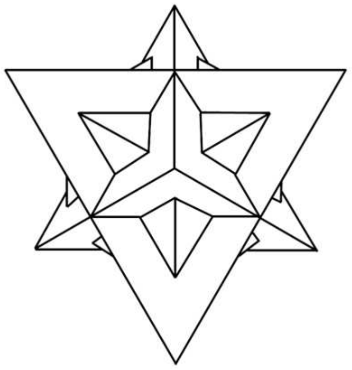
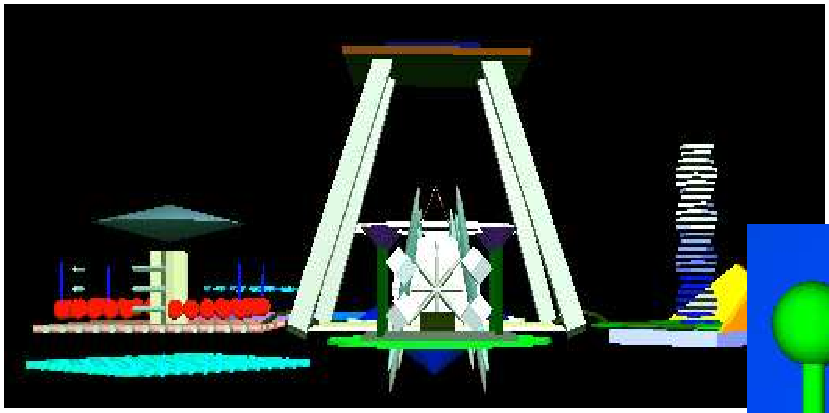
NISTIR 6924

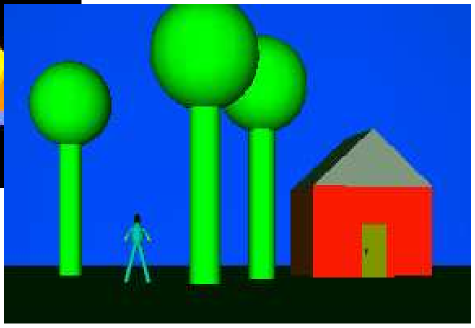
A Glyph Toolbox for Immersive Scientific Visualization



Harry W. Bullen IV, Jessica S. Chang, Alexander V. Harn,
Sean P. Kelly, Steven G. Satterfield, Peter M. Ketcham,
Judith E. Devaney

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaitheresburg, MD 20899





NIST
National Institute of
Standards and Technology
Technology Administration
U.S. Department of Commerce

NISTIR 6924

A Glyph Toolbox for Immersive Scientific Visualization

Harry W. Bullen IV
Jessica S. Chang
Alexander V. Harn
Sean P. Kelly
Steven G. Satterfield
Peter M. Ketcham
Judith E. Devaney

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

October 30, 2002

U.S. DEPARTMENT OF COMMERCE
Donald L. Evans, Secretary

TECHNOLOGY ADMINISTRATION
Phillip J. Bond, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arden L. Bement, Jr., Director

Abstract

We describe a set of software, The Glyph ToolBox (GTB), for creating three dimensional (3D) glyphs. This software defines a single, general format for describing glyphs; it includes color and opacity parameters as well as location information.

GTB is written with the UNIX philosophy of small reusable programs that are text based for portability and efficiency. Version 1.0 of GTB currently contains simple figures, manipulation functions, extrusion functions, meta-figure functions, as well as additional functions such as text creators.

We describe four applications of the glyph toolbox: a visualization of the Monk's problem, a relationship highlighter, a smiley emoticon, and a display algorithm for concave surfaces.

We separate the creation of the glyphs from their display. We provide a filter that can translate the GTB format to Inventor format or VRML 1.0. However, any system can incorporate the GTB format into their environment, making the creation and use of glyphs uniform across viewers.

Contents

Chapter 1

Overview and Motivation

1.1 Introduction

As scientists collect more and more data and computers allow them to obtain and store more information, a problem emerges of how to interpret all of this information. The traditional methods of viewing data visually are becoming obsolete under the burden of representing complex information, and viewing raw data is impossible considering the vast amounts of such data. As such, scientists now wish to view their data by stepping into it in three dimensions (3D). This allows them to visualize data like never before and easily communicate their findings to the world. However, setting up an environment in which to do this is a difficult task and requires extensive experience with 3D graphics, which unfortunately few scientists have. So it is desirable to create a group of commands that do not require extensive experience with 3D graphics to use. However, data is also becoming increasingly complex, often involving more than three dimensions, and such data may be represented by creating very complex multi-dimensional objects and projecting them into three dimensions. However, understanding these objects is difficult, making interpreting data from them even harder. One solution to this problem is to use glyphs to represent the data.

A glyph is a symbol that conveys information visually. They can be found even in antiquity in cave paintings and written text. Classic examples of glyph sets are Chinese, Japanese Kanji, and ancient Egyptian heiroglyph writing in which pictures could have various meanings based on context. By the same logic, even our Roman characters and words are glyphs. In this paper, the term *glyph* most commonly refers to an object with various attributes that can be changed to represent various properties of data.

Glyphs have been used for many years to present scientific data and to communicate scientific ideas. Traditional examples include error bars and other technical ornamentations on graphs of experimentally measured data. Modern examples include streamlines and flow vectors in computational fluid dynamics simulations.

