TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY
Eindhoven University of Technology

MASTER

A new graphics interface for the ARCS antenna- and RCS-measurement system

Geraets, A.G.

Award date:
1991

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

ED 544
5826

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
Digital Systems Group

A New Graphics Interface for the ARCS Antenna- and RCS-Measurement System

A.G. Geraets

Master's Thesis

Supervisor: Prof. ir. M.P.J. Stevens
Coach: Dr. ir. V.J. Vokurka
Date: january 1991 – october 1991

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of graduation reports.

Abstract

This is a report about my graduation project carried out at the Eindhoven University of Technology. It handles about the redesign of the graphics part of the ARCS antenna- and RCS-measurement software package.

With this new software it is possible to graphically display data-files in many ways: 2D cartesian, polar, 3D cartesian, contour, etc. Other features include support of markers (interactive placable) and support of various output deviced (plotters, printers and the X11 display).

The program is an OSF/Motif application and makes use of many features of the X Windowing System and the OSF/Motif Windowing Toolkit.

Although the application is designed after specifications of the antenna- and RCS-measurement system, it should be usable for many other applications too.

Acknowledgements

I wish to thank the following persons:

- My coach dr. Vokurka for making this graduation project possible and his support during the project.
- Professor Stevens for his support during my graduation project.
- Paul Derks for finding bugs when I couldn't find them, and the conversation resulting in many new ideas.
- Bert Schluper of March Microwave Systems B.V. for the many conversations and advice and support.
- Tony Richardson for writing **plplot**.
- Sergey Oboguev for giving me the solution to an irritating problem with the X11 color support.
- Many other persons on the internet answering questions in the **comp.windows.x** and **comp.windows.motif** newsgroups.

Contents

1 Introduction **1**
- 1.1 Antenna and Radar Cross Section measurements 1
- 1.2 ARCS Software package 2
  - 1.2.1 Acquisition 2
  - 1.2.2 Data analysis 3
  - 1.2.3 Display software 4

2 Specification **5**
- 2.1 Introduction 5
- 2.2 Necessary functionality 5
  - 2.2.1 Operating modes 5
  - 2.2.2 Supported output devices 6
  - 2.2.3 Concurrency 6
  - 2.2.4 History of settings 6
  - 2.2.5 Interactive placable markers 7
  - 2.2.6 Device independency 7
  - 2.2.7 Plot types 7

3 Implementation **13**
- 3.1 Introduction 13
- 3.2 Concurrency 13
- 3.3 Application structure 14
- 3.4 User interface 14
- 3.5 Processing 15
- 3.6 Graphics library 15

4 Database for graphics application. **17**
- 4.1 Introduction 17
- 4.2 Description of the data in the database 17
  - 4.2.1 Axis 17
  - 4.2.2 Line types 18
  - 4.2.3 General 18
- 4.3 Description of the per plot data 20
  - 4.3.1 Single 2D cartesian graph 20

4.3.2 Single 2D polar graph 20
4.3.3 Single 2D polar graph with "hole" 21
4.3.4 Double 2D cartesian graph vertically split 21
4.3.5 Double 2D cartesian graph horizontally split 21
4.3.6 Double 2D polar graph horizontally split 21
4.3.7 2D cartesian and 2D polar graph horizontally split . . 22
4.3.8 3D cartesian graph 22
4.3.9 Contour graph without margin graphs 23
4.3.10 Contour graph with margin graphs 23
4.3.11 Polar contour plot 23
4.3.12 Polar contour plot with "hole" 23
4.4 Some random considerations 24

5 Implementation of markers 25
5.1 Introduction 25
5.2 Types of markers 25
5.3 Marker placement modes 26
5.4 Visual placement of markers 26
5.5 Positioning 27
5.6 Marker placement 28
5.7 Marker removal 30
5.8 Implementation 32
5.8.1 `X11locator()` 32
5.8.2 `X11setmarker()` 33

6 Font Support 35
6.1 Introduction 35
6.2 The Hershey Font Set 36
6.3 The advantages of the Hershey font set 36
6.4 New position of the font support routines 37
6.5 Interface routines 37
6.5.1 `hershey_get_font_name()` 38
6.5.2 `hershey_parse_string()` 38
6.5.3 `hershey_draw_string()` 39
6.5.4 `hershey_free_string()` 40
6.6 The internal workings 40
6.6.1 The hershey character and glyph data structures . . . 40
6.6.2 The `HersheyString` data structure 43

7 Styles 45
7.1 Introduction 45
7.2 The concept of styles 45
7.3 Contents of a style 46

7.4 Interface functions 46
7.4.1 The StyleValues structure 46
7.4.2 dev_selectstyle() 47
7.4.3 dev_setstyle() 48
7.4.4 dev_getstyle() 48

8 The device drivers **49**
8.1 Introduction 49
8.2 The original device driver programming interface 49
8.2.1 Removed device driver interface functions 51
8.3 Extending the use of device capabilities 51
8.4 Overview of added device driver interface functions 52

9 Description of the X11 device driver **53**
9.1 Introduction 53
9.2 Refresh and resize 53
9.3 Markers and device input 54
9.3.1 Device input 54

10 User Interface **57**
10.1 Introduction 57
10.1.1 X11 environment setup 57

11 Conclusions **59**
11.1 X11 and OSF/Motif programming 59
11.2 Status of the project 59

Bibliography **61**

A Contour Literature **63**
A.1 Introduction 63
A.2 Contour literature list 63

Chapter 1

Introduction

This report describes the design of new graphics interface for the ARCS software package carried out as graduate work my study information technology. The ARCS software is used heavily in the antenna laboratory of the Theoretical Electrotechnology Group of the Eindhoven University of Technology. However, one problem was the aging of the software as it was designed about five years ago and the continuous modifications made a rewrite from scratch of great parts necessary.

1.1 Antenna and Radar Cross Section measurements

Antenna's are used for information transmission thourgh a space. Before an antenna can be used we generally want to know its behaviour. Therefore we want to measure this behaviour. This can be done by measuring the signal —sent by the antenna we want to measure— picked up by an antenna with known characteristics which is (and visa versa), provided that it is measured in an environment with known characteristics (e.g. positions of the two antenna's and influence of signal reflecting/absorbing objects).

Because most antenna's are designed for relative great distances, measurement should also be carried out with a great distance between the two antenna's, to ensure the same shape of incoming/outgoing wave-front (flat), in both real-world applications and measure-system. However this has some disadvantages:

- Much space is needed for the measurement.
- The environmental influence on the measurent is difficult to measure.

The Compact Antenna Test Range (CATR) is often used to overcome these problems. In a CATR, the so called far-field characteristics are simulated with parabolic reflectors (these create a flat wave-front from a point-source). Figure 1.1 shows the layout of such a CATR.

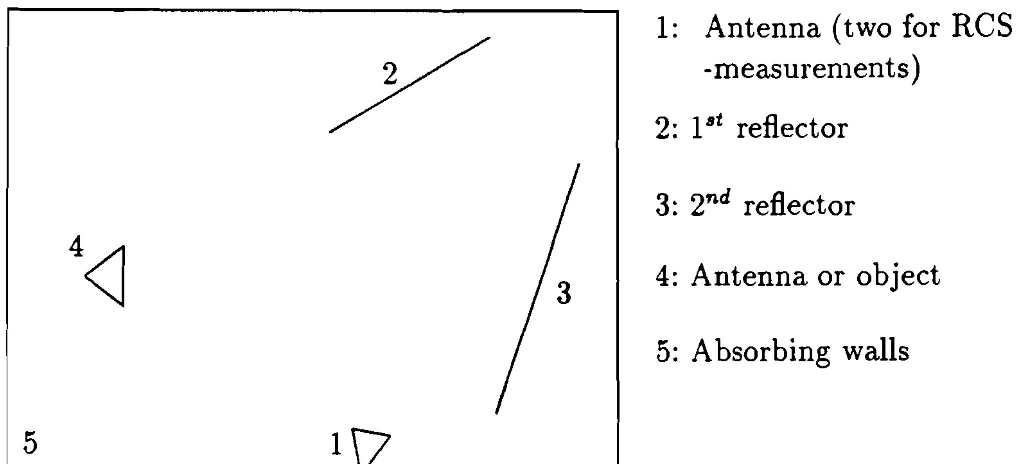

Figure 1.1: Compact Antenna Test Range

With Radar Cross Section (RCS) measurements, we measure the reflections of the objects to be measured. From these measurements we can calculate an image of the object, in which we can see the radar reflecting parts of the object.

1.2 ARCS Software package

ACRS (Antenna and Radar Cross Section measurement system) is an integrated software package for antenna and/or RCS measurements. The package includes:

- data acquisition
- data processing and analysis
- graphic data representation

1.2.1 Acquisition

The data acquisition program controls the RF system and the positioner controller to perform measurements of RCS or power vs. frequency and/or angle. Time-domain operations (range transforms and software gating) can be performed on all measurement types, either in the receiver hardware or in the computer. Computer time-domain transformations are performed concurrently with the measurement, making gated measurements as fast as ungated measurements.

Antenna measurement types are:

- Power vs. frequency and distance.
- Power vs. angle.

- Antenna calibration (absolute gain or normalization).

Power vs. angel measurements record the antenna response vs. frequency while moving the antenna in a sector or raster scan. The slew axis allows measurements with continuously rotating polarization. A real-time plot diplays the measured signals during the measurement.

Power vs. frequency and distance measurements recore the antenna response vs. frequency, and the (optionally) perform software gating and time-domain transformations.

RCS measurement types are:

- RCS vs. frequency and down-range.
- RCS vs. angle.
- RCS imaging.
- RCS calibration (absolute RCS or normalization).

RCS vs. angle measurements record the target response vs. frequency while moving the target in a sector or raster scan.

RCS imaging measurements are similar to RCS vs. angle measurements. The acquisition parameters (angle step, angle span and number of frequencies) are derived from the imaging parameters (cross-range cell-size and target size).

RCS vs. frequency and down-range measurements record the response vs. frequency, and then (optionally) perform software gating and time-domain transformations.

Once an antenna or RCS calibration set is available, all subsequent measurements are automatically calibrated. Alternatively, uncalibrated measurements can be done, which can than be calibrated afterwards.

1.2.2 Data analysis

The ARCS software package includes several data analysis programs, which can be used for both RCS measurements and antenna measurements. These inlcude:

IMAGE performs ISAR imaging on data acquired vs. frequency and aspect angle using various window functions and resolutions.

IMGATE allows modification of image data and then transforms to the frequency and angle domain. Parts of the image can be removed or isolated, and the resulting RCS is calculated. Furthermore the intensity of individual scatterers can be increased or reduced by a specified number of dB's. Various gate shapes are available.

TIME and **GATE** are programs that perform 1-dimensional time-domain transformations and gating, respectively, on data vs. frequency and aspect angle.

FILMATH is a file mathematics program that allows adding, subtracting, dividing and multiplying data files.

FILCOM combines 2 data files with overlapping frequency bands into 1 file, for high resolution imaging.

SMOOTH is a program for data smoothing. It supports various averaging methods.

THINOUT is a program for data reduction. It creates a subset from a large dataset.

Furthermore, for RCS measurements, simulation programs are included that calculate the RCS of targets consisting of points scatterers.

For radiation pattern analysis, a program is included to calculate gain, beamwidth, sidelobe levels and positions, and null levels and positions.

1.2.3 Display software

The original display software consists of two programs —MENU2D and MENU3D— which can display any 1–D or 2–D cut from a data-file. Two independent channels are available for overlaying seperate data-sets (from the same file or from different files).

Plot types supported by MENU2D are: cartesian, polar, horizontal split, vertical split, and polar & cartesian combinations.

Plot types supported by MENU3D are: 3–D (waterfall, isometric and dimetric projections), contour (with or without margin plots) and polar contour.

The programs support a number of marker to mark interesting points in the plots. The position of the marker can be diplayed absolute or relative to another (reference) marker. Other marker functions are absolute maximum/minimun and local maximum/minimum search.

Chapter 2

Specification

2.1 Introduction

The purpose of the graphics application is twofold:

- show data, produced by the ARCS antenna and RCS measurement system, graphically,
- make some kinds of data input easier.

Graphics support is needed in the following applications:

stand alone graphics application
Mainly for graphics representation.

IMAGE
Graphics presentation and to specify several input values.

