PROCEDURAL MODELING

Carlo H. Séquin

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

With contributions by:

R. Bukowski, C. Chan, D. Garcia, R. Keller,
R. Lewis, P. Liu, S. McMains, M. Schiff, J. Sefler,
S. Tager, B. Vaysman, K. Vetter, Z. Yang.

ABSTRACT

This is a report on the seventh offering in the Fall of 1994 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 range from a juggling demonstration program, through interactive tools to control the motion of walking sticks figures or to study the evolution of plants described in differential L-languages, to a haunted walkthrough maze. The projects have been developed on SGI personal IRIS workstations or in a generic X-window framework; the geometric descriptions of the objects typically use the Berkeley UniGrafix language; Tcl and Tk are sometimes used for the user interface.

Table of Contents

INTRODUCTION

This is the seventh offering of a special graduate course concerning the procedural generation of computer graphics models and of interactive scenes. 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 aspect became ever more important as many of the assignments and final projects involved time-varying interactive objects. The instantaneous graphical feedback evokes an extra level of enthusiasm and motivation, and the results achieved by the students are correspondingly high.

In the Fall of 1994, the CS Division had just moved into its new home in Soda Hall. The Iris lab was heavily overloaded by a record number of students in the introductory graphics course CS 184, and its use for CS 285 became rather difficult and frustrating. So most students were looking for other places to do their homeworks and to develop their projects. More than half the students had access to some SGI research machines, others used a generic X-window environment on other graphics workstations. The final demonstrations of the projects took place in a graphics research lab.

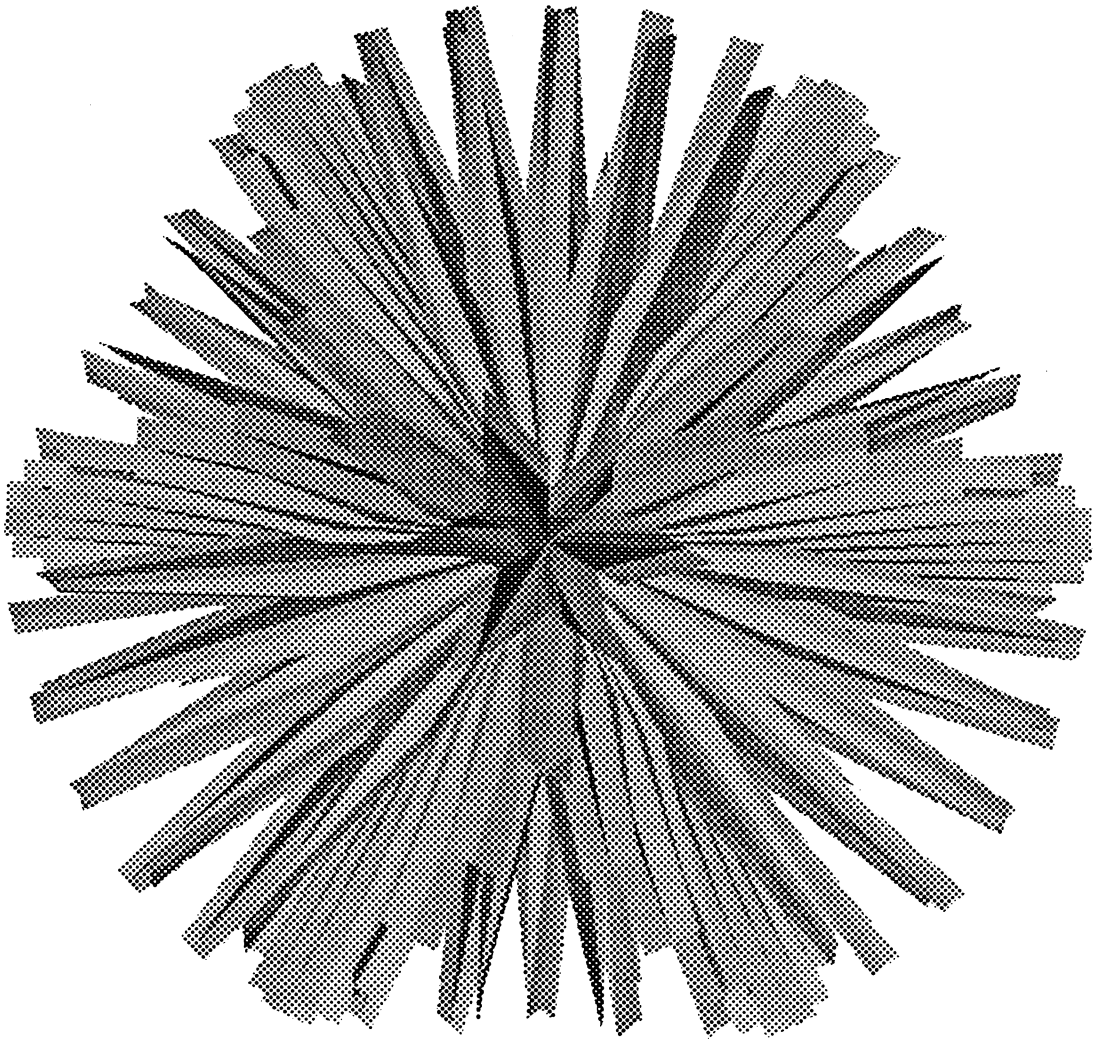
As in the past, the course had formal homeworks during the first ten weeks of the term and concentrated on individual course projects during the last third. Learning from the success of the offering two years ago[5], most assignments were done in teams of two or more students and a prescribed rotation 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. The final project could be done individually or in teams of two.

This course is a lot of fun to teach, primarily because of the tremendous energy and enthusiasm of the students who are willing to put in the effort it takes to deal with twenty assignments and two formal presentations in just 15 weeks. I am grateful to all participants and hope that they too found this to be an enriching experience.

Carlo Séquin

---

1. C. H. Séquin, "Creative Geometric Modeling with UniGrafix," T.R. UCB/CSD 83/162, Dec. 1983.
2. G. Couch, "Berkeley UniGrafix 3.1 - Data Structure & Language," T.R. UCB/CSD 94/830, Sep. 1994.
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.
5. C. H. Séquin, "Generation of Time-Varying Geometrical Models," T.R. UCB/CSD 93/729, Feb. 1993.



STEREOSCOPIC SURFACE MODELLER

Course Project Report, CS 285, Fall 1994

Christopher S. Chan
U.C. Berkeley, Dept. of EECS

ABSTRACT

For my course project, I implemented a program that attempts to generate a 3D model of a surface from a stereo pair of renderings of it. In this report, I review and evaluate my approach.

1. Introduction

The process of generating a 3D model from multiple views of a scene has many applications in a wide range of fields. In the past, stereo methods have been used in cartography to create elevation maps of the surface of the Earth. Additionally, stereo analysis is often used in medical imaging, when 2D cross-sectional images of body parts do not provide enough information and taking a mold would be inappropriate. I have implemented a stereo analyzer that will take two 2D representations of a surface, and generate a 3D model of it.

2. Overview

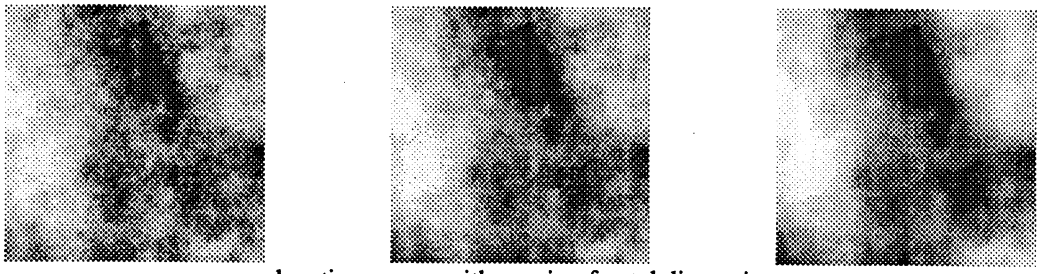
This project consists of a number of relatively small programs, because I have broken my task down into a number of phases. The first program is a simple fractal landscape generator. I needed to be able to generate interesting test cases of 3D surfaces (actually 2.5D, since the output is just a 2D array of elevations). I have also written a program that converts these arrays to UNIGRAFIX and SDL (Scene Description Language) files. The next program is a simple stereo viewing program which renders two wire frame views of a scene, each with their own independent viewing parameters. Then I have written an analyzer program which reads the window coordinates of the two generated views and generates a 3D surface description according to the specified viewing parameters. The output of this program is a UNIGRAFIX file which can be manipulated using animator or ugiris and compared to the original. Finally, I have written two other programs which I have used to further explore 2D to 3D methods. The first is a tie-pointing utility, allowing the user to display two TIFF image files simultaneously and mark corresponding points in the images. Window coordinates of these points can be saved to a file so that they may be analyzed by a stereo solver. The second utility generates shading maps, which I wrote in order to experiment with shape-from-shading methods for generating 3D surfaces from images.

3. Landscape generator

The fractal landscape generator I wrote uses the successive random addition method, which is described in just about any book on fractals (I found it in Mandelbrot's famous book). It is analogous to the midpoint displacement method, only at a higher dimension. Initially, elevations are selected randomly along 8 border points along the edges of the 2D array and at the center of the array. At each generation, the array can be divided into a grid of squares whose corners are at the locations where elevations have already been determined. To generate elevations for the next generation, the center of these squares are located. The elevation at these center locations are interpolated between the four surrounding corner points, and a random offset is added. This process continues until the array is filled. The size of the array is determined by the number of generations specified by the user.

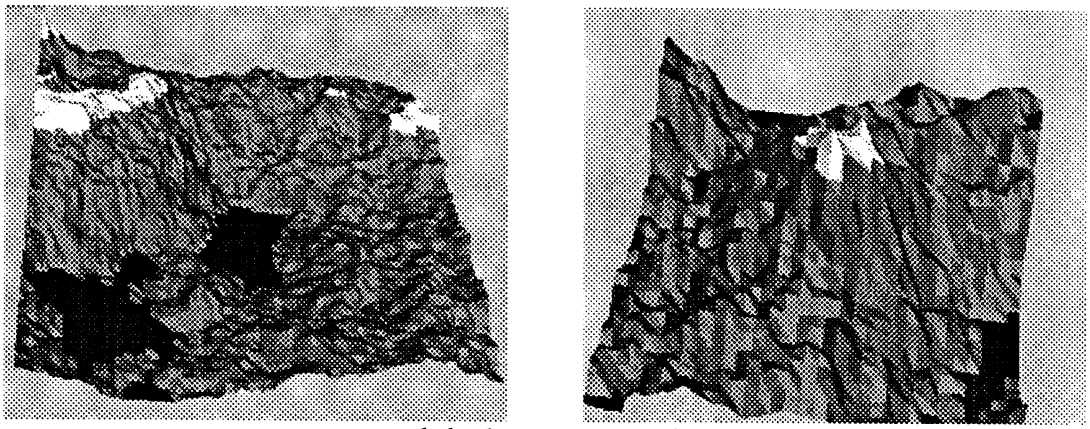
The jaggedness of the elevation map can be controlled by a fractal dimension parameter, ranging from 2.0 (smoother) to 2.5 (more jagged). The output of the generator is a 2D array of elevations. One interesting thing about the fractal mountain generator is that if the resulting elevation values are quantized into brightness values in a pixmap, the results look like pictures of clouds. The images below were produced using the same random numbers (the pseudo random number generator was initialized with the same random seed). The only difference is the fractal dimension parameter, which affects the

variance of the Gaussian random variable used to determine the random offsets. From left-to-right, the dimensions were 2.5, 2.3, and 2.1.



elevations maps with varying fractal dimension

I needed to be able to convert these 2D arrays into models which can be read by a 3D renderer. I wrote a simple conversion utility which translates the 2D elevation arrays into UNIGRAFIX description files. For aesthetic purposes, the program colors the mountains brown, colors land at sea level blue, and colors land at higher elevations white (snow).



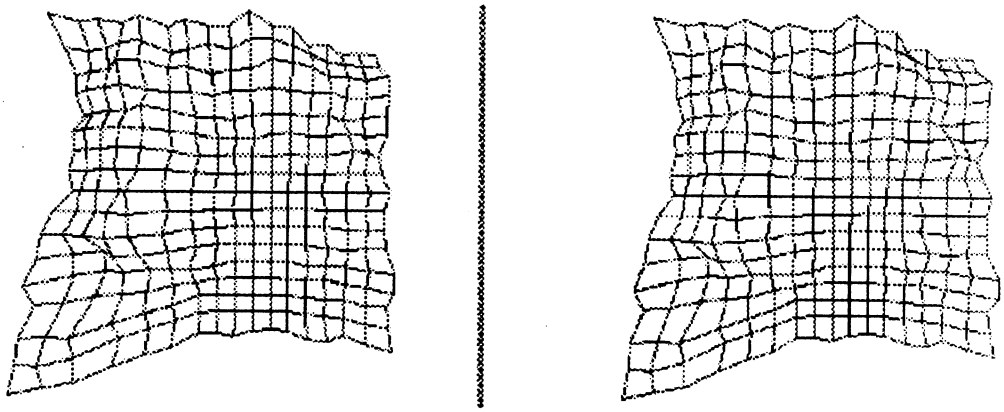
sample landscapes rendered by animator

4. Stereo renderer

In order to perform stereo analysis, I needed to be able to control and have access to viewing parameters, namely the center of projection, port window, and view plane normal parameters. None of the available UNIGRAFIX renderers permit this. The GLIDE format does allow the user to specify these parameters.

However, in order to perform stereo analysis, I must also be able to find the exact locations of corresponding points in each of the views. Finding corresponding points in images is a hard problem and image registration is an area of fairly active research in computer vision, so I decided to cheat and extract the window coordinates of points in the scan converter phase of the renderer. Instead of trying modify the GLIDE viewer code, I decided that it would be easier if I added the modifications to the renderer that I wrote when I took Introduction to Computer Graphics (CS 184) a year ago. This also meant that I also needed to make a program that converts landscape descriptions into the format of my old renderer (the SDL format).

In adding the modification allowing the user to output window coordinates to a file, I also decided that I could make my viewer more fun and interactive if I allowed the user to view two separate renderings simultaneously. Thus, if the user positions the two viewing windows side-by-side and specifies the correct viewing parameters, one can use the cross-eyed technique (or swap windows and use the parallel technique) to view the scene in 3D.



stereo pair of wire frame renderings

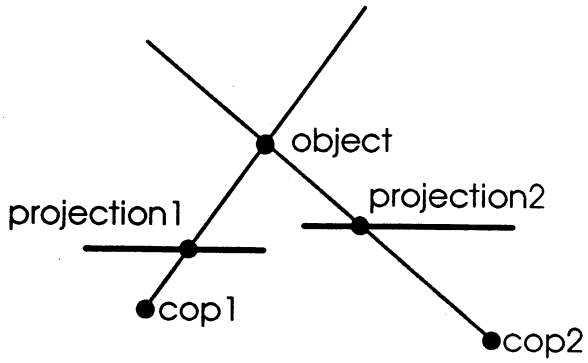
Finally, I decided to fix a few of the view variables, primarily because I felt that it was unnecessary to allow too much control over the scene. In my renderer, I have fixed the view up vector to (0,1,0) and the view reference point to (0,0,0). If I did allow these to be modified, it only would have meant that there would be a few more parameters to the stereo analyzer program.

5. Stereo analysis method

The stereo renderer outputs the location of points in window coordinates. So, in my stereo analyzer, I first convert window coordinates into view coordinates, using the port window parameters specified by the user.

Next, I have to convert view coordinates into world coordinates. This is done by first generating the world coordinate to view coordinate matrix (using the specified view plane normal parameter and the fixed view up vector and view reference point parameters) and computing the inverse of this matrix. I must also make sure that the user specified center-of-projections are also converted into world coordinates. Once everything has been converted to world coordinates, I can then triangulate to find the exact location of the projected point in three dimensions.

For each point in each rendering, there is a line going through the center of projection to it's projection on the projection plane. The location of the projection plane is known (the plane distance parameter is supplied by the user), so finding the equation of this line is trivial. In order to find the location of the actual point in 3D space, I merely need to find where the lines in each rendering intersect. The lines are guaranteed to intersect as long as the parameters are specified correctly.



I have tested the analyzer on a number of cases with different values for the view plane normals and centers of projection. In each case, the program produces an identical model of the original.

I was satisfied with the results, so I proceeded to attempt to model objects from stereo pairs of digitized photos of scenes. I've created a stereo image viewing utility allowing the user to view two images at the same time and mark corresponding points in the images. The window locations of the points can then be saved to a file which can be sent to the stereo analyzer.

I found a stereo pair of images of a shopping mall that I tested my analyzer on. I manually picked the corresponding points (because I was never able to get my image matcher working) in the two images, trying to see if my analyzer would correctly model objects in the pictures. The objects I attempted to model were a window frame in the foreground, a piece of an arch in the middle of the picture, and a hand railing that went from the foreground to the background. After I collected the coordinates of these points, I was ready to feed the values to the analyzer. However, since I did not take the photos myself, I did not know what to specify for the viewing parameters for the analyzer. Basically, I had to resort to random guessing. None of the results were satisfactory.

6. Evaluation

In order to gain some insight into what kinds of errors occur when incorrect parameters are specified in the stereo analyzer program, I ran a series of tests. I made a test SDL file consisting of a square with sides of length 8 and at depth 20. I generated two views. The first view had center of projection (-5,0,0) while the other had (5,0,0). Both had projection planes at z=10.

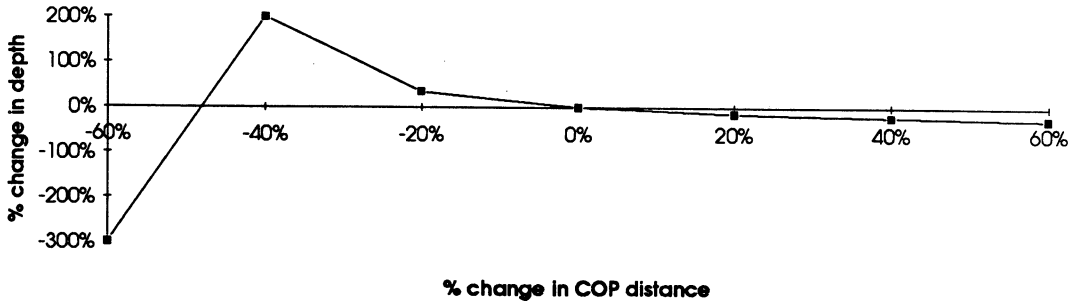
The first test I did was to see what effect wrong projection plane distances would have on the output of the analyzer. What I found was that a change in the projection plane parameter only affects depth, and its affects it linearly. The analyzer program will be more sensitive to depth differences if the projection plane distance is overestimated. The x and y position and size of the object remains the same.

Plane distance	length of square side	depth	% change in plane dist.	% change in length	% change in depth
10	8	20	0%	0%	0%
11	8	22	10%	0%	10%
12	8	24	20%	0%	20%
20	8	40	100%	0%	100%
5	8	10	-50%	0%	-50%

The next test I did was to see what effect wrong center of projection parameters would have on the output.

COP distance	length of square side	depth	% change in COP dist.	% change in length	% change in depth
10	8	20	0%	0%	0%
12	6.857144	17.142859	20%	-14%	-14%
14	6.222222	15.555555	40%	-22%	-22%
16	5.818182	14.545455	60%	-27%	-27%
8	10.666666	26.666666	-20%	33%	33%
6	24	60	-40%	200%	200%
4	16	-40	-60%	100%	-300%

What I found was that changes in the center of projection parameter affect both depth and x and y positioning, non-linearly. In fact, underestimating the distance between the centers of projection is especially dangerous. This is because the lines going through the centers of projection to the projection may intersect behind the projection plane.



With this information, I was then able to make more educated guesses for the parameters to specify for the objects in the mall images. However, I was still unable to get satisfactory results. The arch, which should have been at the same depth as the middle of the railings, always appeared in front of the railings. Additionally, the window frame would sometimes be distorted. This behavior is best explained by my object correlation technique (or lack thereof). Objects in the images were matched by hand, and only to single-pixel accuracy. I probably would have had better results if I had used a computer matching algorithm and if I had matched them to sub-pixel accuracy.

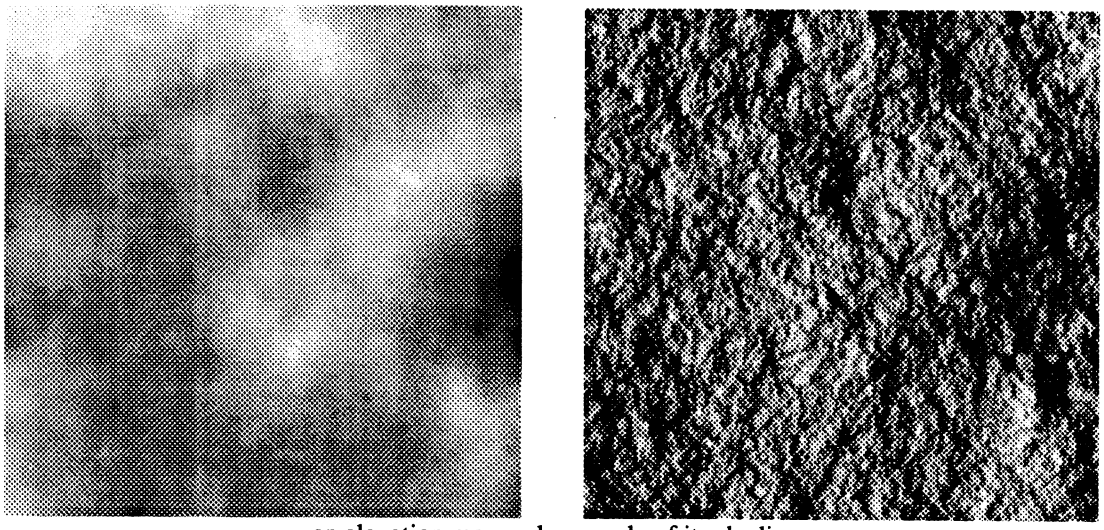
In short, I have found that there are many drawbacks to using stereo methods to model 3D objects from photos. The first is that small errors in input parameters may lead to large errors in output. The second is that matching objects in images is difficult, and it is often necessary to match the objects to sub-pixel accuracy.

7. Shape from Shading

Because of the limitations of stereo methods for determining surfaces, I would have liked to explore other methods of generating shape descriptions from 2D pictures, if I had more time. Most of the methods in use today either solve this problem through special imaging hardware (radar interferometry and videokeratographs), or combine stereo analysis with other methods, such as shape-from-shading. I have already begun some work in shape-from-shading techniques. In order to understand how contour information can be determined from the shading in an image, I wrote a simple program that produces a shading map from an elevation map, given a directional light vector. The shading map program uses the same principle behind Gouraud shading. That is, brightness is determined by the dot product of the directional light vector and the face normal. Face normals are determined by the slope in elevation in the x and y directions. After producing a shading map from a contour map, I attempted to devise a method of generating a contour map from a shading map. However, I was unable to perform the inverse operation because in my calculations, I reached a point where I had two equations with three unknowns.

$\text{sqrt}(\text{normal\_x}^2 + \text{normal\_y}^2 + \text{normal\_z}^2) = 1.0$
$\text{brightness} = k * (\text{normal\_x} * \text{light\_x} + \text{normal\_y} * \text{light\_y} + \text{normal\_z} * \text{light\_z})$

Given more time, I would have liked to have found what oversight I made. One important issue to deal with in shape-from-shading is that an object may not be of uniform color. This is important, because it will alter the way brightness is preceived. It is also important to note that the shape-from-shading technique will only give you information concerning the relative depth of the surface. That is, at best, it will describe the contour of the object, but it will not tell you how far it is from the viewer. What I have found is that stereo methods are often used to determine the general, overall shape and location of an object, while shape-from-shading is used to determine finer surface details in the object.



an elevation map and a sample of its shading map

8. Conclusion

In conclusion, I have discovered that the best methods for generating 3D (or 2.5D) descriptions of objects from 2D representations usually involve special hardware. Most of the methods that don't rely on special instruments aren't very good. Even with the methods that do use special hardware, the algorithms cannot deal with multiple objects and occluded objects (objects obscured by other objects in the foreground). For these cases, more than two images are needed. 3D surface determination is an area of fairly active research in the field of computer vision, and working on this project has motivated me to perhaps investigate this problem more when I continue with my studies in graduate school (I am currently an undergraduate).

CS285

Final Project Report

3D Free Form Modeler and Animator

Zijiang (Z-John) Yang
zyang@cs.berkeley.edu

Abstract

In this project, a 3D interactive modeling system is built. Users are given a set of simple tools to interactively manipulate the location of a set of feature points, which are simultaneously interpolated by triangular Bézier spline surface patches. An attempt has been made to create triangular surface patches of G1 continuity with local control. The user interface issues for 3D direct manipulation has also be studied. The result has shown that users are able to create simple free form objects on this system within a very short amount of time. Other more complicated tools are studied and presented as future work in this report, which would make the object creation more intuitive and user friendly.

1. Introduction

From the very beginning of CAD history, 3D free form design has been an extremely tedious task. To create meaningful smooth 3D shapes with proper mathematical representation, designers have to familiarize themselves with a set of very complex systems. Relying on knowledge of spline surface representation, they first have to turn their desired shapes into a set knots and coordinates of control vertices. After a long period of time of intensive calculation and rendering, the computer finally brings up an image of some objects, which could be very different from the desired shapes. After twiddling around with the control vertices and knots for sometime, the designers may find sometimes the objects become closer to what they want, and sometimes farther apart. The designers inevitably have to compromise for something else, or give up using smooth surface modeling and settle for simple polygonal objects.

This reminds me of the old punch-card days, and things must be made better.

Several companies have incorporated spline based shape modeling into their drawing and modeling software. Attempts have been made to help the designer directly manipulate the shape without dealing with its knots and control vertices. While the 2D spline curve drawing tools have been widely accepted, the 3D spline surface modeling tools haven't gotten very far.

One of the reasons is the fact that 2D drawing is a lot more straight forward than 3D drawing. People are used to 2D drawings and 2 dimensional thinking, but rather quite awkward with 3D drawings and 3 dimensional thinking. Even though we live in a 3D world, information we see are mainly in a 2D form.

Another reason is that computer display is mainly a 2D device. Shapes projected from 3D to 2D will not only lose one dimension of information, but also become clustered and hard to visualize. Even though 3D input devices have been introduced quite a few, and virtual reality is looming on its horizon, complex devices do not guarantee a simpler solution to the problem.

Finally, as the parabolic surfaces and ruled surfaces have been used in CAD for designing simple smooth objects, spline based surfaces are the only form of mathematical representation used for free form objects, which unfortunately do not correspond with computer data structure and rendering hardware naturally. In general, it is non-intuitive to generate the set of knots and control vertices which will represent a desired shape, it is often hard to map their parametric space into simple data structure, and it is time-consuming to render the resulting spline surfaces.

As the 2D modeling and drawing tools have been well developed and have come to certain standards, 3D modeling tools have a long way to go.

In this project, I have built a simple prototype system, which allows the user to input feature points of an object in 3D, and be able to manipulate them interactively until the desired result is achieved.

To build such a system first the difficulties of the spline representation mentioned above have to be overcome. This is done by properly using triangular Bézier spline

representation, plus winged edged data structure and a fast subdivision rendering algorithm developed by the author.

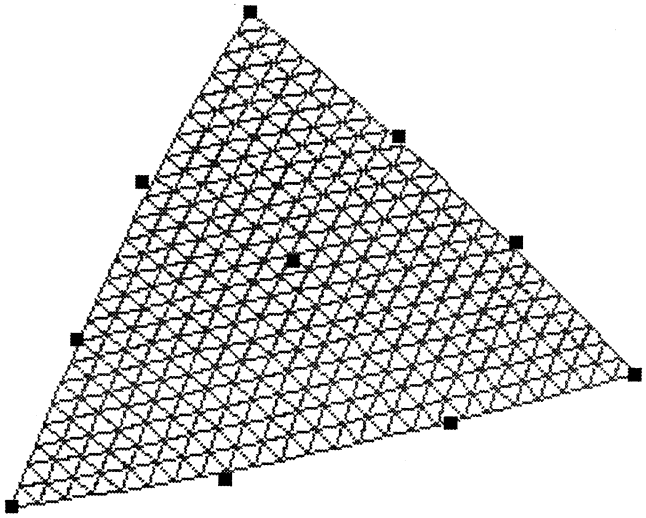
In the following section, I will first discuss the techniques I used to create and render the cubic triangular Bézier surface patches over a set of triangulation, then discuss the user interface issues related to the 3D modeling system.

2. Technical Description

2.1. Rectangular Bézier Patch Vs Triangular Bézier Patch

A rectangular Bézier patch, which has a rectangular parametric domain, is easy to represent and render, but can not be used to define surfaces of arbitrary genus. On the contrary, a triangular Bézier patch, which has a triangular parametric domain, is harder to represent and render, but can be used to model surfaces of arbitrary genus, since any object has an triangulation approximation.

2.2. Creation and Rendering



A cubic triangular Bézier patch is defined by 10 control points over a triangular domain. The underline surface interpolates the three corner control points, but does not go through the other control points in general [Farin1,3, Hoschek].

To render such a patch, first we need to approximate the surface by a polygon mesh. Direct evaluation at a set of parametric values can achieve this task. However, it is computationally expensive and cumbersome in terms of coding due to the triangular shaped parametric domain. The de Casteljau subdivision algorithm has shown much more efficiency for rendering rectangular Bézier patches, unfortunately, when this algorithm applied to triangular patches, it only splits the center of the patch without subdividing the boundaries. Even though a proof of the existence of other subdivision algorithms has been shown in [Goldman], none of these algorithms have been carried out in the recent literature.

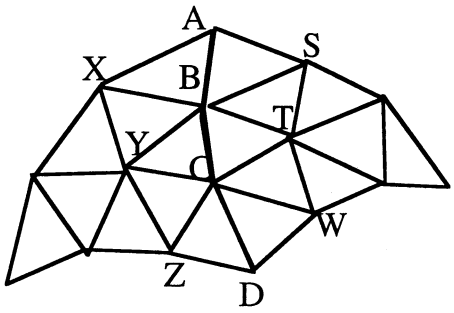
In order to achieve the interactive speed, I have developed a subdivision algorithm which cuts each boundary in two, and splits the entire patch into four sub-patches. The idea behind this algorithm is simple, however the math is messy, thus it is omitted.

2.3. Interpolation

Any object can be approximated by a triangulation over a set of vertices. Once a triangulation is given, our goal is to create smooth surface patches that interpolate this triangulation. Knowing that a triangular Bézier patch interpolates its three corner control points, we can just use the vertices of the triangulation as the corner control points and leave the rest of the control points to be set later. The extra control points give us more degree of freedom which can be used to ensure smoothness between patches.

2.4. Smoothness

A large set of literature has been devoted to pursuing G1 and C1 condition between two adjacent triangular Bézier patches.



The control polygons of two adjacent patches can be shown as in the following figure. In order to achieve G1 continuity, the following condition has to be all satisfied by these two patches [Liu1,2]:

- G0 condition: they have to share the same common control points, i.e. *A, B, C, D.*
- Coplanar condition: control points *X, B, A, S* have to be coplanar, same with *Y, B, C, T*, and *Z, D, W, C.*
- Ratio condition: $a\,\boldsymbol{XB} + b\,\boldsymbol{SB} = \boldsymbol{AB}$, $a\,\boldsymbol{YC} + b\,\boldsymbol{TC} = \boldsymbol{BC}$, and $a\,\boldsymbol{ZD} + b\,\boldsymbol{WD} = \boldsymbol{CD}$ have to be simultaneously satisfied for some real number a and b.

These conditions become more complicated when ensuring G1 continuity condition among multiple patches. Global near G1 continuity has been achieved through numerical methods [Farin2] and optimization schemes [Moreton, Welch]. Various attempts have been made in searching for alternative triangular surface representations for ensuring global G1 continuity [Gregory, Barnhill].

2.5. Locality

Interactive modeling requires local control over the shape of objects, since a global fitting scheme would inevitably slow down the process when creating complicated objects and would not be suitable for real-time interactive rendering.

When one vertex is modified by the user, the patches which are sharing this vertex need to be modified to maintain interpolation condition. This would imply the control points of these patches need to be regenerated, except the corner control points. However, because of the G1 condition give in the previous section, this would also imply that the modification will ripple through other patches which do not contain the modified vertex.

To achieve locality, the generation of control points need to depend only on local properties of the triangulation. That is the vertex location, face normal, vertex normal, and edge normal of a certain triangle in the triangulation, since these features are changed locally when a vertex is modified. This basically means that locality does not go along with G1 continuity for cubic triangular Bézier surfaces.

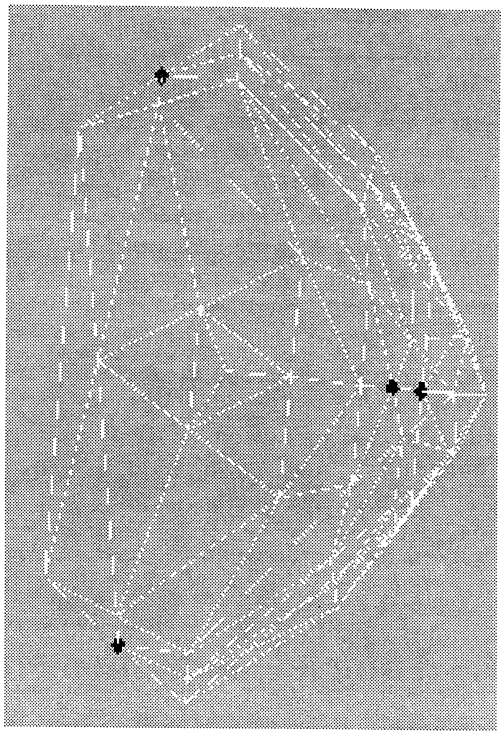
2.6. Previous Work

Since the explicit condition for G1 continuity among multiple patches is hard to pursue, there has not been any description on how to arrange control points to automatically

maintain a global G1 continuous surface. Research work has been devoted to pursuing G1 continuity while assuming boundary control points are fixed. With Gregory patches [Gregory], G1 continuity has been achieved, with the sacrifice of explicit surface representation. With Clough-Tocher [Farin2] approach, near G1 continuous surfaces are constructed locally and numerically, but G1 condition is not guaranteed. Moreton and Sequin [Moreton] have incorporated global optimization into the construction, which produce results that have smooth looking surfaces, but are rather computationally expensive. In [Shirman], G1 continuous quintic surface is constructed over a predefined cubic curve mesh.

Triangular B-splines are used in [Greiner] and [Fong], where the C2 continuous surfaces were built without interpolation of the vertices of the triangulation.

Other triangular surfaces have been used to construct surfaces of arbitrary topology, see [Welch].



2.7. Construction

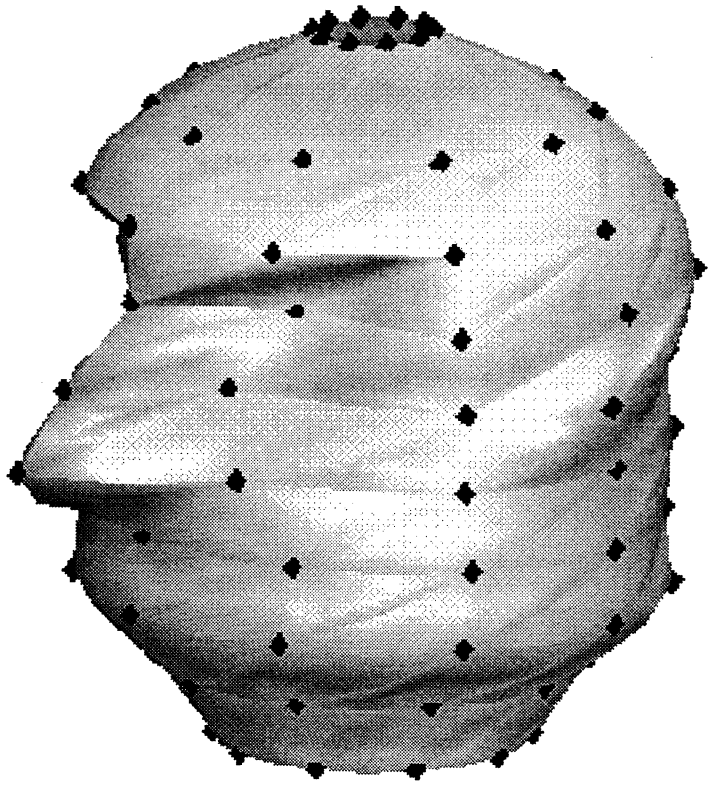
To distinguish from the previous implementations, this system only allows the user to control the position of the vertices of the triangulation, or the corner control points of the Bézier patches. To construct the cubic triangular Bézier surface over this triangulation, two more control points need to be generated along each edge, and one more in the center of each patch. As it was shown in the previous section, to generate G1 continuous surface, the three G1 continuity conditions have to be all satisfied, even with the extra degrees of freedom from the non-corner control points, there is no global construction which guarantees G1 continuity when using cubic triangular Bézier patches. In fact, in general cases, only the first G0 condition can be satisfied. The second coplanar condition would fail when the triangulation does not form a convex polyhedron.

