GENERATION OF TIME-VARYING GEOMETRICAL MODELS

Carlo H. Séquin

Computer Science Division, EECS Department
University of California, Berkeley, CA 94720

With contributions by:

Paul-Henri Arnaud, John Boreczky, Thurman Brown, Mark Brunkhart, Steve Burgett, Roger Bush, Oliver Crow, Paul Debevec, Mitchell Deoudes, Christopher Fuselier, Raph Levien, Allan Christian Long, Peter Lorenzen, Brian Mirtich, Nobuko Nathan, Luis Porcelli, Daniel Rice, Saba Rofchaei, Adam Sah, Sun-Inn Shih, Maryann Simmons, Mohammed Amin Vahdat, Dan Wallach, Robert Wang, Steve Yen, Louis Yun.

ABSTRACT

This is a report on the sixth 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. The projects described include: demonstrations such as a constant-velocity universal joint, a 3-ball juggler, or the muscles and bones associated with a human elbow; interactive objects such as jitterbug mechanisms or a 3-dimensional maze; generator programs for patterning a stone wall or for growing evolving plant models; and utilities such as an efficient convex hull generator or an interactive display program for 4-dimensional objects. Most projects have been developed on SGI personal IRIS workstations, and the geometric descriptions of the objects use the Berkeley UniGrafix language.

Table of Contents

CS 285, COURSE OVERVIEW

This is the sixth offering of a special graduate course concerning the procedural generation of computer graphics models. The course has evolved considerably since its first offering in the Fall of 1983 under the title "Creative Geometric Modeling"[1]. At that time, the students developed some new generator and modifier programs in the UniGrafix framework[2] and used them 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 encountered 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.

In the 1990's, after our graphics class laboratory was modernized through a donation of Personal Iris workstations from Silicon Graphics Corporation, the emphasis of the course shifted to more interactive techniques and the use of visual feedback during program development[4]. This instantaneous graphical feedback evokes an extra level of enthusiasm and motivation, and the results achieved by the students are correspondingly high.

In 1992, this interactive aspect was stressed even more, primarily because many of the assignments and final projects involved time-varying interactive objects. As in the past, the course had more formal homeworks during the first half of the term and concentrated on individual course projects during the second half.

The class was larger than ever! Thirty students handed in the first homework assignments. To moderate the load on the graphics lab – which was shared with another class –, to reduce the programming chores for the students, and to make supervision and mentoring of the projects somewhat easier during the final weeks of the course, most assignments were done in teams of two or sometimes even three students. A prescribed rotation in the composition of these teams guaranteed that every student met almost every other student in the class as a partner on one of the weekly assignments.

To cater to individual tastes as much as possible, the final project could be done in teams of two or individually, and the scope of the project could be chosen to extend over 5 weeks or only over three weeks; in the latter case those students were given two additional formal weekly assignments.

I would like to thank all the participants for their energy and for their enthusiasm which made this course a particularly intense and enjoyable experience.

Carlo Séquin

---

1. C. H. Séquin, "Creative Geometric Modeling with UNIGRAFIX," T.R. UCB/CSD 83/162, Dec. 1983.
2. C. H. Séquin and K. P. Smith, "Introduction to the Berkeley UNIGRAFIX Tools, Version 3.0," T.R. UCB/CSD 90/606, Nov. 1990.
3. C. H. Séquin, "Procedural Generation of Geometric Objects," T.R. UCB/CSD 89/518, June 1989.
4. C. H. Séquin, "Interactive Procedural Model Generation," T.R. UCB/CSD 91/637, June 1991.

Mechedit
An Editor and Animator for Planar Mechanisms

CS 285

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

Steve Burgett
Roger Bush

12/22/92

1 Introduction

Mechedit is an interactive graphical tool for exploring the kinematics of planar mechanisms (bar linkages). It provides an environment in which a user can construct mechanisms with links and joints and then animate them. Although the kinematics must be planar, the mechanism is displayed three-dimensionally, with the links shown as polygonal plates. Animation is accomplished by selecting one link to drive and moving it with the mouse. The entire mechanism then moves according to the kinematics defined by its joints and links.

Mechedit depends heavily on the work of Eric Enderton in his master's thesis [1]. The technique of solving bar linkages in closed form by graph contraction is taken directly from this. As such, this report doesn't belabor the topics that are so thoroughly covered there.

Program use is described in the attached man page. This report documents the technical aspects of the program.

2 The Editor

The editor is based on intuitive point-and-click operations with the mouse. All editing is done from a 3-Dimensional perspective, though the mechanism's planar 2-D nature eliminates any possible positioning ambiguity. Mechanisms can be quickly assembled, edited, and reconfigured, by dragging links to the appropriate positions on the 2-D plane. If a link is dragged by one of its *sites* and comes close to another site, it will snap onto this site forming a joint between the two links. Joints also have sites that act as *controls*. A ground joint has a small crank that can be grabbed to animate the mechanism it is attached to. A prismatic joint has arrows that can be grabbed and repositioned to change its angle and range of operation. The editor has no modes. At any point, one can grab a control and animate the mechanism attached to it. A link can then be grabbed and repositioned as desired. Several unconnected mechanisms can lie about, ready to be animated. In this sense, there is an underlying intuitiveness that is not unlike a video erector set.

2.1 Data Structures

The links and joints package is built on top of a linked list class. The interfacing between links and joints is accomplished through the notion of *sites* which correspond to the *widgets* which are selectable on the screen. These are used to do the polygonal links and the joints with their controls. These could be extended to other joint types without much difficulty.

3 Animation

Animation is divided into two major steps: *plan time* and *move time*. From the users point of view, plan time occurs invisibly and instantaneously whenever a *crank* is grabbed via the mouse. Move time occurs continuously for as long as the crank is dragged, and the animation is displayed graphically on the screen.

During move time, the positions of the links and joints must be continuously recomputed as the user drags the driven link to various positions. The purpose of plan time is to construct a plan for how these positions will be computed during move time. The order of computation is important so that when each unknown is solved for, enough information has already been computed. No simultaneous equations are solved during move time. Sequential, deterministic calculations are made in the order specified by the plan. In addition, there are many variables which remain constant during move time and only change if the mechanism is edited. These values are computed once at plan time and stored in the plan.

3.1 The Planner

The primary responsibility of the planner is to determine the order in which the positions of the joints and links can be solved for during move time. The approach is to identify portions of the mechanism for which positions can be calculated given the information available initially (i.e. the position of ground joints, the position of the driven link, and the structure of the mechanism). Then, assuming that those positions have been found, the planner looks for the next portion of the mechanism for which positions can be calculated based on this new set of known values. The planner repeats this process until it has identified the proper order for computing all the joint and link positions in the mechanism.

A key fact is that the planner can identify solvable portions of the mechanism based only on its topology and does not require geometric information (see [1]). The planner makes use of the mechanism's graph, where links are represented by nodes of the graph, and joints are represented by edges. As each step in the plan is constructed, one of the nodes of the graph represents the *known* information at this point. If there exists a three-cycle through this node, then that represents a portion of the mechanism for which a solution can be computed based on the known information. The planner inserts pointers to that portion of the mechanism into the current plan step, then proceeds to determine the next step. To do this, the graph needs to be modified to reflect that more information is now known. This is done by contracting the three-cycle of nodes into a single node by calling the `Contract()` function described below.

When the planner has completed its task, it will have constructed a *plan*, which is a linked list of small structures that contain the above-mentioned pointers into the mechanism.

3.1.1 Graph Package

The graph package is written in C++ and, is based on the doubly-linked list class in the GNU C++ library. We use the adjacency list representation of a graph rather than the adjacency matrix. This allows for multiple edges between nodes. In the mechanism this would correspond to a rigid submechanism. The graph package is abstract and has no knowledge of mechanisms, links, or joints. The graph data structure is composed of a doubly linked list of nodes and a list of edges. Each node has a list of pointers to edges. Each edge has two pointers to nodes. In addition, each node and each edge has space for one piece of data. This space is used by *Mechedit* to store a pointer. A node contains a pointer to a link and an edge contains a pointer to a joint.

3.2 The Animator

During *move time* the animator is in charge. The animator is simply a loop that repeatedly executes the *plan*. Before each execution, the animator gets the current angle of the driven joint from the user interface. It computes a transformation based on this angle and applies that to the driven link. Then the steps of the plan are executed in sequence to solve for the transformations of the rest of the mechanism. Finally, the mechanism is rendered using the computed transformations. Our approach to rendering is to not bodily move the links to their animating positions, but to just send their transformations to the graphics hardware so that they get rendered in the proper place on the screen. This takes advantage of the speed of the graphics hardware, as well as ensuring only rigid-body transformations are used to animate the mechanism.

4 Discussion

It is important to discuss the lessons that were learned during the implementation of this project. We now have a rich understanding of the nature of the mechanism problem and approaches to its solution. There are many issues that were not explored because of the limited time available, but several original ideas were implemented in our final version of *Mechedit*.

4.1 Putting off the problem

In Eric Enderton's masters thesis, attention is given to the problem of detecting rigid submechanisms. His stated solution is exponential. The detection of these rigid submechanisms is useful since Grubler's formula fails in the presence of overconstrained rigid submechanisms. Our solution, was to ignore rigid submechanisms. Instead we rely on the animation to show rigid mechanisms. As a result, during the contraction phase, we run into overconstrained portions, which correspond to two-cycles in the graph contraction phase. It becomes possible to write a "two-solver" thus solving the problem of overconstrainment and non-genericism in a pragmatic and effective way (see figure 1). This method was robust enough to animate overconstrained mechanisms and non-generic mechanisms correctly without doing complicated analysis beforé animation.

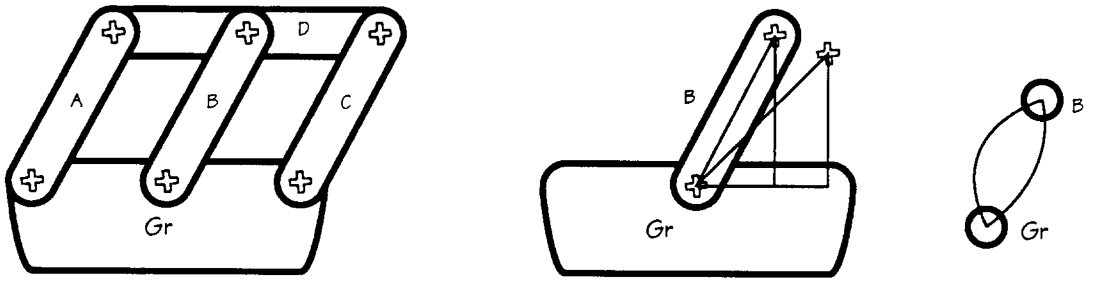


Figure 1: Grubler's formula fails for this simple mechanism due to its non-generic geometry (the three links are exactly the same). Although Grubler's formula indicates the above mechanism is rigid, it has 1 DOF. The diagram to the right shows the "two-solver" step and its associated subgraph.

4.2 The Duplex Mechanism (or answering the question "where is the 'bad' part of the mechanism?")

In studying how to best find rigid submechanisms, we attempted to come up with a method that was better than exponential. The idea was that there may be "bad" subgraphs of the graph of any mechanism. These bad subgraphs were either rigid or overconstrained, and had to be identified and culled away in a preprocessing step. The problem was that certain types of mechanisms were rigid because of what we termed "non-local" interaction. A "local" interaction would be a triangle, which is rigid and easy to detect. In the graph, this corresponds to a 3-cycle. If a subgraph had cycles that weren't of size 3, but had a connectivity that made it rigid, it wouldn't be as easy to detect. In fact, such a mechanism would not be solvable using our method of contracting 3-cycles, since

no 3-cycles exist, although the mechanism has 1 DOF! The simplest of these mechanisms, we christened the "Duplex Mechanism" since it looks like two houses connected together by a bridge of sorts (see figure 2).
It doesn't seem likely that the duplex mechanism can be solved in a sequential manner. It must be solved all at once, perhaps using numerical methods. While it may be possible to write a particular solver for this mechanism, this would have little value beyond showing the movement of this particular mechanism. Perhaps there is some extension to Enderton's technique, that uses simultaneous equations and is simple enough to provide interactive speed for reasonably large mechanisms. There are currently many numerical techniques which converge on solutions for a mechanism after a few iterations. It seems possible, however, that there is some simple extension that would avoid using these numerical techniques in favor of a faster method that is more geometrical in nature.

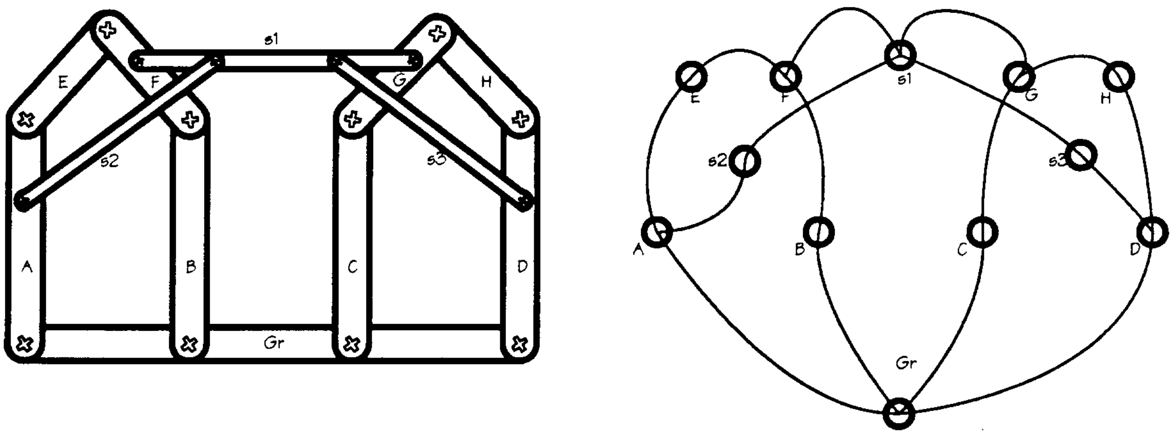


Figure 2: Grubler's formula for the Duplex Mechanism shows that it has 1 degree of freedom: 3 (12 Links - 1) - 16 Joints * 2 = 33 - 32 = 1 DOF. To construct the mechanism's graph to the right, links map to nodes and joints map to edges. The smallest cycle consists of 5 nodes.

4.3 Extension to Three Dimensional Mechanisms

We have thought a bit about the extension of this technique to three dimensions because many of the interesting real physical mechanisms have non-planar kinematics. All of the same considerations exist as they do for the planar simulator, but the problems become more involved. For example, in a planar system with only rotary and prismatic joints, there are (after symmetry) six possible three-cycle cases for which solvers must be written. However, in the general 3-D case, the simplest, general 1-DOF cycle is a 6-cycle requiring 32 separate solvers for rotary and prismatic joint combinations.

We also feel that solvers for degenerate (non-generic) cases become more important in the 3-D system. These solvers operate on submechanisms whose topology would indicate that they are overconstrained, but whose geometry allows them to move anyway. Almost any real physical 3-D mechanism one can think of has these degeneracies, often in the form of intersecting rotary axes (e.g. a universal joint). For the planar system the degenerate solvers operate on two-cycles, but for the 3-D system there must be degenerate solvers for everything from five-cycles on down.

4.4 Interfacing C, C++, Suns and Iris's

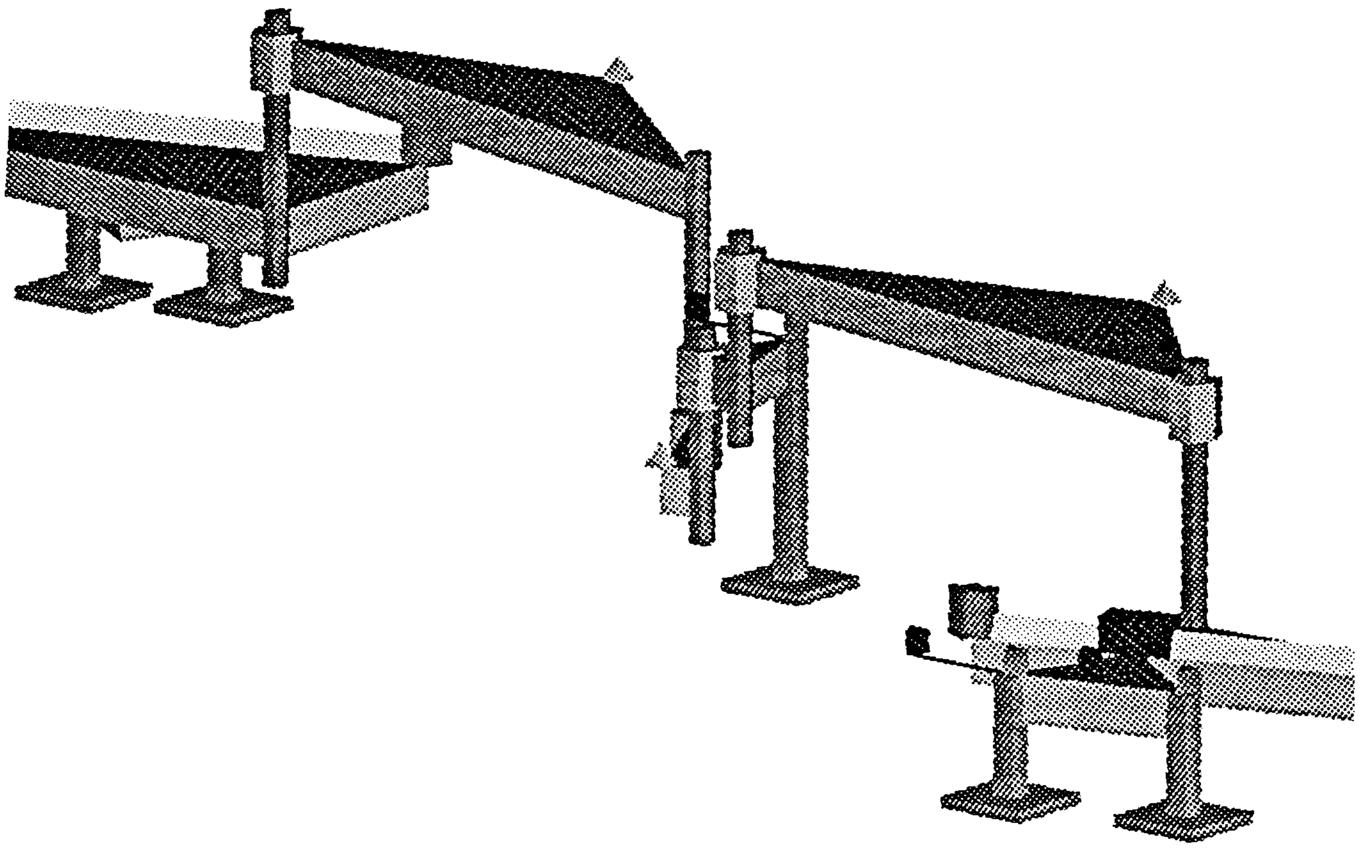
Since the Graph Package was written in C++, and the remainder of the program was written in C, it became necessary to write an interface module so that the main program could call routines in the graph library. Nominally this is a trivial task because the proper magic words are provided in C++. Things became more complicated when we found that the C++ code that had been written and compiled with the GNU gcc compiler on a Sun Workstation, would not compile properly with gcc on the SGI workstation on which we were working (enterprise) due to a problem with the gcc installation there. We were able to compile the C++

on other SGI workstations (torus cluster), but found that the C code wouldn't compile there because of the way the ANSI compiler was configured. We tried putting the C++ objects into a library, but the librarian on one SGI (enterprise) didn't like libraries that were made on the other SGI (torus). The solution was to do all C++ compilation on one of the torus machines, then `rcp` all the object files to enterprise. There the C files would be compiled and everything linked together to produce the executable. Surprisingly, we were able to automate this entire process through creative use of `make`. A shell script running in an infinite loop on a torus machine would watch for files to be `rcp`ed and then compile them.

This generalizes to a larger lesson about constructing software, and that is, to make the interface between code sections as small as possible — to make a code "bottleneck". Since, the majority of software bugs occur in these code interfaces, making the interface as small as possible makes sense. This corresponds to having as few routines to call in this interface as is practical. In graphics, this is a good precaution as well. One particularly successful example of this strategy is the Macintosh computer's graphics subroutines (Toolbox). All drawing to the screen is eventually accomplished through successive calls to one routine, a bit blitter called, appropriately `Copy_Bits()` (now `Copy_PixMap()` for color). Doing all drawing through this primitive creates consistency.

References

[1] E. Enderton, "Interactive type synthesis of mechanisms," Master's thesis, University of California at Berkeley, 1990.



NAME

mechedit - An interactive mechanism editor and animator

DESCRIPTION

mechedit is an interactive tool for building and simulating planar mechanisms (linkages).

SECTION 1: OVERVIEW

A few of mechedit's features are:

- Interactive construction of links and joints using the mouse. A mechanism is assembled by connecting links together at their joints .
- Two joint types: rotary and prismatic.
- Three-dimensional display with mouse-moveable eyepoint.
- Interactive animation of the mechanism using the mouse to drive a joint.
- Schematic rendering of the mechanism. The kinematics of the mechanism must be planar, but it is rendered in 3-D (the shape of the plates is derived from the 2-D convex hull of its joints).

SECTION 2: CREATING A MECHANISM

The first step in using *mechedit* is the creation of a mechanism (bar linkage). Using the mouse, the user constructs the links that will become parts of the mechanism. Each link contains a number of sites where the link can be joined to other links. Links are then dragged with the mouse to join them together to form the mechanism.

SECTION 2.1: CREATING LINKS AND JOINTS

The editor allows quick, easy, assembly of mechanisms through simple point, click, drag operations with the mouse.

Each link is defined by several sites, displayed as light blue pyramids. The link body is a polygonal plate whose shape is defined by the two-dimensional convex hull of its sites. Thus the shape of the plate continuously stretches as sites are edited with the mouse. To create a link, choose **New Link** from the menu. A new link with three sites will appear at the center of the screen. Alternatively, an existing link can be copied, as described below. This new link can now be dragged around with the mouse, edited, and joined to other links.

A link is edited by editing its *sites*. Sites can be added to a link, moved around on the link, and deleted from the link. During editing, the mouse keys are assigned as follows: the **left mouse button** moves things, the **right mouse button** adds things, and the **middle mouse button** deletes things. With no keyboard keys held down, these actions apply to whole links. With the **shift key** held down, individual sites are edited. To copy a link, drag it with the **right mouse button**.

SECTION 2.2: ASSEMBLING THE MECHANISM

To assemble a complete mechanism, links must be connected together by *joints*. It is these connections that will ultimately define the dynamic behavior of the mechanism. A joint is formed whenever a site of one link is dragged close to a site of another link. The sites will snap together and form the joint. As stated above, with no keyboard keys held down, dragging a site with the **left mouse button** moves the entire link. If the dragged site comes close to a site on another link, the sites will snap together to form a joint. When the **shift key** is held down, dragging a site with the **left mouse button** moves only that site; the link stretches to accommodate the motion. If this site comes close to a site on another link, it will snap to it to form a joint.

When a joint is formed, it will be of the *current joint type*. The current joint type is selected from the menu to be either *rotary* or *prismatic*. Existing joints can be changed from one type to the other by holding down the **shift key** and clicking on them with the **right mouse button**. Additionally, prismatic joints have a length and an orientation. A prismatic joint, can be rotated and enlarged by grabbing either of the "sizing arrows". Dragging an arrow will change both the direction of the prismatic joint, and the size of the adjacent links it joints.

Finally, for a mechanism to operate, it must have some joints to *ground*. The *ground link* is an implicit link that never moves, and is not shown except that it has sites. These sites can be edited just the same as the sites on other links. They can be added, moved and deleted. A **ground joint** can be formed by dragging a site on a link close to a ground site, or by dragging a ground site close to a site on the link. Ground joints are always rotary.

SECTION 3: ANIMATING THE MECHANISM

The second step in using *mechedit* is animating the mechanism. The mouse is used to drive one of the ground joints, and *mechedit* properly animates the mechanism according to the kinematics defined by the links and joints. Ground joints are the only joints that can be driven. They are displayed with *cranks* which can be dragged with the *left mouse button.*

SECTION 4: MENU COMMANDS

Several menu commands are alluded to above. In addition, there are commands for file manipulation, lighting, and something. This section describes each command in detail.

Save Saves the current mechanism to a file. If the current mechanism has not been saved previously, the user is presented with the Save File dialog box.

Load Loads a previously saved mechanism from a file.

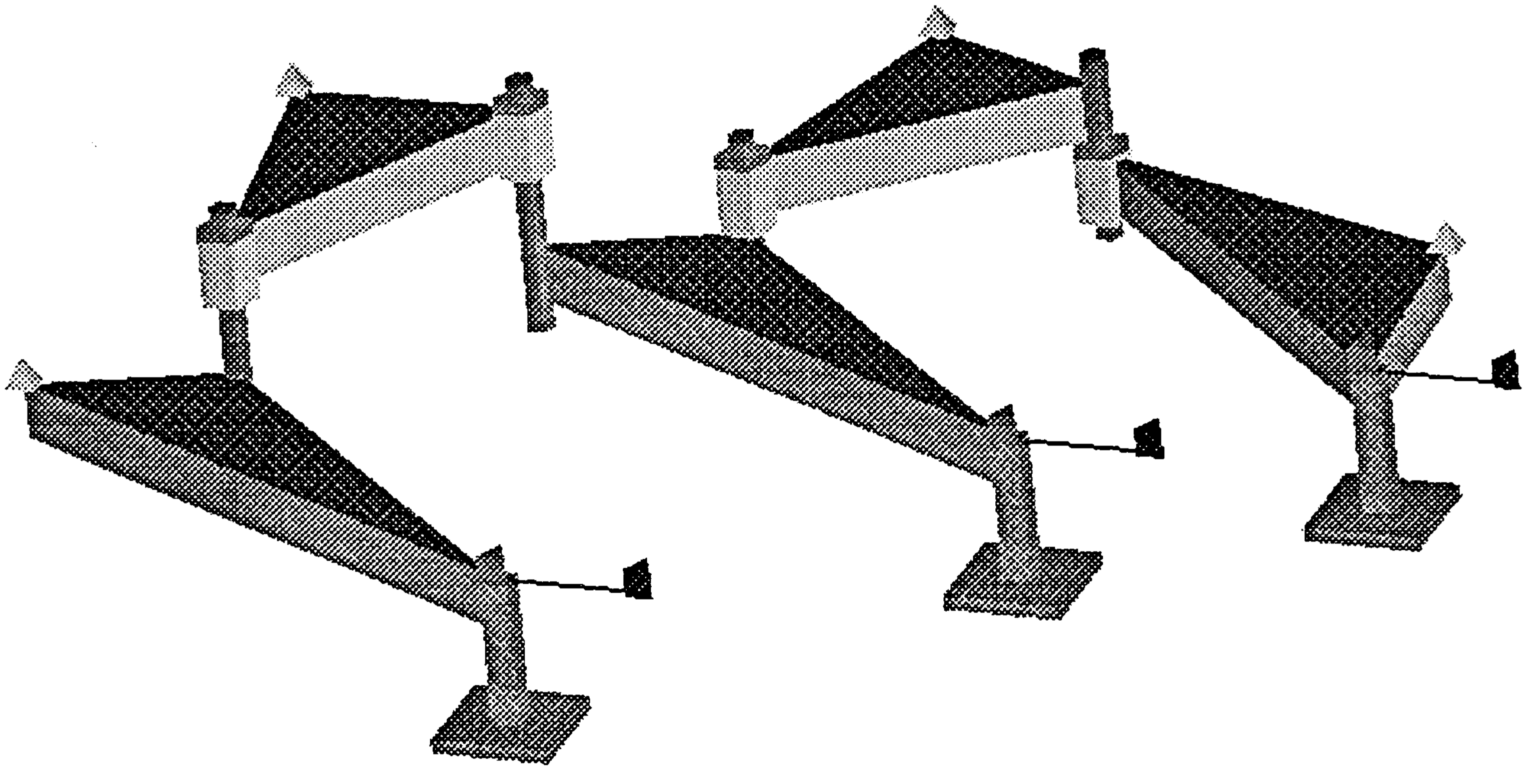
View > Perspective
Renders the mechanism with perspective.

View > Orthogonal
Renders the mechanism without perspective.

View > Reset View
Resets eye-point to program startup position.

AUTHORS

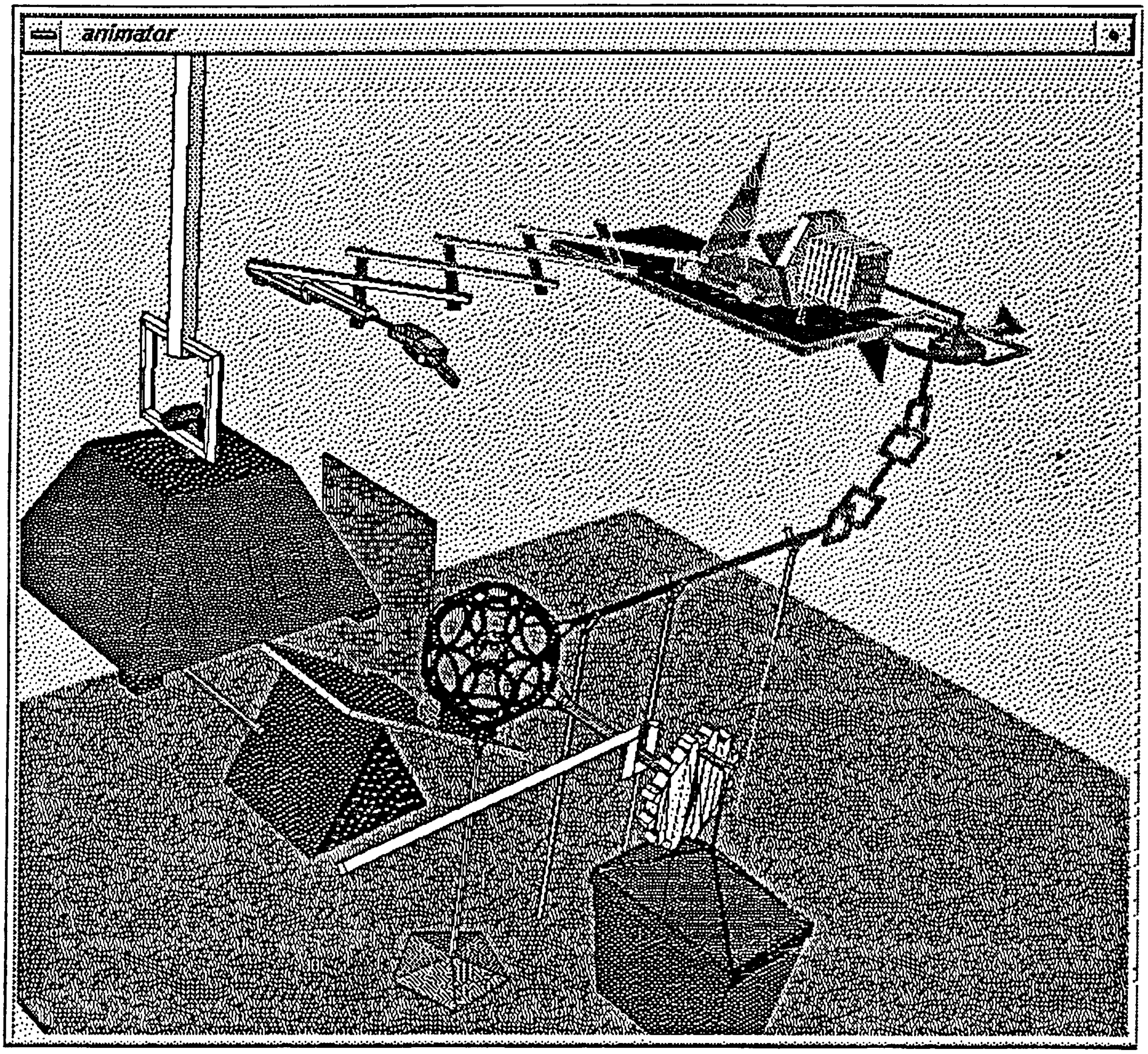
Steve Burgett - burgett@robotics.berkeley.edu
Roger Bush - rbush@miro.berkeley.edu



285 Final Project
A Machine for Mr. Goldberg: Integrating Mechanisms

Mark Brunkhart

December 11, 1992



Universal Joint. Planetary Gear. and Bellows

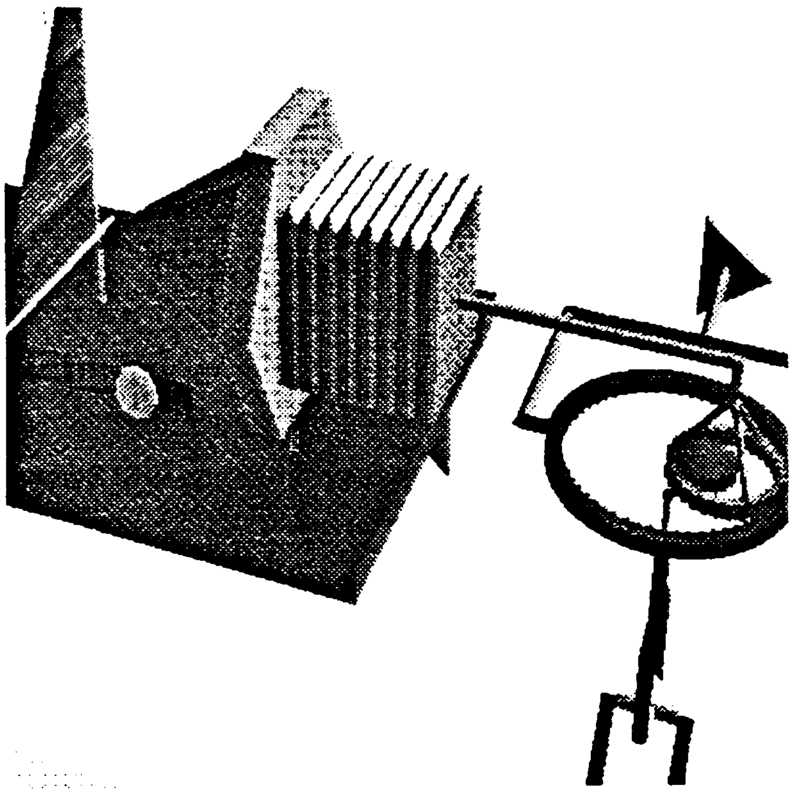


Figure 1: *The constant velocity double universal joint is connected to the interior gear of the planetary gear. The output shaft of the planetary gear is connected through a revolute joint to a shaft which forces the action of the bellows.*

1 Motivation

Much in the manner that VLSI design has moved from an environment in which tired designers push around rectangular pieces of laminate to an environment in which designers use point-and-click graphical environments to construct and test hierarchical circuits, so too does it seem reasonable to believe that engineers in the future will construct machines by snapping together mechanisms, viewing the motions that result, and re-evaluating their original designs. The design process, from initial design to ultimate testing, which currently can take months, should be reduced to a few hours. The value of working in a computer environment, as opposed to a physical environment, is obvious. Erroneous designs can be caught on the fly, more time can be spent considering the value of different designs, and the tedium of diagraming, calculating, and then realizing one's errors only too late to change them can be minimized.