IMGATE
Graphics presentation and to specify a surface to be gated.

Because graphics functionality is needed in more than one program, it should be modular, to make it easy to include in a program as a module.

2.2 Necessary functionality

The list of necessary functionality is determined from various sources:

- The current ARCS software.
- Conversation with users.
- The competitors.

The results follow.

2.2.1 Operating modes

Ideally the application must be able to be driven in a number of ways:

Graphical user interface (GUI)
The user can modify the settings of various parameters by (for example) pointing and clicking with the mouse.

Batch-mode
If a batch of plots has to be made in the same way, its convenient to be able to describe in a file what must be done to generate this output. For this mode we need a description language to describe the settings, to enable the user to modify them himself. (Of course there also must be a possibility to capture the operations done interactively with the GUI).

Command line interpreter (CLI)
Because we have a command language available (as a result of the batch-mode), we can also use it for interactive input via a command line interpreter[1].

2.2.2 Supported output devices

The graphics application must be able to generate output for a range of output-devices. The most important are:

- X11 display
- HPGL plotter (possibly different types)
- Laser-printer (HP-Laserjet; PostScript)

2.2.3 Concurrency

It must be possible to driver the output devices concurrently, i.e. if we started a plot to a plotter, we must be able to continue working on the X11 display.

2.2.4 History of settings

Because the ARCS software is mainly used within laboratory environments, it should be easy to change various settings of a graph. Also, it should be easy to recall previous settings (e.g. if a the change did not result in the desired effect). Furthermore it should be easy to select a number of default (often used) values.

To hold this history of settings, some kind of a database is needed, which of course, can also hold the current setting.

1. If life is a menu driven universe, how can we escape to a command line interpreter?

2.2.5 Interactive placable markers

The user must be able to place indicators (markers) at points of interrest and, if necessary, get the values of those points. On interactive devices this placement should also be interactive.

2.2.6 Device independency

The graph to be produced must be as independent of the output device as possible, i.e. it should not be necessary to change many parameters in order to display a graph on another device.

2.2.7 Plot types

The user must be able to represent some data in many ways. Therefore the application should support various plot types. The next sections describe the plot types that should be supported.

2D cartesian graph

The 2D cartesian graph is the most simple graph of all. Figure 2.1 show an example of such graph. Many variations with the axes are possible, such as for example a logarithmic distribution.

2D polar graph

A polar graph is often used if one of the axes represents an angle. Figure 2.2 show an example of a 2D polar graph.

2D polar graph with hole

If we want to show data of a RCS-measurement in a polar graph, we sometimes want to make clear the connection between the shape of the object and the measerement data. With a 2D polar graph with a hole, we are able to do this by projecting the shape of the object in the hole, as can be seen in figure 2.3.

3D cartesian graph

A 3D cartesian graph is often used to display a parameter as a function of two variables. There are various methods to display this:

- waterfall

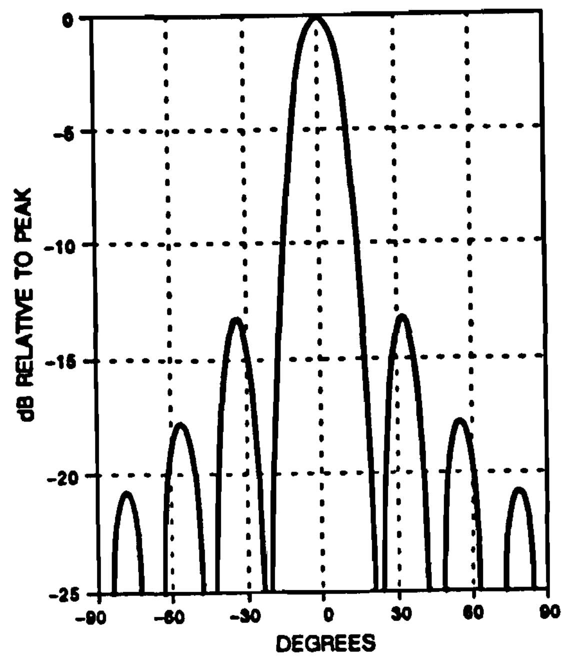

Figure 2.1: Example of a 2D cartesian graph.

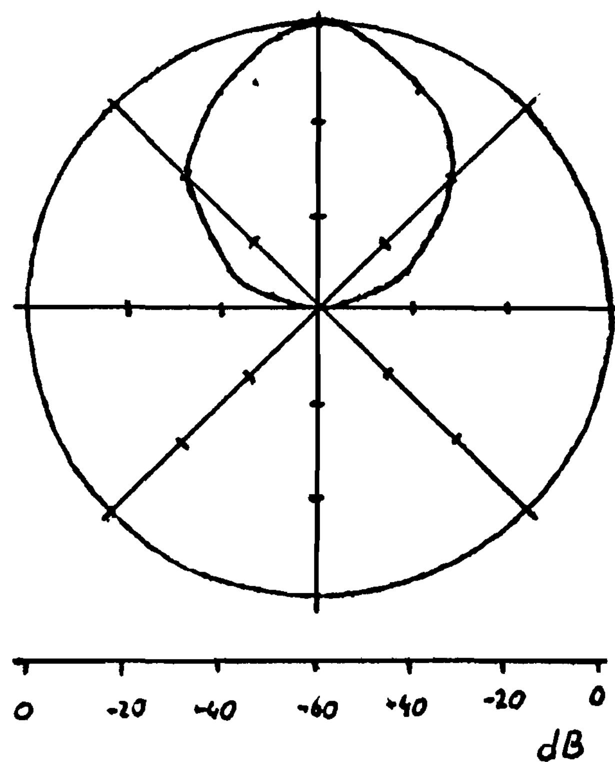

Figure 2.2: Example of a 2D polar graph.

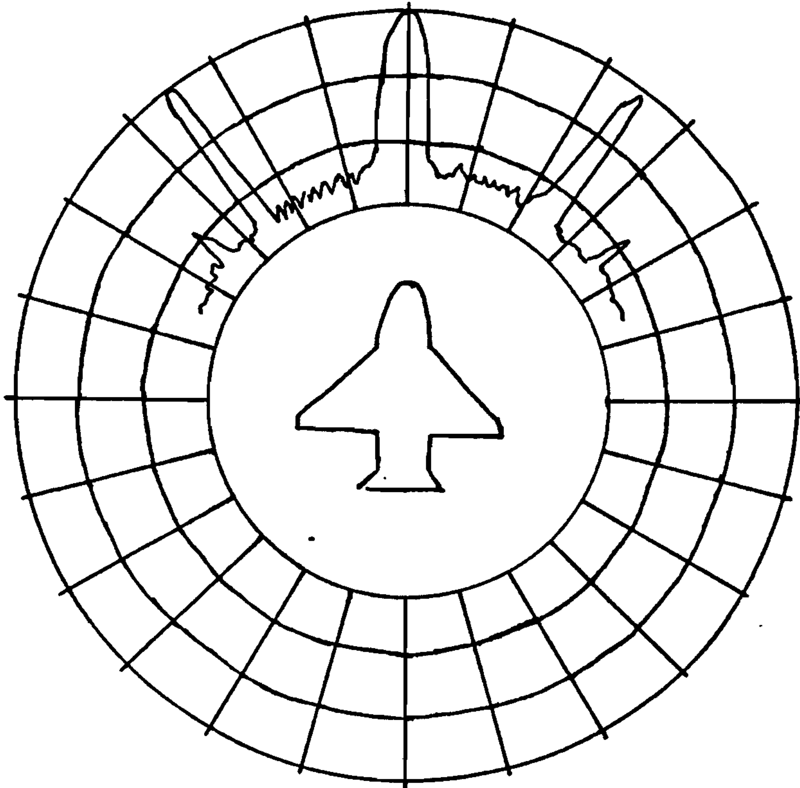

Figure 2.3: Example of a 2D polar graph with hole.

- dimetric
- isometric
- perspective

The perspective view is not very usefull for our application, because no values can be measured from such a graph. Figure 2.4 shows one of the other 3D representations. In the figure you can also see an object projected above the graph to illustrate the source of the responses in the graph.

Contour graph

Another way to represent 3 dimensional data is with a contour graph, which displays height lines. Figure 2.5 shows an examples of such a contour graph, again with an object projected in the graph.

Contour graph with margin graphs

In a contour graph it is sometimes difficult to imagine the exact "heights". Therefore it must be possible to place margin graphs along the sides of a contour graph. These margin graphs can display a slice through the contour, of for examples the maxima in one direction. Figure 2.6 showns an example.

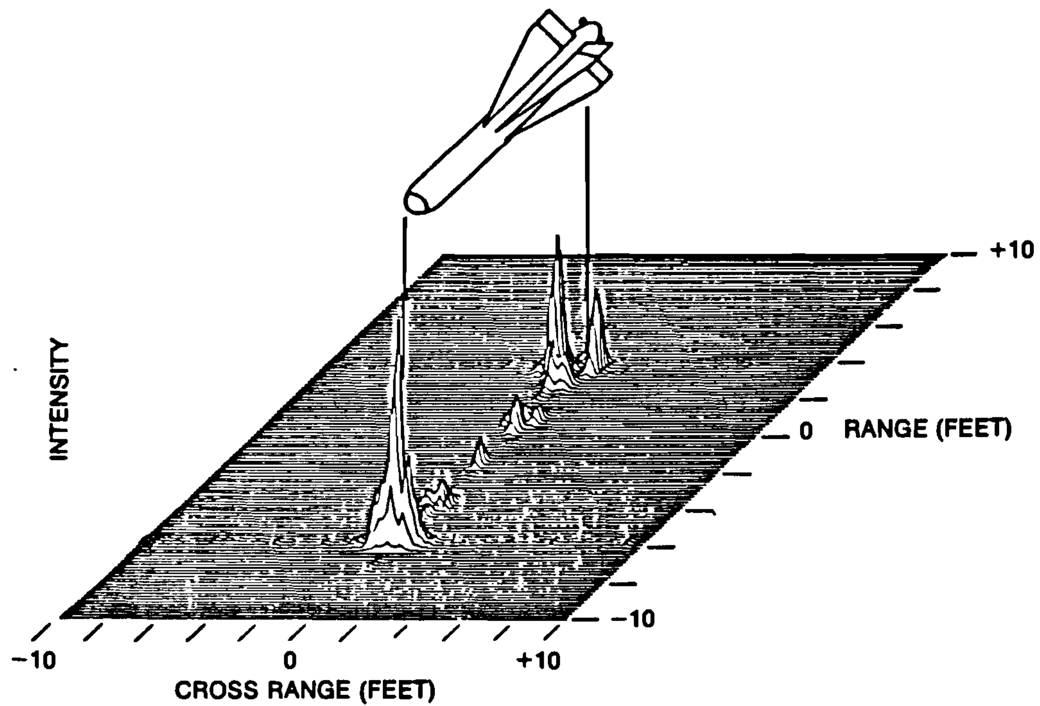

Figure 2.4: Example of a 3D cartesian graph.

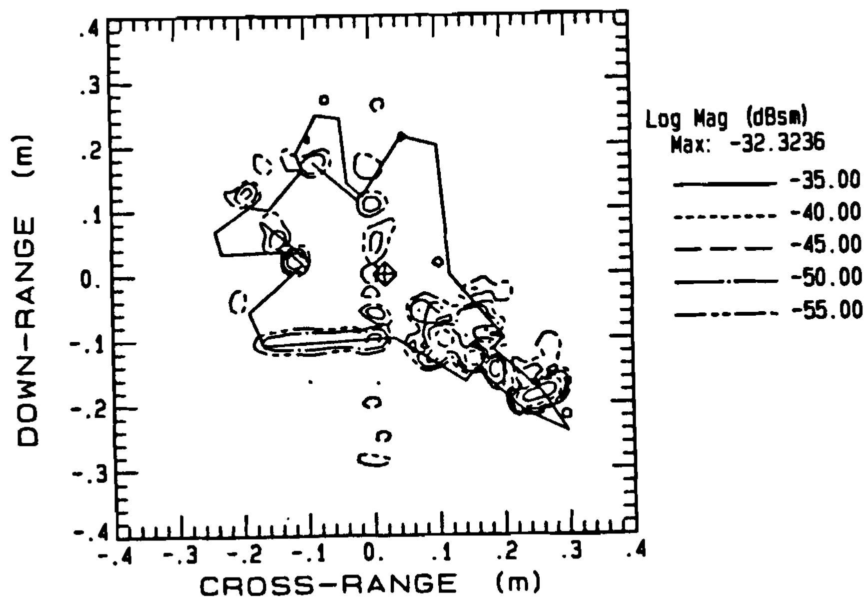

Figure 2.5: Examples of a contour graph with projected object.