To generate more degrees of freedom, one method is to raise the degree of the Bézier spline basis function to quartic or quintic. Another method is to add one more auxiliary point in the center of each triangle, and split this triangle into three sub-triangles. The second method has been implemented in the current system, in which three sub-patches are used to model each triangle in the triangulation.

We also introduce the idea of a new type of *near G1* continuity: when the triangular patches are joined together, satisfying only the first two G1 continuity conditions, i.e. the G0 condition and coplanar condition, without satisfying the ratio condition, we call the constructed surface to be near G1 continuous.

It can be proven that given any triangulation, a near G1 surface can be constructed with local control when using the second method. In fact the three sub-patches can be joined with true G1 continuity. However, in order to achieve this, it involves solving a system of

27 equations of degree two, which would tremendously slow down the construction. Thus the construction used in the current system only guarantees near G1 continuity between all sub-patches.



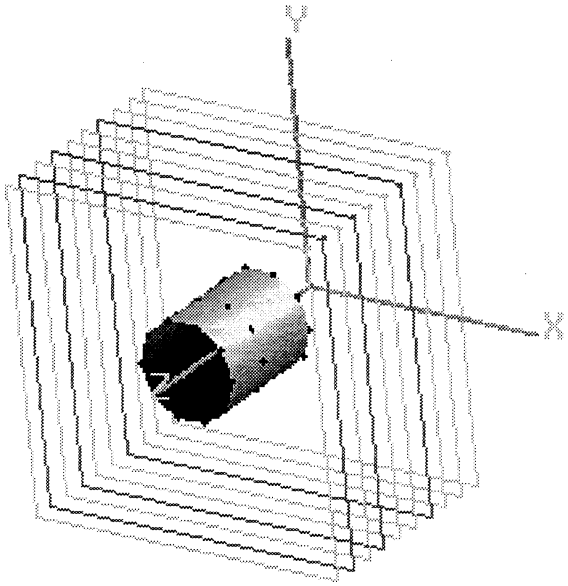
3. Conclusion

A prototype of 3D interactive free form modeling system has been built. Using this system, users can create meaningful 3D free form objects within a very short period of time, and can modify the objects interactively. This system can also be used directly with medical imaging data such as MRI. It can also be used to generate objects for animation. Once the G1 continuity is achieved, CAD community may use this system for any free form design.

4. Tutorial and Instructions

4.1. How to Input Feature Points

The feature points are the set of points defining the shape of an object, they eventually become the vertices defining the triangulation approximation of the object. The resulting spline surface representing this object will interpolate these points. The ordering of these points will also characterize the topology of the object. The more complicated the shape is, the more the feature points need to be used.



It is often easy to think of feature points in 2D. They can be simply thought as the points generated by sampling the contour of a 2D shape.

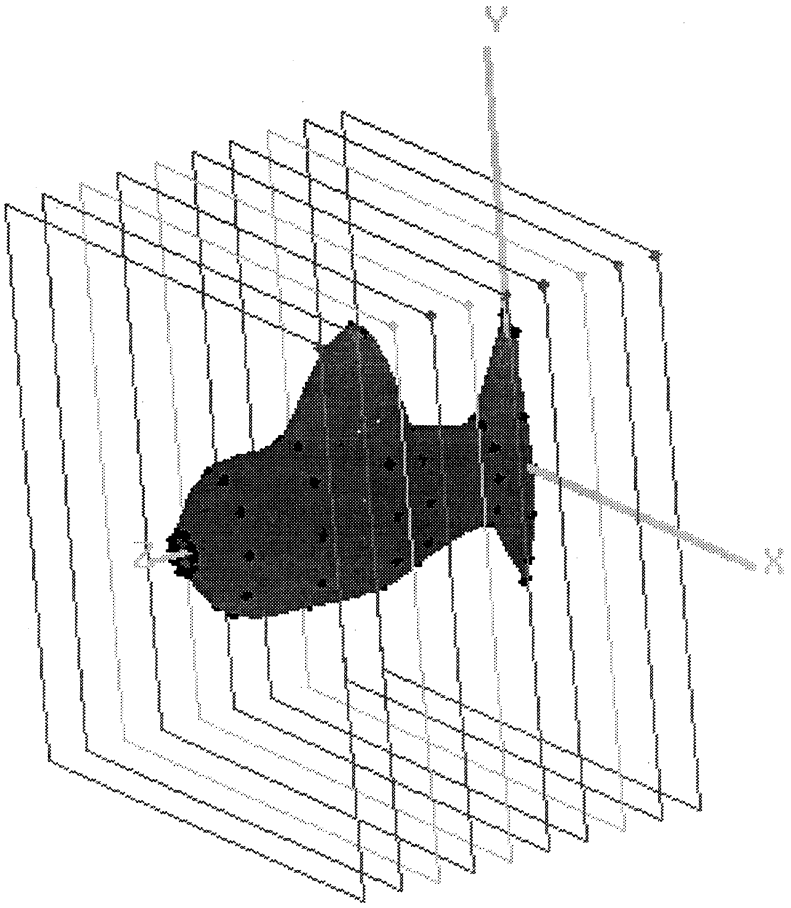
In 3D, to generate the set of feature points which can represent a 3D object can be quite cumbersome. The ordering of these points becomes more complicated, and the sampling of these points can not be done arbitrarily.

Based on the fact that people are more comfortable with the ordering and distance between 2D points, I have decided to use the idea of lofting.

Several parallel lofting planes are first given to the user, shown in the color yellow. Each plane has a handle which is represented by a small polyhedron on its corner. With crystal ball rotation tool, the user can select anyone of these planes by clicking on its handle. Once a plane is selected, the user can then enter a 2D drawing mode and start creating contour points on this plane. The contour points are also represented by small polyhedra in the color of blue, which are also the feature points of the expected shape within this plane. These points can be moved around individually or as a group. After creating these contour points, the user can return to the 3D drawing mode by clicking on the handle of the plane again.

4.2. How to Modify the Shape

The lofting planes with contour points are highlighted as cyan. The user can modify these contour points by selecting these planes again or add other contours by selecting some other planes.



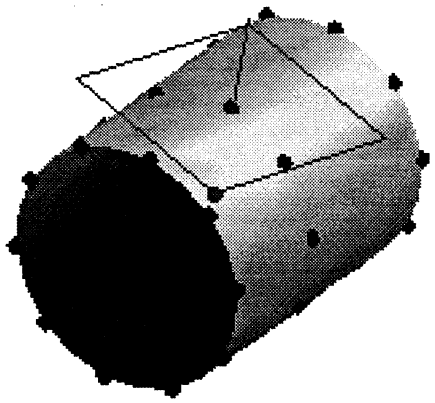
Once more than two lofting planes have been selected and two contours are defined, the system instantaneously wraps a smooth surface around these contours.

The user can then look at the created surface from any angle by rotating it or zooming in and out. A better view can be achieved by turning off the displaying of the lofting planes and axes.

Usually further modification will be required. The entire object can be resized as a whole, as well as each contour can be resized

and modified individually.

4.3. 3D Direct Editing

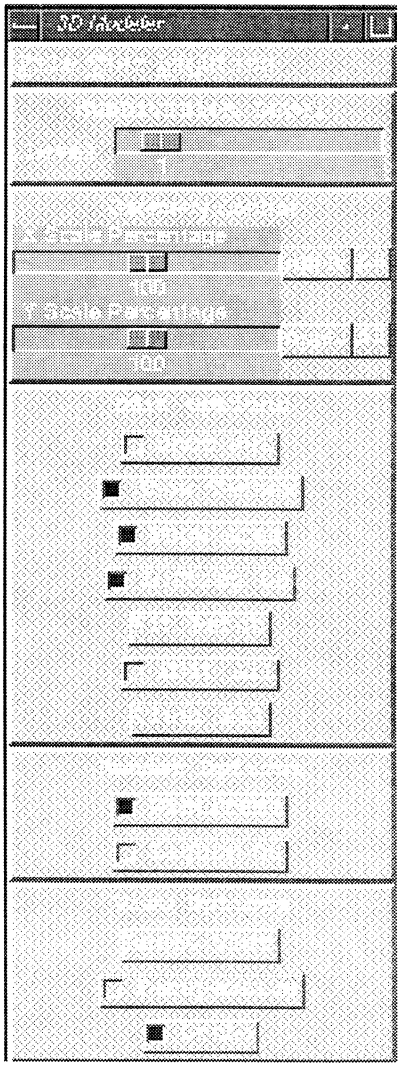


It is often desirable to modify the 3D object without going to 2D drawing mode. Because the relative 2D cursor motion does not provide enough information to define a 3D motion, special tools have to be created to define a 3D translation.

Once a feature point is selected by clicking the left mouse button on top of it, it is highlighted as red. In the same time, a small tangent plane outlined as magenta is drawn, with a normal vector shown in its center. A 3D translation motion can then be decomposed to the motion within this tangent plane and the motion along the normal vector. Since the geometric meaning of the tangent plane and normal vector is clear and relative to the shape of the object, the user can easily accomplish the desired modification.

Notice that the tangent plane also has a handle. This is to enable the user to be able to rotate the tangent plane and choose an arbitrary plane of translation.

4.4. The control panel



The control panel has the following control options:

- **Subdivision Level** slider allows the user to adjust how fine the surface patch is subdivided and rendered.
- **X Scale** Slider controls the percentage that the x coordinates of the features points are scaled. It can be applied to the current selected contour or to all of the contours by pressing the **Apply** button or **All** button respectively.
- **Y Scale** controls the y coordinates scale factor in the same fashion.
- **Show Net** button allows the user to hide or to display the control polygons.
- **Show Vertex** button toggles between displaying and hiding control points modes.
- **Show Axes** gives the user the option not to display the X-Y-Z axes.
- **Lofting On** switches between the displaying lofting plane mode to hidden mode.
- **Fit Patch** updates the surface patch once the control points have been modified.
- When **Edit All** button is selected, the modification is done simultaneously to all the feature points in the selected contour.
- **Clear All** erases the current object.

- **Wire Frame** button toggles between filled polygon mode and wire frame rendering mode.
- **Lights On** button turns on Phong Shading mode.
- **Reset View** allows the user to clear all the 3D rotations of the object.
- **Perspective** button switches the viewing mode to perspective projection mode.
- **Ortho** button turns the view into orthogonal projection mode.

4.5. Other Features

When the 3D object can be arbitrarily rotated and zoomed in and out, the feature points and handles of lofting planes often become clustered, and vary in size. To allow accurate selection, the size of the bounding boxes around this points are adjusted accordingly.

Both wire frame rendering and lighting mode rendering are implemented at interactive frame rate.

4.5. Animation

After designing the object, creating an animation of it becomes a matter of creating frames of the object with slightly modified feature points.

5. Future Work

Creating the shape and modifying the shape by editing individual feature points is still a non-intuitive tedious task. The future of these types of modeling tools lies on the ability of editing a group of points in a user easy understandable fashion, such as to flatten or sharpen the corners, to pinch out a cone, or to drill a hole. It is also a has to have feature to position the feature points as functions of time, thus the animation can be done by simply evaluating the functions at different instance of time.

The G1 continuous construction with local control is still a challenge. A variety of construction scheme still need to be studied, and they may then be improved or combined into new algorithms which fulfill the goal.

References

[Barnhill], R.E. Barnhill and G. Farin, C1 quintic interpolation over triangles: Two explicit Representations, International Journal for Numerical Methods in Engineering 17, 1763-1778.

[Farin1], G. Farin, A construction for the visual C1 continuity of polynomial surface Patches, Computer Graphics and Image Processing, 20, 272-282.

[Farin2], G. Farin, Smooth interpolation to scattered 3D data, Surfaces in Computer Aided Geometric Design, Robert, E. Barnhill, Wolfgang Boehm, 1983, 43-63.

[Farin3], G. Farin, Triangular Bernstein-Bézier patches, Computer Aided Geometric Design 3, 1986, 83-127.

[Fong], Philip Fong, An implementation of triangular B-spline surfaces over arbitrary triangulations, Computer Aided Geometric Design, 10, 1993, 267-275.

[Gregory], J. Gregory, C1 rectangular and Non-rectangular surfaces patches, Surfaces in Computer Aided Geometric Design, Robert, E. Barnhill, Wolfgang Boehm, 1983, 25-34

[Goldman], R. Goldman, Subdivision algorithms for Bézier triangles, CAD 15, 159-166.

[Greiner], Gunther Greiner and Hans-Peter Seidel, Modeling with triangular B-splines, IEEE Computer Graphics and Applications, March 1994, 56-60.

[Hoschek], J. Hoschek and D. Lasser, Fundamentals of Computer Aided Geometric Design, 1993, AK Peters, Ltd., 287-311.

[Liu1], D. Liu and J. Hoschek, GC1 continuity conditions between adjacent rectangular and triangular Bézier surface patches, Computer-Aided Design, 21, 1989, 194-200.

[Liu1], D. Liu, GC1 continuity conditions between two adjacent rational Bézier surface patches, Computer Aided Geometric Design, 7, 1990, 151-163.

[Moreton], Henry P. Moreton, and Carlo Sequin, Functional minimization for fair surface design, Computer Graphics Proceedings, 26(2), SIGGRAPH, July 1992.

[Piper], Visually smooth interpolation with triangular Bézier patches, Geometric Modeling, Algorithms and New Trends, Gerald E. Farin, 221-233.

[Shirman], Leon A. Shirman and Carlo H. Sequin, Local surface interpolation with Bézier patches, Computer Aided Geometric Design 4, 1987, 279-295.

[Welch], W. Welch and A. Witkin, Free-form shape design using triangulated surfaces, Computer Graphics Proceedings, SIGGRAPH 1994, 247-255.

Evolving Architectural Forms

Michael Schiff
Computer Science Division — EECS
University of California
Berkeley, CA 94720

December 7, 1994

Abstract

Arch-evolve is a user-driven evolutionary graphic system for creating models of architectural structures. In this report I describe the system's interface, design, and limitations.

1 Introduction

Evolutionary systems have been used in a variety of ways in computer graphics, from Sims' work with bitmaps generated by symbolic expressions[1] and Todd and Latham's work with pseudo-organic forms[3], to Sims' later work with evolving creatures[2]. Systems based on user-driven evolution, in particular, can effectively allow a user to search through a large space of graphic models.

Arch-evolve is a user-driven evolutionary graphic system for models of architectural structures. In this rest of this report, I discuss the design of the system and some of its limitations.

2 An evolutionary system: user interface

Arch-evolve has a simple user interface. A user is presented with nine graphic models in a single window. He or she can use the mouse to rotate these models as wireframes (using a crystal-ball interface), zoom in and out, or see a higher-resolution view of the model, rendered with a more sophisticated algorithm (Phong shading as opposed to flat shading). When the user decides on the model they like best, they select it.

Initially, the nine models the user is presented with are chosen randomly from the space of possible forms the system can create. After the user makes a choice, however, the system

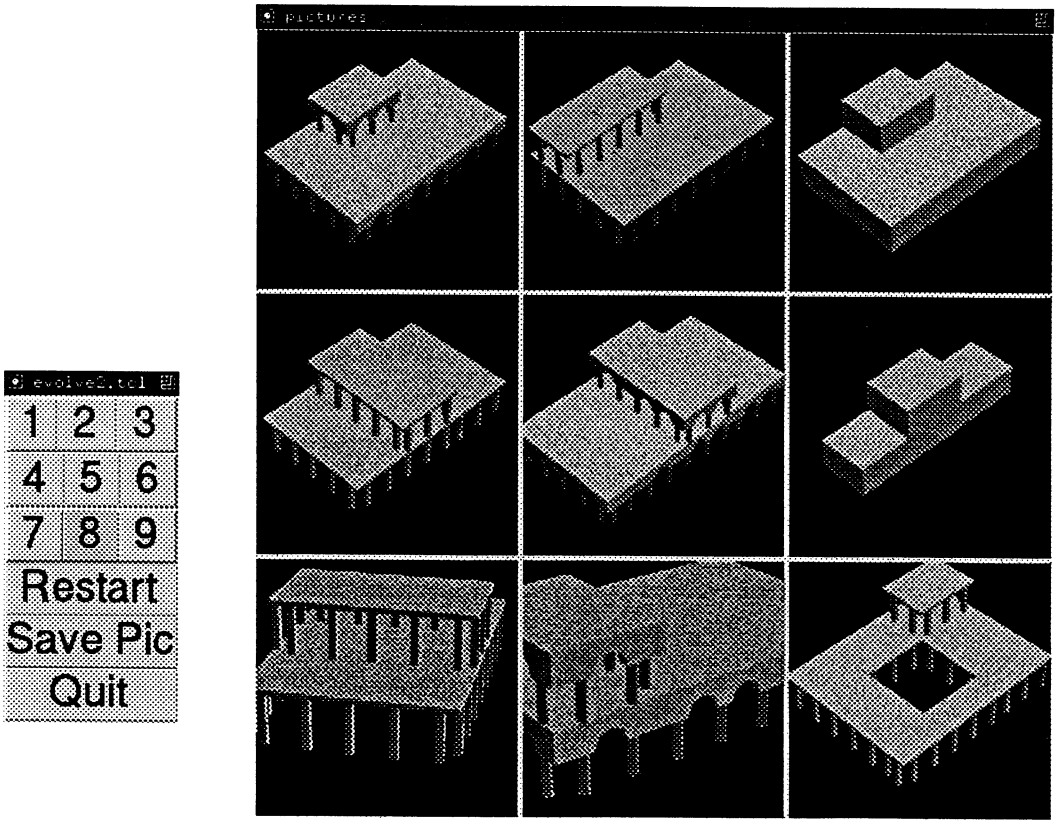


Figure 1: User interface

preserves that model and generates eight new models based on "mutating" that it — i.e. choosing random points near it in the space of possible models. This select—mutate—generate loop provides the user with a way of searching through the space of possible models. By always choosing the model they like best a user should eventually find on a model that satisfies them in some way. Figure 1 shows a screen dump of the system.

The system also allows the user to generate 9 random forms ("restart") or to save a higher-resolution rendering of the model.

3 The space of possible models : structure and style

Although the framework for a system like this is rather simple, the implementation required a number of significant design decisions. The most difficult ones involved defining the space of possible forms. First, it was important that the space include a wide variety of forms, and that a significant percentage of the forms be interesting. It would be easy to

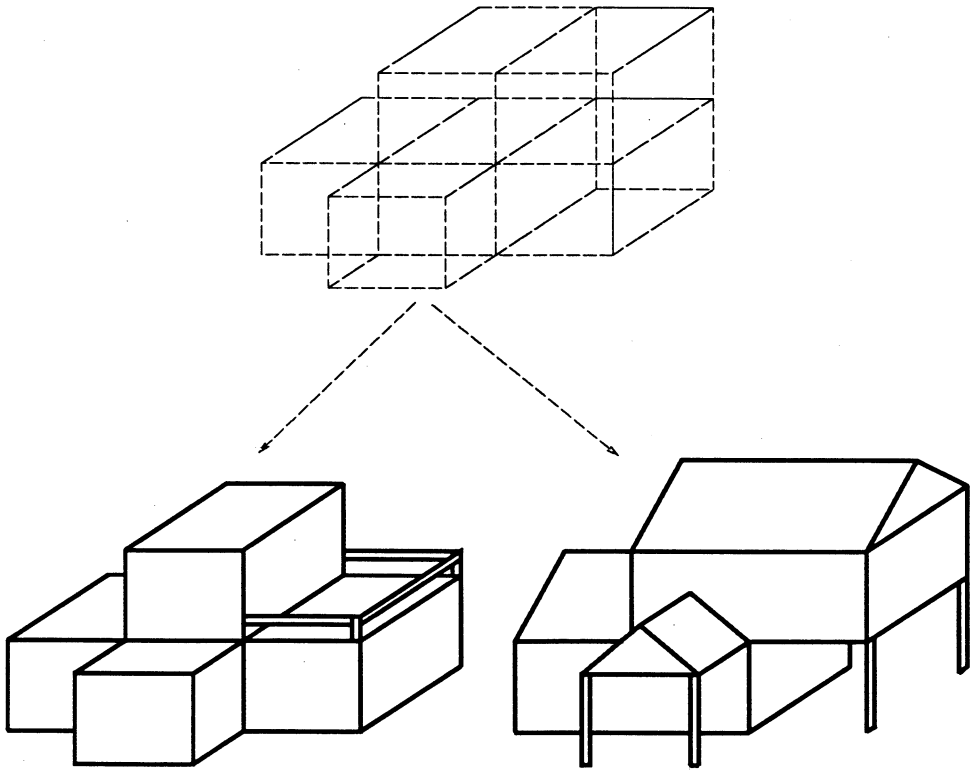


Figure 2: Structure and Style

build a system that generates a model of a house, only varying the parameters for the size and locations of the windows, or to build a system that varies the positions and orientations of an arbitrary group of polygons, occasionally generating building-like structures. Neither system would be very satisfying, though.

Furthermore, to make a user-driven evolutionary system work well, it is important that there be some degree of continuity in the space of possible designs. That is, if mutating a form always changed it to something with no recognizable similarity to the original, it would be impossible for a user to make a rational choice. The general method I used to guarantee continuity was to make the different parameters to the system as independent as possible. That is, by designing the parameters so that changing the value of one didn't change the possible ranges of values of all the others, I tried to make it possible for a model to change significantly without changing completely.

The first step in my parametrization of the space of models was to separate style from structure. Structure refers to the space that the building occupies or otherwise delimits. Style refers to the the way that space is realized. Figure 2 illustrates this distinction, showing how one structure can lead to two quite different forms, if different styles are used.

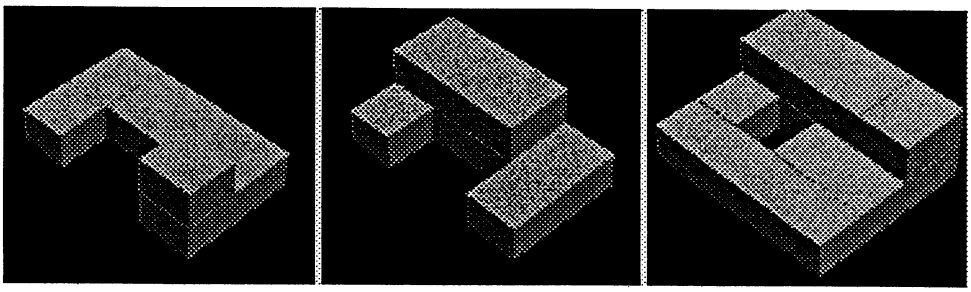


Figure 3: Three possible structures

3.1 Structure

The structures of the forms created by *Arch-evolve* are based on a three-dimensional grid, with parts removed. More specifically, the structures are determined by three sets of parameters (or genes):

- Parameters for producing a two dimension grid of $m \times n$ blocks with varying lengths and widths.
- Parameters specifying the location and size of a rectangular "cutout" region, where blocks are removed from the grid to generate courtyards or L- or U- shaped buildings.
- Parameters specifying the location and size of a rectangular regions where blocks remaining in the grid are stacked with an additional level, to generate a second floor area.

Figure 3 shows three representative structures that can be specified by these genes.

3.2 Style

The style parameters used by *Arch-evolve* fall into two categories: "block" parameters which determine the style of individual blocks in the grid, and global parameters, which affect the whole form. Several sets of block parameters and one set of global parameters are generated for each model.

The block parameters used by the system are:

- The type of wall delineating the block — solid or a row of arches/columns.
- The placement of windows and doors in solid walls
- The shape and size of various combinations of arches and columns

- The colors (or textures) of particular parts.

The global parameters, which handle pieces of the form that span multiple grid blocks, are:

- The type of roof — flat, symmetric wedge, asymmetric wedge, pyramidal — and the other parameters that determine it (angle, overhang).
- The location of doors

Of course, this list of parameters could easily be extended or changed.

3.3 Combining Structure and Style

To combine structure and style, each block in the structural model is assigned (randomly, subject to mutation) to one of the sets of style genes. The polygons that are generated for any given block in a form actually depend on both the style genes and the context of the block. For example, a single block at the end of a long row of single blocks will generally have three walls drawn in a manner determined by its style genes. A block in the middle of a single row of blocks will only have two walls drawn.

At the moment, the only context taken into account by any of the styles are the presence or absence of neighboring blocks, but it would be possible to take into account the styles of neighboring blocks as well.

4 Mutation

As with parametrizing the space of possible models, there is no theoretically correct solution to the question of how to mutate the sets of parameters that make up the "genome" for a model. However, two features are desirable. First, given some model the mutation algorithm should be capable of producing both highly similar models and extremely different models, as well as models in between. This allows a user to fine-tune a model they already like, or to choose a model completely different from the ones they have seen. This is necessary to reasonably search the space of possible forms. It is also valuable to be able to regulate the mix of these similar and different models, with some sort of user-set mutation rate, so the user could control the search to meet their needs.

In order to allow for both of these features, the mutation routines in *Arch-evolve* decide which mutations to make based on the generation of a random number r between 0 to 1. Low numbers correspond to small-mutations (tweaking the parameter values), high numbers correspond to major ones (completely re-randomizing some parameters). By raising r to various powers inversely related to the mutation-rate, the system can control the relative frequency of major and minor mutations.

Another feature of the mutation routines in *Arch-evolve* is that they sometimes set groups of parameters together, using special routines. This allows particular desirable points in the space of possible models to occur with greater frequency than pure chance would dictate. For example, although several parameters determine the location and size of a "cutout" from the floorplan, and they are usually set randomly, occasionally a mutation will set the courtyard to the middle of the grid, occupying all but one block in each direction.

5 Implementation

Arch-evolve is mostly written in C, incorporating TCL/Tk, the tkPhoto Widget (used to display bitmaps), and the SIPP rendering library (a freely available scanline rendering package). The user interface is written in TCL, for flexibility. It should compilable for any machine running X windows.

In general, I tried to make the code as modular as possible. The TCL interface could be used for any sort of 3D evolutionary graphic system with almost no changes. Also, the C code is modular, so that to make a system that evolved models of things other than buildings would require replacing only a few functions (and writing new subroutines called by these functions, of course).

6 Conclusions

Despite its limitations, *Arch-evolve* demonstrates the feasibility and potential of a system for evolving architectural models. Although in its current form, it doesn't really generate a large enough variety of models, or detailed enough models, it seems clear that it could be improved and could become an interesting system.

It is certainly possible to add in new parameters (genes) to the existing system, but I think that if I were to start over again, with more time, I would redesign the space of forms. Although I think the idea of separating structure from style was a good one, it is not clear that it makes sense to decompose a model into independent blocks, and certainly not blocks in a grid. Also, it is not clear that style and structure can be separated completely.

Designing the space of forms is a non-trivial problem, however. I spent a lot of time thinking about about how to do this, only to be disappointed with my results. It turned out to be difficult to come up with parameters that generated a wide range of models, but only generated valid models.

Another difficulty I had was in predicting effects of the mutation routines. In using the system at various stages, it quickly became clear that changing the relative frequencies of various types of mutations could significantly affect the subjective experience of using the system. Unfortunately, there doesn't seem to be any way to figure out how to set these relative frequencies, other than through empirical testing.

References

[1] K. Sims, 'Artificial Evolution for Computer Graphics,' *Computer Graphics*, Vol 25, No. 4, July 1991.

[2] K. Sims, 'Evolving Virtual Creatures,' *Computer Graphics*, Vol 28, No. 4, July 1994.

[3] S. Todd and W. Latham, *Evolutionary Art and Computers* San Diego: Academic Press, 1992.

NAME

arch-evolve - Architectural structure evolver

SYNOPSIS

arch-evolve [no command–line options implemented]

DESCRIPTION

Arch-evolve is a user–driven evolutionary graphics system for constructing models of architectural forms. It is used by examining and selecting among nine models. The next generation of models will be based on the chosen one, with that model in the upper left.

MOUSE COMMANDS

To rotate any of the nine models displayed, drag with the middle mouse button in the image of that model. To zoom in or out, drag with the right mouse button. To display a larger, Phong–shaded image of any model, click with the left mouse button on that image. (Click any button on the larger image to dismiss it).

To select a model which will be mutated, use the control pad displayed in a separate window from the models. The "Restart" button on the control pad can be used to generate 9 completely new models. The "Save" button will save a ppm file with an image of the model in the upper left.

ENVIRONMENT

DISPLAY

to get the default host and display number.

TK_LIBRARY

refer to TCL/Tk man pages

TCL_LIBRARY

refer to TCL/Tk man pages

AUTHOR

Mike Schiff <schiff@cs.berkeley.edu>

StairMaster: An Interactive Staircase Designer

Course Project Report, CS 285, Fall 1994.

Rick Lewis
Computer Science Division
University of California, Berkeley
rwlewis@cs.berkeley.edu

ABSTRACT

StairMaster is tool that allows users to construct a Unigrafix three-dimensional polyhedral model of a staircase from a two-dimensional floorplan. A window with slider and menu controls is provided to allow the user to interactively modify the staircase in real time. *StairMaster* consists of new program modules integrated with the *Animator* Unigrafix viewer and editor program.

Introduction

The need for better architectural design tools has arisen from the basic fact that, while most designs currently consist of two-dimensional floorplans, actual buildings are three-dimensional entities that can be reasonably approximated with three-dimensional polyhedral models. Widely-used design programs such as AutoCAD are predominantly two-dimensional tools. For a class assignment in Professor Sequin's CS285 course in fall of 1994, a group of colleagues and I wrote a program that accepted a Unigrafix file of co-planar polygons, and gave as output a three-dimensional model of a staircase that resulted from the extrusion of the input floorplan. For a brief summary of the technical issues involved in generation of the steps, walls, railings, and runners, see Appendix A.

As we generated different staircase models, we realized that a major limitation of our program was the complete lack of interactivity. Parameters, such as step thickness and railing height, had to be fixed at run time. The program took on the order of 30-90 seconds to generate the 3D model, and a separate viewing program such as *UGiris* or *animator* had to be run in order to see the finished product. Many, if not most, design decisions are most easily made when the structure being designed is currently in view. For a user who was designing a staircase, the slow cycle of choosing parameters, generating the entire staircase, and running a separate viewer would have been inconvenient and unnecessarily time-consuming.

StairMaster uses the *Animator* [2,3] Unigrafix [1] viewer and editor as a platform for interactive staircase design. Floorplans can be loaded and staircase models generated very quickly from within *StairMaster*. A control window allows incremental and independent changes to the staircase parameters, and the staircase model is updated in real-time in the viewing window to reflect parameter changes.

Overview of StairMaster Program

The *StairMaster* program consists of four windows: The command window allows loading and saving of floorplans and generation of staircases. The viewing window displays the floorplan or staircase. The controls window provides parameter sliders and style choice menus for editing the staircase. Finally, the messages window provides the user with some guidance about what steps to take in designing the staircase.

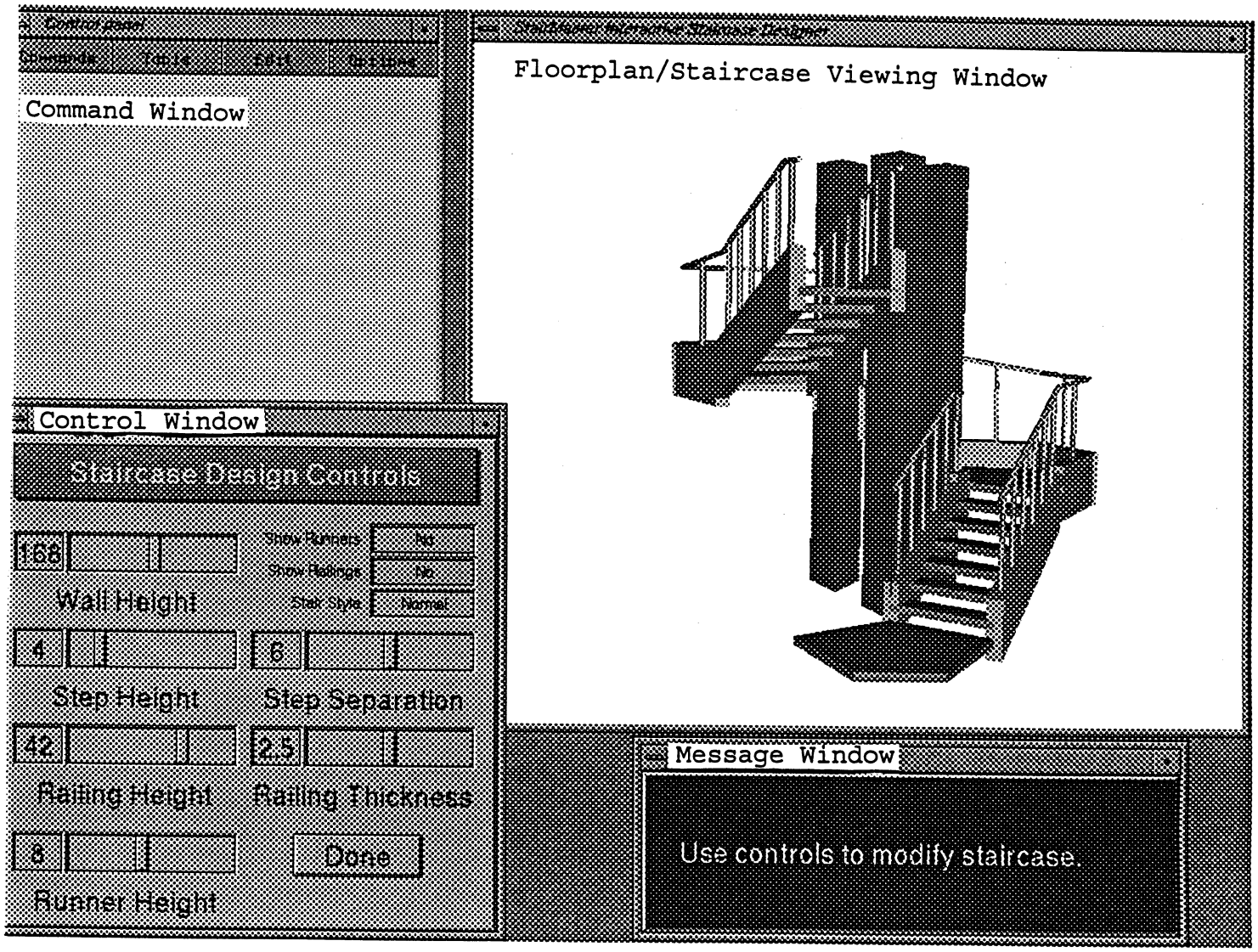
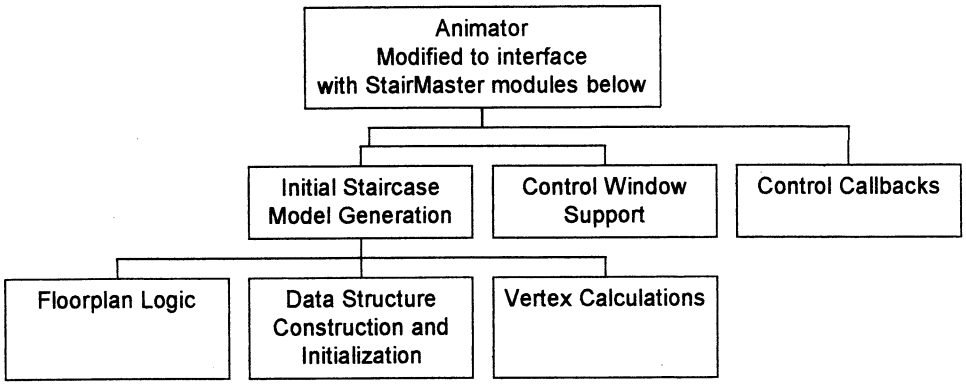


Figure 1
StairMaster in Action

StairMaster Software Components



A modular approach was taken in designing StairMaster. First, the *Animator* was modified to include menu commands that allow the user to load floorplans, specify the ascent of the staircase, generate the staircase, and edit the staircase. These menu commands call upon the subordinate modules shown above to accomplish these tasks.

