SLAC-391
UC-405
(M)

GKS UTILITIES FOR FORTRAN-77*

Robert C. Beach

Computation Research Group

Stanford Linear Accelerator Center

Stanford University

Stanford, CA 94309

January 1992

Prepared for the Department of Energy

under contract number DE-AC03-76SF00515.

Printed in the United States of America. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

---

* Manual.

Contents

Chapter 1

Introduction

This document describes a number of subroutines that can be useful in GKS graphic applications programmed in FORTRAN-77. The algorithms described here include subroutines to do the following:

1. Draw text characters in a more flexible manner than is possible with basic GKS.
2. Project two-dimensional and three-dimensional space onto two-dimensional space.
3. Draw smooth curves.
4. Draw two-dimensional projections of complex three-dimensional objects.

FORTRAN-77 is described in *American National Standard, Programming Language, FORTRAN* [ANS78]. GKS is described in *American National Standard for Information Systems: Computer Graphics - Graphical Kernel System (GKS) Functional Description* [ANS85a] and the FORTRAN-77 interface is described in *American National Standard for Information Systems: Computer Graphics - Graphical Kernel System (GKS) FORTRAN Binding* [ANS85b].

All of the subroutine names and additional enumeration types that will be described in this document begin with the letters "GZ." Since GKS itself does not have any subroutine names or enumeration types that begin with these letters, no confusion between the usual GKS subroutines and the ones described here should occur.

Many concepts will have to be defined in the following chapters. When a concept is first encountered, it will be given in *italics*. The information around the italicized word or phrase may be taken as its definition.

1.1. The Availability of the Subroutines

The subroutines described in this document are available on the IBM mainframe computers running at the Stanford Linear Accelerator Center. These computers run under the VM/XA operating system. Executable versions of the subroutines are contained in the file

```
GKSUTL TXTLIB U.
```

They may therefore be used by anyone at this installation who supplies the proper `TXTLIB` statement.

The source code is also available for those people who have to use the subroutines on another computer. The file

```
GKSUTLTX FORTRAN U
```

contains the text drawing subroutines described in Chapter 2. The file

```
GKSUTLTR FORTRAN U
```

contains the transformation subroutines described in Chapter 3. The file

```
GKSUTLCV FORTRAN U
```

contains the curve drawing subroutines described in Chapter 4. The file

```
GKSUTLSU FORTRAN U
```

contains the surface drawing subroutines described in Chapter 5. Finally, the file

```
GKSUTLUT FORTRAN U
```

contains a group of mathematical and error processing subroutines that are used by the other subroutines described here.

Since the these subroutines are written in something very close to strict FORTRAN-77, they themselves should be transportable to any computer with a FORTRAN-77 compiler and a GKS system. The only non-standard construction in the source code is the use of `INTEGER*2` arrays to store the definition of the character sets. These declarations can easily be changed to `INTEGER`; the only requirement is that the arrays can contain integers of up to 32767.

One possible modification is that of a control value, `INFN`, that appears in a number of subroutines. That value is used to check for things like singular matrices and to guard against division by zero. It may have to be changed for computers with differing word size or precision. In fact, it may be necessary to change this value on the host computer as more experience is accumulated with these subroutines.

Chapter 2

An Alternate Text Generator

There are a number of problems with the GKS text drawing subroutine, GTX, especially as it relates to scientific notation. The most basic problem is that the standards documents only specify a single font containing the ASCII character set. Such things as Greek letters will usually be supplied as extensions to the basic GKS in most implementations, but their font numbers and other properties can be very different among different implementations. Programs that use the Greek letters supplied by a GKS system will therefore probably be implementation dependent. Another problem is that the production of superscripts or subscripts is very difficult. The mixing of Roman and Greek letters is a single line of text is both difficult and implementation dependent.

The subroutines described in this chapter are an attempt to alleviate the problems described above. These subroutines can produce the upper and lower case Roman, Greek, Cyrillic, and Hebrew alphabets, and a wide variety of special characters. A versatile subscripting and superscripting ability is also available and diacritical marks can be applied to any letter. Finally, the subroutines should be transportable to almost any computer.