Wine Rack Mechanism

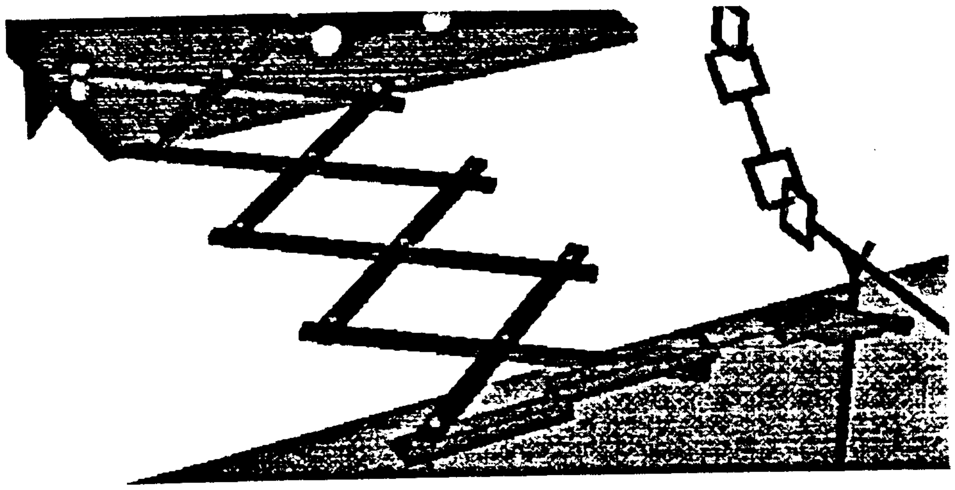


Figure 2: *The wine rack mechanism has one input fixed and the other forced by the motion of the toy boat at the top of the figure. The mechanism is a collection of revolute joints terminating in two sliders attached to which is a model hand.*

2 Animation and CAD

The ability to display moving and interacting parts on screen is a frequently sought after goal derived from two primary objectives. Engineers use computer aided design to simplify the process of constructing mechanisms. For these users, animation is a visualization of the kinematic relationships between the various parts in their mechanisms. CAD tools currently provide substantial capacity for describing objects, analyzing the methods which can be used to machine and assemble these objects, and analyzing the result of applying various forces to these objects. Animation of such mechanisms may be difficult as they are described by the user in terms of shape, boundaries, and locations or as features such as slots and holes. To derive the degrees of freedom and interaction of such parts is difficult because the mechanical role of the different parts is lost in the description of the object as a collection of points, edges, and faces.

A simpler task is that of the animator who simply wishes to construct the sequence of pictures which represent a mechanism in motion. As the animation is the end goal and the basic interaction of the parts is understood in advance, the artist can use a data representation which reveals the kinematic interaction of the parts. For example, a hinge may be described as two plates connected at an edge with one rotational degree of freedom through a limited range. By describing an object as a hinge, the animator reveals substantial information about the kinematics of the object, information which is difficult to extract from a boundary representation of its various parts.

In many ways, this problem is similar to that of extracting features in computer-aided manufacture from a boundary representation. If the computer is presented with an object to be constructed in a representation which reveals the primary information needed to

Gear Shell

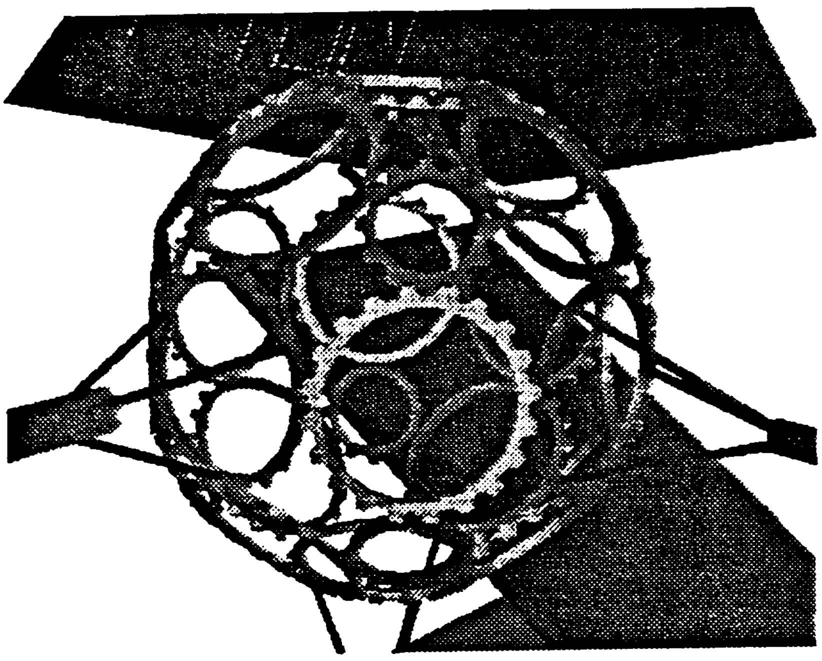


Figure 3: *This mesh of gears represents one of the most complex interacting devices in the entire machine despite the fact that its input and output shafts turn at the same rate. This is a clear example of the value of hiding complexity through hierarchy when determing motion equations.*

machine that object, the problem of creating an efficient machining path is substantially simplified. Constructive solid geometry (CSG) is one such representation. In CSG, an object might be described as a block with a one inch cylinder cut through it and a two inch slot along its base as opposed to a set of points and faces. As the computer understands how to construct a slot or a hole in a block, its task is substantially simplified when determining how to machine this object in a shop. Consider, in contrast, the task of identifying a construction path beginning only with a collection of points and faces.

The task of animating mechanisms requires a data structure which reveals the kinematic structure of an object simply and concisely. Much of the work performed for this project was nothing more than a verification of this need for an efficient language for describing mechanisms and their interactions. As a test case which identifies the difficulty of converting points and faces into interacting mechanisms which can be animated, I attempted to construct a Rube Goldberg machine using the tools currently available at the University of California at Berkeley for describing animations.

3 Work Performed

A Rube Goldberg device is a mechanism of substantial complexity designed to perform the simplest of tasks. By its very definition, a Rube Goldberg machine requires numerous machines to interact in a variety of convoluted ways. By integrating many of the various weekly projects constructed by students in the class throughout this semester, I was able to gain an understanding of the difficulties in the design process and more importantly to gain a feeling for the type of tool which must be built to simplify the task of constructing and integrating mechanisms for animation.

The device constructed is displayed in the figures throughout this document. All of the mechanisms move with a single variable representing time which may be controlled by the viewer. The convoluted task performed is described in Table 1 for the reader's interest and entertainment.

The structure was created using UGmovie, a rendering tool which permits the mathematical specification of points and faces based on one or more variables controlled by the user. By hierarchically constructing the various items in the view beginning with vertex and face locations and developing each of the various mechanisms in the design separately, the primary task becomes simply integrating these items into a single mechanism. Since many of the mechanisms in the device had already been constructed by students, integration became the primary difficulty.

4 Limitations of the Tool

UGmovie is controlled from a text environment. The user is forced to specify every parameter of the drawing from vertex locations to mechanism motion by editing text files. Needless to say, such an environment does not lend itself to the integration of various objects into a single machine. A number of difficulties hinder the construction process.

4.1 Lack of a 3-dimensional Editor

The specification of 3-dimensional locations in text form requires the user to perform mentally a transformation which could easily be performed by the computer. The tendency to invert the direction of translations and rotations, the inability to juxtapose two objects without performing calculations, and the general frustration of being forced to switch between editing a text file and editing a picture lead to a fairly difficult and time consuming process. If a user were able to quickly locate where a mechanism was to be placed, identify faces which are to touch, and expand, move, and otherwise transform objects on the screen, construction time of complex, animated mechanisms would be substantially reduced.

A Machine for Academic Extraction and Doctoral Deployment

Table 1: *This machine offers the university professor and the community at large a method of recovering the wayward eternal student from his perpetual wanderings of the hallowed ivy-covered halls of academia.*

1. *A student enters a professor's office. hands his PhD dissertation final draft to the professor, and seats himself in the chair.*
2. *The professor absent-mindedly drops the paper in the trash can which*
3. *weighs down the trash can. lowering it, cranking the gear, which in turn cranks the gear mesh.*
4. *The gear mesh rotates, spinning the constant velocity, double universal joint, which*
5. *rotates the inner wheel of the planetary gear.*
6. *The outer gear and pin assembly drives the piston causing*
7. *the bellows to pump in and out which*
8. *causes wind from the bellows to push the toy boat.*
9. *The toy boat expands the wine rack which is connected to*
10. *a toy hand.*
11. *The toy hand unlatches the 1 ton weight which falls on*
12. *the seesaw quickly raising the chair on the opposite end.*
13. *Student idly dozing in the chair is launched through a nearby window achieving the necessary escape velocity to exit academia and enter the real world.*

4.2 Lack of Well-Defined Interfaces

Each of the individual mechanisms has certain obvious physical interfaces. A gear can be accessed at the shaft or at the teeth. A universal joint is accessed at one of the two exterior shafts. At these interfaces, rotational and translational speeds are well defined and could easily be passed from one mechanism to the next if a simple interface for connecting them were defined. Under the current textual environment, the user is forced to extract these values from the text of the driving mechanism and insert such values in the text of the driven mechanism. Mathematical complexities in the text tend to propagate from one mechanism to the next, and when the user finally describes his last mechanism, he is forced to interpret some absurdly complex expressions. This is obviously not an ideal environment for mechanism construction.

4.3 Snappable Parts

A mechanism is almost invariably a collection of sub-mechanisms. A mechanism editor should recognize this and permit the user to interactively construct mechanisms from smaller mechanisms. Ideally, this process should be easy when using a graphical interface. A user should simply be able to identify two sub-mechanisms from a library of such mechanisms and connect these mechanisms at the appropriate interfaces, and all location and motion calculations should be calculated for him. Essentially the user should be able to snap together the various parts of a mechanism identifying only those values which affect the kinematic relationships between this mechanism and other mechanisms (radius of a gear, length of a shaft, etc.).

4.4 Mechanical Hierarchy

Finally, a user should have access to a mechanical hierarchy which hides the underlying complexity of submechanisms. If a user wishes to connect a combustion engine to a drive shaft, he should not be forced to access or operate on the complex workings of the combustion engine. The interface to the larger mechanism should be clearly defined making the task the simple one of snapping together parts.

5 Conclusion

Ultimately, this project was a study in handling complexity. As complex mechanisms begin to interact, complexity grows exponentially with the number of mechanisms. This trait makes the construction of complex mechanisms extremely difficult in a textual environment. Modern computer graphics appears to hold a solution to this problem; however, much work remains to be done to create an environment which truly supports mechanism construction and animation. This work can benefit substantially from previous work in two- and three-dimensional graphics and object editing but will certainly require further insight to handle the added dimension of motion and object interaction.

Universal Joint & Constant Velocity Joint Museum Piece

Brian Mirtich
Saba Rofchaei
CS 285
11 December 1992

1 Overview

The goal of our mini-project was to produce a "museum piece" graphical object that demonstrates the operation of both the universal joint and the related constant velocity joint. The final product is a set of *Unigrafix* files that combine to produce a dynamic structure which may be viewed with the *ugmovie* program.

Figure 1 below shows the basic design of our museum piece. Specifically, the structure contains four universal joints and two constant velocity joints arranged in a hexagonal configuration, with six connecting shafts. Because of the placement of the joints and the choice of angles between the shafts, the entire assembly is able to turn without "locking up." That is, each of the six shaft turns around its axis, as the joints also turn.

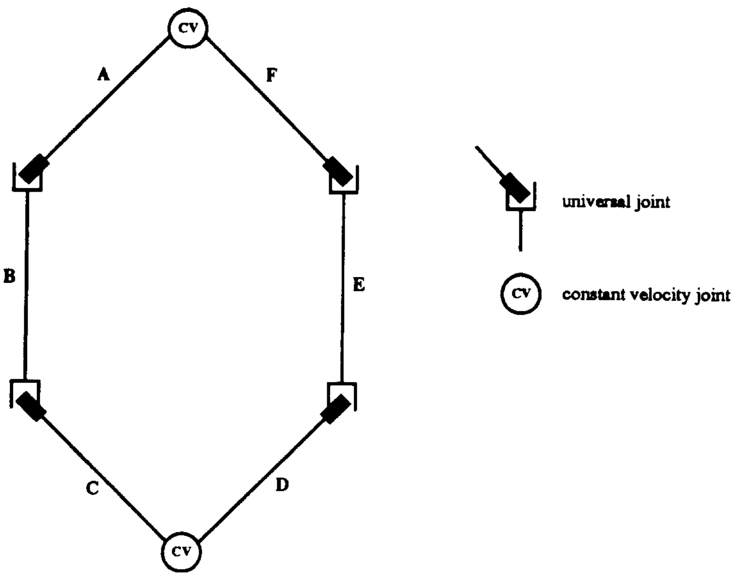


Figure 1: *A museum piece demonstrating universal and constant velocity joints.*

2 Technical Discussion

A large portion of this project was deriving the kinematic equations governing the various moving parts of universal and constant velocity joints.

2.1 Universal Joint Kinematics

A universal joint comprises three moving parts: input and output forks, and a rotating cross piece connecting them. The input and output forks are connected to corresponding input and output shafts. Figure 2 shows a universal joint along with a convenient coordinate system. For clarity, the cross piece is not shown in this diagram. It may assume many different shapes, and its function is to enforce a perpendicularity condition between line segments $\overline{pp'}$ and $\overline{qq'}$. The angular positions of the input and output shafts are denoted by α and β, and θ specifies the bend angle that the output shaft makes with the extended input shaft. The dotted curves in the figure show the paths of the points p, p', q, and q' through space.

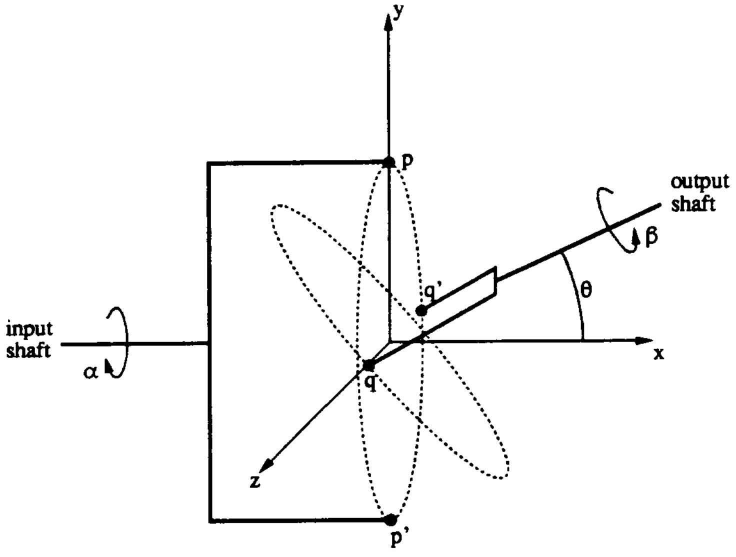


Figure 2: *A universal joint model used to derive kinematic equations.*

Our derivation begins with relating the input and output shaft angles, α and β. Assume the configuration in figure 2 is the zero configuration, i.e. $\alpha = \beta = 0$. Let $R_x(\alpha)$ denote the 3×3 rotation matrix which transforms points in 3-space by rotating them an angle α about the x-axis, and use analogous notation for other rotation matrices. The coordinates of p can then be described as a function of α:

$$\vec{p}(\alpha) = R_x(\alpha)\vec{p}(0) = \begin{bmatrix} 0 \\ \cos\alpha \\ \sin\alpha \end{bmatrix}.$$

Point q is dependent on both β and θ:

$$\vec{q}(\beta,\theta) = R_z(\theta)R_x(\beta)\vec{q}(0,0) = \begin{bmatrix} -\sin\theta\sin\beta \\ \cos\theta\sin\beta \\ -\cos\beta \end{bmatrix}.$$

Now we simply express the perpendicularity constraint imposed by the cross piece to obtain a relation between α and β.

$$\vec{p}\cdot\vec{q} = -\cos\theta\cos\alpha\sin\beta + \sin\alpha\cos\beta = 0.$$

We can solve this nonlinear equation explicitly for β, obtaining

$$\beta = \tan^{-1}(\sec\theta \tan\alpha).$$

If $\theta = 0$ the relation condenses to $\beta = \alpha$, but in general, the relation is nontrivial. For non-zero θ, β will sometimes lead α and sometimes lag α. However, for any values of θ, $\beta = \alpha$ at multiples of $\pi/2$. We therefore see that while constant angular speed is not preserved across a universal joint, the input and output shafts always have the same average revolutions per minute.

In remains to determine the kinematics of the cross piece. We already know how the cross piece endpoints p, p', q, and q' move, but for the *Unigrafix* description we must express the cross piece orientation by beginning with some initial orientation and then applying a sequence of rotations about the coordinate axes. Figure 3 shows how we may always obtain a properly oriented cross piece by beginning with the initial position, then applying a rotation by angle ϕ about the y-axis, followed by a rotation by α about the x-axis. We need only

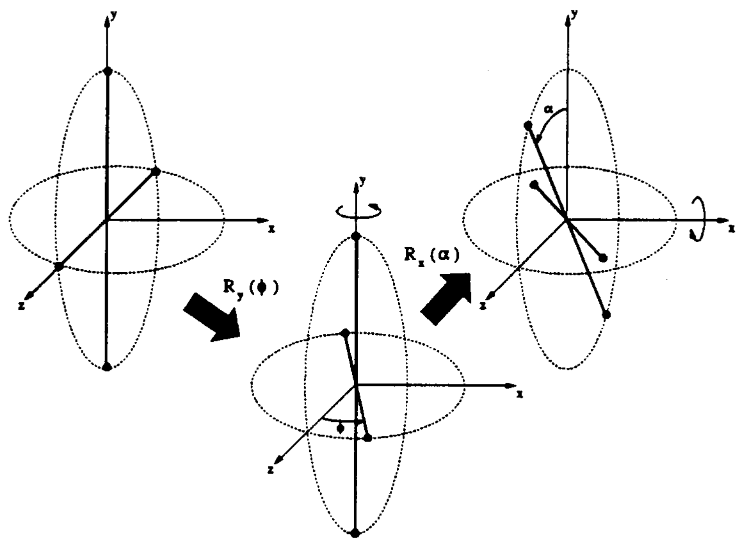


Figure 3: *Obtaining proper cross piece orientation through two rotations about coordinate axes.*

determine the unknown ϕ, which can be obtained by solving the equation

$$R_x(\alpha)R_y(\phi)\vec{q}(0,0) = \vec{q}(\beta,\theta).$$

Solving this equation for ϕ yields

$$\phi = \sin^{-1}(\sin\theta \sin\beta),$$

and we are able to specify the cross piece orientation through a sequence of two rotations as required.

2.2 Constant Velocity Joint

While universal joints provide a way to transfer a rotational motion around a bend, their inability to preserve instantaneous angular velocity from input shaft to output shaft may sometimes be a disadvantage. Constant velocity joints solve this problem. They are closely related to universal joints, and indeed one

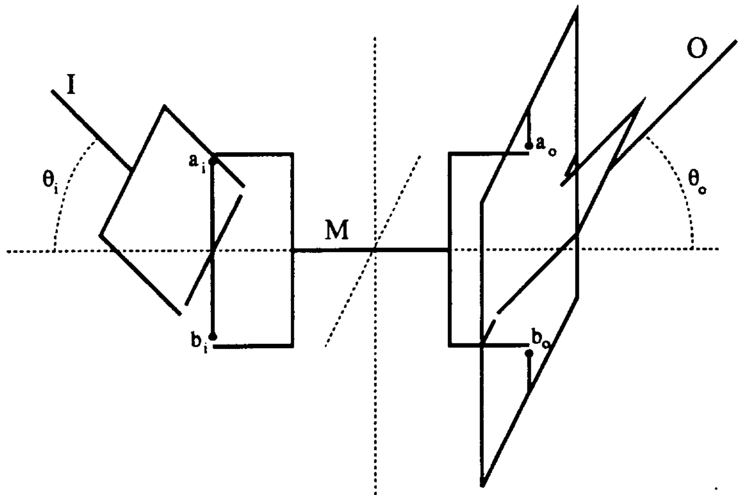


Figure 4: *Obtaining a constant velocity joint with two universal joints.*

can visualize a constant velocity joint by beginning with two universal joints as shown in figure 4. Note the in the universal joint between the middle shaft M and the output shaft O, the cross piece has been replaced with a (square) ring. The ring serves an identical function to that of the cross, insuring the perpendicularity of the segments between the fork endpoints, and the behavior of the ring universal joint is identical to the cross universal joint. Assuming the input shaft I turns at a constant velocity, shaft M's velocity will in general be nonconstant but periodic. However, shaft M is the input shaft to the second universal joint, and this joint will exactly "undo" the velocity transformation accomplished by the first joint, as long as the bend angles θ_i and θ_o are equal. The result is that shaft O turns at a constant velocity, identical to that of shaft I.

We now wish to somehow reduce this double universal joint configuration to a single joint. Imagine shrinking the length of shaft M, bringing the two universal joints closer and closer. This does not affect the constant velocity relation between shafts I and O. We can reduce the length of shaft M to zero, essentially eliminating the shaft altogether, and connecting points a_i and b_i directly to a_o and b_o respectively. This is the reason we began with one cross universal joint and one ring universal joint; it allows us to bring the two joints together without self intersection. There is still a degree of freedom at these connections; the cross is free to rotate about axis $\overline{a_i b_i}$ while connected to the ring. The result is a single joint which transfers a constant velocity around a bend from the input shaft to the output shaft. Furthermore, the kinematics of this joint are trivial once one has the kinematics of a standard universal joint. The shafts turn at identical constant speeds, and the orientations of the cross and ring are the same as they are for the two separate universal joints.

3 Results

Figure 5 shows a snapshot of our museum piece displayed with the *ugmovie* program. The piece is composed of six shafts with forks on each end, four universal joints, and two constant velocity joints. The star-shaped clusters on the shafts are present to aid perception of the rotational speeds of the shafts.

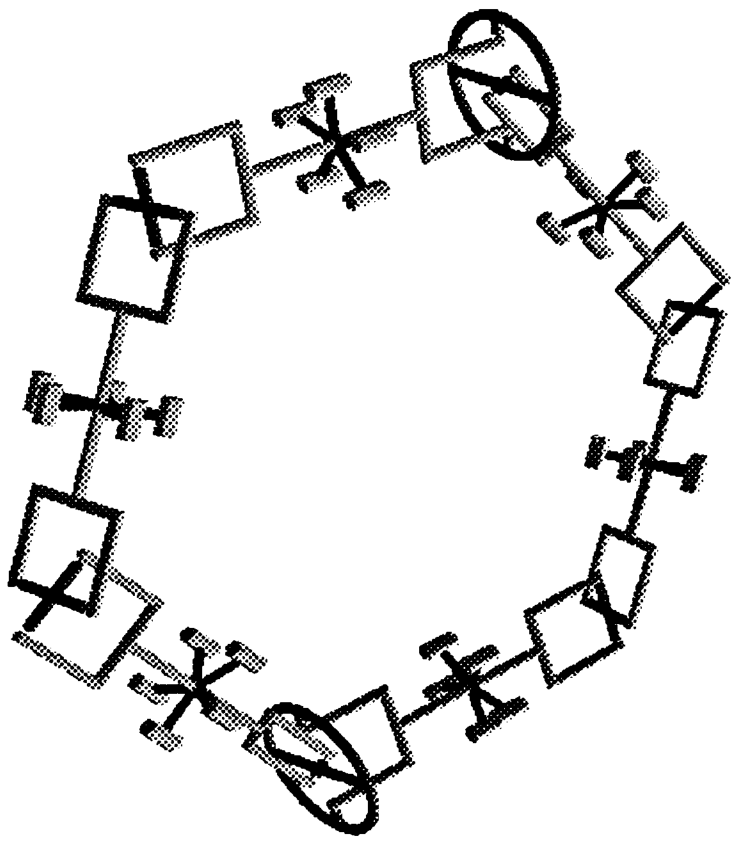


Figure 5: *A snapshot of the museum piece.*

4 Conclusion

This mini-project provided us with excellent experience in deriving the kinematics of mechanisms.

If we had another chance to complete this project, one enhancement we would make would be the ability to vary the shape of the overall system. Currently, the system is in the shape of a regular hexagon, however it would be interesting to experiment with the bend angles of the shafts. One could specify the bend angle at the constant velocity joints with a UGmovie variable slider, and the other bend angles and shaft lengths would be adjusted accordingly. For example, to magnify the velocity disparity of the two non-constant velocity shafts, one could reduce the bend angles at the constant velocity joints thereby increasing the bend angles at the universal joints.

Another idea we would try would be to create more complex kinematic chains. The current museum piece is a single closed loop, but perhaps there are ways to build multi-loop structures from shafts, joints, and gears which are still able to turn. Furthermore our current museum piece has a single degree of freedom; we would consider designing structures with higher degrees of freedom.

Muscle and Bone Animation

John Boreczky
johnb@cs.berkeley.edu
CS 285 - Fall 1992

1. Introduction

There have been a number of attempts to use knowledge of human anatomy to produce more realistic models of the motion of the human body. Chadwick, et.al (1989) determine the approximate shape of a limb based on the position of the bones. They model the shape that layers of deformable material (muscle and skin) would take given the types of joints and muscles involved. Their emphasis is on generating exaggerated motion animation. Chen and Zeltzer (1992) realistically model the forces applied by and to muscles. They use this force information to determine the position of the muscles and bones and the position of the bones. This is very computationally expensive.

I wanted to create a simple animation of a human arm that was reasonably realistic but not so computationally complicated that the animation was prohibitively slow. Since the bones do not deform, it makes sense to do animation by moving the bones and having the attached muscles deform to keep the connections consistent. The big advantage to this is the relatively low computational complexity. Rather than trying to deform the muscles by following physical laws, I simply followed the idea that a muscle should expand in width as it contracts and it should narrow as it lengthens.

2. Mathematical Model

For ease in explaining the mathematical model, assume that we are modeling the positions of the humerus, radius, and biceps of the human arm. Figure 1 shows the interesting angles and measures used for determining the position of the bones and muscles in the two dimensional case.

Given that the biceps muscle is placed at a distance d1 from the end of the ulna, the x and y coordinates of the bottom point of the muscle are given by $\cos(\beta+\gamma)$ and $\sin(\beta+\gamma)$. Since we know d1, d2 and β, the triangle formed by the two bones and the muscle is uniquely determined, and we can specify the transformations needed to place the muscle and scale it properly.

In three dimensions, the mathematics are very similar. We need to rotate the muscle into the third dimension (around the y axis) in order to have the muscle attach to the ulna and not to the empty space between the radius and the ulna.

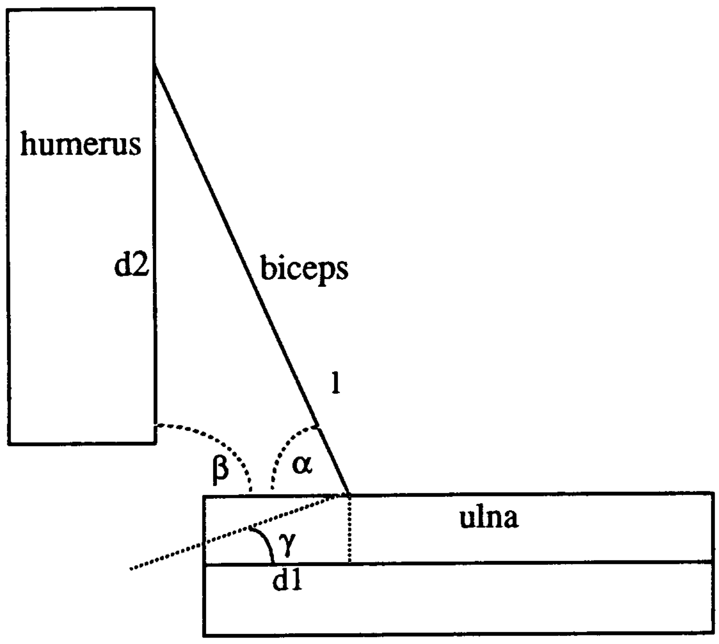


Figure 1

3. Implementation

The arm bone and muscle animation is implemented as an Extended Unigrafix (ugx) file. This file format is readable by the ugmovie program available on the Silicon Graphics Personal Irises. Ugmovie allows the graphical display of a set of objects to change as a function of one or more user controllable variables (e.g., time). The value of each variable can be manually set with a slider or tied to some function of the other variables. Ugx files specify the objects of interest (either by referencing other files or specifing the geometry of the objects explicitly) and how they should be transformed as a function of the variables.

Each of the three main arm bones (humerus, radius, and ulna) and the sholder blade (scapula) is described as a separate object. This allows each of the bones to be modified to more closely resemble the real human bone. The humerus and scapula remain fixed, while the radius and ulna don't move relative to each other, but together rotate around the elbow joint. My early attempts at creating very realistic looking bones were failures. Not only was it taking too much time, but the results were not very good. The current bones are simple cylinders with appropriate knobs and sockets.

The muscles are transformed copies of a single muscle. The current model contains the biceps, the brachialis and the brachioradialis. Adding the triceps made the model look too cluttered. Each muscle is scaled to the correct length and width and placed in the correct position. My early attempts at creating muscles that had the normal curvature and flattening into tendons at the ends proved too complicated to quickly deform. The current muscles are radially symmetric, allowing the narrowing and widening to be done with a simple scale operation. The width is scaled by the inverse square root of the amount the length is changed. This is quick to calculate and produces a realistic looking effect. Work is ongoing to create static tendon clusters that sit between the bones and the muscles in order to provide a more realistic look while keeping the computations simple.

It should be noted that the muscles are attached farther out from the elbow joint in this model than in a real human. The simple reason is that although it isn't technically correct, it looks better.

Figures 2 and 3 are screen dumps of the Extended Unigrafix file arm.ugx, showing the arm with two different angles of the elbow joint.

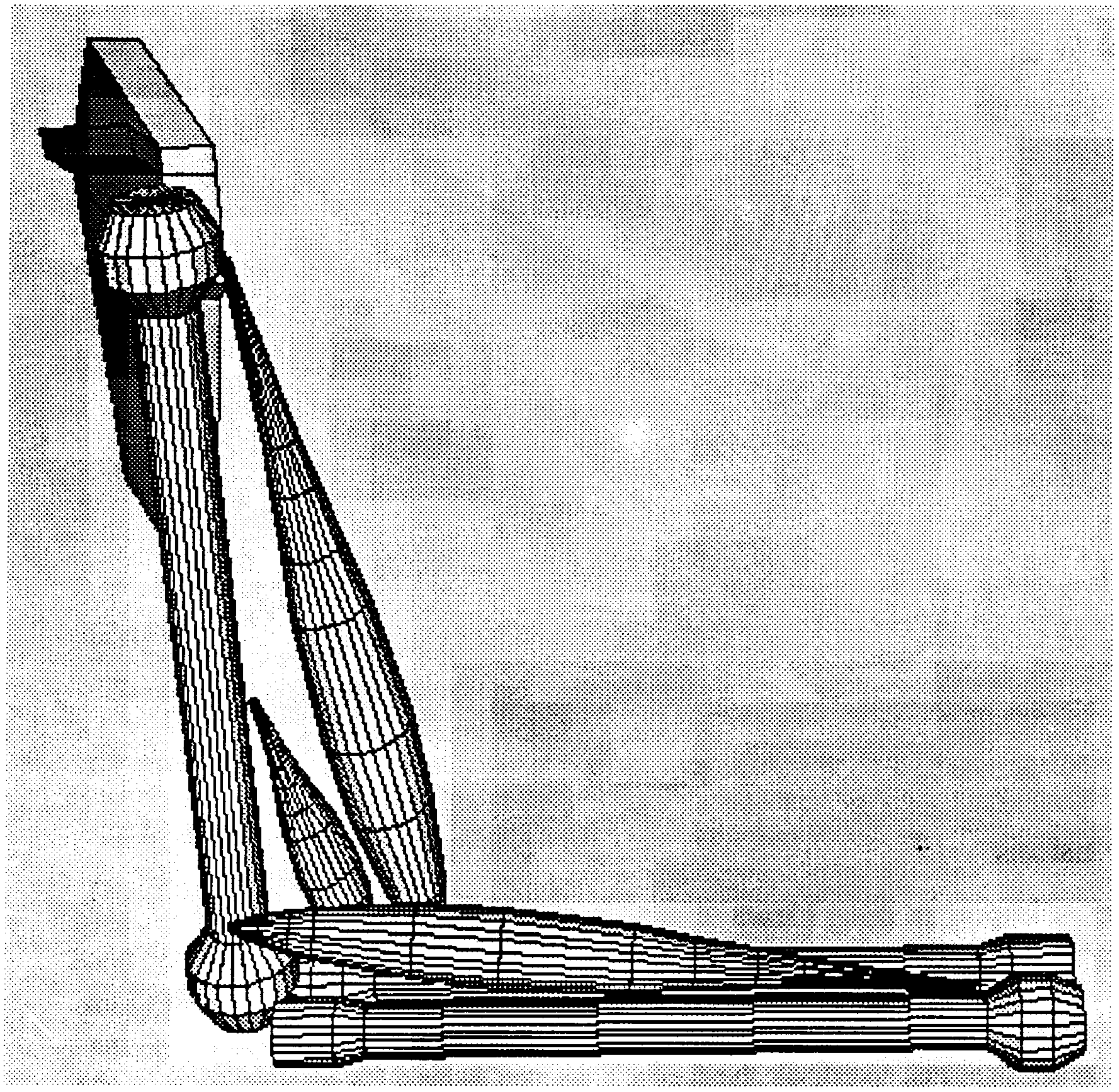


Figure 2

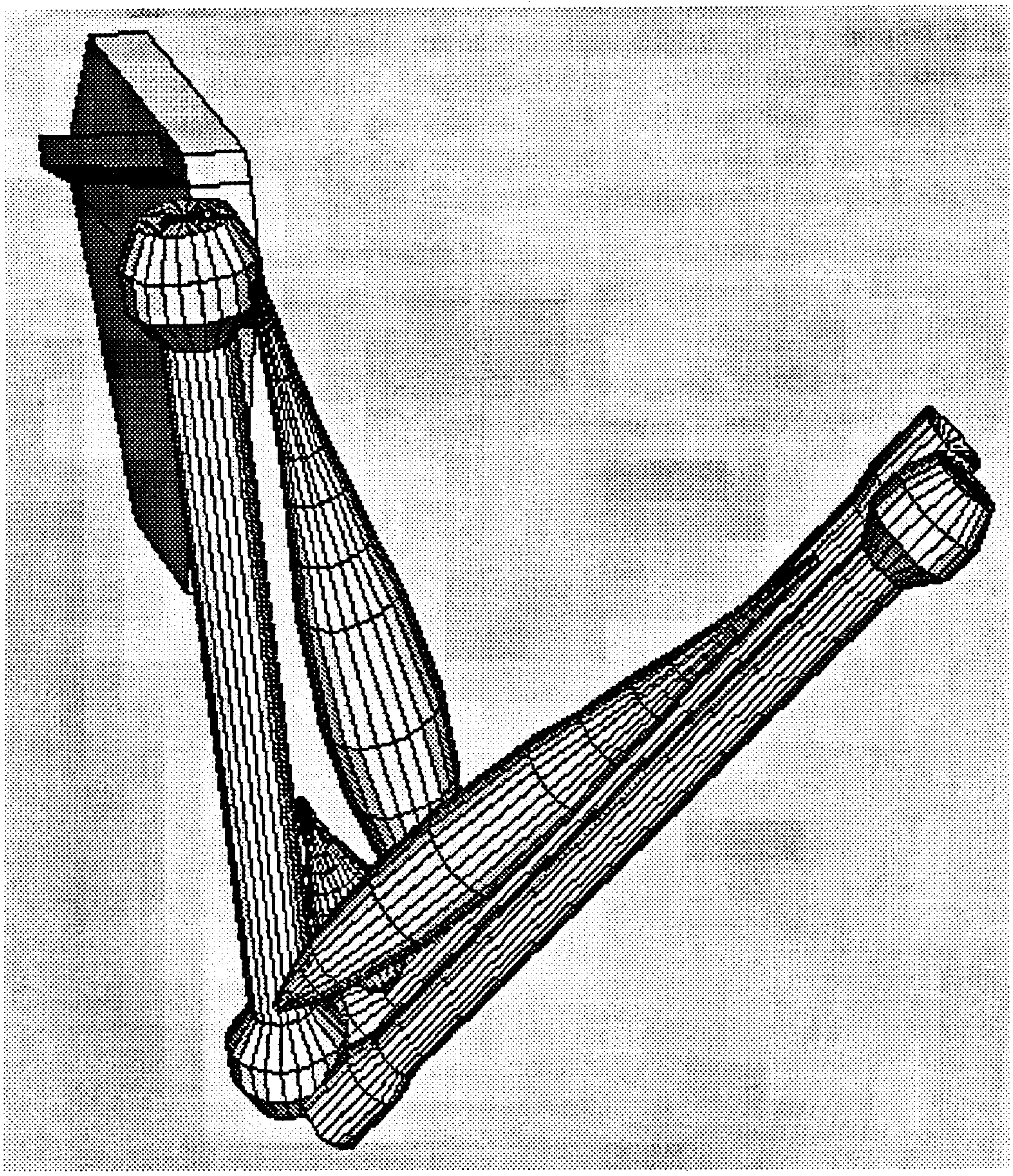


Figure 3

4. Conclusions

The most important thing I learned in this project was that it is very important to keep things simple. Even ignoring the obvious benefits such as reduced implementation time, simplicity is a big win. A detailed, but not quite right model doesn't look right to people. A simple, not quite right model allows people to use their imagination and in the process, overlook the flaws. I wasted quite a bit of time (due to my poor artistic skills) trying to come with realistic bones and muscles that I didn't use.

