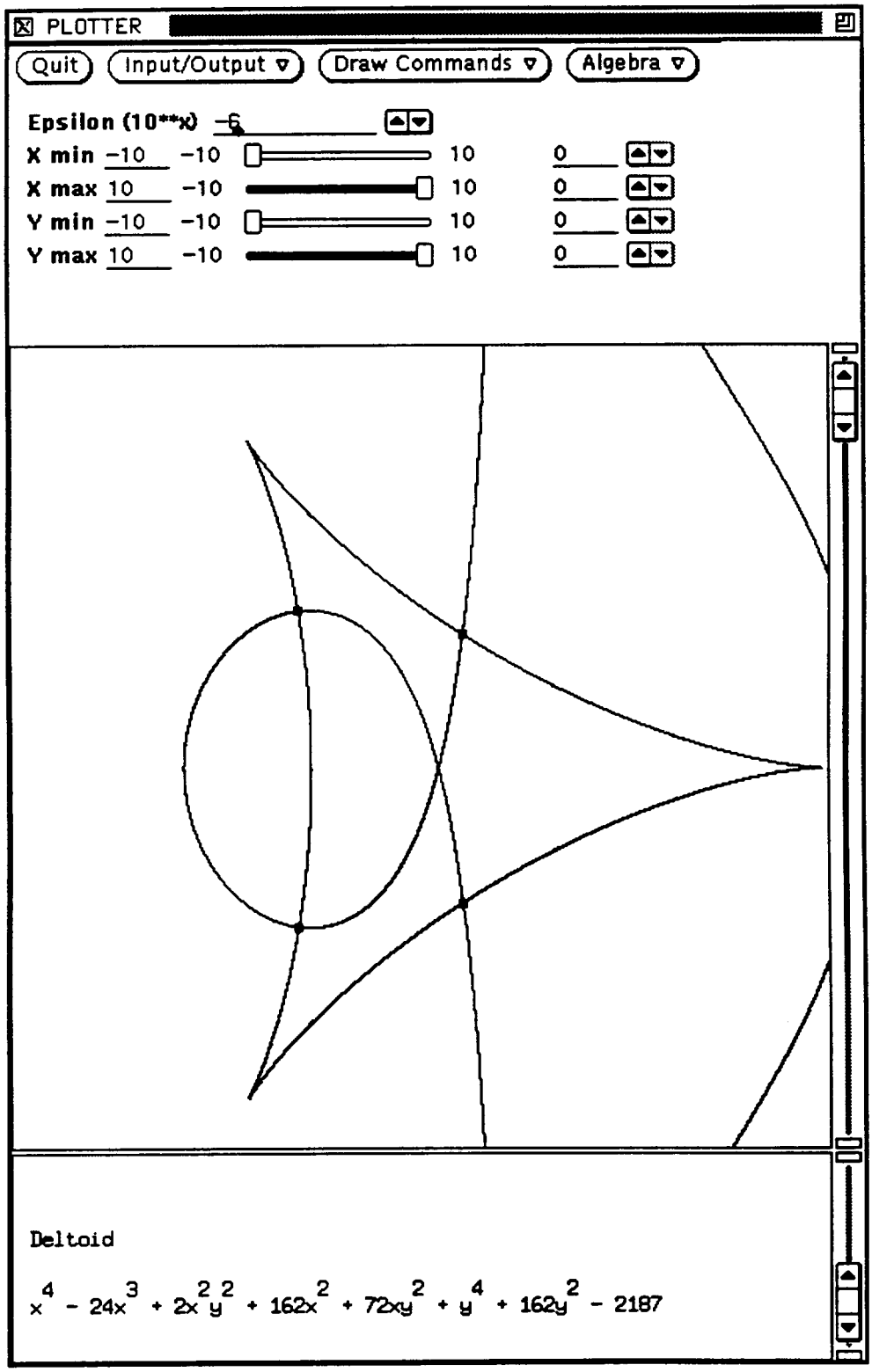
[Col] G.E. Collins, "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition:, Proc. 2nd GI Conf. on Automata Theory and Formal Languages, 1985.

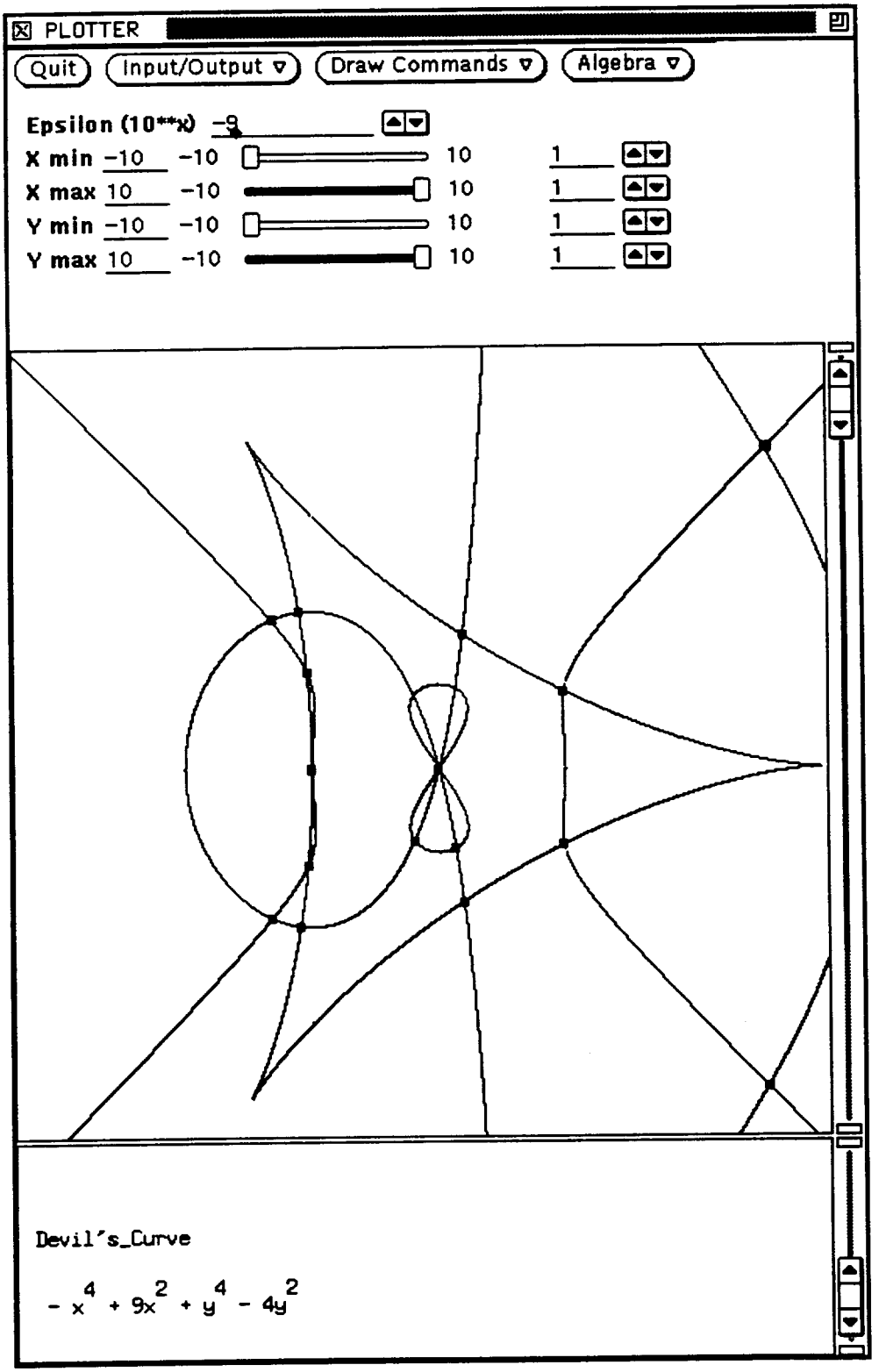
[HH] Hoffmann, C. and J. Hopcroft, "The potential method for blending surfaces and corners", Geometric Modeling : Algorithms and New Trends, G.E. Farin, ed. SIAM, 1987.

[Kn] Knuth, D.,"The Art of Computer Programming", vol II, Addison-Wesley, 1981.

[SS] J. Schwartz, M. Sharir, "On the 'piano movers' problem II", Advances in Applied Math, 1983.


PLOTTER
Quit
Input/Output
Draw Commands
Algebra
Epsilon (10**x) -8
X min -20 -100 100 2
X max 20 -100 100 2
Y min -20 -100 100 2
Y max 20 -100 100 2
Conchoid_of_Nicomedes
x^4 - 4x^3 + x^2y^2 - 60x^2 - 4xy^2 + 4y^2


PLOTTER
Quit
Input/Output
Draw Commands
Algebra
Epsilon (10**x) -6
X min -10 -10 10 0
X max 10 -10 10 0
Y min -10 -10 10 0
Y max 10 -10 10 0
Deltoid
x^4 - 24x^3 + 2x^2y^2 + 162x^2 + 72xy^2 + y^4 + 162y^2 - 2187


PLOTTER
Quit
Input/Output
Draw Commands
Algebra
Epsilon (10**x) -9
X min -10 -10 10 1
X max 10 -10 10 1
Y min -10 -10 10 1
Y max 10 -10 10 1
Devil's_Curve
$-x^4 + 9x^2 + y^4 - 4y^2$

MANUAL PAGE

NAME

plotter - graphical tool for algebraic curves

SYNOPSIS

plotter

DESCRIPTION

plotter is a program which runs under the X window system using the XView tools library. It is a tool for plotting planar algebraic curves and computing various properties such as critical points, intersections between curves, planar decomposition etc. **plotter** can be started up by simply typing in **plotter**. The various commands available are described below.

COMMANDS

plotter provides three basic command facilities : input/output, draw commands and algebra. These are the three main pop-up menus. The menu-choices within each can be accessed by clicking on the desired option with the right mouse button. In what follows NYI stands for Not Yet Implemented. Watch this man page for further changes to **plotter**.

Input/Output :

- **Monomial Form** Clicking on this option allows input to be typed in from standard input (stdin). The format for the algebraic curve to be input is : coeff1 xdeg1 ydeg1 coeff2 xdeg2 ydeg2 ... followed by some non-number character. Here the degrees have to be integers. E.g. to type in the equation for the circle $x^2 + y^2 - 1$:

  1 2 0 1 0 2 -1 0 0 a

  Following the non-number entry will be a prompt for the name of the curve.

  All white-space, carriage returns etc. are ignored.