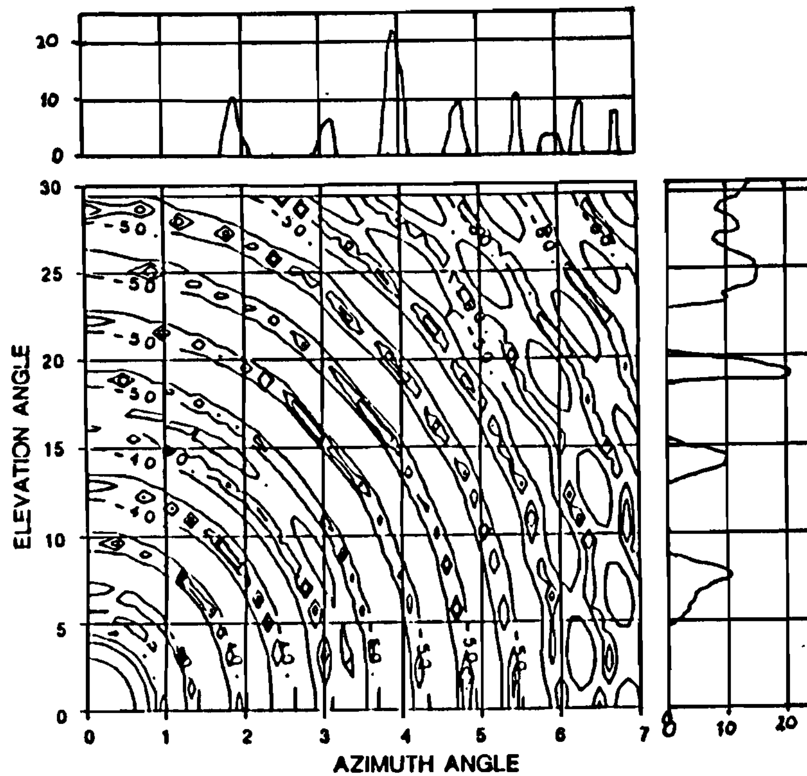

Figure 2.6: Contour graph with margin graphs

Polar contour graph

The polar contour graph is a combination of a polar graph and a contour graph (figures 2.2 and 2.5).

Polar contour graph with hole

The polar contour graph with hole is a combination of a polar graph with hole and a contour graph (figures 2.3 and 2.5).

Chapter 3

Implementation

3.1 Introduction

In the previous chapter a specification of the graphics application was given. So the only thing which has to be done is to implement it.

For the moment we will drop the batch mode operation and the implementation of a command line interpreter. We will see later, that once we have a working graphical user interface working on top of X11, implementation of these two operating modes is rather simple.

Because writing an X-application calls for a programming method called *event driven programming*, most part of the application will already written in such way, that we can easily implement commands for each single action. Writing a command parser will take most part of the implementation of a command line interpreter. To implement the batch mode, we must find a way to automatically generate commands.

3.2 Concurrency

One way to achieve the desired concurrency is to start up an extra process if we want output to another device. Just forking[1] the process isn't enough, because the process is an X11 application and contains a lot of data structures that can no be shared by multiple processes and are difficult to clean up. The method to use is to create a new process.

Somehow the newly created process must be able to take over all settings of the original process. This can be done via a file. Because we already needed a database for some other reasons, it shouldn't be too difficult to save that database for use by the new process.

1. "Create a clone of the original process"

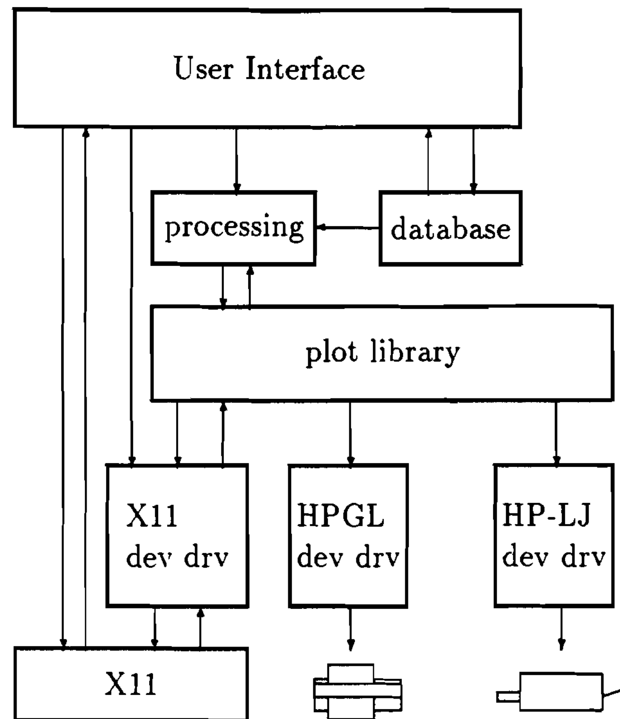

Figure 3.1: Block diagram of the implementation

Another way to achieve the desired concurrency is by exploring the event driven nature of an X11 application and all available X11 functions. The handling of other devices can be done as *workprocs*. For an explanation of this term, please refer to appropriate X11 literature.

3.3 Application structure

A structure for the application follows directly from the specifications. A block diagram of this structure is shown in figure 3.1.

3.4 User interface

The user interface interacts with the user at the top level. It's main purpose is to set-up the necessary X11 and Motif data structures and provide menus by which the user can perform various commands:

- Set-commands: These commands only change some values in the database. These can be subdivided in:
  - Settings that tell what to plot.
  - Settings that tell how to plot. Again these can be subdivided:

  - * Device independent settings.
  - * Device dependent settings.
- Do-commands: These commands perform some kind of action, e.g. plot a graph, place a marker and are passed to the processing.

3.5 Processing

The processing module is activated by the user interface (the user presses the "go" button). Its tasks is it to gather all information it needs from the database and then issues the appropriate function calls to the plot library to draw a graph.

3.6 Graphics library

For the graphics library I adopted[2] a public domain library writen by Tony Richardson named **plplot** as described in [Ric90]. This library already contains a great deal of basic functionality needed to implement graphics application like we need.

Included in the library are (for example):

- Support for many devices.
- Top layer is device independent.
- Many different kinds of standard plots (2D, 3D).
- Plots highly configurable.
- Easy usable.

There are however some things missing from this library and thus have to be added to it:

1. An X11 device driver. An Xterm device-driver, which uses the tek window of an xterm terminal emulator is present, but this is insufficient for our application.
2. Bidirectional devices. **plplot** only supports output-only devices, but as we are making an interactive application, we also need input.
3. Markers. It is already possible to simulate them by "drawing the correct lines," but we need more capabilities, such as dynamically moving and removing.
4. Good font support. The standard font support is very poor, the internals are completely undocumented and it is implemented on application level instead of on device driver level.

2. Why reinvent the wheel?

5. Some plot types which we need for our application:
   - Polar plots
   - Double plots
   - Contour plots with margin graphs.
6. The device's capabillities are not fully explored. This can be improved by changing the device driver interface.
7. Make it a little more device independent. I.e. it is the responsibility of the user (application) to choose the right colors, so he must keep track of the device he wants to output to: if it's a color device, the can use different colors, but if it's a monochrome device, he should rather use dashed lines, or lines of varying width.

How the library is enhanced and changed will be described in the next chapters.

Chapter 4

Database for graphics application.

4.1 Introduction

Although this chapter is a description of the database to be maintained by the graphics application, it also gives a good overview of the desired functionality of the complete application.

4.2 Description of the data in the database

The current settings of the graphics application should be maintained in a database. For each setting a number of previous values should also be maintained for the purpose of easy recalling these old values.

Data is maintained in the following structures:

4.2.1 Axis

Of an axis we want to record the following settings:

- Axis identifier: uniquely identifies this axis-parameter (e.g. frequency).
- Start value
- Stop value
- Axis-type:
  - Linear distribution
  - Linear distribution of logarithmic values (dB-scale)
  - Logarithmic scale: with this axis-type the value of some other elements (e.g. number of sub-ticks) will be useless.
  - Re part
  - Im part
- First tick value (default equal to the start value)

- Scale automatic or manual. If scale automatic is chosen, the program will calculate reasonable values for the number of (sub-)ticks from the start- and stop-values, size of the font, etc.
- Tick value step (if not automatic): the step between two successive major tick values.
- Tick value accuracy (if not automatic): number of significant digits.
- Number of sub-ticks per major tick (if not automatic).
- Unit (e.g. degree, Hz)
- Unit multiplication factor (automatic, μ, m, none, k, M, G, etc.).
- Annotation (e.g. frequency, azimuth, magnitude). Default, this will be the name of the axis (retrieved form the file), but it should be possible to change it.

In addition we record some settings used if the axis serves as a level axis:

- Automatic calulation of levels or manual input.
- Number of levels.
- List of level-values in case the distribution is non-linear.

For axes in general we want to record:

- Font to use for unit on vertical axes.
- Font to use for unit on horizontal axes.
- Font to use for annotation on vertical axes.
- Font to use for annotation on horizontal axes.
- Direction of annotation (parallel/perpendicular) on vertical axes.
- Drawing style to use for axes.

4.2.2 Line types

Of a line type we want to record the style, which is mapped to an actual color, line width, line pattern, etc. within the device driver.

Because this mapping from style to actual line properties can be changed by the user (on a per device basis), we should probably also maintain per device a database of these mappings, so the user is able to restore old values.

4.2.3 General

In general we want to record the following settings:

- Description of all known axes with their history buffers.
- Grid type:
  - Full grid on major ticks;
  - Only vertical or horizontal grid.
  - Full grid also in minor (sub-)ticks;

  - Only ticks on non-empty axis-sides;
  - Also ticks on empty axis-sides;
  - No grid;
  - etc.
- Calibration of output to real-world measures: should a unit on an axis be a real-world unit (i.e. 1 major tick/inch).
- Polar calibration of output to real world angles: in case an angle is plotted against the angular axis of a polar plot, should the displayed angle match the real-world angle.
- Grid type of polar graph:
  - (grid types to be defined).
- Placement of angle 0 in polar graphs (right, top, etc.).
- Should the angle of the polar graph be the same as the range of the angle-axis (i.e. should the graph be a complete circle or only a pie).
- 3D representation:
  - waterfall;
  - dimetric;
  - isometric;
  - etc.
- Plot type:
  - Single 2D cartesian graph.
  - Single 2D polar graph.
  - Single 2D polar graph with "hole".
  - Double 2D cartesian graphs vertically split.
  - Double 2D cartesian graphs horizontally split.
  - Double 2D polar graphs horizontally split.
  - 2D cartesian and 2D polar graph horizontally split.
  - 3D cartesian graph.
  - Contour plot without margin graphs.
  - Contour plot with margin graphs.
  - Polar contour plot.
  - Polar contour plot with "hole".
- Title and further annotation.
- Fonts to be used for title etc.
- Filename of a picture. This is the (2D or 3D) picture which should be plotted:
  - On a contour graph (with or without margin graphs). This works only if both axes display distance.
  - Above a 3D cartesian graph. This works only if both axes (of the ground plane) display a distance.
  - In the "hole" of a 2D polar graph.
  - In the "hole" of a polar contour graph.

- Center (3D-)coordinates of picture (the point which must be positioned in the center of the hole) in source coordinates (coordinates of the file). In fact this is the inverted translation vector.
- Rotation angles of center picture: the angles through which the picture should be rotated. Rotation takes place about the center of the translated picture.
- Default scaling of center picture: the picture is scaled such that it fits exactly within:
  - Prescale the picture so that it fits exactly within a rectangle enclosed by the center hole. (Only for graph with a hole.)
  - Prescale the picture so that it fits exactly within a rectangle which encloses the center hole. (Only for graph with a hole.)
  - The scale factor of the picture is the scaling from picture coordinates to real world units (e.g. 10 picture units is 1 meter). This can only be used for 3D cartesian and normal contour graphs. Using this option implies that size size of the picture varies when changing of the start- and/or stop-values of an axis.
  - No default scaling (scaling factor 1.0).
- Scaling of picture: scaling applied after the default scaling to make the picture fit (for 'fine tuning').
- Extra options for picture display, e.g. draw as wire-frame; draw solid (mentioning these options doesn't imply that these options will be implemented).

