The Graph Example

Overview

Graph plots 2D line graphs of functions of the form y = f(x) and 3D surfaces given by the set of equations x = f(u,v); y = f(u,v); z = f(u,v). Important things that Graph demonstrates are being a source of Object Links, use of the 3D Kit and general NeXTSTEP programming of a small, multi-document application.

The heart of the application is the **Expression** class. Objects of this class parse the text of mathematical expressions, and can then evaluate the expressions given a set of values for the variables found in the expression. The Expression class gives Graph a great flexibility in that it can graph novel

expressions, intead of being limited to a pre-compiled set.

The Expression class is built using the Unix tools **yacc** af**lex**. The parsing code will probably be confusing for anyone not familiar with these tools. However, the Expression class is designed to have a sufficiently complete interface that it can be used without understanding these internals. Hopefully, between the documentation in the header file and the example usage in this program will allow others to incorporate this object in other applications. The ability to dynamically interpret new expressions should greatly enrich applications which make modest use of formulas.

The rest of the application is fairly standard NeXTSTEP programming, using the Expression class' abilities to create the data points for 2D and 3D graphs. A Display PostScript userpath is used to draw the line graph, and RenderMan is used to image the 3D surfaces. Both types of documents

function as Object Link sources.

Road Map

Expression.m implement the expression class. They are supported by the files **expr.ym** (a yacc source file), **token.lm**(a lex source file) and **exprDefs.h** (some common declarations shared between these files). The files **expr.m**, **token.m**, **y.tab.h** are by-products of the yacc and lex phases of the build.

GraphDoc.m and **GraphDoc.nib** implement a 2D Graph document. For each 2D Graph document, a GraphDoc object is created and a instance of the Graph document user interface is loaded from GraphDoc.nib. The GraphDoc object coordinates the user actions performed on the various

controls, and uses an Expression object and a LineGraph view to create the graphs requested. Object Link support is implemented in GraphDoc.m.

LineGraph.m implement a view which knows how to draw an xy line plot. It uses Display PostScript userpaths for top performance.

Graph3DDoc.m and **Graph3DDoc.nib** implement a 3D Graph document. For each 3D Graph document, a GraphDoc object is created and a instance of the 3DGraph document user interface is loaded from Graph3DDoc.nib. The Graph3DDoc object uses three Expression objects to evaluate the user's equations, a PointMesh surface to render the data and a RotatorCamera to view it. Object Link support is implemented here also.

PointMesh.m is a subclassgN3DShape, which uses a RenderMan bilinear patch mesh to render a two dimensional mesh of points in 3-space

(the sort of surfaces Graph plots).

RotatorCamera.m is a small subclass of N3DCamera. Its only purpose is to implement the mouse tracking needed to use a N3DRotator to rotate the graph.

ThreeDPanel.m and **ThreeDPanel.nib** implement a very simple accessory panel to allow the user to change a few attributes of the 3D graphs.

GraphApp.m and **GraphApp.h** implement the delegate of NXApp. There is one instance of this class in the entire application. This class mostly responds to non-document specific commands, such as putting up the Help or Info panels. It also receives messages from the Workspace Manager to open documents. Finally it is the keeper of the application's

NXStringTable, which holds the strings used by Graph, translated to a particular language.

GraphApp.nib holds the global user interface for the program, such as the menus. It also holds the application's small NXStringTable. These strings are loaded from the file **Graph.strings**. **Help.nib** and **Info.nib** hold the user interface for the Help and Info panels.

AppIcon.tiff, **DocIcon.tiff** and **DocIcon3D.tiff** hold the images for the application and document icons. **Graph.h** is used to make the precompiled header file, **Graph.p**. **Makefile**, **Graph_main.m**, **PB.project**, **PB.gdbinit** and **Graph.iconheader** are the usual files that Project Builder manages. **Makefile.postamble** and **Makefile.preamble** contain a few additions to the build process. **vers.c** is a derived file created by the **vers_string** command as part of the build process. It contains some useful version information

about the program.

Highlights

GraphDoc and Graph3DDoc have the minimal amount of code needed to make a simple document be an Object Link source.

The PointMesh class shows how to turn a set of data points in 3-space into a surface RenderMan can display.

The RotatorCamera shows the minimal glue needed to use the Nhtator class to manipulate an Object. However, there should be more user feedback during the rotation, such as a lightly overlaid virtual sphere to give the user some idea of the rotation axes he is working with.

GraphDoc's and Graph3DDoc's **-copyGraph:** methods are a good example of how simple it is to implement copying a view's PostScript or RenderMan into the Pasteboard. Normally this wouldn't be a separate command in the user interface, but this was done because there is no notion of selection in the view where the graph is drawn.

The GraphApp class is a good example of the minimal amount of glue you need to respond to Workspace messages to open documents (see **-appOpenFile:type:** and **-appAcceptsAnotherFile:**). The **-appDidInit:** method shows how to open a new document when launched if the user didn't double click on a document.

GraphDoc has a good use of TypedStreams for reading and writing its documents.

The save methods in GraphDoc might be helpful in dealing with the various cases of saving.

LineGraph has a nice example of userpaths. It stores its data in a form that can be directly passed to **DPSDoUserPath()**.

In LineGraph.m, **drawSelf::** and **drawAxes()** have a PostScript trick of maintaining a consistent line width regardless of how the view has been zoomed.

Graph takes a fairly minimal approach to localization. Since it has so few strings, the strategy of putting them in one global NXStringTable works fine. Any real application will want to break up their strings into tables along functional boundaries.

Less exemplary parts (exercises for the reader?)

GraphDoc and Graph3DDoc share a lot of code. Some common functionality could certainly be factored out into an abstract super-class, eliminating the current code duplication.

Graph needs some good icons for itself and its documents!

A SplitView couple be used to enlarge the viewing area oie graphs. I think it would be best to slide the lower half of the window down instead of covering it from top to bottom, since the parameter sliders are more useful than those controlling the domain.