In addition, the characters are available in three fonts. However, to fully understand these fonts, it is necessary to describe how the subroutines work. The user of these text drawing subroutines supplies two character strings of equal length. The first string is the *primary* character string, and the second is the *secondary* string. The actual characters produced is determined by examining corresponding positions in the two strings. The first string gives an approximation to the desired character while the second string gives a modifier character. As an example, suppose the primary string is "AAA" and the secondary string is " LG." In this case, the first character drawn is an upper case Roman "A" (because the first secondary character is a blank), the second character is a lower case Roman "A" (because the second secondary character is an "L"), and the third character is a lower case Greek alpha (because the third secondary character is a "G"). The subroutines process these characters and break them down into polylines or fill areas, and call the appropriate GKS subroutine to send them to the workstation. Two of these fonts, the *simplex* and *duplex* fonts, are drawn with polylines while the third, the *solid* font, is drawn with fill areas. The simplex font minimizes the complexity of the characters, while the duplex font has some of the properties of typeset characters. The solid font can be useful when large lettering is required. Examples of all of the characters and their corresponding primary and secondary character are shown in Section 2.5 of this chapter.

The organization of the subroutines described in this chapter is similar to the GKS standards document. There is a single subroutine that is used to initialize this alternate text generator. There are two subroutines that break their primary and secondary character strings down into polylines or fill areas. There are four subroutines that may be used to set the attributes of the characters. Finally, there are four subroutines that can be used to obtain the current setting of the attributes.

2.1. Control Functions

This section describes a subroutine that must be called to initialize the alternate text generator. It may also be called at any time to reset the attributes to their default values.

2.1.1. Subroutine GZOPTX: Open the Alternate Text Generator

This subroutine may be used to initialize the attributes for the alternate GKS text drawing subroutines. If this subroutine is not called before the other alternate text drawing subroutines, the results are unpredictable.

The calling sequence is:

```
CALL GZOPTX
```

This subroutine does not have any parameters.

2.2. Output Functions

This section describes two subroutines that process the primary and secondary character strings and produce either polylines or fill areas. The first subroutine, GZTX, sends the polylines or fill areas directly to the active workstations. The second subroutine, GZTXS, sends the polylines or fill areas to a user supplied subroutine.

Since the data for the first subroutine is sent to the workstation by calling subroutines GPL or GFA, the user may control the display attributes of the characters, such as color, by setting the polyline or fill area attributes. It is the users responsibility to assure that the polyline or fill area attributes are appropriate for the characters being drawn; for example, the line type should be set to GLSOLI (solid) when polylines are drawn

These subroutines always produce their output in the "stroke" precision of GKS. It is therefore also the users responsibility to assure that the aspect ratio of the window and viewport of the normalization transformation are the same when the polylines or fill areas are sent to the workstation. If that is not the case, the characters, like the stroke precision characters of GKS, will be distorted.

2.2.1. Subroutine GZTX: Alternate Text

This subroutine may be used to draw a string of characters. The characters

produced by this subroutine are quite varied and include the upper and lower case Roman, Greek, Cyrillic, and Hebrew alphabets, and a wide variety of special characters. They may be drawn in a simplex, duplex, or solid font. A versatile subscripting and superscripting ability is also available. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GTX.

The calling sequence is:

```
CALL GZTX(PX,PY,PCHS,SCHS)
```

The input parameters are:

PX	A real value that gives the x coordinate of the location point of the character string in world coordinates.
PY	A real value that gives the y coordinate of the location point of the character string in world coordinates.
PCHS	A character string containing the primary characters.
SCHS	A character string containing the secondary characters.

2.2.2. Subroutine GZTXS: Alternate Text to User Supplied Subroutine

This subroutine may be used to process a string of characters in a manner similar to the way subroutine GZTX does. However, instead of sending the data directly to the workstation, this subroutine calls a user supplied subroutine with the data. The user supplied subroutine can do anything it wants with the data.

The calling sequence is:

```
CALL GZTXS(SUBR,PX,PY,PCHS,SCHS)
```

The input parameters are:

SUBR	An external variable that specifies the subroutine to which the computed polylines or fill areas will be sent. The calling sequence of this subroutine is the same as that of the GKS subroutines GPL or GFA.
PX	A real value that gives the x coordinate of the location point of the character string in world coordinates.
PY	A real value that gives the y coordinate of the location point of the character string in world coordinates.
PCHS	A character string containing the primary characters.
SCHS	A character string containing the secondary characters.

2.3. Output Attributes

The subroutines in this section may be used to set the attributes for the alternate GKS text generator. They are all similar to native GKS subroutines and perform operations similar to those native subroutines.

If one of these subroutines detects an error in the data supplied to it, the subroutine prints an error message and sets the attribute to its default value.

2.3.1. Subroutine GZSTXF: Set Alternate Text Font and Spacing

This subroutine may be used to set the text font and spacing for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GSTXFP.

The calling sequence is:

```
CALL GZSTXF(FONT,SPAC)
```

The input parameters are:

FONT An integer that gives the font to be used:
- GZSMPL (= 1) means the simplex font,
- GZDUPL (= 2) means the duplex font, and
- GZSOLD (= 3) means the solid font.