- **Coefficient Form** (NYI) Reads input polynomial in coefficient form. E.g. (y**2 + 1)*x**3 - (y+10)*x

- **Load File** Prompts for file name to be read in. The file should be in the same format as the monomial form. Each curve should also have its name as in above format.

  E.g. 1 2 0 1 0 2 -1 0 0 a Circle

- **Save** (NYI) Will save desired polynomial(s) to the specified file.

Draw Commands

- **Decomp** Attempts to perform planar algebraic decomposition of the curves given. The algorithm implemented at present is not complete and needs to be debugged to handle complicated input.
- **Plot** Plot the algebraic curve specified.
- **Intersect** Computes and displays the intersection of two specified curves.
- **Intersect All** Computes and displays the intersection of all curves.
- **Critical Points** Computes and displays the critical points of given curve.
- **Clear** Clears the display.

Algebra

- **Derivative** Computes and displays the derivative of specified polynomial; prompts for variable to differentiate w.r.t.
- **PseudoRem** Computes and displays the pseudo-remainder of two specified polynomials.
- **Sturm Sequence** Computes and displays the Sturm sequence of a polynomial and its derivative
- Resultant Computes and displays the resultant of two specified polynomials with respect to a specified variable

PARAMETERS

All of the following parameters can be set by sliders provided in the display.
Epsilon This controls the accuracy of computation. Defaults to 1.0e-8
Xmin, Xmax, Ymin, Ymax Controls the display ranges.

AUTHOR

Ashutosh Rege

Three Dimensional Mandelbrot Like Sets with Applications to Stability Portraits

Raja R. Kadiyala

12 May 1991
CS 285
Final Report

Abstract

We have all seen the familiar Mandelbrot sets in two dimensions in both plain black and white and the enhanced color versions where a color is mapped to the *escape* time of a given point. We can also consider this as a problem in system theory where our equation of interest is $x_{k+1} = f(x_k) + g(x_k)u_k$. It is easy to see that the case of the Mandelbrot sets is a subclass of the above equation with $f(x_k, u_k) = x_k + c$ and $g(x_k) = 0$ where $x, c \in \mathbf{C}$. If we consider the situation where $x \in \mathbf{R}^3$ and $u \in \mathbf{R}^m$ with $f(\cdot)$ taking $\mathbf{R}^3$ to $\mathbf{R}^3$ and $g(\cdot)$ taking $\mathbf{R}^3$ to $\mathbf{R}^{3 \times m}$, we then have a general description of a nonlinear discrete time system with a state space dimension of three which we can use to generate three dimensional Mandelbrot like sets and observe the stability characteristics of a system.

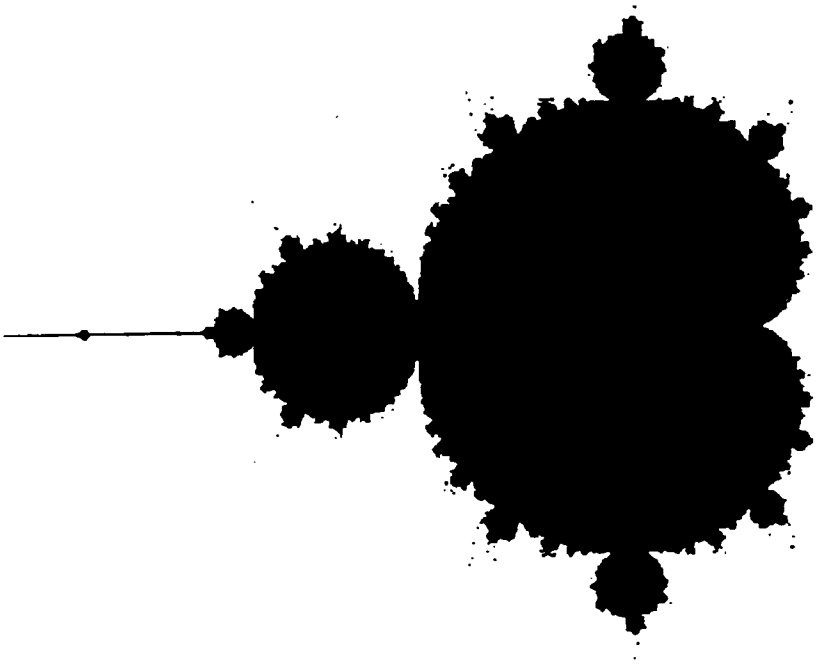


Figure 1: Unenhanced Mandelbrot Set

1 Introduction

So called phase portraits have been used to study the stability characteristics of two dimensional systems with a good deal of success, but we are truly limited by this visualization scheme since we have the restriction of two dimensions. We will show that a natural extension is to use a framework similar to the equations for the Mandelbrot sets but defined in three space.

One can see quite readily that this will be a rather large computational task. If we note that a majority of the calculations do not depend on each other, then we can see that the algorithm is ideal for parallel computing. We will show such a procedure for this problem using multiple workstations to offload the computation.

We will also present the visualization tools developed to survey all the information inherent in these plots.

2 Extension into Three Dimensions

The defining equation for the Mandelbrot sets is given by:

$$x_{k+1} = f(x_k) + R \tag{1}$$

where $x, R \in \mathbf{C}$. This gives rise to the familiar vanilla mandelbrot set shown in figure 1.

These sets may be enhanced by color to give a more artistic flair and display more information about the system dynamics. A typical mapping is to assign each point a color according to how many iterations it takes to escape some finite area (the so called *escape time*).

Equation (1) is very similar to a general class of nonlinear discrete time systems which is affine in the control input. These systems may be described by:

$$x_{k+1} = f(x_k) + g(x_k)u_k \tag{2}$$

where

- x is known as the *state vector* ($\in \mathbf{R}^n$)
  - The state vector x describes the current configuration of the system.
- u is called the *control input* ($\in \mathbf{R}^m$)
  - The control input u is what you have control over to *move* the system.
- $f(\cdot)$ and $g(\cdot)$ describe how the system behaves.

It is quite easy to see that the Mandelbrot sets are a subclass of the above system definition. Hence we have a simple extension into three dimensions.

3 Computational aspects

The computation of the dynamics in three space can become quit large very fast since we essentially grid up a cube of specified size (see figure 2) and compute the characteristics of each subcube. Since the calculation of the

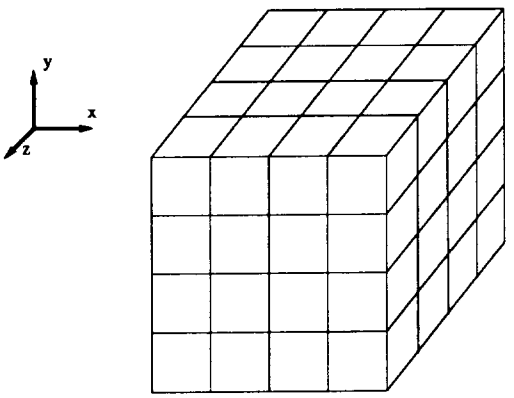


Figure 2: Gridding of the State Space

dynamics in each subcube depends only on itself, we can carry on the computations in parallel. With this in mind, a graphics server/computational client scheme was set up to offload the calculations. This also alleviated the need to write a full blown parser to gather the information necessary to define the system equations since some prewritten could be used that would dynamically link in a subroutine.

The software to link in routines worked for Sun computers but not for the Silicon Graphics machines, hence the logical choice was made to make the SGI the graphical sever which would then talk to the various Sun computers over ethernet using sockets. So we gained the ability to do parallel computation and had a simple way to specify the system equations (i.e. the equations could be described in a simple C or FORTRAN subroutine).

In the current version there can be up to sixteen computational clients that would receive the subroutine, dynamically link it in and wait for the chunk of state space points to calculate the escape time properties on and report back to the graphics server where the results would be displayed. This cuts down the time to calculate immensely. A pictorial description is given in figure 3 and we have the following loop performed over and over again:

1. Server initiates communication to each client (max of 16).
2. Server downloads (to each client) a source file which describes the dynamics of the system.
3. Each client dynamically links in the routine and is ready to compute.

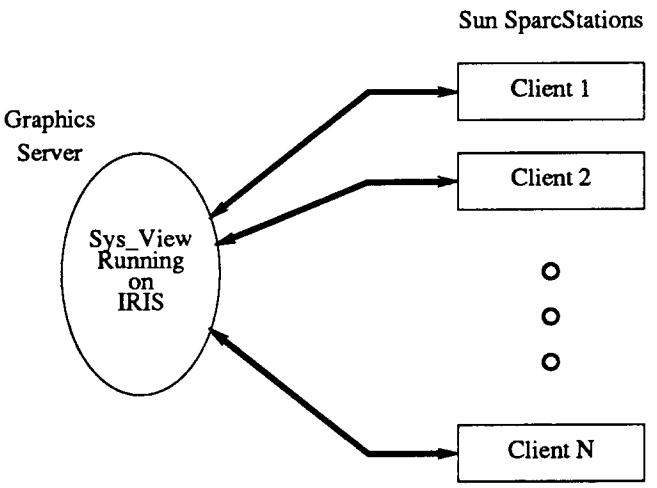


Figure 3: Graphics Server/Computational Client Set Up

4. Server grids up the state space and divides it into equal portions for each client.

5. Clients report back with results.

Since computations are essentially done in parallel we get an N times speed up, with hardly any overhead, and we again get the advantage of using dynamic linking to pull in the system description in the form of a C or FORTRAN subroutine. In addition, if we are only changing the grid size parameters and not the subroutine describing the system, then only steps 4 and 5 need be repeated.

4 Viewing the Results

One of the original ideas to view the data was to traverse along some axis and look at 2-D slices of the system (much like CAT scans). Multiple slices could be stored and in conjunction with transparency one could look past the initial slice to get more global information. Since the Iris' we have in our lab do not have α -planes and hence do not allow for transparency, this idea was not used. One could still display one slice at a time, but we feel that the global information lost would make this option not as attractive.