The initial staircase generation is performed by coordinating three modules:

- Floorplan Logic: Prepares data structures that maintain proper ordering of the 2D step faces, in response to the user's clicks on the lowest and second-to-lowest steps. This ordering is used to control where the staircase starts from (its lowest step) and in which direction it ascends during construction of the staircase.
- Data Structure Construction and Initialization: Creates linked lists and pointers used in building, and later in modifying, the 3D model of the staircase. For example, a linked list of "step_objects" is created which contains, among other things, pointers to the vertices of the top and bottom faces of the 3D polyhedron that comprises that particular step in the 3D model. Similar structures are used for walls, runners, and railings.
- Vertex Calculations: The functions within this module properly position the vertices of the steps, runners, railings, and walls in the 3D model. Very specific functions are provided; for example, there is a function that modifies only the Z coordinate value of the top faces of steps. This is useful when the user changes the "Step Height" slider in the control window.

Two modules support user editing of staircase parameters through a control window, that has sliders and style choices for staircase parameters. Support for the control window was created using the *Forms* user interface toolkit. When sliders are moved, functions in the "Control Callbacks" module of *StairMaster* are called. These callback functions modify the staircase according to the changed parameter. For example, if a user changes the "Step Height" slider, the "StepHeightChange" callback function is called, which calls a function in the "Vertex Coordinates" module to modify z-values of the step vertices.

Technical Challenges

Interpreting a Floorplan

A floorplan consists of a coplanar set of non-overlapping polygon faces.

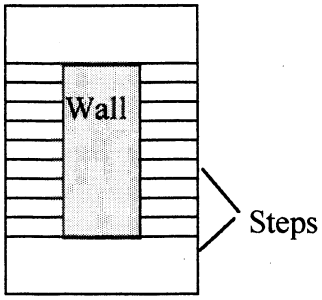


Figure 2
A Floorplan.

Each step polygon may share an edge with at most two other step polygons, so branching stairwell corridors are not allowed. A Unigrafix polygon face that will be extruded into a wall is labeled with the prefix "w" in the floorplan file. Face names for polygons that will be extruded into steps are prefixed with an "s". No special ordering of the faces is required.

It is obvious that the floorplan in figure 2 is ambiguous in that there is no sense of where the lowest step is. Also, it is unclear whether the step ascends in clockwise or counter-clockwise fashion. To remedy this, *StairMaster* provides a "Specify Ascent" feature. The user can click on floorplan faces that correspond to the lowest and second lowest steps. StairMaster correctly orders the rest of the faces so that the constructed staircase will ascend in the desired manner.

To implement this feature, the ordered list of faces is built by testing the current step for adjacency to another face. Therefore, adjacent step faces should share vertices. That is, each step face has an edge where it is adjacent to the next step; for both step faces, the edges should use the same vertices as endpoints (see figure 3). This can be guaranteed by applying the Unigrafix *vmerge* tool to the floorplan. Conveniently, this can be done from within *StairMaster*, thanks to its *Animator* heritage.

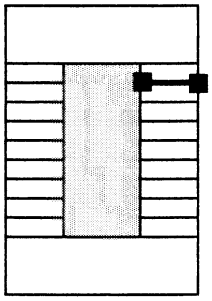


Figure 3
Vertices shared between
adjacent step faces

Data Structures for Fast Updates

Since it is desirable for updates to the staircase model to be reflected in the visual model quickly, the data structures which are modified must be straightforward to traverse, and the coordinate values for the vertices must be reachable for modification after only a small amount of indirection. For example: for each step, *StairMaster* maintains an array of pointers to the coordinate values of the vertices of the top face of that step. So for the vertex calculation routines to change the z-value of a particular step STEP_X, all that is required is:

STEP_X->topVertices[1][2] = (some new value)

Having this capability required that the array of vertex pointers be initialized when the faces are originally constructed when the user first selects "View Staircase". So there exists a one-time overhead cost for providing this capability. This overhead is justifiable by comparing how an update would be performed if *StairMaster* simply maintained a pointer to the Unigrafix face statement for the top face:

cl = contour list for STEP_X->topFace
con = contour from contour list cl
el = edge list from contour con
e = edge from edge list el
v = vertex from edge e
f = pointer to vertex v's coordinates
f[2] = (some new value)

The performance penalty for all of the pointer indirection makes *StairMaster*'s direct pointer method attractive.

The data structures for step and runner objects is shown below also contain delta_x, delta_y, and delta_z floating point values which are related to certain specific staircase parameters. These values are the incremental increase in the x, y, or z coordinate of a vertex in response to a change in the slider value of a parameter. As a simple example, if the "Step Height" slider is changed by 1, the z-value of vertices on the top face of each step also increase by one. So then delta_z_StepHeight is 1. A slightly more complex example is runner width (See figure 4). Say a runner object extends from 0,0 to 10,10 in the xy plane. If the runner width changes, the x and y values of the side faces of the runner also should change. Since we don't wish to recompute the positions for each update, we use the delta values for runner width. If the runner width increases by 1, the left face moves "out" by 1/2, and the right face moves "out" the other way by 1/2. Simple math dictates, then, that delta_x_RunnerWidth is $(1/2)/\sqrt{2}$. The delta values can be precomputed because their values rely only on the floorplan.

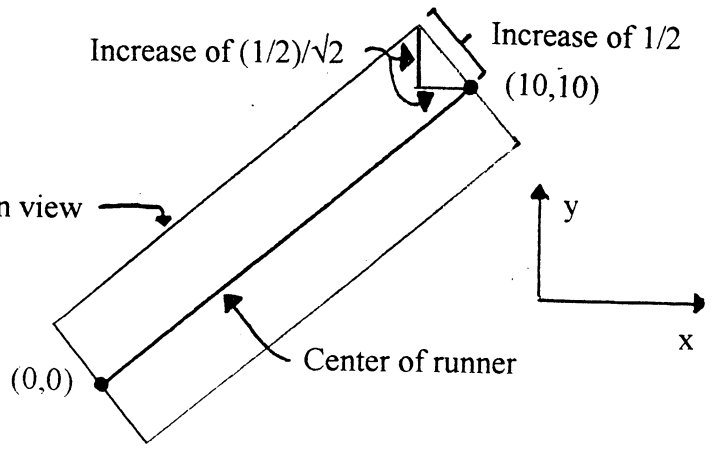


Figure 4
Runner Width

Comments

This project has been very enjoyable for me. I have learned an interesting lesson about reusing previously written code. While I did spend an annoyingly large amount of time on learning the *animator* and integrating changes into it, I clearly was able to complete a more robust staircase generator because the *animator* provided a complete viewer. So in hindsight, what seemed like unnecessary overhead in learning the *animator* probably would pale in comparison to the amount of time it would have taken to write my own viewer.

If I had it to do over again, I might not use the *forms* user interface package; perhaps Tcl or Motif would have been better. *Forms* has proven restrictive and buggy in its event handling. The result is that the staircase currently can't be moved or rotated while the *forms*-based control window is open.

If I had more time, I would like to add a simple floorplan generator and editor to the package. Also, I would add support for branching staircases.

Conclusion

This report has described the *stairmaster* staircase generator and editor for UNIGRAFIX. Staircases can be generated from a floorplan, viewed, and edited all within the same program. *stairmaster* uses the *animator* viewer and editor as a platform.

For this project, I integrated modules into the *animator* that perform ordering of floorplan steps, extrusion of steps and walls into 3D polyhedra, and generation of railings and runners. I also devised and implemented data structures that allow incremental modifications and instantaneous update of the staircase in the viewing window. A user interface is provided, complete with sliders and menus, that permit the modifications to be performed conveniently. A non-invasive approach was taken in integrating these new modules into the *animator* such that the functionality of the original *animator* is preserved in *stairmaster*.

References

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

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

[3] Funkhouser, T. *An Interactive UNIGRAFIX editor*, Unpublished, 1991.

A Brief StairMaster Tutorial

Here is a step-by-step instructions for generating a sample staircase with *stairmaster*.

1) From the UNIX prompt, type **stairmaster**.
The command window, viewing window, and message window will appear.

2) Load a floorplan. Select "Load Floorplan" from the "commands" menu. Type the name of the floorplan in the green box. Press accept. The floorplan will appear in the viewing window.

3) Specify how the staircase ascends. Select "Specify Ascent" from the "commands" menu. The message window will instruct the user to click on the lowest step in the floorplan, and then the second-lowest step in the floorplan. Do so.

4) Generate the steps and walls in 3D. Select "View Staircase" from the "commands" menu. The steps and walls will appear in the viewing window.

5) Edit the staircase. Select "Edit Staircase" from the "commands" menu. The staircase design controls window will appear. In this window are a number of sliders and choice boxes. The following steps needn't be performed in order.

6) Add Railings. Select "Yes" in the "Show Railings" choice box. Railings appear.

7) Add Runners. Select "Yes" in the "Show Runners" choice box. Runners appear.

8) Modify the wall height. Move the "Wall Height" slider. Notice how the model is automatically scaled so that it remains in full view at all times.

9) Modify step separation. Move the "Step Separation" slider. Notice how the steps, runners, and railings all respond to the changes instantaneously.

10) Use the other sliders to make further modifications.
Click the "Done" button when done.

11) Select "Save File" from the "commands" menu. Enter a filename for your staircase.

At any time, you can view the floorplan using "View Floorplan" under the "commands" menu. Any changes made to the floorplan with editing features under the "edit" menu will be reflected when the staircase is viewed again with "View Staircase" under the "commands" menu.

NAME

stairmaster - interactive Unigrafix staircase generator and viewer

SYNOPSIS

stairmaster

DESCRIPTION

stairmaster is an interactive staircase generator and viewer for Silicon Graphics IRIS 4D workstations. Floorplans can be loaded from the command menu and staircases can be generated from the floorplan. A control window can be opened which allows interactive editing of the staircase parameters.

Floorplans are simple Unigrafix files with co-planar faces representing steps and walls. Adjacent steps must share vertices along the edge that seperates them. Currently, no branching is allowed; each step may have at most two adjacent steps. Faces for steps should be prefixed with 's' and faces for walls should be prefixed with 'w'.

stairmaster uses the "animator" Unigrafix viewer as a platform. All of the original functionality of the animator is retained in stairmaster.

Commands:

Load Floorplan: Reads in the 2D floorplan and displays it.

Specify Ascent: Permits user to click on lowest and second-lowest steps, to indicate the origin and ascent of the staircase.

View Floorplan: Switches to floorplan view at any time.

View Staircase: Builds the 3D staircase model if it hasn't already been built. Displays the staircase in the viewing window.

Edit Staircase: Opens a staircase design control window. The window permits the following modifications in real time:

Show Runners: (Yes/No) Display runners which support steps on outside of run.

Show Railings: (Yes/No) Display railings on entire staircase.

Stair Style: (Normal/Basement) Toggle between normal and basement style steps. Normal style has elevated steps; basement style has steps that extend to the ground level.

Wall Height: Modify the height of all of the walls in the model.

Step Height: Modify the thickness of each step.

Step Separation: Modify the difference in elevation between steps.

Railing Height: Modify the height of the railings, measured from the center of a step.

Railing Thickness: Modify width and height of railings.

Runner Height: Modify the height of the runners.

The "Done" button hides the editing window and allows rotation and translation of the model in the viewing window.

AUTHOR

Rick Lewis
(rwlewis@cs.berkeley.edu)

Appendix A

Brief Summary of Staircase Generation Issues

Steps, Walls, Runners, and Railings
The steps and walls are simple right-rectangular extruded solids. The runners, which are often at an angle, are constructed by computing a vector between the two points of the step edge where the runner is located. This vector is used to locate the eight corner vertices at
proper offsets from the step edge.

Railing and Runner Placement and Orientation
Placement and orientation of the railings corresponds directly to that of the runners; that is, where a support runner is flat (or level), the railing above is also flat, and similarly for slanted runners and railings. So similar algorithms are used for placement of both runners and railings.

Algorithm for Railing/Runner Placement
For each step,
  For each edge in that step's 2D projection (floorplan face)
    if edge shares no vertices with other steps -> Flat railing and runner
    else if edge shares two vertices with another step -> No railing or runner
    else if edge shares one vertex with step above and one with step below -> Slanted
        UNLESS edge completely separates the two steps (clip on edge to test for this special case)
    else if edge shares one vertex with adjacent step, other vertex not shared -> Flat

Appendix B

Where To Find Things

The stairmaster program is located in **~cs285-ag/SM** on the **torus** cluster. The executable is:

stairmaster

Sample floorplan files are:

fp*n*.ug

where n = {1, 2, 3, ...}

The code for the original *animator* is located in **~funk/walkthru/util/animator** on host **ignatz**.

A Haunted 3D Maze Walkthrough

John F. Sefler

Fall 1994

CS 285 Procedural Generation of Objects

Abstract:

Imagine yourself stranded in a dark hallway of a haunted house. Your objective is to find your way out. As you move down the hallway, you are faced with many decisions. Should you go to the left, should you go to the right, are you going the right way? Then all of a sudden from around the corner, you see a life-sized creature walking directly toward you. There is nowhere to go except up the stairs where another creature awaits.

The haunted 3D maze walkthrough is an exciting project that gives the user an objective: to finish the maze!

I. Introduction

The Haunted 3D Maze Walkthrough project is well suited as a final project for CS285. It expounds upon prior course projects and addresses issues in system integration. In this project, a UniGrafix 3D maze generation program is developed. It is used to produce a static maze scene for viewing with the Walkthrough visualizers. Contributing to the static scene, a UniGrafix staircase generator, implemented as prior course assignment, is used to design a general purpose staircase for inclusion in the maze. The haunting aspect of the maze comes from dynamic six-legged walking creatures that are randomly placed by the maze generator and oriented to walk up and down the corridors of the maze.

II. 3D Maze Generation

The core 3D maze generation algorithm is a straight forward extension of the 2D maze generator developed as a previous course assignment. The construction of the maze is based on a depth-first-search algorithm that visits nodes in a 3D gridded data structure. The diversion from the core maze generation occurs in the following options.

i. Level Access Restrictions

Like most building floor plans, the user may want to restrict the number of paths (stairwells) between adjacent levels (floors) of the maze. To accomplish this desire, the maze generation algorithm keeps track of the number of times it takes a step across adjacent levels. Once this limit is reached, further visit attempts across this boundary are blocked.

ii. Maze Looping

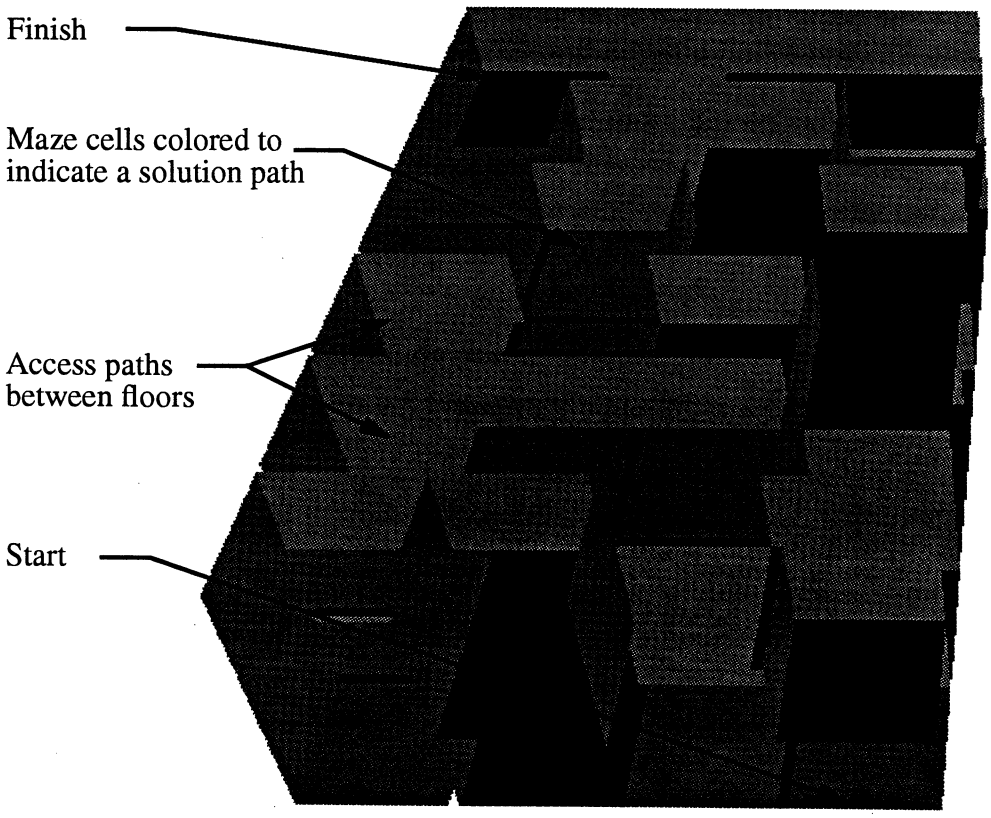
As an additional challenge, the user may also want to include loops in the maze.

To include loops, the maze generation algorithm is followed by a post processing step. After completing the construction of the maze, the algorithm randomly chooses a user specifiable number of walls for removal. With every wall removed a new loop is inserted into the maze. However, to insure that the removal of the selected wall actually produces a loop, the wall must be checked for connection to adjacent walls. Removing a wall that has no connecting adjacent wall would create a large room, not a loop.

iii. Maze Solving

Not only should the maze be generated, the solution path should also be shown when instructed by the user to show it. The solution algorithm is based on a breadth-first-search through the 3D gridded data structure. Beginning at the selected start cell, adjacent cells are visited in a breadth-first-search manner until the finish is reached. Because loops may have been inserted in the maze, the breadth-first-search algorithm continues until all branches of the search tree have either joined the solution path, thereby creating solution loops, or have reached a dead end. Dead end branches are not counted in the solution.

Having incorporated these maze generation features, a UniGrafix description of the maze walls and portals can be written to a flat file readable by the Walkthrough visualizers. Figure 1 below shows a 6-by-4-by-3 solved maze with no restriction on the number of paths between adjacent levels.



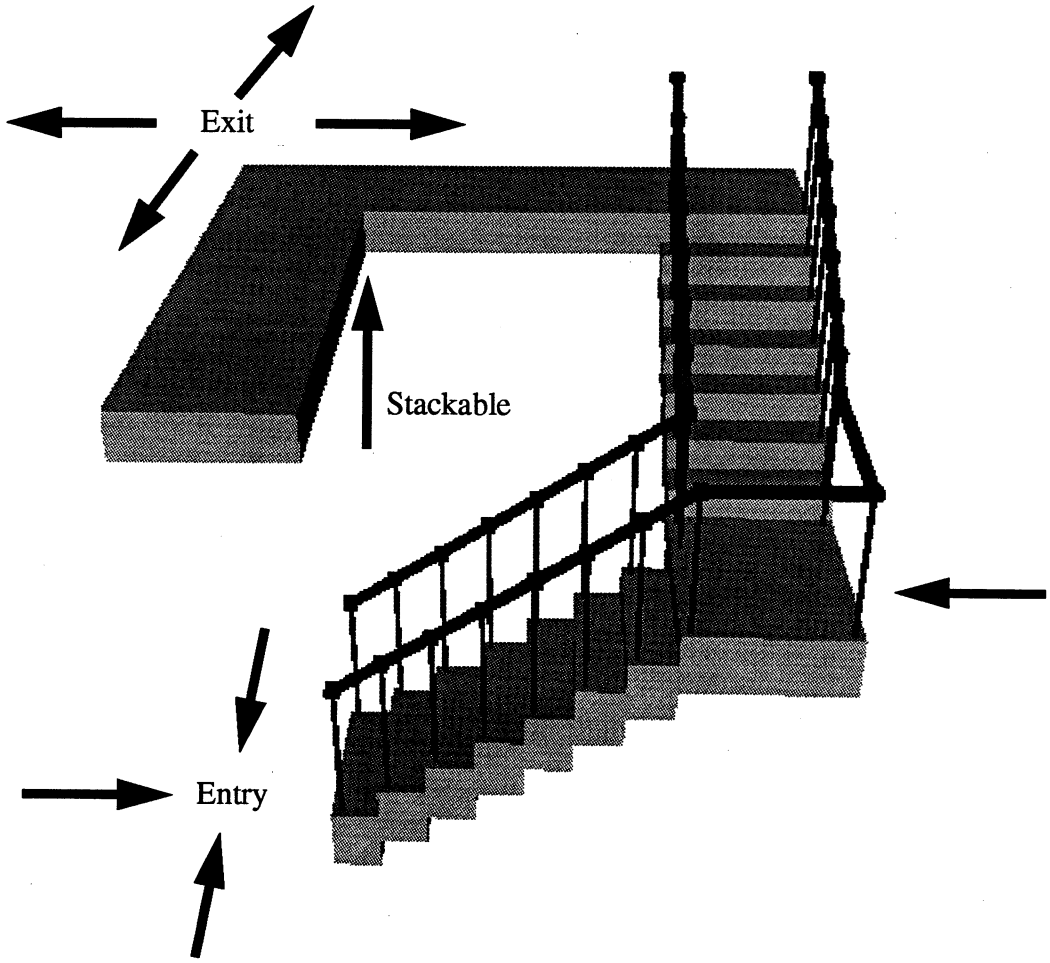
* Copied from a color image - It may be difficult to distiguish features.

Figure 1: A solved 3D UniGrafix maze viewed with `wkmotif`.

III.Staircase Design and Insertion

Having generated the maze and rendered it in UniGrafix, the Walkthrough project allows inclusion of UniGrafix instances of predefined objects. Therefore, we can include multiple instances of a staircase object placed in the stairwell cells for use as a mechanism for traversal from one level to another. However, the staircase must first be designed and generated. To do this, a previous course assignment was resurrected, the UniGrafix staircase generator.

Shown below in Figure 2 is a general purpose staircase design that is accessible from all sides on the entry level and exit level. Likewise, its stackability addresses the issue of stairwells that span more than one floor. With these properties, the placement of the staircase in the maze is independent of the adjacent cell topology.

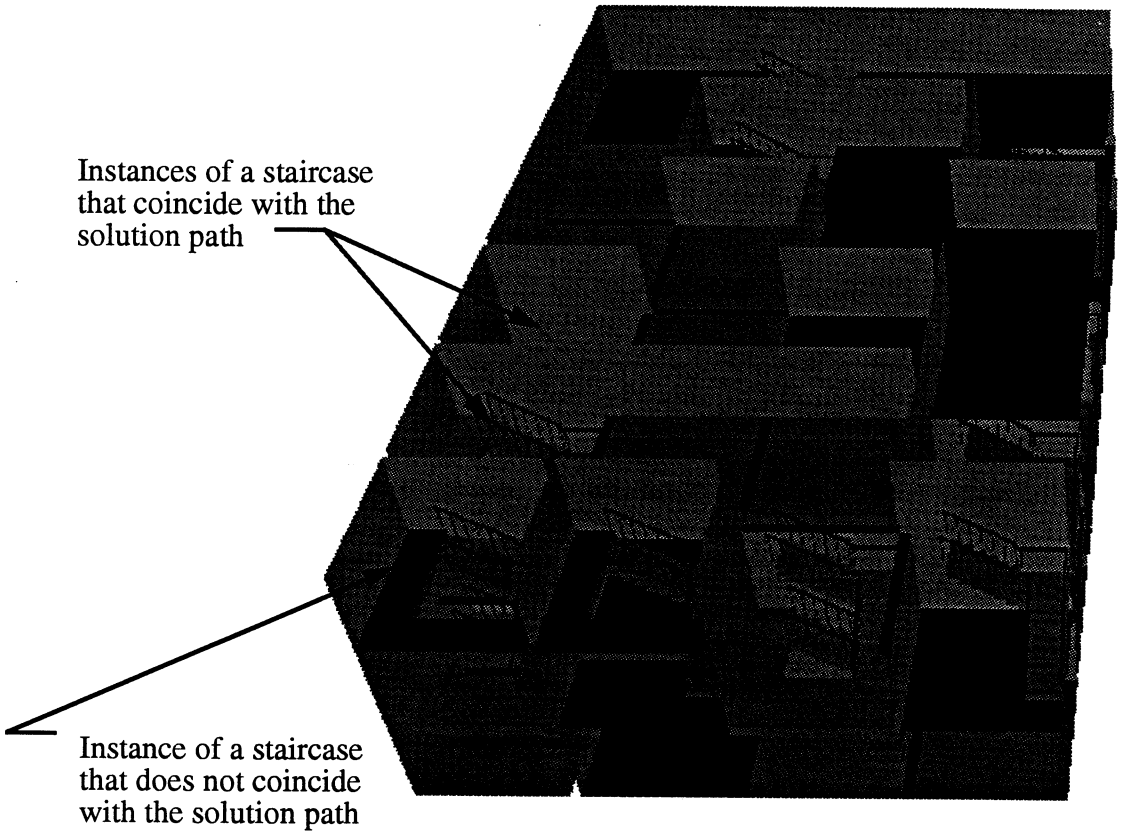


* Copied from a color image - It may be difficult to distiguish features.

Figure 2: A maze topology independent staircase design.

Having designed the maze topology independent staircase, Figure 1 of the maze can now be enhanced to include instances of the staircase object. Of course, the staircases are also colored appropriately when they coincide with the solution path of the maze. The enhanced maze is

shown in Figure 3.

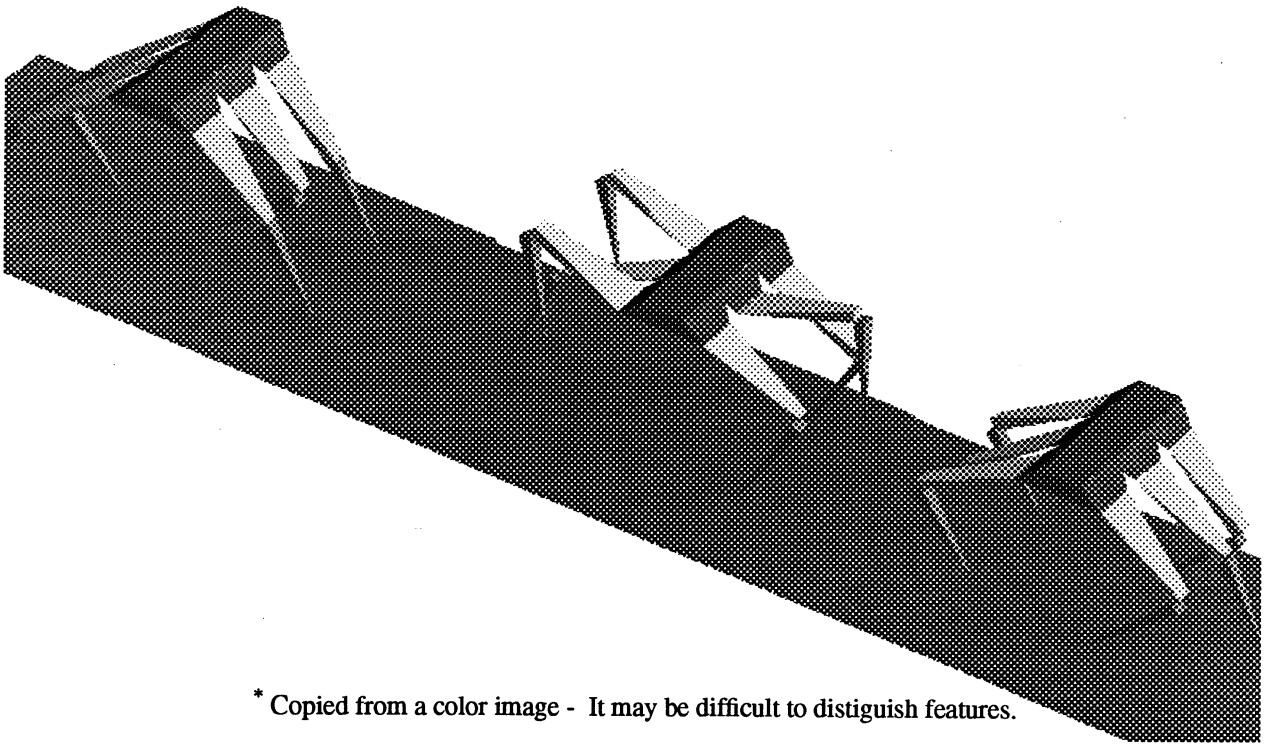


* Copied from a color image - It may be difficult to distiguish features.

Figure 3: 3D maze with staircase objects inserted.

IV. Walking Crab Design and Insertion

To add a haunting aspect to the maze, the user may want to randomly distribute walking creatures throughout the maze. To accomplish this aspect, a previous course assignment is once again exploited. A six-legged hermit crab-like walking creature previously designed for viewing with the UniGrafix `ugmovie` viewer is redesigned for compatibility with the Walkthrough viewers: `wkmotif` and `wkedit`.



* Copied from a color image - It may be difficult to distiguish features.

Figure 4: Views of the walking crab using `ugmovie`.

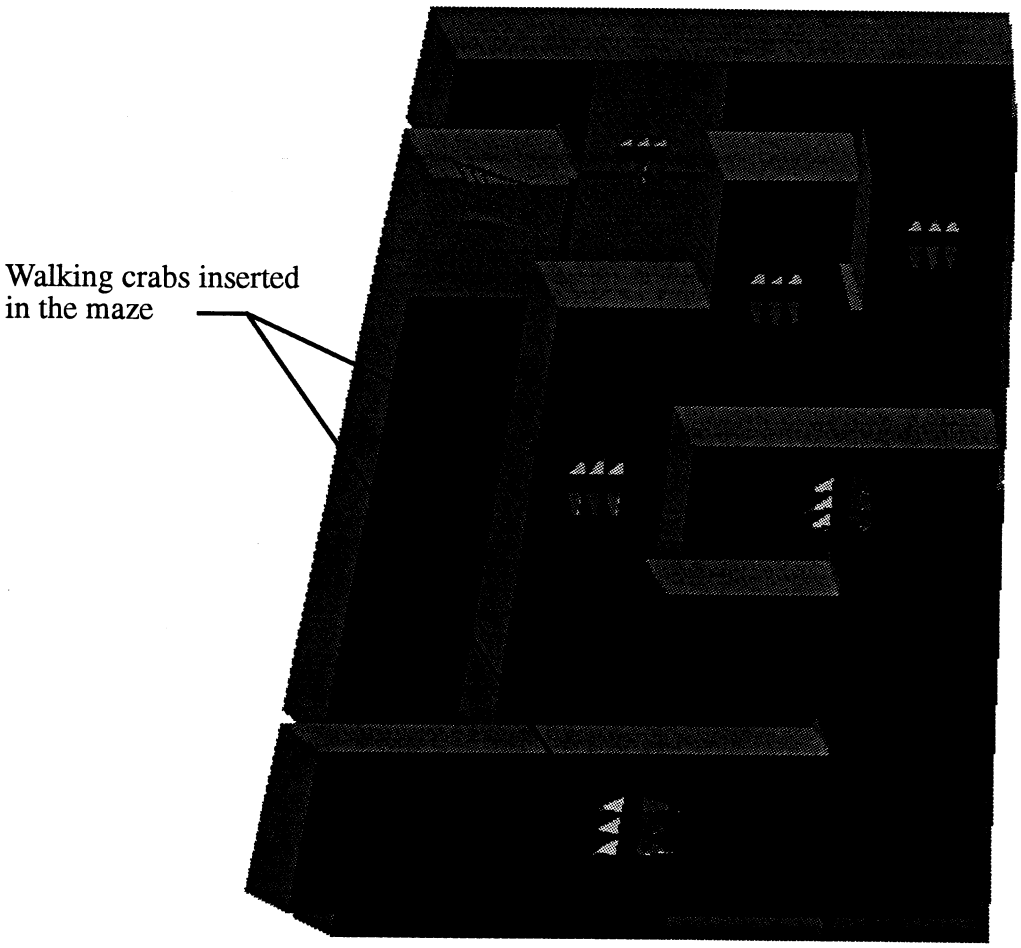
To include a dynamic object in the walkthrough, the blocks of `meval` expressions must hierarchically be positioned at the top level. Unlike the UgMovie format which allows an `meval` expression to be defined within a `def` block, the walkthrough database requires the `meval` expression to be outside all `def` blocks. Therefore, all time dependent transformations on an object must occur on a single instance of the object. Schematically, the format of the UgMovie-to-Walkthrough compatible description is shown in Figure 5.

Top-Level `meval` block

```
{...some title information}
def object_1;
   {...colors, vertices, and faces...}
end;
def object_2;
   {...colors, vertices, and faces...}
end;
(meval
   i instance_1 (object_1
      {...dynamic transformations...}
   );
   i instance_2 (object_2
      {...dynamic transformations...}
   );
)
```

Figure 5: Schematic of a dynamic UniGrafix object compatible with the Walkthrough project.

Having redesigned the UgMovie constructed walking crabs to be compatible with the Walkthrough project, multiple instances of the crabs can now be included in the maze. The crabs are randomly inserted into different corridors such that they walk the entire length of the corridor. To do this, a `#define` specifying the number of steps accompanies each crab instance and is included in the `maze.ug` object file to be described in the next section. Also accompanying every crab instance is a unique position and rotation for placement in the maze. The algorithm for placing the crabs in the maze begins by randomly choosing a cell followed by a traversal of the corridor until it reaches the end. If the entire length of the corridor is greater than one cell and is free of the start, finish, staircases, and other crabs, the crab is positioned at the end of the corridor. Figure 6 shows a few crabs inserted into the second level of a maze.



* Copied from a color image - It may be difficult to distiguish features.

Figure 6: Walking crabs traversing up and down corridors of the maze.

V. Software Organization and Execution

There are two steps for creating a haunted 3D maze. First, the maze must be generated using `wkmazegen` which generates the 3D UniGrafix maze. Second, the walkthrough database is compiled using a script written specifically for the haunted maze walkthrough. As part of the second compilation step, all the static objects (staircases, ceiling lights, furniture, etc.) and the dynamic `meval` objects (walking creatures) are added to the walkthrough database.

i. wkmazegen

This is the executable for generating the haunted 3D maze. It produces two files; `maze.macro` and `maze.cpp.ug`. The `maze.macro` file is a flat UniGrafix file describing the walls and portals that comprise the maze. This geometry is used during the compilation of the walkthrough database (described in the next step) to determine adjacent cell visibility. The `maze.cpp.ug` file contains instances of static objects and #includes of the dynamic `meval` objects. Because `#defines` and `#includes` are used in the `maze.cpp.ug` file, part of the compilation procedure will use the C preprocessor; `/usr/lib/cpp`.

The usage of `wkmazegen` follows:

`wkmazegen [-x #] [-y #] [-z #] [-s #] [-l #] [-g #] [-p]`

`-x` number of rows (default: 4)

`-y` number of columns (default: 5)

`-z` number of levels (default: 2)

`-s` number of stairwells between levels (default: 1) (0 -> no restriction)

`-l` number of loops per level (default: 0)

`-g` number of ghosts in the maze (default: 0)

`-p` show all solutions paths

Using an unrecognized option such as "`-help`" will also display this usage.

iv. Steps to compile the walkthrough database

Having used `wkmazegen` to generate the `maze.macro` and `maze.cpp.ug` files describing your very own haunted 3D maze, the walkthrough database can now be compiled. In a single directory, the following files are needed:

`maze.macro`	- generated by wkmazegen in step i.
`maze.cpp.ug`	- generated by wkmazegen in step i.
`maze_stairwell.ug`	- contains UniGrafix definitions for a single staircase.
`maze_crab.cpp.ug`	- contains UgMovie time dependent crab transformations.
`maze_crab.defs`	- contains UniGrafix definitions for the crabs body and legs.
`maze_hingebug.cpp.ug`	- contains UgMovie time dependent hingebug sculpture.
`maze_furniture.ug`	- contains definitions for furniture objects placed in maze.
`maze.reset`	- used by wkedit and wkmotif for setting viewing defaults.
`wkmakemaze`	- instruction script for compiling the maze walkthrough.