4.3 Description of the per plot data

4.3.1 Single 2D cartesian graph

For a single 2D cartesian graph we want to record the following settings:

- Description of the axis on the left of the graph.
- Description of the axis on the rigth of the graph (default empty).
- Description of the axis on the bottom of the graph.
- Dependency between axes in the same direction (e.g. must ticks on the rigth axis be on the same places as the ticks on the left axis). If this dependency is selected, only the tick value step and the number of subticks will be changed to match the other axis. The start- and stop-values won't be changed.
- Size of the graph.

4.3.2 Single 2D polar graph

For a single 2D polar graph we want to record the following settings:

- Description of the radial axis of the graph.
- Description of the angular axis of the graph.
- Size of the graph.

4.3.3 Single 2D polar graph with "hole"

For a single 2D polar graph with a "hole", we want to record the following settings:

- Description of the radial axis of the graph.
- Description of the angular axis of the graph.
- Relative size of the hole (fraction of the radius of the total graph).
- Size of the graph.

4.3.4 Double 2D cartesian graph vertically split

For a vertically split double 2D cartesian graph we want to record the following settings:

- Description of the axis on the left of the bottom graph.
- Description of the axis on the left of the top graph.
- Description of the axis on the bottom of the graph.
- What axes of the two graphs must be coupled, i.e. have the same scale (e.g. both left axes.). This is only valid for axes of the same parameter type (e.g. two phase axes).
- Size of the graph.

4.3.5 Double 2D cartesian graph horizontally split

For a horizontally split double 2D cartesian graph we want to record the following settings:

- Description of the axis on the left of the left graph.
- Description of the axis on the right of the right graph.
- Description of the axis on the bottom of the left graph.
- Description of the axis on the bottom of the right graph.
- What axes of the two graphs must be coupled, i.e. have the same scale (e.g. both bottom axes.). This is only valid for axes of the same parameter type (e.g. two frequency axes).
- Size of the graph.

4.3.6 Double 2D polar graph horizontally split

For a horizontally split 2D polar graph we want to record the following settings:

- Description of the radial axis of the left graph.
- Description of the angular axis of the left graph.
- Description of the radial axis of the right graph.
- Description of the angular axis of the right graph.
- What axes of the two graphs must be coupled, i.e. have the same scale (e.g. both radial axes). This is only valid for axes of the same parameter type (e.g. two angle axes).
- Size of the graph.

4.3.7 2D cartesian and 2D polar graph horizontally split

For a combined 2D cartesian/polar graph we want to record the following settings:

- Description of the axis on the left of the cartesian graph.
- Description of the axis on the rigth of the cartesian graph (default empty).
- Description of the axis on the bottom of the cartesian graph.
- Description of the radial axis of the polar graph.
- Description of the angular axis of the polar graph.
- Dependency between axes in the same direction (e.g. must ticks on the rigth axis be on the same places as the ticks on the left axis) of the cartesian graph.
- What axes of the two graphs must be coupled, i.e. have the same scale (e.g. the right axis and the radial axis). This is only valid for axes of the same parameter type (e.g. two angle axes).
- Ordering of the two graphs: cartesian left and polar rigth or reversed.
- Size of the graph.

4.3.8 3D cartesian graph

For a 3D cartesian graph we want to record the following settings:

- Description of the x-axis of the graph.
- Description of the y-axis of the graph.
- Description of the z-axis of the graph.
- Size of the graph.
- Should level-colors be used: the color of the graph is dependent of the function-value (height). The levels used are the same as the contour levels.

4.3.9 Contour graph without margin graphs

For a contour graph without margin graphs we want to record the following settings:

- Description of the x-axis of the graph.
- Description of the y-axis of the graph.
- Description of the level-axis of the graph.
- Representation: Filled or lines.
- Size of the graph.

4.3.10 Contour graph with margin graphs

For a contour graph with margin graphs we want to record the following settings:

- The same items as mentioned for a contour graph without margin graphs.
- Description of the the left axis of the top margin graph (default the same as the level axis).
- Description of the the bottom axis of the right margin graph (default the same as the level axis).
- Relative size of margin graphs. (E.g. 0.3 times the size of the contour part.)

4.3.11 Polar contour plot

For a polar contour plot, we want to record the following settings:

- Description of the radial axis of the graph.
- Description of the angular axis of the graph.
- Description of the level-axis of the graph.
- Representation: Filled or lines.
- Size of the graph.

4.3.12 Polar contour plot with "hole"

For a polar contour plot with a "hole", we want to record the following settings:

- Description of the radial axis of the graph.
- Description of the angular axis of the graph.
- Description of the level-axis of the graph.
- Relative size of the hole (fraction of the radius of the total graph).
- Representation: Filled or lines.

- Size of the graph.

4.4 Some random considerations

- A plot with Re/Im axes must be placed somewhere.
- In the positioning of the picture the order of operations (translation, rotation) has to be reconsidered.
- When drawing polylines, an optimization can take place if some consecutive parts have the same (horizontal or vertical) direction.
- When drawing a 3D cartesian graph, the ground plane is drawn if the function value lies below it.
- In contour graphs the boundaries of the file are marked if these are within the plotting area.
- A 3D plot with a circular (polar) base.
- User must be able to choose between use of automatic selection of ranges in selected file or use of last selections made.
- After reading a data-file we build two lists:
  1. List of primary parameters. This list includes:
     - Independent parameters
     - Monotonous increasing/decreasing parameters
  2. Dependent parameters ("function values")

  As we can see, it is possible for a parameter to be in both lists.
- Per parameter-type we keep a central history buffer of the last x chosen values (and possibly some default values). Every axis of a graph keeps its current value local, but references to the central history when choosing a new value. In this way we are able to have two axes of the same parameter-type with different scales.

Chapter 5

Implementation of markers

5.1 Introduction

In the graphics application, the user must be able to place markers in the graphs to indicate points of interest and, if necessary, get the values of those points. At the moment, this functionality isn't present in **plplot**, so this has to be added to it. For the time being it will only be implemented for the X11 device, because it will only be of use for interactive devices. However the software must be able to *display* these markers also on non-interactive devices, which is in general not very difficult.

5.2 Types of markers

The program has to support a number of different marker types. These are:

Single point
: The marker indicates a single point in the graph. Some different shapes must be available to represent the marker.

Vertical line
: The marker indicates a single x-value in the graph.

Horizontal line
: The marker indicates a single y-value in the graph.

Cross (Crosshair)
: The marker indicates a single point or a combination of a single x-value and a single y-value. The marker is visualized by a vertical and a horizontal line which extend to the boundaries of the graph. (This marker type can be used in the contour plot with margin graphs, where the horizontal and vertical lines indicate the position where the graph is intersected to obtain the margin graphs.)

Rectangle
: This marker indicates an area in the graph. (This marker type can be used for example in the gating program to specify the gate.)

Diagonal line

With this marker we are able to simulate a marker in a 3 dimensional graph. This marker will be the only one (among the other 2D types) supported by the device driver, because the driver is 2D oriented and doesn't know anything about 3D, it all has to be done at a higher level.

Other marker types can be constructed by combining these marker types.

5.3 Marker placement modes

The placement of markers can be divided in the following way:

Automatic placement

The application itself decides where a marker has to be placed.

Interactive placement

The user decides where to place a marker. In this case the application may want to be able to adjust the exact placemenent to get the markers exactly on data-points. There a two kinds of interactive placement:

By value

The user specifies a numeric value to identify the position where to place the marker. This placement method can be handled mostly in the same way as automatic placement of markers, because the data enters the graphic routines at a rather high level.

Visual placement

The user specifies the position on the device where to place the marker. If we use the X11 device, this will be done using the mouse pointer (or cursor keys). Using this method, the user must have *real time* feedback, i.e. the user must be able to move the marker (and see it moving) by dragging the mouse pointer.

Because the interactive positioning by using the mouse pointer is the most demanding method with respect to the marker placement routines we will first concentrate on this method.

5.4 Visual placement of markers

To make the program ergonomically justified, the user must be able to move and place markers at any time (as long as it makes sense).

With this type of placement the action stems from the (X11) device driver which gets the events from the X-server. Therefore the device driver must be able to initiate a marker action, because polling of the device driver by the upper layer is intolerable. However the upper layers are needed in the placement process to calculate the exact position of the markers and to administrate the position of the markers (if we want to output the graph to another device, we must be able to place the markers there on the same places).

This calls for the use of callback functions: The upper layers register a function to the device driver, which the device driver shall call whenever it gets some placement events. The callback function in turn determines what to do with that data. But of course there are restrictions in what the callback function may do with the data: the handling of this data should be consistent.

To restrict the freedom op marker placement as litle as possible, there is no direct coupling between the mouse position acquisition and the marker placement within the device driver: the coupling should be done by the callback function. By doing it this way, we have the possibility to place several markers as a result of one position input (this is of use if we have for example a contour plot with margin graphs). Also, we will then have the opportunity to use the mouse input for other purposes.

5.5 Positioning

Besides the positioning with the mouse as described before, positioning must also be possible with keystrokes, especially the cursor keys, to make the program more user friendly. It's obvious, that by using cursor keys relative positions are specified. Although it is possible to convert this to an absolute position within the device driver, it's more flexible not to do so (and let the callback function do it if necessary), because then we have an extra functionality available: we have the possibility to interpretate the relative positioning in another way, e.g. go to the next local maximum.

The device driver should not send all mouse movement events to the callback function, otherwise moving the mouse through the graphics window should generate a lot of calls of the callback function. In general we are only interested in mouse motion events if one or more mouse buttons are activated. We can solve this by using the following convention: the drivers starts calling the callback function (supposed it is enabled) when one or more mouse buttons are pressed and stops if they are all depressed.

If the callback function is able to determine which button(s) is (are) pressed,

it is able to react different upon them (e.g. when placing a rectangular marker we can specify two corners, or we can specify a center point and an extend). To be able to simulate this behaviour if the user uses the keyboard, we generalize this idea:

Instead of the position of the mouse buttons, we pass a function number to the callback function (which normaly correspondents with the button number) and the state of that function:

start This is the first call with this function (mouse button just pressed). When placing a marker: start placing the marker.

continue The same function as in the previous call is still activated (mouse moved when mouse button still pressed). When placing a marker: move the marker.

stop This is the last call (for the moment) with this function (mouse button released). When placing a marker: freeze the marker at this position.

cancel This function has to be canceled (another mouse button is pressed while the first was still down). When placing a marker: remove the marker or keep it on the place it was before the start of this function.

(For all these functions the will have to be keyboard equivalents.)

Now we can have only one button active, which limits the number of options we have while operating the mouse. This should however not be a problem, since overloading a few buttons with functionality isn't a good idea either.

It's imaginable that there are occasions in which we also want the callback function to be called when the mouse pointer moves, but there are no buttons pressed. For this case an option should be available in the function which enables the calling of the callback function.

5.6 Marker placement

A marker is placed on the graph by calling a device driver function to which we specify the marker type and the position (which includes the size). Because we want the possibility to remove or move (which implies remove) a marker, we must be able to restore the original contents of the graph at the position of the marker. The most simple solution would be to redraw the complete graph, but this isn't usable in this case because it would be to slow, especialy if we want to move the marker real-time. Several other solutions come to mind:

- While drawing the graph to the screen, we also draw it to an of-screen pixmap (on which we don't draw any markers). If we have to remove a marker, we copy this pixmap to the screen (and redraw all other markers). In general, this method will still be relative slow.