SPAC An integer that gives the spacing to be used:
- GZMONO (= 0) means mono-spacing, and
- GZPROP (= 1) means proportional spacing.

The default values are GZSMPL and GZPROP. The mono-space option does not work well when superscripts, subscripts, or character size or movement control is used.

2.3.2. Subroutine GZSCHH: Set Alternate Character Height

This subroutine may be used to set the character height for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GSCHH.

The calling sequence is:

```
CALL GZSCHH(CHH)
```

The input parameter is:

CHH A real value that gives the character height.

The default value is 0.01.

2.3.3. Subroutine GZSCHU: Set Alternate Character Up Vector

This subroutine may be used to set the character up vector for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GSCHUP.

The calling sequence is:

```
CALL GZSCHU(CHUX,CHUY)
```

The input parameters are:

CHUX A real value that gives the x component of the up vector in world coordinates.

CHUY A real value that gives the y component of the up vector in world coordinates.

The default values are 0.0 and 1.0.

2.3.4. Subroutine GZSTXL: Set Alternate Text Alignment

This subroutine may be used to set the text alignment for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GSTXAL.

The calling sequence is:

```
CALL GZSTXL(TXAH,TXAV)
```

The input parameters are:

TXAH An integer that gives the horizontal alignment to be used:
- GALEFT (= 1) means left,
- GACENT (= 2) means center, and
- GARITE (= 3) means right.

TXAV An integer that gives the vertical alignment to be used:
- GACAP (= 2) means top of text,
- GAHALF (= 3) means center of text, and
- GABASE (= 4) means bottom of text.

The default values are GALEFT and GABASE.

2.4. Inquiry Functions

The subroutines in this section may be used to obtain the attributes for the alternate GKS text generator. They are all similar to native GKS subroutines and perform operations similar to those native subroutines.

2.4.1. Subroutine GZQTXF: Inquire Alternate Text Font and Spacing

This subroutine may be used to obtain the text font and spacing for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GQTXFP.

The calling sequence is:

```
CALL GZQTXF(FONT,SPAC)
```

The output parameters are:

FONT An integer that gives the font being used:

GZSMPL (= 1) means the simplex font,
GZDUPL (= 2) means the duplex font, and
GZSOLD (= 3) means the solid font.

SPAC An integer that gives the spacing being used:
GZMONO (= 0) means mono-spacing, and
GZPROP (= 1) means proportional spacing.

2.4.2. Subroutine GZQCHH: Inquire Alternate Character Height

This subroutine may be used to obtain the character height for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GQCHH.

The calling sequence is:

```
CALL GZQCHH(CHH)
```

The output parameter is:

CHH A real value that gives the character height.

2.4.3. Subroutine GZQCHU: Inquire Alternate Character Up Vector

This subroutine may be used to obtain the character up vector for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GQCHUP.

The calling sequence is:

```
CALL GZQCHU(CHUX,CHUY)
```

The output parameters are:

CHUX A real value that gives the x component of the up vector in world coordinates.

CHUY A real value that gives the y component of the up vector in world coordinates.

These values are always returned as a unit vector.

2.4.4. Subroutine GZQTXL: Inquire Alternate Text Alignment

This subroutine may be used to obtain the text alignment for the alternate GKS text drawing subroutines. This subroutine performs an operation very similar to the operation done by the GKS subroutine named GQTXAL.

The calling sequence is:

```
CALL GZQTXL(TXAH,TXAV)
```

The output parameters are:

TXAH An integer that gives the horizontal alignment being used:
GALEFT (= 1) means left,
GACENT (= 2) means center, and
GARITE (= 3) means right.

TXAV An integer that gives the vertical alignment being used:
GACAP (= 2) means top of text,
GAHALF (= 3) means center of text, and
GABASE (= 4) means bottom of text.

2.5. The Alternate Character Sets

This section defines all of the characters that may be produced by subroutines GZTX or GZTXS. The following table gives the primary and secondary character followed by its description. The symbol "␣" stands for a blank.