References

Chadwick, J. E., Haumann, D. R. and Parent, R. E., "Layered Construction for Deformable Animated Characters", ACM Computer Graphics, Vol. 23, No. 3, pp. 243-252.

Chen, D. T. and Zeltzer, D., "Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle Using the Finite Element Method", ACM Computer Graphics, Vol. 26, No.2, pp. 89-98.

Appendix - Running the Model

The arm description and the various object definitions are located in the directory `~c285-ap/proj`

In order to view the model, execute the ugmovie program from the above directory and load the file "arm.ugx". The single variable 't' controls the bend angle at the elbow joint.

CS 285	PAUL DEBEVEC
PROF. CARLO SÉQUIN	DECEMBER 11, 1992

Mini-Project Report
Modeling a Three-Ball Juggler

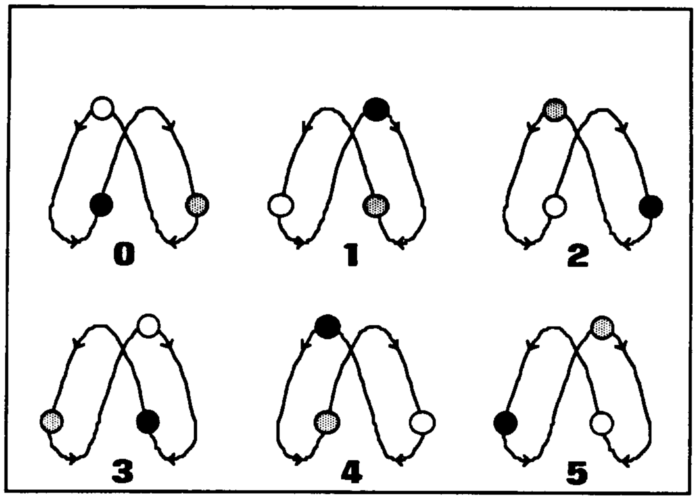
1. Overview

For my CS285 mini-project, I modeled the motions of the balls and hands while juggling three balls in the standard pattern. To show the mathematical model in action, I wrote a GL program to animate the juggler with 3D graphics. I also implemented the equations in a UniGrafix Movie file. This report describes how I modeled the motions using parabolas and Hermite Polynomials to satisfy the physics and dynamics of the juggler.

2. The Modeling Process

a. Figuring out the pattern

The reason that juggling is interesting to watch is that it is not obvious what is going on. To begin modeling the motions, I first had to determine exactly what motions I was modeling. By watching a videotape of a juggler, I decomposed the juggling process into six time intervals, with all of the catches and throws occurring at the interval boundaries. The pattern, showing the balls only, is as follows:



The juggling pattern at the beginning of each of the six time intervals

All of the objects remain in the same plane. Note that the white ball is exactly two time intervals behind the black ball, and the gray ball is exactly two time intervals behind the white ball. Also, it takes two time intervals for a ball to fly from one hand to the other, but only one interval for it to be thrown after it is caught.

b. Choosing coordinates

Next I chose a system of coordinates for the juggling space. I placed the origin at the point where the left hand releases the balls that it has thrown. I chose to refer to the width of an arc as **w**, the height of an arc as **h**, and the offset between the two arcs as **d**.

c. Modeling the ball motion

Since the acceleration due to gravity is constant, the balls must move in parabolic arcs as they fly from one hand to the other. Since the arc height **h** is specified and the flight must take two time units, **g** is constrained to be 2**h**. These parameters quickly lead to a quadratic expression for the ball's position along either of its arcs. During the time that a ball is not flying from one hand to the other, it is being held by one of the hands and consequently has the same motion. Thus all that remained to do was to model the hand motion.

d. Modeling the hand motion

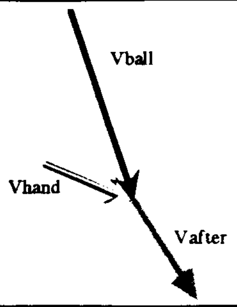
The hand motion is more complicated to model since the various forces a juggler exerts on his or her hands are difficult to gauge. However, I knew that the hands have certain goals to accomplish, and from these goals I placed constraints on the hand motion and then chose a mathematical model to meet these constraints. The goals of a hand are as follows:

- There is one time interval between catching a ball and throwing it. There is also one time interval between throwing a ball and catching the next one.
- At the time of a catch, the hand must be positioned under the ball. Also, the hand's vertical velocity must be more positive than the ball's.
- At the time of a throw, the hand must also be positioned under the ball. Moreover, the hand's velocity must be such that the thrown ball will land in the other hand in two time units. Also, the hand must have decreasing vertical velocity just after the throw in order for the ball to leave the hand.

These constraints neatly characterized the motion of the hands in two C2-continuous segments of one time interval each: from the throw to the catch and vice-versa. For each segment, the position at the beginning and end of the interval is known. Also, the velocity at the time of the throw can be calculated from the parabolic arcs. The velocity of the hand, before and after the catch, remained to be specified.

Since the ball actually strikes the hand as it is caught, there is an inelastic collision at the time of each catch. Thus the hand velocity will change as the ball is caught, according to the law of conservation of momentum. The velocity of the hand before the ball is caught is largely at the discretion of the juggler—the hand only needs to be below the ball. The velocity of the hand and ball together after the catch depends on the relative masses of the hand and ball:

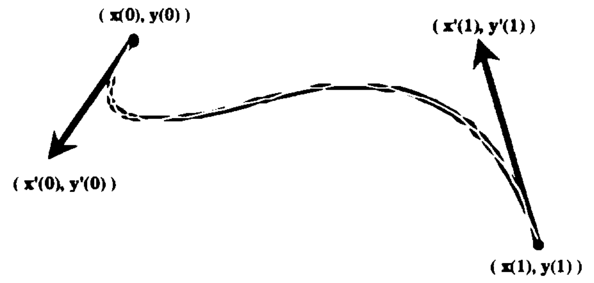
$$V_{after} = \frac{M_{ball}V_{ball} + M_{hand}V_{hand}}{M_{ball} + M_{hand}}$$



The inelatic collision between the ball and the hand

After a little experimentation, I decided that it was most reasonable for the hand to have zero velocity at the time of a catch. Thus the hand moves to the catch point and waits for the ball to land in it.

With the inelastic collision modeled, the hand motion was constrained to two unit time interval paths, with each path having its starting and ending positions and velocities known. I chose the Hermite form of the cubic polynomial to create paths that met these constraints. The resulting motion is both fluid and realistic.



$$\begin{cases} x(t) = (2t^3 - 3t^2 + 1)x(0) + (t^3 - 2t^2 + t)x'(0) + (-2t^3 + 3t^2)x(1) + (t^3 - t^2)x'(1) \\ y(t) = (2t^3 - 3t^2 + 1)y(0) + (t^3 - 2t^2 + t)y'(0) + (-2t^3 + 3t^2)y(1) + (t^3 - t^2)y'(1) \end{cases}$$

The Hermite form of the cubic polynomial

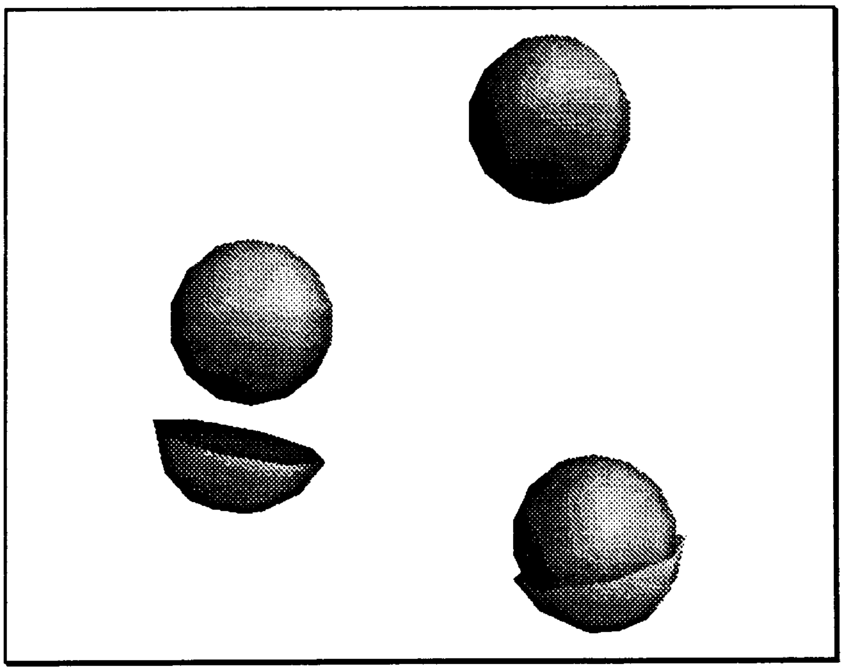
3. The GL Program

a. Man Page

```
jug(6D)                                                  jug(6D)

Name   jug - animate the juggling of three balls

Syntax
       jug  [-w  width]  [-h height] [-d distance] [-m massratio]
       [-s timestep]

Description
       jug is a GL program that  animates  two  hands  (shown  as
       green  cups) juggling three balls.  The effects of gravity
       and the collisions between the hands and  balls  are  mod-
       elled  correctly according to physics.  The program itself
       is not interactive, but all of the physical dimensions may
```

```
        be specified as command-line arguments.

Options
        -w width     Specifies the width of the juggling arcs.  The
                     default is 2.0.

        -h height    Specifies the height  of  the  juggling  arcs.
                     The default is 2.0.

        -d distance  Specifies  the  distance  between the throwing
                     point and the catching point of the hands.  In
                     effect,  this  is  the  offset between the two
                     juggling arcs.  The default is 1.2.

        -m massratio
                     Specifies the ratio of the mass of a  ball  to
                     the mass of a hand.  This is used to correctly
                     model the inelastic collision that occurs when
                     a  hand  catches  a ball.  The default is 1.0,
                     i.e., the ball and hands have the same mass.

        -s timestep  Specifies the time interval between frames  of
                     the  animation.   Larger  numbers give faster,
                     coarser animation.  The default is 0.05.   The
                     number  of  frames actually plotted per second
                     is dependent  on  the  machine  --  jug  plots
                     frames as fast as possible.

User Interaction
        After receiving sufficient enjoyment from the jug program,
        press escape to exit.

Author
        Paul E. Debevec (debevec@cs.berkeley.edu)  Fall 1992
```

b. Sample Screen



4. The UG Movie

The UG Movie produces similar output to that of the GL program, but the parameters can not be adjusted. It relies on the (cond)?(exp1):(exp0) construct to implement the six time intervals of the juggling pattern. In order to make use of macro expansion, the movie uses the #define C compiler directive. Thus the movie needs to be "compiled" by the C preprocessor in order to run. The unprocessed movie is called jug.cpp and the processed movie is called jug.ugx. The command "make jug.ugx" will perform the preprocessing.

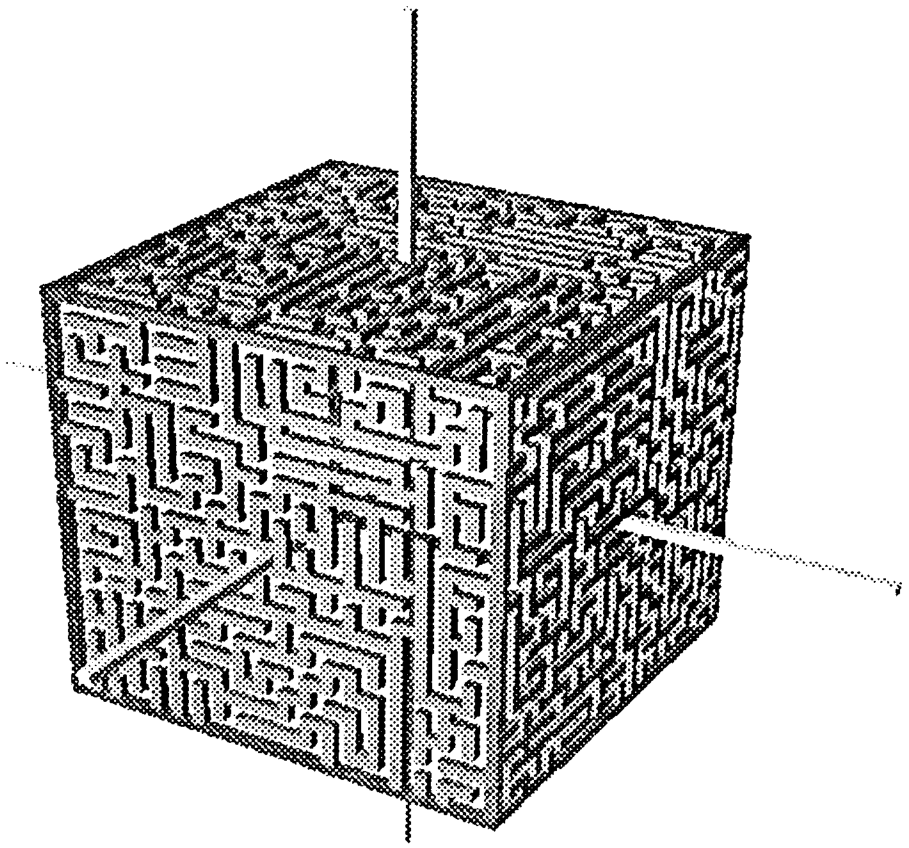
4. Conclusion

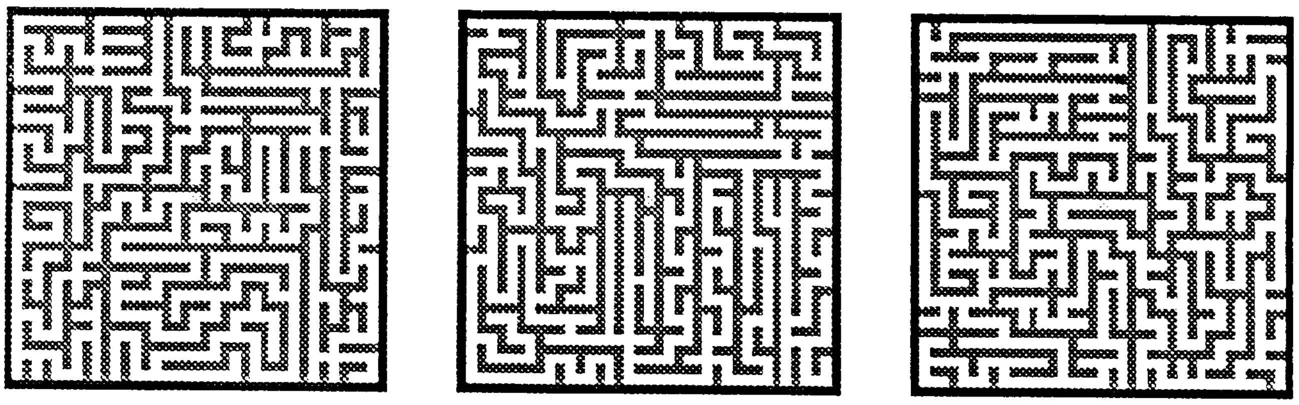
The animated output of the GL program looks very smooth and believable, but after a minute or so it can get boring since it is repetitive. The same can be said of a live juggler. To make things more interesting, the program should do what real jugglers do: tricks. The best way to implement tricks would be to define a language that can describe all of the tricks the juggler should be able to do. Then the juggling program could interpret and animate a script of tricks. It would also be worthwhile to model the juggler as some sort of being with a body, arms, and fingers, rather than the two disembodied cups that currently represent its physique.

AN N-DIMENSIONAL MAZE GENERATOR AND 3D MAZE GAME

by

Mitchell Deoudes





I. OVERVIEW

This project provides two distinct utilities. The first is a maze generator capable of producing n-dimensional mazes and (n-1)-dimensional projections of those mazes simultaneously and in quite a bit of generality. This is intended as a self-contained subpackage for inclusion in other software applications. The second utility is a viewer/manipulator for playing a 3D maze game. This portion of the project may be viewed as a sample of the type of application the generator code might included in. The maze game is written for three dimensions and is not a fully general viewer utility.

II.i AN N-DIMENSIONAL MAZE GENERATOR

The basis of the maze generator is a recursive "tree-growing" algorithm which considers the volume to be filled by the maze as an n-dimensional array of cells. Given a starting cell, the algorithm chooses some permutation of the possible directions to exit that cell. Then, it attempts to "jump" from the cell along each direction in turn. A legal jump is defined as one which does not introduce a cycle into the resulting graph (i.e. one that does not return to any previously visited cell). Recursion occurs upon successfully jumping to a new cell.

Note that if jumps are defined to be of length 2, the walls of the maze can be generated "for free". This is accomplished by adding both the new cell (old+2) and the cell directly between the new and old cells (old+1) to the path:

*	<	*	<	*	
v					
*	>	*	>	*	
				v	
		2	1	0	

To this basis a further constraint is added. The generator must maintain connectivity not only in the n-dimensional maze, but also in the (n-1)-dimensional orthogonal projections of the maze. In three dimensions, the significance of this is more clear: a maze is produced which can be physically manufactured as a hollow cube with the projected paths cut into its faces. (For more on this topic, see Section III.)

In order to satisfy this constraint, the definition of a legal jump must be changed. A jump may now be made if it does not introduce cycles in either the nD maze or any of the (n-1)D projections (note that there are n of these). This definition may be simplified by recognizing that any cycle in the nD path will be projected as a cycle in at least one of the (n-1)D paths. Thus, only (n-1)D cycles need be checked for.

The check for (n-1)D cycles is not as simple as the nD check, however. There are several cases:

1) The new cell (old+2) is not yet on the path for any face (projected maze) - the jump is legal.

2) Both the new cell (old+2) and the new intermediate cell (old+1) are on the path for all faces. This indicates that the new cell is actually the cell most recently exited - the jump is obviously not legal.

3) For some face, cell old+2 is on the path but cell old+1 is not on the path. This indicates the introduction of a (n-1)D cycle into that face - the jump is not legal.

4) At least one face falls into each of categories #1 and #2, but no face falls into category #3. This indicates that a legal path extension is being "cut" into some faces, while in the other faces the "current cutting position" is merely sliding back along a previously cut section - the jump is legal.

Surprisingly, the legality test can be made quite simple by careful consideration of the four cases. Case #2 is eliminated by never attempting to jump back along the direction most recently jumped from. Now only case #3 need be checked for - any jump passing the test for case #3 for all N faces is legal.

This scheme will produce mazes that completely cover each face for most small to medium size volume specifications. However, due to the projective connectivity constraint just discussed, it is not guaranteed that in general an nD maze will be generated such that each (n-1)D face is completely filled with path area. (Once again, this becomes clearer in the context of the 3D game.) Thus, it may be necessary to generate "bogus" path areas on the (n-1)D faces. This is handled quite nicely by running the same generator on each face, while constraining the case #3 checking to that face alone. This allows "sliding" forward along the paths already cut, prevents cycles, and allows all unvisited areas to be filled with bogus path.

Solution path generation is also elegantly handled by the same routine. Upon determining that the cell just added is a target cell (by comparison with a list of desired targets), the recursive function simply adds that cell to the solution path, and returns this information to the next level "up". In turn, the "parent" level adds its cell, and returns up one level. This continues all the way up the recursion tree.

II.ii GENERATOR CODE

The maze generator code is contained in the following files:

globals.h	maze data structure & defining specs
cube.c	init, data manipulation & main() routine
perms.c .h	jump direction permutation routines
carve.c .h	generator routines

The generator has been coded for n-dimensional generality. All data structures, manipulation routines, and procedure control flow schemes are defined based on the named dimensions specified in the top line of the "globals.h" file.

By passing the desired dimensions to the package, mazes can be generated to fill arbitrary volumes (i.e. the cube is a special case of the general box shape of the maze volume).

Only the three orthogonal projections are actually stored in memory, reducing memory requirements from O(n^D) to O(n^(D-1)), where D is the number of dimensions.

Also, cell content types are user-definable. The main generator routine will "carve" a path made up of any cell type, and will process (n-1)D faces to produce bogus paths that are distinguishable from actual paths.

III.i A 3D MAZE GAME

This portion of the project is a simulation of a 3D "Cross-and-Cube" game. A hollow cube with maze-carved wall is displayed. Within the cube is a 3D cursor in the form of a set of three mutually perpendicular bars, which extend through the path areas on the walls. The game is played by sliding the cross along the three principle directions and attempting to manipulate it into specified target positions (initially the corners of the cube).

The basic structure of the control routine is a simple event loop that detects user requests for menu actions (printing, displaying solutions, restart, etc.) and viewing manipulation mouse inputs. Within this loop are routines to detect game events (reaching the target cell, bumping into a wall), and display routines.

Unlike the generator, the algorithms involved in this section are not as interesting as the code structure.

III.ii GAME CODE

The routines for displaying the maze based on the n-dimensional constructs defined in the generator section, but are optimized for 3D and are thus not intended for use with higher order mazes. For instance, the cube drawing function has only one internal loop that handles all faces in a general manner.

Files are as follows:

cube.c	main() routine
light.c .h	lighting routines (low level)
draw.c .h	display routines (intermediate level)
play.c .h	special game routines (high level)

The most notable point from a technical standpoint is that the display routines key off the generalized organization of the maze data structure. User-defined cell contents types can be displayed as arbitrary objects, and are reconfigurable on-the-fly. This makes switching between display modes (normal, solution, target, etc.) trivial, and also allows for rapid customization of display aesthetics. In a similar vein, the special graphical sequences for various game events are implemented as generic function calls and are thus subject to user customization.

Also of note are the following speed optimizations. First, since the maze is stored sequentially in memory, a pointer may be quickly stepped along the data structure in an incremental fashion without performing address arithmetic at each step. This method is used in the maze display routines. The result is that these functions are made 3D specific, but also afforded a visible speedup. Additionally, only three planes of the cube and three sides of each cell need be displayed, since the viewing angle is constrained to a single octant of space. This reduces display time by approximately 75%.

IV. CONCLUSION

The major change that occurred during the design process was in the decision to store data and perform manipulations in (n-1)D instead of nD. Given the project to do over again, this is the area that I would devote more consideration to prior to starting in on the code. A more careful analysis of which operations are suited to which data structure might have been in order. Surprisingly enough, the remainder of the project went relatively according to plan.

Future work might include writing various higher-dimensional applications. Also, aesthetic improvements to the maze game interface, as well as the addition of various bells and whistles are directions that may be explored almost ad infinitum.

NAME

NeatOKeen N-Dimensional Maze Generator Demo and Game Thingy

SYNOPSIS

```
nok x y z s
```

x, y, and z are small integers indicating the dimensions of the maze. These values generally fall in the range 3-9. s is an integer to be used as a random seed.

DESCRIPTION

nok is a 3D maze game suitable for killing time waiting for compiles, or as a stress-relieving study break. nok currently runs only on the Silicon Graphics platform.

The player is presented with a 3D maze in the form of a hollow cube with maze-carved walls. The current position is determined by a "cross", or set of three intersecting axial-aligned bars which extend through the paths on the cube walls. The player must manipulate the cross from the starting position at the center of the maze to as many of the corners as possible in order to complete the game. nok will tabulate the total number of mistakes made while traversing the maze, and the total number of corner goals attained. From these two numbers, an integer score is calculated and printed out.

The cube maze is displayed in the major subdivision of the window. Auxiliary 2D views of the individual walls may be enabled, and will display in the lower portion of the screen. A small set of arrowhead coordinate axes are shown in order to facilitate interaction.

The user interface is intended to be self-explanatory to a great extent. However, the following is a summary of interaction procedures:

Keys - the arrow keys control cursor movement in the horizontal plane (the plane seen as a horizon in respect to the screen). Holding down the spacebar while using the up & down arrows controls movement in the vertical direction. When the user attempts a physically legal move, the cross will shift position accordingly. An attempt to move through a wall is indicated by a red flash. Reaching a target position is also indicated by a special graphical sequence.

Mouse - the mouse is used for manipulating the view of the main maze. The middle button performs rotations, while the left button performs a zooming operation. The right button calls an options popup menu.

Menu - the popup menu contains choices for all special functions. Included are: displaying 2D & 3D solution paths, producing hardcopy, etc.

Note that within the nok package is an n-dimensional general maze generator. For information on integrating the generator code into other applications, see the detailed nok code synopsis forthcoming.

Icosahedron Transformation by Rotation-Translation Operation

Nobuko Nathan

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Abstract

A collection of UNIGRAFIX files which demonstrate interesting transformations of an icosahedron are available. Hinge mechanisms are shown in each file to demonstrate how we can construct a real object. These files can be viewed by the ugmovie program.

1 Introduction

In his book, *The Geometrical Foundation of Natural Structure*, Robert Williams explains that all of the regular and eleven of the thirteen semi-regular polyhedra can be generated by rotation-translation operations from either an icosahedron, a snubcube, or a snubdodecahedron. I chose the icosahedron as a starting object and modeled its transformation to an icosidodecahedron and to a truncated dodecahedron.

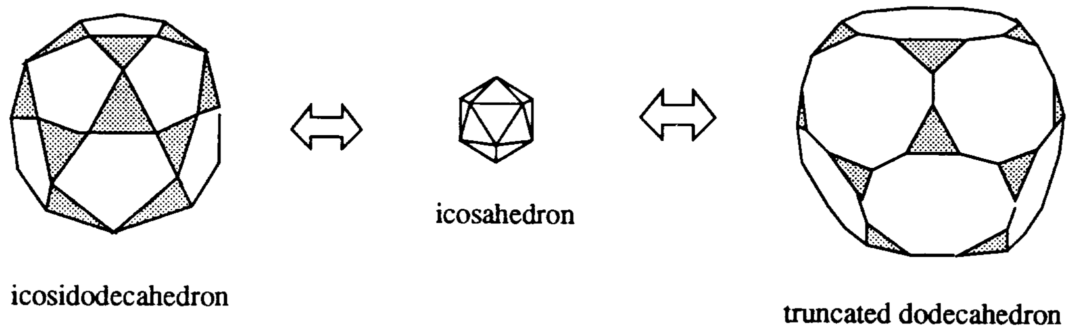


Figure 1: Two transformations from an icosahedron

2 Usage of the files

We begin by starting the ugmovie program. With the file/open menu, we load icosabug1.ugx file or icosabug2.ugx file. When we open the view parameter window, there are 5 variables which control the object.

thickness This controls the thickness of each triangle. The scale is in proportion to the distance of each face from the center of the icosahedron. Please note that when the thickness is zero, we cannot see the hinges which connect the triangles.

width For a better view, each triangle face can have a triangular hole in the middle. When the width is 1, there is no hole. When the width is zero, only the rim of the triangle is left.

phase This controls the transformation phase of the icosahedron. When the phase is 0 and 2, the icosahedron is closed. When the phase is 1, triangles form an icosidodecahedron with pentagonal faces missing if the file is icosabug1.ugx, or a truncated dodecahedron with decahedral faces missing if the file is icosabug2.ugx. In the latter case, the edges between missing faces will be provided by bars with hinges.

xyz For visual aid, the xy, yz, and xz planes can be placed in the scene. The chosen parameter controls the size of these three planes. When it is zero, these planes are invisible.

cone Each triangle moves in a pyramid-shaped space enclosed by three planes. By making this variable large, we can place one such pyramid in the scene. This is probably more useful for viewing the icosabug1.ugx file.

3 Mechanism of Transformation

3.1 Transformation to Icosidodecahedron

3.1.1 Rotation of Faces

To model the transformation between regular and semi-regular polyhedra, it is important to realize that the objects have a great deal of symmetry. Especially for the regular solids, all faces, edges, and vertices are equivalent. All the axes which go through the center of the faces of an icosahedron intersect one another at the same point. The distance from this point (I will call it the *origin* from now on) to all faces is the same.

During the rotation-translation transformation, each face rotates around one of these axes and also moves along this axis, maintaining the face perpendicular to it. The degree to which it rotates, as well as the distance it moves, should be the same among all the faces. For the object to be a real jitter-bug, some connection between faces must be maintained throughout the transformation in addition to the two conditions above. First, let us assume that we will maintain the connection at the corners of the triangles in an icosahedron. (This is modeled in the icosabug1.ugx file.)

When it comes to the direction of rotation, the symmetry becomes less clear with an icosahedron. I explain why and how to assign the rotation to each face in the following section.

The main goal of this project is to construct a model which we can actually build. By this, I mean that each face should maintain enough connection to other faces, and with the help of hinge mechanisms, there is only one degree of freedom left for the object. If we compare icosahedron and icosidodecahedron, it is clear that each triangle face in the icosahedron should maintain connection to three other triangles which border the triangle on its edges. Also, each face should rotate 60 degrees, (not 180) while moving outward for some distance. Unlike an octahedron, however, it is impossible to maintain all three connections for a face in the icosahedron because there is an odd number of faces coming together at one vertex. Figure 2 illustrates this.

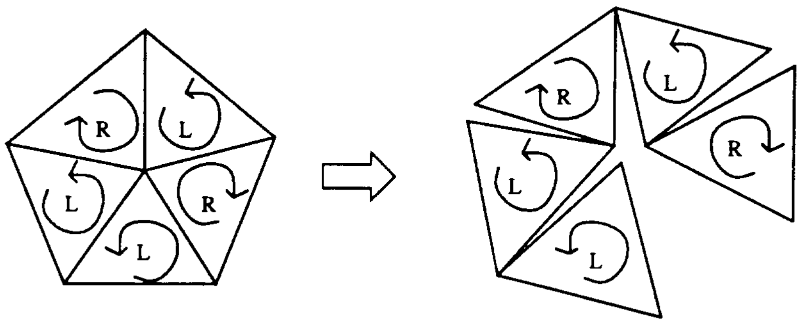
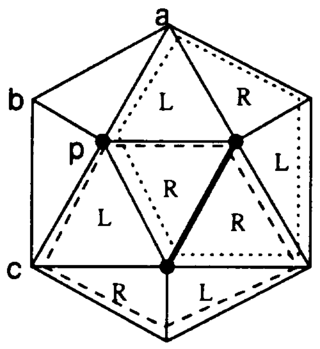


Figure 2: Rotation of triangles around a vertex in an icosahedron

As you can see, if two adjacent triangles rotate in the same direction, they will not maintain connection between them. The hinges are at the corners of triangles. Each edge corresponds to one hinge with the next face. Figure 2 suggests that at least one out of five edges would not have hinges and the connection which corresponds to the hinge will break during the transformation.

The remaining question is how to assign rotation to each face so that 1) Each triangle has at least two connections and 2) each *ring* of five triangles in an icosidodecahedron has at most one breaking point. These conditions came directly from the requirement that the object must have only one degree of freedom. (It naturally follows that the object must be connected and cannot have a *floating triangle* during the transformation.) We also want to maintain the maximum number of connections, if it is more than enough, and make the object as symmetrical (or regular) as we can.

Under these conditions, exactly one out of five edges breaks loose. If a vertex is an end-point of one of these edges, it should not be an end-point of another; otherwise, this ring will have more than one breakpoint. The assignment of rotation is equivalent to choosing edges in a graph so that they compose the minimal dominating set; all the vertices must be covered, but we want to avoid having edges that share an end point in the set. Figure 3 shows when we are ready to choose the second edge.

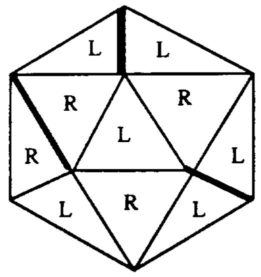


The triangles next to the dark edge do not maintain connection.
The five triangles around an end point of such edges already have one breakpoint.
The first edge determins the rotation of eight triangles.

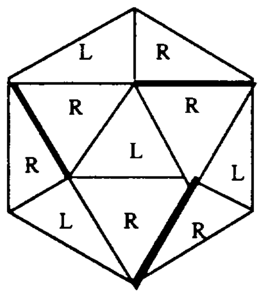
The next edge can be one of pa, pb, and pc.

Figure 3: Assignment of rotations to triangles

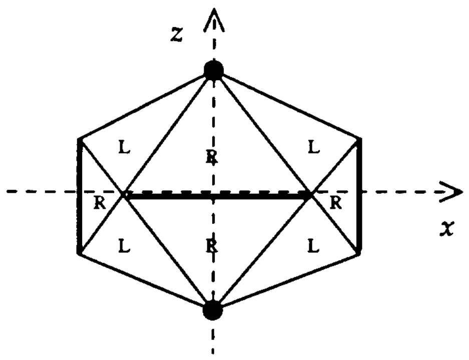
A scheme to assign such rotations to all icosahedron faces are described in Figure 4.



(A)



(B)



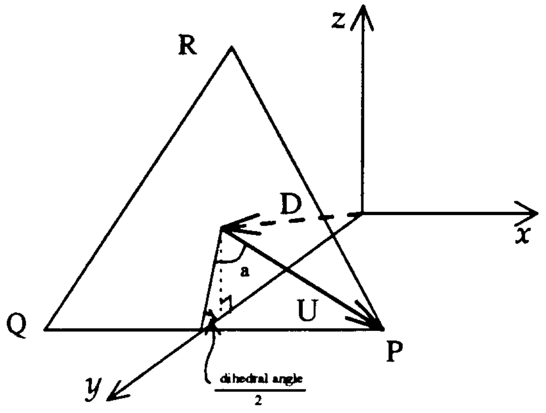
(B) from different angle

Figure 4: Choosing Edges

Note that both (A) and (B) are acceptable solutions, but (B) is more regular. Please note that all the breaking edges are perpendicular to each other in (B). The left-rotating faces will have all three connections while right-rotating faces maintain only two. The number of faces rotating in one direction, however, is different from the number of faces rotating in the other direction in this scheme.

3.1.2 Translation of Faces

In order to maintain the connection between two triangles during the transformation, the rotation and translation must be related to each other by some function. The relationship is obtained by solving the equations that express our constraints. Since all the faces are symmetric to each other we can choose one triangle in which it is easy to figure out its coordinates. In Figure 5, we have one of the center triangles in the drawing labeled *(B) from different angle* in Figure 4.



The coordinate of corner P is computed by adding vector D and vector U.
D is always perpendicular to the triangle and has the length of the distance between the triangle and origin.
U has a constant length and rotates around D.
By decomposing D and P into x, y, z, directions, we can express the coordinate of P in terms of distance and rotation.

Figure 5: Movement of a triangle

```
Let the length of D = d,
    the length of U = u,
    the half of the dihedral angle of icosahedron = H,
    and a = the rotation angle measured clockwise around D.
    (In this figure, a = -60)
  D = (0,    d*sin(H),    d*cos(H))
  U = (-u*sin(a), u*cos(a)*cos(H), -u*cos(a)*sin(H))
Thus, the coordinate of P is
  P = (-u*sin(a), d*sin(H)+u*cos(a)*cos(H), d*cos(H)-u*cos(a)*sin(H))    --(1)
```

While the triangle moves, P remains on the xy plane. That is, z = 0:

```
d*cos(H)-u*cos(a)*sin(H) = 0
d = u*tan(H)*cos(a)    --(2)
```

Q moves on a plane defined by the origin and the initial locations of Q and R. The triangle is symmetrical, however, and solving the constraints on P will suffice.

This is not the only way to express constraints. Interestingly enough, the functions I obtained by solving different sets of equations look very different from each other. Nonetheless, they are equivalent and all satisfy the requirements.

3.1.3 Hinge Mechanism

Two faces are connected to each other by a non-elastic hinge. Each hinge has two rotation axes which are held by a rigid triangular panel. The two axes are perpendicular to the face of an icosahedron they belong to, as is the triangular piece between the two faces. The triangular piece is originally perpendicular to the edge of the icosahedron face and it rotates 60 degrees in respect to the icosahedron face while the object transforms into an icosidodecahedron. The degrees of rotation of hinges are in proportion to the degrees of rotation of the faces of the icosahedron.