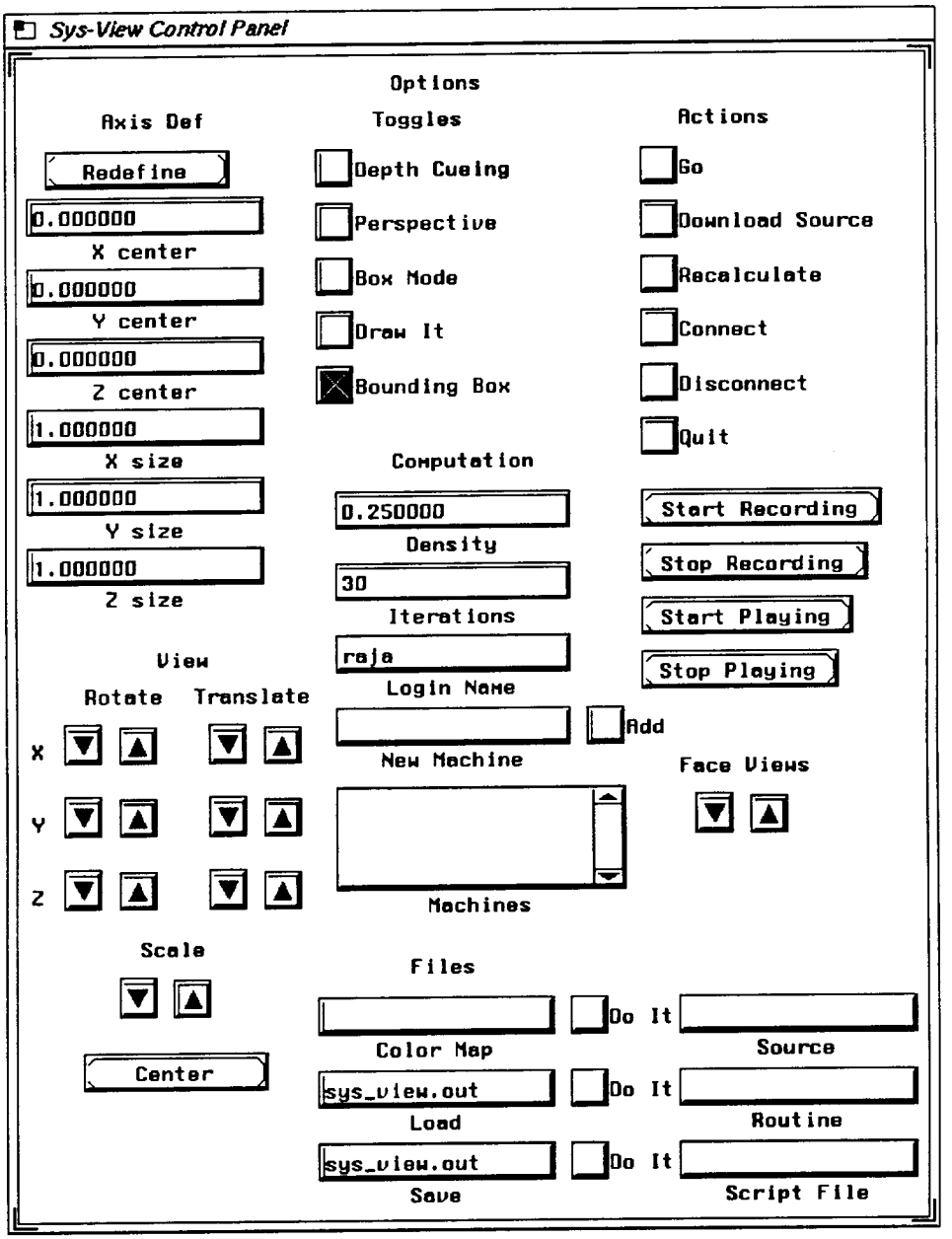


Figure 4: Sys-view Control Panel

The route taken was to create a portrait cloud of points that was not fully dense (thus allowing us to see *through* things to view the behavior of the system at various points). This was accomplished by defining a random point constrained to be in a subcube of the gridded cube from figure 2. This allows us to see quite a bit of detail in the global aspect of the dynamics while still allowing us to view the local nature. Depth cueing is also available to give some depth perception in the visualization.

The randomness of the state space points was necessary to see the distant points and it should be noted that it is easier to view the data in an orthonormal projection then in a perspective projection.

A control panel was created as the main user interface (see figure 4). It allows the user to chose from the plethora of various combinations given to view the data, and a complete description is given in the man pages for *sys-view*. A couple of features worth mentioning are the ability to record a sequence of button clicks in the control panel and then play them back for a sort of movie to traverse the escape portrait and secondly the ability to define your own color map for the escape portrait. Along with the control panel a plotting window to view the data is poped open (see figure 5 for a black and white version).

The region of attraction for a system is the locus of initial conditions whose trajectories steer towards the origin and is valuable piece of information to compute for a system. The region of attraction may be simply plotted by using a predefined color map which defines the 0^{th} entry to be white and all others to be black. Hence the points that do not leave the region are white while all points that do leave are black, thus giving us the region of attraction.

5 Conclusions

A tool for viewing the dynamics of three dimensional discrete time systems has been developed in an interactive environment. With the addition of computational clients on remote machines the calculations necessary can be carried through relatively quickly. Possible additions for the future would be to include the CAT scan like slices mentioned above as another possible viewing scheme and to add continuous time systems into the domain of possible systems to view. The latter would entail adding a runge-kutta type

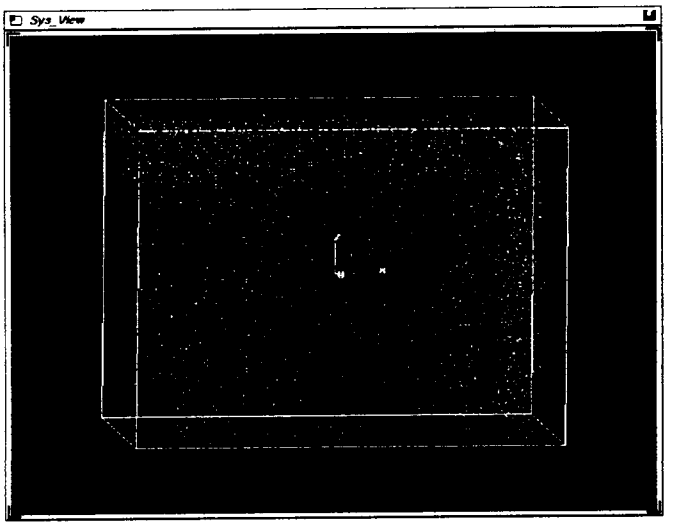


Figure 5: Sys-view Plotting Window

integration scheme to the computational clients and would most likely add in the computational delay, but would further enhance sys-view's ability to help visualize more systems.

NAME

sys_view – Interactively view a three dimensional dynamic system

SYNOPSIS

sys_view [-b *box mode*] [-c *color mapping*] [-d *pixel density*] [-i -j -k *position of bounding box*] [-l *login name to use on client*] [-m *remote machine*] [-n *number of iterations*] [-o *output file*] [-p *use perspective projection*] [-q *turn on depth cueing*] [-r *routine*] [-s *source_file*] [-x -y -z *dimensions of bounding box*]

DESCRIPTION

sys_view creates a three dimensional escape portrait of a dynamical system described by the subroutine in *source_file*, which may be either C or FORTRAN and is called by the name in *routine*. **sys_view** creates connections up to the machines specified through repetitive uses of the -m option (i.e. sys_view -m mach1 -m mach2 ...) and then downloads the file to the clients. The clients then dynamically link in this routine and compute equal portions of the trajectories. The results are then reported back to the server machine and a plot is created which the user may interactively view. Up to 16 machines may be used as clients.

OPTIONS

–b go into box mode which will draw cubes instead of pixels at the grid point. This is useful if a low density is used.

–c color mapping; Use the file contained in *color mapping* to define the colors to use in the escape portrait. This should be an ASCII file with each line containing four integers ranging from 0 to 255 specifying the index and the standard red, green, and blue (rgb) color values. (i.e. 0 12 87 39 would specify entry 0 to have a red value of 12, a green value of 87, and a blue value of 39). Typically entry 0 will be black and entry 255 will be white.

–d density; Use floating point value *density* as the parameter defining how dense of a pixel cloud to use when generating the escape portrait; valid range is 0 to 1, the default is 0.25 (anything larger than 0.6 will take more than a few minutes to run).

–i –j –k center of bounding box; Use these three numbers to define the origin of the bounding box. This is useful for studying behavior away from the origin; the default is zero.

–l login name; If the account name on the remote machine is different than on the local machine then this option must be set accordingly. The remote machine should allow entry of the local account through an entry in .rhosts.

–m machine; Specifies which machines to run the calculations of the trajectories on. For a low density it is not necessary to specify too many machines. In fact this may hurt you since the overhead involved in the communication may slow things down. For large densities it will be well worth the effort to spread the computing across as many machines as possible. A maximum of 16 machines may be specified in the following fashion: sys_view -m mach -m mach2 ...

–n number of iterations; Use the integer value *iteration number* as the parameter defining how many times to iterate a state point before evaluating its escape characteristics for the escape portrait; the default is 30.

–o output file; Use *output file* as the file to save to; the default is sys_view.out.

–p perspective projection; This allows the user to specify a perspective projection be used as opposed to the default orthonormal projection.

–q turn on depth cueing; This will allow the user to gain some depth perception as points further away are darker than closer points. This will, however, slow down the redrawing of the plot in interactive mode.

-r routine; Use *routine* to tell **sys_view** the name of the routine to call to execute the system equations.

-s source_file; Use *source_file* as the file to dynamically link in to get the system equations.

-x -y -z bounding box dimensions; Use these three numbers to define the boundary of the plot. If not specified sys_view tries to set them automatically.

USER INTERFACE

The user interface may be divided into two sections. The first being the control panel and the second being the interactive or plot window. The control panel is partitioned into seven groups. The characteristics of the bounding box may be defined in the *Axis Def* group, while various attributes may be toggled on and off in the *Toggles* group. These attributes include depth cueing, perspective projection, box/cube mode, draw it, and turn on and off the bounding box. The third subgroup is the *Actions* group which allows the user to start calculations, download a new source file, disconnect, reconnect to the clients and to quit. Note that this is the only way to quit sys_view. The *Computation* group allows the user to specify the density and number of iterations and also allows the addition of a new machine to the client list which is also displayed. It is suggested that a disconnect occur before the addition of a new client followed by a connect. The next subgroup is the *Scripting* group which allows the user to record a series of button clicks thus creating a movie to be played back. Typically one records the actions from the *View* group which allows the user to rotate, translate, and scale the portrait in a precise manner. The *Face Views* section is a simple pair of up-down buttons which allows the user to view all six of the viewing cube's faces in an easy manner. The final group is the *Files* group which allows the user to specify the load/save files and the source and routine names.

The plot window allows for interactive viewing of the portrait much like the *View* group above, but in a less precise but faster manner. The left mouse button controls scaling, while the middle mouse button controls rotations and the right mouse button controls translations. All operations are made with the given mouse button down and moving the mouse on the pad in the appropriate direction. Z axis rotation and translation may be obtained by holding the shift key down and performing the normal rotation or translation. Pressing the c key will recenter the portrait.