- As in the previous solution we keep an of-screen pixmap, but we only copy the regions which were hidden by the marker. This method doesn't free us from the redrawing of all other markers, because we might have overwritten some of them.
- We draw the markers using the exclusive-or function: all pixels which are part of the marker are exclusive-ored with the marker pixel-value. In this way, we can remove the marker by simply redrawing it (also with exclusive-or). This method works only save if we don't draw anything else in the graph without exclusive-or after we have drawn something with exclusive-or. If we use color, we can't say much about the resulting color, so it may be possible that the markers are not very visible.
- We combine the previous methods:
  - If we are moving a marker, we use exclusive-or. In this case the problem of visibility isn't as big as with a fixed marker, because the user can see something moving. An advantage of using this method is, that it's fast.
  - If we place a marker for a longer period, we draw it solid (and use a copy from a pixmap to remove it) because the visibility is better and we can give it a predetermined color.
  - If some part of the graph has to be redrawn, we copy that part of the of-screen pixmap to the screen and redraw all solid markers (they are not on the pixmap).
    Redrawing of the exclusive-or marker would be a litle tricky if we just restore a part of the graph, because we would erase the marker in the area which wasn't restored (redrawing an exclusive-or marker erases it), unless we draw the marker only in the region of the graph we restored. We won't have these problems however, if we stick to the use of the exclusive-or markers described here: We will always have at most one exclusive-or marker active and we are using it (we hold a button down to place it), so in normal condition we are not able to mess things up.

In case the reader hasn't noticed yet: the last method is probably the best one.

This leads to the following marker placement strategy:

- If we want to place a fixed marker, we draw it solid.
- If we want to interactively place a marker, we use the exclusive-or mode, and at the moment we want to fix it at the current position (we depress the mouse button) we draw the marker solid.

- If we want to reposition a fixed marker (which is drawn solid) interactive we use an exclusive-or marker to mark the new (moving) position while the old (solid) marker is still in its place. If the user fixes the marker in a new position, we remove the old solid marker and draw it in the new position. If the user choses to cancel the replacement, we only delete the exclusive-or marker and keep the old solid marker in its place.
- The user must have the ability to point at a marker he wants to repositon (or delete). In general we can not demand from the user an exact positioning of the mouse pointer at the marker. Therefore we will have to find a marker that is closest to the specified position (or within a specified range).

5.7 Marker removal

It's no problem to remove a temporary marker: it's just drawn again. To remove a permanent marker is a little more problematic, because we have to restore the original contents in the part of the graph which was obscured by the marker.

In X, the easiest way to do this, is to clear that part of the graph. The X-server will send (as a consequence of the clear) an expose event for that part of the window. This expose event is handled by the applications expose callback function, which must be present in every X-application to handle this kind of events resulting from other windows disapearing, etc.

To use this method effectively, it would be wise not de clear to much, because this would result in flicker of parts of the display. Therefore we only clear the exact lines of the marker, if possible. This results in clearing more than one rectangle for some kinds of markers. We can combine the resulting expose events provided that the rectangles which are exposed do not overlap.

To make sure the exposed rectangles resulting from a marker removal do not overlap, we will calculate these rectangles very precisely. Figure 5.1 shows the rectangles to be cleared for the various markers (only the interesting markers are displayed). In out calculations we use $hw = \frac{linewidth}{2}$ instead of the linewidth itself to ensure we don't miss part of the marker due to rounding errors. The resulting rectangles are displayed as 4 coordinates representing the start x and y values and the widht and height of the rectangles.

Vertical marker

This marker can be covered by one rectangle:

$$(x - hw), (ystart), (2 \times hw + 1), (yend - ystart)$$

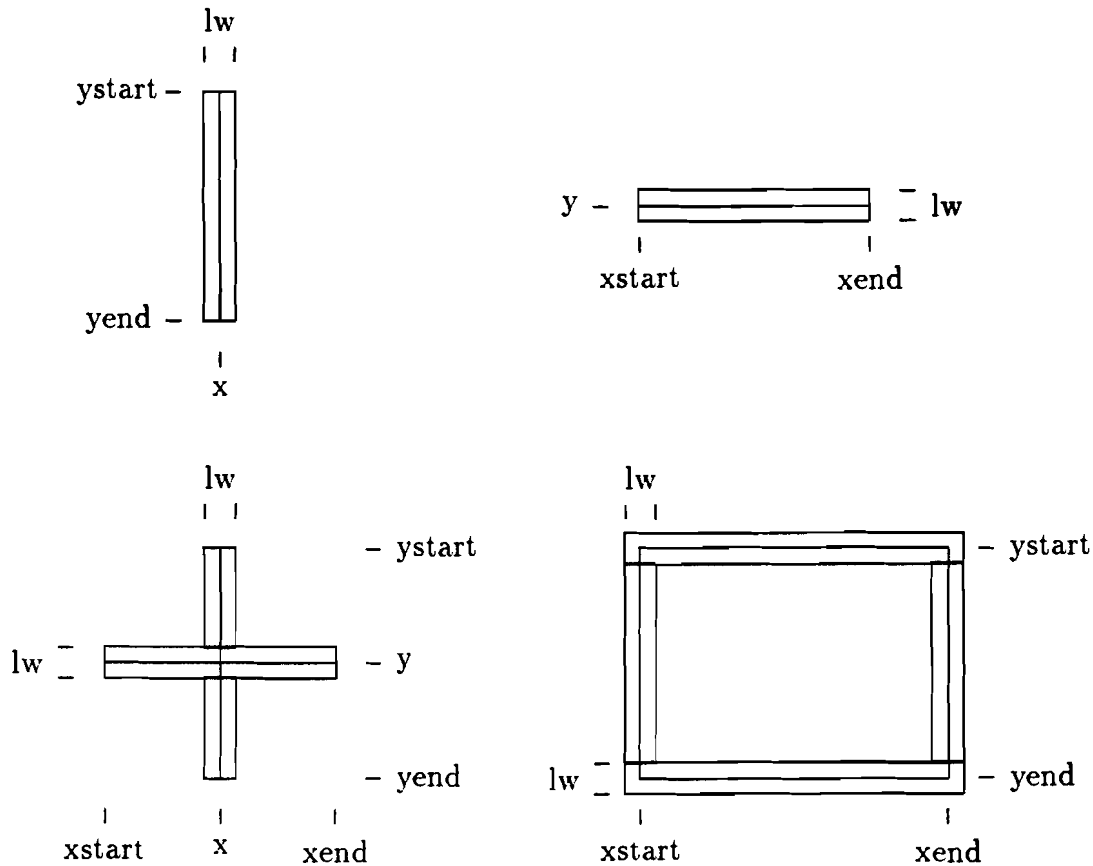

Figure 5.1: Rectangles to remove a marker

Horizontal marker

This marker can also be covered by on rectangle:

$$(xstart), (y - hw), (xend - xstart), (2 \times hw + 1)$$

Cross marker

For this marker we need 3 rectangles to cover it completely:

$$(xstart), (y - hw), (xend - xstart), (2 \times hw + 1)$$

$$(x - hw), (ystart), (2 \times hw + 1), (y - hw - ystart)$$

$$(x - hw), (y + hw + 1), (2 \times hw + 1), (yend - y - hw - 1)$$

Rectangle marker

For this marker we need not less than 4 rectangles to cover it:

$$(xstart - hw), (ystart - hw), (xend - xstart + 2 \times hw + 1), (2 \times hw + 1)$$

$$(xstart - hw), (yend - hw), (xend - xstart + 2 \times hw + 1), (2 \times hw + 1)$$

$$(xstart - hw), (ystart + hw + 1), (2 \times hw + 1), (yend - ystart - 2 \times hw - 1)$$

$$(xend - hw), (ystart + hw + 1), (2 \times hw + 1), (yend - ystart - 2 \times hw - 1)$$

5.8 Implementation

The marker placement and positioning code will be implemented in the device driver and be accessible to higher levels via two functions:

`X11locator()`
: Enables or disables the calling of the callback function on positioning events from the X interface.

`X11setmarker()`
: Places (or removes) a marker on the drawing area.

The normal way of interactive placement of markers is to enable (with `X11locator()`) the sending of positioning events to the callback function, which calculates from the display position the (rounded) data position and places on that position a marker (with `X11setmarker()`).

5.8.1 X11locator()

The prototype of `X11locator()` is as follows:

```
typedef void (*callback)(int coord_mode,
                         int function, int status,
                         int x, int y)      locatorCB;


void
X11locator(int mode, locatorCB callback);
```

`mode`
: Should be one of the following:
  `LOC_DISABLE`
  : Disables locator placement.
  `LOC_ACTIVE_EVENTS`
  : Locator events are generated if mouse buttons are pressed, or when operating keyboard.
  `LOC_PASSIVE_EVENTS`
  : Locator events are generated if mouse pointer is within the drawing area.

`callback`
: The function that should be called whenever an event occurs.

`coord_mode`
: Will be one of the following:
  `ABSOLUTE`
  : The x and y coordinates are absolute.
  `RELATIVE`
  : The x and y coordinates are relative to the previous position.

`function`
: The number of the activated button.

`status`
: Will be one of the following:

  `START`
  : This event starts a locator placement sequence.

  `STOP`
  : This event ends a locator placement sequence.

  `CONTINUE`
  : This event continues an already started locator placement sequence.

  `CANCEL`
  : This event cancels a locator placement sequence.

`x`
: The x coordinate of the mouse pointer (can be absolute or relative).

`y`
: The y coordinate of the mouse pointer (can be absolute or relative).

5.8.2 X11setmarker()

The prototype of `X11setmarker()` is as follows:

```
void
X11setmarker(int mkr, int mode, int mkr_type, int style,
             char *detail, int x, int y,
             int xstart, int ystart, int xend, int yend);
```

`mkr`
: Specifies the marker which should be placed/removed.

`mode`
: Specifies what should be done with this marker:

  `MKR_MODE_FIX`
  : Place a fixed marker (draw the marker solid) on the specified position. If this marker is already somewhere else, the old marker should be removed.

  `MKR_MODE_TMP`
  : Place a temporary marker (draw the marker with exclusive-or) on the specified position. This doesn't erase the fixed marker (if present).

  `MKR_MODE_ERASE`
  : Erase the specified marker, both the fixed instance and the temporary instance (if present).

`MKR_MODE_ERASE_TMP`
: Erase the specified temporary marker.

`mkr_type`
: Specified how the marker should look like. The following type are defined:

`MKR_CURRENT`
: Used on all calls except the first one for a specific marker: The marker type is already specified in a previous call.

`MKR_POINT`
: A point marker is placed on position `(x,y)`. The shape of the marker is specified as a string in the `detail` parameter, and is respresented with hershey fonts (see 6).

`MKR_VERTICAL`
: A vertical line between the points `(x,ystart)` and `(x,yend)`.

`MKR_HORIZONTAL`
: A horizontal line between the points `(xstart,y)` and `(xend,y)`.

`MKR_CROSS`
: A crosshair is placed, consisting of the lines
`(x,ystart), (x,yend)` and `(xstart,y), (xend,y)`.

`MKR_RECTANGLE`
: A rectangle is placed with the points
`(xstart,ystart), (xstart,yend),`
`(xend,yend), (xend,ystart).`

`MKR_DIAGONAL`
: A diagonal line is drawn between `(xstart,ystart)` and `(xend,yend)`.

`style`
: Specifies the style (see chapter 7) in which the marker should be drawn. This parameter is only used on the first call for a specific marker of this function.

`detail`
: Specifies the glyph to be used for a point marker. This parameter is only used on the first call too.

`coordinates`
: These specify the position where to place the marker. The use is already described.

Chapter 6

Font Support

6.1 Introduction

In order to be able to anotate graphs, it must be possible to draw text in the graphs. Therefore we need some kind of font support.

The **plplot** library already supports text in a limited way. The font support is positioned in the device independent part of the device driver, as can be seen in figure 6.1. This standard font support has some disadvantages however:

- The font data is only available in a binary file, without any documentation of the format, other than the sources which read these data.
- The font files always have to be present seperate of the application and is loaded on demand by the application.
- The number of fonts/glyphs present is limited.

To overcome these limitations, I decided to reimplement the font support, which has some other advantages too, which will be described later.

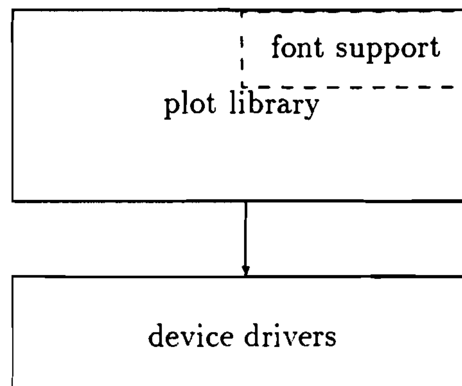

Figure 6.1: Position of font support in the **plplot** library

6.2 The Hershey Font Set

The hershey font set is a set of (scalable) stroke fonts which were digitized by Dr. A. V. Hershey while working for the U.S. Government National Bureau of Standards (NBS). The set consists of more than 2000 glyph (symbol) descriptions in vector format and can be grouped as almost 20 *occidental* (english, greek, cyrillic) fonts (see table 6.1), 3 (or more) *oriental* (Kanji, Hiragana and Katakana) fonts, and a few hundred miscellaneous symbols (mathematical, musical, cartographic, etc.), see [Her].