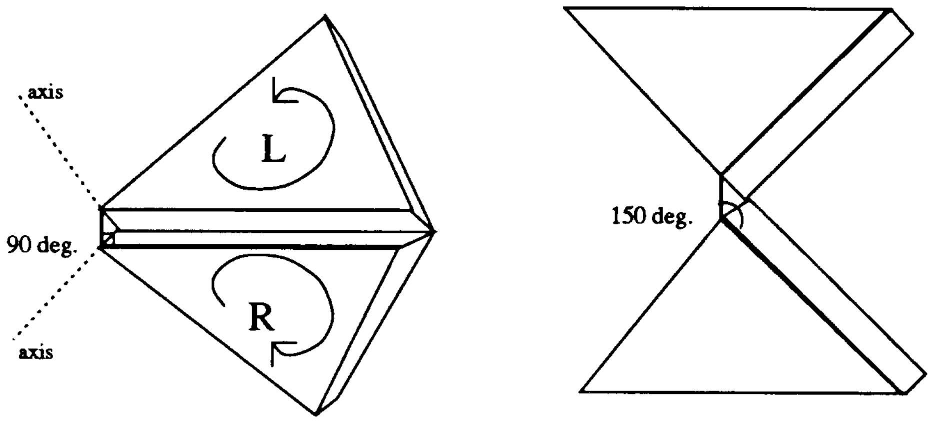


Figure 6: Movement of triangular hinges

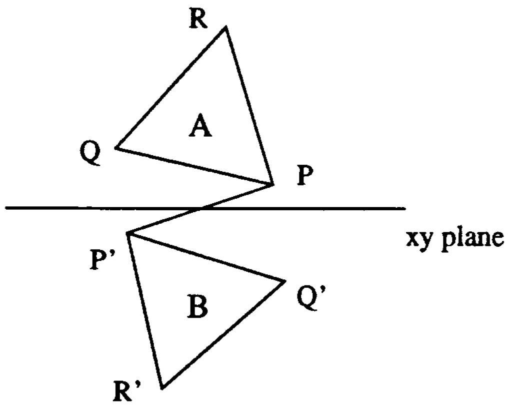
Please note that when a movement of an object is defined in respect to another object which also moves, we can define the movement of the first object in respect to the second in a ugfile, then define the second object so that it will include an instance of the first, and finally rotate and translate the whole object by the movement that the second object is defined with.

If we choose (B) in Figure 4, we can cover all the hinges by assigning three hinges to eight left-rotating triangles. This is another reason why I say that (B) is more *regular* than (A).

3.2 Transformation to Truncated Dodecahedron

3.2.1 Rotation and Translation of Faces

The transformation from an icosahedron to a truncated dodecahedron is achieved by maintaining the same distance between two triangles. This transformation is more regular than the other transformation we saw earlier; all twenty triangles of our icosahedron move in exactly the same manner including the direction of rotation. As for the other transformation, each face will rotate 60 degrees while it moves outward. Obviously, the distance it moves is larger than the first transformation. The constraints are expressed by the equations below. Thus, we can obtain the function from the amount of rotation of each face relative to the distance it moves. We can use the same expression (1) for the coordinate. Only the constraints are different.



Triangle A and triangle B are symmetrical around y axis. Thus, P and P' have the same y coordinates and their x and y coordinates have opposite signs.

Figure 7: Movement of triangles with a bar and hinges

```
The distance between P and P' is maintained the same as the length of PQ (=u*sqrt(3))
    sqr(u*sin(a)*2) + sqr{(d*cos(H) - u*sin(H)*cos(a))*2} = sqr{u*sqrt(3)}

  When we solve this equation in regards to d, we obtain
         d = u*tan(H)*cos(a) [+|-] u/cos(H)*sqrt[3/4 - sqr{sin(a)}]      --(3)
To move the face outward, the sign in the middle of this equation must be plus.
```

3.2.2 Hinges with Bar

The two faces of an icosahedron are connected by a bar with hinges in this transformation. The connector has four rotation axes. Each triangular piece is equal to half of the hinge part explained earlier in this paper and it rotates 60 degrees during the transformation in respect to the icosahedron face with the speed proportional to the rotation of faces.

The rotation of the center piece is NOT linear to the rotation of the faces, since the distance the face moves affects the angle it will make with the face. It needs to be computed from the same constraints we used to figure out the relationship between the rotation and translation of the faces. The bar can be expressed by the rotation around the axis which goes through the middle of the bar and the translation from the origin.

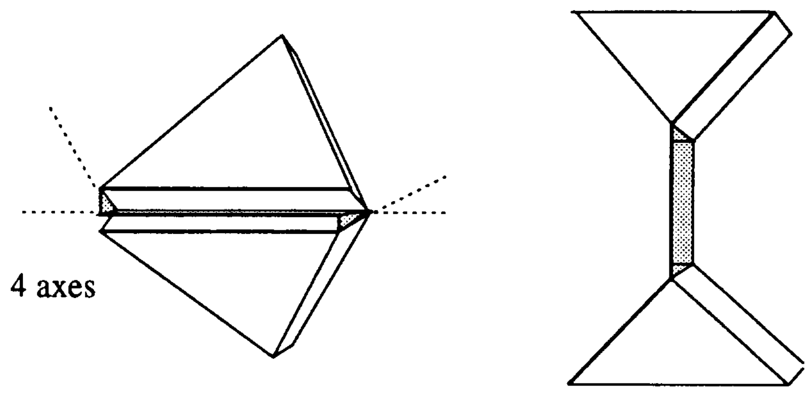
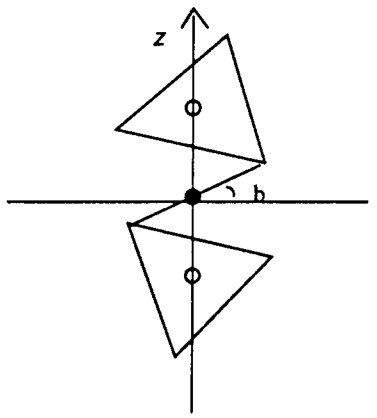


Figure 8: Movement of triangles with a bar and hinges



The bar rotates around the center while it moves forward along the y axis.

Figure 9: Movement of triangles with a bar and hinges

```
From the value of x and z coordinate for corner P as in (1)
   tan(b) = P.z/P.x = {d*cos(H)-u*sin(H)*cos(a)}/-u*sin(a)
By substituting d with the value in (3) and we get:
   tan(b) = {-sqrt[3/4-sqr{sin(a)}]/sin(a)}
Thus, b is expressed in terms of a by
   b = atan{-sqrt[3/4-sqr{sin(a)}]/sin(a)}  when a is not 0
   b = pi/2  when a = 0

The translation is the same as the y coordinate of the triangle corner.
   translation = P.y = d*sin(H)+u*cos(a)*cos(H)
By substituting d in (1) with the value in (3) we get:
   translation = u/cos(H)*cos(a) + u/cos(H)*sin(H)*sqrt[3/4-sqr{sin(a)}]
```

4 Conclusion

These two types of transformation and mechanism are the basis for all polyhedra transformations. For different regular solids, the value of dihedral angles are different, but the principle of analysis is the same. Transformation from the dodecahedron is an easy modification of these two. I would have liked to produce a series of transformations starting from other solids.

While I worked on this project, some problems with ugfiles became clear to me. The lack of global variables and definition of constants makes the file hard to read and the execution probably inefficient. Also, there is no easy way to refer to a part of an instance. For example, if we have a triangle defined as an object, we cannot use a vertex in its instance to make a new face. Solving these problems will make it much easier to use UNIGRAFIX files.

December 15, 1992

From Soccer Balls to Bucky Balls: Polyhedra Transformations

Department of Electrical Engineering and Computer Science

University of California

Berkeley, California 94720

CS285 Final Project
Sun-Inn Shih
Louis Yun

1.0 INTRODUCTION

Platonic and Archimedean solids are fascinating for their mathematical properties and their simple symmetric beauty. We chose to explore a unifying link between different family members - the transformation from one polyhedron to another by rotation and translation in three dimensions [1]. We achieved the transformations between solids by rotating about and translating along the same axis, while maintaining one connection between two faces. The connection can be either a vertex or a new edge.

There are three major paths transformations in our project and they all go through the same intermediate stage: the $3^4.4$ solid. The paths are labeled in Figure 1. The user can view the transformations by manipulating a hexslider. It is a perspective viewing environment with Goraud shading. The user can change the viewpoint by using the left mouse and right mouse for object rotation and zooming respectively. We will next discuss how each path is calculated. All calculations follow two basic concepts:

- explore symmetry.
- identify constraint.

It will become clear why these two principles enable us to calculate our solutions systematically.

2.0 TECHNICAL DESCRIPTION

Path 1.

In this path the connection between faces are maintained by a new edge, AD. Let's follow our two guidelines:

1) *symmetry:* All the square faces are rotated images of each other. Therefore, we only have to find the rotation angle θ about the z-axis and the translation R along the z-axis of the top square. Likewise, the wire (edges) are all rotated images of each other. Therefore, we need only consider the wire that pierce through the yz plane, and is in the +z and +y quadrant of the space (see figure below). This wire rotates about (0,1,1) with angle θw, and translates along (0,1,1) by Rw.

2) *constraint:* AD has to maintain length one.

Solve: by representing the x,y,z value of point A and point D, we can solve for R(θ):

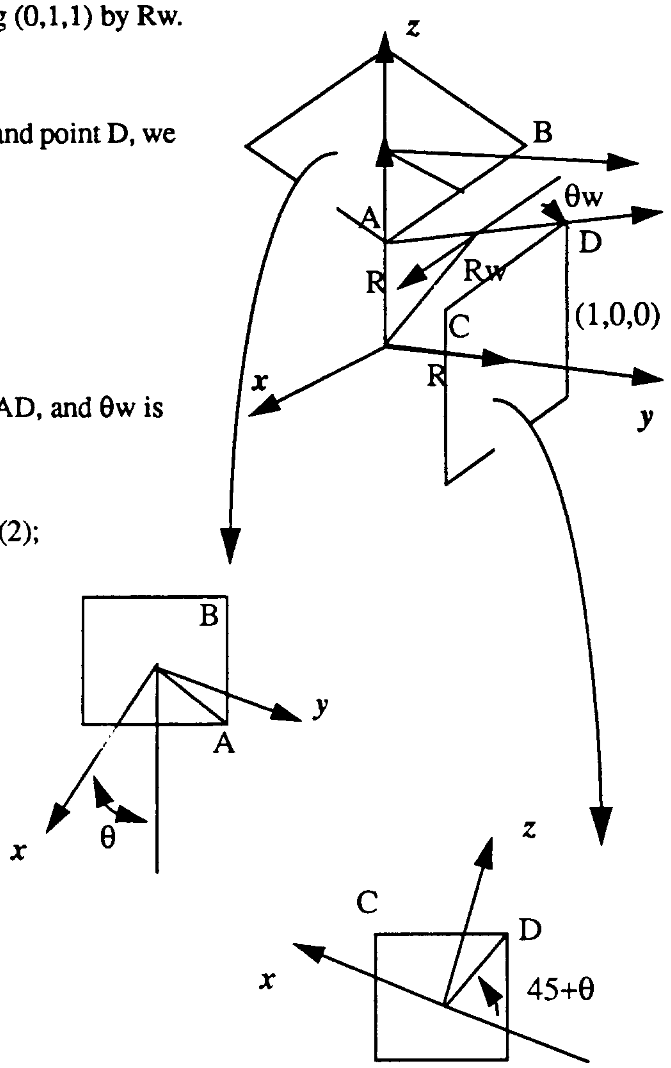
ǁA-Dǁ = 1; substitute A and D and we can get:

R(θ) = (cosθ+sinθ)/2 + sqrt(sinθ*cosθ)

Note the wire is rotating about the axis(0,1,1);

Rw is the distance of origin to the mid point of AD, and θw is the angle between AD and (1,0,0) vector:

Rw(θ) = sqrt((R+sin(45+θ))^2)/sqrt(2);
θw(θ) = acos(dot(A-D,(1,0,0)) = cos(45+θ)/sqrt(2);



A: x = cos(45+θ)/sqrt(2);
y = sin(45+θ)/sqrt(2);
z = R;
D: x = -cos(45+θ)/sqrt(2);
y= R;
z= sin(45+θ)/sqrt(2);

Path 2:

In the path, again the connection between faces are maintained by a new edge: TS.

1) *symmetry*: All triangular faces are rotated images of each other. We only have to find the rotation angle θ about axis: (1,1,1) and the translation R along this axis of the triangle in positive x,y,z quadrant. The wires are also rotational images of each other. We only need to calculate Rw and θw of one wire. We choose the wire that pierce through the positive x and z plane. It is rotation about axis (1,0,1);

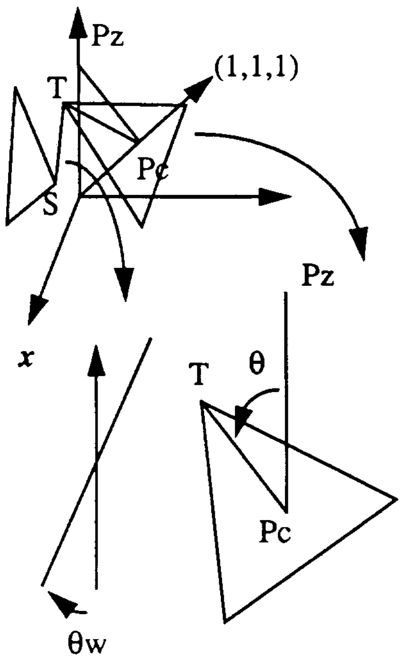
2) *constraint*: The new edge TS has to have length one always.

Solve: by representing the x,y,z value of point T and point S we can solve for R(θ).

It is a little trickier to find the x,y,z value of T and S. We have to note that:

a. T is on the plane A: x+y+z=R*sqrt(3);
b. the plane A hit z axis at Pz: (0,0,R*sqrt(3);
c. The center of the triangle T is

Pc: (R/sqrt(3),R/sqrt(3),R/sqrt(3));



Let Vz be the unit vector from Pc to Pz, and Vt be the unit vector from Pc to T (xt,yt,zt). The three equations we have to solve is:

a. cross(Vz,Vt)= sin(θ)/sqrt(3)* (1,1,1);
(two independent equations)
b. dot(Vz,Vt)=cos(θ);

We can therefore find:

xt = (2*sqrt(3)*R - sqrt(2)*cos(θ) + sqrt(6)*sin(θ))/6;
yt = (2*sqrt(3)*R - sqrt(2)*cos(θ) - sqrt(6)*sin(θ))/6;
zt = (sqrt(3)*R = sqrt(2)*cos(θ))/3;

similarly:

xs = (sqrt(3)*R = sqrt(2)*cos(θ))/3;
ys = (-2*sqrt(3)*R + sqrt(2)*cos(θ) + sqrt(6)*sin(θ))/6;
zs = (2*sqrt(3)*R - sqrt(2)*cos(θ) + sqrt(6)*sin(θ))/6;

finally: ||T-S||=1 and solve for R(θ).

R(θ) = (sqrt(2)*sin(θ) + sqrt(2/3)*cos(θ)+ sqrt(2*sin(θ)^2 + 6*sin(θ)*cos(θ)/sqrt(3))/2

Again, the Rw is the distance from origin to the midpoint of TS and θw is the angle between TS and (0,0,1);

Rw = sqrt(24*R^2 + 1 + 2*sin(θ)^2 + 4*sqrt(6)*cos(θ)*R + 4*sqrt(18)*R*sin(θ)+ 2*sqrt(3)*sin(θ)*cos(θ))/6;
θw = acos(cos(θ)-0.577*sin(θ);

Path 3:

This path is a little different. The two adjacent faces are connected by common vertices.

1) *symmetry*: we can use symmetry to isolate our calculation to only concentrate on the top square and the side triangle;

2) *constraints*: Ct = Cs.

Thanks to the calculation of the last paths, we can readily obtain Ct and Cs:

xt = (2*sqrt(3)*Rt - sqrt(2)*cos(θt-60) + sqrt(6)*sin(θt-60))/6;
yt = (2*sqrt(3)*Rt - sqrt(2)*cos(θt-60) d- sqrt(6)*sin(θt-60)/6;
zt = (sqrt(3)*Rt = sqrt(2)*cos(θ))/3;

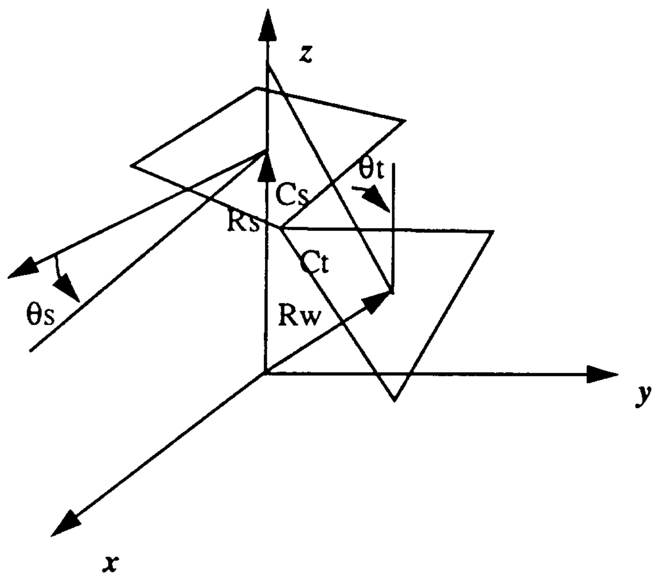
(there is a 60 degree difference in the starting angle of the triangle)

Cs = (sin(θs)/sqrt(2), xy = cos(θs)/sqrt(2), xz = Rs);

Note that we now need Rt(θt), Rs(θt) and θs(θt). Fortunately, with θt given, we have three equations and three unknowns.

Solution:

Rt(θt) = cos(θt-60)/sqrt(3) + sqrt(2*cos(θt-60)^2 + 1)/2;
Rs(θt) = Rt/sqrt(3) + sqrt(2)*cos(θt-60)/3;
θs(θt) = acos(2*Rt/sqrt(6) - cos(θt-60)/3 - sqrt(12)*sin(θt)/6;



3.0 CONCLUSIONS

It was immensely satisfying to be able to visualize all 6 polyhedra continuously deform from one to another using our program, vindicating the math framework outlined here. Things that we would do differently the second time around: we calculated the rotational motion along each of the 3 paths independently of each other. We should have realized that a constraint ties all three paths together: at the center position (the position of the snub cube), all three paths share the same object, namely the snub cube, with one common rotational orientation. Hence, it would have made sense to fix the object orientation at mid-time to be the same for all three paths.

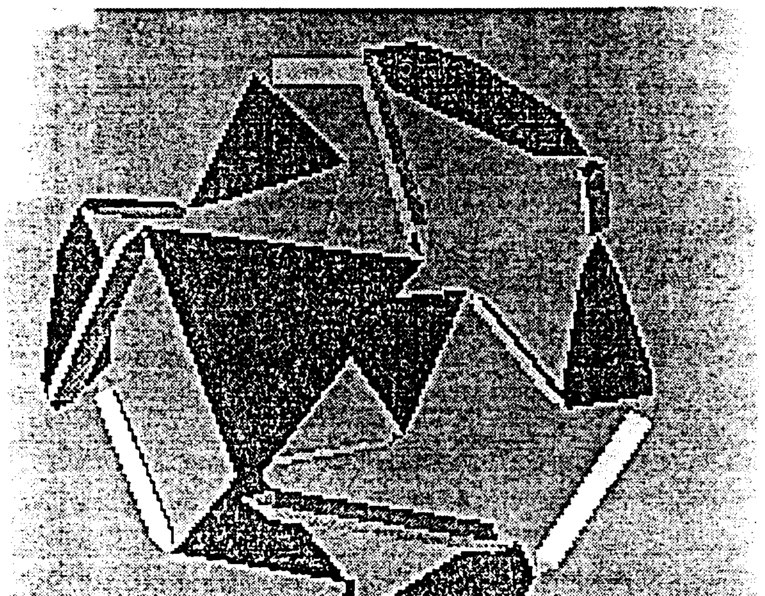
An interesting extension of this project would be to investigate the transformation between *4-D* Archimedean and Platonic solids (if it exists!) and how to visualize it by using 3-D and 2-D projections. But this is a question best left to the next CS285 class.

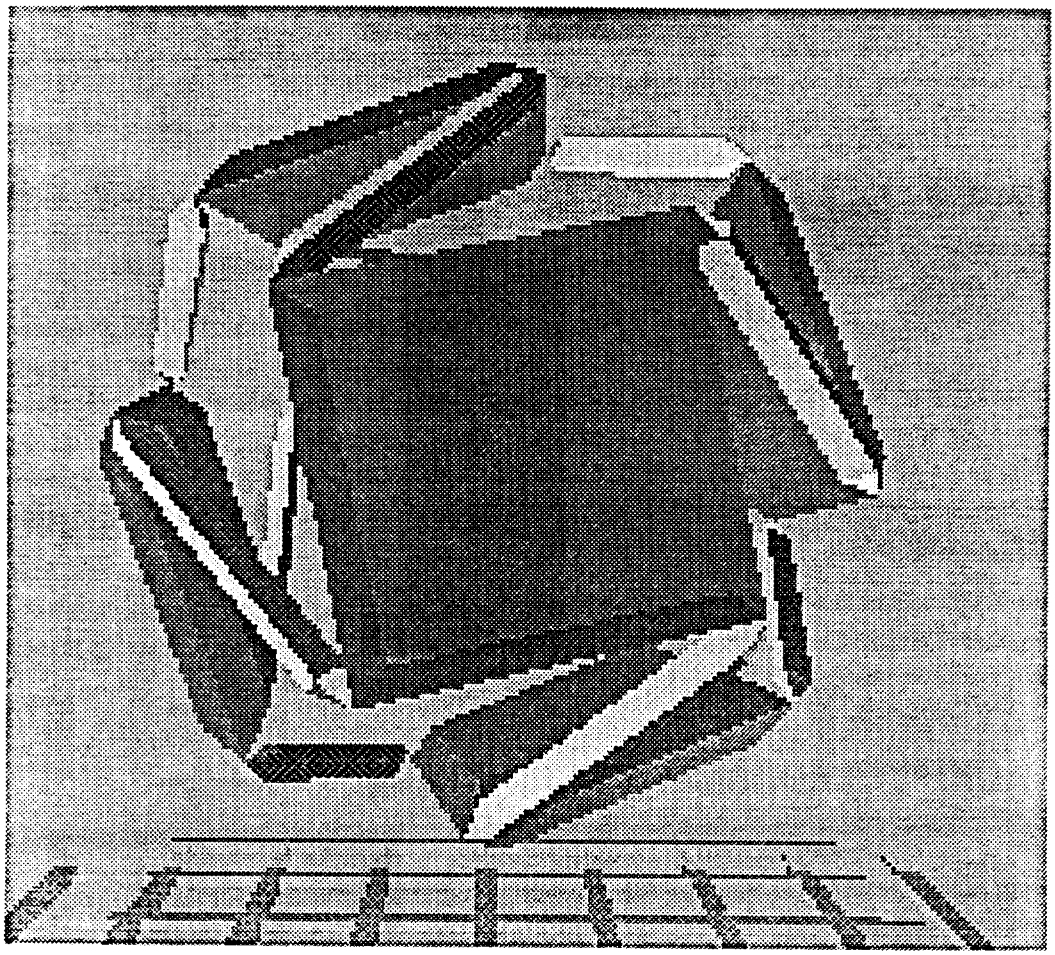
4.0 ACKNOWLEDGEMENTS

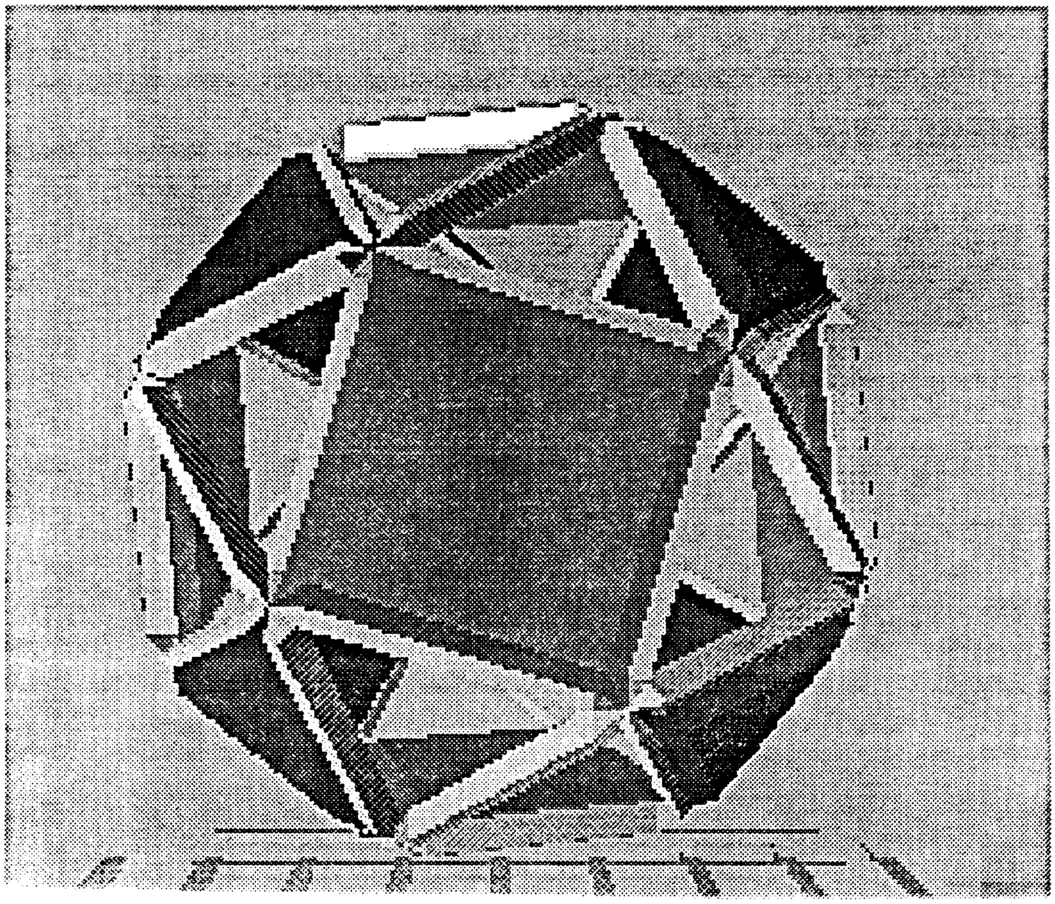
We wish to thank Tom Chen-Ming Lee for providing the hexslider widget for this project; after a few nudges it performed beautifully.

5.0 REFERENCES

[1] Robert Williams, *The Geometrical Foundation of Natural Structure: A Source Book of Design.*







UGMORPH: A Polyhedral Morphing Tool

Raph Levien and Daniel Rice

December 15, 1992

1 The UGMORPH Tool

UGMORPH is a Unigrafix-based tool which takes two star-shaped polyhedral objects, each annotated with the coordinates of a point known to be within its kernel, and produces an object whose topology is a merger of that of the input objects. This object's vertices may be interpolated from positions on the first object to positions on the second, resulting in a smooth transition between the objects. The output is written in .ugx format so as to allow animations by means of ugmovie. An example of a morphing sequence is shown in Figure 1.

2 The Importance of the Kernel

Recall that a star-shaped object is one for which the entire interior is visible from at least one point. The kernel of an arbitrary polyhedral object is given in general by the intersection of the inner halfspaces of each of the faces, which may of course be empty. UGMORPH provides a companion tool, UGKERNEL, which interactively displays slices of the kernel and allows a kernel point to be chosen by means of the mouse. It also allows the user to visualize the consequences of his or her center selection by showing the object's spherical projection.

Since there exists a simple halfspace test to determine whether a point lies inside the kernel, it does not appear worthwhile to explicitly construct the kernel polytope. Rather, the chosen central

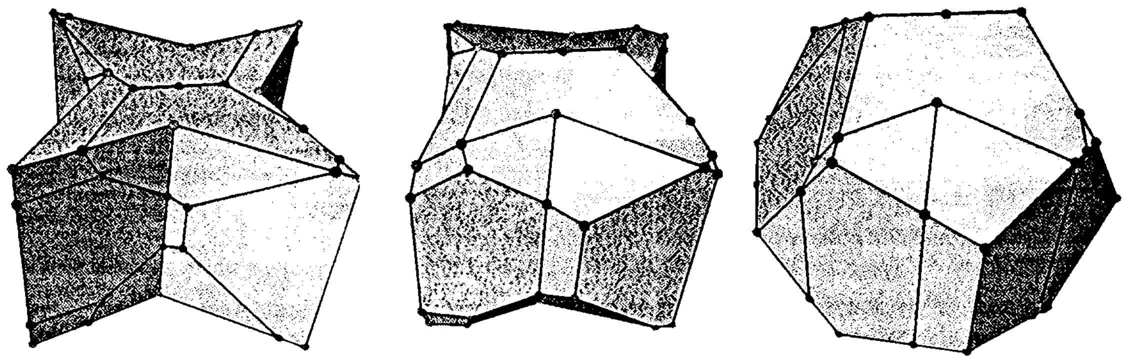


Figure 1: Three Stages of a Morphing Sequence

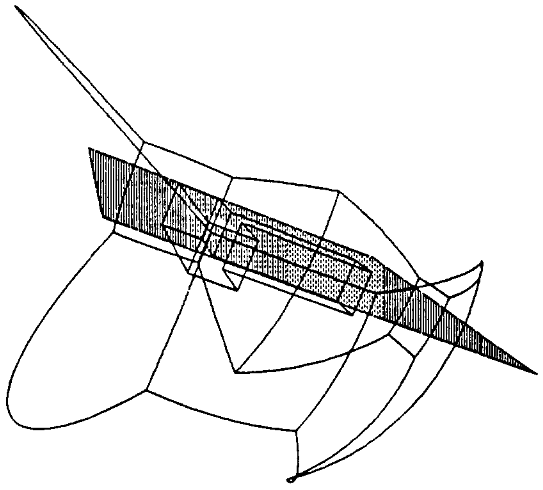


Figure 2: The Kernel Selection Interface

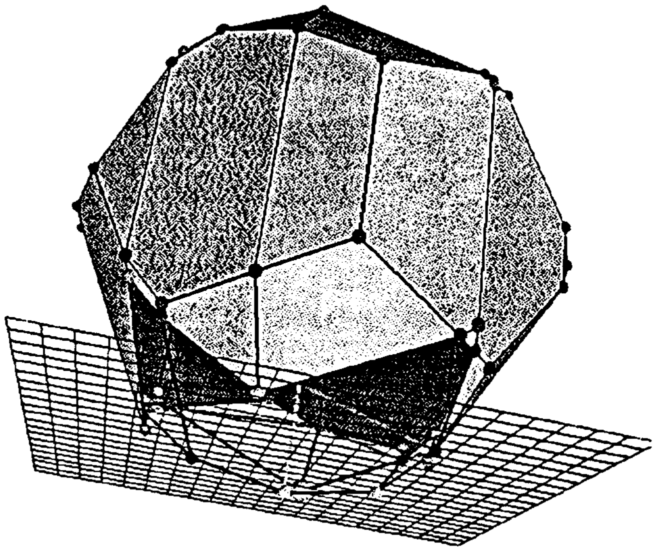


Figure 3: The Interactive Sweep-Plane Display

point is displayed in green whenever it is found to be within the kernel, and red when is passes outside.

As a further aid to selecting a kernel point, the user interface performs a graphical (image-space) construction of the intersection of the kernel with a user-selected plane. The plane itself is displayed as a rectangle large enough to contain the bounding box of the object. Each face of the object is considered in turn, and the intersection of its outer halfspace with the rectangle is drawn as a semi-transparent polygon. The union of the outer halfspaces, by DeMorgan's law, is exactly the complement of the intersection of the inner halfspaces, i.e., the complement of the kernel. The intersection of the rectangle with a halfspace is drawn by considering the vertices and edges of the rectangle in counterclockwise order, and emitting a vertex whenever we encounter either a vertex within the halfspace, or an edge which crosses the halfspace. In the latter case, the intersection of the edge with the halfspace is computed and emitted. One must take care in this process not to emit an empty polygon, which wreaks havoc with some versions of the Silicon Graphics pipeline.

Once a kernel point has been identified, the objects are conceptually projected onto the surface of a sphere centered at that point. Since the object is star-shaped, points on the sphere will correspond to points on the object's surface in a one-to-one manner. The new spherical polyhedron will have vertices, edges, and faces corresponding to those of the original object. The user interface contains various bells and whistles to help users visualize this spherical polyhedron. An example is shown in Figure 2.

Consider projecting both objects onto a common sphere. This sphere will now have vertices at the locations of all of the vertices of the two original objects as well as at the points where edges from the objects cross. The faces of the merged projection are the intersections of pairs of faces, one from each object. The main challenge of UGMORPH is to identify the new vertices and faces in an efficient manner.

3 The Morphing Algorithm

Numerous algorithms are possible for constructing the merged morphing topology, ranging from pure brute-force, with a running time of $O(nm)$, where n and m are the number of edges in the respective input objects, to more sophisticated approaches with running times on the order of $O(n \log n)$. The fundamental observation is that the projected edges of each object will tend to be distributed around the sphere more or less evenly. Accordingly, it should be possible to reject most of the edges of one object as incapable of having any intersection with a given edge of the other object without any computation.

3.1 Kent, Carlson, and Parent's Algorithm

The technique described in [1] requires that the input objects be triangulated, and identifies which faces of one object are intersected by a given edge of the other by a process of "walking triangles." Topological information about the adjacency of triangles is thus used to disambiguate the processing of coincidences and near- coincidences. Although this technique appears to work, and does not seem too hard to implement, we chose to use a different technique based on sweeping a plane through the projection sphere. Our technique does not require triangulation.

3.2 The Sweep-Plane Algorithm

Conceptually, our sweep-plane algorithm is simple. We wish to identify which edges intersect other edges without performing a brute-force $O(n^2)$ search. By maintaining an active edge list containing exactly those edges which cross the sweep-plane, we can limit our intersection tests to members of the list. Initially, we place all of the projected vertices of the original objects onto a priority queue, sorted by maximum z value. As we remove a vertex from the queue, any downward edges emanating from that vertex are added to the active list; edges emanating from the vertex which point upwards are removed from the active list. Each new edge is compared with the other active edges for potential intersection, and the points where such intersections are found are installed as new vertices, and placed into the queue for later processing. In the absence of coincidences and numerical uncertainty, this algorithm would suffice to determine all edge-edge intersections.

An advantage of a sweep method is that it allows topological information to be maintained incrementally, rather than having to be reconstructed *en masse* at the end of the algorithm. In particular, face information is maintained by ordering the edges of the active list by the longitudes of their sweep-plane crossings. As new edges are added to the active list, they are placed in their proper place immediately, and as they are removed, faces of the merged object are constructed and output.

3.2.1 Spherical Geometry

The morphing algorithm makes very heavy use of spherical geometry or, more precisely, the geometry of vertices, edges, and faces on the surface of the sphere. A large part of the code is devoted to standard geometrical and topological algorithms that have been adapted to the surface of the sphere.

There are essentially two ways of performing geometry on the surface of a sphere. The first is by halfspaces, the second is by the longitude/z-coordinate pair. Both methods have their advantages. Halfspaces are simple, efficient, and numerically fairly robust. On the other hand, many operations of interest are limited to the sweep plane itself, which has a constant z coordinate. Thus, the longitude/z-coordinate method reduces the problem to a single dimension. We use both techniques where appropriate.

The code which performs intersection tests is based on several key observations. The most important is the fact that we are performing a projection from the center of a sphere, so the projected edges are just segments of great circles, which have many nice geometrical properties. A great circle can be represented by the normal vector of the unique plane which passes through it, and points on the sphere may be classified as lying above or below the great circle by a simple dot-product test. The standard algorithm for determining whether line segments in the plane intersect does not carry over into the great-circle world without modification: two arcs of great circles such that the endpoints of each lie on opposite sides of the other may not intersect. In order to make the test strictly correct, some longitude comparison must be added. There is also a robustness problem on the sphere which is not present in the planar case. We discuss this further in Section 3.3.