1.2 Previous Work on 3D Glyphs

Curington and Dodd [1] developed a glyph library that contains a large number of glyph modules representing both 2D and 3D geometries. The Graphics, Animation, and Visualization Lab at the University of Maryland Baltimore County used 2D and 3D glyphs in their information visualization work [2]. This work included the use of glyphs in an immersive environment. Ribarsky and van Teylingen [3] developed a user-friendly, human-computer interface for glyph-based scientific visualization in a virtual environment. Reise [4] applied 3D glyphs to the visualization of current fields in a single-blade pump. Rober [5] experimented with 3D glyph styles and rendering techniques and used a variety of 3D glyphs to demonstrate flow visualization in a medical data set [6]. Batagelj and Mrvar [7] applied 3D glyphs to the presentation of multivariate data. Sobel et al. [8] reported their experiences with 3D glyphs in the visualization of pulsatile-flow simulations within coronary arteries.

Ebert et al. [9] used 3D glyphs in their Stereoscopic Field Analyzer and Lodha et al. [10] used 3D glyphs for visualizing uncertainty in fluid flow. Boring and Pang [11] applied various shading and coloring techniques to 3D glyphs in order to visualize flow fields. In work closely related to that of Boring and Pang [11], Shaw et al. [12] extended their glyph-based visualization system. Finally, Cedilnik and Rheingans [13] employed glyphs for the visualization of uncertainty in a data set.

Scientific visualization packages in general, and Open Visualization Data Explorer (OpenDX) [14] in particular, can be used to create, position, and render 3D glyphs within a given visualization.

1.3 Motivation

The motivation for this work is to create a set of UNIX style commands and filters that can be used to create geometric objects, i.e., glyphs, useful for data visualization or the creation of virtual environments. At the heart of the command set is a single file format that represents the glyphs in a rendering independent manner. Some of the commands generate a glyph as a file. Other filters modify or transform the glyph data. The approach used to create the command set follows a classic UNIX philosophy [15] of creating simple and reusable tools that lend themselves to be combined in a variety of ways to perform useful tasks. Additionally, utilizing the basic glyph tools, more complicated glyphs are easily leveraged.

Another motivation for this work is to promote creativity. Thus, a command line paradigm was chosen. The advantage of using a command line approach versus a graphical user interface (GUI) is flexibility. The variety for combining command and filters to accomplish a desired goal is limited only by the users imagination and not by predefined functions provided by a GUI developer.

This work is also meant to be system and rendering independent. It provides a set of simple tools that are easily implemented on any system with a C compiler. Utilizing a simple file description allows it to be easily transformed into a suitable graphics format.

This work was primarily done as a summer project by four high school students visiting the Scientific Applications and Visualization Group (SAVG) of the Mathematical and Computational Sciences Division at NIST. The remainder of this report will describe the implementation of the Glyph Toolbox. It will then describe four extension projects done by the individual students utilizing the Glyph Toolbox.

Chapter 2

The Glyph Toolbox

2.1 Overview of The Glyph Toolbox

In this paper we describe a Glyph Toolbox (GTB). This toolbox contains both primitive and meta commands for creating a large number of glyphs. The approach taken in creating the Glyph Toolbox parallelled the UNIX [15] philosophy. Each function is written as a simple command designed to do one small task well; the GTB file format is an ASCII text file [16]. This maximized the flexibility of the toolbox, facilitated the execution of complex jobs based on simple steps and greatly facilitated the location and correction of bugs and errors in the developing code. This facilitates the viewing and manual editing of data files on any system [16]. Additionally, the actual GTB object files are simple, standardized, and rendering-independent. They contain only coordinate and other numeric data, no information on the actual drawing and display processes. This maximizes the number of systems able to display the glyphs. There are five main groups of GTB functions.

These glyphs are viewable in the NIST RAVE (Reconfigurable Automatic Virtual Environment) by converting the GTB format to Inventor format with the gtbtoiv filter. This filter can also convert to VRML 1.0 which adds flexibility to the toolbox by making the glyphs viewable with a VRML enabled web browser.

2.2 Basic Object Functions

These functions, which contain the platonic solids, compose one of the four main groups of GTB functions. This group consists of cube (gtb-hexa), tetrahedron (gtb-tetra), dodecahedron (gtb-dodeca), octagon (gtb-octa), and icosahedron (gtb-icosa).

2.3 Object Manipulation Functions

These functions take an existing GTB file as input and output a modified GTB file. These filters are used to scale, translate, rotate and color objects. Other filters reduce file size by removing excess comments and data from files. Some filters return values based on objects, such as their x, y, and z coordinate ranges, others create object augmentations such as the bounding-box generator gtb-bound.

2.4 Extrusion and Manipulation Functions

These functions are extrusion and surface of revolution objects. They are based around *surface of revolution* (gtb-sor) and its derivative, *regular polygon* (gtb-regpoly). While a function roughly equivalent to gtb-sor already existed, the addition of gtb-regpoly allowed for other unique functions able to form 3D figures by manipulating point sets. A gtb-regpoly point set could be fed into gtb-sor to generate a *lathe* object, fed into gtb-prism to create a basic *extruded* object, fed into gtb-pyramid to create converging objects such as pyramids and cones or simply connected into a single face by gtb-face. Later, functions such as gtb-sweep and gtb-polar took advantage of point sets and revolution functions to generate even more unique objects.