6.3 The advantages of the Hershey font set

The Hershey fonts set has several advantages compared to other font sets:

- It's public available.
- "Sources" available, so it's possible to change and/or add glyphs.
- The format of the font sources is described clearly.
- Glyphs are fully scalable an transformable.
- The graphical quality of the glyphs is reasonable.
- They can be represented on every graphic device.

Because of the above mentioned advantages I choose to use this font set. However there is one disadvantage: There a no support routines available, so the have to be written.

Fonts with ASCII coding	
Hershey-Script-Simplex	Hershey-Italic-Complex-Small
Hershey-Script-Complex	Hershey-Italic-Complex
Hershey-Roman-Plain	Hershey-Italic-Triplex
Hershey-Roman-Simplex	Hershey-Gothic-German
Hershey-Roman-Duplex	Hershey-Gothic-English
Hershey-Roman-Complex-Small	Hershey-Gothic-Italian
Hershey-Roman-Complex	
Hershey-Roman-Triplex	
Fonts with non-ASCII coding	
Hershey-Greek-Plain	Hershey-Cyrillic-Complex
Hershey-Greek-Simplex	Hershey-Special-Math-1
Hershey-Greek-Complex-Small	Hershey-Special-Math-2
Hershey-Greek-Complex	

Table 6.1: Occidental fonts

6.4 New position of the font support routines

As mentioned before, the original font support of **plplot** is positioned in the device independent part of the library (figure 6.1). The main reason for this is that the font support code is device independent and all device independent code should be placed in the device independent part of the library. There should be strong reasons to place it somewhere else.

I think there are such strong reasons to place it at device driver level. These reasons are:

- Device specific optimizations are possible:
  - Some devices support download fonts: after a font is downloaded to the device, a specific character can be referenced by its index.
  - Some devices can scale and/or transform characters.
  - On some devices, some fonts can be replaced by the device's fonts.
- Glyphs can be used for markers: see the description of markers in chapter 5

To prevent duplication of great parts of the font support routines, these routines are available to all device drivers (see figure 6.2) and are not part of one device driver.

6.5 Interface routines

The hershey library has four interface routines by which a device driver can communicate with the library. These are:

`hershey_get_font_name()`
: Returns the name of the specified hershey font.

`hershey_parse_string()`
: Converts an ASCII string (with formatting commands) to an ab-

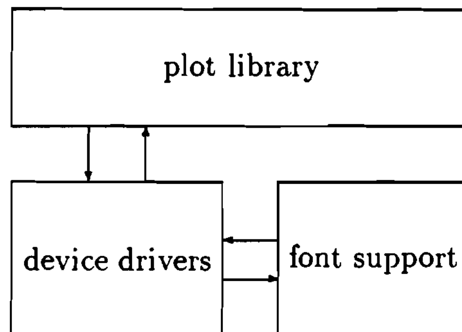

Figure 6.2: New position of font support routines

stract data structure of type `HersheyString`.

`hershey_draw_string()`
: Draws the contents of a `HersheyString` abstract data structure on the device.

`hershey_free_string()`
: Destroys a `HersheyString` abstract data structure.

6.5.1 `hershey_get_font_name()`

```
char *
hershey_get_font_name( HersheyFont font );
```

`font`
: The font number of which the name is to be returned.

Return value
: A pointer to a read only string in which the name is placed, or a NULL-pointer if `font` is out of range.

This function converts a font id to the name of the font. The name is one of the strings of table 6.1.

6.5.2 `hershey_parse_string()`

```
HersheyString
hershey_parse_string( char *str, HersheyFont default_font,
                      int *xpos, int *ypos,
                      int *matrix, int scale );
```

`str`
: The character string to be converted. The string is only used during the function call, so it may be changed or freed after the call.

`default_font`
: The font to use if no other font is specified in the string itself.

`xpos, ypos`
: Pointers to the x- and y-position where the string should be placed. On return the end position of the string is written in them.

`matrix`
: An array of four integers specifying the default transformation matrix for this string. All points (x_g, y_g) of all glyphs are transformed as follows:

$$\begin{pmatrix} \frac{\text{matrix[0]}}{100} & \frac{\text{matrix[1]}}{100} \\ \frac{\text{matrix[2]}}{100} & \frac{\text{matrix[3]}}{100} \end{pmatrix} \begin{pmatrix} x_g \\ y_g \end{pmatrix}$$

You can see the values should be multiplied by 100.

`scale`
: The default value of an extra scale factor applied to all points times 100. So if no extra scaling should take place, this value should be 100.

Return value
: If non 0 this value should be used on every call to one of the hershey library functions to specify this string. This value should never be discarded, because is represents some dynamically allocated memory which should first be freed with the appropiate function (`hershey_free_string()`). If the value is 0, some error occured in this function and no space is allocated.

This function "compiles" a string with formatting commands into some internal representation.

Formatting commands

At the moment, no formatting commands are defined yet. But they should be defined for at least the following functions:

- Font selection
- Change transformation matrix
- Change scale factor
- Specify a glyph
- Sub- and superscript

6.5.3 `hershey_draw_string()`

```
int
hershey_draw_string( HersheyString hershey_str,
                     int x, int y,
                     DevPolyLine func, void *funcdata );
```

`hershey_str`
: Identifies the string to be plotted to the output device. This value can be obtained by a call to `hershey_parse_string`.

`x, y`
: An extra offset in device coordinates to be applied to the complete string.

`func`
: A pointer to a function to be used for drawing polylines. This function is used to output the string. The prototype of the function is:

```
void
(*DevPolyLine)(void *funcdata, Point *pt, int npoints);
```

`funcdata`
: Pointer to a function defined data structure.

`pt`
: Array of points which for a polyline and should be plotted.

`npoints`
: The size of the `pt` array.

Return value
: None.

`funcdata`
: Pointer to a `func` specific data structure. It is passed as first parameter to `func`.

Return value
: One of the following values:
: `R_ERROR` An error is detected.
: `R_SUCCESS` No error detected, string plotted successfully.

Actually draws a string.

6.5.4 hershey_free_string()

```
void
hershey_free_string( HersheyString hershey_str );
```

`hershey_str`
: A return value of a previous `hershey_parse_string()` function call. After a call to this function this value should not be used any longer.

Return value
: None.

This function cleans up internal data structures of the hershey font library which are in use for the string identified by the `hershey_str` value.

6.6 The internal workings

For a precise description, the sources are the best documentation. To make these sources easier understandable, a description of various data structures is presented here.

6.6.1 The hershey character and glyph data structures

The task of the hershey font library is to convert strings (with formatting commands) to commands to the device to plot the corresponding glyphs. To do this, the library contains the following data:

hershey_data
An array which contains the actual description of all glyphs in (x, y) coordinate pairs. The exact structure will be described later.

hershey_index
This is an array which maps the glyph number to an index in the **hershey_data** array where the description of the glyph starts.

hershey_map
This is a two-dimensional array which is a per font map of a character number to glyph number.

hershey_base_line
The y offset of the base line for each font.

hershey_cap_line
The y offset of the top of capital letters for each font.

hershey_font_name
The name of each font.

The connection between these data-structures is shown in figure 6.3 (I omitted **hershey_cap_line** which is equivalent to **hershey_base_line**).

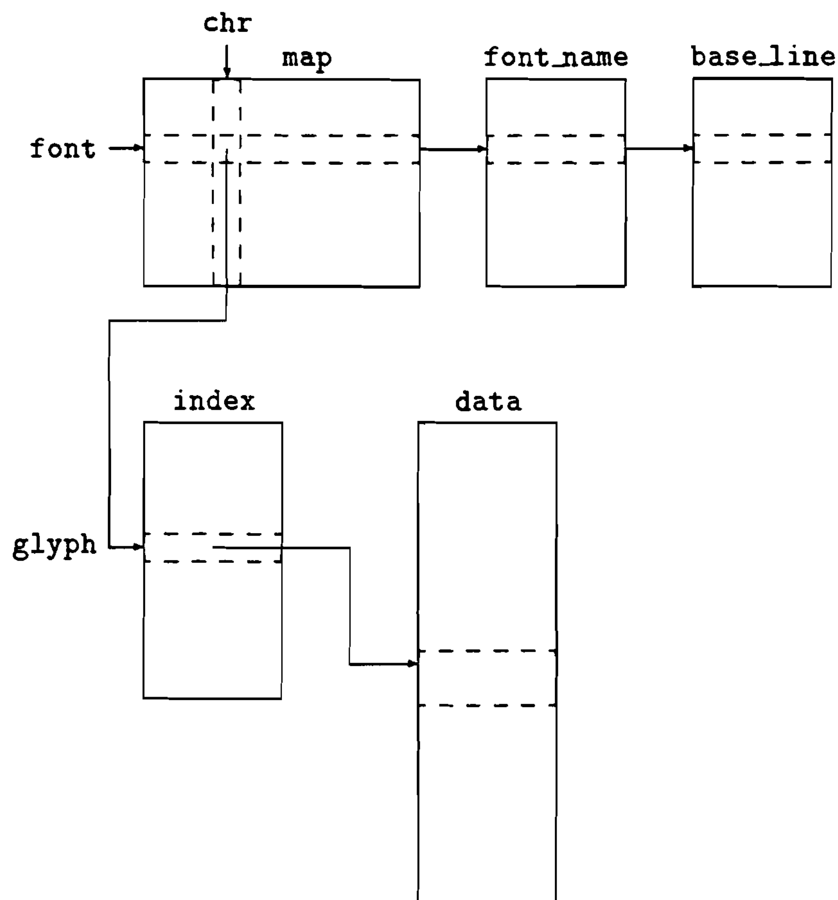

Figure 6.3: Relationship of hershey data structures

Each glyph consists of a number of polylines, each polyline consists of a number of connected points. The number of points per polyline and the number of polylines per glyph are variable. Besides the polylines, each glyph also defines a negative and positive x-extend, and per font a base-line and a cap-line are defined. These parameters are shown in figure 6.4.

The format in which I obtained the hershey fonts, was not very compact and efficient. Therefore I designed a new format to contain the above mentioned data.

Polylines of variable length can be represented by the number of points forming the polyline, followed by the points (x and y coordinates) themselves. To represent a number of polylines, we can place them after each other if we are able to mark the end of the sequence. This can be done by storing the number of polylines, but another solution is to finish the sequence with a 0. However, this could be seen as a new polyline, but this would represent a polyline consisting of 0 points, which would be meaningless.

So the structure per glyph becomes:

- A negative x-extend. This determines the x-offset of the character. Together with the following element, this also determines the width of the character (extra whitespace between characters is taken into account).
- A positive x-extend.
- Repeatedly the following sequence (the end is marked by the 0)
  - The number of (x, y) pairs that follow and should be connected to each other.
  - The (x, y) pairs.
- 0 to mark the end of the glyph.

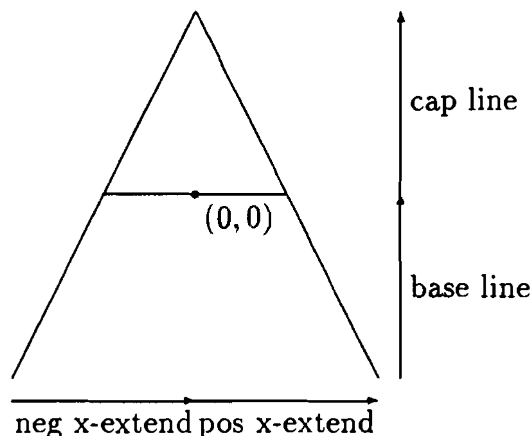

Figure 6.4: A hershey letter

The descriptions of all glyphs are place in on array, **hershey_data**. To find the data for a specific glyph, another array, **hershey_index**, has, for each glyph, a start point. For practical reasons, the index array points to the first polyline (starting with the number of points). Doing this, index 0 is not a valid index and thus can be used for special purposes.