Example C Program

The following is an example C program which will show the format necessary to be linked in and run by the remote computational server. The routine takes three arguements with the first being *flag* which is zero for the initialization call and one for calls to get the state equations (i.e. if flag = 0 then the routine should do any initialization it needs to do. When flag = 1 the value for x sould be set). The second arguement is the state variable *x* which is of length *len* and is the return information for the system.

```
#include <math.h>

test(flag, x, len)
int flag, len;
double *x;
{
register i;

        if (flag == 1)
          for (i = 0; i < len; i++)
            x[i] = exp(sin(x[(i+1)%3])*x[(i+2)%3]);
}
```

AUTHOR

Raja R. Kadiyala, Dept. of EECS U.C. Berkeley. *email: raja@robotics.berkeley.edu*

BUGS

The error checking for the client processes aborting is poor. A close eye should be kept on the window which sys_view was started in to locate possible problems. There is no error checking on the validity of the subroutine specified in source_file. If sys_view dies for some reason the remote process (named create_x) may still be running.

FINAL REPORT ON GLUEHEDRON

by

Joseph Maurice Rojas

Computer Science 285

Prof. Carlo H. Séquin

May 15, 1991

1 Description

GLUEHEDRON is a fast, simple tool for gluing together polyhedra in a regular fashion. Through a few simple heuristics, GLUEHEDRON provides a *canonical* method for such gluings. Thus if polyhedra are viewed as abstract symbols, GLUEHEDRON can be seen as a tool for generating a regular grammar of polyhedra. The goal of this tool is to provide a playground for experimentation in this grammar, i.e., a playground for constructing polyhedral fractals and families of polyhedral "gluings."

GLUEHEDRON originally evolved from a fractal generation tool named GROHEDRON. The latter tool simply took one polyhedral shape and appended self-similar copies of itself according to some user-specified parameters. GLUEHEDRON is more general in the sense that more than one polyhedron can be used in the gluing process. The present tool is also well suited for use in custom-tailored shell-scripts for generating complex objects with a specific structure. For instance, one could write

a shell-script invoking GLUEHEDRON to construct trees with a specific branching pattern.

It thus seems useful to have a canonical method for attaching polyhedra, but then how does one define "canonical." The algorithm used by GLUEHEDRON makes a plausible definition in terms of incircles, and the resulting images reveal that our method is quite reasonable. We now describe our algorithm in greater depth.

2 Algorithm

The cornerstone for our gluing algorithm is a "glue-list." A glue-list is simply list of pairs that specifies which faces are glued to which. Bearing this in mind, our algorithm can be described as follows:

INPUT t = "twist" value, s = "scale" value, R = randomness flag, gluefile = glue-list, oldfile = old polyhedral shape in UNIGRAFIX format, capfile = polyhedral cap shape in UNIGRAFIX format

OUTPUT newfile = new polyhedral shape in UNIGRAFIX format

($G1$) Find the inward normals, incircles, max vectors, and tangency angles of the "from" faces. The "from" faces are simply the faces of oldfile which will be glued.

($G2$) Find the inward normals, incircles, max vectors, and tangency angles of the "to" faces. The "to" faces are simply the faces of capfile which will be glued.

($G3$) Find $\{L_j\}$ - the set of all affine transformations moving a "to" face, T, to a "from" face, F. Moving T to F in this context means that their incenters coincide, their max vectors lie on the same ray, and their inward normals lie on opposite rays.

($G4$) For any pair of "to" and "from" faces, T and F occuring in gluefile, glue a copy of capfile to oldfile using the appropriate L_j. Further scale the transformed copy of capfile so that the max vector of T has the same length as the the max vector of F. Then scale down again by a factor of s. Now rotate capfile by an angle of $t\alpha$ where α is the tangency angle of F and scale one last time by a factor of $cos(alpha)/cos((1-t)\alpha)$.

We must of course define what we mean by "max vector" and "tangency angle." These definitions relate the boundary of a polygon to the polar coordinates defined by its incenter. Let P be a convex polygon in $\mathbf{R}^3$ and c its incenter. Then the *max vector* of P is simply $x - c$ where x is the point of ∂P farthest away from c. The *tangency angle* of P is simply half the angle between the tangents of the incircle of P through x. Though these definitions perhaps seem ad-hoc and abstract, they actually make good sense geometrically and are based on the intuitive notion that "less over-hang looks better" when taking the union of two polygons.

3 Running

The source code for this tool is in the file gluehedron.c in the FINAL directory of the author's torus-cluster account. The author's login is c285-ak. The author is currently

a student in the Ph.D. math program at U.C. Berkeley and may be reached via e-mail at rojas@math.berkeley.edu. He will gladly answer any questions relating to this project.

The FINAL directory also contains a makefile for simple compilation, a copy of the man pages of GLUEHEDRON (gluehedron.man), and a latex copy of this report (glue.tex).

4 Graphics

The following postscript print-outs briefly illustrate the myriad shapes that can be generated with GLUEHEDRON. For convenience sake, we list the files necessary for building the first shape.

5 Conclusion

This project was a great learning experience for the author in that he had almost no prior knowledge of C or UNIX. Writing the code for GLUEHEDRON gave the author an interesting crash-course in software for survival. Nevertheless, the author would like to add the following few remarks:

A fundamental portion of the code was devoted to exploiting the power of UG3 - a tool for manipulating the geometric data structures defined in UNIGRAFIX. From coding this project, the author learned that UG3 is indeed quite powerful - but slightly lacking. What would have made this project this project almost trivial is a few macros for simple UNIGRAFIX file manipulation. Such macros do exist (such as

UGcopyfile, etc...) but certain very intuitive ones are lacking, e.g., UGrenamestmt. Thus with the right macros, the code could have been done within mere hours.

Another large portion of the code was devoted to using MAT3 - the matrix manipulation library. This library was quite useful, but poorly documented. For instance, it took a few dozen errors before the author realized that the MATangle function returned angles in degrees. This fact was never documented.

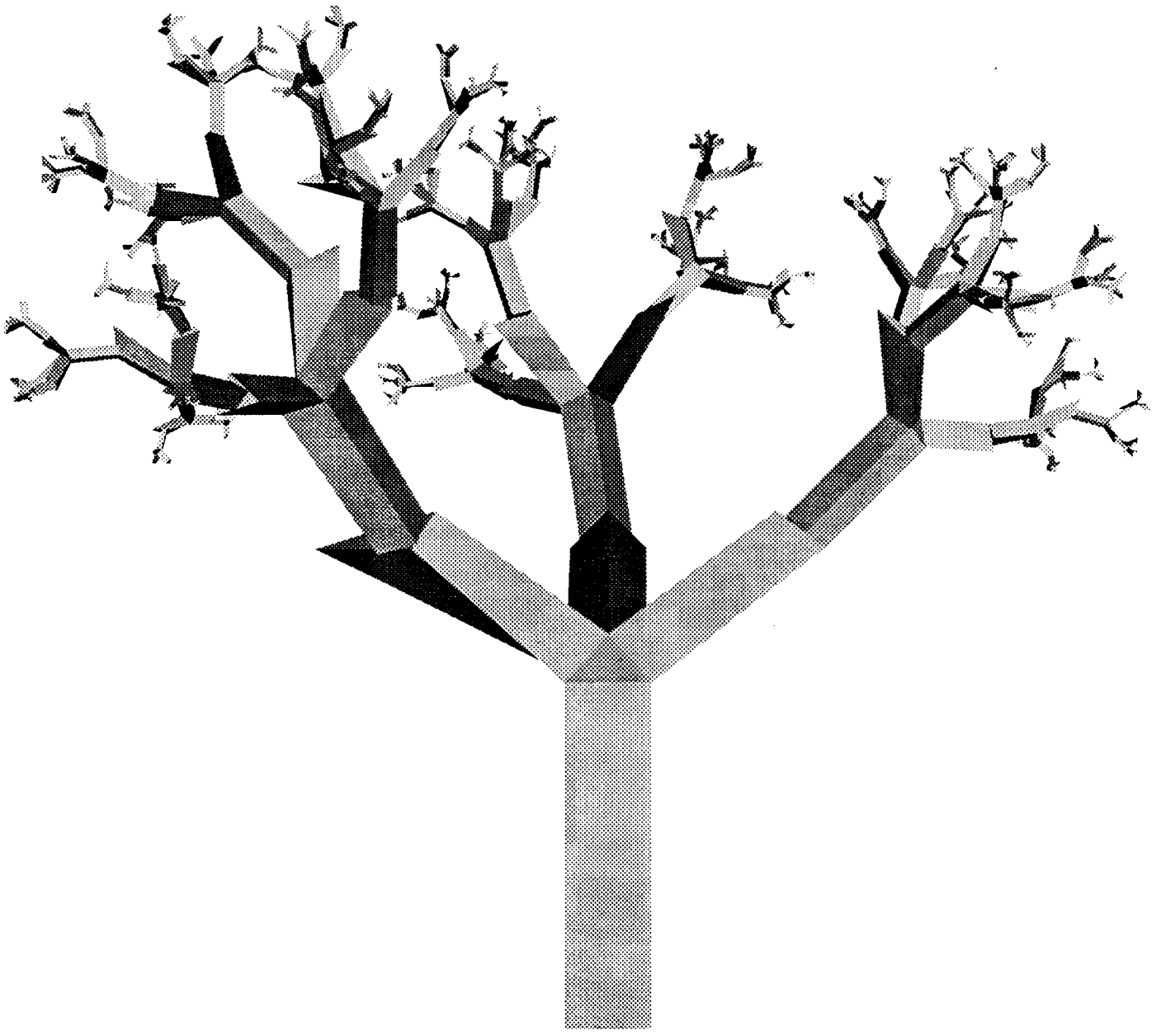
As far as future directions and improvements, there are many. One can certainly write shell-scripts for interesting special-purpose applications of GLUEHEDRON but it would be nice to have a small library of the common applications, e.g., fractals and spirals. Also, the code for GLUEHEDRON is a bit hackish in the sense that it was written quickly without much thought for memory usage. Efficient memory allocation becomes particularly important for generating beautiful large fractals quickly. These are concerns the author would have truly liked to address more strongly. However, for the present, GLUEHEDRON is quite fun and provides the groundwork for a greater systematic study of polyhedral grammars.