2.5 Meta Functions

These functions create more complex, and frequently recognizable objects such as birds, fish, thermometers and faces out of the basic object commands. Some meta commands create objects with variable features such as ice cream cones (multiple scoops), multicolored stars and faces, but others display particular types of data. Gtb-confetti plots similar but orderable glyphs over a region of 3D space. Gtb-surf can take up to four rectangular arrays to generate a colored topography. Gtb-vol, an extension of gtb-surf, can make a rectangular 3D wireframe block based on values in up to three 3-dimensional arrays, with an option to mark outstanding points with colored *dualtetra* stars.

2.6 Miscellaneous Functions

These functions don't fit well into any category. Gtb-text creates line-drawn 3D text based on input strings. Gtb-extr will extrude points into lines and lines into planes, but unlike gtb-prism, won't connect points that weren't connected to begin with. Other functions create reference objects, such as axes, arrows, and unit-long bars or extend the functionality of previously written programs. Several filters allow the display of concave objects, a capability that many of the graphics viewers most commonly used with the Glyph Toolbox lack.

Chapter 3

Individual Extension Projects

3.1 The Monk Problem: Harry W. Bullen, IV

One of the applications of the glyph toolbox was the visualization of the monk problem. The monk problem was devised to test learning algorithms. It consisted of similar tests. In each test a group of “robots” was divided up into two groups by some rule. The algorithms were then given a subset of these two groups and had to go about determining the rule for separating them out.

Each individual robot had six characteristics: head shape, body shape, color, item, smile, and tie. There were a total of 432 different possible robot combinations in this manner. Using the existing commands in the glyph toolbox, a “robot” glyph command was created. This command takes the six characteristics as arguments and created a glyph toolbox representation of the robot.

The initial data format contained each robot’s characteristics and whether or not they satisfied the rule. Robots from the two groups were mixed together. The two groups were separated out resulting in two files, one that contained the characteristics of robots that fit the rule, and one that contained characteristics of robots that did not. These two files were then processed in the same manner, producing structurally similar visualizations.

The files of robot characteristics were combined with the robot glyph and other glyph toolbox commands to create a grid containing each robot group, such as that in Figure 1.1. These grids clearly illustrated the different rules that the learning algorithms were trying to reproduce.

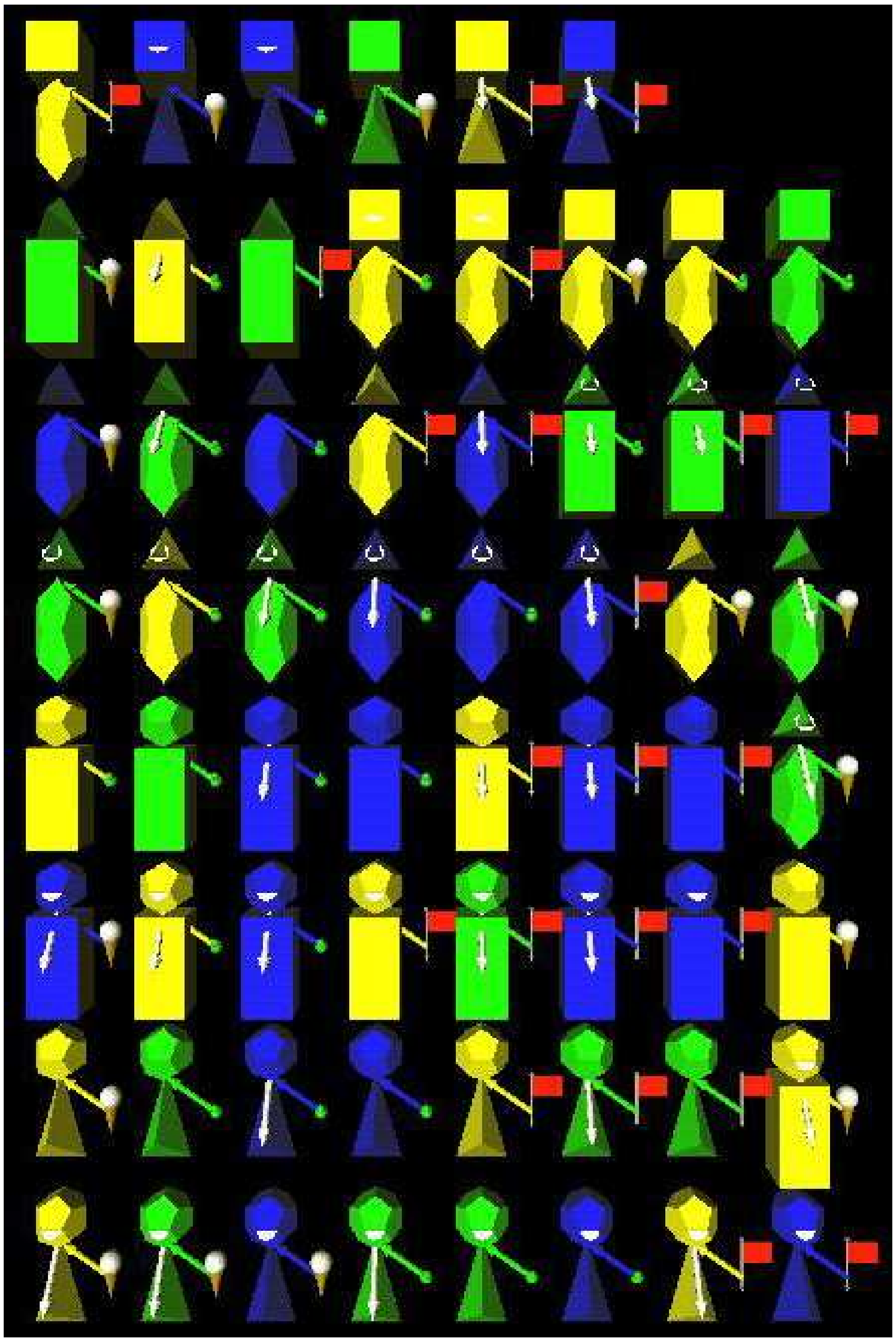


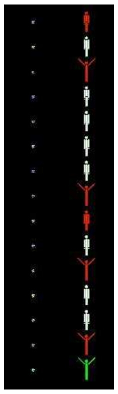
Figure 3.1: One of the subsets of robots created from the Monk Problem data