6.6.2 The HersheyString data structure

In order to be able to be more flexible in handling text, all text is first converted to a data structere of type `HersheyString` which is a double linked list of `HersheyCharInfo` structures. This structure contains the following parameters:

`chr`
: The character of this element, if available.

`font`
: The font in which the character should be represented, if available.

`glyph`
: The glyph of this element. It is calculated from `chr` and `font`, or directly specified.

`matrix`
: A transformation matrix, consisting of four intergers.

`scale`
: An extra scale factor.

`xoffset, yoffset`
: An offset in hershey coordinates[1] which should be applied to all coordinates of the glyph before the transformation and scaling.

`position`
: A vector in device coordinates, specifying the offset of the start of this glyph to the start of the complete string.

`height`
: A vector in device coordinates specifying the height[2] of this glyph. This vector points from baseline to the top of the glyph. The direction is the transformed perpendicular line of the baseline.

`width`
: A vector in device coordinates specifying the width of this glyph. This vector points from the left of the glyph to the right, in the direction of the baseline.

`depth`
: A vector in device coordinates specifying the depth[3] of this glyph.

1. The coordinate system in which all glyphs are represented in the **hershey_data** array.
2. The distance between baseline and top of character.
3. The distance between bottom of character and baseline.

The vector points from the bottom of the glyph to the baseline, and the direction is the transformed perpendicular line of the baseline.

`previous, next`
Pointer to create a double linked list.

As can be seen, the size of the glyph is specified as a set of vectors: height, width and depth. Because the glyphs can be rotated, only a (set of) scalar(s) is not enough to specify the extends of the glyph. If a scalar size is required, the length of the vector can be calculated.

Chapter 7

Styles

7.1 Introduction

In the original **plplot** library the characteristics of a line are specified by three parameters:

- color
- width
- dashes

These parameters are passed to the device drivers. However, not all devices have the same amount of colors, or an unlimited spectrum of widths. This makes it difficult to make fully device independent applications. A solution to this problem could be to let the device driver make some kind of automatic mapping if a color or line width is not available. But I suppose everyone knows Murphy, and this will always be the wrong mapping in the users point of view.

7.2 The concept of styles

To overcome the device dependencies I changed the concept and introduce *styles*.

- To the device independent part of the library, a *style* is an abstract data type specifying some set of characteristics for each object drawn. These characteristics can not be specified seperate, only as a set.
- To the device driver, a *style* is a set of device dependent characteristics that can be used when drawing an object.

So in each device driver, the styles are mapped to object attributes (e.g. color, width). Because this mapping is done in the device driver, it can be different for each device driver. So a line which is magenta on one device, can be green on another device.

By introducing styles, we eliminated all device dependencies in the upper layers of the device drivers. However, now we don't have any control over the characteristics of an object, which is undesirable. Therefore we add two device driver funtions to set and get the device characteristics of a style. This function changes only the style characteristics of one device.

Although we introduce a new device dependency in the library (setting a style), this overcomes the device dependency problems, because the newly introduced functions is normaly only used during startup, or on explicit request of the user.

7.3 Contents of a style

A style can contain many attributes of an object. Currently the following are included:

color
: The color of the object.

linewidth
: The width of the lines in a device specific unit.

line-style
: Contineous, dashed or dottet lines.

font
: Default font for text

font-size
: Default size of text (scale parameter used for hershey fonts).

7.4 Interface functions

To support styles, three new functions were added to the device driver interface. These are:

`dev_selectstyle()`
: Selects a style to be used in successive drawing functions.

`dev_setstyle()`
: Change attributes of a style of this device.

`dev_getstyle()`
: Get attributes of a style of this device.

7.4.1 The `StyleValues` structure

The functions `dev_setstyle()` and `dev_getstyle()` both use a structure to pass values.

```
typedef struct {
    char        *color;
    int         line_width;
    char        *dash_type;
    char        *font_name;
    int         font_id;
    int         *matrix;
    int         scale;
} StyleValues;
```

`color`
: A pointer to a string containing the colorname.

`line_width`
: The width of the lines, in a device driver specified unit, often device coordinates.

`dash_type`
: A pointer to a string specifying the dashes (contineous, dashed, dotted, dash-dotted lines). The interpretation is device specific: some devices only allow a choice from a limited set of dash-types, other set allow one to specify the lengths of the various parts.

`font_name`
: The name of the selected font. Is only used to retrieve the value, to set a font, only the next element can be used.

`font_id`
: A sequence number specifying the font.

`matrix`
: An array of four intergers forming a 2×2 transformation matrix for all text operations. The unit value is 100.

`scale`
: The scale factor by which all text related coordinates are multiplied. The unit is 100.

For a more precise description of the text related elements, see chapter 6 about font support.

7.4.2 dev_selectstyle()

```
void
dev_selectstyle( int style );
```

`style`
: The number of the style to be selected for use in all successive drawing function calls.

Return value

None.

7.4.3 `dev_setstyle()`

```
int
dev_setstyle( int style, StyleValues *stylevalues, int mask );
```

`style`

The number of the style to be changed.

`stylevalues`

A pointer to a structure containing the attibutes to be changed.

`mask`

A mask identifying what attributes are valid in the `stylevalues` parameters and should be changed.

Return value

A mask identifying what attibutes were changed successfully. If this value is equal to the mask parameter, all attributes were changed successfully, if the value is 0, no attributes were changed at all.

7.4.4 `dev_getstyle()`

```
int
dev_getstyle( int style, StyleValues *stylevalues, int mask );
```

`style`

The style number of which we want to get some attibute values.

`stylevalues`

A pointer to a structure in which the requested attribute will be placed.

`mask`

A mask identifying what attributes are requested.

Return value

A mask identifying what attributes in the `stylevalues` structure are valid. If the value is equal to the mask parameter, all attributes were retrieved successfully, if the value is 0, no attributes were retrieved at all.

The contents of the data structures to which is pointed to by elements of the data structure should not be changed or freed.

Chapter 8

The device drivers

8.1 Introduction

The device drivers are a very important part of the plot-library. They are responsible for an efficient use of the devices capabilities and should overcome any deficiencies of the devices. The interface between the device driver and the higher levels of the library (the device driver programming interface) should ensure device independence.

8.2 The original device driver programming interface

The standard device driver application programming interface of **plplot** has the following functions (exerpts from **plplot** sources):

`dev_setup`
: This routine to sets x and y resolution (dots/mm) and x and y page widths. Some device drivers may choose to ignore any or all of these. A call to this routine is optional! If a particular driver requires any of these parameters and they are not set by a call to **`dev_setup()`** then they should be prompted for in `dev_init()`. The user may call this routine only once and it is called before any output generating function calls.

`dev_orient`
: Set plot orientation: landscape or portrait.

`dev_select`
: Set graphics storage file pointer. Directs the device driver to redirect all data to a file rather then the real device. This routine is also optional. This routine must be called before any output generating functions.

dev_init
: Initialize device. This routine may also prompt the user for certain device parameters or open a graphics file (see note). Called only once to set things up.

dev_line
: Draws a line between two points.

dev_clear
: Clears screen or ejects page or closes file (see note).

dev_page
: Set up for plotting on a new page. May also open a new a new graphics file (see note).

dev_eop
: End current page (flush buffers).

dev_tidy
: Tidy up. May close graphics file (see note).

dev_color
: Change pen color.

dev_text
: Switch device to text mode.

dev_graph
: Switch device to graphics mode.

dev_width
: Set graphics pen width.

dev_cwin
: Switch to command window.

dev_gwin
: Switch to graphics window.

NOTE

Some devices allow multi-page plots to be stored in a single graphics file, in which case the graphics file should be opened in the `dev_init()` routine and closed in `dev_tidy()`. If multi-page plots need to be stored in different files then `dev_page()` should open the file and `dev_clear()` should close it. Do **not** open files in both `dev_init()` and `dev_page()` or close files in both `dev_clear()` and `dev_tidy()`. The purpose of `dev_text()` is to allow the user to place device-dependent characters on the graph. The user responsible for positioning these characters. Its use is discouraged. `dev_cwin()` and `dev_gwin()` are provided to allow the user to switch between a command mode and a graphics mode. In command mode i/o to standard input/output can be accomplished.

8.2.1 Removed device driver interface functions

Some of the functions mentioned above, will not be supported by new or modified device drivers. These are:

dev_color
dev_width
: These functions are superseded by the style funtions, as described in chapter 7.

dev_text
dev_graph
: These functions encourage the use of device dependent applications, the graph mode will be the only mode.

dev_cwin
dev_gwin
: This kind of switching should be done at another level (i.e. not by the plot-library).

8.3 Extending the use of device capabilities

As stated before in section 3.6 the device's capabillities are not fully explored: The only drawing function of the device driver interface is the single line drawing function. Of course this is the basic one, and it can be enough. But the performance can be increased if we make use of the capabilities most devices have. We achieve this by extending the device driver interface to include some more powerfull functions. It is not necessary for all devices to have these capabilities, because it can also be translated to more primitive capabilities by the device driver.

To increase performance, the functions added are:

dev_polyline
: In addition to `dev_line` this function also draws lines. It can however draw more lines in one call. This function is added for more than one reason:

- Most lines to be plotted are joined with other lines.
- Most devices support such a function.
- It can easily be emulated by the device driver if it isn't supported by the device.
- On some devices the result of one polyline is better than a series of single lines (because of rounding on vertices).
- Less data conversion has to take place.

`dev_rectangle`
: This function draws a rectangle. This function is added for about the same reasons as `dev_polyline` is added.

`dev_arc`
: Draws (part of) a circle (or ellips). Again, such a function is supported by many devices. The method to draw an arc whithout this function is to draw many small line-segments (if a device doesn't support arcs, the device driver will probably have to do this). Because **plplot** must now also support polar plots, such a function will be needed often.

8.4 Overview of added device driver interface functions

To replace some removed functions, and to add extra functionality, the following functions are new:

`dev_polyline`
`dev_rectangle`
`dev_arc`
: See previous section.

`dev_drawtext`
: This draws a text string on the device. Described in chapter 6.

`dev_textextend`
: Calculate the dimensions of the specified string. Can be used to determine the best position for a string (or to center a string). Described in chapter 6.

`dev_locator`
: Enable position input from the device, and specify the callback function which is to be called whenever such input takes place. Described in chapter 5.

`dev_setmarker`
: Functions which handles all marker-related functionality, i.e. placement, movement and removal of markers. Described in chapter 5.

`dev_setstyle`
: This function sets (changes) the properties of the specified style. Described in chapter 7.

`dev_getstyle`
: Retrieve the properties of a style. Described in chapter 7.

`dev_selectstyle`
: Select the default style. Described in chapter 7.

Chapter 9

Description of the X11 device driver

9.1 Introduction

The X11 device driver is the most extended device driver of all, because it's the only interactive device driver and because of the special nature of the X11 device: anything can happen anytime. So we must always be prepared to redraw parts of the graph, obey resize request, accept input, etc.

Because programming and understandig X11 isn't very easy, it is very difficult to describe the internal workings of this device driver to someone who has no experience in X11 programming. Therefore this chapter is possibly a little too difficult for the average reader and I'll keep this chapter short. For a complete description, the best description is the **documented** source.

The task of the X11 device driver is not only to convert the device driver interface functions to and from X11 commands, but also to handle various X11 requests (e.g. refresh).

9.2 Refresh and resize

X11 is a windowing system, in which a window can (partly) be obscured by another window. It's the programs own responsibility to update the window whenever requested. There are X-servers that support backing store. If enabled, the server updates the exposed parts of the window itself. The application must however be able to do it itselve, because not all servers support it, and if the server has not enough resources available, it can decide not to do it anymore on any moment.

An often used method to update exposed regions of the window is to redraw the window (parially or totally) from the original data. In our application, this can be very time consuming and therefore unacceptable. We can overcome this problem by maintaining a backing store ourselves: a pixmap with

the same size as the window is created and all objects drawn in the window are also drawn in the pixmap. If an expose event is received, the exposed region is copied from the pixmap to the window, resulting in a fast update.

A window can also be resized at any time. There are two possible responses to such a resize event:

- Redraw the window with the appropriate scaling.
- Don't do anything, but wait till the user explicitly tells to resize and redraw.