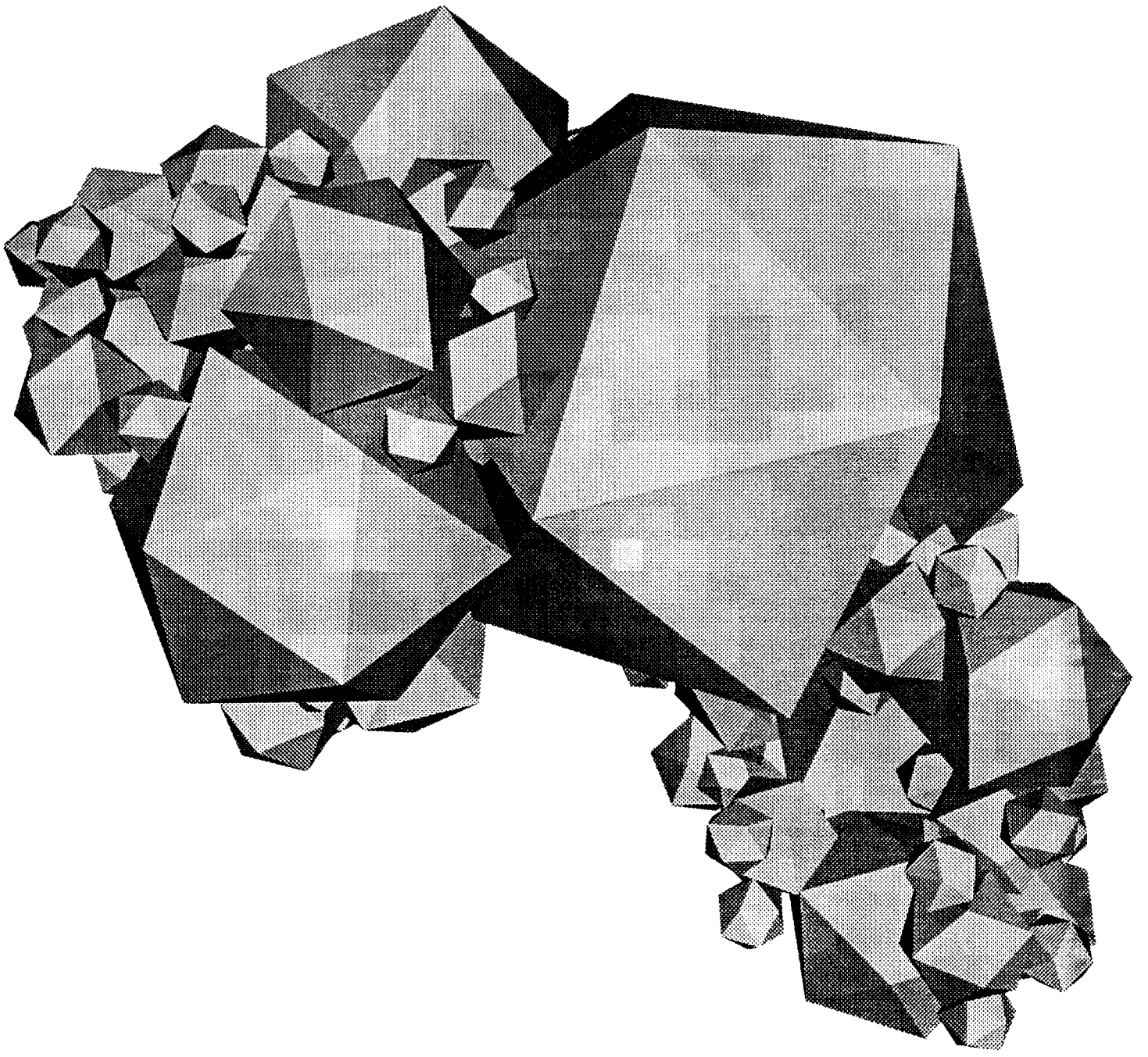
gluehedron -t .5 -g gluetree -o tree0 -c tree0 > tree1

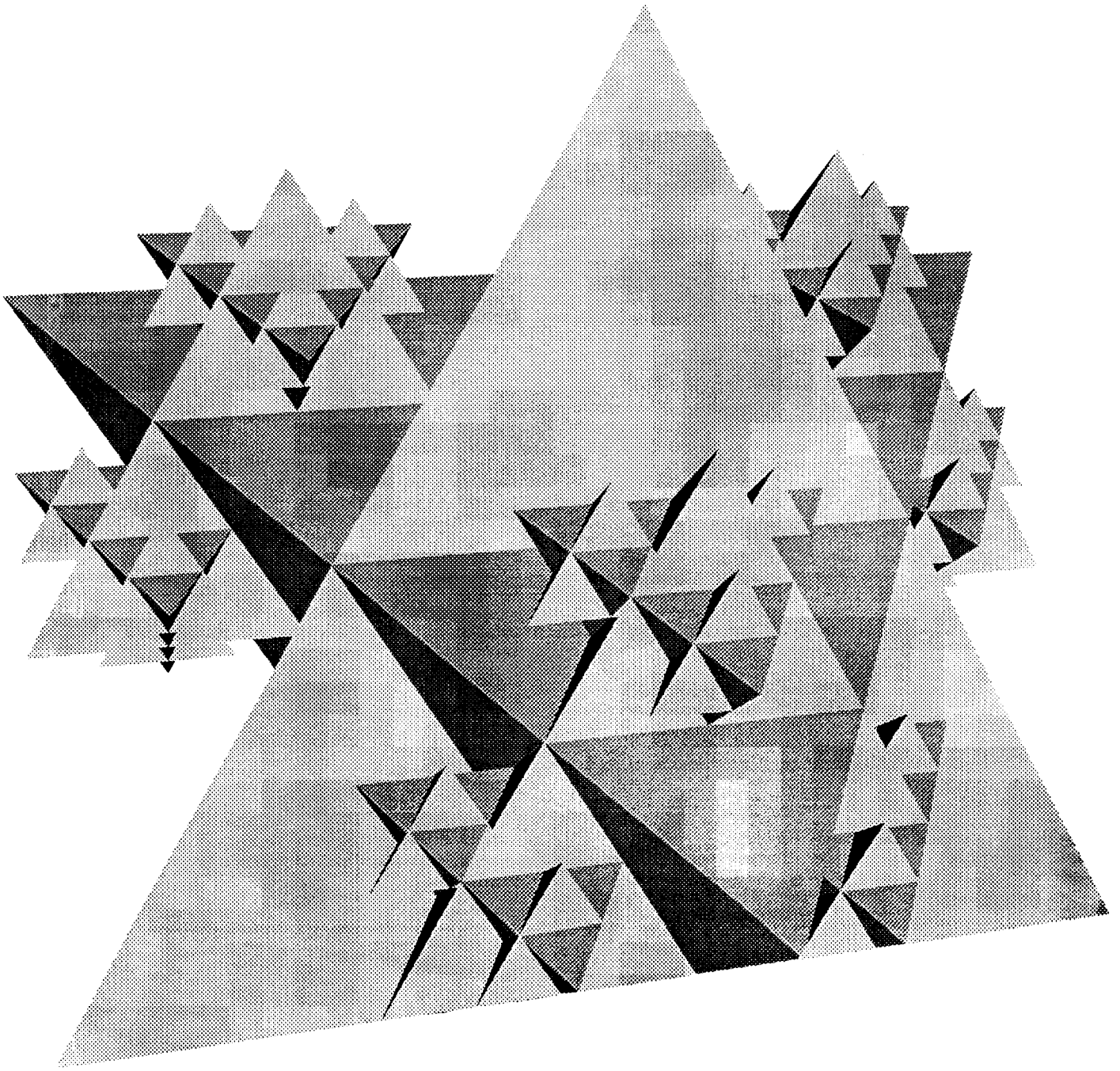
" " " " " " -c tree1 > tree2

gluehedron -t .5 -g gluetree -o tree0 -c tree0 > tree1
gluehedron -t .5 -g gluetree -o tree0 -c tree1 > tree2
gluehedron -t .5 -g gluetree -o tree0 -c tree2 > tree3
gluehedron -t .5 -g gluetree -o tree0 -c tree3 > tree4

Tree 4







Joseph Maurice Rojas
CS 285
Prof. C. Sequin
May 15, 1991

NAME

gluehedron - Glues input polyhedra in a visually pleasing way, according to certain specified user parameters.

SYNOPSIS

grohedron [-t twist] [-s scale] [-R] [-g gluefile] [-o oldfile] [-c capfile] > newfile

DESCRIPTION

This tool takes the UNIGRAFIX description of 2 polyhedra and creates a "glued" output polyhedron in UNIGRAFIX format. The algorithm takes an old shape and glues on scaled rotated copies of a cap shape face-to-face. The user can specify exactly which faces are glued to which, and the resulting new shape is written to a UNIGRAFIX file. This simple process can be repeated many times with different objects (perhaps with the aid of shell-scripts) to produce interesting shapes such as spirals or fractals. The possibilities are endless and the options are as follows:

-t Faces are glued in a canonical manner. The variable "twist" introduces an additional rotation of angle "twist" on each glued cap shape to produce a more interesting shape. The default "twist" value is 0.2 .

-s Introduces an additional scaling by a factor of "scale." The default "scale" value is 1.0 .

-R Negates the -t option and introduces a random twist.

-g Allows user specification of the gluing: a small rotated copy of the cap shape is placed so that the b_i face of the copy lies flat on the a_i face of the old shape. This is done for for all i from 1 to n. The glue file must be of the following format:

a_1 b_1
a_2 b_2
.
.
.
a_n b_n

The ith face of an input polyhedron is simply the ith face occuring in its unigrafix description. If no file is supplied, then gluehedron assuems a default of (a_1 , b_1) = (1 , 2). The a_i are referred to as ''from'' indices, while the b_i are referred to as ''to'' indices.