Maintaining the so-called "cut ring," which stores the coordinates of the crossings between active edges and the sweep plane in sorted order proved to be tricky. We maintain sorted order to facilitate face reconstruction. The basic algorithm to place new edges into the cut ring is to consider neighboring pairs of edge crossings and to test whether the new crossing lies in between. The major source of difficulty lies in introducing a crossing which lies exactly 180° apart (within the limits of floating-point precision) from an existing one, as we shall discuss later. Since there can be only a single instance of this configuration within any given cut ring, we treat it as a special case. All of the other pairs are considered first, and only if all of these test fail do we insert the edge between the obtuse pair.

An interesting case occurs when an edge projects onto a "bowed" great circle segment – that is, one whose maximum z value does not occur at an endpoint. For such edges, we compute their maximum z value and introduce a new vertex at that point. This behavior is somewhat undesirable, since it introduces an artifact of the particular orientation of the inputs into the output. A better approach is to introduce a "virtual" vertex to ensure that the edge is brought into the active list at the appropriate time, but to maintain the edge as a single entity. This complicates the maintenance of the ring of crossings, since a single edge may now cross the sweep plane more than once.

3.2.2 Internals

The morphing algorithm maintains five important data structures:

- The vertex, edge, and face topology, in similar fashion to Unigrafix
- The vertex schedule (queues events for the sweep plane)
- The list of active edges
- The cut ring of active edges, sorted in counterclockwise order
- The contours of the result faces, as they are being constructed

The cut ring is similar to the active edge list, but differs in two important respects. First, it is sorted. Second, a single edge can correspond with two cuts, as in the case of a bowed edge.

The sweep plane algorithm works by processing all the vertices in order of increasing z coordinate (south to north). We use a priority queue to represent the vertex schedule. To process a vertex, we delete edges incident on the vertex that are below the sweep plane, and insert edges incident on the vertex that are above the sweep plane. To insert an edge, we insert it into the active list, insert the cut or cuts into the cut ring, and calculate intersections with other edges already in the active list. If there are any such intersections, the new intersection vertices are created and placed in the vertex schedule, and the edge and vertex topologies are merged. The last stage in processing a vertex is linking the contours of the result faces. If any such contours become circular, the contour represents a complete face, which is then recorded in the appropriate data structure.

Other important geometric algorithms include:

- Find the two adjacent cuts in the cut ring which enclose a given vertex (used to identify opposite face for back projection)
- Find the two adjacent cuts in the cut ring which enclose a given cut (used to insert new cuts into the cut ring)
- Identify bowed edges, and generate extremal vertices
- Link partial face contours

Back projection is the final step in the morphing algorithm. Each vertex must be assigned a starting and ending position such that it starts on the surface of object A and ends up on the surface of object B. To accomplish this, we determine which face the vertex lies within using topological information, and project it onto the plane of that face.

Although each face of the merged object begins and ends as planar (since it lies on the surface of the input objects), interpolating the vertices linearly between the starting and ending positions does not necessarily maintain planarity. Accordingly, we see two options: triangulate the faces, or assign nonlinear trajectories to the vertices such that planarity is maintained. The Iris GL seems to handle nearly planar faces well enough in practice that we have not had to implement either of these techniques.

3.3 Edge Intersection Testing

The edge intersection test illustrates the problems of working in spherical geometry. At first it would seem that two edges can be tested for intersection by simply testing whether the endpoints cross each other's great circles, in analogy to the planar case. However, this test might falsely report the intersection of two edges on opposite sides of the sphere. When we consider the additional insight that no pair of edges can intersect unless their longitude/z bounding boxes overlap, we can accurately classify all pairs of edges as intersecting or not. The z coordinate portion of the bounding box test is implicit, as no two edges can be in the active list at the same time unless they both cross the sweep plane. The longitude comparison is performed explicitly. Only when the longitude ranges of two edges overlap are the endpoints checked for crossing the halfspaces. In this particular case, the hybrid approach also increases the efficiency, because most edge pairs can be rejected immediately as the result of two longitude comparisons.

The edge intersection test also illustrates a robustness problem with spherical geometry algorithms: the case when a cut is to be inserted between other cuts separated by exactly 180°. In this case, neither the half-space nor the longitude gives an unambiguous answer. It is important to treat this case very carefully.

4 What We Would Do Differently

Although the model of projecting objects onto a sphere is conceptually simple, it requires that all geometrical computation take into account a number of quirks of spherical trigonometry, which are often somewhat unintuitive. It might have been simpler to "paint" the faces of one object directly onto those of the other, using an approach more like that of [1].

At the moment, the kernel selection process and the morph tool itself are separate programs which duplicate each other's code significantly. We hope to integrate the two into a single tool, which will allow users to see the effect of their center point selection on the final morphed object rapidly.

The spherical geometry algorithms gave us a lot of problems. It probably would have been a good idea to work out, carefully and thoughfully, which algorithms we would need and how they could be implemented robustly and efficiently. We delayed these design decisions because we were not familiar enough with the sweep-plane algorithm and the spherical geometry. Most of the remaining robustness problems stem from the "180° coincidence," as we have termed it.

One of the early design decisions was to implement the sweep plane algorithm rather than the one presented in [1]. Our approach has some advantages, most notably that it eliminates the need to triangulate the objects prior to morphing them. In any case, we had more problems with the basic spherical geometry than the sweep plane itself, which we would also have needed for Kent's algorithm. It is plausible that our algorithm was not significantly more difficult to implement than theirs.

We maintained an interactive 3D display throughout the development of our program. This helped us a great deal. For example, we had a small typo in the priority queue code. We discovered the problem almost immediately when we saw the sweep plane moving backwards. Without the large amount of information provided by this display, these types of problems would have been much harder to track down. A screen image of the interactive display in action is shown in Figure 3.

References

[1] Kent, James R., Carlson, Wayne E., and Parent, Richard E. Shape Transformations for Polyhedral Objects. *Computer Graphics (Proc. SIGGRAPH '92)*, 26, 2 (July 1992), 47-54.

Stone Wall Generator

A mini project for CS 285 --- Fall 1992

By
Luis O. Porcelli
Thurman A. Brown

The stone wall generator is an interactive tool used to create interesting stone wall patterns. The program has a simple user-interface, which allows the user to create a realistic stone wall pattern. There are various parameters which can be adjusted to change the resulting outcome of the generator. The program renders the stone wall in a GL window. The user can create a UNIGRAFIX file of the final pattern by selecting the UGout option or take a shap snot to generate a texture file.

RUNNING THE PROGRAM

To start the program the user enters:

stone_wall <wall_width> <wall_length> <stone_width> <stone_length> <stone_width_variance> <stone_length_variance> <+d> <+s>

All of the options in the stone generator have reasonable defaults, so the program can be run without any parameter. If desired, most of the parameters can be set from the command line. Wall_width and wall_height determine the width and length of the stone wall to be generated. Stone_width and stone_length, determine the average width and length of the stone to be created. The user is constrained not to make the stone_width and stone_length greater than the wall_width and wall_length respectively. Stone_width_variance and stone_length_variance specify the variance of the width and length of the stone respectively. The variance value is a percentage of the average stone_width and stone_length. So a width_variance of 0.5 would allow stones to vary in width size anywhere from 1/2 the average stone size up to 1.5 times the average stone size. Specifying the +d option causes the program to displace the initial wall pattern before it is displayed. Specifying the +s options causes the program to shrink the initial stone pattern before it is displayed. After starting the program, a risizeable GL window will appear in which the stone wall pattern will be displayed. A user-interface window also appears which allows the user to set the program options and perform the various operations.

USER INTERFACE

There are two interaction windows for the stone wall generator program. The first window is a GL window. The stone wall pattern is displayed in this window. The user uses the left mouse button to scale the object, the middle mouse button to rotate the object and the right mouse button to translate the object. This interface uses the same conventions as UGIRIS. The other interaction window is the user interface. The user interface is divided up into four sections: the stone options, the wall options, special values and the buttons.

STONE OPTIONS

The stone options are located at the top of the window. To the left are three input fields that allow the user to input the width, length and height of the wall respectively. To the right of each input field is a slider. The sliders range from zero to one, and allow the user to change the variance for the width, length and height values. For example, to set a new average width of 8 units and to set the stone width variance to 0, the user would locace the stone width field at the top of the window, change the value in this field to 8.0 and then slide the slider to the right of the stone width field to zero, thus setting the stone width variance to zero. Note, the change in the stone options, the wall options and the special features options do not get implemented into the stone wall until the restart button is pressed. This button generates a new stone wall pattern based on the new input values.

WALL OPTIONS

The wall options are below and to the left of the stone options. In the wall options section there are two input fields, stone width and stone length. These fields let the user change the overall width and length of the wall pattern.

SPECIAL VALUES

To the right of the wall options are located the special features options. This section consists of three input fields: seed value, epsilon and roughness value. The seed value input field allows the user to specify the seed that is used to generate random numbers used to displace the original node diagram and to generate the stone height. The epsilon value is used to set the size of the smallest allowable stone width and stone length in the initial wall pattern. This value defaults to one tenth of the minimum of the average stone length and width. The user can change this value to prevent small narrow stones from appearing; all stones that would be generated smaller than epsilon are merged with their neighboring polygons. The roughness value is used in generating the fractalized triangles that approximate the surface of the stone. The roughness value can range from 1.0 to 2.0. A value a 1.0 generates a smooth stone with small variations in the surface whereas a value of 2.0 generates a much more irregular stone surface.

BUTTONS

The final section of the user interface contains the buttons to perform the various stone generating operations. There are eight buttons and they are primarily arranged in the order in which they should be used. The buttons are named, starting at the top and going down the left and then the right column: Restart, Displace, Shrink, Subdivide, Edges, Stone Color, UG out and Quit.

RESTART BUTTON

Restart is used to create a new stone pattern. If the user does not like the current pattern that has been created then restart is used to create a new pattern. If the user changes some of the generator variables, then the restart button must be pressed in order to begin a new pattern using the new variable values.

DISPLACE BUTTON

The displace button is used to create a stone pattern of displaced (non-rectangular) rocks. The displace button distorts rectangular stones into convex (or nearly convex) polygons. If the displace button is not used, then the initial pattern will consist solely of rectangular stones. Note that the displace button can be used only once on a particular stone pattern. If the user does not like the pattern created, he must press the restart button and then press the displace button to get a new pattern.

SHRINK BUTTON

The shrink function is used to introduce spacing between the stones in the pattern. The shrink can be used any number of times on a particular pattern. With each use of the shrink operation, the stones will get smaller and the spacing between the stones will get larger. Note that once the shrink or the subdivide button is pressed, the displace button can no longer be used on the present wall pattern. The user must press the restart button in order to use the displace feature once again.

SUBDIVIDE BUTTON

The subdivide button is used to perform another step of "triangular fractalization" on each of the triangular faces of the stone polytope. The first time this button is pressed after a stone space pattern has been obtained, a stone primitive (or "fractalized triangulation seed") is created for each stone space. As this function is repeatedly invoked, increasingly smaller triangular faces are generated yielding greater detail in the stone surface. The subdivide operation can be performed as often as desired although this should be used sparingly since the number of faces in the stone polytope grows exponentially.

EDGES

The edges button is used to recursively distort the straight edges defining the base of the stone polytope. The usage of this button creates the distorted edges which are common in a stone wall at the interface between a stone and the cement joining the stones.

UG OUT BUTTON

The UG_OUT button is used to create a UNIGRAFIX file of the current wall pattern. This button can be pressed at any time to write the current wall pattern to a unigrafix file.

QUIT BUTTON

The last button is the quit button. This button closes the window and exits the user from the program.

MAKING A STONE WALL PATTERN

This section will discuss a typical session with the stone wall generator. For this exercise the user will generate a displaced (non-rectangular) wall pattern. The size of the pattern will be 50 x 50 and the average stone size will be 5 x 5.

The first step is to start the program. We will initialize the wall size from the command line and then set the stone size with the user interface.

```
% stone_wall 50 50 <ret>
```

After the windows are opened, the user clicks the mouse in the stone width input field. The user changes the value in this field to 5 and presses return. The same operation is performed on the stone length input field. To generate a new stone pattern with the correct stone width and length value, the user presses restart. Pressing displace generates the initial stone wall pattern with the displaced stones. The user then presses the shrink key to generate spacing between the stones. The user desires more spacing between the stones, so he presses the shrink button twice more. Next, the subdivide button is pressed twice in order to generate a stone polytope with the desired level of detail. The edges button is pressed to distort the base edges. Finally, the UG out button is pressed to generate the UNIGRAFIX output file. Quit is pressed to exit the program. UGIRIS can be used to view the output file.

TECHNICAL DESCRIPTION

GENERATING THE RECTILINEAR NODE GRAPH

The stone wall generator begins by tiling a rectangular wall with non-overlapping rectangles. The length and width of each rectangle is obtained by using the average length and width of each rectangle distorted by a noise function so that the set of lenghts and widths has the desired variance. The node graph is generated by creating layers of rectangular stones until the entire wall has been tiled. Each layer is generated by creating stones of random length and width and placing them from left to right until the length of the wall has been exhausted. When a new stone is placed, it is placed at a height corresponding to the maximum height within the corresponding left to right interval on the previous layer. Then for each height in the corresponding interval in the previous layer, a horizontal line is drawn to the left and right as far as is necessary so that only rectangular areas exist below the newly placed stone. If necessary, an edge is dropped from the bottom right of the new stone down to the previous layer. The process of creating layers is discontinued as soon as every left to right interval of a layer hits the top of the wall. The epsilon parameter requires all rectangles in the initial tiling to be wider and longer than epsilon. The node data structure used for this tiling step contains four pointers -- left, right, up and down -- which point to the adjacent nodes as well as the x and y position of the node.

DISPLACING THE NODE DIAGRAM

Each rectangle is then distorted into a convex polygon when the displace function is invoked. Each polygonal area is defined by nodes. Since adjacent stones have nodes in common, a displacement of a node will result in a corresponding displacement of all the incident edges, resulting in a new tiling which is also non-overlapping. If the nodes are displaced following certain heuristics, the resulting tiling will consist solely of convex polygons. Convexity is desirable since the forces of erosion tend to make an object more convex; as a consequence, stones are more likely to be convex. Once the node graph is obtained, a procedure called check_consistency is called which makes sure that the pointers of the node graph are self-consistent. For example, if node 1 points up to node 2, then node 2 should point down to node 1. Once self-consistency is ascertained, each node is displaced by a recursive procedure which traverses the node graph. Only nodes at a T-junction are allowed to be displaced. If the angle of displacement at either side of a T-junction does not exceed 45 degrees on either side, then the displacement will not introduce concavity. Edges with multiple T-junctions are not displaced as a simplifying measure. A more complicated displacement procedure would displace these intelligently, based on the displacement of neighboring nodes.

EXTRACTING FACES FROM THE NODE DIAGRAM

The faces defined by each polygon in the tiling must be extracted into a polytope data structure which initially contains just the polygonal stone space. The faces are cut out by a recursive program which traverses the node graph and cuts out each face. Each node data structure contains fields indicating whether the left_lower, left_upper, right_lower, and right_upper faces have been extracted.

As each node is visited, if one of its face indicators is not set then, that face is cut out. The rules for traversing the node graph are as follows: taking a down link follow the first right link, taking a right link follow the first up link, taking an up link follow the first left link, and finally taking a left link follow the first down link. While doing this, we keep track of edges belonging to the current face. Once we return to the original node, a face is created. As each intermediary node is visited, the corresponding indicator flags are set to reflect the current face being cut out. Subsequently, each polytope data structure is created and initialized.

SHRINKING THE STONE SPACES

Shrinking the stones is performed in order to introduce spacing between the stones of the wall pattern. This spacing represents the cement between stones in a real wall. This operation is performed by moving all the points defining the polygonal space towards the center of mass. If the stone is too small, some of the points cannot be moved. If epsilon is set appropiately, this problem will not occur. The method specified for shrinking, outlined in the paper by Miyata, is suitable for use on polygons with rounded corners, but may introduce concavity into rectangular polygons. The parameterized method that we implemented is more suitable for rectangular polygons.

DISTORTING THE BASE EDGES

The base edges are distorted to simulate the jagged edges that are characteristic of the interface between stones and cement. The distortion is performed by a recursive procedure which randomly displaces the midpoint of each edge. The distance of the distortion is based on the length of the edge.

FRACTAL TRIANGULATION OF STONE FACES

The fractal triangulation technique is frequently utilized to model natural phenomena including landscapes. This technique proves useful in simulating stone surfaces. Initially a stone primitive is constructed in each polytope data structure. This primitive is used as a starting seed for the fractalization of the stone surface. The height of this primitive must be chosen intelligently to avoid the creation of odd spikes. The fractal triangles are generated by displacing the midpoint of each edge in a direction which is the average of the normals of the two adjacent faces. This point must be shared by contiguous faces and the newly created incident edges must be shared in order to prevent the introduction of drastic discontinuity (cracks) into the stone pattern. The offset distance of this point is a function of the length of the edge, the subdivision level, the roughness factor, and a random normal variable.

GENERATING THE UNIGRAFIX FILE

A UNIGRAFIX representation of the stone wall can be optionally generated. This is done by traversing the polytope data structures, creating the necessary vertex and face statements.

DATA STRUCTURES

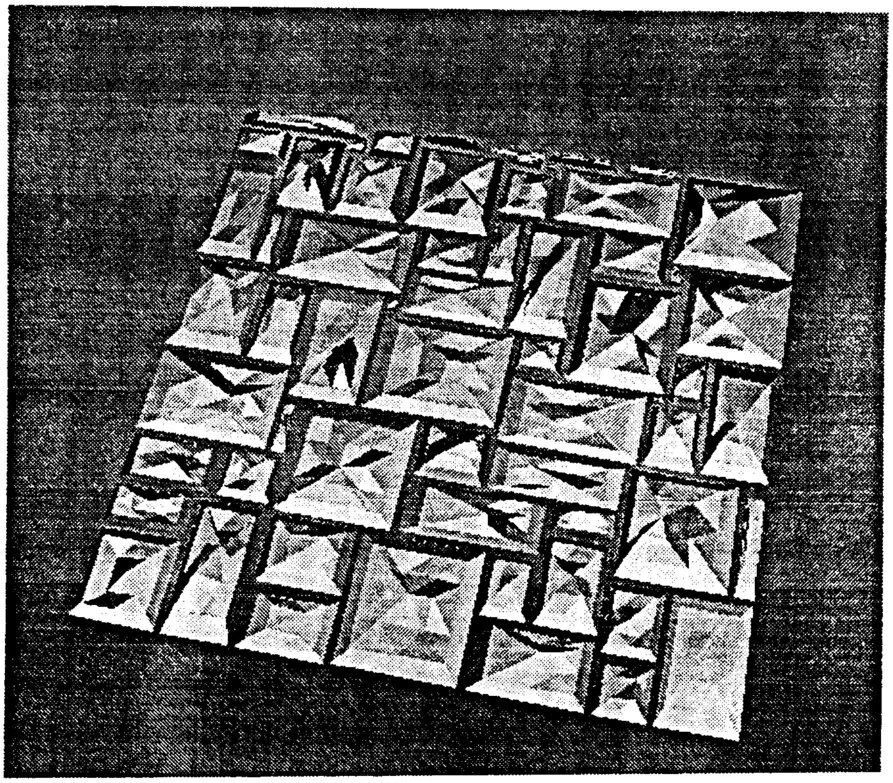
Many data structure were created in order to implement this system. The node data structure is used to create the node graph. The polytope data structure is used to hold each polytope approximation of a stone. In addition, a subset of the ug3 data structures had to be created. These include an edge list, vertex list and face list.

CONCLUSION

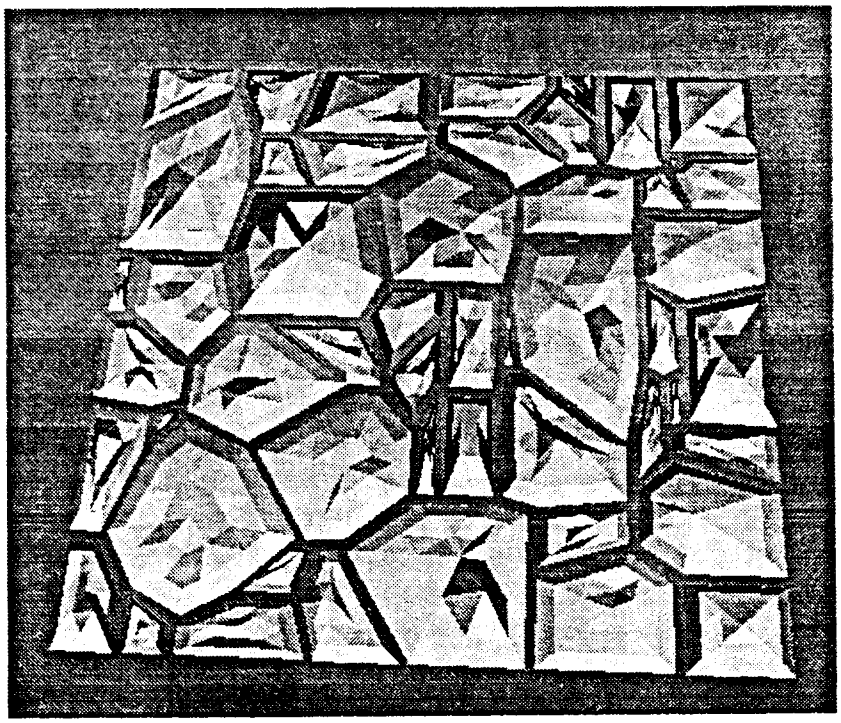
The stone wall generator was inspired by the article "A Method of Generating Stone Wall Patterns", by Kazunori Miyata. At first glance the algorithms seemed very straight forward. However, many details were omitted from the article and occasionally there were errors in the formulas. Furthermore, slight changes in the implementation yielded drastic changes in the output. As a consequence, a great deal of time was spent adjusting these parameters to get realistic results. For instance, the triangulation seed plays a very critical role in the resulting stone polytope.

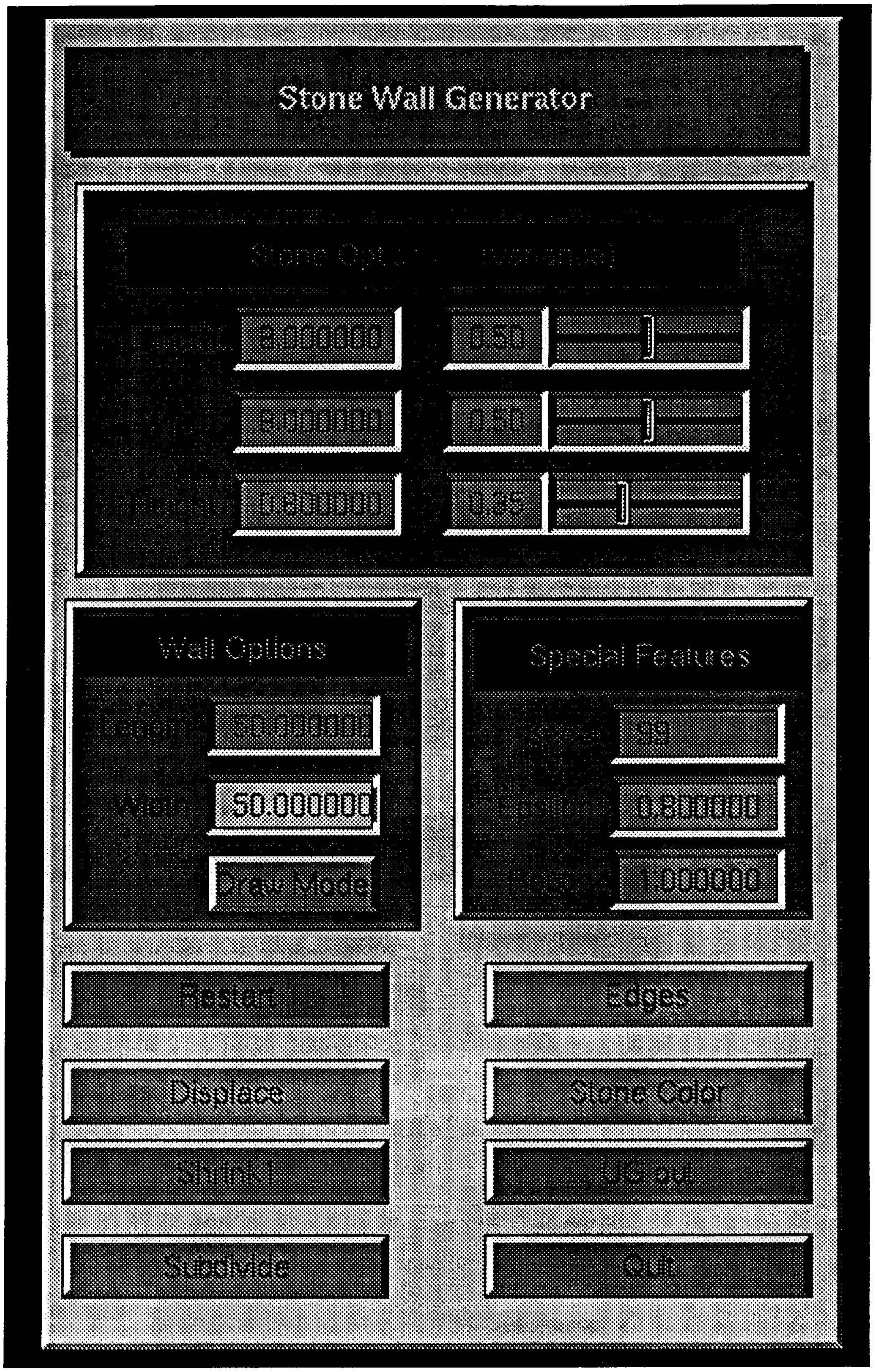
The code to generate the standard normals was obtained from the programmer's bible -- "The Art of Programming." Additional insight into stochastic models was gained from the article "Computer Rendering of Stochastic Models" by Alain Fournier.

Example of a stone wall generated with the stone wall generator. This wall was generated without using the displacement function, so all stones have a rectangular base. The shrink function was used in order to produce the spacing between the stones.



Stone wall generated with the use of the displace option. The displace option produces the non rectangular stones shapes. The shrink options was again used on this figure.




Stone Wall Generator

Stone Opt[illegible]

Length 8.000000 0.50

[illegible] 8.000000 0.50

Height 0.800000 0.39

Wall Options

Length 50.000000

Width 50.000000

Draw Mode

Special Features

[illegible] 99

[illegible] 0.800000

[illegible] 1.000000

(fig. 3: User Interface)

Mini-Project: GROW - An Ivy Generator
An Example of Environment Based Plant Growth

Paul-Henri F. Arnaud
UC Berkeley
Fall 1992
CS 285
Prof. Carlo H. Séquin

Overview:

The goal of this mini-project was to model environment based plant growth. Due to the time constraints of a mini-project, this study focuses on the particular case of modeling ivy growth. The motivation behind this project is that realistic plant growth requires more than just growing a plant in uniform, empty 3-d space. Constraints such as other objects and the availability of space and light affect growth in important ways[1]. This paper will describe one such method to model these effects.

Method:

In this project, the world in which the plant grows in is available to the growth algorithm; a 3-d array, a voxel space, contains information about the world contents. Similar to the way in which pixels divide 2-d space, voxels divide 3-d space. Voxels may be empty, free for plants to grow in or solid, filled by an impenetrable space. At any point in time, for example right before placing a new branch, the program may check to see if the branch intersects with any other objects by enumerating which voxels the branch is contained in, then simply doing an array lookup to see whether those particular voxels are empty or filled.

More Detailed Implementation Notes:

Data Structure:

One key simplification of this program is the data structure. Rather than keep a

[1] Refer to Ned Greene's paper "Modeling with Stochastic Growth Processes in Voxel Space" [ACM Computer Graphics, Vol. 23, Number 3, July 1989] for original research and more detailed information.

true tree structure of the entire plant grown so far (admittedly a more powerful construct, and a likely improvement given more time) the program keeps a doubly linked list of all outermost buds.

Main Loop:

First, initial "seed" buds are inserted into the active bud list. At every iteration, each bud has random chances of dying, branching or sleeping. If it dies, the bud is removed from the active bud list. If it sleeps, nothing happens and the next bud is dealt with. If it branches, a random number of branches to branch from that bud is selected. For each branch, several attempts are made to place it. First relatively long branches are placed. These branches have a length 2-3 times the width of a voxel. This help ensure that the branches making up the vines travel relatively straight and smoothly along the surface of the object. Short branches might have a tendency to zig-zag about in tangles if they had the chance to. However shorter branches are needed to help "navigate" sharp corners. The geometry may not allow long branches to be placed without any intersections around the corner of an object. The other factor that helps with corners is increasing the spread of possible random angles that the new branch exits with. These values for the three sets of tries with different branch lengths is user-specifiable in the inputparams[2] file. If a branch cannot be placed, the bud dies and does not continue growing; there is no place for it to grow. Usually, a branch can be placed. In this case, the branch is written with the proper rotations and translations to the UG output file. Then a few leaves are placed in the proper positions, with slight random perturbations. Finally a new active bud at the end of the branch is placed into the active list, and the bud at the base of the active list is deleted.

These branches and leaves have different levels of detail. These levels of detail allow less complex objects, with fewer vertices and faces, to be used when the object is small and the extra complexity of the shape wouldn't be seen or appreciated. Which instance of the object is used depends on two factors - the user's input and the age/size of the branch or leaf. The user may force all objects to be the simplest possible; in this case the program has no choice. If the user selects a higher level of detail, the program will automatically substitute simpler instances of objects for the youngest generations. These generations are smaller and less visible than older, larger leaves and branches. The end result looks approximately the same as if fully

[2] See GROW man pages.

detailed objects were used for all generations. However the number of faces that need to be rendered is significantly less, thus decreasing rendering time. Additional age-dependent behavior is a color shift for the newest leaves. Just as in real ivy, the youngest leaves are brighter, lighter green than older leaves.

Voxel Space Tools:

Several sets of routines are needed to use voxel space. The first is the ability to define the voxel space. The voxel world space can be defined in two ways; one way is in hard-coded symmetrical objects; these are easily and compactly define using simple loops and have predetermined seed placement. This is useful for demonstration purposes, but not particularly flexible. The second way uses a worldfile format[3] that allows users to specify any object and seed placement; a UniGrafixs program that turned UGobjects into voxels or 3-d bitmap could be useful in this case.

The second key tool is the function that checks for intersections between a branch (essentially a 3-d line) and world voxel space. It is an incremental, scan line based algorithm, an extension of the 2-d algorithm that converts a line into pixels[4]. It is a robust algorithm, calculating and scanning across the dominant plane, but not particularly efficient.

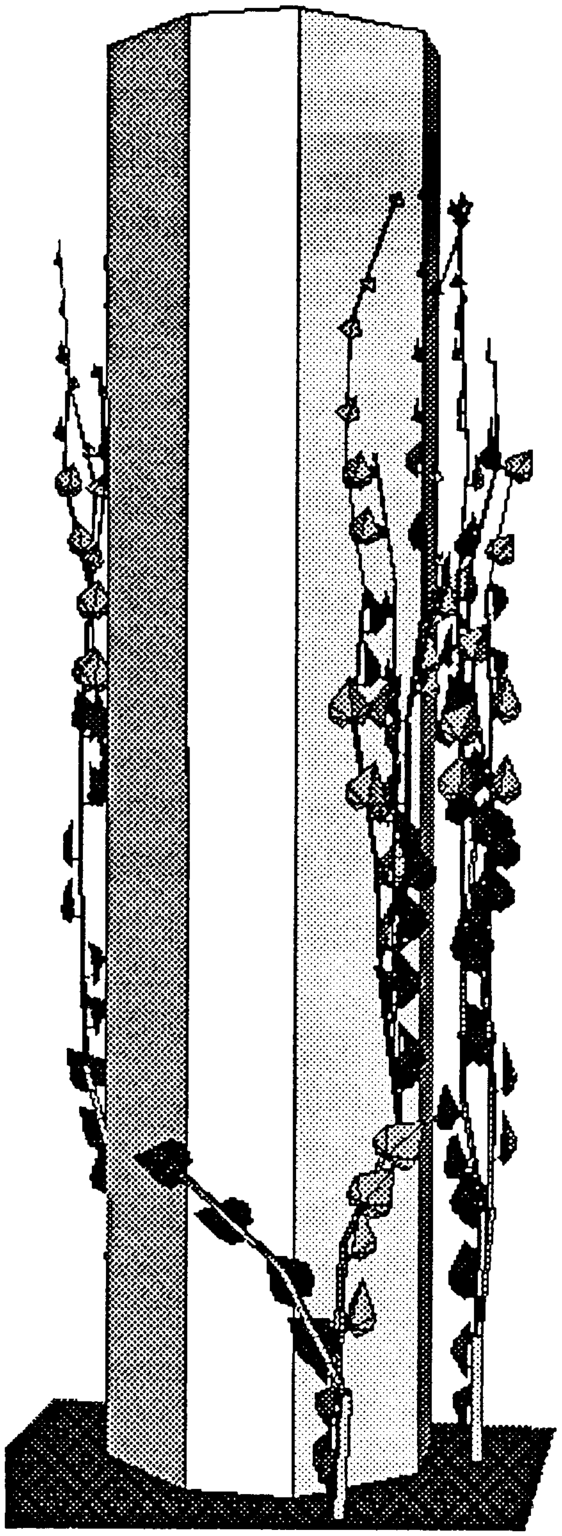
Conclusion:

This concept of environment based plant growth deserves much more treatment than the time that a mini-projects allows. Given another chance I would do this again as a normal project with a partner. These increased resources of time and people would allow us to fully exploit the power of such a realistic and physically based model. Several improvements might include storing a full tree structure of the plant, allowing GL interactive viewing of the growth process and modeling of more complex phenomena. Additionally a less random, more efficient plant growth algorithm would add to the program. Even with these shortcomings, this report shows the usefulness and feasibility of such an environment, voxel-based approach.

[3] See GROW man pages.

[4] Foley, van Dam, Feiner, Hughes, *Computer Graphics: Principles and Practice*, pg. 73

Sample Output



Ivy About Column

NAME

```
     grow - grow ivy
```

SYNOPSIS

```
     grow [-f worldfile] < inputparams > outputfile
     grow [-w] < inputparams > outputfile
```

DESCRIPTION

grow simulates the growth of ivy around objects. It must take as standard input various parameters describing the growth characteristics of the plant. This is most easily done by using the standard inputparams file or by modifying the included file. Output is to stdout, in UniGraphics format. Without any options, growth is simulated around a column. Command line options may be -f or -w but not both.

OPTIONS

-f worldfile	Instead of simulating growth around a default object, the user may choose to define the object and seed locations in a worldfile.
-w	Select default object to be wall instead of column.

FILE FORMATS

inputparams -

```
20
245
.1 .8 .1
.7 .25 .05 0
.4 .4 .5
.4 .8 1.4
2
```

this example file specifies 20 growth cycles, a random seed of 245, 10% chance of a bud dying, 80% that it grows and 10% that is sleeps. It has a 70% chance of having a single branch at each growth cycle, 25% of two branches, 5% of 3 branches and 0% of 4 branches. These branches randomly vary with a spread of .4, .4, and .5 radians about centers of .4, .8 and 1.4 radians from the parent bud. The last 2 specifies the highest level of object detail. 1 is medium and 0 the lowest. 2 looks the best but has the most vertices and polygons.

worldfile -

```
XSIZE 5
YSIZE 3
ZSIZE 3
GROWTH 0
2
1 1 1
3 1 1
2 2 2 2 2
```

```
2 2 2 2 2
2 2 2 2 2
---0
2 2 2 2 2
2 0 0 0 2
2 2 2 2 2
---1
2 2 2 2 2
2 2 2 2 2
2 2 2 2 2
---2
```

This file is a very simple one that builds the following world: it is a 5x3x3 array, GROWTH = 0 tells the computer not to automatically add empty growth space around solid objects (1 would tell it to do so) and it has 2 seeds at location (1, 1, 1) and (3, 1, 1). (Note, seed location ranges from 0 to (?SIZE-1), not 1 to ?SIZE.) The world is defined in x-y cross-sections beginning with z = 0, divided by any string for easy reading. 2's represent solid space, not penetrable by plants, 0's empty space, 3's empty non-growth space, and 1's plant occupied space. This world describes a very small, rectangular box in which the seeds must grow. In most usage 2's and 0's are all that are necessary.