3.2 Relationship Highlighter: Jessica S. Chang

Gtb-relhi was written to identify the relationships between various attributes in a given data file for a set number of examples. It ultimately processes a section of the information and highlights a certain aspect of the data through immersive visualization so that the viewer can analyze the data with no more than a cursory glance.

Often, sets of data have attributes that affect one another or are closely related to each other. In such a case, it would only hinder the analysis of the data to have each of the related attributes in their own category. However, this program will create a meta-attribute, which condenses the group attributes into one. In this way, the data is represented more efficiently.

All attributes must be binary. A mask file specifies which attributes are part of the group and which are not. From this file, the program processes only the group attributes for each example and condenses them into one meta-attribute. All other attributes are ignored. Though many types of relationships exist, this program deals specifically with the M-of-N relationship. If M of the N group attributes are true, then there is a M-of-N relationship among those attributes for that particular example and the meta-attribute returns true for that example.



After setting the meta-attributes, the program then creates glyphs in an organized manner to display the data and conclusions. A human figure glyph is used. If the meta-attribute is true, then the person's hands are always up. If the conclusion is true and the meta-attribute is also true, then the person is red. This person always has his hands up; red makes visibility easier. If the conclusion is true, but the meta-attribute is false, then the person will be yellow. If the conclusion is false, and the meta-attribute is also false, then the person is white. However, if the conclusion is false, but the meta-attribute is true, then the person will turn green.

This color arrangement becomes very useful when the user does not know which attributes in a data file are of the group. There exists an option that enables the user to generate masks for every possible combination that satisfies the M-of-N relationship. (Note: the examples and conclusion columns may not ever be specified as part of the group and therefore cannot be specified with a 1 in any mask.) The program is run for each mask automatically with this option and the results of each mask are written to the output file as well as visualized.

Using this program, he or she can compare the conclusion with the meta-attributes of each mask and conclude which masks are correct and which ones must be eliminated. The visualization would allow the user to immediately discern which masks are incorrect. If there are green or yellow people, then the conclusion and the meta-attribute do not agree. If either of these types of people are found in any mask visualization, then that mask must be eliminated as the correct mask.

3.3 Proper Display of Concave Surfaces: Alexander V. Harn

Concave objects tend to display improperly when created by the Glyph Toolbox; this is really a problem with the renderer used to view the gtb files rather than a problem with the files created by Glyph Toolbox programs. The typical algorithm used by a renderer for creating a triangle mesh of a polygon is based on connecting the first point with each pair of other consecutive points along the polygon's edge. This is almost totally incapable of properly displaying a concave polygon. The filter gtb-breakface was created to solve this problem. It uses an alternate algorithm, based on that described in *Computational Geometry in C* [17] based on removing vertices that are the center of non-reflex angles and repeating until three or fewer points remain. The primary modification to the algorithm described in [17] is that while the original assumed that the face was in the xy-plane, and therefore 2D, the version in the Glyph Toolbox only assumes that the face is contained by a single plane. This allows proper display of many concave objects, such as that in Figure 3.2, although it still has limitations.

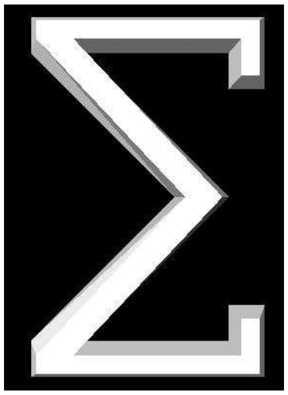


Figure 3.2: A sample 3d concave object created with gtb-breakface

As noted, the algorithm assumes that all of the points on a face are coplanar. To fix this, a filter, gtb-plane, was created to split each face of an object into faces with all points on the face being coplanar.

In addition, the algorithm in gtb-breakface cannot handle non-simple polygons, so a filter to make the most common type of polygon that is usually drawn as a non-simple polygon was created. This program is gtb-ngram, and is used to create simple polygon versions of regular n-grams, or star-shaped figures. However, these n-grams are concave and need to be passed through gtb-breakface to properly display. There are a wide variety of other types of usually non-simple polygons, and gtb-breakface is still incapable of properly creating a triangular mesh for those types of shapes, but due to time constraints it was unfortunately not possible to develop the ability to handle all of the possible non-simple polygons. The probable method for achieving such a capability would be the creation of another filter to make any polygon into one or more simple polygons.

The algorithm for gtb-breakface is as follows:

For each face of the gtb-object repeat the following steps.

Identify a component of the crossproduct of the vectors formed by points 1,0 and 1,2, to use in various comparisons, such that the chosen component is non-zero for the points tested. Because of the assumption that the face is contained by a plane, this component should be non-zero for all of the points tested.