To compile the walkthrough database, simply type:

`wkmakemaze`

The `wkmakemaze` compilation script was written specifically for the haunted 3D maze walkthrough. This script executes numerous walkthrough utilities for creating the maze

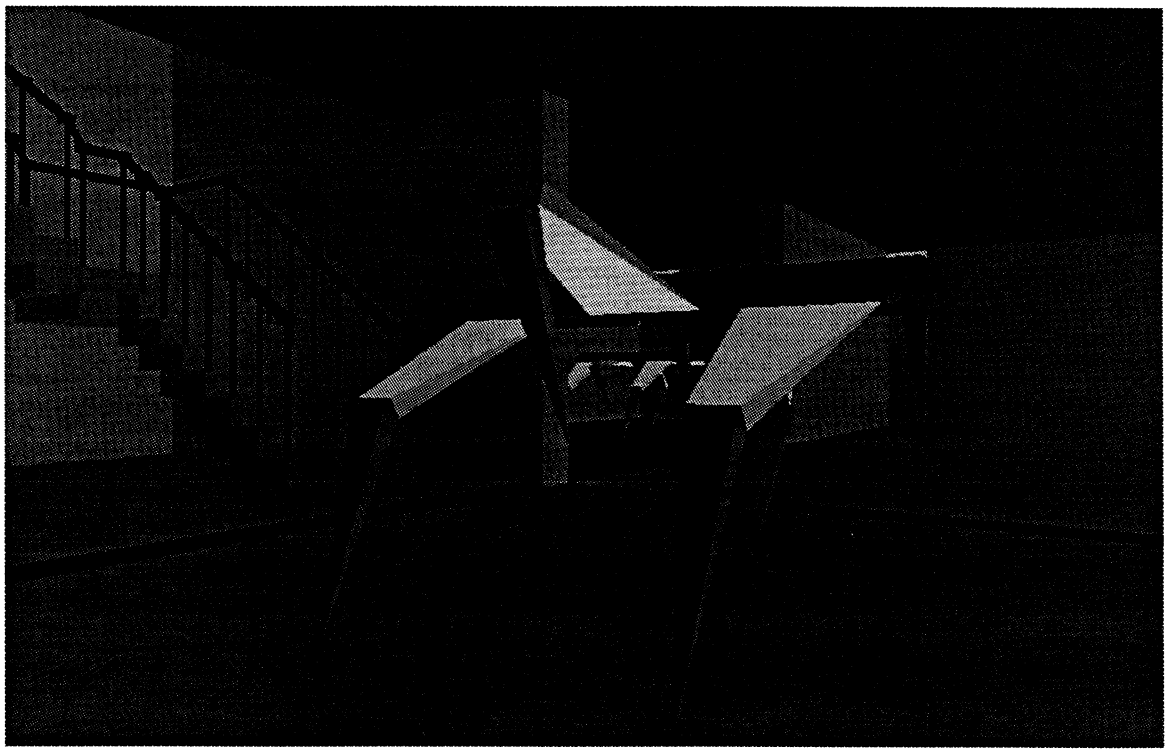
walkthrough database called `maze.wk`. Among these utilities are `/usr/lib/cpp` used to replace the `#defines` and expand the `#includes` used in the `*.cpp.*` files, and `wkadd maze.wk -m -i -f -v -u maze.ug` which is used to make database objects for each Ug meval, Ug instance, and Ug face found in `maze.ug`.

To view the Haunted 3D Maze, use either of the walkthrough visualizers `wkmotif` or `wkedit` by typing:

```
wkmotif maze.wk  or
wkedit maze.wk.
```

VI.Summary

As presented in the sections above, this project largely consisted of integrating previous course assignments into a single package; the Haunted 3D Maze Walkthrough. This involved extending a 2D maze generator into 3D and redesign of previous assignments for compatibility with the walkthrough visualizers; `wkmotif` and `wkedit`. The life-sized walking crabs are especially exciting as they independently walk up and down the corridors in search of a victim to walk all over!



* Copied from a color image - It may be difficult to distiguish features.

Figure 7: Trapped inside the Haunted 3D Maze Walkthrough.

Dynamics and Collision Detection in Soda Hall
Course Project Report, CS 285, Fall 1994

Richard Bukowski
University of California, Berkeley

December 7, 1994

Abstract

The Canny-Lin closest feature algorithm has been integrated with the Soda Hall Walkthrough program. The algorithm has been placed into a Walkthrough library that provides two major new functions. It is now possible to perform collision detection on WK database objects, and it is possible to perform a dynamic simulation on a database object which will cause it to react realistically to gravity and external forces in a frictionless environment. This new functionality provides added realism as well as new capabilities for the walkthrough editor.

1 Introduction and Motivation

In the design and construction of the walkthrough editor, we often wanted to have the ability to do various sorts of dynamic simulation. Part of the goal of our object association technique was to be able to make objects behave similarly to the ways they behave in the real world. This seems natural in our virtual environment, and when objects behave naturally, manipulation of those objects is far more intuitive. To make objects truly behave similarly to the way they do in the real world, you need some degree of dynamics. Unfortunately, dynamic simulation is a time-consuming algorithmic process, dominated in most cases by collision detection. Because the walkthrough had a greater goal, that of interactive frame rates, we had discounted dynamics to preserve our frame rate.

However, in the recent past, Canny and Lin have developed an algorithm that can maintain and quickly update the closest feature of two convex polytopes. With this algorithm, doing limited dynamic simulation should be possible while maintaining the frame rates that make the walkthrough program interactive. Once we have this capability, we can reach further with both our realism level and our interactive techniques.

Research into realistic rendering and visualization techniques is an ongoing effort for the walkthrough. With the ability to do dynamic simulation, we can make objects settle properly; if there is a table with a pair of legs in the air, that table will no longer float above the floor, but will fall to its natural resting position. If a cup is sitting on an incline, it will not stay still, but will slide to the next lower surface. If a book is hanging over the edge of a table, it will fall off and hit the floor. Such events will make the walkthrough a much more realistic experience.

Another field where dynamic capabilities will be helpful is in my own research into interactive manipulation techniques. The ability to do true collision detection will enable the pseudo-gravity association to settle objects to the supporting surface in the ways mentioned above. Further research will also be able to make use of dynamics and collision detection to improve our user interface. For example, we could implement a "rubberband" type dragging interface, where dragging an object produces a force that drags the object to the cursor. Such a force-based approach to movement has advantages for alignment of objects; aligning two desks, for example, is as easy as pressing one against the other. The natural torques generated by their

meeting will force them to rotate and press their backs evenly together. These new approaches will negate the need ~~to~~ for the user to do some of the explicit constraining of objects necessary where there is no collision detection. Another use is in design of buildings for construction. For example, is it possible to move a piano up a staircase? If we have dynamics and collision detection, we can find out by simply trying to drag it up the stairwell.

2 Algorithms and Functions

2.1 Object Representation

Representing objects for simulation is a problem in the walkthrough. The binary database format for the walkthrough is generated from UNIGRAFIX source code, but it does not maintain the UG winged-edge data structure during the actual walkthrough. The binary database represents an object as a list of 2D facets, each of which has a separate list of vertices. This representation is not compatible with the Canny-Lin algorithm, which requires a convex decomposition of each database object and a winged-edge structure for each convex subpolytope. Simulation also requires the storage of a center of mass, mass, and inertia tensor with each object. These values are not a part of the standard WK database format.

Due to the fact that the walkthrough and Canny-Lin algorithms have very different needs for object representations, the new simulator package maintains a separate object class database. For each object class, the alternate database contains a convex decomposition of the object into winged-edge subpieces, the relative poses of the pieces, and physical parameters such as the center of mass, mass, and inertia tensor of the object. The simulation database is initialized from two auxiliary files that must be present with the walkthrough binary database. One of these files holds the winged-edge convex pieces; the other contains the simulation information and construction information on which pieces to put together for which database class. When the database is loaded, a set of *polyobject* structures are created, one for each database class, containing this information. Only two polyobjects are allocated for each type to save space, since the algorithms in the simulation library only operate on pairs of objects. They are stored in a lookup table; when the simulation algorithms need to operate on a database object, they request a polyobject representation of that object by its class identifier in the lookup table.

Additionally, two external tool programs were written in this phase of the project. The *converter* takes as input the original UG file for the WK database and creates the winged-edge structure, center of mass, and mass for each class object in the file, in the formats required by the simulator. Center of mass of a polytope is computed by finding a point inside the polytope and taking the mass-weighted average of the centers of mass of all pyramids formed with the given point and each facet of the polytope. This program does not perform a convex decomposition, and does not compute an inertia tensor; I did not have time to implement either. It is interesting to note that the latter problem is a very complex one, involving at least three nonlinear triple integrals over the object, and, unlike the center of mass computation, cannot be decomposed into subcomputations on smaller, simpler pieces; the integrals must be evaluated on the entire volume of the object at once. The second utility program generates the complete polyobject representation for a rectilinear object of given dimensions; this was used to make a "blockworld" for testing the simulator.

2.2 Integrating the Canny-Lin Algorithm and Linking Objects

The Canny-Lin algorithm maintains the closest pair of features of two convex polytopes. Features in this context are points, edges, or facets of the polytope. There are two important procedures in Brian Mirtich's implementation of the algorithm. The first procedure is an initializer; when simulation is begun, the initializer is called to determine the closest features. The procedure returns the closest features and the distance between them. Once the closest features have been determined for the first time, the incremental update procedure can be called with the previous set of closest features as the starting point. This procedure searches from the old set of features to the new set of closest features and returns them. Because the old features are

used as a starting point, finding the new features is very fast if the objects did not move by a large amount.

Brian's code was easy to port and compile into the simulation library for the walkthrough. The polyobject representation uses his structures for the convex subpieces, so there is no translation layer involved; the library can simply call a function on pieces of two polyobjects to find their closest features. However, in order to provide any real benefit, the system must retain the closest features of two particular subparts of two particular database objects between calls to the Canny-Lin code.

This benefit is realized through the linking mechanism implemented in the simulation library. When the user wishes two database objects to interact during simulation, a library function is called that allocates a link between the objects. Each database object has a single additional pointer added to its external data field, which points to a singly linked list of simulation links. Each simulation link is a part of two such linked lists, one for each object. The link contains pointers to the objects, pointers to the next elements in the respective lists, and a matrix of the closest features of the subpolytopes as of the last call to the Canny-Lin algorithm. This matrix contains $2nm$ entries, where n and m are the number of subpolytopes of the two objects. Each entry is a single void pointer that points to a feature node in the central, generic representation of the subpolytope, making the link as compact as possible.

When the user tries to move or simulate with an object, the linked list is traversed to find which other objects in the world can possibly interact with the selected object. Thus, the programmer can control the extent of the interactions between objects by making and deleting links as the objects move around; if an object moves to a different room, for example, it can be unlinked from the objects in the old room, and those objects will cease to impact the running time of the simulation. In the sample code written for this project, when an object is selected, the grouping function in the object association mechanism of the walkthrough editor is called. The grouping function has been modified to link every object it touches in the local search to the selected object. This has provided very satisfactory local linking for the demo program. No unlinking is done in the demo; in practice, objects would unlink themselves when they move to a different room or database cell.

The Canny-Lin algorithm is accessed via a function call that takes two database objects and a transformation matrix for each. The function retrieves the previous closest features from the link structure and uses those as the starting point to find nm new closest features, pairwise for each subpolytope, on the polyobjects transformed by both the object's current pose and the given matrices. The smallest distance between any two subpolytopes is the smallest distance between the polyobjects. This smallest distance is returned, and the link is updated with the new closest features.

2.3 Collision Detection

Once links are established, the user may use the collision detection routine on any database object. The collision detector takes a database object, a motion vector for that object, and a maximum time value. The motion vector is a complete 6-element 3D velocity vector, with a linear velocity component and an angular velocity component, where the angular velocity is specified about the object's center of mass. The function computes the following: Starting with the object at its current position, for what maximum value of t, $0 \leq t \leq t_{max}$, can the object move by vt, where v is the given velocity vector, before the object collides with an object to which it is linked?

This function uses a numeric algorithm on $d(t)$, the minimum distance returned by the Canny-Lin algorithm between the polyobject and any object linked to it as a function of time, to find t_c, $0 \leq t_c \leq t_{max}$, for which $d(t) \geq \delta$ for all $0 \leq t \leq t_c$. For numerical stability, no objects may get closer than some preset small δ; the value used in the demo is about one thousandth of an inch.

In creating this function, the properties of the Canny-Lin algorithm were very important. Canny-Lin cannot handle interpenetration or even contact between objects; if such a condition occurs, the algorithm goes into an infinite cycle. Thus, it was imperative that the collision detector be very conservative, never even computing $d(t)$ for any value of t that might result in interpenetration or contact. To get this property, we

use a "stepping" algorithm. We start at $t = 0$ and attempt to step forward in time as much as possible with each iteration while remaining absolutely safe. Since our velocity vector is rigid and linear, if we know $d(t_c)$ for some t_c, we can compute a step size bound b_{step} for which $d(t) \geq 0$ for $t_c - b_{step} \leq t \leq t_c + b_{step}$. b_{step} is the maximum delta time such that any point on the object can move at most $d(t_c)$ in that time. If v is the translational velocity, a is the smaller of the angular velocity or π, and r is the radius of the bounding sphere of the object, we can write

$$vb_{step} + arb_{step} \leq d(t_c)$$

or

$$b_{step} \leq \frac{d(t_c)}{v + ar}$$

The collision detector algorithm works roughly as follows:

1. Set $t = 0$;
2. While $t < t_{max}$ and $d(t) \geq \delta$:
   (a) Compute b_{step} for distance $d(t)$;
   (b) $t = t + b_{step}$;
   (c) if $t > t_{max}$ then $t = t_{max}$;
3. if $t = t_{max}$ then return t;
4. Set t to the last valid time when $d(t) \geq \delta$;
5. While $d(t) \geq \delta + \delta^2$:
   (a) Compute b_{step} for distance $d(t) - \delta$;
   (b) $t = t + b_{step}$;
6. Return t;

The first loop attempts to advance t to t_{max}. If this loop fails, then a collision occurs; the second loop backtracks and attempts to step as close as possible to δ distance between the objects before returning a value.

Note that this program loop is not called on all objects to which the selected object is linked. Before any motion is attempted, a culling step is performed where the bounding sphere for the selected object is swept along the velocity vector and compared with the bounding spheres of all of the linked objects. Only linked objects whose bounding spheres intersect with the selected object's bounding sphere somewhere along the path are tested in the main loop. This provides a large advantage in efficiency, since the bounding sphere computation is very fast, is only done once, and tends to remove all but a few of the linked objects from consideration as blockers.

2.4 Contact Forces

Contact force computation is the next component in the simulation mechanism. To determine how an object is going to move, you must be able to compute what total force is acting on it at a particular time. The contact force algorithm is taken from a paper by Baraff in SIGGRAPH '94 [1]. The paper points out that you can model contact forces on polyhedral, frictionless rigid objects as point forces of two major types.

Vertex to face contacts generate a force at the vertex-face intersection point, in a direction purely normal to the face. Edge to edge contacts generate a force at the intersection of the two edges in the direction of the cross product of the direction vectors of the edges. The reason we can specify the directions of the forces is that frictionlessness implies contact forces must be purely normal to the contact surfaces. Other contacts may be decomposed into these two primitive types; for example, face to face contact may be modelled by taking the region of overlap of the faces, which will itself be a polygon. Each of the vertices of the overlap region will be either a vertex-face or edge-edge contact, and these vertices completely describe the contact force between the two faces. Figure 1 shows these contact formations.

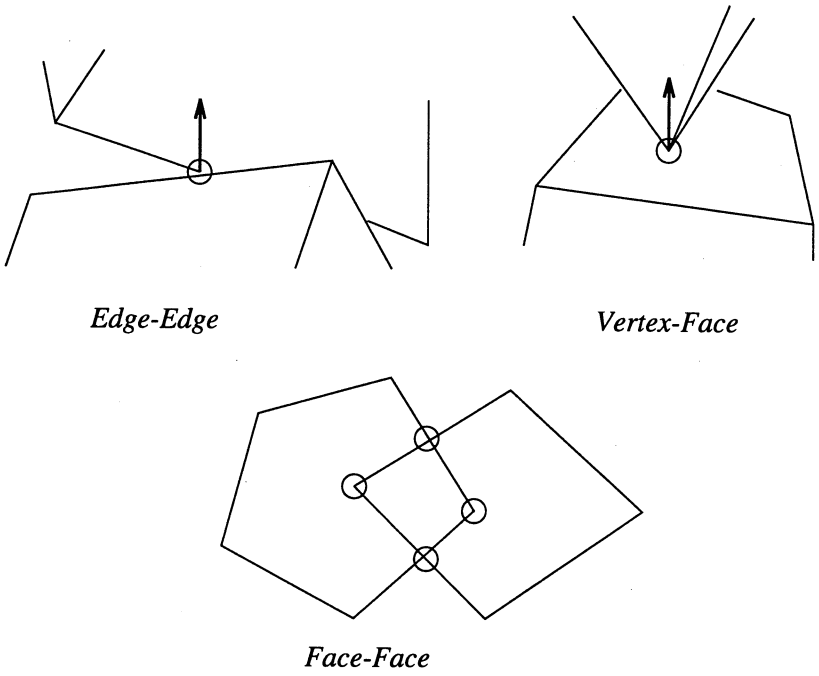


Figure 1: A diagram showing the contact points and normals for Vertex-Face and Edge-Edge contacts. The decomposition of a Face-Face contact into two V-F and two E-E contacts is also shown.

The first step is to compute all such contact points between the selected object and other objects to which it is linked. This step was not covered by the paper, so I implemented my own algorithm. I take advantage of the fact that we have the convex decomposition. Once more, when testing two objects against each other to find the contact points, all nm combinations of subparts are tested. The Canny-Lin algorithm is called to determine the closest features for each pair of subpolytopes. Once the closest features are found, the only valid contact points that can exist between the polytopes are the immediate neighbor features of the closest features. For example, if a vertex v of object 1 is closest to a face f of object 2, the only valid contacts can be between the edges and vertices on facets incident to v and the edges and vertices on face f. The regions of interest are the vertices and edges on faces incident to a closest vertex, the vertices and edges on the two facets adjacent to a closest edge, and the vertices and edges bounding a closest facet. This considerably reduces the number of comparisons between features necessary to compute all of the contact points between objects. Furthermore, a trivial reject can be performed in the case that the closest distance returned from Canny-Lin is larger than 2δ.

Once the full set of contact points and force directions at each point are determined, the SIGGRAPH algorithm comes into play. We will define two vector quantities, the relative acceleration a_i and the contact force f_i at each contact point p_i. a_i is the relative acceleration between the two bodies at p_i; a positive value means the objects are moving apart, and a zero value means they are sliding or pushing against each other. f_i is positive if the objects are pushing together at p_i, and zero of the objects are moving apart. From this description, it is clear that $a_i \geq 0$, $f_i \geq 0$, and $f_i a_i = 0$ for all i, since objects can either be moving apart or pushing against each other, but not both. Also, from physics,

$$\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b}$$

where $\mathbf{A}$ is a matrix representing masses and inertial forces, and $\mathbf{b}$ is a vector representing external forces such as gravity. These conditions define a *linear complementarity problem* or LCP. The SIGGRAPH paper gives an algorithm for solving it.

Given the solution for the forces f_i, we multiply the previously determined contact force direction times these values and apply the forces to the center of mass, giving a single 3D force and torque vector that represents the total force on the object at the given instant in time. Given the total translational and rotational force (f and τ) and the mass (m) and inertia tensor (I), we can determine the acceleration of the body (a and α) by Newton's and Euler's equations $f = ma$ and $\tau = I\alpha$. Since we are limiting this simulation to "pseudo-static" conditions, i.e. no momentum or velocity, we do not need to include the full formulation of Euler's equation $\tau = I\alpha + \omega \times I\omega$, which includes angular velocity ω, accounting for the resistance of a rotating object to off-axis forces.

2.5 Moving Bodies with Contact Forces

The provided functions can now be used to generate simulated dynamic motion of objects. We simply call the contact force computation routine, which returns a force on the object. We call another routine which converts this to an acceleration. The acceleration is used as an instantaneous velocity vector and is passed to the collision detector, which determines how far the object can move in the direction of the acceleration. If the object cannot move at all, it is stable and the simulation is completed. If the object can move, it is moved as far as possible, and the process is repeated at the new location, which will in general have an entirely new set of contact forces.

There is a problem with the interaction of these algorithms in the simulation loop. Since the math is subject to small numerical fluctuations and the accelerations are not really being applied "properly" (as accelerations on a continuously updated velocity vector), the collision detector can cause objects to get "stuck." This usually happens when a sliding contact is indicated by the algorithms. Sliding contact involves a motion that is exactly parallel to the two surfaces. Even double-precision arithmetic is not enough to make the velocity vector exactly parallel the surface, so the object will tend to try to move infinitesimally toward the surface. The collision detector then stops the object from performing any motion at all, since the motion would cause the objects to get a tiny bit closer together than δ. Since the object doesn't move, the contact forces don't change, and the object gets jammed in a nonrealistic position for no apparent reason..

The present solution to this problem is to have the contact forces repel the object slightly. A repulsive force proportional to some user-set constant times the mass of the object times an inverse square of the distance between objects is applied at each contact point. This tends to cause the objects to want to separate slightly at their contacts, giving the collision detector some slack and allowing the object to slide along the surface. This solution actually still gets stuck sometimes, but it can be "un-stuck" by simply increasing the repulsion value. I have not yet found a situation where increasing the repulsion value has not admitted the correct behavior. This suggests a fully functional adaptive algorithm which checks to see if the collision detector is forcing the object to remain in an unbalanced position, and if it is, increases the value of the repulsion slowly until the collision detector "lets go" of the object.

2.6 Implementation in the Walkthrough Editor

A specially modified version of the walkthrough editor, called *wkcm*, has had the simulation library integrated into its user interface. There is a new control panel in the options menu with three controls. The first is a checkbox that turns on collision detection for normal interactive motion. While this option is active, any motion the user applies to an object with the standard editor operations is collision checked, and if a collision

is detected, the object is only moved far enough to collide. This demonstrates the speed and interactivity of the collision detection and linking mechanisms.

The second control activates simulation. When this is on, the user may click and hold on an object, and that object will be subjected to the simulation loop until the user releases the mouse button. The user can cause blocks to settle, octahedra to fall over, or bricks to tip off of surfaces. The third control is a numerical entry box that allows the user to control the repulsion force parameter, increasing it if the objects get stuck or decreasing it if the objects are "bouncing" too violently (which happens when the repulsion gets too large).

3 Results

This project has integrated fast closest-feature finding, collision detection, and dynamic force simulation into the Soda Hall walkthrough. There is a new version of the walkthrough editor that allows users to perform dynamic simulations on objects with gravity as well as collision detect their movements while editing objects. A simple sample world has been provided which contains some basic shapes (spheres, octahedra, rectangular prisms, and barbells) that the user can play with. The algorithms are sufficiently robust to handle a wide variety of contact situations, and with a bit of help do a good job of simulating frictionless interactions between objects. Figure 2 shows a screen shot of the modified editor during a simulation; a brick is falling off the edge of a cube. The dynamic balance on the edge of the cube is easy to see.

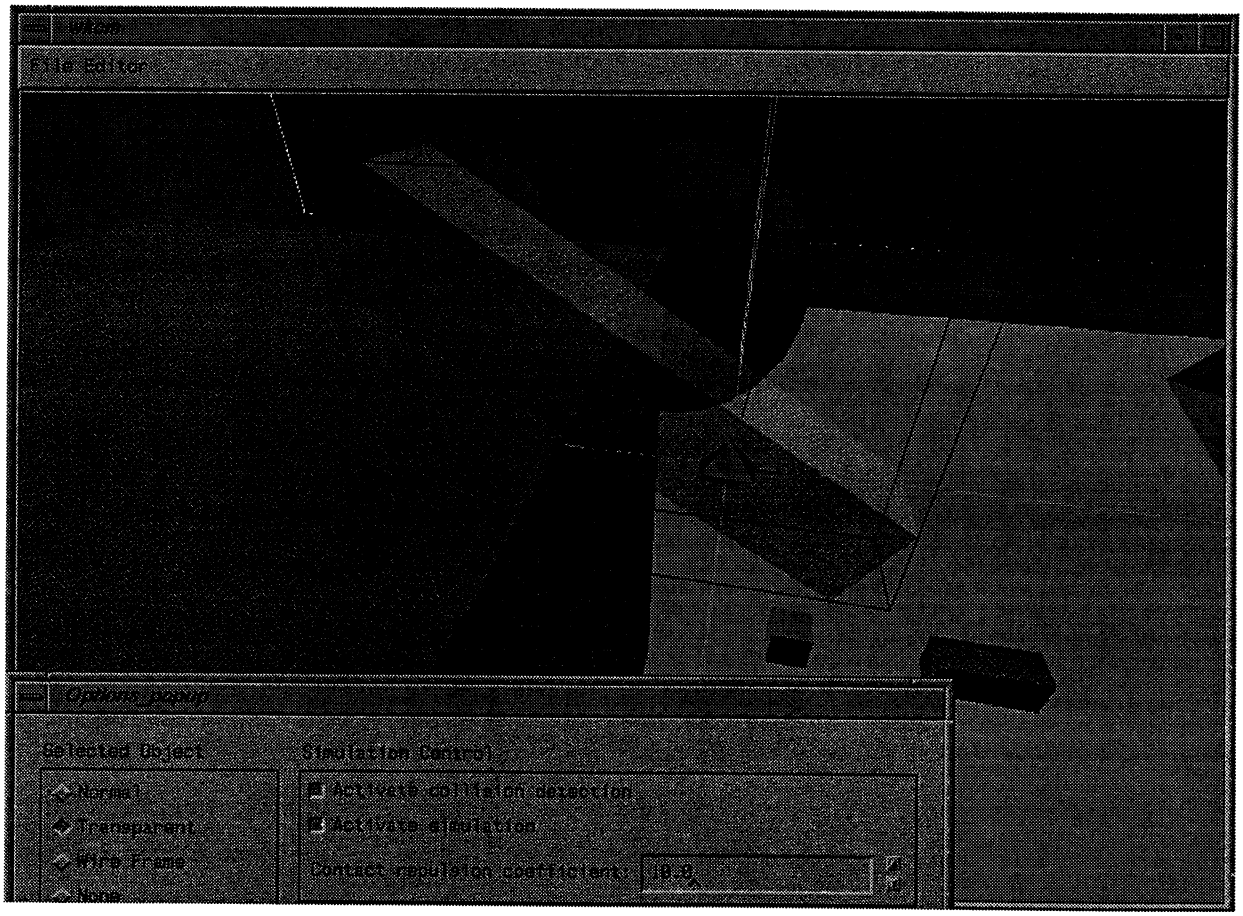


Figure 2: A screen shot showing a brick falling off of a cube. Note the overbalancing of the brick and the hingeing action; this configuration has two Edge-Edge contacts.

This was a very interesting project. I learned a great deal about the physics and algorithms of doing

dynamic simulation, including some surprising dead ends. For example, computing the mass of an arbitrary 3D polyhedral object took one hour of coding; computing the angular equivalent of mass, the moment of inertia, has taken Brian Mirtich a month, and is "more difficult than [he] had thought." That particular problem has forced me to implement a less interesting world than I would have liked; I had to include only objects that were easy to integrate the inertia functions over. I also experienced many problems with scaling in the transformation matrices, and eventually had to produce a database with no scale operations in it to run the algorithms. As a general use tool, it is difficult to say how much more needs to be done to use these algorithms productively in the Soda Hall walkthrough.

The foundation has been laid for doing more interesting physical simulation in a walkthrough environment. Future directions include better collision detection and dynamic simulation algorithms, smarter linking and unlinking, and handling of arbitrary scalings and automatic computation of physical parameters and convex decompositions of models. Eventually, this system could be of great benefit to the building walkthrough research program in both the user interface and the realism of the walkthrough.

References

[1] Baraff, D. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. *Proceedings of SIGGRAPH '94*, Orlando, Florida, July 1994, pp. 23-34.

Origami Folding Demonstration

Course Project Report, CS 285, Fall 1994

Sara McMains

University of California, Berkeley
Soda Hall, UC Berkeley, Berkeley, CA 94708

Abstract

This project is a series of ugmovie files that demonstrate the steps in folding two different origami figures, as well as a tool developed to aid in creating such files by outputing equations in a form ugmovie can read.

1 Introduction

Many origami instruction books have been written, but the instructions are often difficult for a novice to follow. One problem is that the diagrams show what the work in progress looks like after each step, but often it is not clear exactly what was done to get from one diagram to the next. An interactive demonstration addresses this problem by animating what happens in between each step, showing the folding in action.

I have modeled the folding of each origami figure in a series of ugmovie files that the user can start up with a single command, sequencing through the animation of each step in turn after completing the previous step on a physical sheet of paper. The user can cycle through the animation of the current folding step as many times as necessary to figure out exactly what to do, using the crystal ball interface to view the object from any angle.

While some of the folds used to make the figures can be described as simple rotations of polygons, and therefore can be described succinctly using ugmovie's *move* command,

most all origami also involves some complex folds that cannot be described in this manner. During these complex folds, different vertices of the same polygon rotate around different axes. To model these folds, I have developed a tool called *mkvar* that outputs the appropriate ugmovie *vardef* statements for moving each vertex individually, taking as input the starting coordinates of the vertex and the parameters used to rotate it.

2 Technical Description

The steps involved in creating each file include determining the exact measurements and the topology of the mesh of folds in the final figure, breaking the construction into discrete steps and grouping the polygons together for the different steps in folding, and actually modeling the folds in ugmovie.

2.1 Mesh Analysis

I have used a brute force approach to mesh analysis, employing lots of trigonometry. The figures had a great deal of symmetry that I exploited to make the job easier.

I spent a great deal of time trying to come up with a design of a simple tool that would make this analysis easier, but nothing short of an general-purpose 3-D folder with an extremely complicated user interface seemed to be useful for determining even just the folds for the crane.

2.2 Dynamic Hierarchies

When grouping polygons together during folding, the ideal grouping changes during different stages in construction. For example, to fold a square into four smaller squares, one would first like to group together the two top quarters to fold the top down, and then group together the two left quarters to fold the left half over. However, we only want the top left quarter to be instantiated once. What we really want are dynamic hierarchies of polygons.

We can accomplish this in ugmovie via a hack with the scale function. We instantiate all the different groupings we want, so that polygons appear multiple times in different groups. We then scale all groups except the one we are currently using to subpixel size. To switch groups, we simultaneously scale up the new groups and scale down the old groups. Each visible polygon must appear in the same position in both sets of groups at the time of the transition, to avoid having polygons disappear or appear in the middle of the animation.

2.3 Simple folds

For simple folds, all the polygons on one side of the fold are rotated 180 degrees around the fold line. This is accomplished in ugmovie by first aligning the fold line with the major axis. Then the polygons to be rotated are translated away from the axis slightly, creating a gap that indicates to the user where the fold will take place. The folding action is described by a *move* statement with linear interpolation to avoid expensive *mevals*. If the fold is merely to create a crease, it is then unfolded by rotating it back almost all the way, and the gap is made smaller, indicating a crease.

2.4 Complex folds

For complex folds, different polygons are simultaneously rotating around different axes. This makes it impossible to align the figure so that all of the *move* statements can be applied simultaneously. In some cases, while it is straightforward to find the axes that individual vertices of a polygon rotate around, describing the rotation of the whole polygon is non-trivial. In these cases, the physical model goes through non-planar phases, so a small amount of stretching is acceptable in our model. For these folds, *vardefs* are used to define the motion of individual points, and an *meval* describes the topology and connectivity, which remain the same throughout the fold.

Since these *vardef* expressions can get quite complicated, I developed a tool to output the entire symbolic expression for a rotating vertex. These expressions can then be pasted into a ugmovie file. My *mkvar* program takes translation and rotation parameters and applies them to a given vertex, and outputs the *vardef* for the vertex. The program generates the appropriate matrices and performs actual and symbolic matrix operations, returning an expression in the variable angle of rotation.

3 Tutorial

To start up an origami demo, go to *sara*/285, preferably while logged into ignatz, then to the sub-directory of the demo you wish to use, say the crane, in *crane*/. Type *demo* and wait for the screen to become tiled with ugmovie programs. While you are waiting, prepare a square piece of paper so that you can follow along.

The ugmovie programs should be followed in left-to-right, top-to-bottom order. Starting with the top left window, go to the *Time* menu and choose *sequence*. This will show you the first fold. You can repeat the fold, cycle it continuously, or freeze it by selecting different options in the *Time* menu. Rotating the figure may be necessary when it is not clear from the front view whether folds are towards or away from the viewer. To look at the figure from a different angle, click-drag the middle mouse button in the window to rotate an imaginary crystal ball around the image. To move forwards or backwards, click-drag the middle mouse button while pressing the *Shift* key.

After completing the first fold, choose *sequence* in the adjacent window to see the next fold. Follow each step in turn to fold the completed origami figure.

For simple folds, the line to fold along in the current step is shown as a thick, black line. Creases remaining from previous folds are shown as thinner black lines.

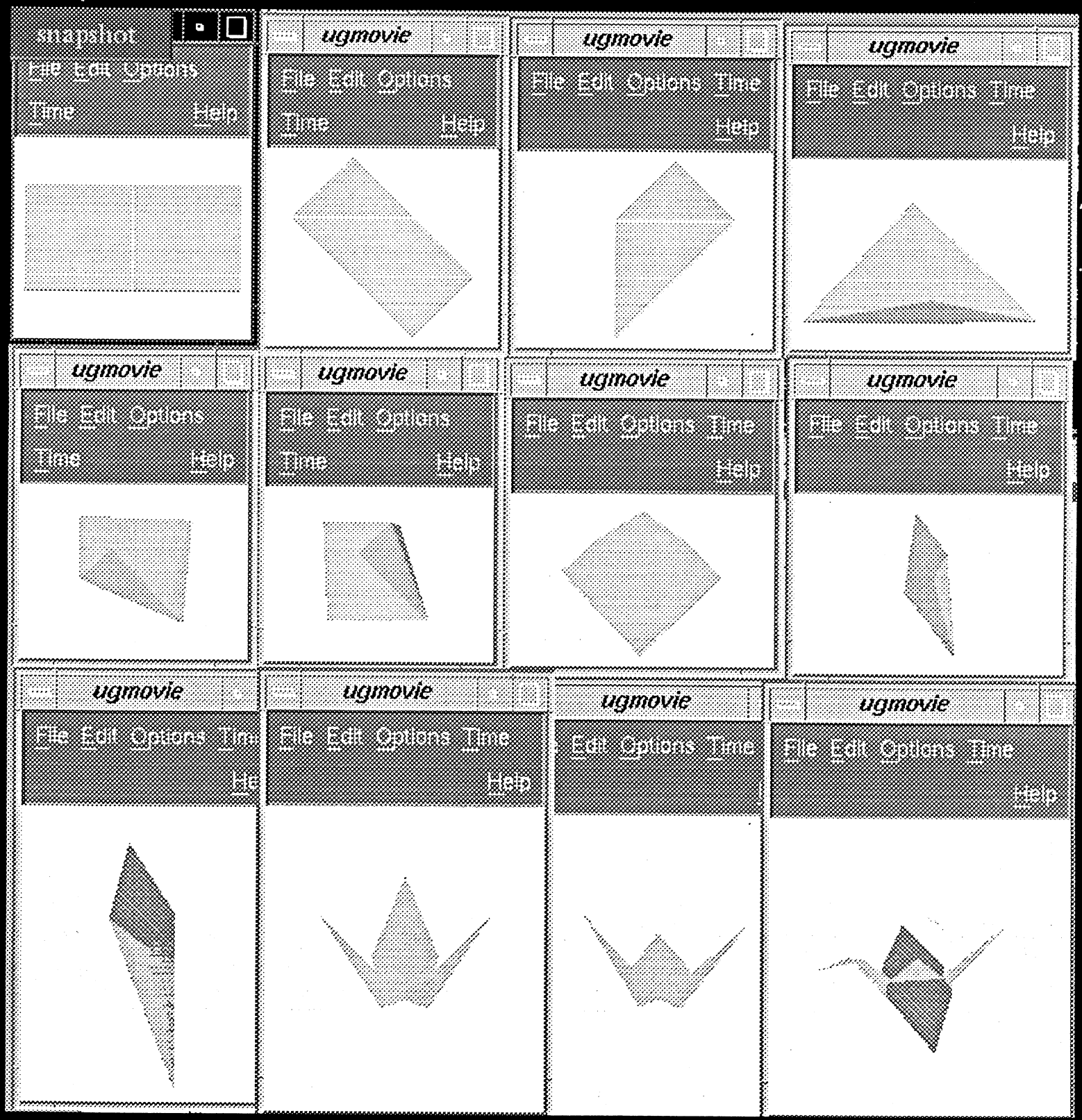
4 Notes on ugmovie

Additional obscure libraries necessary to compile ugmovie can be obtained from ftp.x.org, but hopefully the local versions won't have been deleted next time bug fixes to ugmovie are needed.