EXAMPLES

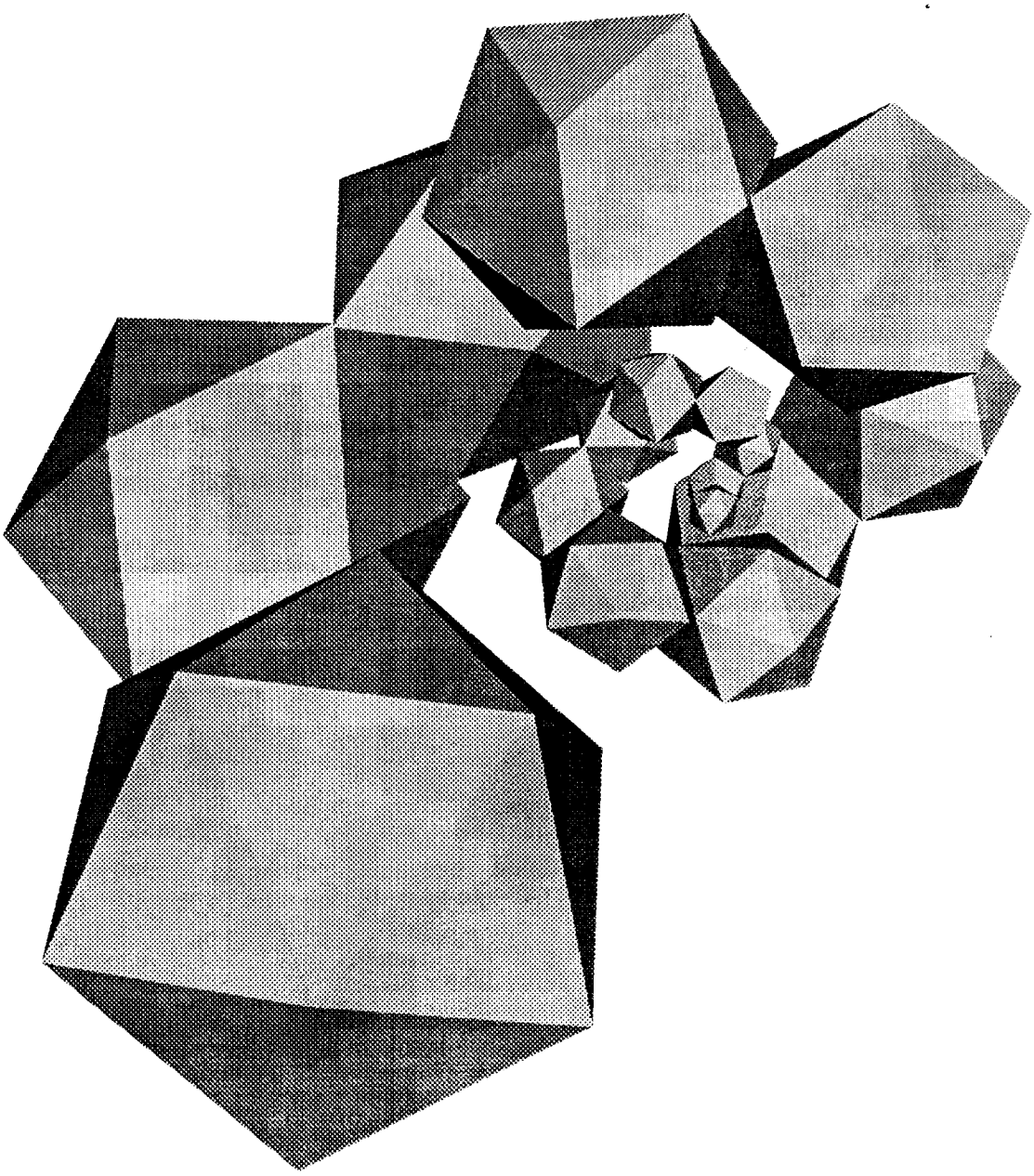
```
grohedron -t .3 -s .9 -g gluetree -o tree0 -c tree0 > tree1
grohedron -t .3 -s .9 -g gluetree -o tree0 -c tree1 > tree2

grohedron -t .5 -s .2 -g glueico -o icosa -c tree2 > planet
```

WARNINGS

The gluing algorithm is based on the assumption that all polyhedral faces are convex. Also, there is a minor bug which appears only if there is a polyhedral face which lies on a plane that goes through the origin. The first restriction is fairly minor, while the latter can be fixed by a generic perturbation, i.e., translating the vertices by a small random amount. Lastly, the ''from'' indices in a glue file must occur in increasing order.

For more information, please see the latex file gluehedron.report .



Final Project Report for

UGTREE

May 15, 1991

Laura Downs

Purpose:

The function of *ugtree* is to generate UNIGRAFIX descriptions of trees which are realistic in appearance and yet reasonable to render. The tree generation should occur quickly. Also the method of description should be understandable to the user. To achieve realism, I believe that the user should have a lot of control over the description of the tree and therefore I allow the user to set any of the shape parameters that *ugtree* uses to determine the tree.

Design:

The execution of *ugtree* follows these steps:

- If an option file is specified, read in the options
- Generate an interior representation of the tree
- Generate a UNIGRAFIX description of the tree

Reading the Option File:

Before reading the option file, all of the parameter names are entered into a hash table. Then the scanner reads in tokens from the option file and looks up their codes in the hash table. After a parameter has been identified and if it is not a flag, the scanner reads its new value. There are a large number of parameters and it is very easy to add more so the hash table does speed things up considerably. Comments can be included in the option file by starting a line with a '#'. Any errors in the option file are reported to stderr. If the option file contains errors, a tree description is not generated.

Generating the tree representation:

This is the meat of the program. A record is created for each branch segment which contains its length and radius, the number of leaves along the segment, the number of branches off the segment, and a turtle representation of its orientation consisting of an axis, a left vector, and an up vector, . These values are computed in the routines `grow_branch()` and `grow_tree()`. For each branch off of a segment, the length is

calculated by the length of the parent segment $\times$ a contraction ratio. The radius is calculated by the radius of the parent segment $\times$ a width ratio. The orientation is calculated from the parent segment's turtle by rolling it along its axis by a divergence angle for the branch and then twisting it around its up vector by a branching angle. The orientation may also be affected by **flattening** - which overrides the divergence rotation - or by **tropism** - a global force on the tree in the direction of gravity or wind or some other unidirectional force.

Tropism:

The tropism vector represents an overall force on the tree and so is useful for making trees like a weeping willow or a cypress. The actual computation on each branch segment is to rotate the turtle toward the tropism vector by an angle equal to the magnitude of the cross product of the branch axis with the tropism vector scaled by the ratio of the length of the branch to its area. This computation has the effect that branches growing orthogonal to the force and which are thin will be affected most strongly by the force.

Branch Flattening:

The branch flattening imposes a horizontal branching constraint such that at any branching juncture the two branches lie in the plane which is closest to horizontal of any plane through the parent branch. The computation is to roll the turtle so that its left vector is orthogonal to the z-axis. This is calculated only on the branches, not on the trunk.

Terminating a Branch:

A branch is terminated by one of two constraints. If it is shorter or thinner than the minimum size (designated by the option "`twig`") the branch is ended. This creates reasonably natural branch terminations. The other limit to branch growth is a maximum depth of branching. This is mostly useful to limit the complexity of the tree description. If the level of branching is K, the maximum number of final branches will be 2^K in the case of binary branching or 3^K in the case of ternary branching. Since this growth is exponential, K should be kept small if you want an effective cap.

Creating Leaves:

To specify a leaf description, the option "`leaf_file`" should be set to the name of a file which contains a UNIGRAFIX description of a structure called "`leaf`". This file will be included in the final UNIGRAFIX description file of the tree. The *ugtree*

program does no processing whatsoever on the leaf file, not even to check if it contains the needed definition.

In the tree description, one leaf is always created at the end of each final branch. Leaves may also be created along the branches of the tree by setting the parameters `"leaves"`, `"leaf_level"`, and `"leaf_radius"` which control the number of leaves per branch, the branching level at which leaves start and the branch radius at which leaves start. The actual placement of the leaves along the branch is calculated when the UNIGRAFIX description is generated.

Generating the UNIGRAFIX description:

Wire description:

A UNIGRAFIX wire description of a tree can be generated by setting the number of sides of the prisms to a number less than 3. If the level is 1 or 2, the description will include leaves. This is particularly useful for designing new tree descriptions because a picture of the output can be created almost instantaneously in which such variables as branching angles, divergence angles, contraction ratios, tropism and flattening can be observed easily.

Level 2:

This is the most general representation of the tree. Every branch segment is represented as a prism (8-sided default) with a cap on the top. The ends of its child branches will be embedded in the cap. This creates reasonably nice pictures of most trees. All leaves are included in this representation.

Level 1:

This is much the same as Level 2 except that the branches which are smaller in radius than 4 times the minimum radius are represented as wires. Because of the exponential growth in the number of branches of each smaller radius, this effects most of the branches. This representation can be rendered by ugiris in about 1/3 the time of the Level 2 representation.

Level 0:

One branch segment between each level of branching is created up to the third level of branching. Then one big green segment wider than the actual branch is created

extending to the tip of the branch to represent leaves. This representation is only good at a very great distance.

Getting help:

Command Line Options:

To get a list of all of the command line options, just call *ugtree* with an illegal command line option (like most programs).

Option Files:

Call *ugtree* with the command line option `-h` to get a list of all of the possible option file parameters along with a description of each one. Either redirect this into a file or copy an existing tree description file and then change the parameter values that you want to change.

Extending ugtree:

Adding Parameters:

This is very easy. To add a new parameter to *ugtree*, you just have to include its name, type, and a description of it in the `parameters.c` file, include it in the enumerated type `OPTCODE`, and write whatever code is necessary to achieve the desired effect of that parameter. I will include a more detailed description of this in a `README` file along with the program.

N-ary branching:

One addition I would like to make to ugtree is the ability to generate trees with more than ternary branching. Also it would be useful to be able to specify different degrees of branching for the trunk and the branches. Neither of these would be difficult. The second requires the addition of one new parameter : "`branch_n`" with the flags "`branch_binary`" or "`branch_ternary`" and using that parameter in the routine `grow_branch()`. The other requires more decisions. If any number of branches can be specified then it is not reasonable to specify a branch angle, divergence angle, and contraction ratio for each one. Perhaps they could all use the same branch angle and contraction ratio after the third branch and increment the divergence angle by some fixed amount.

Creating Better Tree Descriptions:

I would like to have a representation of the tree in the spirit of ugworm. The main problem is that at each juncture there might be four prisms meeting, all of different radii. However, if I could create such a representation, then the resulting description would contain about 1/4 the number of vertices and about 1/2 the number of faces. Also the faces in different segments would share vertices so that Gouraud shading would make the tree look much nicer. As it is, Gouraud shading makes my trees resemble sausage links.

Printing Directly:

I could make my program generate Iris graphics calls to render the tree directly. This would make the tree viewing much faster. It would require that I break down my segment structures and perform all of the transformations on them - which is not and unreasonable amount of work. It would also require me to parse the leaf description file and generate some internal representation of that and I would need to borrow the ugiris interface for viewing the object from different rotations and distances. Perhaps this summer I will have some time to work on this,

Texture Mapping:

Well it would sure be cool if UNIGRAFIX had texture mapping for tree bark and other fun things ...

Conclusions:

The actual process of generating the tree is pretty simple. Once the parameters are specified, the tree generation is mostly a lot of easy math. Most of the work came up in reading and interpreting the option files and in generating the UNIGRAFIX description. I think that my solution to reading the option files was quite satisfactory since it gives the user a lot of flexibility in creating option files and it will work quickly even if large numbers of options are added to the program.