SEE ALSO

UC Berkeley, Fall 1992, CS285 Tech Report on Grow - Ivy Generator by Paul-Henri F. Arnaud

WARNINGS

Beware of running too many iterations or having too many branchings - growth can be exponential.

The Genetic Plant Visualizer

Oliver Crow and Peter Lorenzen
CS285 Fall 1992
Professor Carlo H. Séquin

1.0 Introduction

We have taken ideas from the domains of genetic algorithms and recursive graphical tree modeling to produce a tool for visualizing and automatically generating three dimensional rendered trees. The tool allows users to view the trees and interactively edit them. The trees are represented by 'gene strings', which define the tree characteristics.

Automatic tree generation is performed by selecting and mating parent trees to create new trees. Since the characteristics of the offspring are close to (or between) those of the parents, the user may follow trends in the population by iteratively removing undesirable trees from the current population and generating new ones by using the genetic generator to simulate the growth of the offspring.

The idea, inspired by the work of Richard Dawkins [1], is extended in our project to include populations of genes rather than of a single gene per stage in his *Evolution* program. Moreover, full 3D rendering is used in place of simple line drawings.

2.0 What this tool does

The genetic visualizer allows users to view genetically described plants, to edit the plant genes and to generate new plants based of the current population of genes.

3.0 Gene based plant description

Each of the plants generated by the visualizer is specified by a gene string. All of the information stored in the gene strings is displayed and can be edited by the user in the gene editor window.

The gene representation stores information about the plant as a whole, and also information describing a number of branch types.

The plant header is the part of the plant gene that describes attributes of the entire plant, rather than those which pertain to a particular part of the tree structure. These include the types of the leaves and the size and granularity of the leaves. Leaves are drawn at the tips of the terminal branches (id est, those branches lacking children.) There are four leaf types implemented, including a null leaf.

The number of branch types is variable - more complex tree structures require genes with more branch types. Stored in the gene are a number of attributes for each branch type, plus up to six children for that branch type.

The trunk of the tree is always a branch of the first type in the gene. The children of a branch specify which of the branch types in the gene are to grow out of a given branch. This definition is recursive, so that each of the child types of the trunk specify branches to grow out of the end of the trunk, and the children of these branches are grown from them, and so on. The recursion is limited by the branching depth, which is one of the parameters specified for each branch type. If drawing a given branch would exceed the depth of any of the ancestor branches in the tree, the recursion is halted and that branch is not drawn.

Thus, by having three branch types we may specify a tree with a branching factor of three, as shown in figure 1.

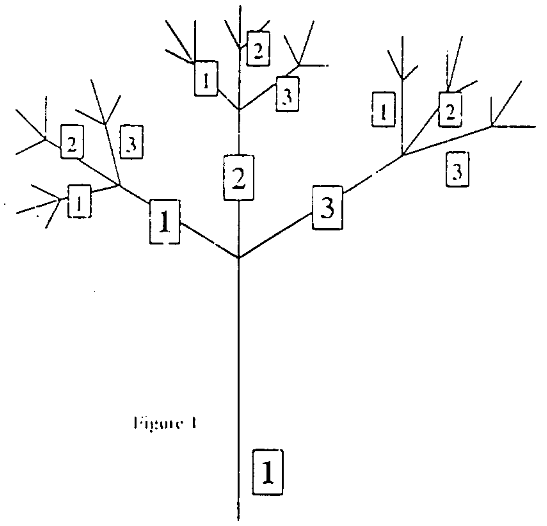


Figure 1

The structure of the gene that generates this plant is displayed in figure 2.

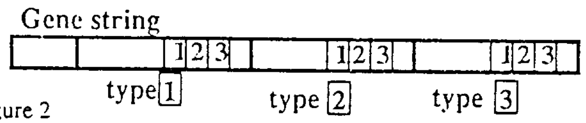


Figure 2

4.0 The Visualizer

In the visualizer interface the user may control a population of genes through their tree representation. There are three parts to the interface: Viewing control from the mouse and keyboard, tree/gene/light/model control, and the gene editor control panel. In addition, there is a help button which invokes a help window.

Presently one can not complete a full 360-degree turn, rather the A key allows the user to turn up to 180 degrees to the left and, similarly, the D allows the users to turn up to 180 degrees to the right. Moreover, we have implemented a crystal ball interface whereby allowing the user to rotate the entire scene with the mouse while holding the left mouse button down.

4.1 The Visualizer Control

The visualizer control interfaces (see Figure 3) is a combination of *Forms* panels and associated call-backs that control various and sundry commands, such as those governing tree placement, gene selection, lighting control and model type. For the latter, we have implemented four model types: THIN, where all branches are represented as line segments, WIRE-FRAME, where all branches and leaves are represented with simple wire-frames, FLAT shading, where all the branches and leaves are rendered with straight forward flat shading, and GOURAUD shading, where the branches are rendered with the GL Gouraud shading model and the leaves with the flat shading model. We elected to employ the Gouraud model for the branches alone because the normals were easy to calculate and more meaningful for the cylindrical branches and not for the rather awkward and non-manifold leaves and flowers. We decided to include the three inferior, (id est, non Gouraud) models to allow for greater real time viewing. Initially the user may begin with the Gouraud model activated and as plants are added may wish to switch to a lower model in the hierarchy to attain a speedup in the rendering of the plants.

4.2 The Gene Editor

The gene editor panel (see Figure 4) allows the user to edit the genes in the gene pool by altering their attributes with various slider, counters and dials. At any one time the editor panel displays the current gene and current branch type within that gene. These may both by altered simply by choosing the required selection from their respective browser panels. The editor simply changes the gene description in memory which may include the extension of the gene to allow for additional user-defined branch types. The associated tree is automatically redrawn as the edit panel settings are updated.

5.0 Automatic plant generation

The primary aim of using the gene-based tree description was to allow the automatic generation of new trees based on an existing population of trees. When the user selects the Generate command, the program randomly selects memembers of the current gene population, and generates new genes by a random chromosome inheritance mechanism.

Our algorithm for this was inspired by classical genetics and work on genetic algorithms. The use of strict cross-over within the gene string, an approach, which is sometimes used for genetic algorithms, is inappropriate in this case. The reason for this is that randomly cutting the gene string destroys some of the information structurally represented in it, specifically the branch type structure is not kept. The result of this would be that trees generated by cross-over of two parent genes would not necessarily resemble either of the parents.

We instead produce new trees by treating each characteristic in the tree gene as a chromosome. Generation consists of randomly selecting which of the parent chromosomes are to be put into the new gene. The new gene structure is created, and the associated tree is rendered. In a rough simulation of tree growth, we place the new trees near (randomly offset from) one of their parents.

6.0 Conclusion

6.1 Extensions

Whilst we had hoped to extend the idea of plant regeneration to natural selection of plants based on specified criteria, rather than having the user select which plants are best at each stage, we recognized that this would probably lie outside the bounds of this project. The reason that this is difficult to implement is that in order to select only those genes that generate trees that look tree like, one would have to model a number of complex environmental factors, such as sunshine, water uptake, tree mass and support, and so forth.

One of the problems with the strictly chromosome based approach is that all the characteristics of generated plants are directly inherited, and so without user intervention characteristics not represented in the original population will not be generated. One extension to alleviate this would be to include automatic and random gene mutation. Currently this is effected by the user.

The Genetic Plant Visualizer can be further expanded in the following ways: new leaf types may be added, resolve branch intersections by not allowing polygonal intersections (id est, non-manifold trees) at great cost to speed, cache the leaf objects into a table of GL objects as is done with the trees. The tradeoff, in the latter action, lies in the increased speed gained from the actual caching verses the overhead induced by the changing of the context with respect to editing the attributes of a gene.

6.2 What did we learn?

As is the case in almost any thing related to computer science there is the size/speed tradeoff. Plants growth is exponential with respect to level of generation, O(x to the n) where x is the number of children branches and n is the level of recursion. Moreover, a high constant factor is usually induced by both increased branch facet definition and number of trees as one generates a forest. The size, number, and definition of the trees are the main factors considered in this tradeoff as it is the rendering time, which we were attempting to optimize. We also learned from this ambitious endeavor to model *real* genetics that there are many complicated issues that need to be addressed. For example, in the real world biological life is cellularly based (something which was trivially rejected as being to difficult to handle in the four week period of this assignment.) Moreover, gene cross over is a tricky business and that gene inheritance is a much simpler task in which to achieve reasonable results (an application of the KISS or Keep It Simple Stupid approach to designing or implementing a project.) Nevertheless, our gene representation is vastly superior to that of real genes with respect to efficient use of space. In nature, large portions of DNA appear to be unsed, whereas with our gene description all the gene space is used. In some degenerate cases there will be branches with zero length, or overlapping branches and our genes display a similar, but to much less of an extent, redundancy to that observed in natural gene DNA.

6.3 Acknowledgements

The authors would like to thank Gavin Bell and Andrew Choi for the crystal/track ball interface code, and for the sundry bits of advice from the students and Professor Carlo H. Séquin of CS285.

Plant Figures

Plant Figure 1.
The first is a sample tree with the structure given in figure 1.

Plant Figure 2.
The following figure is a tree with a slightly more complex branching structure. Note that some of its branches split four ways. This tree was created with the gene editor.

Plant Figure 3.
The final figure is an example forest with some automatically generated trees.

Bibliography

1. Richard Dawkins *The Blind Watchmaker: Why the evidence of evolution reveals a universe without design.* W. W. Norton & Company, New York, 1986.

2. James Hanan, Aristid Lindenmayer, and Przemyslaw Prusinkiewicz *Developmental Models of Herbaceous Plants for Computer Imagery Purposes.* Proceedings of SIGGRAPH August 1988. In Computer Graphics 22, 4 (1988), 141-150.

3. Claude Edelin, Jean Francon, Marc Jaeger, Claude Puech, and Philippe de Reffye *Plant Models Faithful to Botanical Structure.* In Computer Graphics 22, 4 (1988), 151-158.

4. Larry Gonick & Mark Wheelis. *The cartoon guide to genetics.* Updated edition, Harper Perennial, New York, 1991.

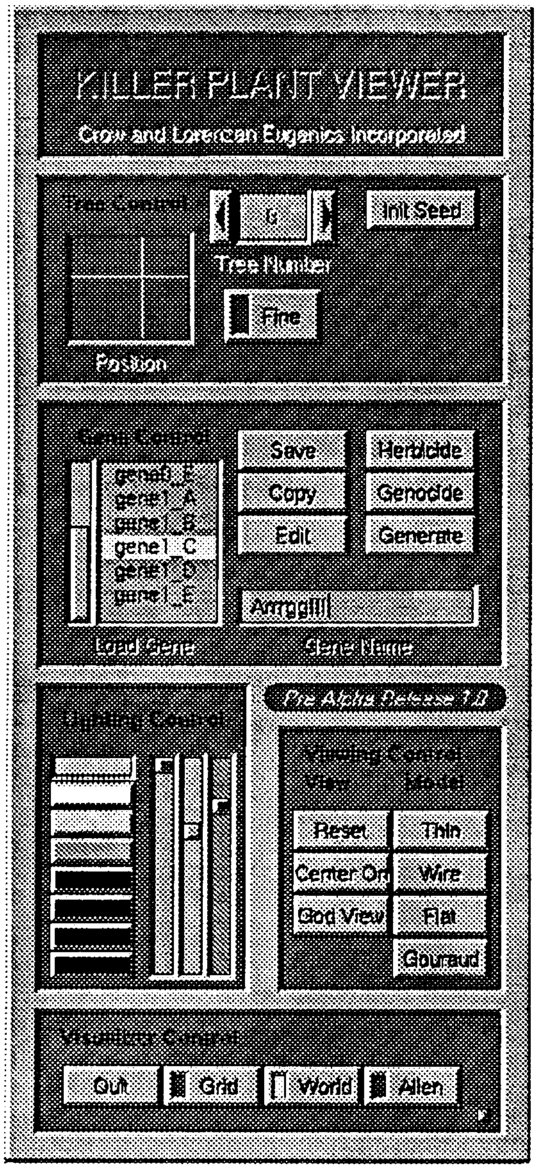
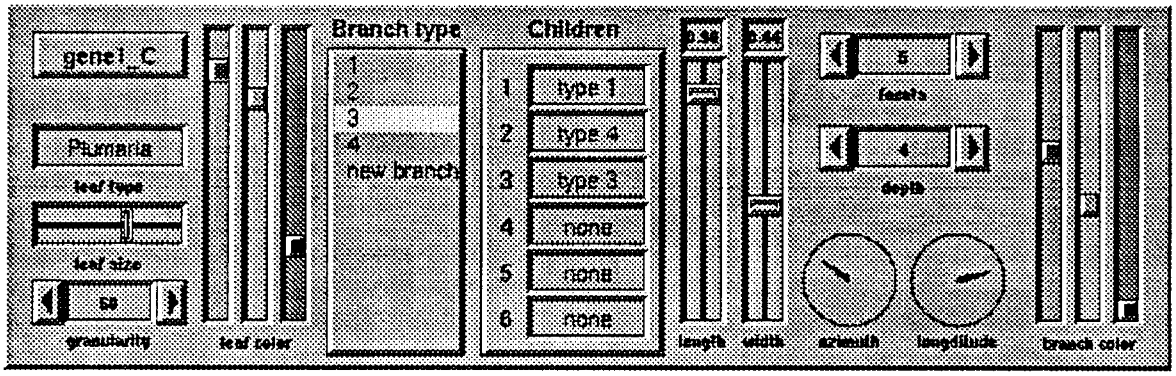
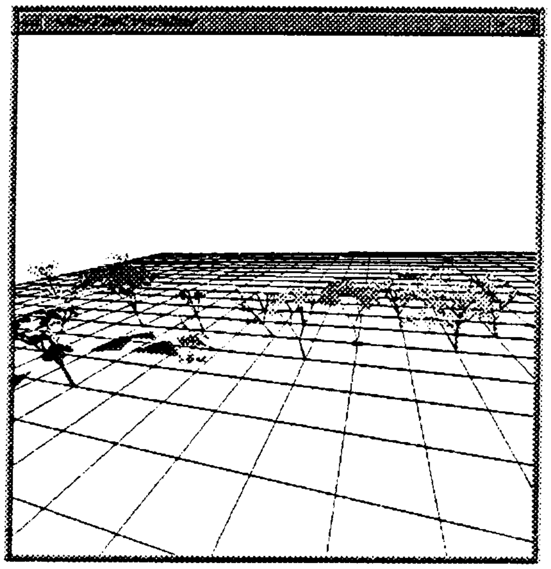
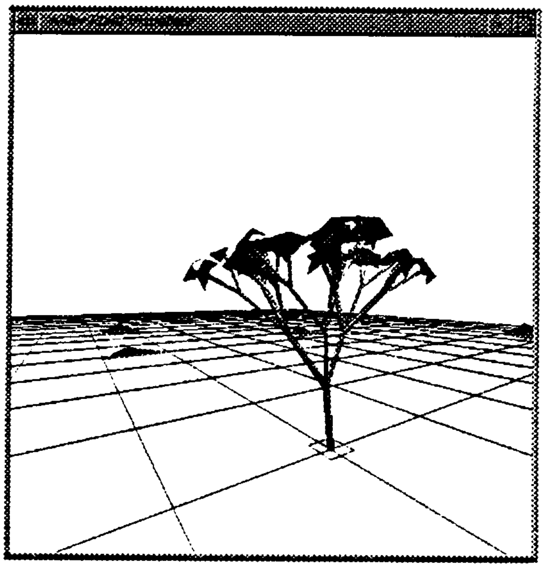


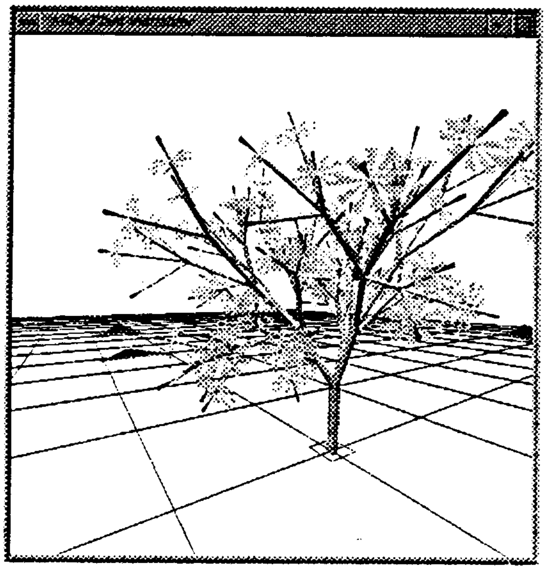
Figure 3

Figure 4




Plant Figure 3.


Plant Figure 1.


Plant Figure 2.

NAME
plant - Genetic Plant Visualizer

SYNOPSIS
plant [no command-line options implemented]

WARNING
This manual page is an extract of the documentation of the Genetic Plant Visualizer. Please consult the technical documentation associated with professor Carlo H. Séquin's Fall 1992 Computer Science 285 class for further details.

DESCRIPTION
The Genetic Plant Visualizer allows the user to interactively view a set of trees/plants defined by genes. Initial gene descriptions are stored in *.gene files in the current working directory. The user may also interactively modify individual genes as well as the gene pool itself. The visualizer runs as fully interactive tool using GL to graphically view trees defined by genes.

SIDE EFFECTS
This program generates a temporary file with the name "_genes_" in the current working directory. If you have a file with this name in your current working directory we recommend that you move it elsewhere as the ensuing results may be unpredictable.

INTERFACE
Tree Control:

- Position
Moving the cursor in the forms positioner box moves a red square wire-frame box to a *current* position where the user may plant a tree.

- Tree Number
Moves the cursor to the *current* tree.

- Init Seed
Loads in the initial population of genes defined by all the files in the current directory with the .gene suffix.

- Fine
This allows the a fine adjustment to the movement of the cursor with respect to the grid. When the Fine button is depressed the tree cursor is moved 0.01 times the distance of the grid spacing per step, where as when the Fine button is de-selected, the tree cursor moves the distance of the grid spacing per step.

Gene Control:

- Load Gene
Loads a gene from the gene pool into the *current* gene. Not to be confused with the Init Seed button which loads in information from files on disk in the current working directory.

- Save
Saves the *current* gene to a text-based gene description file. Writes a file with the name in Gene Name. See Gene Name.

- Gene Name
Sets the name of the current gene. The text in this box

is only considered when the *return* button is pressed. Gene Name is used in conjunction with Save. See Save.

- Edit
Invokes the gene editor panel. Selecting this button again hides the editor panel.

- Copy
Copies the *current* gene and grows a tree, with this gene, in the location specified by the tree cursor.

- Herbicide
Deletes the current gene and corresponding tree.

- Genocide
Deletes the lowest order generation of genes in their entirety.

- Generate
Creates a new set of genes based on the current set of genes.

Light Control:
There are eight light sources whose RGB values are controlled by three sliders. These lights are arranged in a cubic formation around the scene.

View Control:
- Display Mode
  - Thin
  Simple bone-headed model where all branches are represented by line segments.

  - Wire
  Branches and leaves rendered with wire frames.

  - Flat
  Branches and leaves rendered with flat shading.

  - Gouraud
  Branches rendered with Gouraud shading and leaves with flat shading.

- Center
Places the user directly in front of the desired tree. This saves the user from manually *flying* to that tree.

- God View
Places the user in a position to view the tree world from a point well above the trees.

Visualizer Control:
- Quit
Exits the program.

- Grid
Allows the user to view the plants with a grid. This is useful for placing objects.

- Help
Invokes a help window.

- World
Allows the user to view the plants in a scene that includes green grass, blue sky, and a hazy yellow sun.

- Alien

Toggles the appearance of a space alien. The alien follows the tree cursor. This is useful in comparing the relative sizes of trees that are not in the immediate vicinity of each other.

Gene Editor:

- Leaf Type
This menu allows the user to select between the different hard-coded leaf/flower types.

- Leaf Color
These sliders control the RGB values of the leaves in the manner described in the lighting interface.

- Leaf Granularity/Definition
This counter controls the *definition* of a leaf. This is only applicable for certain leaf types. For the plumaria it controls the number of petals, for the cardioid leaves it controls the smoothness of the cardioid curves, et cetera.

The following controls specify and display the branch type attributes. These all, with the exception of color, are specified relative to the parent branch.

- Branch Type
This menu controls the current branch type being edited. Also, the user may select the *new branch* option to create a new branch. This new branch will be initialized to nothing so the user will have to define its characteristics.

- Branch Children
These six slots hold the *branching* information. Where a slot is not empty a child branch will bud from the current branch.

- Branch Color
Performs the operation as Leaf Color save for branches this time.

- Branch Length
This slider controls the length of the current branch being edited.

- Branch Width
This slider controls the width of the current branch being edited.

- Branch Azimuthal Angle
This dial controls the relative azimuthal angle between the current branch (as seen in the view up position) and its children.

- Branch Longitudinal Angle
Like the Branch Azimuthal Angle dial, this dial allows the user to modify the relative longitudinal angle between the children branches.

FILES

Makefile	- Controls compilation
cb.c	- Crystal/Track ball interface
control.c	- Initiates program control
draw.c	- Rendering module (GL primitives)
event.c	- Event handler module
gene.c	- Gene module

grow.c	- Tree growth module
interface.c	- Forms interface call-backs
light.c	- Lighting interface
main.c	- Program initialization and entry
matrix.c	- Basic mathematics module
normal.c	- Addendum to the mathematics module
panel.fd	- Forms description of the user interface generates panel.h and panel.c
perpara.c	- Perspective control
random.c	- Simpler random number generator
trackball.c	- Crystal/Track ball control
tree.c	- Tree control module
vect.c	- Addendum to the mathematics module
window.c	- Window control module

LIBRARIES

-lc_s	- Shared C
-lforms	- Forms
-lfm_s	- Forms
-lgl_s	- Shared IRIS Graphics Library
-lm	- Mathematics

SEE ALSO

Nothing.

BUGS

Bugs have not been modeled for our trees as of yet.

COPYING

Copyright (c) 1992 Free Bogus Software, Inc. Permission is granted to make, modify, and distribute verbatim copies of this manual, source code, support files, and executable at will.

AUTHORS

Oliver Crow and Peter Lorenzen (lorenzen@cory.berkeley.edu)

Plant Maker Report

Maryann Simmons and Steven Yen

Computer Science Division
University of California
Berkeley, California 94720

simmons@miro.cs.berkeley.edu

syen@postgres.cs.berkeley.edu

Introduction

The PlantMaker is an interactive Tcl/Tk + GL + UG based tool for creating and displaying three dimensional models of plants. These models can be saved in PlantMaker model file format and/or in UNIGRAFX file format. PlantMaker uses grammars to describe plants, utilizes caching and compacting to optimize sentence handling, and includes a simple constraint based growth system.

Using Grammars to Describe Plants:

By incorporating the use of plant grammars with simple aging equations, the PlantMaker provides a method for describing plant models and their growth which is very simple, but very powerful in its generality. It is possible to describe the developmental growth of many plants through a grammar (a set of production rules) and aging equations. This idea utilizes the formalism of L-system grammars as presented by Prusinkiewicz and Lindenmayer in their book, *The Algorithmic Beauty of Plants*. L-systems are formal grammars in which the production rules are applied in parallel. The PlantMaker incorporates these grammars in its plant description language, which is described below. Tcl/Tk is used to parse this language.

Productions:

The following are two example productions rules, as specified in the plant description language.

```
production i S(ejL)(ekL)S
production p ir(p)de(dL)i(cL)l(p)cp
```

If the initial (generation 1) sentence is "p", the next two generation sentences produced would be...

(generation 2)

```
ir(p)de(dL)i(cL)l(p)cp
```

(generation 3)

```
S(ejL)(ekL)Sr(ir(p)de(dL)i(cL)l(p)cp)de(dL)S(ejL)(ekL)S(cL)l(ir(p)de(dL)i(c
L)l(p)cp)cir(p)de(dL)i(cL)l(p)cp
```

Interpretations:

The above generated sentences of characters are interpreted as a sequence of turtle walking actions and plant parts.

Table 1: Interpretations

Character	Interpretation
c	rotate Y 60.00
d	rotate Y -60.00
e	rotate Z 90.00
j	rotate X -30.00
k	rotate X 30.00
l	rotate Y -30.00
r	rotate Y 30.00
L	part 0
S	part 1
(	push
)	pop

Actions:

The interpretation command can associate any turtle walking action with any character. For example, "`interpretation c rotate Z -60`" associates with the character c the action of rotating around the Z axis by -60 degrees (i.e., turtle turn left by 60 degrees). Any of the standard transformations (rotate, scale, translate) as well as push (save) turtle state and pop (recall) turtle state can be specified with the interpretation command.

Parts:

The interpretation command can also associate any UG object with any character. These are the lowest level renderable parts of the plant model. For example, "`interpretation L part 0`" associates part 0 with the character L. Part 0 is further specified in the following manner "`part 0 leaf.ug {{-rx 90.0} {-sa 8.0*$a}}`". This indicates that part 0 should be the UG model specified in the file "`leaf.ug`" with the indicated age equation.

Aging Equations:

Each terminal character in the generated sentences has an associated 'age'. This age is initially zero and gets incremented with each generation. The user can specify equations as a function of this age as illustrated above. If the terminal character L has age 2, for example, then the interpretation of L would be the "`leaf.ug`" model prescaled by 8.0*2 and prerotated by 90.0 degrees. This age factor can be used to determine attributes of parts, for example size and shape, as the plant grows over time.

Caching and Compaction:

Because of the nature of the grammar rewrite rules, sentences can grow exponentially larger with higher generations. To prevent these large sentences from crippling the interactive speed of PlantMaker, caching and compaction schemes are applied to these sentences.

After productions are loaded, each sentence from generation 0 to N is precomputed and cached. This allows users to move easily between generations, as opposed to computing each generation from scratch when needed. However, the large size of the sentences necessitates a sentence compaction scheme.

PlantMaker includes a simple compaction scheme. All characters without interpretations and degenerate cases (e.,g., "()") are stripped from the sentence, which can sometimes reduce sentence size and rendering time by half.

Barriers:

With PlantMaker the user can impose constraints on the growth of plants by specifying barriers. These barriers are triangles and quadrilaterals which the plant limbs should not pass through. If a plant limb encounters a barrier, the PlantMaker turns the limb away from the barrier and allows growth to continue. Basically, as each part encountered while processing a generated sentence, a line segment that passes through the part is compared in turtle-walking space to every barrier for intersection. If an intersection point is detected, then the part is first rotated 90 degrees left or right away from the intersection point, depending on the orientation of the barrier to the line segment. Then growth can continue from that new turtle walking position. This very simple barrier constraint implementation gives the user more control in the plant modeling process.

Conclusion:

The PlantMaker is based on the use of grammars to describe plants and their growth. As an implementation of such a production rule based system, it is very general. In fact, PlantMaker can be used to create any type of model that could be described with L-systems, given the appropriate input. However, the PlantMaker could be improved as a tool for generating plant models by making it less general. If the PlantMaker could incorporate randomness into the sentence generating process, it could produce more realistic, less symmetric plants. Also, the tool could know more specifically about plant elements such as flowers, leaves, stems, pots, etc., instead of treating all parts generically.

NAME

PlantMaker – GUI editor of grammar based plant models

SYNOPSIS

PlantMaker [*modelfile*]

DESCRIPTION

PlantMaker uses simple grammars to concisely describe plant models, which users can interactively view and edit in 3D. The final output of a PlantMaker session should be a UniGrafix format file which describes the plant model. PlantMaker runs on Silicon Graphics machines only.

COMMAND LINE ARGUMENTS

The model specified by the optional *modelfile* argument is automatically opened and loaded by PlantMaker.

WINDOWS

Two main windows are created by PlantMaker. One window contains pull-down menus of various commands provided by PlantMaker. These commands are described later.

The second main window contains a three dimensional view of the current model. Across the top of this 3D view window is a label which shows the path and current generation (in parentheses) of the current model. If there were unsaved changes or unsaved modifications to the current model, the tag "(modified)" is also displayed along the top in this label.

MOUSE COMMANDS

You can use the mouse to rotate, scale, and translate the three dimensional view of current model. The interface is similar to the virtual-crystal-sphere metaphor used by the UniGrafix **Animator** application.

To rotate the current model, drag with the middle mouse button pressed down. To scale the model, control-drag with the middle mouse button pressed down (i.e., hold down the "Control" key). To translate the model, drag with the right mouse button pressed down.

MENU COMMANDS

PlantMaker has the pull-down menus of *File*, *Edit*, *View*, *Barriers*, and *Help*. These menus are activated with the left mouse button. You can "tear-off" any menu using the middle mouse button.

The *File* menu has the following commands:

New

This command replaces the current model with a new, blank model, named "unnamed.model". PlantMaker asks for confirmation first if the current model has unsaved modifications.

Open... This command gives you a modal file browser where you may choose another model to be the current model. This file browser also can give you access to a library of prebuilt models which you might use as templates.

Save

This command saves the current model.

Save As...

This command gives you a dialog box where you may enter a new path for the current model before saving.

Save UG...

This command gives you a dialog box where you may save the current model as a UniGrafix format file.

Quit

This command exits PlantMaker, but ask for confirmation if the current model has unsaved modifications.

The *Edit* menu has the following commands:

Grammar...

This command brings up an editor for modifying the grammar of the current model. PlantMaker uses this grammar to compute the generation N+1 sentence from the generation N sentence. The initial sentence (generation 0 sentence) can be modified in this editor.

Interpretations...

Characters in the generated sentences are interpreted as turtle-walking instructions. This command brings up an editor allowing users to edit the turtle-walking instructions associated with the characters (i.e., how characters are interpreted). Characters are interpreted as either rotatations (e.g., turn left 30 degrees, turn right 30 degrees), translations (e.g., move forward 5, move sidways 10), or as parts (e.g., part 0, part 4). The characters "(" and ")" are always interpreted as save (push) turtle walking state and recall (pop) turtle walking state. The turtle walking coordinate system has -Z as forward, +X as right, and +Y as up.

Parts... This command brings up an editor for modifying the current Parts Library. You can swap different Parts Libraries to be the current Parts Library through this editor. Every part (e.g. part 0, part 1) has an associated UniGrafix format file. For example, part 0 might be "leaf.ug", part 1 might be "stem.ug", and part 3 might be "strawberry.ug". Each part may be pretransformed by a set of transformations, specified in a list of format "{xform expr} {xform expr} ...". For example, the transformations "{-rx 50} {-rz 44.0+1.2} {-sa 1.0-0.2}" will scale the part by 0.8, rotate the part around the Z axis by 45.2 degrees, and rotate the part around the X axis by 50 degrees. The xform may be one of -rx, -ry, -rz, -sx, -sy, -sz, -sa, -walk (respectively, for rotating around an axis, for scaling along an axis, for scaling along all axis, and for turtle walking a distance). The expr may be any simple expression (including nested parentheses) but with no internal whitespace. The expr may also access the age of the character which is referencing the part. Use variable "$a" for this. For example, if part 0 has the transformations of "{-sz $a*8.0} {-ry ($a/10.0)*90.0}" then part 0 will be scaled and rotated by a factor of the age of the characters which use part 0.

Generation

This is a cascading menu that allows you to display either an older generation sentence (*Up*) or a younger generation sentence (*Down*). You can also change the current generation displayed with the up and down arrow keys.

The *View* menu has the following commands:

Reset

This command resets the view of the current model.

Spin

This command spins the view of the current model around the Z axis.

Cache

This checkbox item tells PlantMaker whether to compute and cache information about the current model (which might take some time) in order for faster rendering of the model.

The *Barriers* menu has various commands relating to the barriers feature of PlantMaker. Barriers are triangles or quadrilaterals which the model can not "grow" through. Model parts are transformed away from barrier faces if a barrier intersection occurs.

Load... This command gives you a dialog box where you may enter the path of a UniGrafix format file, which is interpreted by PlantMaker as a set of barrier faces.

Show Barriers

This checkbox item flags whether to display the current barriers.

Show Intersections

This checkbox item flags whether to display the intersection points between the current barriers and the current model.

Apply

This command applies the current barriers against the current model.

The *Help* menu lists various topics about which on-line help is available.

SEE ALSO

See the *Plant Maker Report* for more information about PlantMaker.

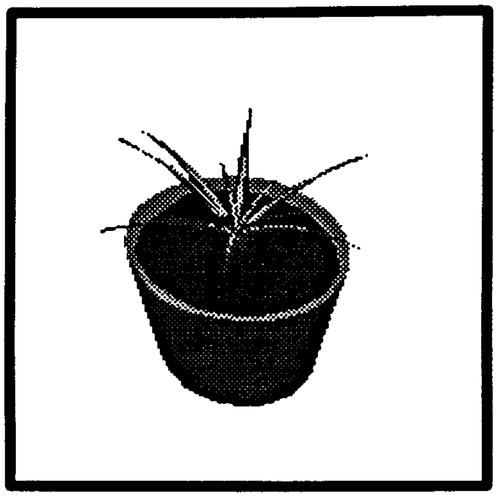
See the A. R. Smith paper "Plants, Fractals and Formal Languages," *SIGGRAPH 84*, for more information about using grammars to model plants.