5 Conclusion

I have succeeded in creating animations of the constructions of origami creatures that are (hopefully!) easier to follow than the static diagrams that I learned them from. I

have created one tool that can be used to help speed up the creation of additional such demonstrations, and techniques such as dynamic hierarchies for making this process simpler. Unfortunately, much of the work is still very tedious. A general folding tool would go a long way towards relieving that tedium, but it has only been through making these models that I have gotten a good sense of the requirements for such a tool. I very quickly realized that the simple 2-D tool that I once thought would suffice would be inadequate to describe the complexities of the 3-D folds used in origami.


Snapshot
File Edit Options
Time Help
ugmovie
File Edit Options
Time Help
ugmovie
File Edit Options Time
Help
ugmovie
File Edit Options Time
Help
ugmovie
File Edit Options
Time Help
ugmovie
File Edit Options
Time Help
ugmovie
File Edit Options Time
Help
ugmovie
File Edit Options Time
Help
ugmovie
File Edit Options Time
Help
ugmovie
File Edit Options Time
Help
ugmovie
Edit Options Time
ugmovie
File Edit Options Time
Help

NAME

mkvar - vardef generator for ugmovie

SYNOPSIS

mkvar [-f <x y z>] [-x <theta1>] [-y <theta2>] [-z <theta3>] [-l <x y z>] [-a <axis>] [-p <x y z>] [-n <vtx_number>]

DESCRIPTION

mkvar is an accessory program for ugmovie that generates vardefs for rotating points around lines in three dimensions. Inputs are the point to rotate; parameters for transforming the point to line up the axis of rotation with one of the major axes: first an optional translation, then rotations about the x, y, and/or z axes, and an optional subsequent translation; the major axis to rotate around (including direction of rotation); and the vertex number of the point to rotate. Output is in the form of three vardefs for the x, y, and z coordinates of the point as functions of theta, the angle of rotation.

FLAGS

-f <x y z>	Coordinates of first translation
-x <theta1>	Angle of preliminary x-rotation
-y <theta2>	Angle of preliminary y-rotation
-z <theta3>	Angle of preliminary z-rotation
-l <x y z>	Coordinates of last translation
-a <axis>	Axis to perform the variable rotation around. Specify : +/- X as +/- 1, +/- Y as +/- 2, +/- Z as +/- 3.
-p <x y z>	Coordinates of point to rotate
-n <vtx_number>	Vertex number of the point to rotate

EXAMPLES

To take vertex 5, the point (0 -1 0), and rotate it around the line y = -x, no translations are needed, and with a z-rotation of +45 degrees the axis of rotation is X (1). Assuming we want to rotate out of the screen from bottom to top, this is a negative rotation, so we must specify -X (-1) for the axis:

```
mkvar -z 45 -a -1 -p 0 -1 0 -n 5
```

which returns:

```
(vardef x5  ((-0.5000*cosd(-theta)) + 0.5000 ) -10 10)
(vardef y5  ((-0.5000*cosd(-theta)) -0.5000 ) -10 10)
(vardef z5  ( + (-0.7071*sind(-theta))) -10  10)
```

To take vertex 7, the point (0 -1 0), and rotate it around the line y = -tan(22.5)*x + (tan(22.5) - 1), we first translate the point (1 -1 0) to the origin, a translation of -1, 1, 0. Then we rotate around z 22.5 - 90 = -67.5 degrees. There is no translation after the rotation. To rotate around the y-axis, out of the screen from left to right, we specify -2 as our axis:

```
mkvar -f -1 1 0 -z -67.5 -a -2 -p 0 -1 0 -n 7
```

which returns:

```
(vardef x7  ((-0.1464*cosd(-theta)) + 0.1464 ) -10 10)
(vardef y7  ((-0.3536*cosd(-theta)) -0.6464 ) -10 10)
(vardef z7  ( + (-0.3827*sind(-theta))) -10  10)
```

FILES

~sara/285/mkvar/mkvar	executable
~sara/285/mkvar/*	source code

DIAGNOSTICS

mkvar gives a return code of zero unless it encounters a fatal error.

SEE ALSO

ugmovie(1), ug(5)

BUGS

The original points to be rotated should be fairly close to the origin, so that the rotated coordinates never exceed 10 or fall below -10, or ugmovie will complain about variable out of bounds errors.

AUTHOR

Sara McMains

Juggler

Course Project Report, CS 285, Fall 1994
Keith P. Vetter
University of California at Berkeley

Abstract

Juggler is an interactive program that juggles any number of balls with controls provided for the user to adjust the juggling parameters, including the number of balls and the juggling pattern. A powerful juggling model is used allowing for greater realism and flexibility.

1. Introduction

This project is about juggling balls. The goals were two fold. The first goal was to provide an interactive program with the basic juggling motion with user controllable parameters. Once that was achieved, the second goal was to extend it to allow for the fact that humans cannot throw the balls perfectly every time. The model developed to handle this randomness turned out to be an extremely powerful and flexible one. Not only could it handle the variable tosses, but is easily adaptable to juggle in other patterns or to do tricks.

This program is useful on two levels. First, it is fun to see it juggle, and to play with the various parameters. Second, it provides a framework which can easily be extended to incorporate different juggling patterns or tricks.

2. The Basic Juggling Model

The basic juggling pattern is called cascade with the balls being tossed from hand to hand. It can be modeled in two ways: with symmetry or with randomness. The simpler, symmetric model is when every toss and every catch are perfect. This results in an elegant, symmetric cycle that all the balls follow at different phases. The model with randomness, which I call the object model, is more realistic allowing for throws that can be too high, too low, late or early. In this case, there is no symmetry—each ball is independent with its own path and toss timing. This section describes the simpler model to make clear the basic properties of juggling. The next section describes the more powerful and realistic object model.

2.1 The Cycle

The symmetric model is when every toss and every catch are perfect and the balls all follow the same path. There is a four-phase cycle that each ball follows. To see the phases, consider what happens to a single ball:

1) tossed from the left hand to the right hand
2) held in the right hand

3) tossed back to the left hand from the right hand.
4) held in the left hand until it time to throw it again

This cycle consists of two tosses and two catches. If **F** is the flight time of a toss and **H** is the time the ball is held in a hand, then the length of the total cycle is **L** = 2 (**F**+**H**).

This cycle is independent of the number of balls being juggled. It does not matter if you are juggling three balls or fifteen balls, all the balls follow the same path and cycle, all that differs is where they are on the cycle. Because of symmetry, all the balls are equally spaced on the cycle. Thus, for juggling N balls, the offsets on the cycle are:

0*(**L**/**N**) 1*(**L**/**N**) 2*(**L**/**N**) ... (**N**-1)*(**L**/**N**)

The constraint on this cycle is that two balls can never be in one hand at a time. In other words, the hold time **H** must be less than the time between balls **L**/**N**. This yields the constraint: **H** < 2*F/ (**N**-2). The value of **F** is set by the user. The value of **H** has an optimal setting described in section 2.3.

This cycle also explains a little known juggling fact: it is impossible to juggle four balls in the normal, cascade pattern. The problem is that the phase interval between every other ball–2(**L**/4) = **L**/2 = 2(**F**+**H**)/2 = (**F**+**H**)–exactly equals the interval between phase 1 and 3 of the cycle. Thus they get tossed and caught simultaneously and follow mirror image paths. This applies to any even number of balls. Section 4 describes a way to juggle four balls.

2.2 Modeling the ball trajectory

The next problem is modeling the trajectory of the toss. I chose a coordinate system with the origin halfway between the two hands. I let **w** be the distance between the hands and **h** be the height the trajectory reaches. I also assumed that the balls stay in one plane.

When a juggler tosses a ball in the air, it has an initial velocity and direction and gravity acts to accelerate it downward. This results in an inverted parabola. Since the ball takes **F** time to travel width **w** and reach height **h**, we can solve for the equation of the path. Using time **t** as the parameter, the equation of the trajectory is:

u = (2***t** / **F**) - 1
x = **w** * u
y = **h** * (1 - u*u)

Finally, the trajectories from each hand are offset from one another—the point where a ball is caught is outside the point where it is thrown. This offset distance **S** is added to or subtracted from the x coordinate depending on if the throw is from the left hand to the right hand or vice versa.

2.3 Modeling the hand motion

The hands follow their own two phase cycle. The first phase is when the hand is holding a ball. It moves from where it catches a ball to where it tosses it. The second phase is

when the hand is empty. It moves from the toss point back to catch the next ball. The cycle starts anew when the next ball is caught; thus its length is the time between balls: **L/N**. The first phase corresponds to phase 2 and 4 of the ball cycle, and has length **H**. The second phase, thus, has length **E = L/N - H**.

The values of **H** and **E** control how the transfer of balls between hands happens. At one extreme, when **H** is near zero, the ball gets tossed immediately after it is caught. At the other extreme, when **E** is near zero, the ball gets held onto until just before the next ball lands. The most symmetric solution is when the two are equal. This corresponds to what seems to occur in normal juggling—a ball gets tossed halfway between the time it is caught and when the next ball is caught. Mathematically, this occurs when **H = E = L/N - H = (2(F+H))/N - H = F / (N-1)**.

Unlike the balls, the hands have more than just gravity controlling its motion. This implies that it is impossible to claim that any given motion is physically correct. On the other hand, it also implies that many different paths are reasonable. I choose a motion that satisfied four criteria, three physical and one aesthetic. They are:

1. The hands go through the catch point and the toss point.
2. When tossing a ball, the hand velocity must equal the velocity the ball will have.
3. When catching a ball, the hand must sink to absorb the impact.
4. The hands follow a roughly oval pattern.

To satisfy second criteria, I ensure that at one time unit before the toss the hand is as far below the toss point as the ball will be one time unit after the toss. Likewise, one time unit after catching the ball the hand is as far below the catch point as the ball was one time unit before the catch.

Thus the problem is to define the hand motion that is roughly oval and goes through the four points. For that I choose two parabolas. The first is an inverted parabola going from the toss point to the catch point. The second is parabola going from the point one time unit after the catch to the point one time unit before the toss. This is not what the motion a human juggler's hands follow but it is reasonable.

3. The Object Model

The symmetric juggling model produces very pretty juggling patterns. Unfortunately, it is not realistic since no human juggler could ever be that accurate. In order to increase realism I changed to a model to add randomness to the ball motion.

In the symmetric model, five parameters (**N**, **F**, **S**, **h**, and **w**) completely describe the motion of every ball. If you just know these values and where a ball is on the cycle then you knew exactly where that ball is. With balls taking random paths, however, there is no cycle. Each ball has different values for these parameters. Additionally, the hands will not follow a cycle either.

In the object model, each ball or hand keeps track of its own state information. Each ball will have a path with random variations. So it must record if it is in the air or being

tossed, when it was or will be tossed or caught, the height, width and time of the toss, and into which hand it will land. Likewise, each hand keeps track of its state, such as which ball it is holding, when the next ball should be tossed or caught and the velocity the ball should be thrown with. Thus in the object model, each object knows how to position itself. The interesting time is when a ball is thrown or caught because then the state of an object changes.

3.1 Catching and Throwing

After a ball is caught, two values must be updated: when to throw the ball, and what path it should take. In juggling different patterns, as described in section 4, we also must decide where to throw the ball two. In the cascade pattern, however, it is to the other hand.

The constraint on when a ball can be thrown is that it must be thrown before the following ball lands. This interval is no longer fixed so we must check the following ball to see when it lands. As described in section 2.3, a human juggler tends to toss the ball halfway through the interval before the next ball landing. Therefore, we choose a time that varies around the halfway point.

The path a ball takes is constrained by the fact that it must land after the preceding ball lands. Also, balls should be tossed to roughly the same height. If a ball is thrown to height **k*h** then the flight time is sqrt(**k**)***h**. We choose a value for **k** near 1 that is large enough so the ball does not overtake the preceding ball.

Tossing a ball is less complicated, mainly because most decisions were made when the ball was caught. The ball changes from being held to being in the air. The hand changes to empty and we must determine when we have to be back at the catch point to catch the next ball.

4. Other Juggling Patterns

The juggling pattern described in sections 2 and 3 is called cascade. It is the standard pattern but not the only one. This section will describe several variations and show how the object model can handle them.

One simple variation is called overhand. Here, the balls are caught on the inside and tossed on the outside. Except for the hand motion, this is equivalent to normal cascade juggling but with time moving backwards. Overhand juggling can be modeled by setting the trajectory offset value **S** to **-S**. This applies to both the symmetric and the object model.

Another variation is called the shower, which is what people often naively think is the basic juggling pattern. Here, the balls travel in a circle: the right hand tosses the balls high to the left hand which immediately passes it back to the right hand. The object model can handle this pattern with two small changes. Normally, the height the ball gets

tossed is the same for both directions. Instead, we make the ball toss from the left hand to the right hand be low and fast. Also, the trajectory offset value S gets set to zero.

As mentioned in section 2.1, it is impossible to juggle an even number of balls in the cascade pattern. Rather, a different pattern is used which for lack of a better name I call even. In this pattern, one hand juggles half the balls in a shower pattern and the other hand juggles the other half in a shower pattern. To make it look better, the two hands are out of phase. This pattern, while mostly used for an even number of balls, can also be used for an odd number of balls with one hand juggling one more ball than the other. Modeling this pattern is easy. When selecting a new trajectory for ball after being caught, we simply choose one that goes to the same hand.

5. Implementation

Juggler is written in Tcl/Tk using the YART extensions if available. Tcl/Tk provided a good environment but lacks 3-dimensional capabilities. Attempts to move to UG movie and Glide were abandoned because UG Movie does not have a powerful enough programming model, while Glide does not have a powerful enough user interface.

Instead I used an 3-d extension to Tcl/Tk called YART. If the program detects these extensions, then the juggling will be in three dimensions with spheres. If these extensions are not present, then the output is in two dimension with circles for the balls.

6. Further Work

The object model, as it is implemented, has some limitations and room for future work. One limitation is in the distribution of the randomness that gets added to the motion. In the real world, the randomness probably follows a normal distribution around a mean. My model currently only uses uniform distribution.

Additional realism could be added in two ways. First, this program ignores the fact that two balls may collide with one another. Instead of bouncing off each other, they just pass through each other. Second, this model assumes that the hands can be infinitely fast. No matter how quickly a ball lands after another is tossed, the hand will get there in time to catch it. A more realistic approach is to limit the speed and acceleration the hand can do. If it cannot reach the ball in time, the ball should be dropped.

Another extension that would be easy to implement and would allow for more realism is that balls should be able to pass each other in the sequence. If one ball gets thrown high and the next low, the second one may get caught before the first. Currently, when I select a new path for a ball to get tossed on, I ensure that it will land after the preceding ball. The difficulty in removing this constraint is that determining where and when the hand should move to catch the next ball is more complicated. It requires examining every ball as opposed to just the next ball.

7. Conclusion

The basic model was quick and easy to program. It produced a nice program that was fun to play with. The change to the object model was difficult. The result, however, was worth it. I was surprised at how easy it is to adapt the model to handle new patterns and tricks.

The one aspect I would do differently is the user interface. I spent much time porting to UG movie and to Glide, only to decide that neither was sufficient. The former has a good user interface but poor programmability. The latter has a poor user interface but good programmability. In addition, I could only run these programs on old, slow machines that another class was also using. The result was a painful waste of time.

Next time I would not bother with either program. Instead I would have used Tcl/Tk with the YART extensions from the beginning. Tcl/Tk may have problems, especially with speed, and YART still has a few bugs, but it is still better than trying to use the Irises.

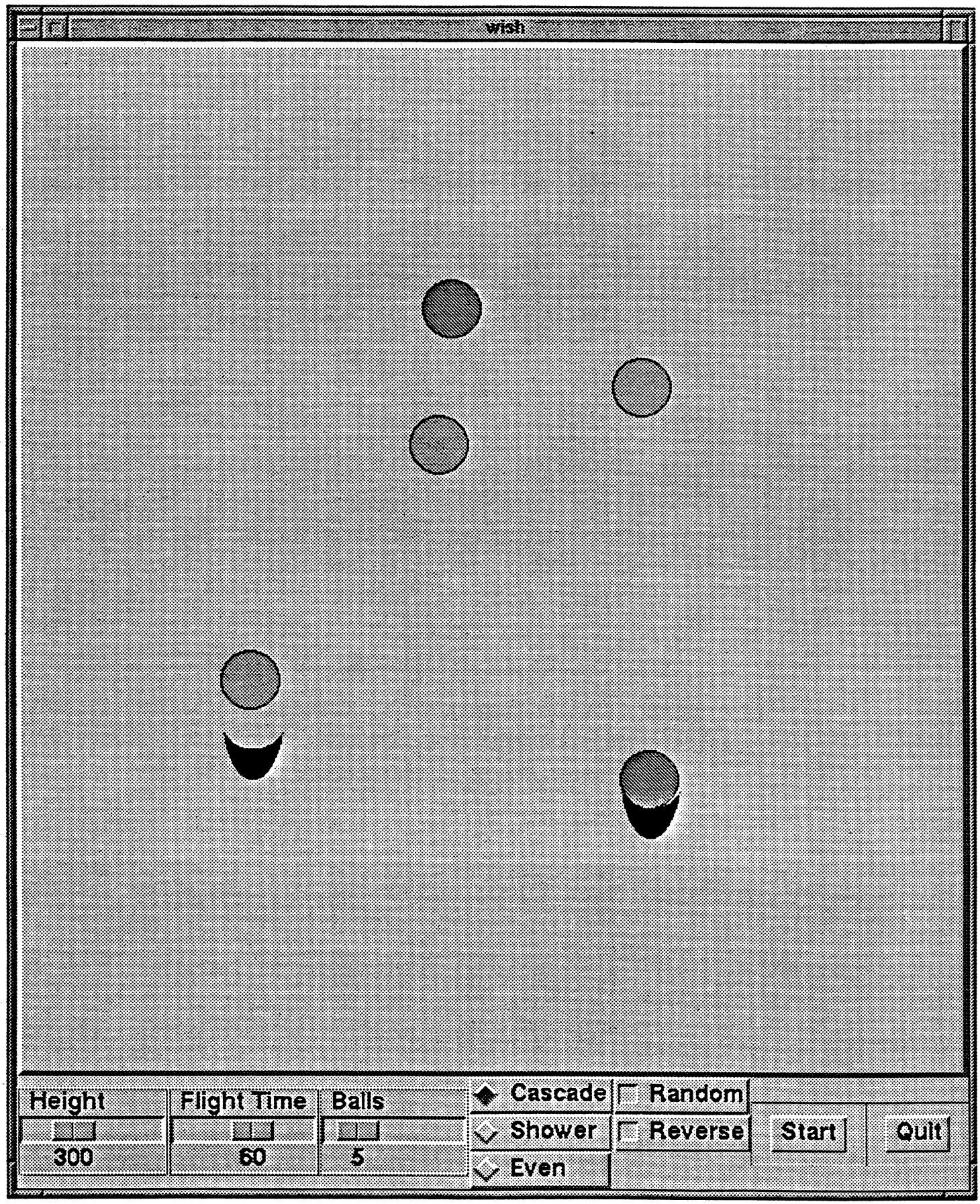


Figure 1. - Screen Snapshot of Juggler

8. Man

Name

juggler — an interactive juggling program

Syntax

wish -f juggler
tkrt -f juggler

Description

Juggler is a Tcl/Tk and YART program that animates juggling of any number of balls in several different juggling patterns. Controls are provided to vary the juggling parameters such as the number of balls to juggle and how high the balls should be tossed.

If juggler determines that it is running with the YART extensions to Tcl/Tk then the objects appear in three dimensions. Otherwise, all the objects are two dimensional.

Controls

Height - adjusts how high the balls get tossed.

Flight - how long it takes for a ball to travel from one hand to the other.

Balls - how many balls to juggle.

Cascade - selects the normal, cascade, pattern of juggling.

Shower - selects the shower pattern of juggling.

Even - selects the pattern used when juggling an even number of balls.

Random - adds randomness to the paths and timing of the balls.

Reverse - selects juggling with time flowing backwards

Start/Stop - button to start and stop the juggling animation.

Quit - exits the program.

Author

Keith Vetter (keithv@cs.berkeley.edu) Fall 1994

Pomylgorph

An Interactive Polygon Morphing Tool

Randy Keller and Seth Tager
CS 285, Procedural Object Modeling
Fall 1994

Abstract

Pomylgorph is an interactive two dimensional polygon morphing tool. The tool displays two input Glide files and allows the user to interactively establish a morphing between them. In addition, the tool can output a dynamic Glide file which displays the user directed morphing.

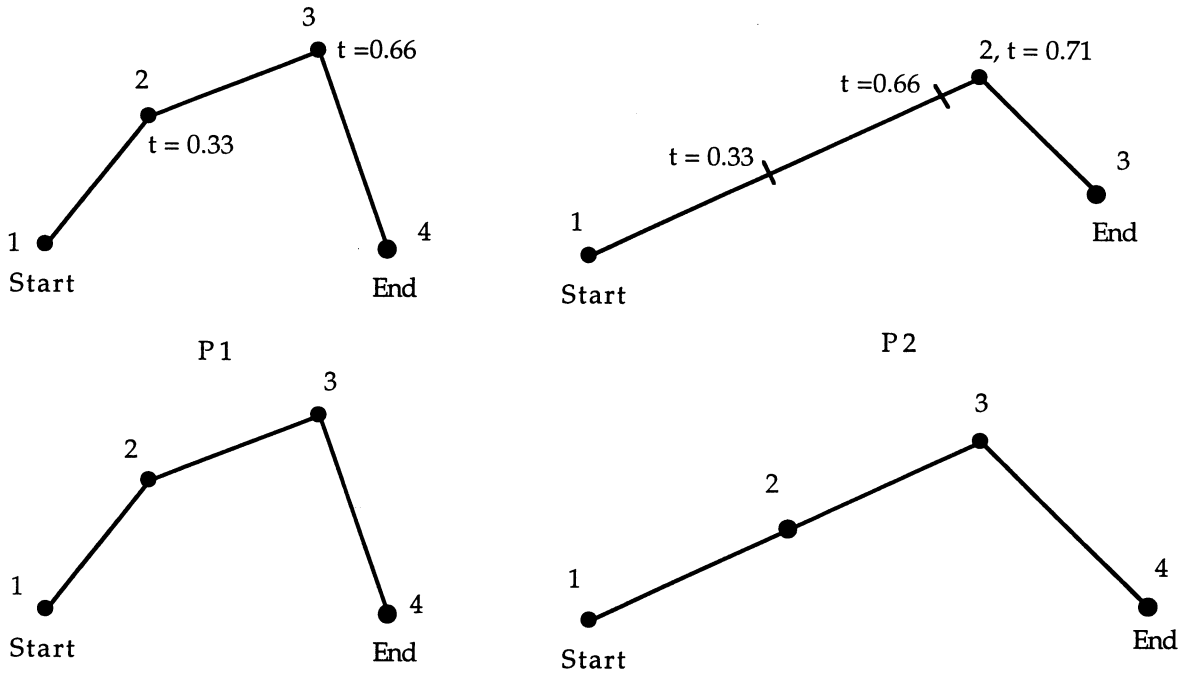
1. Introduction

There are two natural problems which arise in a general solution to two dimensional polygonal morphing. The first is the vertex correspondence problem. The morphing algorithm requires that each vertex on the initial polygon be mapped to a vertex on the final polygon. Vertex correspondence can be established by either adding new vertices on existing edges of the polygons, or by mapping multiple vertices of one polygon onto a single vertex of the other. The second problem is one of determining the path a vertex takes as it moves between its relative positions on each polygon. In developing this tool we looked at segment length parameterization and minimum work methods to solve the vertex mapping problem. As a solution to the path problem, we implemented a turtle graphics method developed by T. Sederberg and E. Greenwood [2].

2. Segment Length Parameterization

In general there are different approaches to what a mapping between vertices actually means. For example, depending on the morphing algorithm which is being used, the initial two polygons might be required to have the same number of vertices. This is clearly the case for a simple linear interpolation between the vertices of the two given polygons. It is also the case for the turtle graphics approach we adopted.

To balance the number of vertices in each polygon, we incorporated a segment length parameterization. The user is allowed to specify regions along the contour of one of the polygons and have these regions map to various other regions on the other polygon. In the event that a different number of vertices are internal to each of these end point pairs a segment length parameterization is performed (see Figure 1).



Final parameterization has only 4 vertices because P1 - 3 and P2 - 2 are within 0.1

Figure 1 Parameterization

The process is relatively simple. We first find the total length of a line segment by adding the lengths of all edges within that segment. A parameter is defined to vary from 0 to 1 along both contours. The parameterized location of a vertex is computed relative to where the vertex lies on its own contour. The relative position corresponding to this parameterized value is found on the second contour. A new vertex is added to the second polygon at this new position. There is one exception to this rule which occurs when a vertex would be mapped to a location very close to an existing vertex on the target polygon. In this case no vertex is added to either polygon. There is no inherent reason to add as many vertices as possible. It seems reasonable that two vertices that are "pretty close" to each other should have a correspondence In addition, this ensures that rounding errors do not produce vectors of zero length in the morphing calculations. By adding a new vertex on the second polygon for each existing vertex on the initial polygon we will generate equal numbers of vertices in each contour.

Although it is simple to implement, this process has a number of drawbacks. Foremost among these is that unless the two contours are identical to begin with, additional vertices are added. This can lead to morphings which distort the original structure of the object. Consider the key frame animation of a walking stick figure. The parameterization will add

new vertices that will not be at the figure's joints. The figure would then bend in the middle of what should be solid sections, i.e. arms, legs, and torso. Since the algorithm lacks any real physical knowledge of the underlying structure of the polygon, the strictly mathematical correspondence of vertices may result in peculiar morphings which lack image coherence. To solve this problem we allow the user to specify multiple parameterization regions in order to give them more control over the morphing, and to restrict the location of new vertices along the contour.

3. Least Work Solution

The least work solution is a second way to establish a correspondence between the vertices of the two polygons. What follows is a summary of an algorithm described in [1]. The algorithm models each polygon as a piece of bent, elastic wire. To form new shapes the elastic segments can be stretched or shrunk, and the angles can be bent. There are costs associated with bending and with stretching. The algorithm matches up vertices from both polygons in a way that minimizes the total amount of work done in transforming the wire from one shape to the next. If the polygons have different numbers of vertices, new vertices will be inserted to make up the difference. To keep the solution manageable and tractable new vertices are only inserted at existing vertices.

Given polygons P1 and P2 and an established vertex correspondence between vertex i on P1 and vertex j on P2, there are three possible correspondences that may follow. Either i -> j+1, i+1 -> j+1, or i+1 -> j. Any other correspondence of i, j, i+1, or j+1 will cause a polygon edge to either bend over itself in reaching its final destination or split apart during the morph. By solving the stretching and bending equations for all three possibilities, you can calculate which choice will require the least additional work. The process of finding the least work solution is simply a matter of starting at the initial vertex of each polygon and repeatedly deciding which correspondence to choose next.

This algorithm has a couple of flaws. Since it only adds new vertices where others already exist it ensures that whenever vertices are added, there will always be a line segment on one polygon that will collapse down to a point on the other. (Or conversely, what looks like one point on the initial polygon will always expand to become an edge of the final polygon)

4. Vertex Path Determination

Establishing a good correspondence between the vertices can have a major impact on the actual path these vertices take in the morphing. The method we chose to implement is discussed at length in [2]. Here I will summarize the algorithm and discuss its limitations.

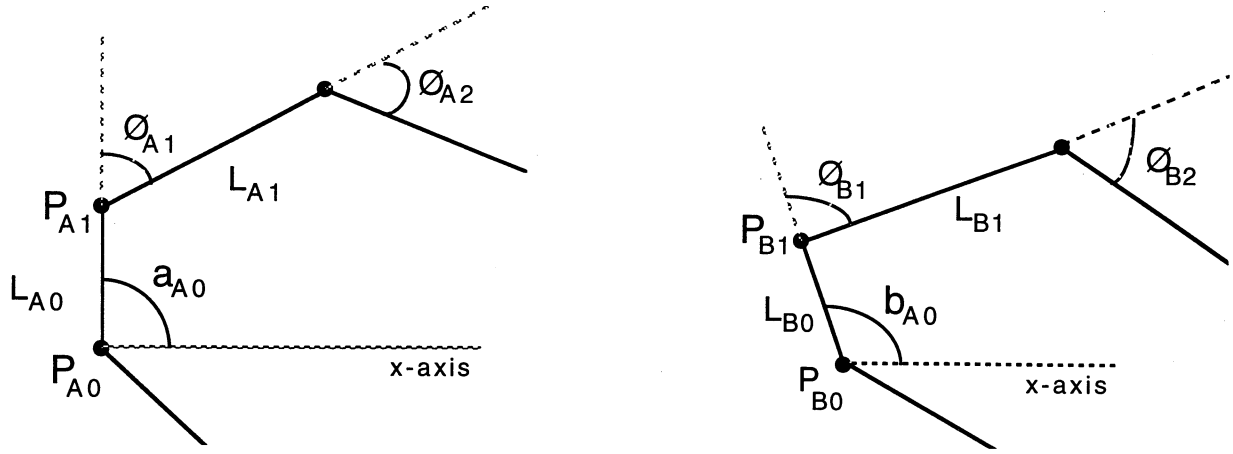


Figure 2 Basic Definitions

The algorithm relies on a turtle graphics specification of the two polygons. Such a definition is comparable to a series of instructions. For example, go forward five units, turn right 30 degrees, go forward three units..etc. The idea behind the algorithm is that given two such descriptions we can interpolate between the distance and turning instructions to generate an intermediate polygon.

(1) anchor angle interpolation $a_0 = (1 - t)*a_{A0} + t*a_{B0}$
(2) angle interpolation $q_i = (1 - t)*q_{Ai} + t*q_{Bi}$
(3) distance interpolation $L_i = (1 - t)*L_{Ai} + t*L_{Bi}$

note: theta in the diagram corresponds to q in the above equation

Figure 3 Standard Interpolation Equations

The equations in Figure 3 specify the angles by which the polygon turns are interpolated and the distances between the vertices (see Figure 2 for definitions). In general however, this method does not produce a true polygon since the end points do not meet (i.e. the contour does not close). In order to force closure at the ends of the polygons we must change equation three to $L_i = (1 - t)*L_{Ai} + t * L_{Bi} + S_i$. The new component S_i is an edge tweaking factor whose value is designed to force the intermediate polygons to close see Figure 4.

$$L_{ABi} = \max\{ |L_{Ai} - L_{Bi}| , L_{tol} \}$$
$$L_{tol} = \text{used in case } L_{Ai} - L_{Bi} = 0$$
$$S_i = -0.5*(L_{ABi})^2 * (l_1\cos(a_i) + l_2\sin(a_i))$$
where: l_1 and l_2 are used to satisfy the constraint of closure,
$$a_i = a_{i-1} + q_i$$

Figure 4 Edge Tweaking Parameters

Altering only the edge lengths can pose problems since the difference between the lengths of corresponding edges is squared. This value can become relatively large for one or two edges, creating a runaway polygon which needlessly enlarges itself in intermediate stages only to shrink again as the parameter approaches one. The computations of the actual x and y coordinates for a given vertex are shown in Figure 5.

$$X_i = X_{i-1} + L_{i-1}*\cos(a_{i-1})$$
$$Y_i = Y_{i-1} + L_{i-1}*\sin(a_{i-1})$$

Figure 5 Vertex Coordinate Equations

5. Output Format

In order to enhance the practicality and usability of our tool we decided to have it read in polygons in the Glide format and to output a complex dynamic Glide file which demonstrates the morphing. Glide is ideally suited for this type of application because actual C code can be embedded in a Glide file and various pre-compiled functions can be called to evaluate the geometry. A number of special concerns had to be dealt with in adapting the code to function for Glide. Of primary importance is out of order execution. Since the various formulas displayed in the figures have dependencies on previous lengths and on previous angles there must be a definite order to the evaluation of vertex positions (see also Figure 5). However, the order of rendering vertices in Glide is not defined and may happen in any arbitrary manner. Therefore we attempted to compute all of the geometry at once for each intermediate before the intermediate is rendered. This would allow the vertices to look up their values for this frame in a special table created during computation. The time to compute a complex morphing was prohibitively long so instead we pre-compute all of the morphings before the first one is displayed and simply store the results in a larger table so that the vertices need only look up their respective positions.

6. Conclusions

Polygon morphing is useful primarily in animation. The unbound nature of the animator's will creates a huge number of diverse scenarios for morphing one polygon into another. Because of this, a general purpose morphing algorithm will fail a great deal of the time. A better solution would be to provide an animator with an interactive tool to generate well-defined morphings of different styles. It is essential to allow the styles to be mixed and matched over different portions of the two target polygons.

This project was an attempt to do just that. We thought that if we allowed the user to specify different vertex correspondences, various

morphing solutions would be easily found. We overlooked the fact that the morphing algorithm we used attempts to preserve segment lengths. In so doing it tends to cause explosive ballooning of intermediate shapes when there are large changes in the angles of corresponding vertices. We might be able to get around this by specifying an intermediate polygon that would serve as a constraint on the more general one, but that compromises the utility of such a tool. The title of this paper illustrates the problem. We wanted to choose a title that would be representative of the paper's topic. One possibility would have been to show an image of the word "polygon" blending into the word "morph." The algorithm we chose, however, was one that maintained the physical structure of each character, while attempting to alter the respective orientations. In choosing a morphing algorithm it is important to pay attention to what polygon characteristics you wish to maintain, because, in the end, the vast multitude of possible morphings will be drastically reduced by idiosyncrasies of the algorithm implemented.

7. References

[1] Thomas W. Sederberg and Eugene Greenwood. A physically based approach to 2-d shape blending. *Computer Graphics* (*Proc.* SIGGRAPH), 26 (2):25-34, 1992.

[2] Thomas W. Sederberg et al. 2-D shape blending: an intrinsic solution to the vertex path problem. *Computer Graphics* (*Proc.* SIGGRAPH), 15-18, 1993.

Pomylgorph Manual

The initial screen of Pomylgorph shows a blank canvas divided in half with buttons at the bottom of the screen for loading polygon files. To load a file, type a name of a legal GLIDE file into the edit widget and hit the "Load File" button. The left side of the screen displays the initial polygon and the right side of the screen displays the target (or end) polygon. The program allows you to adjust the polygons by adding, deleting, and moving vertices, and running the segment length parameterization algorithm on a portion of each polygon.

Manipulating Vertices

The interface is very intuitive but there are a couple of special symbols that you should be aware of. The morphing algorithm requires that there be the same number of vertices on each polygon, and it needs to know what is the initial vertex of each polygon. The initial vertex is circled. You can change the initial vertex by selecting any vertex and choosing "Set Anchor" from the "Vertices" menu.

Selected Vertex - Creating and Deleting vertices

To select a vertex just click on it with Button-1. When a vertex is selected it is enlarged. Only one vertex may be selected at a time. To delete a vertex, select it, and choose "Delete Vertex." To create a new vertex just click anywhere on the canvas, outside of an existing vertex. Vertices can also be moved by selecting and dragging them with Button-1. A selected vertex can be unselected by clicking it a second time.