Traverse the vertices of the face in counter-clockwise order. To achieve this, first traverse the vertices in the order given by the gtb file, but keep a copy of the original vertices in the opposite order. If this does not result in the creation of any new faces, assume that the vertices were traversed in clockwise order, so traverse the vertices in the order previously stored. The traversal is a destructive process and the original order will not be saved, which is why the copy needs to be made prior to the first reversal.

Traversal of the vertices of a face involves looking for internal diagonals that have exactly one vertex between the vertices that they connect(referred to from now on as the intermediate point). When such a vertex is found, add a face containing the two vertices connected by the diagonal and the intermediate point. Remove the intermediate point from the list of points on the face. Traverse the new list of points on the face.

Identifying diagonals is the section of the algorithm that was changed most from the algorithm in [17], because many of the algorithms that work fine in the xy-plane do not extend directly into three dimensions. Diagonals are identified by determining that they are internal to the face and that they do not intersect any of the edges except those which also connect to the endpoints that the possible diagonal connects.

The list of points on the face may be indexed as $\{i_0, i_1, i_2, \ldots, i_{n-1}\}$, where n is the number of points defining the face. Assume the first point on a possible diagonal has is i_j, so the second point on the diagonal is i_{j+2}. The component of the crossproduct used was selected at the beginning of the processing of the current face.

Whether or not the diagonals are internal is checked based on the component of the crossproducts of various vectors, as follows. If the component of the crossproduct of the vectors connecting i_j to i_{j-1}, and i_j to i_{j+2} is less than or equal to 0, the diagonal is internal if both the component of the crossproduct of the vectors connecting i_{j+2} to i_j and i_{j+2} to i_{j-1}, and the component of the crossproduct of the vectors connecting i_j to i_{j+2} and i_j to i_{j+1} are less than 0. If the first calculated component is greater than 0, the diagonal is internal if both the component of the crossproduct of the vectors connecting i_{j+2} to i_j and i_{j+2} to i_{j+1}, and the component of the crossproduct of the vectors connecting i_j to i_{j+2} and i_j to i_{j-1} are greater than 0.

Testing for intersection with all edges except those that are expected to connect to the diagonal at its endpoints is a multi-step process. The first check is that the components of the crossproducts for both points on an edge are on the same side of the diagonal being tested, and vice versa. This is done by multiplying the components together, and if they are positive, the vertices are on the same side of the other line segment. If they are not on the same side of the other line segment in both cases, the edge and the diagonal must intersect, because both are in the same plane. The second step, if the first has not already indicated an intersection, is to check that a vertex of the edge or diagonal is not on the other line segment. This is done based on the triangle inequality rule. If the length of the segments from each endpoint of the edge or diagonal (first segment) to an endpoint on the other segment (second segment) is equal to the length of the first segment, then the endpoint on the second segment intersects the first segment.

3.4 Gtb-smiley: Sean P. Kelly

The image of the human face is recognized by any human who sees it. It can be recognized instantaneously and evaluated effortlessly. Facial expressions can even be inferred from otherwise arbitrary symbols, such as the common colon-parenthesis emoticon, or “smiley.” Facial expressions can convey a large amount of information in a very small amount of time, and thus have a large amount of potential in the data visualization field.

Gtb-smiley was an attempt to create a reasonably intuitive simple emoticon generator capable of generating faces based on numeric data. Statistician Herman Chernoff created one method for generating faces based on numeric data, which was published in his report “The Use of Faces to Represent Points in k-Dimensional Space Graphically” in 1973. [18] Chernoff’s method involves assigning numeric values to features such as eye spacing, mouth size and shape, and head eccentricity, as shown in Figure 3.3. This results in faces that *do* represent data, but have no reliable expressive quality, requiring viewers to actively compute and compare features.

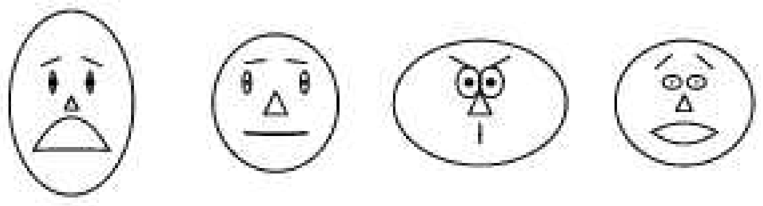


Figure 3.3: Sample Chernoff Faces from MathWorld[18]

While a separate Chernoff function was later added, the main gtb-smiley program was written without knowledge of Chernoff’s work. Many concepts, however, parallel Chernoff’s methods. Each face is generated based on simple measurements and values. While Chernoff’s faces are generated based on roughly 10 values, gtb-smiley faces use 35 variables. Enough gtb-smiley variables were equivalent to Chernoff’s to allow easy conversion from Chernoff’s format to gtb-smiley’s, but the gtb-smiley format provides much flexibility that Chernoff’s does not. The *method* for face generation in gtb-smiley is also different from Chernoff’s, being much more oriented towards turning numeric data into recognizable mood-based expressions than simply attribute measurements. While custom data can be entered manually into gtb-smiley, the preferred method for generating faces is using pre-defined emotion values such as “happy,” “sad,” “angry,” or “surprised” as-is or blended with other emotions or custom data. This allows the resultant figures to have at least some basis on the types of expressions commonly and readily interpreted by the human brain.