I would really like to have better representations of the trees. The solution I used gives reasonable descriptions of trees in most cases but it is not very efficient in the number of faces and vertices and it could definitely be improved upon. I would like to continue to look for a solution to joining the prismatic branches in a nice way.

The problem of generating trees is only a small subset of the problem of generating plants. If what we want is a set of plant generators so that we can create an entire landscape, more than just my tree generator is necessary. A library of different botanical generators

could be used to describe any garden using grammars, growth models, and other methods. Using L-sytems, a wide variety of plants could be generated. I think that a program which understood L-system grammars with a reasonable set of primitives could be used to generate some very interesting plants - including all of the trees which can be created with *ugtree*. I think it would entail a lot of work to make that fully general but it would be very interesting. There are several models for generating leaves and flowers which could be added for more detail as well. L-systems can also be used to model plant growth which can lead to some interesting animations or which can be used to generate plants which appear to have grown naturally and plants, like vines, which interact strongly with their environment. Some more unique generators, such as fractally generated ferns could also be included. Grass and other ground cover plants is a somewhat different issue since you don't in general want to model it explicitly, but to give an impression of grass, moss, or flowers underfoot. Much work has been done in this area. All we have to do is implement it.

Finding ugtree:

The files for *ugtree* are in the directory `~laura/tree` on the torus cluster. They include the `Makefile` and the `README` files.

References:

Aono M., Kunii T., "Botanical Tree Image Generation", IEEE Computer Graphics and Applications, vol. 4, no. 5, May 1984

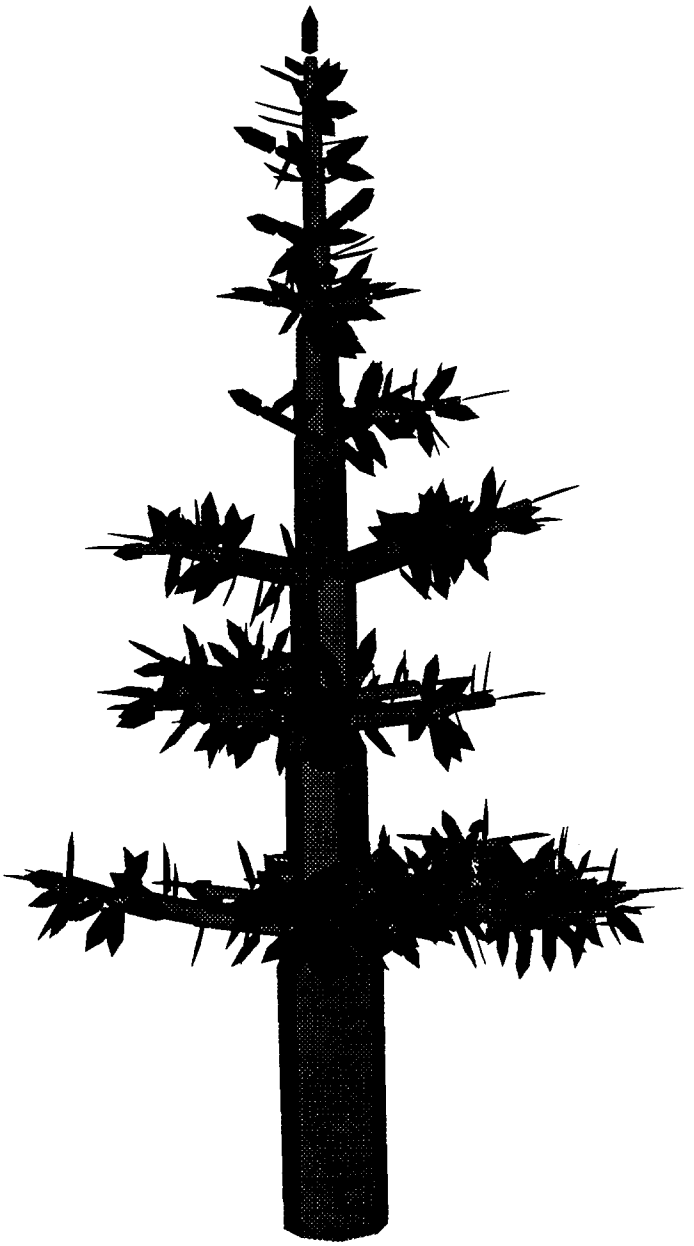
Prusinkiewicz P., Lindenmayer A., *The Algorithmic Beauty of Plants*, 1990

Reffye (de) P., Edelin C., Françon J., Jaeger M., Puech C., "Plant Models Faithful to Botanical Structure and Development", Computer Graphics, vol. 22, no. 4, Aug 1988

Smith A., "Plants, Fractals, and Formal Languages", Computer Graphics, vol. 18, no. 3, July 1984