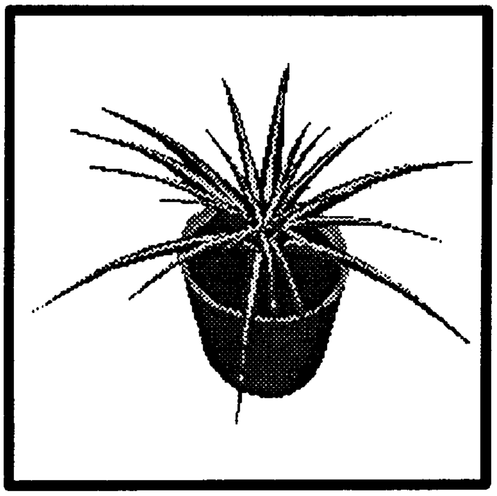
UniGrafix, ug, Silicon Graphics, Tcl/Tk

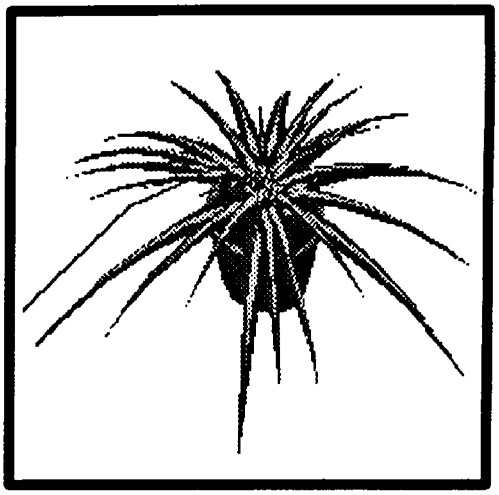
AUTHORS

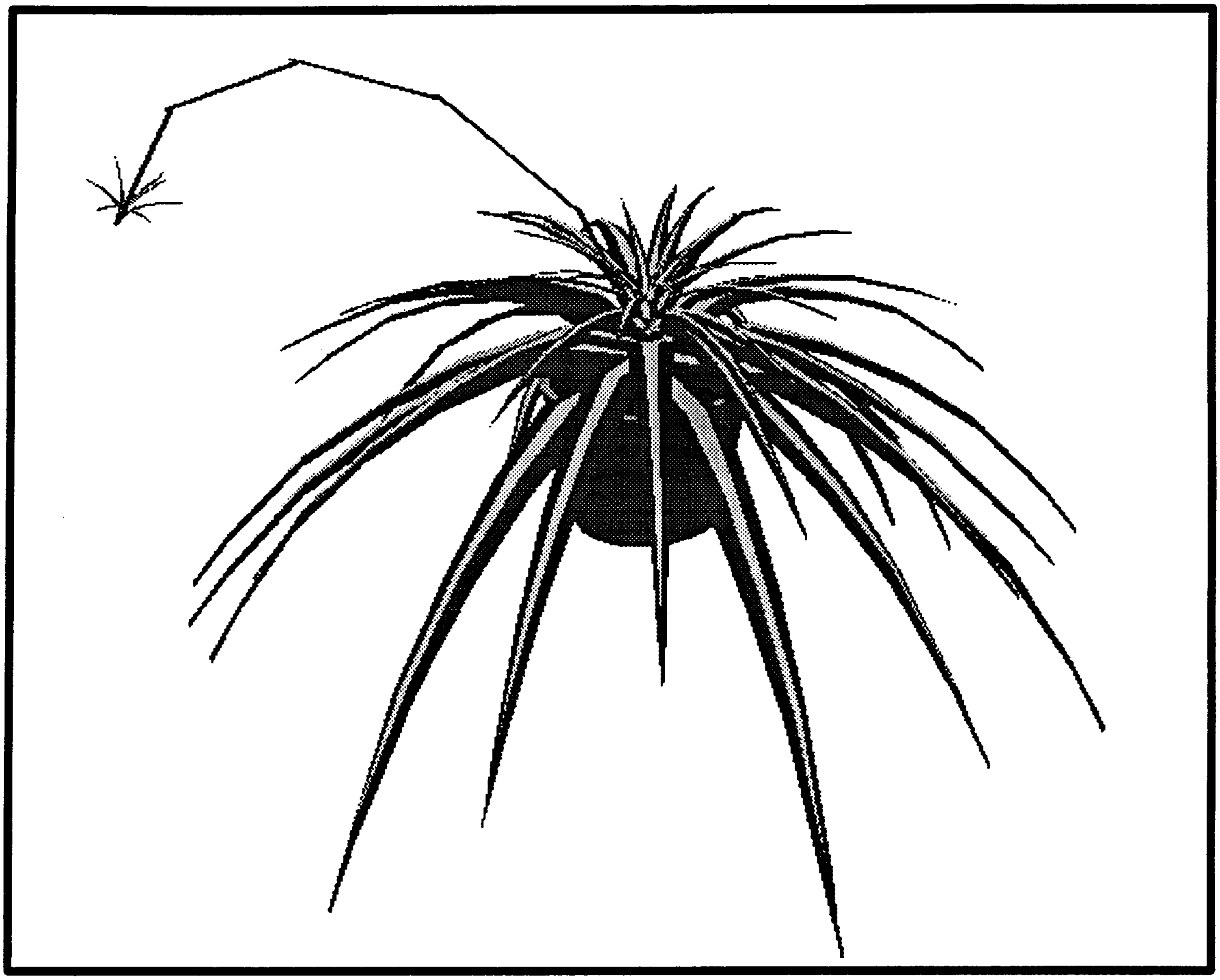
Steve Yen (syen@postgres.berkeley.edu)
Maryann Simmons (simmons@miro.berkeley.edu)

PLANTMAKER –Spider Plant

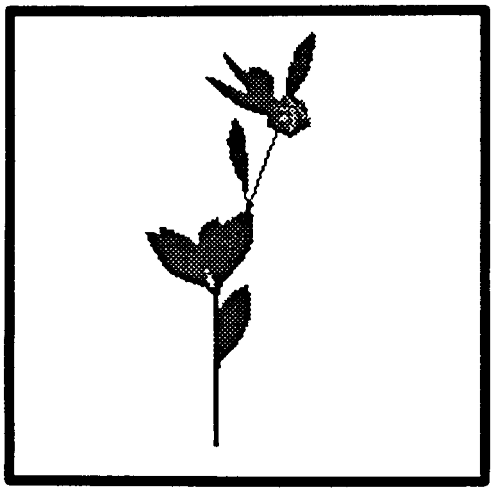

Generation 2

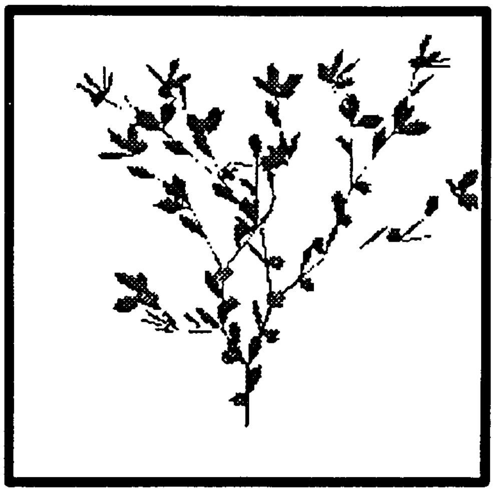

Generation 4

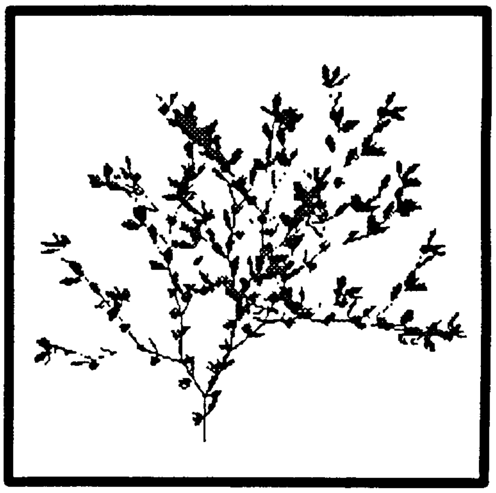

Generation 6

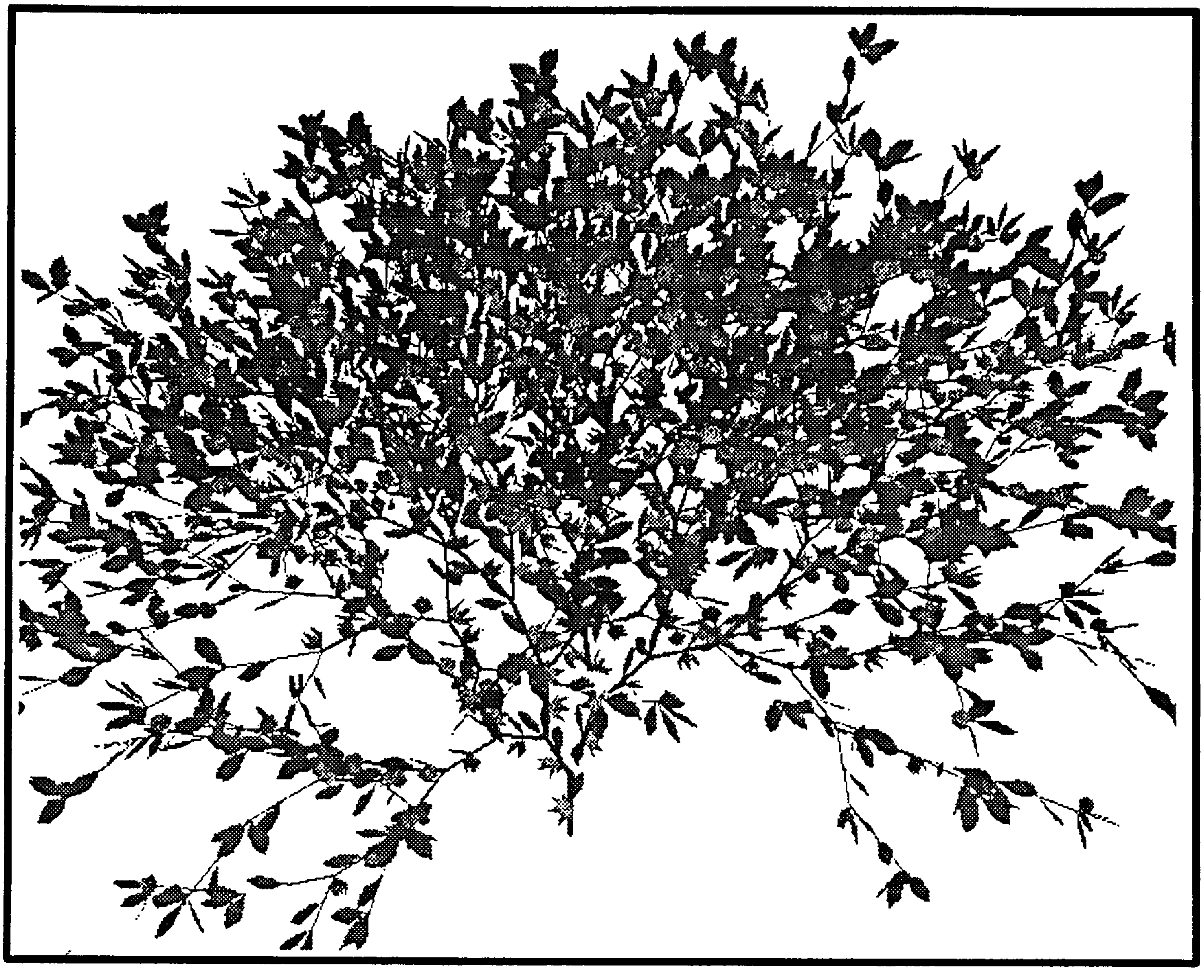

Generation 8

PLANTMAKER –Flowering Plant

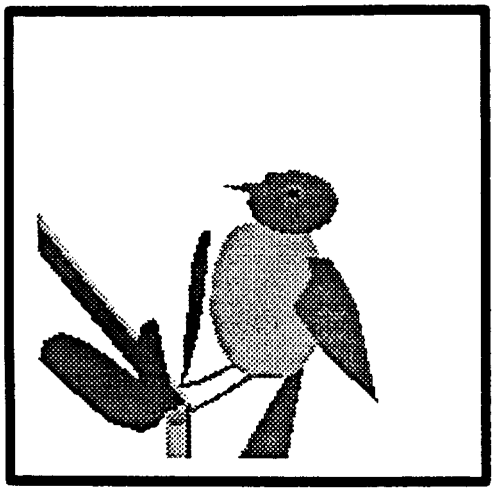

Generation 2


Generation 4


Generation 5

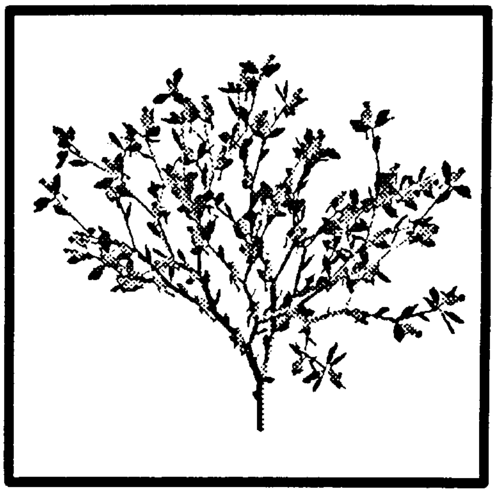

Generation 7

PLANTMAKER -Canaries

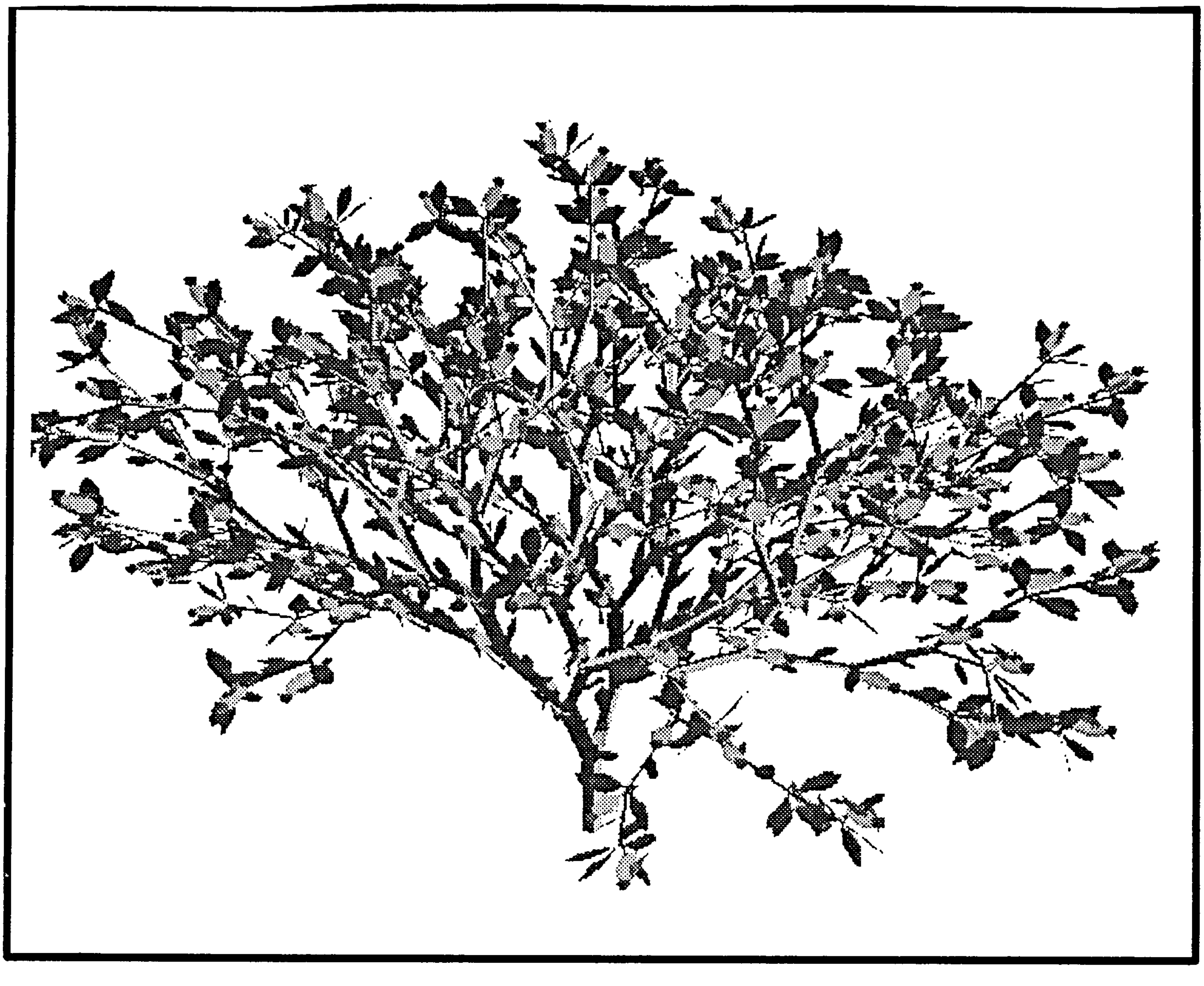


Generation 2

Generation 4



Generation 5



Generation 6

Gen: An Interactive Tool For Rendering of L-Systems

Christopher Fuselier and Amin Vahdat

December 13, 1992

1 Introduction

It has been demonstrated that simple rewriting rules called L-systems can be used to represent a wide variety of three dimensional objects. Recently, much work has been done in using L-systems to represent different types of plants and trees. Inspired by the generality and power of L-systems, we set about designing an interactive tool for specifying these productions and rendering the resulting graphical interpretations on a graphics workstation. While general, our project focuses on the use of L-systems for the rendering of plant-like structures.

The project can be roughly split up into two parts: the interpreter and the user interface. In the next section, we present a brief overview of L-systems. In section three, we outline some internals of the interpreter and describe our data format. Section four describes the user interface. Section five concludes the paper and presents some directions for future work.

2 An Overview of L-Systems

A large part of the elegance of L-systems comes from their very simplicity. In this section we present a brief overview of L-systems. Readers who are further interested in this topic are encouraged to refer to [PL90].

To specify a three dimensional object, one simply provides one or more rewriting rules and an initial value. These production rules may be parametric as well as probabilistic. For example, the input:

```
Seed: A(1)

Rule 1: A(w) : (w = 1) -> A(w+1) rotateZ(45) draw(2)
Rule 2: A(w) : (w > 1) -> draw(1) rotateZ(45) A(w+1)
```

will produce the following resulting string after three generations:

```
Generation 0: A(1)
Generation 1: A(2) rotateZ(45) draw(2)
Generation 2: draw(1) rotateZ(45) A(2) rotateZ(45) draw(2)
Generation 3: draw(1) rotateZ(45) draw(1) rotateZ(45) A(3) rotateZ(45) draw(2)
```

Elements of the string could then be assigned graphical meanings used to render the resulting string in turtle walk fashion. Some statements would then move the turtle, while others would vary its orientation. In the simple example above, *draw(n)* might mean draw a cylinder of length n and move n units in the turtle's current orientation, and *rotateZ(deg)* means rotate the orientation of the turtle by *deg* degrees around the local coordinate system's z-axis.

3 The Interpreter

The interpreter consists of the following components:

- A module to read specifications for L-systems and parse them into internal data structures.
- A module to expand the initial string based on rewriting rules for a given number of generations.
- A module to interpret resulting systems graphically and render them in a GL window.
- A module to read a unigrafix file and translate it into the specification for a GL object which can then be used as a terminal in the rewriting rules.
- A module to handle memory management and caching.

In this section we describe the supported syntax for our L-systems and detail some of the memory management techniques used to improve the performance of our system.

3.1 Production Syntax

The production rules for our grammar may be modified interactively through an editor window or specified within a file. The syntax can be split up into *productions*, *terms*, and *parameters*. A simple BNF grammar for our language might be:

```
start      -> initial ';' prod-list
initial    -> term-list
prod-list  -> production prod-list
            | empty;
production -> term prod-param '-' '>' term-list ';'
term-list  -> term term-list
            | empty
term       -> string opt-params
prod-param -> ':' parameter
opt-params -> parameter opt-params
parameter  -> '(' expr ')'
expr       -> literal
            | literal '+' literal
            | literal '-' literal
            | literal '*' literal
            | literal '/' literal
```

```
             | literal '>' literal
             | literal '<' literal
             | literal '=' literal
literal     -> string
             | floating-point-number
```

In short, an input file consists of a number of terms which make up the initial-value or seed for the system. This seed is followed by a number of production rules (which may be parametric or boolean) that specify the rewriting rules. Terminals in the grammar are members of the set *('+', '-', '*', '/', '¿', '¡', '=', ';', ',', string, floating-point-number)*. Certain semantic checks are also made; for example, parameters to the left hand side of a *production* can only be of type *string*. Also parameters in a *prod-param* can only be one of the three boolean operators.

The following terms have special graphical interpretations when rendering an expanded string:

- F(len)- Draw a cylinder of length *len* from the turtle's current position at an angle specified by the current heading.
- !(width)- Set the thickness of subsequent drawn cylinders to be *width.*
- [- Push the current position, heading, and cylinder thickness onto a stack.
- [- Restore current position, heading, and cylinder thickness from the top of the stack (and pop the stack).
- */(deg), &(deg), +(deg)-* rotate by *deg* degrees around the x, y, or, z axis respectively. This production modifies the turtles current heading.
- *leaf(optional scaling), flower(optional scaling)-* specifies the drawing of the GL object corresponding to a leaf or flower. The optional parameter specifies a scaling factor for the object (default is 1).

3.2 Memory Management

The strings which we produce for our system can grow exponentially with the number of generations, often exceeding the size of available physical memory. To improve system performance we use the following techniques:

- *Customized malloc routines-* The default malloc routine is very small when used to allocate numerous small chunks of memory. We ask for a large chunk of memory at the beginning of the program and then perform our own memory allocation.
- *Caching of intermediate expansions-* Thus when going from *n* to *n+1* generations, much of the work of expansion has already been done. When going from *n* to *n-1* generations, no work needs to be done in expansion.
- *Storing GL expansions of strings into objects-* Storing rendered forms into internal GL objects greatly improves redraw time since there is no need to iterate over the string on every redraw.

4 The User Interface

The user interface allows for the run-time editing of images that have been produced by the program. The following list describes some of the functionality provided by the user interface:

- Production rules may be saved, loaded and edited.
- Colors may be specified (in terms of red, green, and blue components) for branches, leaves, flowers, and the light source.
- A "seed" to use in the first generation may be loaded and saved along with the production rules and may be changed by the user.
- The number of generations for which to apply the grammar may be set to any integer value between 0 and 20.
- Drawing styles to apply to the resulting image may be turned on and off. These options include Gouraud Shading/No Shading and Solid Style/Line Style.
- Images displayed on screen may be rotated, scaled and translated.
- Objects described in Berkeley UNIGRAFIX format may be loaded for use as leaves and flowers.
- Images may be saved to disk in UNIGRAFIX-format files. This and the previous feature allow users to integrate the rule-based image generation capabilities of *gen* with a wide range of three-dimensional viewing and manipulation tools available for images described in UG format.

Upon startup, an **output window** and **editor window** are shown. The output window is used to display the images produced by *gen* as they are generated. Mouse movements in this window rotate, scale, and translate the images.

The editor window handles most of the user interaction. Six production rules at a time are displayed in the editor window, although the user may scroll among a maximum of twenty such rules. Each production rule consists of three editable text fields: the axiom, an optional boolean test, and the consequent. Rules are applied in the expected way. Given a word in the grammar, each term in that word is replaced by the consequent of the corresponding production rule, if appropriate. If not, the term remains unchanged. When a boolean test is included in a rule, the consequent replaces the particular term only if the test evaluates to true.

The editor allows the user to specify a seed (initial word) and the number of rules for which to apply the grammar to that seed. Menu choices are available for loading and saving files, displaying additional graphics options, and viewing on-line help.

5 Future Work and Conclusions

Currently the system allows for the specification of only a single leaf type and flower. It would be interesting to extend the system to allow for a number of trees and flowers to be present in the same object. Furthermore, since L-systems are so general, this would allow for arbitrary objects to be rendered in our system. Such an extension should be fairly simple to implement since we it can currently read in files in UG format.

Certain modifications of the basic rewriting rules must be made for different applications. For example, with plants certain effects such as tropism and randomness in growth are difficult with rewriting rules. A system allowing the user to specify such considerations for different applications would be a powerful extension of our system.

We used the forms library to design our user interface. While very useful for rapidly prototyping a user interface, the system is inflexible and thus not appropriate for serious user interface design. In the future we would use a system such as Tcl/Tk for UI design.

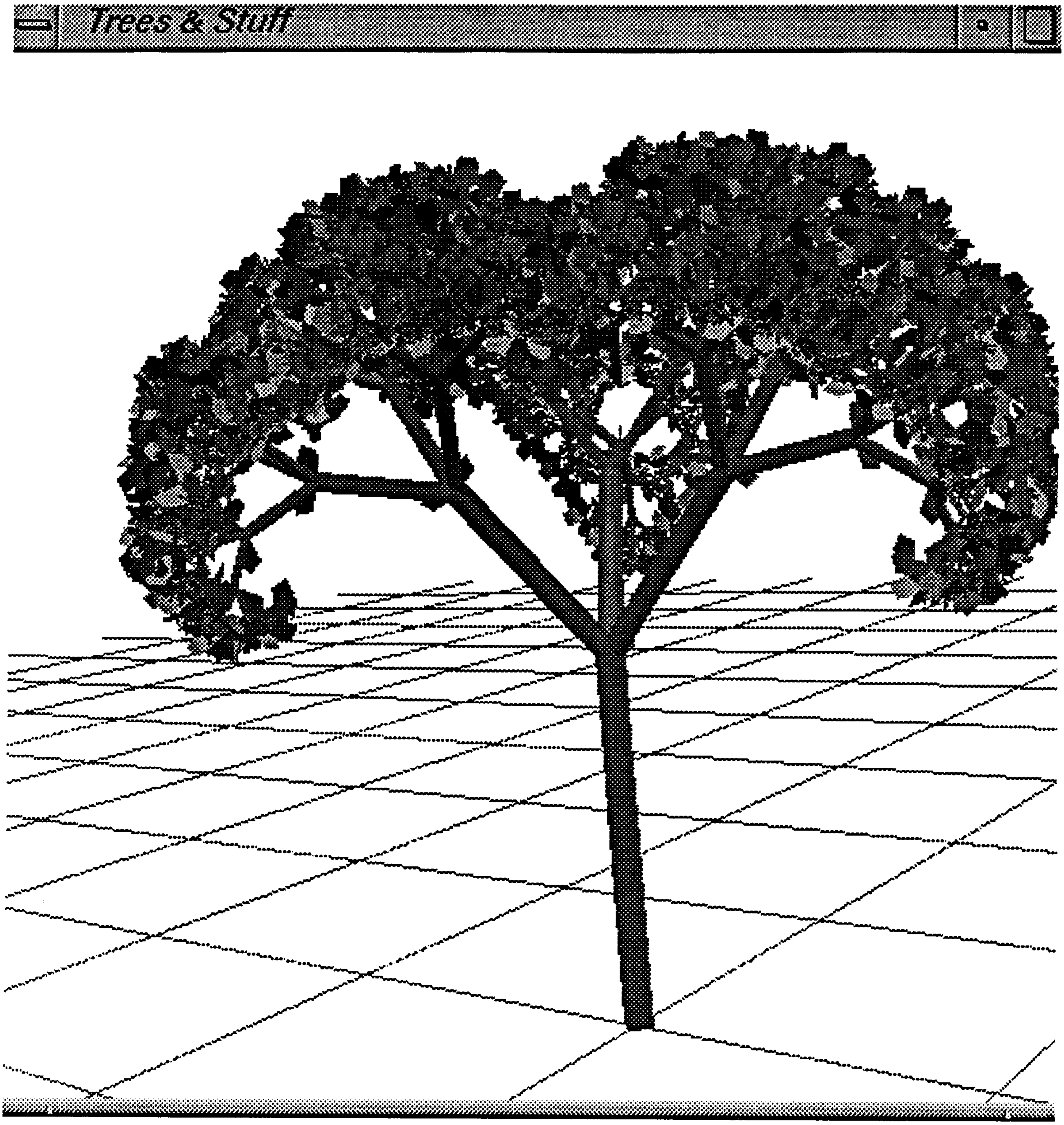
In conclusion, L-systems are a very powerful mechanism for the specification of arbitrary three dimensional objects demonstrating regular growth patterns. They are especially useful for the rendering of very realistic plant-like objects. An interactive tool for specification of these systems can assist users in quickly modeling these objects. Examples of a number of objects generated by our project can be found in attached figures.

6 Acknowledgements

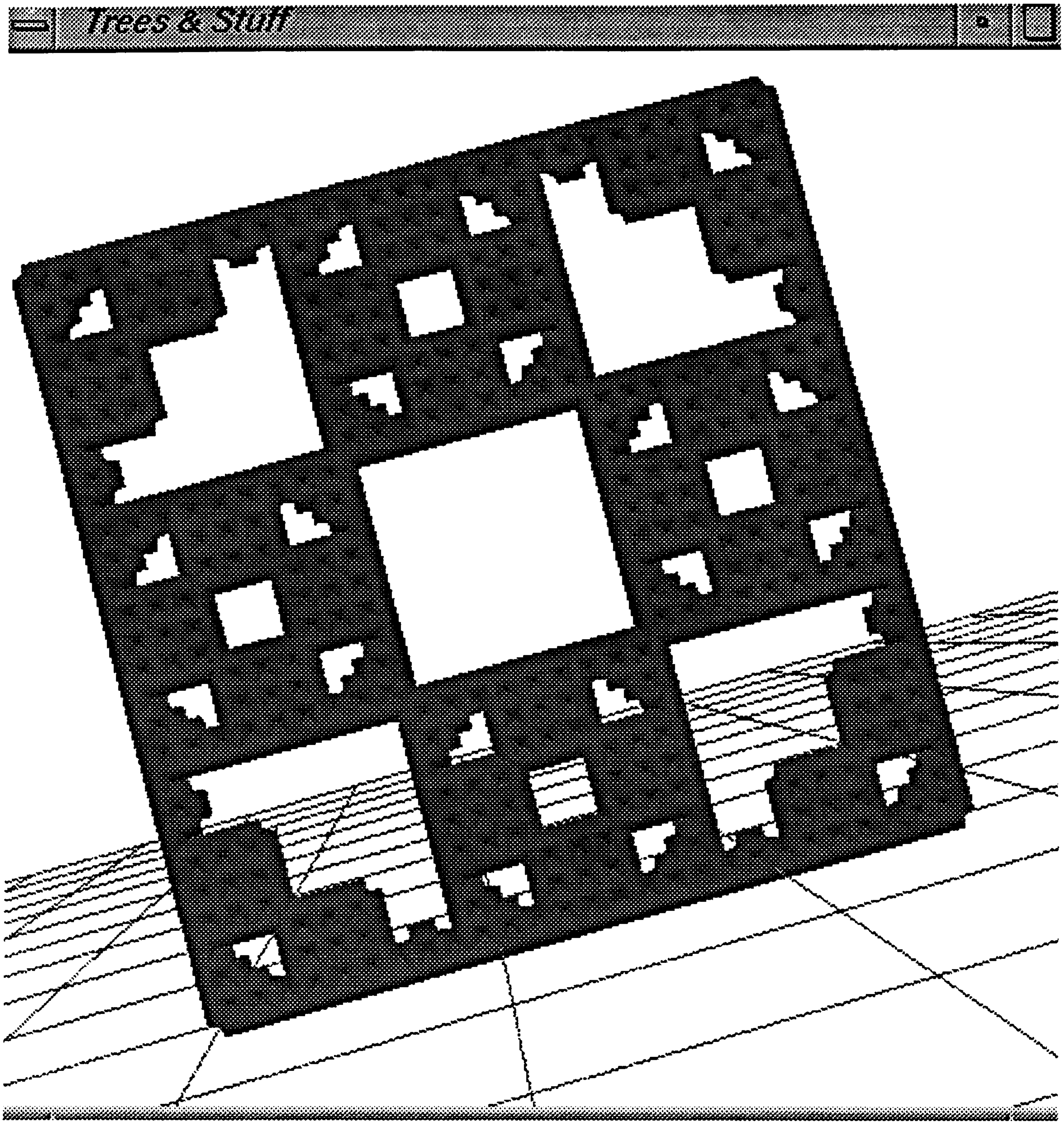
We modified code originally written by Seth Teller to rotate and scale our rendered images. The code was very well written and its incorporation into our system was quite straightforward. We would further like to thank Carlo Sequin for his feedback and comments on this project.

References

[PL90] Przmyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer Verlag, New York, 1990.



6



7

NAME

gen - generate 3D models of plants and trees

SYNTAX

gen [filename] [generations]

DESCRIPTION

gen is an interactive program for producing three-dimensional models of plants, trees and other objects. The models are generated through repeated application of production rules, which may be specified by the user. After n such applications, the resulting "word" of the grammar is given a graphical interpretation which is displayed in a GL output window.

User interaction takes place largely in three windows. The Editor is the primary application window. Users may input the production rules for a language, a symbolic "seed", and the number of times to apply the rules, beginning with the seed. Grammars may be loaded and saved from the Editor window. A menu option allows output to be dumped to a static Unigrafix file, and Unigrafix-format leaves and flowers may be loaded. On-line help may be invoked from within the Editor window.

The Options window allows users to modify the drawing parameters that are used when displaying images on screen. Colors may be specified for branches, leaves, flowers, and the light source. Gouraud shading may be turned on and off. The draw style may be set to solid drawing or line drawing. Users may show and hide the Options window via menu commands in the Editor window.

The Output window displays the graphical interpretation of a word in the grammar once the rules have been specified and applied for a given number of generations. Clicks and drags with the various buttons of the mouse can be used to rotate, scale, and translate the image.

OPTIONS

[filename]	Read in a grammar from filename and initialize the Editor to contain the rules for this grammar.
[generations]	Initially display in the Output window the result of apply the given grammar for this (integer) number of generations. Invalid when used without the [filename] option.

NOTES

Developed by Christopher Fuselier and Amin Vahdat, the University of California, Berkeley, with Professor Carlo Sequin. Refer to the on-line help accompanying the program for information concerning valid syntax and reserved words for production rules.

SEE ALSO

GL(1), ug(1)

UGGH:

(UniGrafix General Hysteria)

The System for Making a Mess!

Dan Wallach <dwallach@cs.berkeley.edu>
Adam Sah <asah@cs.berkeley.edu>

CS 285, Fall 1992
Carlo H. Sequin
December 17, 1992

UGGH was motivated by the lack of realistic clutter and mess in most computer-generated scenes. Books and papers stack up straight on desks. Clearly, this just isn't right — it's too neat. Our project attempts to model some of the ways real people generate a real mess.

The Placement Algorithm

Our goal is to make potentially messy stacks of objects on top of surfaces. The messiness is modulated by a number of parameters:

Anal Retentiveness – how rigidly boxes are aligned against each other.
Pile Height – how high piles can go.
Pile Spaciness – how much space should be between piles.
Pile Groupiness – how hard we should try to keep piles together.

By varying the sliders corresponding to these parameters, radically different messes can be described.

We start with a scene having a number of surfaces, but with no piles on them. Then, for each object we're trying to add, we will make three attempts to add it to the scene.

1) First, we try to put it on top of a pile. If we're completely anal retentive, we'll exactly center the new box on the top of the pile. As we get more relaxed, we'll deviate from the center. It's entirely possible for the placement to fail because of collisions with neighboring piles, or the center of the box being off the pile (it would

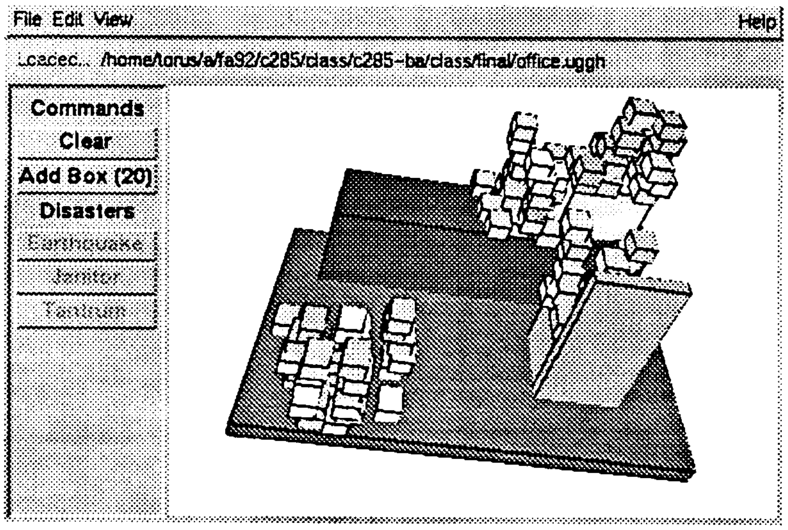
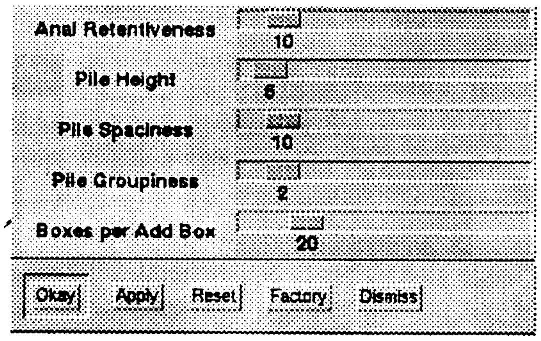


Fig. 1: UGGH User Interface.



Anal Retentiveness 10
Pile Height 6
Pile Spaciness 10
Pile Groupiness 2
Boxes per Add Box 20

otherwise fall down). If we fail, we just pick another pile and try again. We'll try more piles if Groupiness is higher. If we succeed, we're done, otherwise...

2) Second, we try to make a new pile, near a current pile. If we're completely anal retentive, we'll line up exactly on a fake grid, with Spaciness as specified. As we get more relaxed, we'll randomly deviate from the exact positions (this is more obvious to see in Fig. 2 and Fig. 3). This can clearly fail, especially when there are more objects cluttering up the ground. We'll try to be near more piles if Groupiness is higher. If we succeed, we'll make a new pile and call ourselves done, otherwise...

3) Third, we'll pick random coordinates on one of the available surfaces and try to start another pile. Eventually, this can fail, too. If we're completely incapable of placing the box, we generate an error. Generally, when over 100 boxes are scattered in messy piles covering an entire surface, one in ten attempts may fail. This turns out not to be a big deal.

Other Fun and Excitement

Usually, people do more than just bring stuff into their office. We model these behaviors with special-case code. A **Temper Tantrum** will pick up a random pile and redistribute all its members around the office. An **Earthquake** will knock over tall piles, and increase randomness. A **Janitor** will throw things out, and maybe straighten things up. Although we didn't have the time to implement these features, they wouldn't be very hard to do.