Due simply to the nature of the information conveyed naturally by the human face, it is difficult and may be impossible to create an entirely accurate and valid function for conversion of raw numerics to appropriate facial features. Gtb-smiley is by no means perfect, and in its complexity it may not even be as “good” for easily representing large amounts of in-

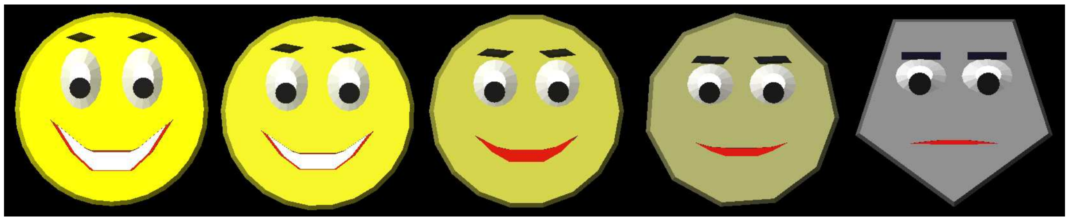
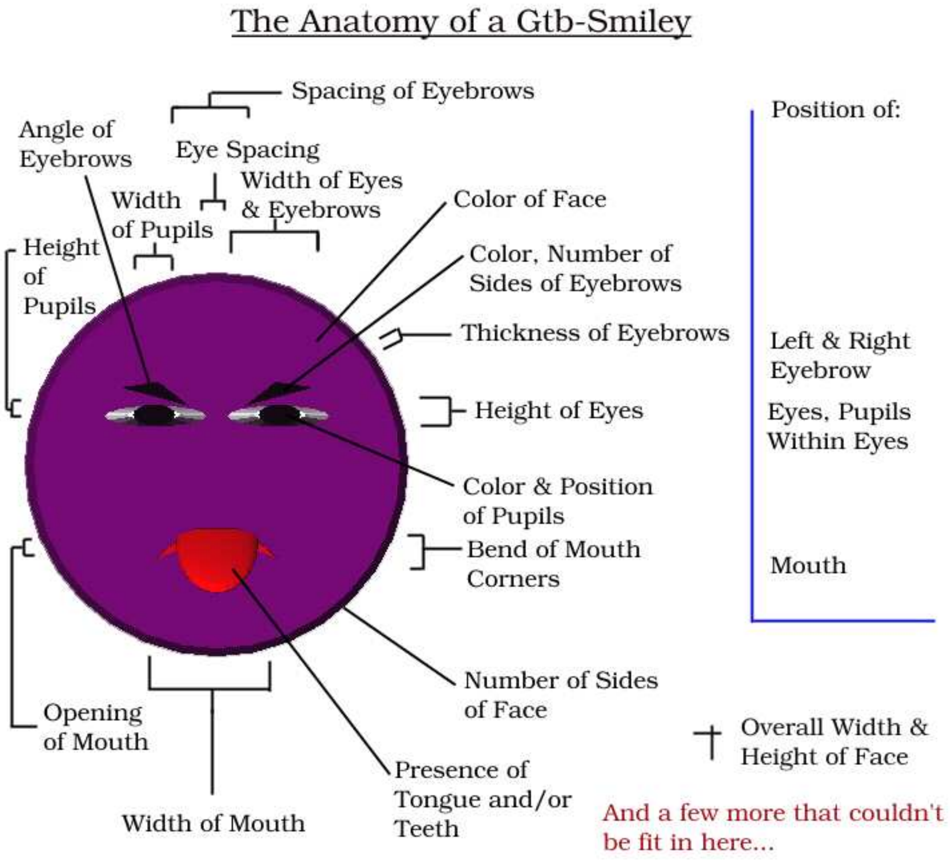


Figure 3.4: Blended gtb-smiley sequence from “happy” to “impassive”

formation as Chernoff’s variable face. It is meant to offer another visualization strategy, and like any strategy it is better used under some circumstances than others. The program and file format was designed with expansion and revision in mind, so the possibility of refining and building on gtb-smiley remains high.



Chapter 4

Conclusion and Future Work

4.1 Applications

The Glyph Toolbox provides a simple format for 3D glyphs. Moreover, complex objects can be created out of simple ones. Color and opacity are included in the format.

The actual application of the Glyph Toolbox is up to the user. The Toolbox contains many functions, and none of them have to be used in any fixed way. Basic tabular data can be fed into functions such as gtb-surf and gtb-vol to generate colored surface plots or wire-frame blocks for “quick-and-dirty” visualization of mass numerics. Values with associated coordinates but no inherent continuity can be plotted volumetrically using gtb-confetti, or drawn out of a larger data set using certain gtb-vol options. Data, especially coordinate data, representing continuous streams of values can be visualized with gtb-sweep before or after being connected by gtb-curve. An assortment of orderable glyphs, such as gtb-meterbar, gtb-icecream and the collection of platonic solids, can be used to show variability in a single data point or type. A smaller assortment of more advanced glyphs is available to represent recognizable objects such as birds, fish, robots and emoticon (smiley) faces. All glyphs can be recolored at least in part, adding another dimension of visual representation, and if the commands provided aren’t sufficient, the rendering-independent GTB file format is based on simple ASCII and can be used to create custom objects. There are even GTB programs available to create simple point sets, such as regular polygons, for use in other objects and to add finishing touches, such as breaking complex concave faces into triangles, to ensure custom objects are displayed properly.