I choose for the last solution, because a user wants often make some other modification before replotting the graph, and the redraw can take some time. To make a resize a little more user friendly, some precautions are taken in the user interface, see chapter 10.1.

If a resize occurs, the pixmap should be resized too, which is impossible. Therefore we will have to destroy the pixmap and create a new one. Consequently we lose all data contained in the pixmap, and have to redraw the complete window.

9.3 Markers and device input

The implementation of the markers and device input is already described in chapter 5, so I will limit myself to mentioning some items.

9.3.1 Device input

There are two forms of device input: by mouse and by keyboard. Both are implemented in another way:

Input via mouse

Input via a mouse is implemented with an event handler. To be able differentiate between a start of marker placement and a cancel, a little state machine is used.

Input via keyboard

Input via the keyboard is implemented with X11 action routines. Because the only usable event is a keypress event, we are forced to use a state machine.

Absolute positioning is not very intuitive when using a keyboard, therefore the keyboard input gives always a relative position, which has to be transformed to an absolute position in a higher level.

Chapter 10

User Interface

10.1 Introduction

The user interface is the top part of the application (see figure 3.1). It's task is twofold:

- Set up the X11 environment.
- Supply menu's to interact with the user.

At the moment only the first part is implemented.

10.1.1 X11 environment setup

The setup consists of the standard start of an OSF/Motif application followed by the creation of a DrawinArea widget within a ScrolledWindow widget. This last widget makes it possible to view the complete DrawingArea (in which the graphs are plotted) even if the the window is made smaller, by providing scrollbars.

Also a functions is provided to force a resize of the window. In fact it resizes the DrawingArea widget to the current size of the ScrolledWindow widget. After it has done this, a **plplot** library function is called to force the device driver to do a resize too.

Chapter 11

Conclusions

11.1 X11 and OSF/Motif programming

Programming X11 and OSF/Motif applications differs in many respects from programming some other non-X11 application, because of it's event driven structure. This requires quite a while to get used to, but once you get used to it, it's a nicer kind of programming, because you are almost forced to make the program modular.

Besides from the other program structure, programming X11 and OSF/Motif requires a lot of experience and programming practice. This is caused by several reasons. First, the libraries consist of a couple of hundred functions, which must be known by the programmer at least in such a way that he knows there is a function that does something like that. Second, there are often more posible solution to a certain problem. Most of these solutions turn out to be very complex or inefficient. A programmer should avoid these solutions.

11.2 Status of the project

The project turned out to be to big to complete in the available time. This was partly caused by a slow start and partly by the complexity of X11 programming.

Things that have to be done are:

- database
- some minor extensions to the **plplot**-library
- extension of the user interface

The estimated time to complete this is a couple of months, provided it's done by a programmer with experience in programming in unix, C, X11 and OSF/Motif.

Bibliography

[Der91] Derks, P.G.M.J. A New User Interface for the ARCS Antenna- and RCS-Measurement System. Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering, october 1991.

[Her] A contribution to computer typesetting techniques: Tables of Coordinates for Hershey's Repertory of Occidental Type Fonts and Graphic Symbols. NBS Special Publication 424.

[OSF89a] *OSF/Motif Programmer's Guide: Toolkit*, Revision 1.0, 1989. Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142.

[OSF89b] *OSF/Motif Programmer's Reference Manual*, Revision 1.0, 1989. Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142.

[OSF89c] *OSF/Motif Style Guide*, Revision 1.0, 1989. Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142.

[OSF90] *OSF/Motif Release Notes/Porting Guide*, Revision 1.0.A, 1990. Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142.

[Ric90] Richardson, Tony. *The PLPLOT Plotting Library Programmer's Reference Manual, Version 3.0*, November 1990.

[Xma88a] *X Manual Set. Volume 2: X Library Interface*, first edition, 1988. Addison-Wesley Publishing Company.

[Xma88b] *X Manual Set. Volume 3: X Intrinsics & Athena Widgeds*, first edition, 1988. Addison-Wesley Publishing Company.

[You90] Young, Douglas A. *The X window system: programming and applications with Xt / OSF/Motif edition.* Prentice Hall, 1990.

Appendix A

Contour Literature

A.1 Introduction

The contour algorithm implemented in the **plplot** library is not the most efficient one. Therefore the idea was to implement a new algorithm. Unfortunately, there wasn't any time left to do this, but I already had done some literature research, which I will present here.

A.2 Contour literature list

[Aki70] Akima, Hiroshi. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of the ACM*, 17(4):589–602, 1970.

[Aki74a] Akima, Hiroshi. A method of bivariate interpolation and smooth surface fitting based on local procedures. *Communications of the ACM*, 17(1):18–20, January 1974.

[Aki74b] Akima, Hiroshi. Algorithm 474: Bivariate interpolation and smooth surface fitting based on local procedures. *Communications of the ACM*, 17(1):26–31, January 1974.

[Aki78a] Akima, Hiroshi. A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software*, 4(2):148–159, 1978.

[Aki78b] Akima, Hiroshi. Algorithm 526: Bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software*, 4(2):160–164, 1978.

[Ber76] Berry, G. Algorithm 92: The drawing of dashed lines. *Computer Journal*, 19(4):361–363, 1976.

[BGS80a] Barr, Roger C., Gallie, Thomas M., and Spach, Madison S. Automated production of contour maps for electrophysiology: II. Tri-

angulation, verification and organization of the geometric model. *Computers and Biomedical Research*, 13:154–170, 1980.

[BGS80b] Barr, Roger C., Gallie, Thomas M., and Spach, Madison S. Automated production of contour maps for electrophysiology: III. Construction of contour maps. *Computers and Biomedical Research*, 13:171–191, 1980.

[BH87] Brinkkemper, Sjaak and Hendriks, Harrie. A new algorithm for contour-plotting. Internal report 100, Department of Informatics, University of Nijmegen, Toernooiveld, 6525 ED NIJMEGEN, April 1987.

[BN64] Bengtsson, Bengt-Erik and Nordbeck, Stig. Construction of isarithms and isarithmic maps by computers. *BIT*, 4:87–105, 1964.

[Bro80] Brodlie, K.W., editor. *Mathematical Methods in Computer Graphics and Design*. Academic Press, 1980. Conference.

[BZ85] Beatson, R.K. and Ziegler, Z. Monotonicity preserving surface interpolation. *SIAM Journal on Numerical Analysis*, 22(2):401–411, April 1985.

[CLM69] Cottafava, G. and Le Moli, G. Automatic contour map. *Communications of the ACM*, 12(7):386–391, July 1969.

[Cra72] Crane, C.M. Algorithm 75: Contour plotting for functions specified at nodal points of an irregular mesh based on an arbitrary two-parameter coordinate system. *Computer Journal*, 15(4):382–384, 1972.

[Day63] Dayhoff, M.O. A contour-map program for X-ray crystallography. *Communications of the ACM*, 6(10):620–622, October 1963.

[DBV89] Dickinson, Robert R., Bartels, Richard H., and Vermeulen, Allan H. The interactive editing and contouring of empirical fields. *IEEE Computer Graphics and Applications*, 9:34–43, 1989.

[Ear85] Earnshaw, Rae A., editor. *Fundamental Algorithms for Computer Graphics*. NATO ASI Series F: Computer and Systems Sciences volume 17. Springer-Verlag, 1985.

[Eva74] Evans, D.J., editor. *Software for Numerical Mathematics*. Academic Press, 1974.

[FN80] Franke, Richard and Nielson, Greg. Smooth interpolation of large sets of scattered data. *International Journal for Numerical Methods in Engineering*, 15(11):1691–1704, 1980.

[FRKA78] Faber, D.H., Rutten-Keulemans, E.W.M., and Altona, C. Computer plotting of contour maps: An improved method. *Computer & Chemistry*, 3:51–53, 1978.

[GCR77] Gold, C.M., Charters, T.D., and Ramsden, J. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. In George,

James, editor, *Computer Graphics: Proceedings of SIGGRAPH '77*, pages 170–175, Summer 1977.

[LN79] Leipälä, T. and Nevalainen, O. A plotter sequencing system. *Computer Journal*, 22(4):313–316, 1979.

[McC71] McConalogue, D.J. Algorithm 66: An automated french-curve procedure for use with an incremental plotter. *Computer Journal*, 14(2):207–209, 1971.

[McL74] McLain, D.H. Drawing contours from arbitrary data points. *Computer Journal*, 17(4):318–324, 1974.

[McL76] McLain, D.H. Two dimensional interpolation from random data. *Computer Journal*, 19(2):178–181, 1976. Errata to this article: *Computer Journal* 19(4):384.

[Mer73] Merrill, R.D. Representation of contours and regions for efficient computer search. *Communications of the ACM*, 16(2):69–82, February 1973.

[ML72] Mor, M. and Lamdan, T. A new approach to automatic scanning of contour maps. *Communications of the ACM*, 15(9):809–812, September 1972.

[Mor68] Morse, Stephen P. A mathematical model for the analysis of contour-line data. *Journal of the ACM*, 15(2):205–220, 1968.

[Mor69] Morse, Stephen P. Concepts of use in contour map process. *Communications of the ACM*, 12(3):147–152, 1969.

[MR81] McAllister, David F. and Roulier, John A. An algorithm for computing a shape-preserving osculatory quadratic spline. *ACM Transactions on Mathematical Software*, 7(3):331–347, 1981.

[NP84] Ngai, Eugene C. and Profera, Jr., Charles E. Applications of bivariate interpolation to antenna related problems. *IEEE Transactions on Antennas and Propagation*, 32(7):735–739, July 1984.

[Pow74] Powell, M.J.D. Piecewise quadratic surface fitting for contour plotting. In Evans [Eva74], pages 253–274.

[Pre84a] Preusser, Albrecht. Computing contours by successive solution of quintic polynomial equations. *ACM Transactions on Mathematical Software*, 10(4):463–472, December 1984.

[Pre84b] Preusser, Albrecht. Algorithm 626. TRICP: A contour plot program for triangular meshes. *ACM Transactions on Mathematical Software*, 10(4):473–475, December 1984.

[Pre89] Preusser, Albrecht. Algorithm 671. FARB-E-2D: Fill area with bicubics on rectangles — a contour plot program. *ACM Transactions on Mathematical Software*, 15(1):79–89, 1989.

[Sab80] Sabin, M.A. Contouring - a review of methods for scattered data. In Brodlie [Bro80], pages 63–86. Conference.

[Sab85] Sabin, M.A. Contouring — the state of the art. In Earnshaw [Ear85], pages 411–482.

[Sch82] Schagen, I.P. Automatic contouring from scattered data points. *Computer Journal*, 25(1):7–11, 1982.

[Sew88] Sewell, Graville. Plotting contour surfaces of a function of three variables. *ACM Transactions on Mathematical Software*, 14(1):33–44, 1988.

[Sny78] Snyder, William V. Algorithm 531: Contour plotting. *ACM Transactions on Mathematical Software*, 4(3):290–294, September 1978.

[SS78] Schultheis, Hildegard and Schultheis, R. Algorithm 35: An algorithm for non-smoothing contour representations of two-dimensional arrays. *Computing. Archives for Informatics and Numeral Computation*, 19(4):381–387, 1978.

[ST81] Sibson, Robin and Thomson, Graeme D. A seamed quadratic element for contouring. *Computer Journal*, 24(4):378–382, 1981.

[Sut76a] Sutcliffe, D.C. An algorithm for drawing the curve $f(x, y) = 0$. *Computer Journal*, 19(3):246–249, 1976.

[Sut76b] Sutcliffe, D.C. A remark on a contouring algorithm. *Computer Journal*, 19(4):333–335, 1976.

[Sut80] Sutcliffe, D.C. Contouring over rectangular and skewed rectangular grids — an introduction. In Brodlie [Bro80], pages 39-62. Conference.

[War78] Ward, Stephen A. Real time plotting of approximate contour maps. *Communications of the ACM*, 21(9):788–790, September 1978.

[Wat74] Watkins, Steven L. Algorithm 483: Masked three-dimensional plot program with rotations. *Communications of the ACM*, 17(9):520–523, 1974.

[WP84] Watson, D.F. and Philip, G.M. Systematic triangulations. *Computer Vision, Grahhics and Image Processing*, 26:217–223, 1984.

[Zyd88] Zyda, Michael J. A decomposable algorithm for contour surface display generation. *ACM Transactions on Graphics*, 7(2):129–148, April 1988.