Implementation

We designed and wrote a system for maintaining clutter objects in **C++**. Clutter objects support numerous things like stacking (parent/child relationships), collision detection (via bounding boxes), and rendering (**GL** and **UniGrafix**). In our current system, the only clutter object type is the **Box**. It would be fairly simple to add an arbitrary object type, as long as it had a bounding box. It would also be straightforward to add other collision detection algorithms.

A **Tcl/Tk** interface is provided to the system. This turned out to be a big win in terms of testing and compile-time – we could test more of our objects features, and we didn't have to recompile nearly as often: about 1/4 of all the code we wrote

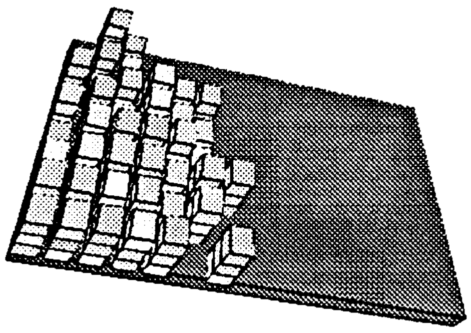
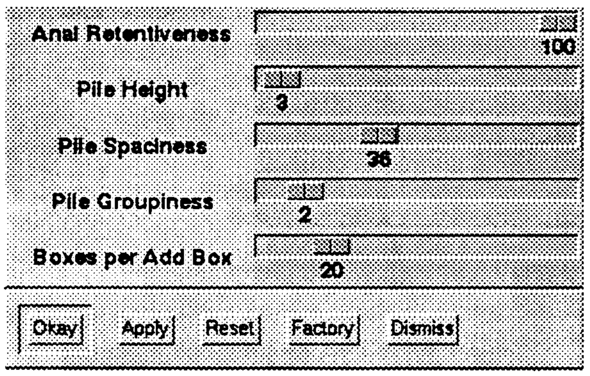
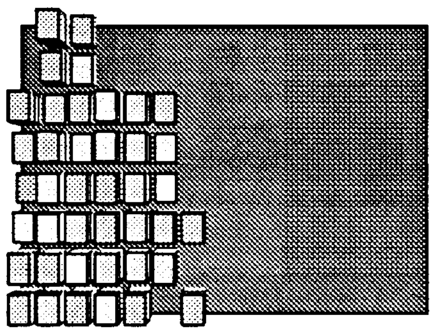


Fig. 2: High Anal Retentiveness.



Anal Retentiveness 100
Pile Height 3
Pile Spaciness 36
Pile Groupiness 2
Boxes per Add Box 20



was **Tcl**, which is interpreted and easy to change.

Limitations
Placing objects is fairly slow, usually between 1/4 and 1/2 second per box. This is mainly due to use of linked lists, internally. We also can't really deal with bookshelves, where objects stack both ways, as well as at an angle. We don't support a real notion of gravity; some of the stacks that we generate would never stand up in the real world.

Overall
The system solved its main design goals – creating a scene with a big mess of objects. Subjectively, our messes look like real messes. UGGH!

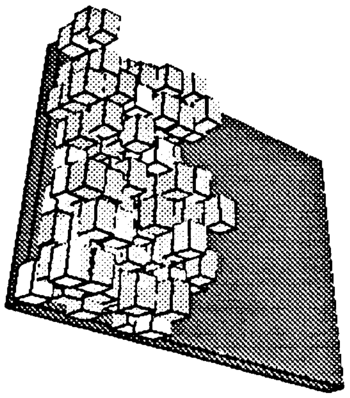
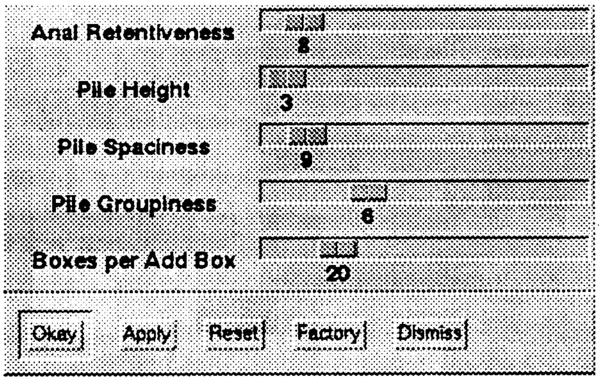
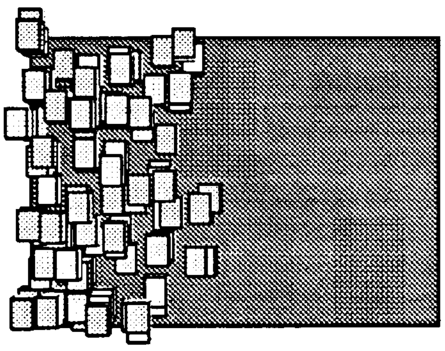


Fig. 3: Low Anal Retentiveness.



Anal Retentiveness	8
Pile Height	3
Pile Spaciness	9
Pile Groupiness	6
Boxes per Add Box	20



Ughull - A 3D Convex Hull Utility for the UniGrafix System

Robert H. Wang

1. INTRODUCTION

Ughull generates a convex hull of a set of points in three dimensions. If the point set is randomized, ughull exhibits linear CPU time behavior. Ughull is compatible with other UniGrafix tools such as ugshrink and ughole. Figure 1a-c show the convex hulls of 10, 100, and 1,000 points which are randomly generated inside a 2x2x2 bounding box centered at (0,0,0). Figure 1d plots the performance of ughull for 1,000, 5,000, and 10,000 points running on the Silicon Graphics Personal Iris and IBM RS/6000 Model 530 workstations. The faces of the convex hulls have been shrunken using ugshrink to demonstrate compatibility with other UG tools and highlight the tessellated hull surfaces.

2. TECHNICAL DESCRIPTION

Ughull is a three-dimensional extension of Ken Clarkson's algorithm [1] in that it incrementally builds an approximate tetrahedralization of the point set and creates the convex hull from exposed vertices and triangles. As Figure 2 illustrates, once the initial triangle is found, the two-dimensional Clarkson algorithm ignores collinear degeneracies which may arise during vertex addition, and avoids the unnecessary and undesirable task of splitting existing triangles. This fact greatly simplifies the three-dimensional implementation and eliminates the difficulty typically found in gift-wrapping algorithms of adapting two-dimensional convex hull algorithms to handle coplanar points.

Therefore, extending the Clarkson algorithm to three dimensions mostly reduces to replacing two-dimensional operations with appropriate, more expensive three-dimensional operations. For instance, initializing the tetrahedralization involves handling coplanar as well as collinear degeneracies. Point-triangle visibility is used rather than point-edge visibility, where a triangle on the plane $Ax + By + Cz + D = 0$ is visible to point P at (Px,Py,Pz) if A (Px) + B (Py) + C (Pz) + D is strictly greater than 0 plus some tolerance. Hidden triangles must be detected instead of hidden edges, and it is slightly more expensive to check for identical triangles. Figure 3 illustrates some of these operations by showing a three-dimensional analog of the example in Figure 2. As the tetrahedralization extends beyond an exposed tetrahedron, the exposure state of the tetrahedron and its component triangles are appropriately toggled. Incrementally updating the exposure states saves processing time by enabling the program to ignore hidden tetrahedra and triangles during each vertex addition, and makes extracting hull vertices and triangles a straightforward task.

Ughull uses a light-weight extension of the basic vertex-triangle-tetrahedron list data structure which is customized for efficient point-triangle visibility computation and rapid identification of exposed vertices, triangles, and tetrahedra. In addition to basic connectivity information, each vertex, triangle, and tetrahedron contains flags indicating their exposure and visibility states. The plane equation for each triangle is cached for visibility calculations and is computed during triangle creation by taking the cross product of the direction vectors of its edges.

3. CONCLUSION

A three-dimensional convex hull utility, ughull, is available for the UniGrafix system. By decomposing the convex hull problem into a sequence of hull vertex additions rather than hull face additions removes collinear and coplanar degeneracies and makes ughull fundamentally robust. Ughull is easy to maintain and efficient because its data structures and routines are self-contained, portable, and customized for point-triangle visibility calculations and hidden-triangle removal.

An interesting extension to this work is to build a companion tool to refine the hull surface tessellation. One of the original goals of the project was to generate tessellated convex hulls suitable for numerical simulation. However, randomly generated point sets tend to result in slivers and large facets which form meshes that are unsuitable for applications such as simulation of surface diffusion.

REFERENCES

[1] Seth Teller (seth@miro.berkeley.edu), private communications.

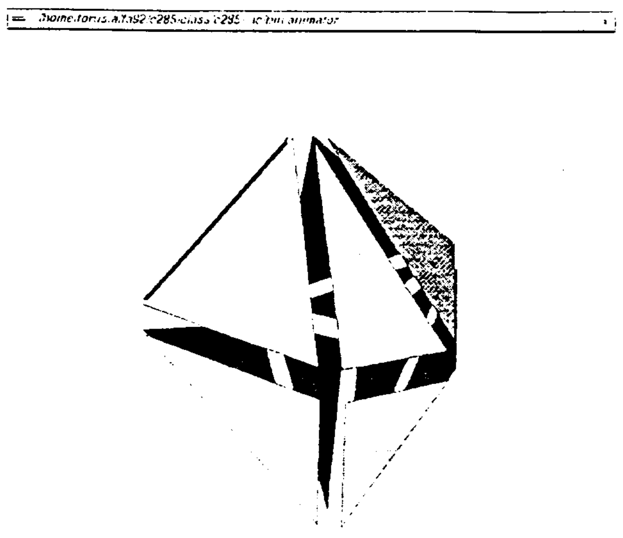


Figure 1a. Convex hull of 10 points with shrunken hull faces.

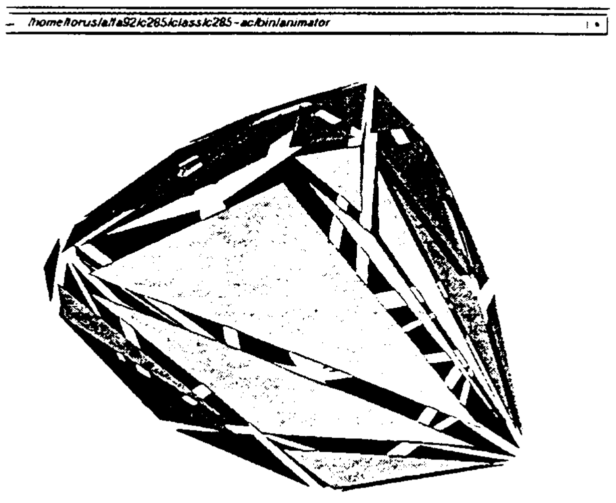


Figure 1b. Convex hull of 100 points with shrunken hull faces.

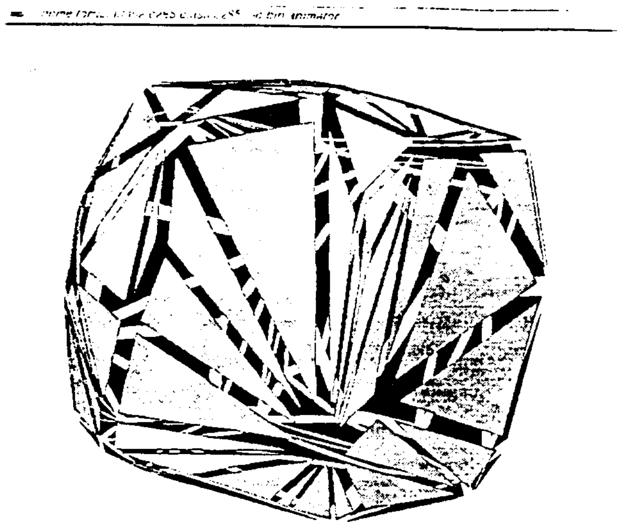


Figure 1c. Convex hull of 1,000 points with shrunken hull faces.

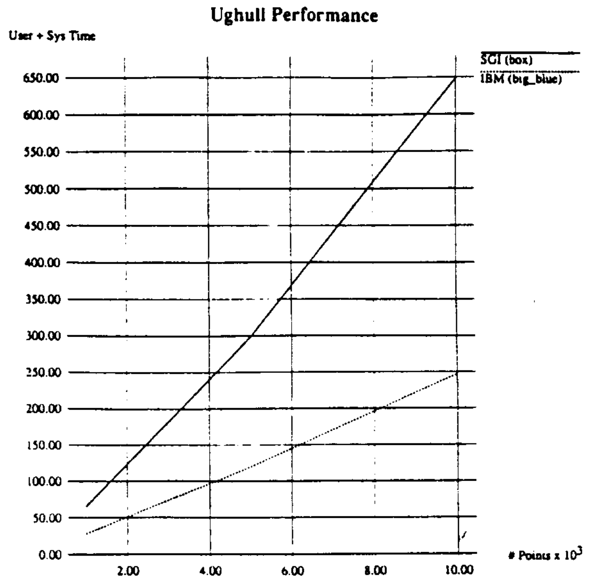
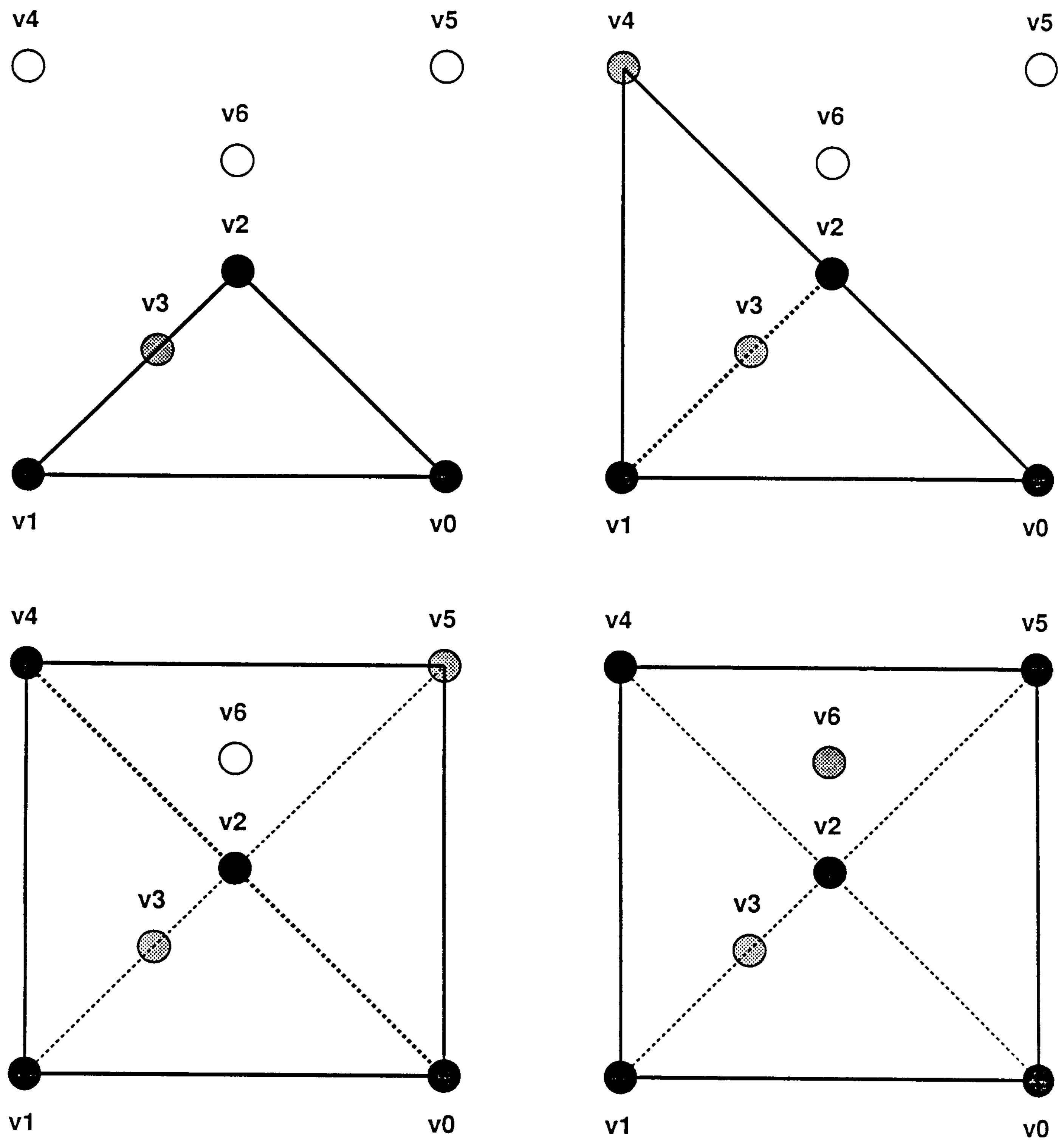


Figure 1d. Ughull performance for 1,000, 5,000, and 10,000 points.



The Clarkson algorithm naturally removes degeneracies such as the one involving v3. This fact greatly simplifies a 3D implementation.

Figure 2. Illustration of Ken Clarkson's algorithm in two dimensions.

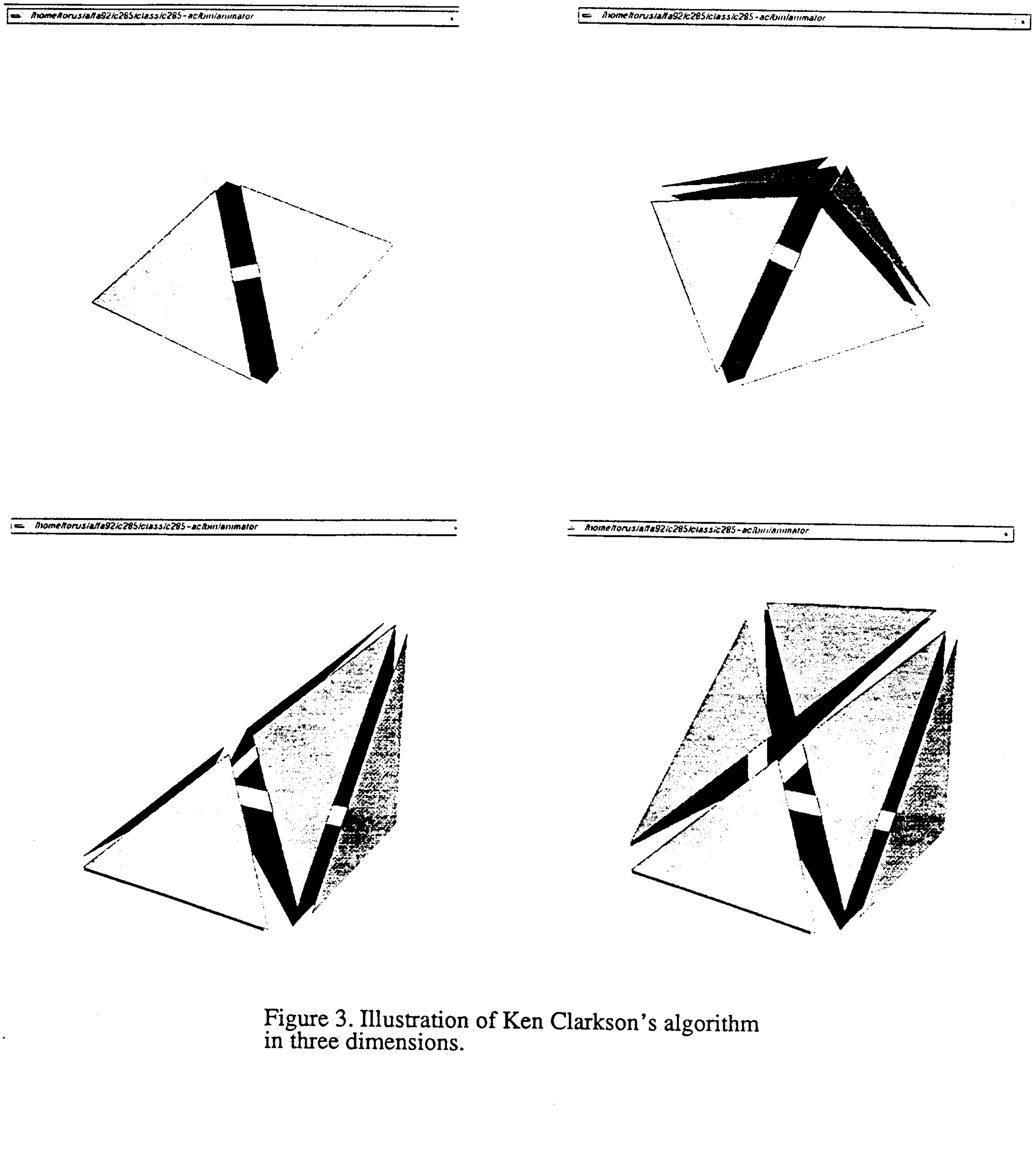

/home/torus/a/fa92/c285/class/c285-ac/bin/animator
/home/torus/a/fa92/c285/class/c285-ac/bin/animator
/home/torus/a/fa92/c285/class/c285-ac/bin/animator
/home/torus/a/fa92/c285/class/c285-ac/bin/animator

Figure 3. Illustration of Ken Clarkson's algorithm in three dimensions.

NAME

ughull – construct 3D convex hull of a set of vertices stored in UG format

SYNOPSIS

ughull [**–i** *input_file*] [**–o** *output_file*] [**–tolerance** *tolerance_value*]

DESCRIPTION

Ughull takes a set of vertices and generates its convex hull in the form of a tessellated polytope using a three-dimensional extension of Ken Clarkson's algorithm [1]. Unless otherwise specified, the UG input file is read from standard input. Internally, the **ughull** parser ignores all UG statements except the vertex statement and the convex hull is extracted from a tetrahedralization. Unless otherwise specified, the vertices and triangular faces of the convex hull in UG format are written out to standard output. **Ughull** also generates a log file and an UG file for viewing the tetrahedralization.

OPTIONS

–i	Read vertices from an user specified file. Default is standard input.
–o	Write convex hull to an user specified file. Default is standard output.
–tolerance	Set tolerance value for incidence calculations. Negative values are ignored. Default is 1.0e-08.

DIAGNOSTICS

tetrahedralize: Fewer than four vertices

Exits because user specifies fewer than four vertices.

get_first_tetrahedron: Fewer than three non-collinear vertices

Exits because the program can't find at least three non-collinear vertices to initialize a first triangle for the tetrahedralization. This error message will always be followed by the next message.

tetrahedralize: Fewer than four non-coplanar vertices

Exits because the program can't find at least four non-coplanar vertices to initialize the tetrahedralization.

BUGS

Probably should handle the cases of fewer than four non-coplanar vertices more elegantly.

For large input, the log file and additional plot files will consume substantial disk space. For instance, running **ughull** with 10,000 points generates a log file and plot files totaling 3 MB.

EXAMPLE

```
ughull -i foo.ug -o bar.ug -tolerance 1.0e-12
```

Generates the convex hull of the vertices stored in foo.ug and writes to bar.ug. Use a tolerance value of 1.0e-12.

```
generator 100 | ughull | ugshrink -f 0.80 | ugiris
```

Generates 100 points within or on the bounding box of (-1,-1,-1) and (1,1,1) and create a convex hull around those points. Shrinks the facets in the convex hull so that the tessellation can be viewed using ugiris.

REFERENCES

[1] Seth Teller (seth@miro.berkeley.edu), private communications.

CS 285 Final Project Report
Ug4view – An Interactive Viewer with Graphical User Interface for 4-Dimensional UniGrafix Objects

Allan Christian Long, Jr. (allanl@cs.berkeley.edu)

December 11, 1992

1 Introduction

Ug4view is a program that allows interactive viewing of 4-dimensional objects. It is essentially an enhanced version of ugiris4d (which itself is based on ugiris). Ugiris4d used keyboard commands and mouse dragging on the image window itself as the only means of interaction. Ug4view adds a GUI control panel for controlling the interaction as well as animation.

2 How to Use It

Ug4view uses two different windows, the window in which the object is displayed (the "object window") and the control panel. Each has its own interaction techniques associated with it.

2.1 Object Window

To maintain compatibility, the user may interact in this window as if it were the ugiris4d window. These interactions are summarized below. An example object window is shown in Figure 1.

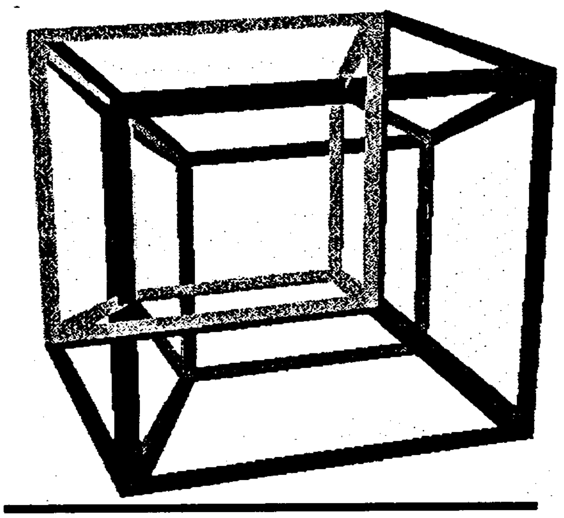


Figure 1: Example object window

2.1.1 Zooming

Dragging with the left mouse button zooms the image. Left-to-right drags move the viewpoint closer, right-to-left farther away.

2.1.2 Scaling

Dragging with the middle mouse button and the shift key pressed scales the image. Left-to-right scales up, and right-to-left scales down. The eye point and the object position are unchanged.

2.1.3 Rotation

`Ug4view` allows the user to rotate the object through 4-space, or to rotate the projection of the object in 3-space as if it were a normal 3-dimensional object. The way the object is rotated is determined by the Dimension radio buttons on the Control Panel (see Figure 2 and Section 2.2.3).

In 4D mode, the user may rotate the object in any of the six principle coordinate planes by dragging the middle mouse button in combination with keyboard modifier keys. With no keys pressed, left/right movement causes rotation in the x-z plane and up/down in the y-z plane. With the left shift key pressed, left/right rotates in the x-y plane and up/down in the z-w plane. With the right shift key pressed, left/right rotates in the x-w plane and up/down in the y-w plane. (This is the same as in `ugiris4d`.)

In 3D mode, the user may rotate the *projection* of the object about the three principal axes by dragging the middle mouse button in combination with keyboard modifier keys. With no keys pressed, left/right rotates the projection about the y-axis and up/down about the x-axis. With the left shift key pressed, left/right dragging rotates about the z-axis. (This is the same as in `ugiris`.)

2.1.4 Translation

In 4D mode, dragging the right mouse button translates the object. With no keys pressed, left/right motion translates along the x-axis and up/down along the y-axis. Pressing the left shift key causes left/right motion to translate along the z-axis and up/down along the w-axis.

In 3D mode, the right mouse button translates the object. The controls for this are identical to those for translation while in 4D mode(see above), except that translation along the w-axis no longer occurs.

2.2 Control Panel

The Control Panel is the major advantage of `ug4view` over `ugiris4d`. It provides the user with a more intuitive interface to many options affecting the display of the four dimensional object. It is written using the Simple User Interface Toolkit (SUIT), developed at the University of Virginia.

The use of SUIT enables `ug4view` 's interface to be easily modified while the program is running. Knowledge of SUIT is not at all necessary to use the interface, only a familiarity with common mouse-style interaction. A full explanation of SUIT is beyond the scope of this paper. Interested users are encouraged to look at the SUIT tutorial[1]. If the control and shift keys are held down, the left mouse button and certain keys interact with the interface instead of the application. The left mouse button moves widgets when dragged and selects them when clicked. If a widget is selected, it may be resized by dragging the handles that appeared when it was selected. Use SUIT-M (i.e. hold down the control and shift keys and

[1]`Tutorial.ps` is installed on the torus cluster in `/usr/local/suit/doc`. It is only ten pages long, unlike the reference manual, which is about 160.

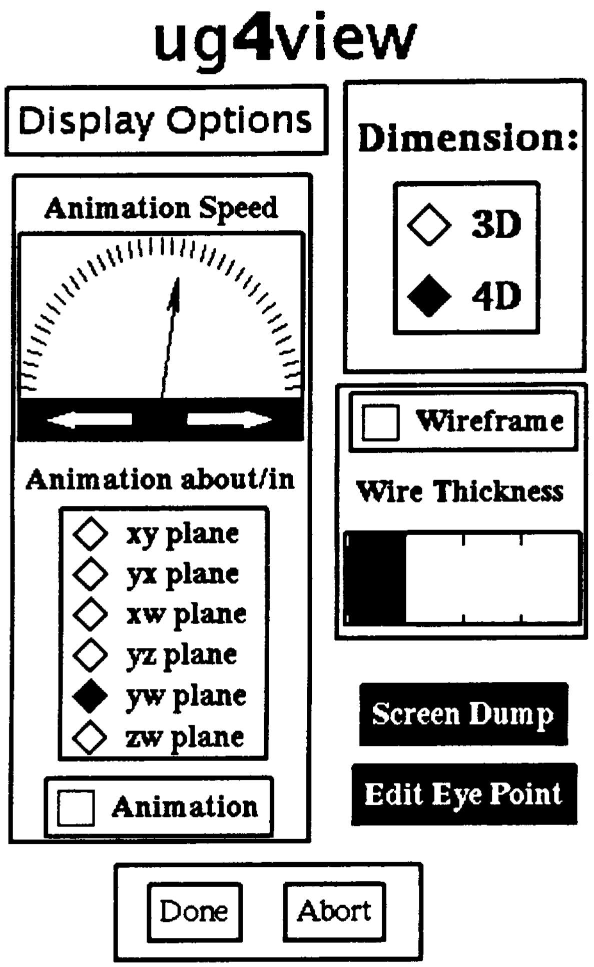


Figure 2: Control Panel

press "M") to bring up the SUIT menu for other SUIT commands[2].

2.2.1 Wireframe Selectors

This collection of widgets enables the user to toggle the display of the wireframe and the thickness of the wireframe. (See also the Show Faces option of the Display Menu (section 2.2.4 and Figure 3).)

2.2.2 Animation Selectors

These widgets allow the user to animate the object, either in three or four dimensions. The animation mode is controlled by the Dimension radio buttons. The user may select the plane in which or the axis about which to rotate the object. The user may also turn animation on and off and change the animation speed.

2.2.3 Dimension Selector

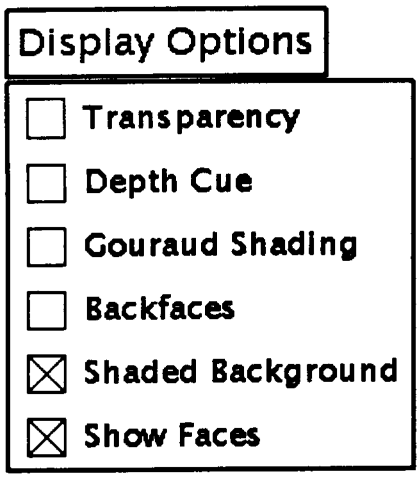
This widget selects between 4D mode and 3D mode, the two basic modes in which `ug4view` operates. This widget determines the planes through which and the axes about or along which the object may be animated, rotated, or translated. See sections 2.2.2, 2.1.3, 2.1.4.

2.2.4 Display Options

This menu contains check boxes to toggle infrequently-used display options. (See Figure 3.) The Display Options menu contains the following toggles:

Transparency Toggles on transparency on machines that support it. Also toggles the z-buffer on when transparency is off, and

[2]For more information about SUIT, anonymous ftp to `uvacs.cs.virginia.edu` and look in `/pub/suit`, or send email to `suit@virginia.edu`.


Display Options

☐ Transparency
☐ Depth Cue
☐ Gouraud Shading
☐ Backfaces
☒ Shaded Background
☒ Show Faces

Figure 3: Display Option Menu

off when transparency is on. If the machine doesn't have alpha bitplanes, only the z-buffering is affected.

Depth Cue When on, varies the shade of each part of the object depending on its distance from the viewpoint.

Gouraud Shading When on, varies the shade of each part of the object depending on its w-coordinate.

Backfaces Toggles the hardware backface capability. When on, the back faces of polygons are drawn. When off, polygon back faces are not drawn.

Shaded Background Toggles between the shaded gray background and a solid white background. It is intended to be used primarily for screen dumps, for which a white background is preferred.

Show Faces Toggles the display of polygon faces. See also section 2.2.1. Note that if both the Show Faces and Wireframe check boxes are off, nothing will be displayed.

2.2.5 Screen Dump

This button outputs the current 3D projection as a UniGrafix file. A dialog box is used to allow the user to pick a file name.

2.2.6 Edit Eye Point

This button invokes a dialog box in which the user may manually set the eyepoint to anywhere in 4-space.

2.2.7 Done

This button quits the application and saves the state of the user interface (writes the `ug4view.sui` file). If it cannot write the hints file, it will print an error message and quit anyway. For information about how to modify the interface itself, see section 2.2.

2.2.8 Abort

This button exits the application without saving the user interface.

3 Implementation Details

`Ug4view` is not a great deal more complex than `ugiris4d`. The kernel of the program is essentially the same. The input and output routines have been slightly modified to integrate with the SUIT control panel. Details about the "back-end" can be found in the `ugiris4d` documentation.

The program has been converted from K&R C style into ANSI C style with the help of GNU's `protoize` utility. This should help in both readibility and enable greater type and other checking by the compiler.

The most invasive modifications to ugiris4d were made to its input loop to accomodate the use of SUIT, which is based on external control[3]. I anticipated great difficulties in obtaining input with raw GL code (as in the object view window) and with an X window (the SUIT control panel) simultaneously.

My first strategy was for the code to keep track of which window had focus and only allow that window to attempt to get input. However, that did not work becuase the program was sometimes wrong about which window had focus, especially at startup. After spending considerable effort trying to debug that method, I discovered that the simplest solution worked. The solution that I found was to call the GL input loop until no more events remained and then to call the SUIT input routine so it could handle any events that had occurred. This works flawlessly, except that occassionaly the object view window will not repaint itself when it should. This is not a major problem, as any subsequent interaction causes a refresh anyway.

The major additions to ugiris4d are in the file ug_suit.c, which are prototyped in ug_suit.h. Routines in this file create the SUIT widgets with which ug4view interacts and manage events from them. I would recommend that anyone who wants to modify this code first do the SUIT tutorial, as the purpose of the SUIT functions may not be immediately obvious to the completely uninitiated. For serious modifications or additions, the reference manual (or at least portions of it) may be needed.

The only really new feature that was added was animation. This is accomplished by checking to see if animation is on each time through the input loop. If it is, the object is rotated about the specified axis or through the specified plane (depending on the Dimension radio buttons and the axis or plane selected) by the specified number of degrees (from the speedometer). (See also section 2.2.2.) There are currently two problems with animation: 3D mode is much slower than 4D mode and the user cannot animate in an arbitrary direction by dragging the mouse (as in many SGI demos).

A great deal of the implementation time was spent interactively customizing the appearance of the interface. For SUIT programs, the .sui file (the one that contains the informataion about the interface, also called the "hints" file) is almost as important as the .c files. The .sui should be guarded as if it were source code. (See also sections 2.2.7 and 2.2.8.)

[3]For a good introduction to external control and an example thereof, see the appendix in the SUIT tutorial.

4 File Format

Ug4view uses a slightly extended form of UniGrafix files. It is completely compatible with standard UG files, except for the vertex statement. The extended syntax is:

```
v id x y z w;
```

Lighting at the vertex level is ignored.

5 Future Work

While I think that ug4view is a distinct improvement over ugiris4d, there is room for improvement. Unfortunately, I did not have time to user-test the user interface, so I probably do not even know of all the improvements that could be made. Here is a list of some things that I think would enhance the program:

- Optimize the program for efficiency. (glprof and/or prof would probably be very helpful.)

- Allow animation in arbitrary directions (as in SGI demos).

- Add motion blurring. (I saw a demo of motion blurring at UIST '92 and am convinced that it can greatly improve virtually *any* computer animation. Ideally, this would use transparency, but I think it could be done without alpha bitplanes.)

- Allow a new file to be loaded without quitting and restarting.

- Add a slider that controls the rotational position of the object in a plane. This would let the user find the exact points where significant changes in the object's projection occur.

- Remember the position of the eyepoint from one run to the next (using the SUIT "hints" file). This would be useful for going back to the same object again and again, as I did many times. Or better yet, store the viewpoint in the UG file (as a comment, I suppose).

- User test the interface. I think both users who have experience with 4-dimensional visualization and those who have no such experience would provide useful feedback (it would of course be up to the UI designer how to weight their feedback).