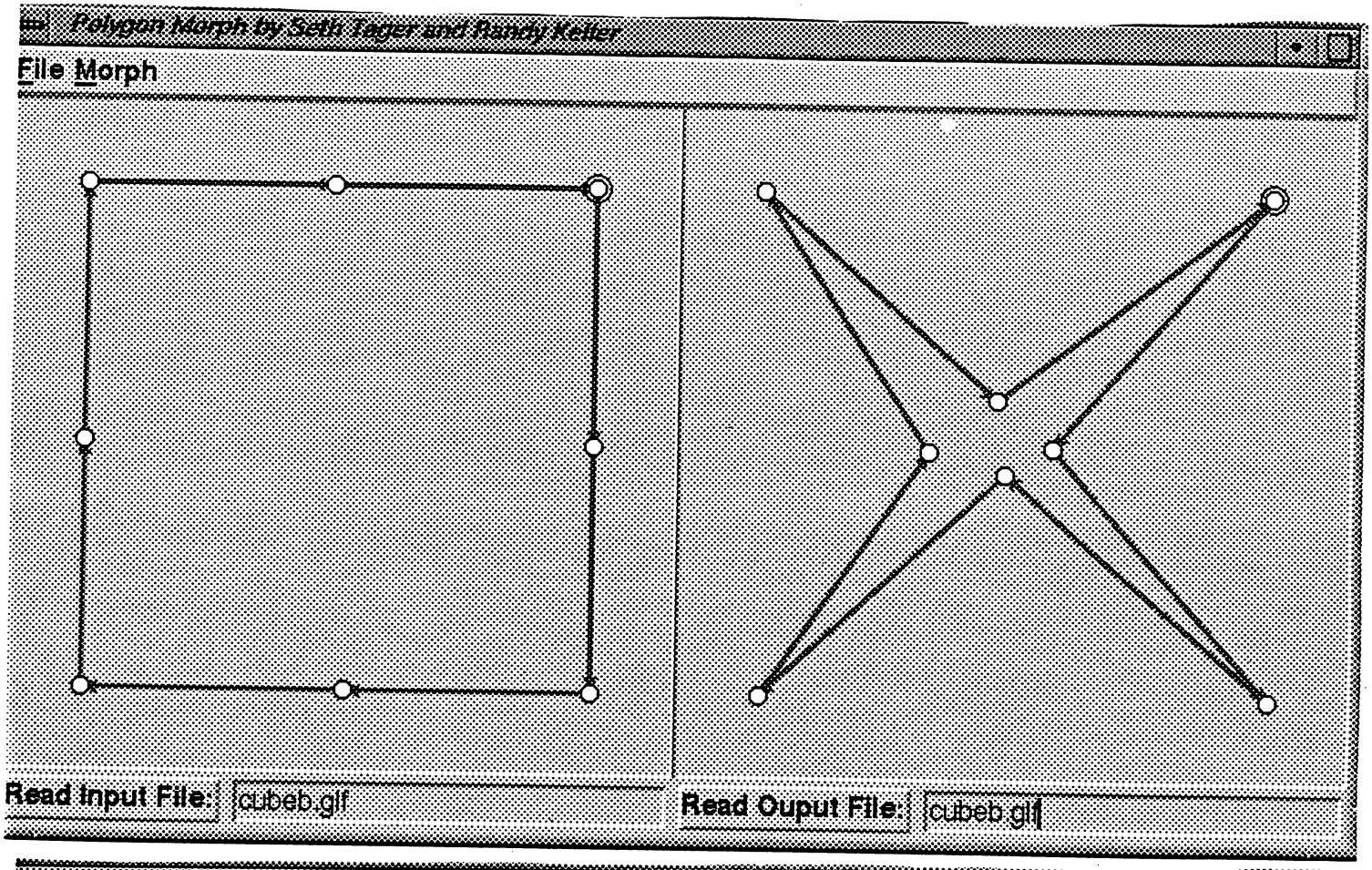
Segment Parameterization

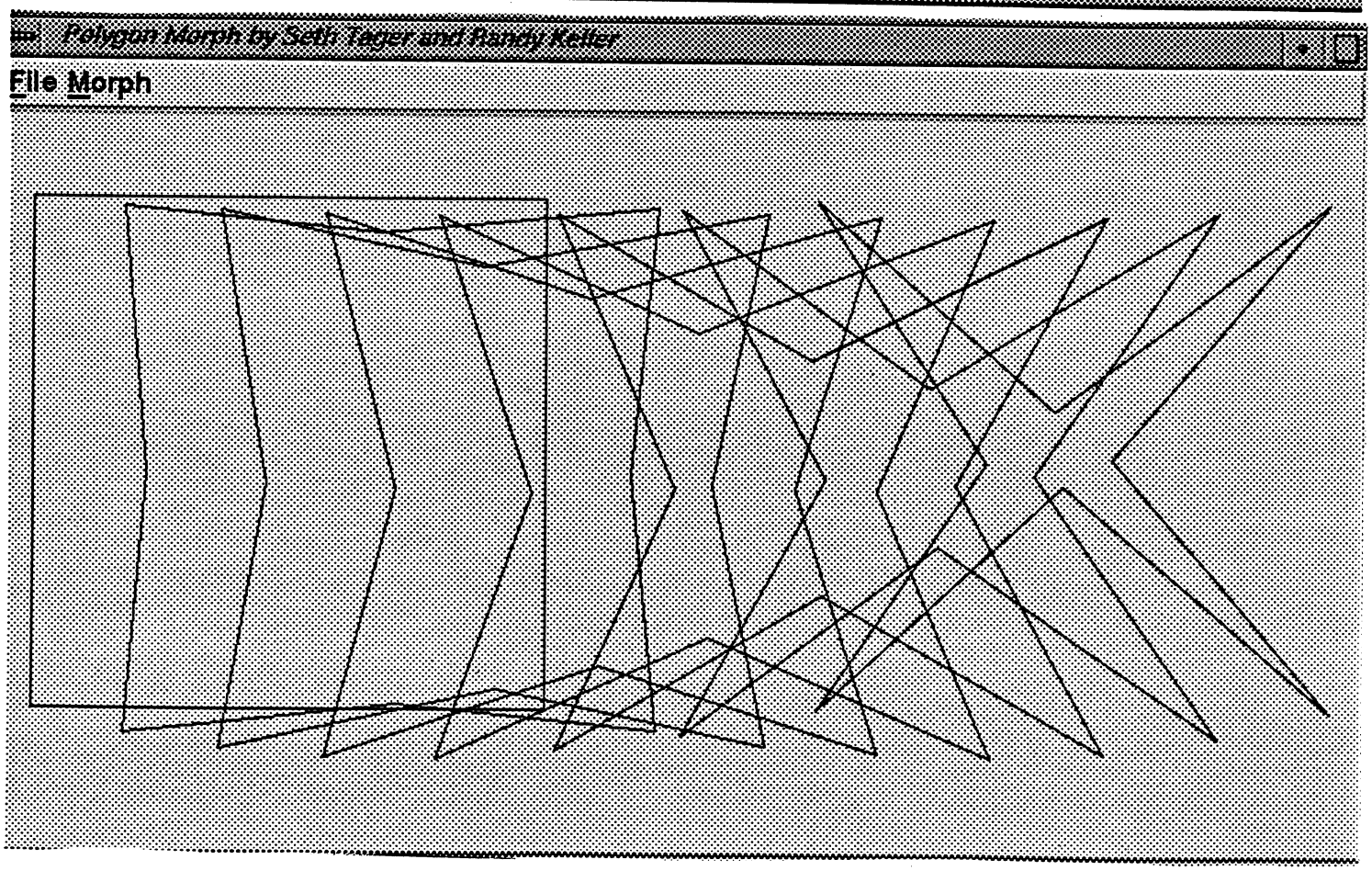
The Parameterization algorithm depends on pairs of vertices from the two polygons being mapped to one another. To map a vertex on one polygon to a vertex on another, simply select each vertex in sequence. If you select more than one pairs of vertices you will notice that the first pair is colored black, and all other pairs are colored gray. The black pair indicates the mapping anchor and all other mappings are taken in order. This prevents you from crossing mappings. To change the mapping anchor, select the vertex you want to be the anchor and choose "Set Mapping Anchor" from the vertices menu.

Once you have mapped vertices between the two polygons you can choose "Parameterize" from the Morph menu. New vertices will appear on each polygon. If you don't like what you see, choose "Undo Params" from the Morph menu and you can start over.

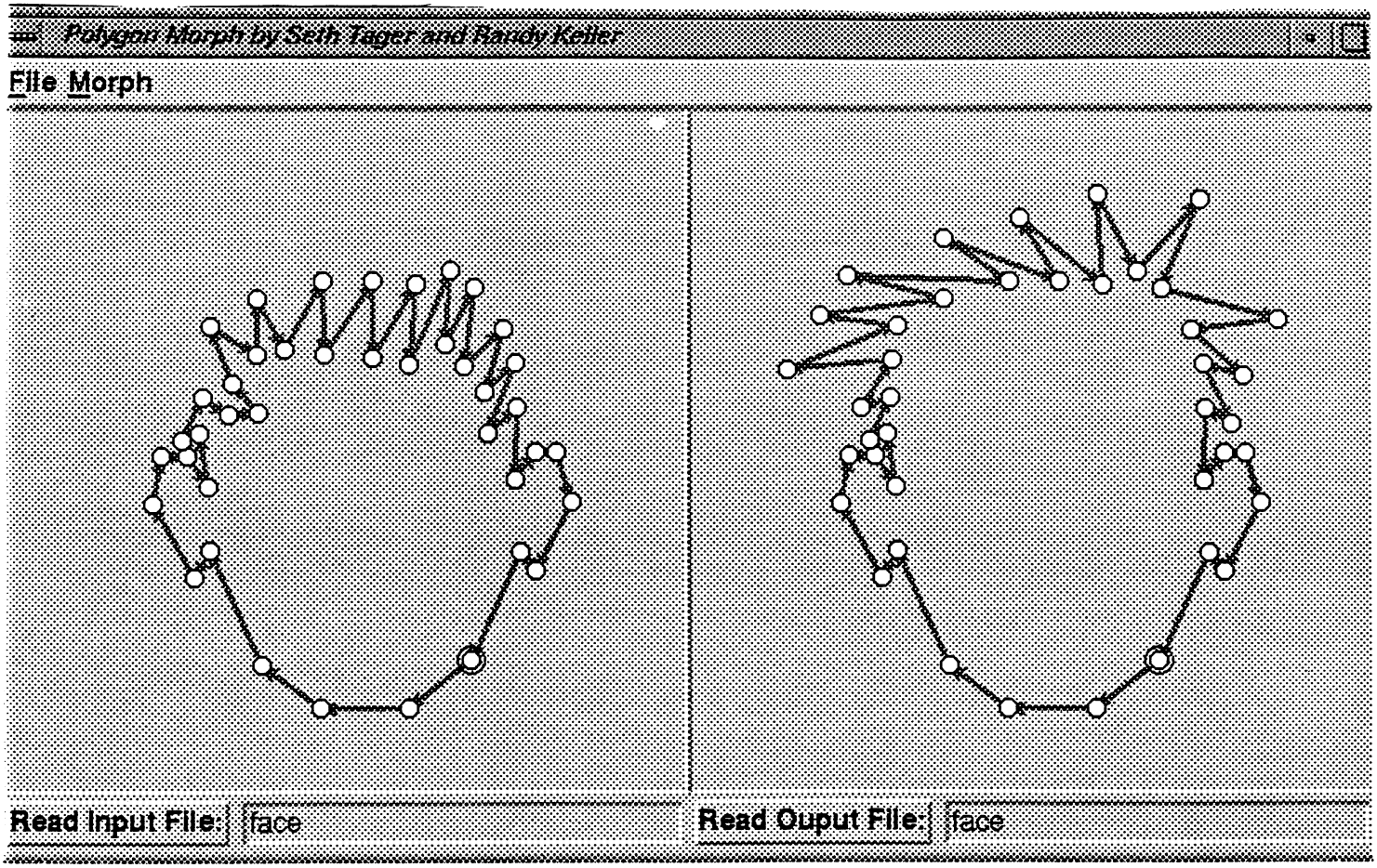
Morphing

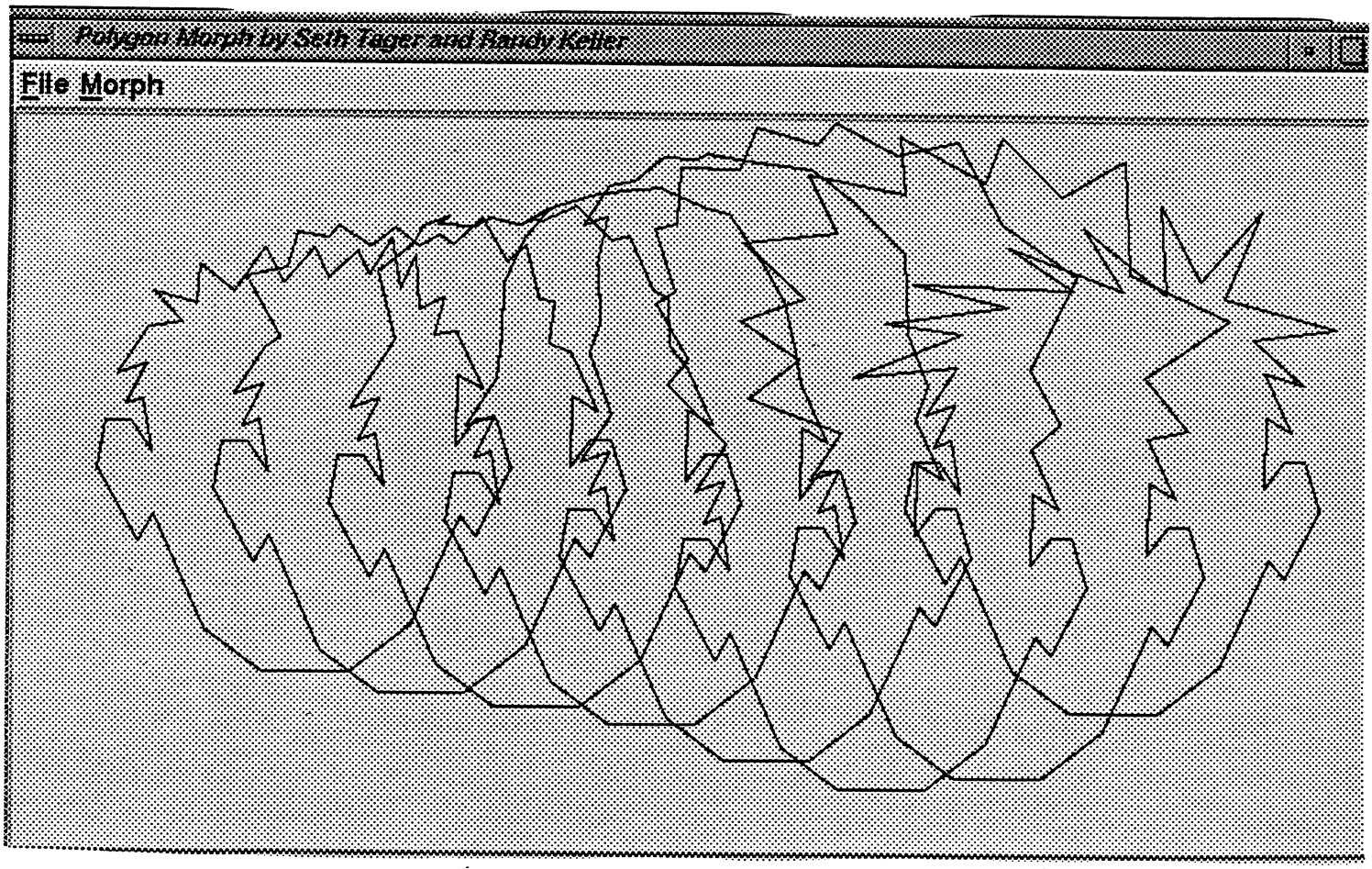
When you are satisfied with the vertex correspondences you can choose "Show Morph" from the Morph menu. Make sure that each polygon has the same number of vertices. The start and end polygons will be replaced with a series of polygons, one representing each frame of the morph. A frame scale will also appear, allowing you to animate the morph by dragging the slider to change the frame. To see all frames at once, choose "Show All Frames" from the Morph menu. When you are done looking at the morph, choose "Show Polygons" and you will return to the polygon screen where you can make changes and try again.





Sample Output of a Square to a Star





A Change of Hairstyles

PGL: Plant Growth Lab

Course Project Report, CS285, Fall 1995

Peter Liu & Boris Vaysman
University Of California at Berkeley

Abstract

Plant Growth Lab (PGL) is an interactive tool for modeling of plants and animation of their development based on the differential L-system (dL-system) framework. It is a complete development environment that includes a plant designer interface for specifying plant structures, an interpreter for context-sensitive dL-grammars, extensible libraries of shapes and growth functions, and MPEG movie recording and playback capability.The system also provides various rendering modes such as wireframes and ray-tracing.

1.0 Motivation

Static procedural models of plants have received much attention in the computer graphics community over the last few decades. Such models are usually based on parallel rewriting grammars first introduced by Lindenmayer in 1968 [1]. Alvy Ray Smith proposed L-systems as a tool for synthesizing realistic images of plants and pointed out the relationship between L-systems and fractals [2]. L-systems give rise to an efficient data representation characterized by a high database amplification factor. They also suggest an insight into the processes determining the formation of the plant structure. Until very recently, however, much of the effort has been devoted to formalizing, representing, and visualizing the underlying structure rather than the temporal behavior of the developing system. For this project, we have implemented a tool for modeling of continuous temporal development as well as discrete structural changes of plants.

Realistic visualization of the plant developmental processes has several important motivations. The animation of processes that could not be observed in real life due to their long duration may assist in the analysis of plant physiology. The interactive access to model parameters and immediate feedback make model validation faster and less error prone. The resulting plant models provide a valuable educational tool. An interactive tool for modeling of plants and simulating their development would be valuable for biologists in describing sequencing of stages in plant development and overall plant geometry.

2.0 Mathematical Framework

The key component of PGL is the smooth integration of discrete and continuous aspects of plant development. The original L-systems provide a good formalism for specifying qualitative changes. The modeled structure is represented as a finite collection of modules each of which could be in a finite number of states. An extension, called parametric L-systems increases the expressive power of L-systems by introducing a continuous element into the states [5]. The structure of the system is also modeled by a set of modules. Each module could be represented by an identifier denoting its type and a state vector of parameters describing the temporal behavior of the module. A(2, 3.5) is an example of a simple module A with the state vector {2, 3.5}. The interpretation of individual parameters in the state vector is determined by the semantics of the module.

dL-grammars, a formalism adopted for PGL, extend parametric L-system by attaching a set of differential equations describing the temporal behavior of the state vector to each module. dL-systems have a dual discrete/continuous nature. As long as the module state vector remains within a certain domain, the behavior of the module is described by a fixed set of differential equations and the module develops in a continuous fashion. Once the state vector reaches the boundary of the domain, an L-production is applied resulting in a qualitative state transition. New modules are generated as a result of such applications. A predefined interpretation is assigned to each of the modules, and resulting structures are visualized using a turtle interpretation of the strings.

A dL-grammar for a developing dragon curve is given in Table 1.

TABLE 1. dL-grammar for the dragon curve. Initial string Fr(1,1), k=sqrt(2)/2

Module	Condition	Action
$F_r(x,s)$	x<s	solve dx/dt=s/T, ds/dt=0
	x=s	produce $z(-45)F_r(0,sk)z(45)F_h(s,s)z(45)F_l(0,sk)z(-45)$
$F_l(x,s)$	x<s	solve dx/dt=s/T, ds/dt=0
	x=s	produce $z(45)F_r(o,sk)z(-45)F_h(s,s)z(-45)F_l(o,sk)z(45)$
$F_h(x,s)$	x>0	solve dx/dt=-s/T, ds/dt=0
	x=0	produce ε

Modules in the given grammar have the following semantics: F_r, F_l, and F_h are line segments with length determined by their first parameter, z(x) rotates the turtle by x degrees around z axis. The constant T determines the lifetime of the module. Differential equations in the Action column determine the speed with which module parameters change. Picture Panel 1 shows the developing dragon curve based on the given grammar.

The example above illustrates a simple context-free case: production application depends only on the module to be rewritten. PGL also handles a more general context-sensitive case in which the production to be applied depends also on the left and/or right context of the module under consideration.

The expressive power of dL systems could be greatly increased if we add to the grammar the ability to record its states at some intermediate points and later access them.. Bracketed strings are used to visualize branching structures. The semantics of opening and closing bracket is pushing and popping the turtle information at the corresponding points of string interpretation.

In order to simulate plant development fully, it is necessary to provide a mechanism for changing the shape as well as the size of surfaces in time. The approach we adopted is to trace surface boudaries using the turtle and fill the resulting polygons. This allows us to use the same dL-grammar formalism for surface specification as well as for specification of the plant structure. Polygon vertices are specified by a sequence of turtle positions marked by the dot '.' symbol. A reserved module '<' marks the beginning of surface specification, '>' the end. A parametric grammar bellow uses this convention to describe a family of leaves from Picture Panel 2. Picture Panel 3 depicts a leaf in various stages of its development.

TABLE 2. dL-grammar for a family of leaves.Initial string <.A(0)>.

Module	Condition	Action
A(t)	1	produce L(LA, RA)[z(-60)B(t).][A(t+1)][z(60)B(t).]
B(t)	t>0	produce L(LB, RB)B(t-PD)
L(s,r)	1	L(s*r,r)

L(x,s) specifies a line segment of length x. The relationship among constants LA, RA, LB, RB, and PD determines the shape of the generated leaf.

The Appendix contains a tutorial descrbing the syntax of dL-grammars and major parts of PGL user interface.

3.0 Implementation

PGL is implemented on top of a Tcl/Tk based object-oriented graphics package called GOOD. GOOD has the full capability of Tcl/Tk. In addition, it provides a pixmap display and the graphics functionality of GL, allowing PGL to render objects in 3D. PGL could be ported across different architectures and can be compiled with a variety of GL-compatible libraries. Our current implementation runs on Sparc 10 and uses VOGL for rendering purposes.

PGL consists of six major components: user-interface, module compiler, rewriting engine, plant renderer, system libraries, and MPEG movie codec. Figure 1 shows how each component interacts with one another. The following sections will describe each in detail.

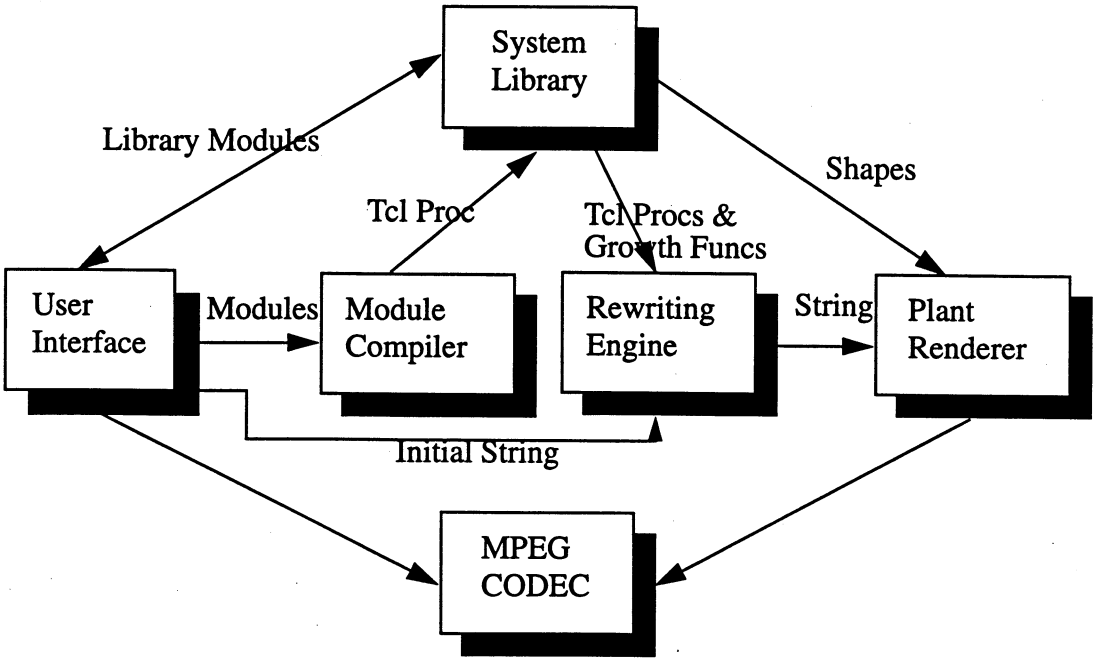


Figure 1 System Architecture

3.1 User-Interface

The UI component of the system is consisted of the following parts: display panel, plant designer, and movie player. Refer to Picture Panel 1 for the different panels mentioned in the following sections.

3.1.1 Display Panel

The display panel provides users with a pixmap display for viewing plant development and VCR controls for stopping, stepping, playing and fast forwarding the development of the plant. It also has controls for recording movies in MPEG format using the Berkeley MPEG encoder for real-time playback. In addition, the user can use the mouse to interactively change the viewing positions. The rendering mode can also be set to wire-frame or ray-tracing.

3.1.2 Plant Designer

The plant designer is a form-base input panel for specifying the dL-grammar of a plant. The user first types in the *Seed* entry box a string from which the plant should grow and then specifies the modules of the grammar in the *Module Panel*. The module is specified by filling out condition-action pairs in the Condition-Action table. The conditions are basically the boundaries conditions. The actions are either solving differential equations or applying a production. The differential equation is applied to each parameter of the module. For 1L and 2L context-sensitive grammars, the user can specify the left/right context in which the module should be evaluated. The user can also specify a predefined shape

such as a cylinder by clicking on the surface specification icon (upper left hand corner of the *Module Panel*). The *Shape Input Box* will then appear.

The plant designer is user-friendly in the sense that it figures out for you the modules that need to be filled out and discards the ones that are no longer needed. The required modules are list in the *Module Listing* panel. It also allows the user to archive grammars describing parts of a plant such as leaves that can later be retrieved and used in a different plant. The *Module Library* panel is a listing of the currently retrievable shapes. The plant designer also provides user with a collection of growth functions and predefined shapes such as line, cylinder and sphere, etc. They are listed in the *Shape Library* and *Growth Library* panels. All of which can be retrieved at the click of a button.

3.1.3 Movie Player

The movie player (not shown) allows users to playback MPEG movies that are previously recorded. This feature is necessary for real-time viewing of the development of the plants since growing a complex plant can be extremely slow. The player is based on the Berkeley MPEG player.

3.2 Module Compiler

The module compiler takes the input from the plant designer and dynamically compiles it into Tcl procedures. These procedures are later used by the rewriting engine for evaluating how a plant should elongate (evaluation of the differential equations) or branch (application of a production). The compilation is done as follow:

1. All user input is first converted into Tcl lists.
2. The name of the module plus the names of the left and right context modules are concatenated together to form the name of the Tcl procedure. The argument list is formed in the same manner.
3. For each condition-action pair, generate an if-statement with the condition copied verbatim (variables are prepended with dollar signs).
4. If the action is solving differential equations, a new variable is created for each parameter of the module and set to the result of the growth function. The module name plus the new parameters are concatenated to form a Tcl list and returned.
5. If the action is applying a production, a Tcl list representing the production string is returned.

The decision to do procedure synthesis instead of procedure interpretation is to take advantage of the Tcl interpreter and its built-in hash table.

3.3 Rewriting Engine

The rewriting engine is basically a tight loop that scans the current Tcl list representing the grammar string and calls the necessary Tcl procedures when modules are encountered. A

new Tcl list is created by concatenating all the non-module symbols and the lists returned by the Tcl procedures. For context-sensitive cases, the engine scans the left and right neighbors of the current module and concatenates their names and arguments to form the name of the Tcl procedure to be called.

3.4 Plant Renderer

Although the plant renderer should be a separate component, it is currently embedded in the rewriting engine so we do not have to rescan the Tcl list that should be of considerable size. The basic function of the renderer is to compute the current turtle positions and create the necessary graphical objects. It is capable of rendering the plant in modes such as wire-frame and ray-tracing.

3.5 System Libraries

The system libraries are basically a collection of user-defined shapes, predefined shapes such as line, cylinder and spheres, growth functions and the dynamically generated Tcl procedures. The plant designer provides easy access to the first three items.

3.6 Movie CODEC

The movie recording and playback component is based on the Berkeley MPEG encoder/decoder. The recording is done by grabbing frames from the pixmap display and MPEG encoding them to save disk space. The playback is accomplish by spawning the Berkeley MPEG player.

4.0 Conclusion

PGL is implemented in Tcl/Tk and C/C++. The user interface and rewriting engine are written in Tcl. Turtle manipulation routines are written in C. Rendering is implemented in C and C++. The system is currently running on Sparc 10, but could be easily ported to various architectures such as HP, SGI, DEC. Good interactive performance for wireframes rendering is achieved for moderately complex plants. However, performance degrades rapidly as the complexity of the modeled structures increases.

Performance could be improved by reimplementing some components in C. In particular, we expect reimplementing the rewriting engine in C should increase interactive performance. We consciously chose to code this part of the system in Tcl to save time and to avoid implementing yet another interpreter. We estimate that by adopting this approach we saved about two weeks of development time.

Having implemented and played with the system, we realize the great expressive power of dL-grammars and the fact that the same framework could be used outside the domain of plant modeling. We believe that PGL could be used as a general purpose animation tool. However, specification of dL-grammars requires a certain level of expertise. Also a care-

lessly designed dL-grammar can lead to exponential time and space complexity making the system impractical for real-time animation.

Bibliography

[1] A. Lindenmayer and H. Jurgensen. Grammars of Development: Discrete-state models of the growth, differentiation and gene expression in modular organisms. In G. Rosenberg and A, Saloma, editors, Lindanmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology, pages 3-21. Springer-Verlag, Berlin, 1992

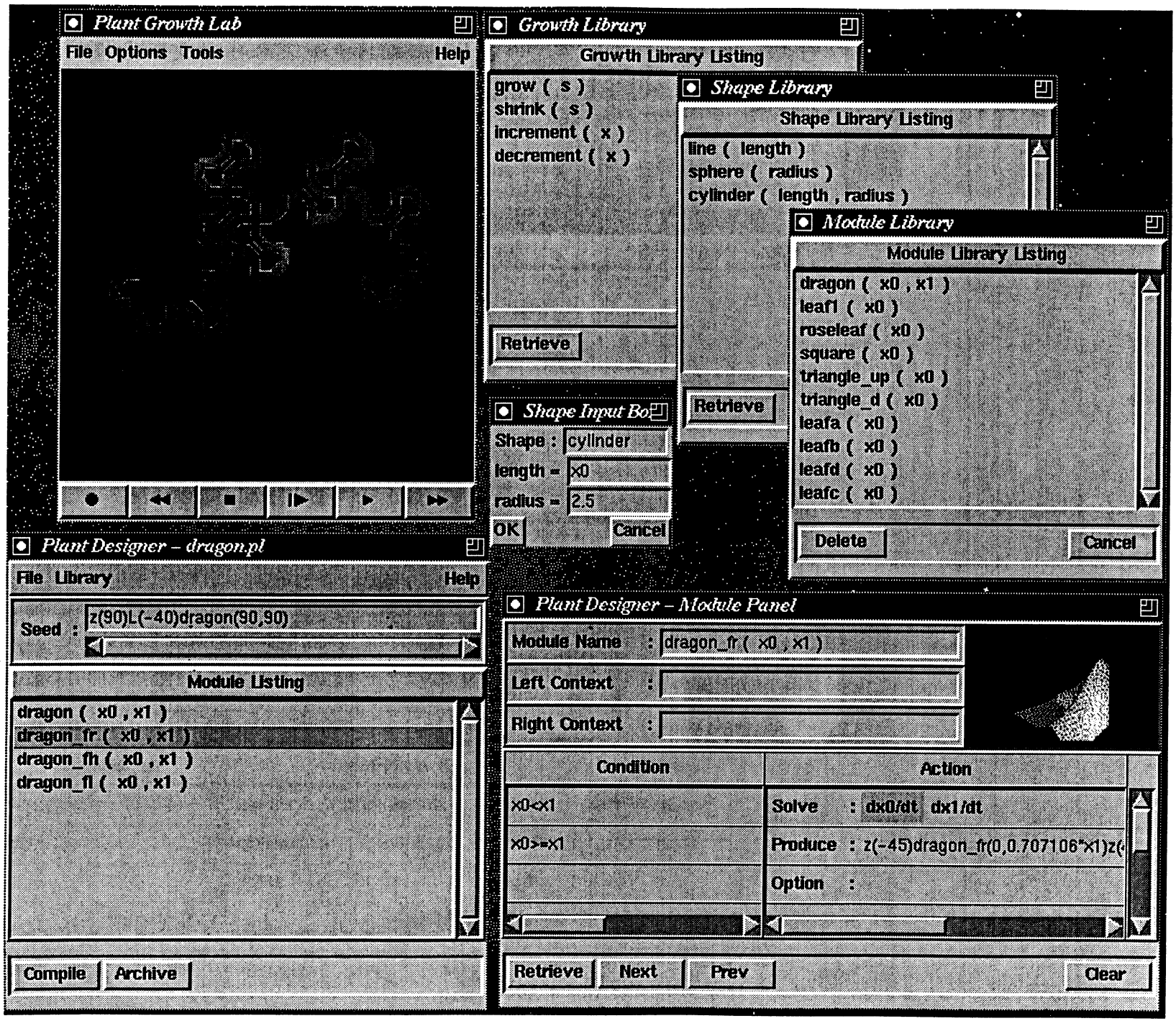
[2] A.R. Smith. Plants, fractals, and formal languages. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 22-27, 1984) in Computer Graphics, pp 1-10.

[3] Przemyslaw Prusinkiewicz, et. al. Animation of Pland Developnet. SIGGRAPH '93 proceedings, pp. 351-360

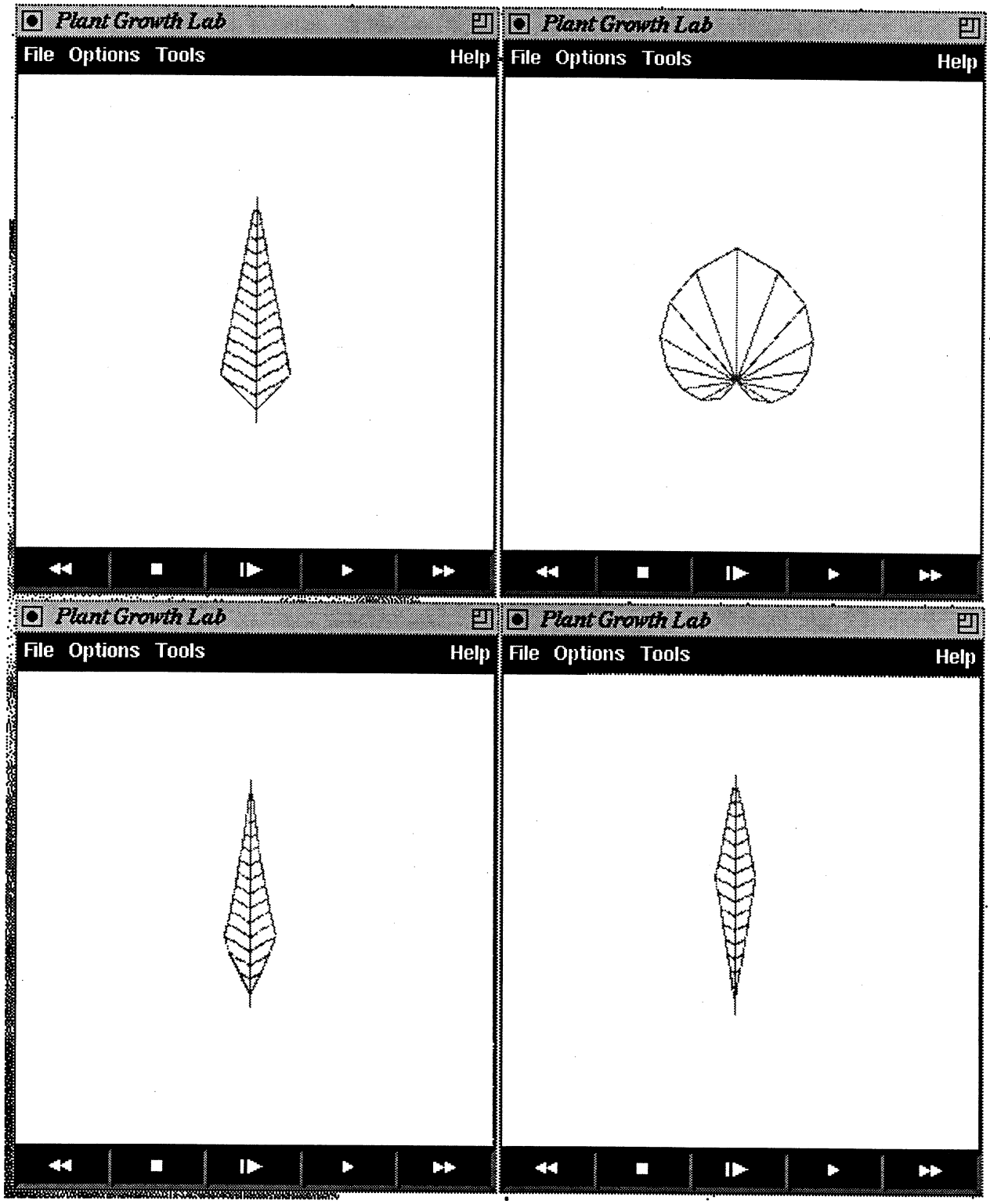
[4] Przemyslaw Prusinkiewicz, et. al. Synthetic Topiary, SIGGRAPH '94. pp.351-358

[5] Przemyslaw Prusinkiewicz, Aristid Lindenmayer. The Algorithmic Beauty of Plants. pp.40-46. Springer-Verlag, 1990

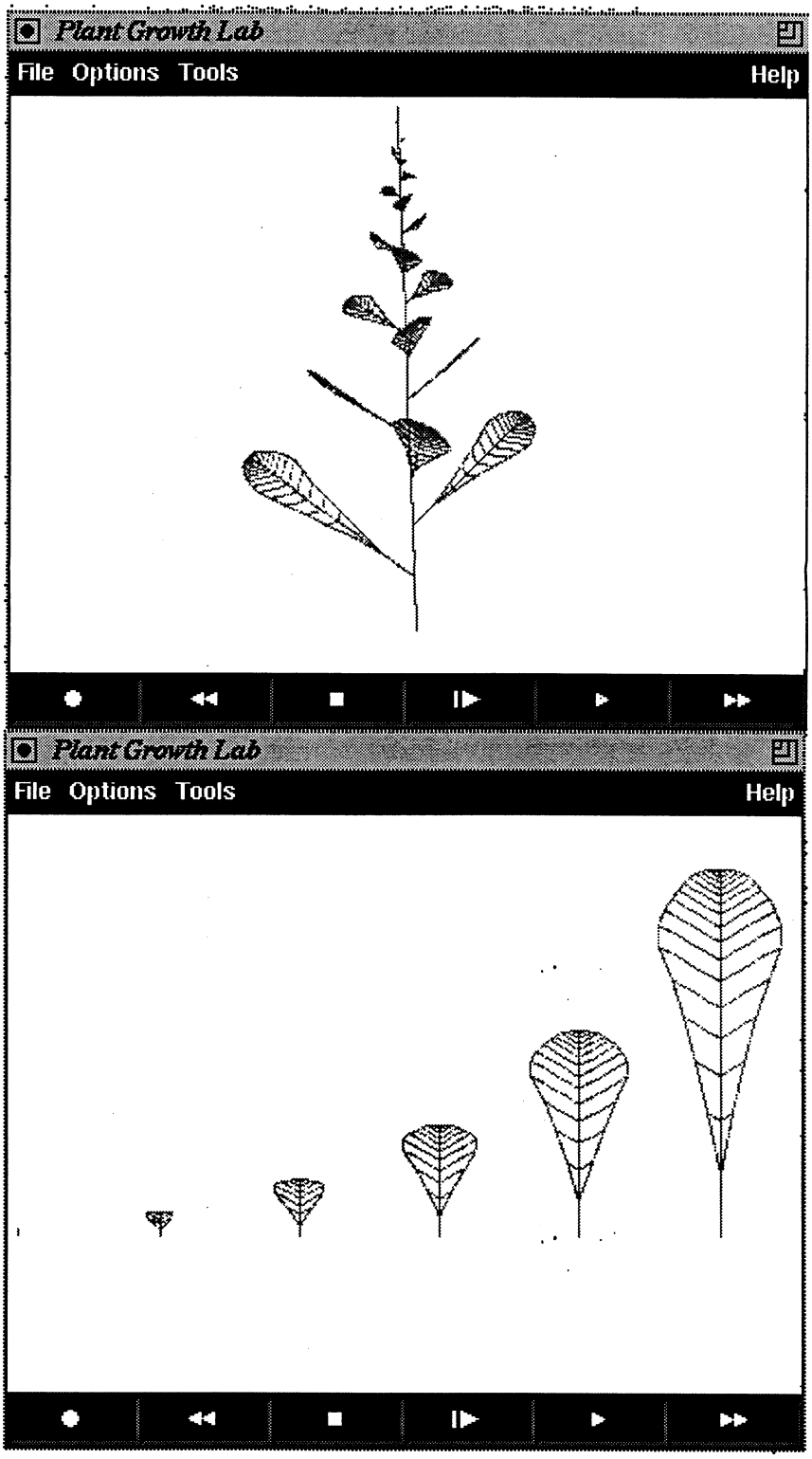
[6] Przemyslaw Prusinkiewicz, James Hanan. Lindenmayer Systems, Fractals, and Plants. Springer-Verlag, 1989



Picture Panel 1



Picture Panel 2



Picture Panel 3

Appendix: A Tutorial

In this tutorial, we will learn the basic functionality of the user-interface component of PGL. In particular, the display panel and the plant designer will be discussed in detail. Please refer to Picture Panel 1 for all the panels mentioned in the following sections or start up the pgl program. Note that the movie recording capability has not been incorporated into PGL at the time of writing of this tutorial.