4.2 Conclusion

The intent of the Glyph Toolbox project was to create a unified group of commands improving on the tool set already in use by SAVG. While there is little way to gather numeric data on the degree of the project’s success, there are now many more options available for visualization than there were before its completion. No single “holy grail” of visualization has been found, and it is doubtful one ever will be, so finding and making new ways to view data is still very much an important endeavor. Thus, regardless of how much and how well the Glyph Toolbox is used, it can be considered a success simply for being what it is and doing, reliably, what it was designed to do.

4.3 Problems Solved & Unsolved

A number of problems were faced in the creation of the Glyph Toolbox, some due to the difficulty of achieving the desired effect. While many of these problems were solved, some remain, although their effects have been mitigated to the greatest extent possible.

Many of the Glyph Toolbox commands are to some extent limited in the number of points or faces that they are capable of handling. A few limitations remain in certain functions, but most have either been removed entirely or reduced in severity, usually to the point where such functions meet or exceed the capacity of the viewers and renderers on the computers currently used. Even so, given the rate of technological advance, many functions may require revision in a matter of years. One or two errors also remain in certain functions regarding use of certain “unexpected” values, even if the values are not large. Nearly all such errors that have been encountered have been fixed, but in the time provided the group was unable to test all possible scenarios, so some revisions may still be required at a later date.

Not all problems were algorithmic, some were conceptual. Something as simple as naming programs posed a problem. It is important for the naming system to be simple and consistent, or the commands will become difficult to use. The group wished to conform to existing naming standards, but also found it necessary to develop their own. At times the standard had to be revised in order to promote simplicity and consistency. At these times it was necessary to go through the entire Glyph Toolbox, change function names and update all other functions and documents using or referencing the functions to support the new standard. Similar work needed to be done when function parameter formats were revised.

Disclaimer

Bibliography

[1] Glyph Library [online]. http://www.iavsc.org/repository/express/source/misc/gly_lib/doc.html.

[2] Graphics, Animation, and Visualization Lab;Computer Science and Electrical Engineering Department; University of Maryland Baltimore County [online]. http://www.cs.umbc.edu/research/projects/InfoVis/.

[3] User-Defined Virtual Environments for Manipulation and Analysis of Multivariate Data[online]. http://www.cc.gatech.edu/scivis/research/ve/ve.html.

[4] Visualization Of Current Fields In A Single Blade Pump[online]. http://www.cc.gatech.edu/gvu/datavis/projects_98/frank/.

[5] Glyph Styles[online]. http://www.cs.uni-magdeburg.de/˜nroeber/english/lecture_viz_final.html.

[6] dSPECT Vector Visualization for Parametric Display[online]. http://www.cs.uni-magdeburg.de/˜nroeber/english/viz.html.

[7] *Visualization of Multivariate Data Using 3D and VR Presentations*, volume 1, September 1997. http://www.iavsc.org/repository/express/source/misc/gly_lib/doc.html.

[8] The Cardiovascular Laboratory: Immersive 3D Unsteady Flow Visualization [online]. http://www.cs.brown.edu/˜jsobel/flow.pdf.

[9] *Two-handed Interactive Stereoscopic Visualization*. IEEE Computer Society, 1996. October 29 - November 1, San Francisco, CA, USA,http://www.erc.msstate.edu/conferences/vis96.

[10] *UFLOW: Visualizing Uncertainty in Fluid Flow*, 1996. October 29 - November 1, San Francisco, CA, USA,http://www.erc.msstate.edu/conferences/vis96.

[11] *Directional Flow Visualization of Vector Fields*, 1996. October 29 - November 1, San Francisco, CA, USA,`http://www.erc.msstate.edu/conferences/vis96`.

[12] *Interactive Lens Visualization Techniques*, October 1999. San Francisco, CA, USA,`http://www.erc.msstate.edu/conferences/vis99`.

[13] *Procedural Annotation of Uncertain Information*, October 2000. Salt Lake City, UT, USA,`http://www.erc.msstate.edu/conferences/vis00/`.

[14] OpenDX [online]. `http://www.opendx.org`.

[15] Brian W. Kerighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, Inc., 1984.

[16] The Unix Operating System [online]. `http://www.cbbrowne.com/info/unix.html`.

[17] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, 1994.

[18] H. Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association*, 68:361–368, 1971.

Appendix

Appendix A

Appendix

A.1 Glyph Toolbox Format

GTB format files are written in standard ASCII and contain only data on point coordinates, point colors, and point connections that form faces. Any line in a GTB file can start with a "#" character instead of a numeric value. Such lines are ignored by all GTB commands as comments. Discounting comment lines, GTB files are divided into three sections.

The first section is a single line containing two space-delineated values. The first value is the number of points in the object, the second is the number of faces (a "face" consists of a connection between two or more points, only faces are actually displayed when an object is viewed).

The second section consists of a series of lines of coordinate values with optional color values. Each of these lines corresponds to a single point in the object. Each line must contain, in order, the x, y, and z coordinate of the point, delineated by spaces. Following the coordinates, four more space-delineated values may be optionally included. These values, all between 0 and 1, inclusive, are the red, green and blue color intensity values (0=dark, 1=bright) and the alpha transparency value (0=transparent, 1=opaque). If the last four values are omitted, the point takes the color of the point described by the previous line, or opaque white if no values are given on the first point line.