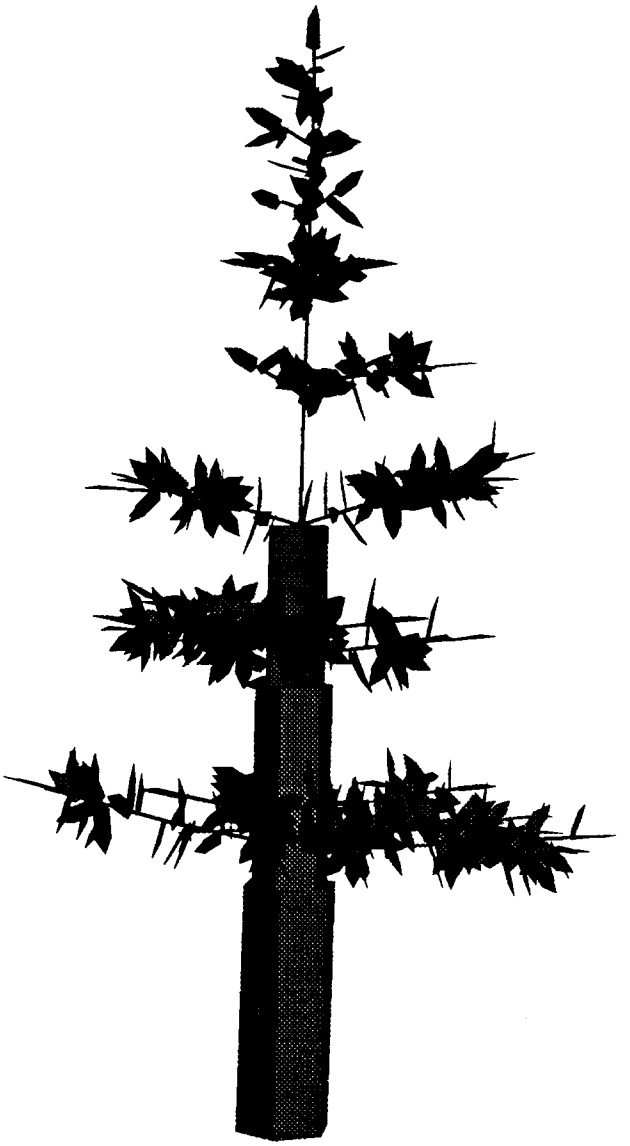
PINE
level 2

PINE, level 2



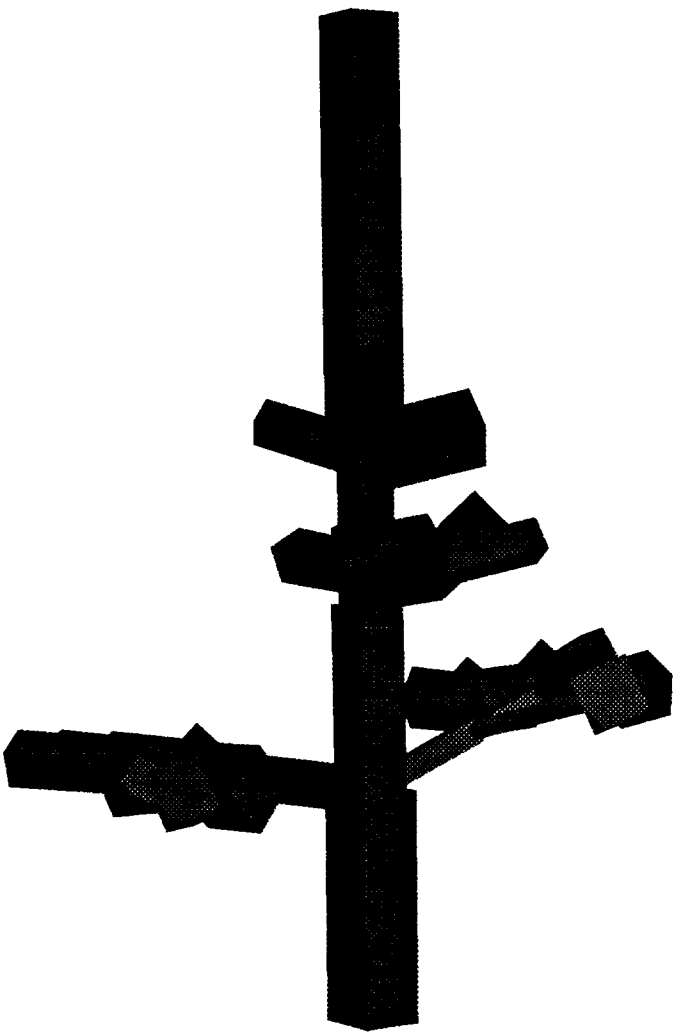
PINE
level 1

PINE, level 1



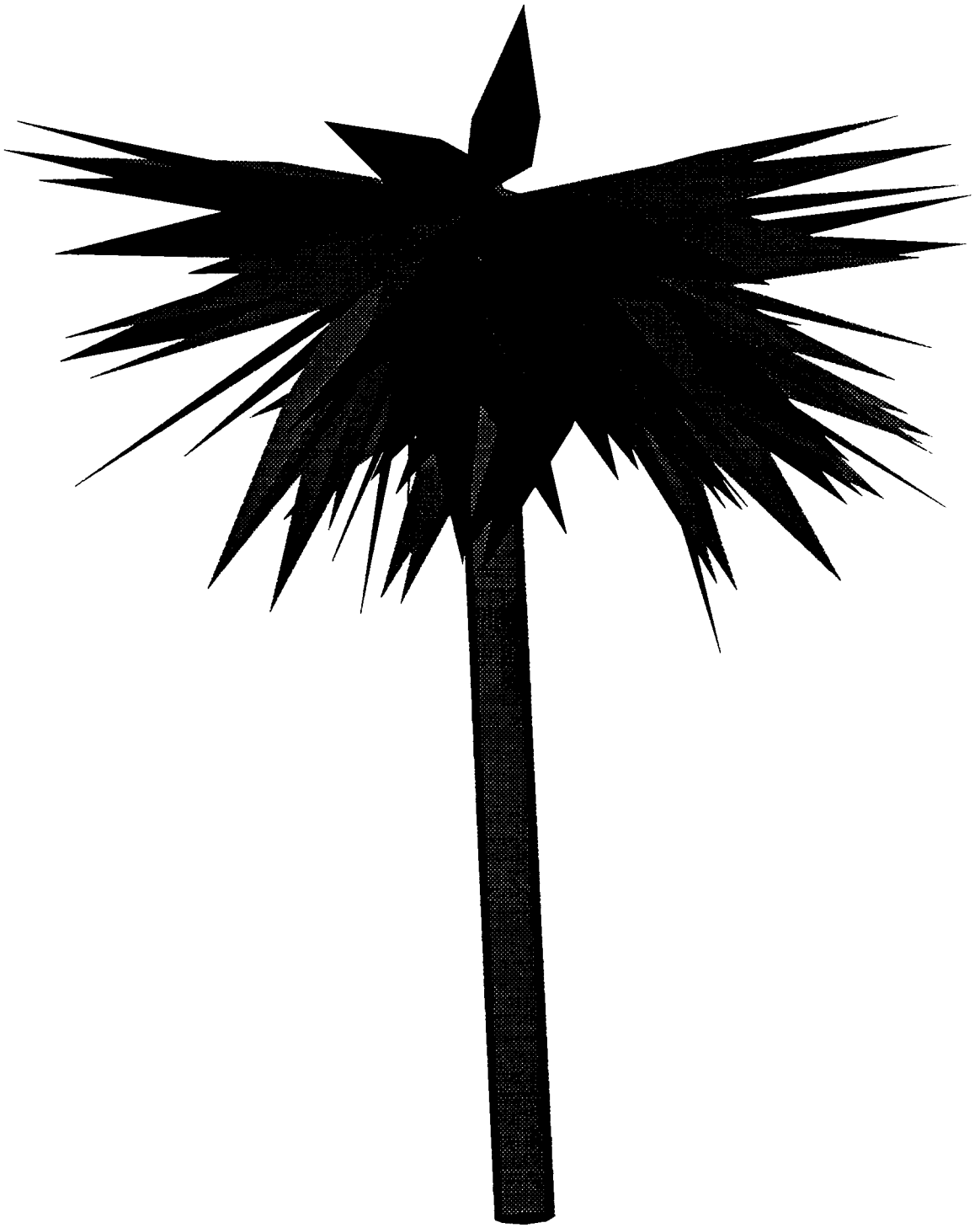
PINE
level 0

PINE, Level C



PALM1
level 2

PALM, Level 2



OAK
level 2

OAK, Level 2

NAME

ugtree - generate a UNIGRAFIX format description of a randomly created tree

SYNOPSIS

ugtree [options]

DESCRIPTION

Ugtree is a generator program for a UNIGRAIX description of trees consisting of prismatic branches with polygonal leaves.

Ugtree takes a few parameters on the command line which relate to the final UNIGRAFIX description of the tree and many parameters which may be set in an option file which is specified on the command line.

Command line options:

-h

This option causes a help file to be printed to the screen which describes all of the command line options as well as a description of the format of the option files and the parameters which may be specified.

-f <filename>

This option specifies the name of the option file which contains the set of parameter values for the desired tree description.

-o <filename> Default: stdout

This option specifies the name of the file which will contain the final description of the tree.

-l <0-2> Default: l = 2

This specifies the level of description of the tree.

0 - Only the trunk and first level of branching will be shown with only a couple of polygons to suggest the shape of the tree.

1 - The trunk and the first few levels of branching will be shown with a somewhat better suggestion of the location of further branches and leaves.

2 - All of the branches will be shown and leaves will be represented as individual polygons.

-s <integer> Default: s = 1

This specifies the seed for the random number generator. Two trees created with the same parameter file, the same height, and the same seed will represent different descriptions of the same tree even if the level of representation or the number of sides to a prism is different.

-n <integer> Default: n = 4,6,or 8
This specifies the number of sides that each prism section should have. There is a built-in limit of 24. If n is specified to be less than 2, the output will be the UNIGRAIX description of a wire along the axes of the branches of the tree. The default value depends on the level of representation.

-H <real> Default: H = 25 feet
This specifies the height of the tree. Other default heights may be specified in the option file, but the command line specified value takes precedence.

The final tree description will be a UNIGRAFIX format file which describes the branches as prisms which join with mitred corners in the spirit of mkworm.

These are most of the parameters which can be set in a ugtree option file. Not every parameter need be set in any file. Many of them will override each other such as 'binary' and 'ternary' branching. For a more current list type: "ugtree -h"

```
#########
# flags #
#########


binary              # Causes binary branching

ternary             # Causes ternary branching

trunk               # Causes branches off of the trunk to
                    # follow different shape parameters
                    # and rules than branches off of other
                    # branches

no_trunk            # Causes all branches to follow the same
                    # parameters
                    # The basic parameters will be used
                    # i.e. alphal instead of branch_alphal

flatten             # Causes every branch to lie flat to
                    # gravity before branching which causes
                    # the branches to split horizontally

no_flatten          # Causes branches not to be flattened so
                    # that they can split at any angle


############################
# string valued parameters #
############################

leaf_file = "OAK_LEAF"
                    # The file "OAK_LEAF" will be included in
                    # the final description and should contain
                    # a UNIGRAFIX description of the structure 'leaf'
```

```
##############################
# integer valued parameters #
##############################

seed = 5          # Seed the random number generator with
                  # the number 5

max_depth = 12    # Do not create more than 12 levels of
                  # branching

leaves = 10       # 10 leaves will grow off the side of each
                  # branch after leaves start

leaf_level = 3    # leaves will begin to grow off of branches
                  # after the third level of branching


#################################
# real vector valued parameters #
#################################

tropism = ( 0, 0, -1 )
                  # Let the tropism vector be in the negative
                  # z direction ( down )
                  # All branches will tend in this direction.
                  # The larger the magnitude of T, the more
                  # the branches tend that way.

color = ( 1, 0, 1 )
                  # An intensity-hue-saturation triple.
                  # The trunk will be this color.

green = ( 1, 120, 1 )
                  # An intensity-hue-saturation triple.
                  # The leaves will be this color.

#########################
# real valued parameters #
#########################

height = 12       # The tree will be about 12 feet tall

radius = 0.5      # At 12 feet in height, the radius of the
                  # trunk will be 0.5 feet

length = 0.3      # The initial branch segment will be 0.3 feet long

twig = 0.03       # Any branch under 0.03 feet in radius will end
                  # in a leaf

leaf_radius = 0.1
                  # Any branch under 0.1 feet in radius will have
                  # leaves along its sides.

leaf_distance = 0.1
                  # The ratio of the length of a branch segment to
                  # the number of leaves on it will be at least 0.1
```

```
##################################################
# real valued angle parameters between -180 and 180 #
##################################################

twist = 5          # The trunk and branches will twist by 5 degrees
                   # on any segment where there is no branching.

leaf_angle = 30 # Leaves grow from the side of a branch at 30 degrees

bend = 10          # The trunk will bend by 10 degrees on any segment
                   # where there is no branching

alpha1 = 2         # The trunk, first, and second branches will
alpha2 = 35        # create 2, 35, and 27 degree angles respectively
alpha3 = 27        # with the axis of the previous trunk segment

diverge1 = 76   # The trunk, first and second branches will be
diverge2 = 178  # rotated radially around the trunk by 76, 178
diverge3 = -57  # and -57 degrees respectively

branch_bend = 10      # A branch will bend by 10 degrees on
                      # any segment where there is no branching

branch_alpha1 = 26    # The first, second, and third branches will
branch_alpha2 = 37    # create 26, 37, and 57 degree angles
branch_alpha3 = 57    # respectively with the axis of the
                      # previous branch segment

branch_diverge1 = 12     # The three branches will be rotated
branch_diverge2 = 118    # radially around the trunk by 12, 118,
branch_diverge3 = 235    # and 235 degrees
```

```
##########################################
# real valued parameters between 0 and 1 #
##########################################

prob = 0.3      # The probability of branching occurring at any
                # segment is 0.3

width_sigma = 0.1       # Any width ratio may vary up to 10%
alpha_sigma = 0.2       # Any branching angle may vary up to 20%
diverge_sigma = 0.35    # Any divergence angle might vary up to 35%
leaf_sigma = 0.3        # The number of leaves on a branch and the
                        # angle at which leaves grow from a branch
                        # might vary up to 30%

contract1 = 0.96  # Each trunk segment will be 0.96 times as
                  # long as the previous trunk segment.
                  # A good value for this is (width1)^(prob)
contract2 = 0.87  # The first and second branch off of each
contract3 = 0.5   # trunk segment will be 0.87 and 0.5 times
                  # as long as the trunk segment.
                  # Similarly (width2)^(prob) and (width3)^(prob)
                  # are good values for these.

width1 = 0.8  # The trunk will decrease in radius by 0.8 at any
width2 = 0.6  # branch and the first branch have a radius 0.6 times
width3 = 0    # the radius of the previous trunk segment.  This
              # choice of values is assuming binary branching.
              # You must be sure that:
              # (width1)^2 + (width2)^2 + (width3)^2 = 1

branch_contract1 = 0.93 # The first, second, and third branches
branch_contract2 = 0.7  # off of each branch segment will be
branch_contract3 = 0.6  # 0.93, 0.7 and 0.6 times as long as the
                        # branch segment.
                        # (branch_width1)^(prob),
                        # (branch_width2)^(prob) and
                        # (branch_width3)^(prob) are good values
                        # for these.

branch_width1 = 0.8     # The first branch will decrease in radius
branch_width2 = 0.6     # by 0.8 and the second by 0.6.  This choice
branch_width3 = 0       # is assuming binary branching.
# You must be sure that:
# (branch_width1)^2 + (branch_width2)^2 + (branch_width3)^2 = 1
```

EXAMPLE

```
ugtree -f PINE -f mypine -H 30 -l 2 -n 12 -s 234
```

will generate a 30' pine with all branches explicitly described as 12 sided prisms and leaves described as polygons. The output file is called 'mytree' and the number 234 will be used to seed the random number generator. This assumes that the file PINE contains the parameter values (such as branching angles) of a pine tree.

FILES

~laura/tree/ugtree

SEE ALSO

mktree (UG)

BUGS

Let me know if you find any!

AUTHOR

Laura Downs