1.0 Display Panel

The display panel is the window with the black pixmap display and VCR controls. It is the first window that comes up when you start up pgl.

1.1 Menus

The **File** menu has controls for loading a plant and exiting the program. Selecting the **Load** command, a file selection panel will appear. All plant grammar files end will .pl extension.

The **Options** menu allows user to set the rendering mode and reset the viewing position. The rendering modes are wire-frame, flat, gouroud and ray-tracing. Not that flat and gouroud shading are not currently implemented.

The **Tools** menu allows user to start up the *Plant Designer* and *Movie Player.*

The **Help** menu displayed the help pages.

1.2 Coordinate System

The display uses a left-handed coordinate system with the positive z-axis pointing into the screen. The user can use the mouse to change the viewing position by holding down the mouse button and dragging the mouse while the mouse is in the pixmap display. **Left** mouse button controls rotation of the plant around the x and y axes. **Middle** button controls translation along the x and y directions and the **right** button translates in the z direction. The user can reset the viewing position by going to the **Options** menu and choosing **Reset View** option.

1.3 VCR Controls

The VCR controls allows the user to step, play and fast forward the plant. In addition, the user can record a particular plant development into a MPEG movie. The VCR buttons are, from left to right, record, rewind to beginning, stop, step, play, and fast forward. For example, to view the development of a plant, press the **Play** button. The plant will slowly

start to grow. To stop the development, pressed the **Stop** button. To rewind to the beginning, press the **Rewind** button.

2.0 Plant Designer

The plant designer is a form-based input panel for specifying the dL-grammar of a plant. It is consisted of two panels: *Program Panel* and *Module Panel.*

2.1 Program Panel

2.1.1 Menus

The **File** menu allow user to load a plant as discussed in Section 1. It also allows user to create a new plant and save the current plant being edited.

The **Library** menu loads the *Module Library*, *Shape Library* and *Growth Library* panels.

2.1.2 Seed Entry and Module Listing Panel

The user first type in the initial string in the **Seed** entry box in the *Program Panel* and press enter. The user can double-click the left button in the **Seed** entry box and a text input panel will appear and user can type in the string. A listing of the modules that need to be specified will appear in the *Module Listing Panel.* By double-clicking the left mouse button on a entry, the selected module will be displayed in the *Module Panel* (see Section 2.2)

2.1.3 Buttons

The **Compile** button tells the Module Compiler to recompile the plant grammars into Tcl procedures.

The **Archive** button allows the user to archive a plant grammar such as a leaf that can be later retrieved. Note that the seed can only contain one module because the archiver uses the seed as the name of the library module.

2.2 Module Panel

2.2.1 Module Headings

If the module has not been filled out, only the name of the module will appear in the *Module Name* entry box. The user can fill in the names of the left/right context modules for context-sensitive grammars. Next, the user need to fill out the Condition-Action table discussed next.

2.2.2 Condition-Action Table

In the condition entry, the user types in the boundary condition such as x<y. In the action entry, the user first click on the **Options** button and select **Solve** or **Produce**. If action is solve, a list of buttons of the form d*/dt will appear. By click on the button, the *Growth Function Input* panel will appear. The user can type in the desired growth function or click on the *Growth Library* panel. The parameters to the function will appear. If the action is produce, the user can type in the production or double click the left mouse button in which the text input panel will appear.

2.2.3 Predefine Shape

The user can specify a predefined shaped by click on the picture on the upper right hand corner. A *Shape Input* panel will appear. This panel works the same way as the *Growth Function Input* panel.

2.2.4 Buttons

The **Retrieve** button allows the user to retrieve an archive library module.

The **Next** and **Prev** buttons cycle through the existing modules.

The **Clear** button clear the *Module Panel.*

BINARY

pgl - An interactive tool for modeling of plants and animation of their development.

SYNOPSIS

pgl

DESCRIPTION

pgl is a complete development environment for animation of plant growth that includes a plant designer interface for specifying plant structures, interpreter for context-sensitive dL-grammars, extensible libraries of shapes and growth functions, and MPEG movie recording and playback capability. The system also provides various rendering modes such as wireframes and ray-tracing.

The key component of pgl is the smooth integration of of discrete and continuous aspects of plant development. The structure of the modeled system is represented by a collection of modules. Each module could be represented by an identifier denoting its type and parameters desribing its temporal behavior. A(2, 3.5) is an example of a simple modules of type A with parameter {2, 3.5}.

Bellow is an example of a toy dL-grammar.

x<s F(x,s) -> F(2*x, s) z(45) F(2*x, s).

There are two different kinds of actions that could be dispatched based on the evaluation of the conditional expression: production application and evaluation of differential equations describing the temporal behavior of module's parameters. pgl provides an extensible database of typical differential equations (Growth Function Library).

Visual representation of modules is determined by choosing an appropriate shape from the Shape Library and attaching it to the module.

pgl allows to archive the modules and later retrieve them into different dL-grammars (Module Library).

dL-Grammar Modules with Predefined Semantics

x(angle) y(angle) z(angle) x_ y_ z_ L_ L(length) [ ] < > .

x(angle) rotates the turtle by angle degrees around the x axis

x_*(angle ...) same as above, but allows additional arguments

y(angle) rotates the turtle by angle degrees around the y axis

y_*(angle ...) same as above, but allows additional arguments

z rotates the turtle by angle degrees around the z axis

z_*(angle ...) same as above, but allows additional arguments

L(length) moves the turtle forward by length units

L_*(length...) same as above, but allows additional arguments

[pushes the turtle

] pops the turtle

< starts surface specification

> finishes surface specification

. saves the current turtle position

ENVIRONMENT

DISPLAY the deafault host display to use

TK_LIBRARY tk library directory (/usr/sww/tcl/lib/tk)

MANPATH the search path for manual pages

FILES

compiler.tcl designer.tcl draw.tcl file.tcl growth.tcl help.tcl library.tcl main.tcl plant.tcl rewrite.tcl shapes.tcl standard_modules.tcl

turtle.c Makefile

KEYWORDS

pgl, dL

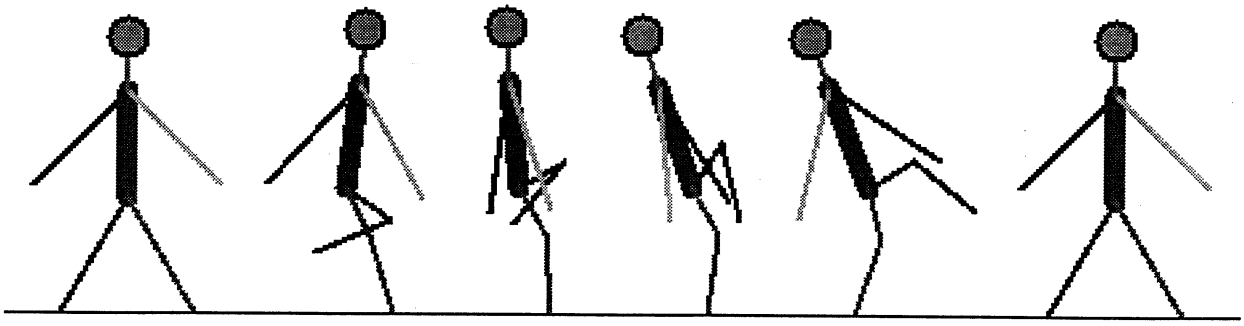
AUTHORS

pliu@cs.berkeley.edu, borisv@cs.berkeley.edu University of California at Berkeley

Spline-Parameterized Adjustable Motion

A tool for creating silly walks[1]

(Revision 1 corresponding to SPAM Version 1.0)



Course Project Report
CS285 / Fall 1994

Dan Garcia
ddgarcia@cs.berkeley.edu
http://http.cs.berkeley.edu/~ddgarcia
Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Abstract

SPAM is a stand-alone X-window utility written in Tcl/Tk which allows a user to adjust the walking motion of an articulated human figure. The user interactively specifies the splines which drive five forward kinematics joints angles (left and right leg, left and right knee and the torso) and drags a scrollbar to smoothly step back and forth through the motion, or just watches as the system animates the motion. A physics-based option exists which models the body as a point mass and allows for jumping and leaping. The parameters of motion may then be saved to a file for later retrieval, playback, modification or for incorporation into another package which allows equivalent forward-kinematic motion specification, such as `UGMovie` or `GLIDE`.

Keywords : Splines, Key-Frame Animation, Articulated Bodies, Silly Walks

---

[1]This work supported by a grant from the **Ministry of Silly Walks**, John Marwood "Otto" Cleese, Founder.

1. Introduction

The human gait is tremendously expressible. A walk can show pride, slyness, urgency and even downright silliness. Transferring this motion to an articulated human model (anthropomorphizing, if you will) is not very difficult, if the model is sufficiently well parameterized. The challenge is in searching the parameter space to find a *particular* motion which suits a certain need. For example, the user might want to have his model walk like Groucho Marx, with the knees bent and hips low off of the ground, or simulate someone walking in deep mud. This utility allows the user to interactively adjust spline parameters describing a walking figure's motion. Using this time-based scrollbar provided, the user can quickly hone in on any desired motion from the visual feedback provided by the articulated stick figure.

Unless the user has chosen the physics-based modeling option, the system does no checks to see whether the motion is physically realistic. That is, a human would fall over before completing the walk as animated by the program. There is a `Ground?` option which checks if either the feet or the knees is penetrating the ground and graphically illustrates the violating time ranges. Other than that, the onus rests on the user to maintain a physically realistic gait, if that is necessary. However, after experimenting with this utility, most users agree that half of the fun is creating very silly walks which could in no way be realized.

2. Technical Description

From an interface standpoint, the user selects one of the four control points for one of the five color-coded Bézier splines and drags it up and down. While the spline parameters are being modified, the articulated figure is updated in real time. The user can move a scrollbar controlling the current time and animate the figure back and forth. Or the `walk` button can be selected, which animates the figure beginning from the currently selected time. The user can check if a foot or knee penetrates the floor, and select from among three different walker constraining options: `"Periodic Lowest Foot"`, `"Arbitrary Lowest Foot"`, and `"Inertia-based model"`. The behind-the-scenes technical issues for these features are described below.

2.1. Spline Formulations

The spline chosen here is one segment of a cubic Bézier spline formulation. Formally, the calculation of the value of the spline is as follows:

$$\mathbf{Q}(t) = \mathbf{V}_0 (1-t)^3 + \mathbf{V}_1 \, 3t \, (1-t)^2 + \mathbf{V}_2 \, 3t^2 \, (1-t) + \mathbf{V}_3 \, t^3 \qquad (1)$$

Since we are using this spline to calculate how a one-dimensional joint angle changes over time, we restrict the $\mathbf{V}_i$ inputs to scalar quantities. The spline interpolates the endpoints, which is tremendously useful when interactively specifying the initial and ending positions. The user specifies these endpoints interactively in the spline window. The two internal scalar quantities are also interactively specified, but placed at t=1/3 and t=2/3 and the motion of the control points is constrained to lie within the vertical line for those values of t.

The spline visualization region of the program has vertical lines indicating lines of constant angle. The vertical range of the window is 180° to -180°. For the end control points $\mathbf{V}_0$ and $\mathbf{V}_3$, the vertical height of the control point from the center is exactly the value of the spline. For the middle control points $\mathbf{V}_1$ and $\mathbf{V}_2$, however, the scale is reduced by a factor of two. E.g., if either of these control points is placed on the 90° line, the internal representation of the value is 180°.

2.2. Spline File I/O

The spline parameters may be saved to a text file for later retrieval. The format of the data file is simple - it is just a space-separated grid of numbers. Each row contains the parameters for a different spline (in the following order: back leg, back knee, front leg, front knee, torso) and each column contains the four control points (x_0, x_1, x_2 and x_3) for that spline. This file must have as an extension ".w" since the loader filters out all other files when listing them. As an example, this is the default file that is loaded on startup:

```
unix% cat default.w
30 -50 5 -30
0 120 -30 0
-30 -10 10 30
0 0 0 0
0 19 -14 0
```

2.3. Symmetrical constraints

We allow the user to specify the motion of the figure as it goes through one cycle. At time frame t=0 both feet are planted, then the left leg will become the free leg and the right foot will be planted throughout the cycle. At t=1 both feet are planted once again. We wish to then continue for the next cycle with the roles of the left and right feet interchanged. For this to be possible, the terminal positions at t=0 and t=1 must be mirrors of one another. E.g., the angle of the right leg's hip at t=0 must be the angle of the left leg's hip at t=1. Similarly for the left leg's hip and the knee angles. The torso's tilt must be consistent with itself - its value at t=0 and t=1 must be the same. We enforce this internally; when the user moves one of the control points at t=0, the constrained control point at t=1 moves in unison, and similarly for the t=1 control points.

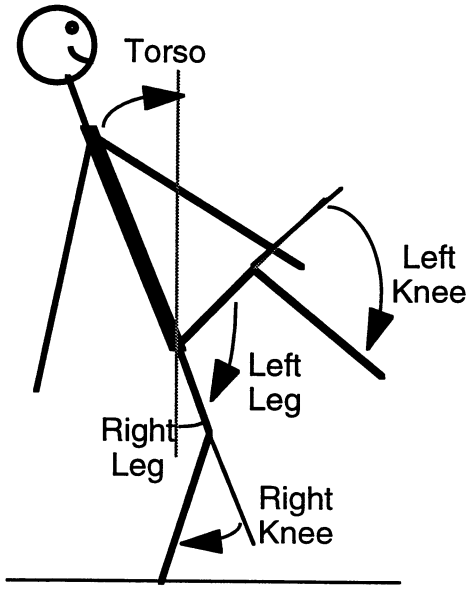
When the user chooses the `Walk` button to animate the system, it begins from the current time value (internally t=[0,1], but the slider reads t=[0,100]), then swaps the curves that drive the motion. This is so the walker then does exactly with its right side that it just did with its left side. Then it continues with the part of the first cycle that it skipped (if any) due to starting at the current slider-indicated time. This way, the system always resets to the time and parity as before the animation, and the user gets to see one full cycle.

2.4. Forward Kinematics

The user specifies interactively how five joint angles change over time. Each angle is calculated based on the relationship between two limbs/directions listed below:

Joint Angle	Relationship
• Right Leg	Right upper leg and lower vertical
• Left Leg	Left upper leg and lower vertical
• Right knee	Right upper leg and the right lower leg
• Left Knee	Left upper leg and the left lower leg
• Torso	Torso and upper vertical

Indicated in the diagram are the five joint angles the user may adjust. The joint angles are oriented so that a positive angle means a clockwise rotation, so it is a left-hand rotation. If realism is desired, then from this convention the knee angles should never be negative.



There is one more joint angle which is internally specified - that of the swinging arms. It was decided that the arm motion was incidental and the focus of the user's energy should be the motion of the legs, so there is no user-control of the arm motion. The angle the arm makes with the body just linearly move back and forth.

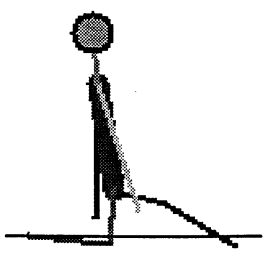
2.5. The use of Color

Selecting the correct curve control points to edit from among five different sets all in one figure seems daunting. It has been simplified by assigning a unique color to each of the joint angles. All the information relevant to that joint angle shares the same color : the spline, control points and the part of the figure the angle immediately affects. For example, the left hip angle is red, so the left leg is red in the diagram. When the system is animating its second cycle, the colors of the symmetric pairs in the spline editor are swapped, so the curves that are displayed correspond to the colors of the limbs that are actually doing the motion.

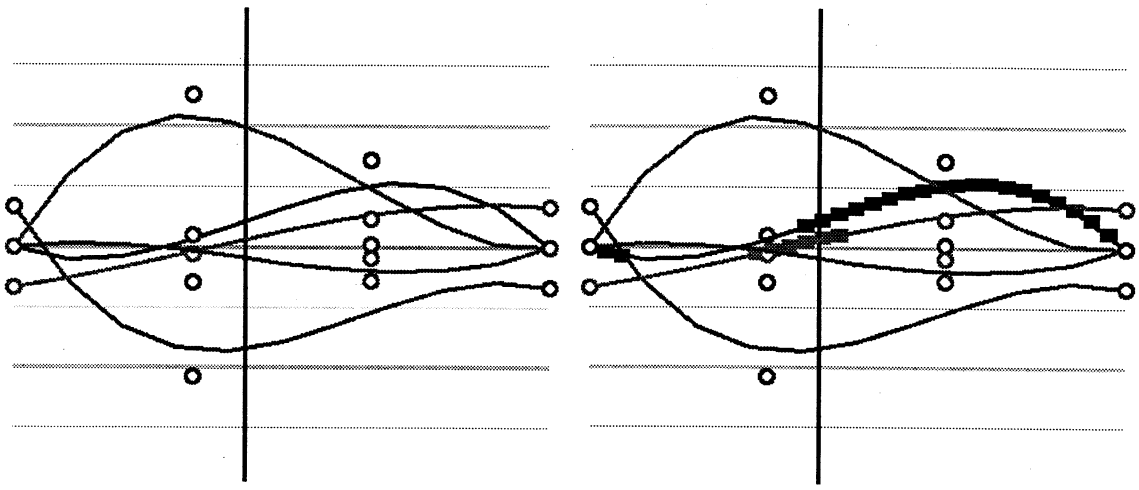
In the case when the motion is very subtle, many joint angles will be near zero, and the center of the spline region may become cluttered. The control point which the user wishes to change may be buried below another. To solve this problem, there are 5 buttons which when pushed bring that spline and control points to the foreground. To aid in finding the right button, the foreground color of the button is also the same as the curve it brings to the foreground.

2.6. Collision Detection

If the free foot goes below the line representing the floor, then it can be said that the foot has *collided* with the floor. If the user cares whether this happens, they may click the `Check?` button which goes through the motion internally and determines whether either the knee or the foot collided with the floor over the cycle of the walk. A collision may happen multiple times over the cycle and by various parts of the body. This information needs to be conveyed back to the user, which is accomplished by highlighting



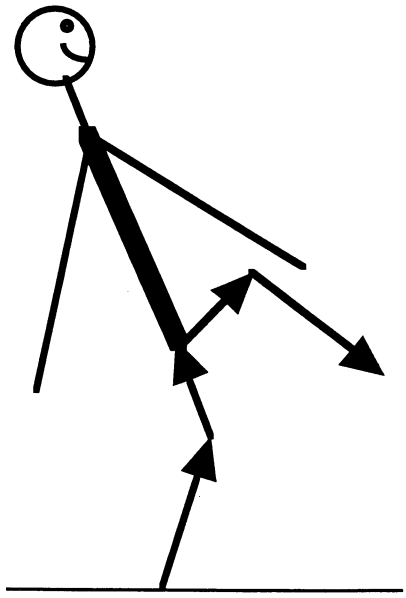
the *closest responsible* curves over the times that the collisions occurred. That is, if the foot is at fault, the knee angle is highlighted, and not the leg angle, even though the knee angle may have been innocent. This is illustrated in the example below. The image on the left is the curve window before the `Check?` button was selected. The image on the right is the same window immediately after the system had performed its calculation. The position of the walker at the time as indicated by the black vertical time line is show on the previous page.



The collision calculation is quite simple. For every time step that would have been performed if the system were animating itself (the speed radiobuttons determine this) the location of the knees and feet are compared with the ground location. If they are below the floor by a certain tolerance (set to 1/2 pixel) then a collision for that time frame is recorded by plotting a rectangular region with the same color as the closest responsible curve right where the curve lies.

2.7. Periodic Lowest Foot

This is the default setting of the system. It means that over the course of one period, the planted foot is always at the place it was as at the beginning of the cycle. There is no other motion constraint. This can easily result in impossible motions, since the free foot could pierce the floor at any time and would have to be isolated and hand-adjusted using the collision-detection algorithm just mentioned.

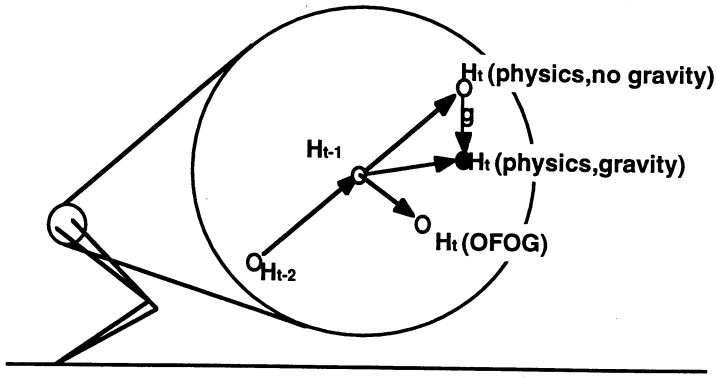


Using this simple model, it is easy to determine the position of the joints as shown in the diagram to the right. We know the location of the grounded foot. From this we use the knee angle to calculate where the knee is, then the hip angle to calculate where the hip is, then the free knee and finally the free foot. The position of the neck is derived from the torso angle, and the arms are calculated given the neck position and using the internal arm angles as mentioned earlier.

2.8. Arbitrary Lowest Foot

One way to handle the problem of an arbitrary foot passing through the ground is to switch pivot feet as soon as it happens. That way the lowest foot is always on the ground. This is the second modeling option for the system.

2.9. Inertia-based model



This model uses the arbitrary lowest foot approach but takes it one step further. It makes an assumption that the body can be modeled by a point mass centered at the hip. To calculate the location of the walker in space, it uses the following approach: each time frame it calculates the instantaneous velocity of the hip based on the location of the previous two frames. This is compared with the position of the leg as a result of following the arbitrary-lowest-foot rule described above. If the height of the leg as calculated by the physics is higher, then the hip is positioned by the physics as shown in the diagram.

This is a problem for our symmetry model as described before. What if the user has the walker jump, and it is in the air at the end of a cycle when it was supposed to be on the ground? In truth, this isn't a problem at all. The system simply tells the walker to follow the same walking motion, but from the air! If (and when, if gravity is turned off) the walker lands, it follows the same walking motion as specified by the cycle, and will probably begin to jump during the next cycle.

3. Future Enhancements

3.1 Multiple interpolating spline segments

One of the drawbacks of the current approach is that only one spline segment is used. This means the motion is limited to a third-order system, and the free leg can't wave back and forth more than twice over the cycle. In addition, the inner two cubic Bézier control points do not interpolate the curve (other than in degenerate collinear cases), which means the user cannot place the control point at a particular angle and expect the spline to go through it.

It would be nice to have a multi-segment interpolating spline, like a Hermite formulation whose internal joints were joined with second-order continuity. A *natural* Hermite curve (in which the second derivatives are zero for the end segments) would be appropriate here. Even better would be a system which allowed the user to draw out (ala a paint program) how the curve changed over time and the system would find the spline and number of control points necessary to interpolate that curve within some error factor.

3.2 Interactive constraint system, ala Mechedit

Instead of specifying the position at key frames by clicking on the spline editing window, it makes much more sense to have the user clicking in the *walking* window to adjust the motion, ala Mechedit. The user could select the foot and drag it around in a circle with either the knee fixed or the knee mobile.

3.3 Quadrupedal motion

Bipedal motion is relatively straightforward since the case when both feet are on the ground and the legs are pushing in opposite directions is degenerate and almost never happens. When it does, one foot is chosen as the planted foot and the other foot is pushed away. But what happens when our walking motion is extended to a quadruped? Then this case becomes the norm - two legs are almost always on the ground at one time and the requested joint angles might not be in phase, so there would be forces acting on the animal to either squash it together or rip it apart. To handle this, one could take the average of the motions as the cumulative motion for the animal.

3.4 More physics

The physics modeled here is very rudimentary. Much more physics would have to be added to begin to make the motion real. Some of the physical realities ignored by this simulation which could make the system much more realistic are:

- Tipping energy and inertia. The body could be modeled as an inverted pendulum pivoting about the single grounded foot, and the walker could tip over if it stopped too fast or if it lifted the wrong leg off the ground and its center of mass was not over the grounded foot.

- The masses of various parts of the body. Currently, the walker may tip all the way back with the torso and not fall over. If we added tipping ability, the walker would have to compensate by lifting one of its legs, but the mass of the legs and torso must be taken into account for this to happen.

- Feet. The current walker has points for the feet, and a realistic model of feet and the joint ranges of the ankles might be interesting and more realistic. However, this is agonizingly hard to implement, as described in the classic line "The thrill of victory and the agony of the feet". :-)

- Friction between the ground and the walker's foot. If a human were to run, jump in the air and stop, the frictional coefficients between the feet and the ground would very much affect what happened after landing. If the ground were ice, the legs might slip out, and if it were a perfectly frictional surface, the person might tip over if they expected to skid a bit.

- Torque limits for the joints. The current walker's joints can accelerate infinitely, which is not very realistic. Putting torque limits on the joints would create a maximum-height jump, which is more realistic.

- Jump cushioning. Currently, as soon as the jumper lands, the system figures out which is the lowest foot and immediately allows the system to jump again. In reality, a jumper would have to cushion the landing energy before it would be able to jump again.

3.5 Generalized Terrain

Once the physics has been worked out, we could have the terrain be generalized to anything: stairs, a hill, a fractal mountain, etc. All the pieces are in place for this generalization (simple inertia-based physics and lowest-foot approach).

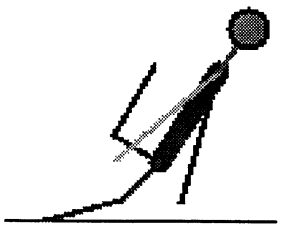
3.6 Exporting the motion into `UGMovie` or `GLIDE`

The system could very easily be modified to spit out a valid `GLIDE` or `UGMovie` file from the required parametric specification. Or a non-interactive converter could be created which would take the parameters of motion and generate a default `GLIDE/UGMovie` file.

4. Conclusion

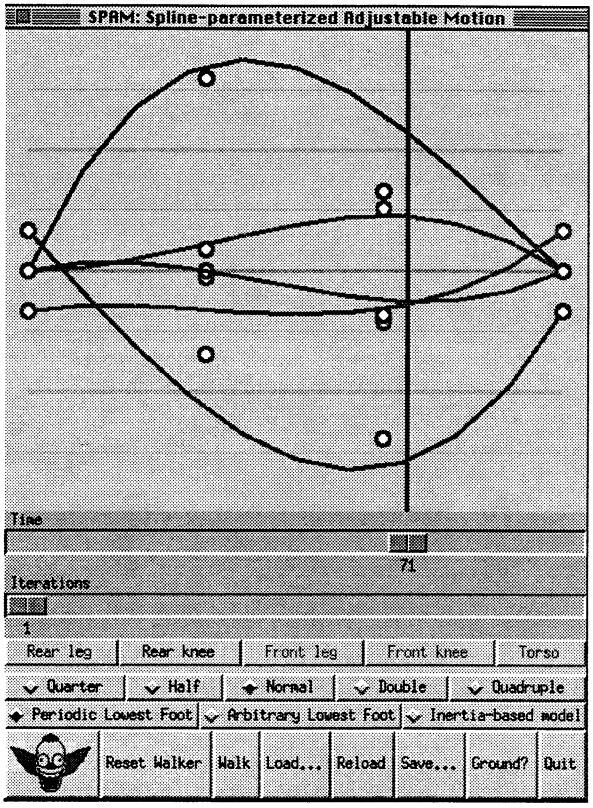
This paper presents `SPAM`, a tool to easily specify the motion of an articulated human figure. Although there are some assumptions made which do not allow for fully generalized motion, it is easy to create quite realistic motion. It is hoped that this will be a useful tool for animators or future CS285 students wishing to anthropomorphize a walking figure.

This was a very enjoyable project! Projects bring the greatest pleasure at the tail end of their life cycle, when all of the hard problems have been solved and small, interesting features can be added at will. This project reached that stage early as much of the geometry had been worked out in a previous programming assignment, and all that was left was the user-interface work and the generalization of the motion (our assignment assumed the front leg remained straight and followed the path of a circular arc at a constant angular velocity). One important piece of advice for future CS285 students working on projects is never to underestimate the power and quick no-compile iteration cycle of Tcl/Tk. It's surprising how much more productive a programmer can become when creating software with an interpreted language that doesn't have a high compile-time cost.



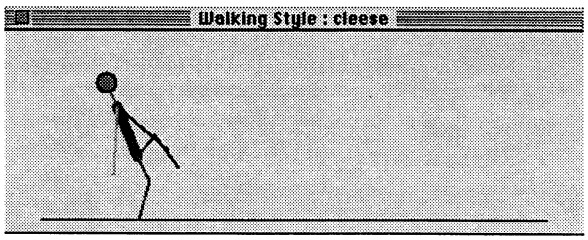
Overall, it was an truly enjoyable fall! :-)

5. Visual Man Pages



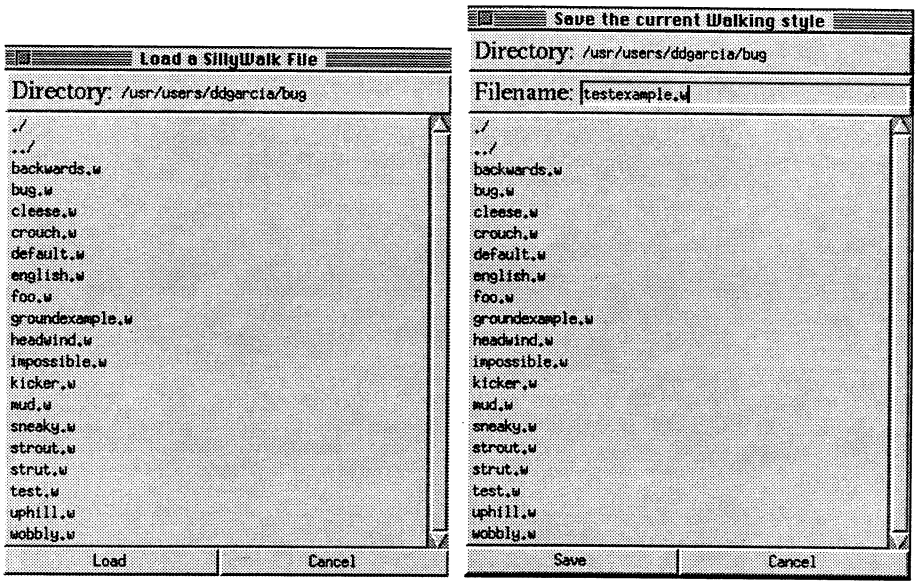
The main SPAM window

The upper region is the spline editing window and the colored circles are the control points for the splines. The vertical black line directly above the `Time` scrollbar indicates where in the current time in the animation cycle. The `Iterations` scrollbar allows the user to select how many cycles to animate when the `Walk` button is pushed. The five buttons below that raise the appropriate spline and its control points to the top (useful when the spline window is too cluttered). Below that is the animation speed and internal representation radiobuttons. The buttons on the bottom row (from left to right): bring up the About window, reset the walker, walk, load / reload / save a data file, perform collision detection with the knees/feet and the floor, and quit the program.



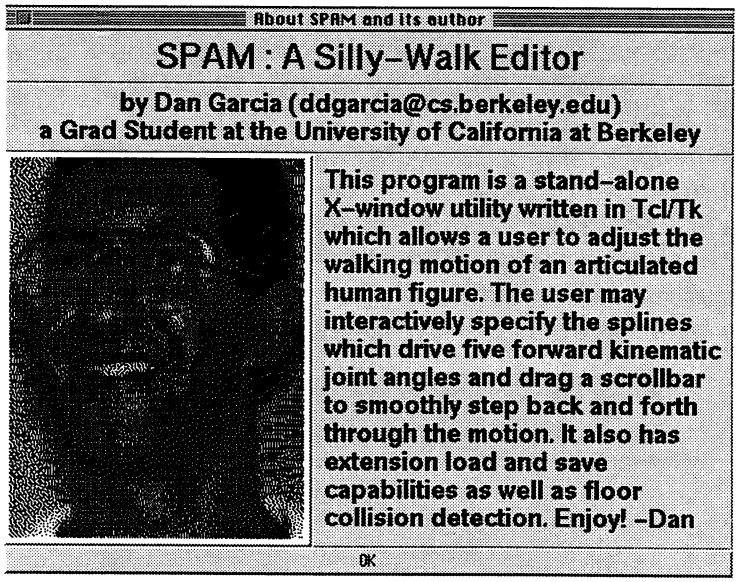
The Walking Figure window

This output-only window illustrates the position of the walker at the specified time.