The third section consists of a series of lines of point indices, each line describing one face of the object. All values in this section are space-delineated integers. The first value of every line is the number of points in the face. Following that are the indices (1-based, with 1 being the point defined by the first line of the points section, 2 being the point defined by the second, and so on) of all the points in the face in the order in which they should be connected. The direction around a face in which points are connected determines the side of the face which will reflect light. Vertices must be ordered counter-clockwise when viewed from the side that should reflect light

Many GTB filters and functions will ignore or fail to transfer comment lines following the last face data line.

The following is an example of a valid GTB file, with comments on format:

```
# Example file: a cube
# 8 points, 6 faces
8 6
# x=-.5, y=-.5, z=-.5, no color so point will be opaque white
-0.500000 -0.500000 -0.500000
# x=.5, y=-.5, z=-.5
0.500000 -0.500000 -0.500000
# etc.
0.500000 0.500000 -0.500000
-0.500000 0.500000 -0.500000
-0.500000 -0.500000 0.500000
0.500000 -0.500000 0.500000
# x=.5, y=.5, z=.5, and a color r=0, g=1, b=.5, a=.8,
# or 80% opaque cyangreen, has been defined
0.500000 0.500000 0.500000 0.000000 1.000000 0.500000 0.800000
# this point will be the same cyan-green as the previous point
-0.500000 0.500000 0.500000
# 4 points in this face, they are the ones defined in point lines
# 4, 3, 2, and 1 respectively
4 4 3 2 1
4 3 7 6 2
4 7 8 5 6
# if this next line were, say, 4 5 1 4 8, the face would be in the
# same position, but would reflect light only from inside the cube,
# being either matte brown
# or invisible from the outside
4 8 4 1 5
4 8 7 3 4
4 1 2 6 5
```

A.2 Glyph Toolbox Command Summary

A summary of the Glyph Toolbox commands, parameters, and English pronunciations.

COMMAND	PARAMETERS	ENGLISH
gtb-arrow	[−h val val val][−t val val val]	arrow
gtb-axis	none	axis
gtb-bird	none	bird
gtb-bound	[−−debug]	boundaries
gtb-breakface	none	break face (triangle mesh)
gtb-chernoff	none	Chernoff face to gtb-smiley data converter
gtb-clock	[−h val][−m val][−s val][−c] [−a val val val]	clock
gtb-color	[−r val][−g val][−b val][−a val]	color
gtb-confetti	−−points file.gtb −−arg argfile.txt	confetti
gtb-curve	[−b | −c | −p][−o val][−n val]	curve
gtb-dodeca	[−d val][−v]	dodecahedron
gtb-dualtetra	[−r1 val][−g1 val][−b1 val] [−r2 val][−g2 val][−b2 val] [−r3 val][−g3 val][−b3 val]	dual tetrahedrons (and hexahedron)
gtb-extr	[−x val][−y val][−z val][−−debug]	extrude (points and lines)
gtb-face	none	face
gtb-fish	[−−fins][−−eyes][−−all]	fish
gtb-flag	[−r val][−g val][−b val]	flag
gtb-hexa	[−d val]	hexahedron (a.k.a. cube)
gtb-icecream	[−s val]	ice cream
gtb-icosa	[−d val]	icosahedron
gtb-join	none	join
gtb-meterbar	[−r val][−g val][−b val][−l val]	meter bar
gtb-ngram	[−n val]	a regular n-gram
gtb-octa	[−d val]	octahedron
gtb-person	[−−left direction][−−right direction]	person glyph
gtb-pie	[−d val][−h val][−a val][−r val]	pie
gtb-plane	none	split faces into coplanar parts
gtb-polar	[−a val][−m val][−−debug]	polar
gtb-prism	[−x val][−y val][−z val][−l val] [−c val]	prism
gtb-pull	[−t file][−q][−r][−o val][−−debug]	pull
gtb-pyramid	[−l val][−h val][−x val][−y val]	pyramid

gtb-range	[−x][−y][−z][−c][−q][−a]	range
gtb-redlin	none	redundant line (removal of)
gtb-regpoly	[−n val][−d val][−f]	regular polygon (2D)
gtb-relhi	−d data.gtb −m mask.gtb −o out.gtb	relationship highlighter
gtb-rmcom	none	remove comments
gtb-rotate	[−x val][−y val][−z val][−r][−−debug]	rotate
gtb-scale	[−x val][−y val][−z val]	scale
gtb-smiley	[happy | sad | angry | impassive | surprised | dead | evil | foo | question | file-filename | −−part val emotion1 emotion2][−−debug][−−vals]	smiley face
gtb-sor	[−x val][−y val][−z val][−n val]	surface of revolution
gtb-sphere	[−d val][−a val][−n val]	sphere
gtb-superellipsoid	[−n val][−p1 val][−p2 val]	super-ellipsoid
gtb-supertoroid	[−n val][−p1 val][−p2 val][−i val] [−o val]	super-toroid
gtb-surf	[−t file][−r file][−g file][−b file] [−q][−d val][−−debug]	surfaces
gtb-sweep	[−−ribbon | −−rope | −−read file.gtb] [−r val][−d val]	sweep
gtb-tetra	[−d val]	tetrahedron
gtb-text	[−w val][−h val][−s val][−x val] [−y val][−z val][−f val][−−debug]	text
gtb-translate	[−x val][−y val][−z val]	translate
gtb-unit	none	unit
gtb-vol	[−r file][−g file][−b file][−rt val] [−gt val][−bt val][−f][−d val] [−−debug]	volume
gtbtoiv	[−r][−rs val][−−VRML][−−debug]	gtb-to-inventor converter