The load and save windows

These windows come up after clicking "`Load...`" or "`Save...`" from the main `SPAM` window. To load a file, simply select the file with the left mouse button and then click the `Load` button. To save a file, simply type in the name of the new file and click the `Save` button. Clicking the `Cancel` button dismisses the windows. Double-clicking a filename is a shortcut for selecting it and the choosing `Load` and `Save`, respectively. To change a directory, simply double-click on the directory name and the new directory's contents will be displayed in the window. Clicking on "`..`" moves to the parent directory and clicking on "`.`" refreshes the current directory in case it has changed.



The "About `SPAM` and its author" window

This is the window created after the user has clicked on the Krusty-the-clown button.

ANALYSIS AND MODELING OF A HOBERMAN SPHERE

Carlo H. Séquin

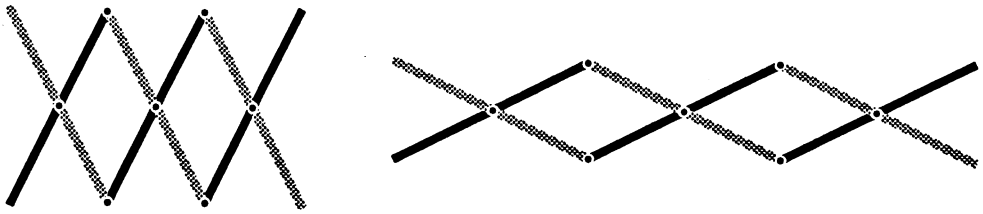
CS Division, U.C. Berkeley

ABSTRACT

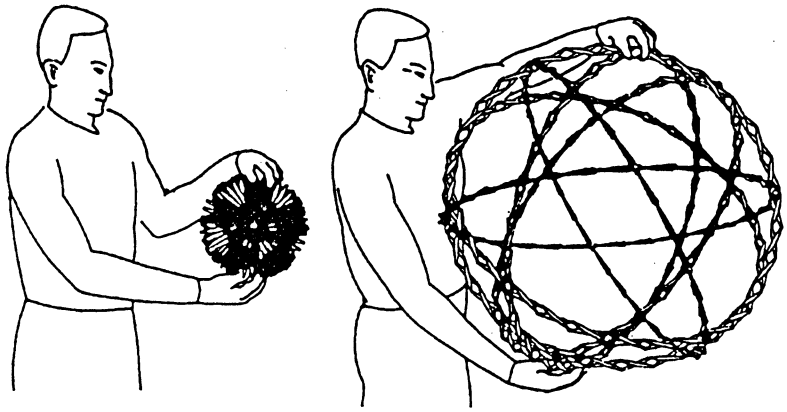
The geometry of the expanding Hoberman sphere mechanism is investigated, and one particular construction is realized as an UgMovie demonstration. This report also serves as a tutorial for the step-by-step construction of a UgMovie model for more complicated constrained structures.

1. HOBERMAN SPHERES

Hoberman spheres are combinations of several carefully tailored sections of expanding rack mechanisms following the edges of a regular or semi-regular polyhedron or of an equivalent tessellation of a sphere. Because of its inherent symmetry, the overall mechanism is highly degenerate and has exactly one degree of freedom, which permits to expand and collapse the structure.



In this report we investigate a particular construction based on the icosidodecahedron.

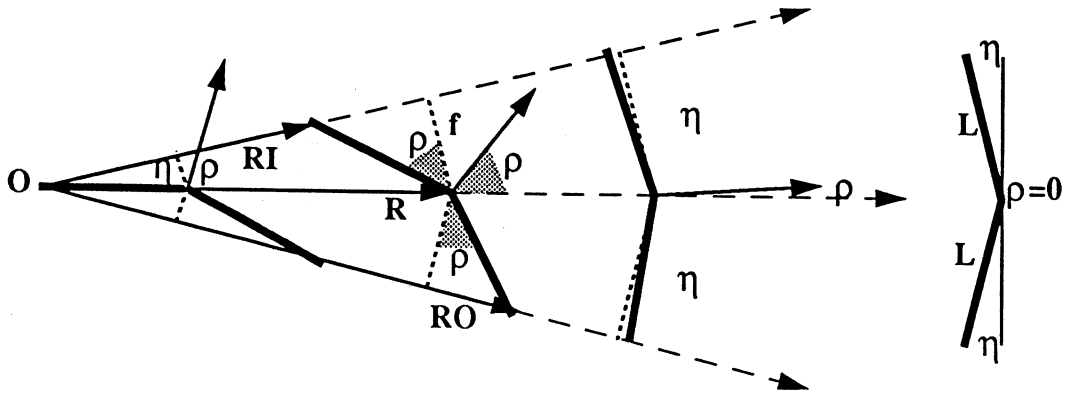


A physical realization of such a HOBERMAN Sphere™ produced by HOBERMAN Toys, Inc. can be purchased at the Exploratorium, or from Design Science Toys, Route 9, Box 1362, Tivoli NY 12583, or through some mail order services. It is a kit of about 500 blue translucent plastic parts and will result in a sphere about 30" in diameter which folds to a 10" diameter ball.

2. ANALYSIS OF THE BASIC RACK SECTION

In the simplest case, the basic rack section follows a part of a great circle on a sphere. As the sphere folds and unfolds, all junctions and cross points between links move strictly radially. Thus the analysis can be concentrated on one cross link in a rack section.

An individual link, i.e., half of one cross link of the rack, must move so that its two endpoints remain on two fixed rays, RI and RO, through the origin. The link is bent so that its midpoint moves on the mid-angle ray, R, between the endpoint rays. The necessary bending angle of the link is equal to the angle between the endpoint rays; half of that angle we call η. As the link moves in and out, it rotates. The symmetric, outermost position defines the neutral, unrotated state; deviations from that are measured by the rotation angle ρ. We set the length of each leg of the link to L.



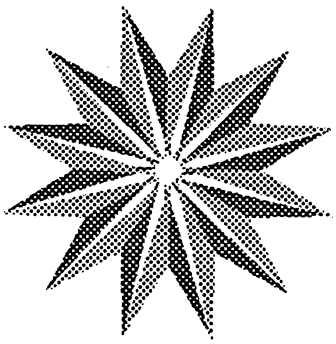
There is a simple relationship between the distance, R, of the link-midpoint from the origin and the rotation angle ρ: $R \sin \eta = f = L \cos \rho$.

The extreme innermost position is assumed when $R_{min} = L$, because then one of the link ends touches the origin O; the rotation associated with this position is $\rho = 90° - \eta$. The outer extreme position is assumed when the other leg of the link lies symmetrically between its outer ray and the mid-point ray; the rotation angle for this situation is $\rho = \eta / 2$, and the corresponding radius is $R_{max} = L (\cos \eta/2) / \sin \eta$; this is derived from the general relation between R and ρ. To check these geometrical relations, we first make a simple 2-dimensional demonstration with UgMovie:

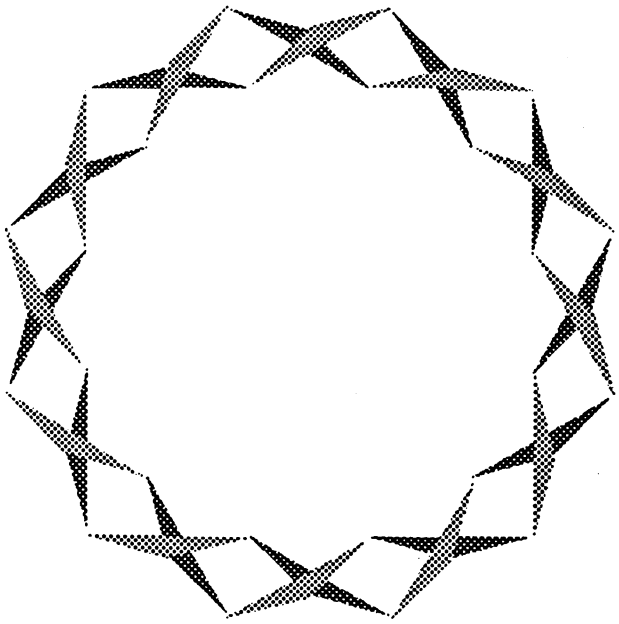
```
{2D demo of basic rack mechanism, ring of 12 cross elements, halfangle eta = 15°}

c C1 1.0 0 0; {define white -- and other desired colors}

def linkP;  {the basic bent link, a simple triangle, defined in a neutral position
             with its mid-joint at the origin, and other vertices at the end joints}
   v VM  0  0  0.1;
   v VI  -0.25881905  0.96592583  0.1;  {sin eta, cos eta, thickness}
   v VO  -0.25881905 -0.96592583  0.1;
   f fr (VM VI VO) ;
end;

{ Get smooth, symmetric motion by using  rad =R as primary variable}
(vardef rad 3.0 1 3.8306488)    { min=1, max=cos(eta/2)/sin(eta) }

def mlink;  { the basic moving link }
   (meval  i lt (linkP -rz $ dacos(0.25881905*rad) $ -tx $ rad $);)
end;
```

```
{One 3-section rack, which then gets used 4 times }
def mrack;
   i L1 (mlink C4 -rz 60);              i l1 (mlink C5 -rz 60 -rx 180);
   i L2 (mlink C4 );                    i l2 (mlink C5  -rx 180);
   i L3 (mlink C4 -rz -60);             i l3 (mlink C5 -rz -60 -rx 180);
end;

i R0 (mrack );
i R1 (mrack -rz 90);
i R3 (mrack -rz 180);
i R4 (mrack -rz 270);

{Notice: we can describe the whole mechanism with a single meval and a single var def. }
```

Almost Fully Collapsed Ring



Almost Fully Expanded Ring



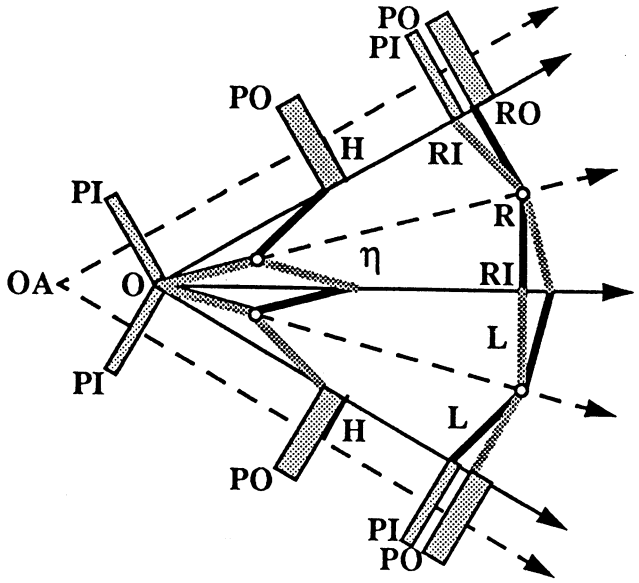
3. ANALYSIS OF LINKS ENDING IN MOVING PLATFORMS

However, a physical realization of a Hoberman sphere needs some platforms (or hubs) of finite size where several links come together -- corresponding to the vertices of the underlying regular polyhedron. While the centers of these platforms can travel on strictly radial rays, their edges travel on offset paths that intersect outside the origin. The totally collapsed position is now determined by the tight packing of the inner hubs. We now extend our analysis to include such finite-size platforms.

It turns out that the racks themselves still move in the same space wedges as before, but these wedges are now suitably offset from the actual origin, OA, to make room for the interspersed moving platforms. The amount of this offset is proportional to the size of the platform (2H):

$O - OA = H/\sin(n*\eta)$,

where n is the number of regular cross-link sections between platforms.



The platforms move on separate rays between those space wedges as a function of R:

The inner platform is at a distance from O equal to: $RI = R \cos(\eta) - L \sin(\rho)$.

The outer platform is at a distance from O equal to: $RO = R \cos(\eta) + L \sin(\rho)$.

For our next demonstration we have extruded all geometry perpendicular to the plane of this page, forming simple prismatic links and brick-shaped platforms. We also animate the whole structure by making it dependent on time, moving it in and out periodically between the extreme positions calculated above. To obtain a symmetrical pulsating motion, we make R(t) the primary independent variable, rather than the rotation angle ρ, and make it a sinusoidal function of time:

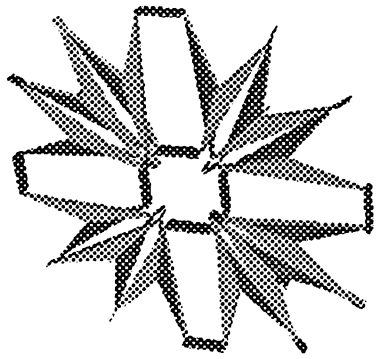
$R(t) = 0.5\ (R_{max} - R_{min}) \cos(0.02\ p * t) + 1.5\ R_{min} + 0.5\ R_{max}$

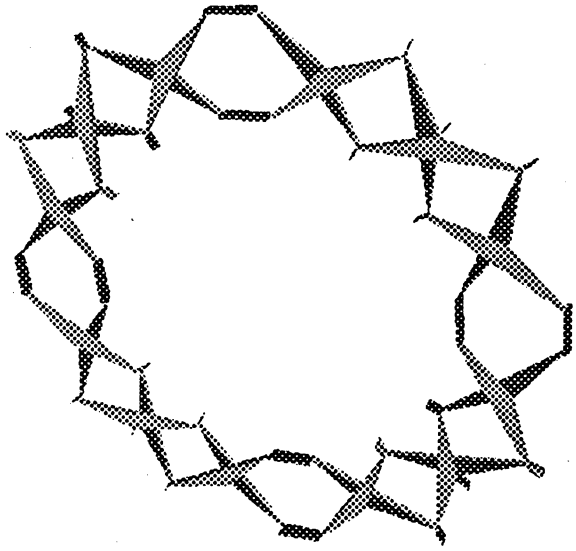
For the UniGrafix/UgMovie demonstration, it is also worthwhile to introduce explicitly the variable ρ; it makes the expressions simpler, and it permits us to watch the variation of ρ with time explicitly on its slider in the variable window. In the file below, we have made a definition encapsulating the movement of one time-dependent link and then use many symmetrical replications to create the whole structure. Here we used a rack section with three cross links spanning 90° of a full circle and then replicated this section four times. We also need separate definitions of the inner and outer moving platforms.

```
{ More advanced demo: one ring with 4 interspersed platforms; eta = 15 degrees}
def flag;
        {just a rectangle to mark hinges}
end;

def linkP;
        {basic bent link, a prismatic shape, as above, but extruded}
end;

def hubO; {Outer platforms to which links are attached}
        {bricks,  halfwidth: H=0.3,  thickness: T=0.05}
end;

(vardef rad (2.425 + 1.425*cos(0.02*pi*t)) 1 3.85 ){ min=1,max=cos(eta/2)/sin(eta)}

(vardef rho ( dacos(0.25881905*rad)  )  0  90 )
```

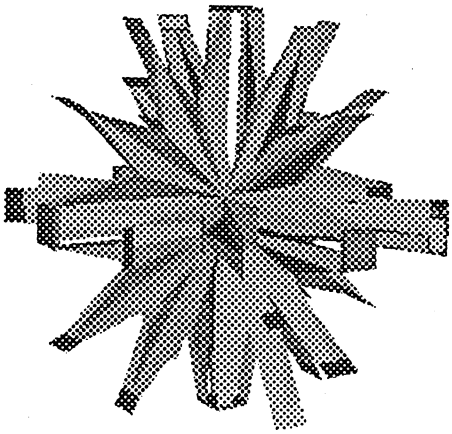
```
def linkOt;   {moving  unit length link}
   (meval  i lt (linkP -rz $ rho $ -tx $ rad $); )
end;

def xFlagOt;  {moving flag at cross point}
   (meval  i fx (flag  -tx $ rad $); )
end;

def oFlagOt;  {outer flag at RO}
   (meval  i fo (flag  -tx $ rad*0.96592583 + sind(rho) $); )
end;

def iFlagOt;  {inner flag at RI}
   (meval  i fi (flag  -tx $ rad*0.96592583 - sind(rho) $); )
end;

def huboOt;  {outer platform, offset -n +x by H/2, same as space wedge}
   (meval  i ho (hubO  -tx $ 0.3 + rad*0.96592583 + sind(rho) $ ); )
end;

def hubiOt;  {inner platform, also connected at inner edge to avoid interference}
   (meval  i hi (hubO  -tx $ 0.3 + rad*0.96592583 - sind(rho) $ ); )
end;

{An actual assembly: define a quarter ring, call it 4 times, add the hubs}
def rackOt; {regular cross link}
   i L0 (linkOt C4 );                    i l0 (linkOt C5  -rx 180);
   i Lp (linkOt C4 -rz 30 );             i lp (linkOt C5 -rz 30  -rx 180);
   i Ln (linkOt C4 -rz -30 );            i ln (linkOt C5 -rz -30  -rx 180);
   i Flx (xFlagOt ); {on reg link}
   i Fli (iFlagOt -rz 15);               i Flo (oFlagOt -rz 15);
   i fli (iFlagOt -rz -15);              i flo (oFlagOt -rz -15);
end;

{Space wedge is offset by sqrt(2)* H/2 to make room for platforms }
i R0 (rackOt -tx 0.424264 -rz 45 );     i R1 (rackOt -tx 0.424264 -rz 135);
i R3 (rackOt -tx 0.424264 -rz 225);     i R4 (rackOt -tx 0.424264 -rz 315);

i Pi  (hubiOt C7 );                     i Po  (huboOt C7 );
i pi  (hubiOt C7 -rz 90);               i po  (huboOt C7 -rz 90);
i nPi (hubiOt C7 -rz 180);              i nPo (huboOt C7 -rz 180);
i npi (hubiOt C7 -rz -90);              i npo (huboOt C7 -rz -90);
```

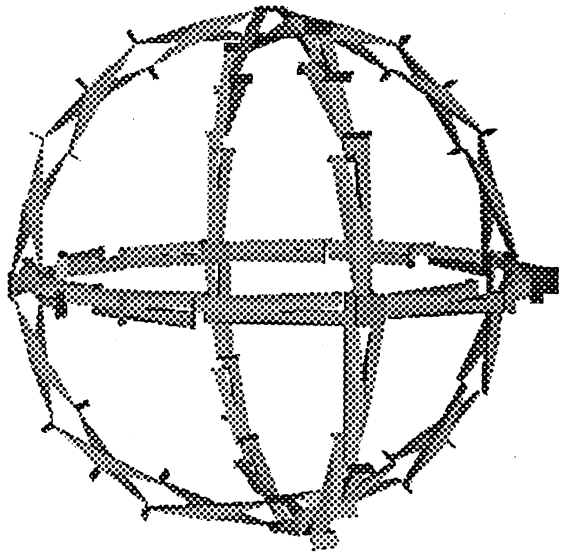
In the figure below, a slightly oblique view is given of the resulting structure in order to better exhibit the enhanced geometry compared to the first simple 2D ring structure.





The 3-dimensional structure becomes even more visible, if we put three such rings together to form a Hoberman Sphere based on the octahedron. Here we use twelve suitably placed 90°-arcs of rack sections with 3 cross-link each, and six pairs of platforms.



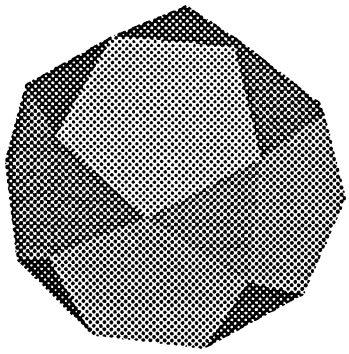


4. PUTTING TOGETHER AN ICOSAHEDRAL SPHERE.

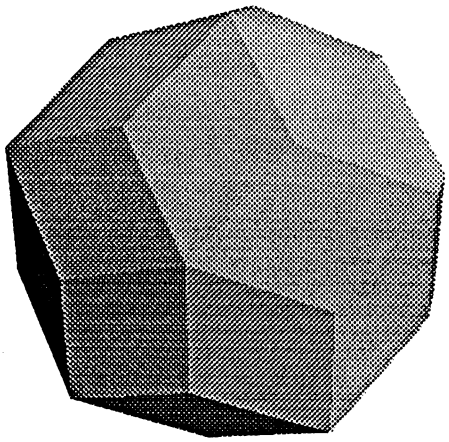
The sphere that we really want to model has icosahedral symmetry. This is somewhat harder to figure out, since the arcs now have to be placed at odd angles. The envisioned structure replaces each edge of an icosidodecahedron with a 3-element rack section. An icosidodecahedron can be constructed from either the icosahedron or from the dodecahedron by vertex truncation to the middle of the attached edges. Its dual is the 30-sided triacontrahedron; its figure below has been generated with Mathematica with the command:

```
Show[ Graphics3D[ { EdgeForm[], Stellate[ Icosahedron[], 1.146 ] } ], Boxed->False ];
```

Icosidodecahedron



Triacontrahedron



In the desired structure, all edges fall on six intersecting great circles that lie in the medial planes parallel to the faces of the dodecahedron. Each circle is subdivided by the hubs into 10 rack sections covering 36° each. But rather than modeling the whole mechanism as a composite

of six circular structures, we will take a different approach. We describe the sphere as a collection of twenty triangles composed of 3 rack sections each, making use of an icosahedral symmetry generator that was developed for an earlier project. In a first step we construct such an icosi-dodecahedral sphere without any platforms -- just with point connections between the racks.

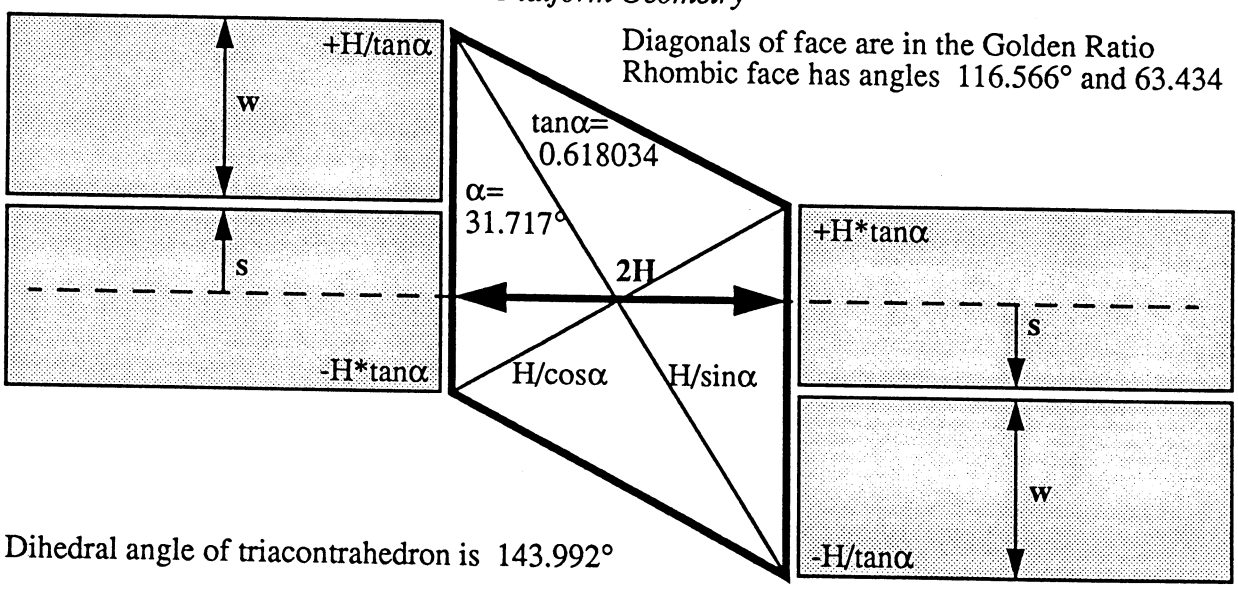
The main task then is to calculate how much the three rack planes are tilted against the 3-fold symmetry axis of the triangle. One way to do this is to start with the icosahedron embedded in a unit cube so that six of its edges are in symmetrical positions on the six cube faces; the icosahedral edge length in this case becomes 2*0.618034. The height, h, of the embedded f2-truncation triangle is 0.53523. Sqrt[1-(2h/3)^2] is the distance, d = 0.93417, from the origin to the center of the face. The offset angle then becomes arctan(h/3 /d) = 10.81232°. Since we have composed the rack in the xy-plane, we need to rotate it through the complement angle of 79.18768° to make it a side of our desired triangle symmetrically located around the z-axis. The other three sides are then formed by additional rotations by 120° around the z-axis.

Simple prismatic links are good enough to model a full ring of such rack sections. If we want to model a geodesic dome structure that collapses as tightly as possible, we need to make the inner link ends pyramidal so that they can move all the way to the origin without interference. It is worthwhile to build ant test this structure to debug all the angle calculations before tackling the final task and inserting the platforms at all the hubs.

5. SPHERE WITH PLATFORMS

The 30 hubs must have rhombic shape and – when completely collapsed – form the dual of the polyhedron outlined by the rack sections: a rhombic triacontrahedron. Emerging from the 60 edges of this 30-sided polyhedron are wedges suitably offset from the origin which each contain the rack mechanism analyzed in the previous section. Below are dimensions of the platforms and of the attached links. Note, that the links each must be w = 1.118*H wide, and that the racks must be laterally offset by an amount s = 0.5*H (1/tan α – tan α) = H/2.

Platform Geometry



Diagonals of face are in the Golden Ratio
Rhombic face has angles 116.566° and 63.434

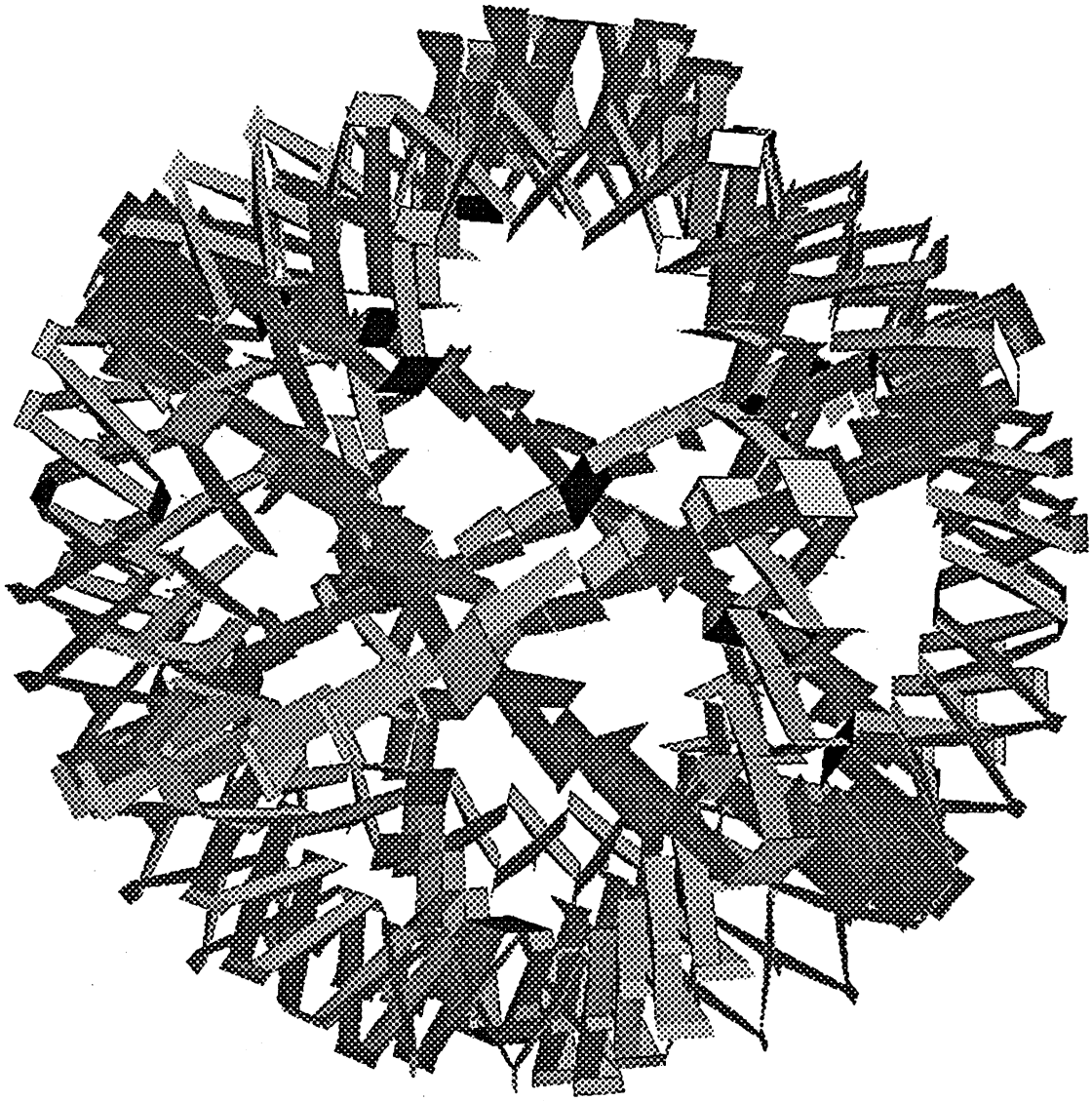
Dihedral angle of triacontrahedron is 143.992°

Now we can put together the whole structure with 180 cross-links and 60 platforms:

```
{Hoberman Sphere based on icosidodecahedron with 30 platforms forming triacontrahedron}
{20 triangular meshes, each composed from 3_X-els racks, half-angle eta = 6 degrees}
{Platform size is taken from triacontrahedron embedded in unit cube, H= 0.32492}

{Definitions of colors and parts as above ...}

def link;  {the basic prismaticlink}
  v VO  0  0  0.363;                      v vo  0  0  0;
  v VP  -0.1045   0.9945  0.363;           v vp  -0.1045   0.9945  0;
  v VN  -0.1045  -0.9945  0.363;           v vn  -0.1045  -0.9945  0;
  f fr (VO VP VN) C2;                     f bk (vo vn vp) C2;
  f (VO vo vp VP);    f (VP vp vn VN);    f (VN vn vo VO);
end;

{The crucial motion definitions}
(vardef rad (4.27683 * sin(pi * 0.02 * t) + 5.27683)  1  9.5536613)
(vardef rho (  dacos(0.10452846*rad)  ) 0 90 )

def LT;  {the time dependent link}
  (meval   i lt (link -rz $ dacos(0.10452846*rad) $  -tx $ rad $); )
end;

def rack3;
   i L1 (LT C4 -rz 12 -tz -0.1622);     i l1 (LT C5 -rz 12 -rx 180 -tz -0.16219);
   i L2 (LT C4 -tz -0.1622);            i l2 (LT C5 -rx 180  -tz -0.16219);
   i L3 (LT C4 -rz -12 -tz -0.1622);    i l3 (LT C5 -rz -12 -rx 180  -tz -0.16219);
end;

def F;   {One triangular facet of three rack sections; place on icosahedron faces}
{Shift rack wedge outward by tx = H/sin(3eta) = 1.05146 to make room for platforms}
   i R1 (rack3 -tx 1.05146  -ry -79.187683);
   i R2 (rack3 -tx 1.05146  -ry -79.187683 -rz 120);
   i R3 (rack3 -tx 1.05146  -ry -79.187683 -rz -120);
end;

{20 instances of the basic triangular mesh facet in icosahedral placement}
   i i0 (F C1);
   i i1 (F C2 -ry 41.810315 -rz 180);
   i i2 (F C3 -ry 41.810315 -rz -60);
   i i3 (F C4 -ry 41.810315 -rz 60);
   i i4 (F C5 -ry 41.810315 -rz -120 -ry -41.810315);
   i i5 (F C3 -ry 41.810315 -rz 120 -ry -41.810315);
   i i6 (F C5 -ry 41.810315 -rz -120 -ry -41.810315 -rz 120);
   i i7 (F C4 -ry 41.810315 -rz 120 -ry -41.810315 -rz 120);
   i i8 (F C5 -ry 41.810315 -rz -120 -ry -41.81035 -rz -120);
   i i9 (F C2 -ry 41.810315 -rz 120 -ry -41.810315 -rz -120);
   {and all 10 instances again with "-ry 180" appended to yield opposite facets }

{ THE PLATFORMS }

def plate;  {triacontra rhombuses are aligned along the edges of icosa}
   v Zy  0  0.618034  1 ;              v Zb  0  -0.618034  1 ;
   v Zx  0.381966  0  1 ;              v Za  -0.381966  0  1 ;
   f Z (Zx Zy Za Zb) C1;               f z (Zb Za Zy Zx) C6;
end;

def hubiOt;  {inner platform, radial offset (tz) already included in plate def.}
   (meval i hi (plate  -tz $ 0.0 + rad*0.9945219 - sind(rho) $ ); )
end;

def huboOt;  {outer platform}
   (meval i ho (plate  -tz $ 0.0 + rad*0.9945219 + sind(rho)  $ ); )
end;

def plpair;  {pair composed of inner and outer hub platform}
   i ppi (hubiOt);                     i ppo (huboOt);
end;
```

```
def tcube;   {Six of the triacontra faces lying in the faces of a cube}
   i p1 (plpair);                          i p2 (plpair -rx 180);
   i p3 (plpair -ry 90 -rx 90);            i p4 (plpair -ry -90 -rx -90);
   i p5 (plpair -rx 90 -ry 90);            i p6 (plpair -rx -90 -ry -90);
end;

{Five instances of the above cube group yields all 30 triacontra faces }
   i t1 (tcube C1 -ry -20.9051575 );
   i t2 (tcube C1 -ry  20.9051575 -rz  120 -ry -41.810315 );
   i t3 (tcube C1 -ry  20.9051575 -rz -120 -ry -41.810315 );
   i t4 (tcube C1 -ry -20.9051575 -rz  120 );
   i t5 (tcube C1 -ry -20.9051575 -rz -120 );
```

UniGrafix View of the Completed, Partially Open Icosidodecahedral Hoberman Sphere



6. CREATING AND DEBUGGING UGMOVIE MODELS

Creating a properly working UgMovie model is an interactive process; I have never gotten a model completely right in the first try. It pays to build such models from simpler parts which are

individually debugged, and to keep all the part descriptions very simple until the whole mechanism is fully debugged; one can easily substitute more sophisticated part geometries subsequently. It also proved useful to put in some auxiliary geometrical features such as the rails along which certain points are supposed to move (e.g., the hinges at the edges of the hubs) to help in debugging the initial – potentially chaotic – collection of parts.

Another concern is efficiency. Since parsing and evaluating "meval" expression, which has to be performed for every frame displayed is a time-consuming process, we aim to define as few such movements as possible. Try to find the minimum number of generic motions which can then be reused for all the parts that move in related ways. Also try to find a primary variable that carries the basic time-dependence which leads to simple expressions for all the other variables. With such a primary variable with built-in time-dependence, one only has to click "cycle" in the "time" menu to obtain a continuously moving display. And if you want to control the action of the mechanism manually, we can just open the variable window and grab the slider button of the "time" slider. This is better than grabbing the slider of the primary variable, since once you have moved the latter one manually, it will be disconnected from its dependence on time and we will have to explicitly edit its expression window to get back the original time dependence.

It pays to introduce other variables and make them dependent on your primary variable. These variables can then serve as intermediate variables, reducing the complexity of the final expressions for a particular instantiated part. It also helps debugging, since in the variable window you can see the various variables move in relation to your primary variable.

In general, during the debugging process, it is most convenient to work in the directory containing the UgMovie file. Call the UgMovie program in a separate window and keep the ASCII file open in editing mode. Now you can quickly load this file, check its behavior, modify the file, and try again until all bugs have been eliminated. Since the overall size of the mechanism may vary substantially as a function of time, it may not fill properly the UgMovie window for the exact details that one wants to study. It is convenient, that the value of time slider is maintained when one file is replaced by another one. The selected time value is used to calculate the initial size of the new structure and determines the new scale to make the figure reasonably window-filling when it is first displayed; thus leaving the time slider in an appropriate position allows some control over the initial scale factor. As the time slider gets moved, the overall structure may get clipped by the UgMovie display viewport; but with suitable manipulation of the orientation of the virtual object (using the middle mouse button), a crucial detail of the mechanism can often be brought into the front position in the display frame and can then be examined closely at a larger scale than the default scale. The right mouse button also provides some zoom capabilities.

7. CONCLUSIONS

Hoberman Spheres are fascinating mechanisms. While rather intricate, the underlying mathematics is quite tractable when approached properly, and it is not too difficult to model this mechanism in UniGrafix and animate it with the UgMovie extension. The most important point in such an effort is to keep the description compact and to reuse the same defined parts and motions as often as possible. The motions of the individual parts should be separated from the definition of the geometry of the individual parts, so that these can be gradually enhanced and embellished. Under no circumstances should one try to move individual vertices of individual parts, thus violating the notion that the whole mechanism is composed of a set of connected rigid parts.