The Upper Case Roman Alphabet:

A␣ Upper case Roman A
B␣ Upper case Roman B
C␣ Upper case Roman C
D␣ Upper case Roman D
E␣ Upper case Roman E
F␣ Upper case Roman F
G␣ Upper case Roman G
H␣ Upper case Roman H
I␣ Upper case Roman I
J␣ Upper case Roman J
K␣ Upper case Roman K
L␣ Upper case Roman L
M␣ Upper case Roman M
N␣ Upper case Roman N
O␣ Upper case Roman O
P␣ Upper case Roman P
Q␣ Upper case Roman Q
R␣ Upper case Roman R
S␣ Upper case Roman S
T␣ Upper case Roman T
U␣ Upper case Roman U
V␣ Upper case Roman V
W␣ Upper case Roman W
X␣ Upper case Roman X
Y␣ Upper case Roman Y
Z␣ Upper case Roman Z

The Lower Case Roman Alphabet:

AL Lower case Roman A
BL Lower case Roman B
CL Lower case Roman C
DL Lower case Roman D
EL Lower case Roman E
FL Lower case Roman F
GL Lower case Roman G
HL Lower case Roman H
IL Lower case Roman I
JL Lower case Roman J
KL Lower case Roman K
LL Lower case Roman L
ML Lower case Roman M
NL Lower case Roman N
OL Lower case Roman O
PL Lower case Roman P
QL Lower case Roman Q
RL Lower case Roman R
SL Lower case Roman S
TL Lower case Roman T
UL Lower case Roman U
VL Lower case Roman V
WL Lower case Roman W
XL Lower case Roman X
YL Lower case Roman Y
ZL Lower case Roman Z

Upper Case Auxiliary Roman Characters:

10 Upper case Latin and Scandinavian ligature AE
D0 Upper case Icelandic Eth
L0 Upper case Polish suppressed L
00 Upper case Scandinavian O with slash
20 Upper case French ligature OE
T0 Upper case Icelandic Thorn

Lower Case Auxiliary Roman Characters:

A1 Lower case alternate Roman A
11 Lower case Latin and Scandinavian ligature AE
D1 Lower case Icelandic Eth
31 Lower case Roman ligature FF
41 Lower case Roman ligature FI
51 Lower case Roman ligature FL
61 Lower case Roman ligature FFI
71 Lower case Roman ligature FFL

G1 Lower case alternate Roman G
I1 Lower case dotless Roman I
J1 Lower case dotless Roman J
L1 Lower case Polish suppressed L
O1 Lower case Scandinavian O with slash
21 Lower case French ligature OE
S1 Lower case German double S
T1 Lower case Icelandic Thorn

The Upper Case Greek Alphabet:

AF Upper case Greek Alpha
BF Upper case Greek Beta
GF Upper case Greek Gamma
DF Upper case Greek Delta
EF Upper case Greek Epsilon
ZF Upper case Greek Zeta
HF Upper case Greek Eta
QF Upper case Greek Theta
IF Upper case Greek Iota
KF Upper case Greek Kappa
LF Upper case Greek Lambda
MF Upper case Greek Mu
NF Upper case Greek Nu
XF Upper case Greek Xi
OF Upper case Greek Omicron
PF Upper case Greek Pi
RF Upper case Greek Rho
SF Upper case Greek Sigma
TF Upper case Greek Tau
UF Upper case Greek Upsilon
FF Upper case Greek Phi
CF Upper case Greek Chi
YF Upper case Greek Psi
WF Upper case Greek Omega

The Lower Case Greek Alphabet:

AG Lower case Greek Alpha
BG Lower case Greek Beta
GG Lower case Greek Gamma
DG Lower case Greek Delta
EG Lower case Greek Epsilon
ZG Lower case Greek Zeta
HG Lower case Greek Eta
QG Lower case Greek Theta

IG Lower case Greek Iota
KG Lower case Greek Kappa
LG Lower case Greek Lambda
MG Lower case Greek Mu
NG Lower case Greek Nu
XG Lower case Greek Xi
OG Lower case Greek Omicron
PG Lower case Greek Pi
RG Lower case Greek Rho
SG Lower case Greek Sigma
TG Lower case Greek Tau
UG Lower case Greek Upsilon
FG Lower case Greek Phi
CG Lower case Greek Chi
YG Lower case Greek Psi
WG Lower case Greek Omega
1G Lower case Greek Epsilon (variant)
2G Lower case Greek Theta (variant)
3G Lower case Greek Pi (variant)
4G Lower case Greek Rho (variant)
5G Lower case Greek Sigma (variant)
6G Lower case Greek Phi (variant)

The Upper Case Cyrillic Alphabet:

AB Upper case Cyrillic Ah
BB Upper case Cyrillic Beh
VB Upper case Cyrillic Veh
GB Upper case Cyrillic Geh
DB Upper case Cyrillic Deh
EB Upper case Cyrillic Yeh
XB Upper case Cyrillic Zheh
ZB Upper case Cyrillic Zeh
IB Upper case Cyrillic Ee
1B Upper case Cyrillic Ee S Kratkoy
KB Upper case Cyrillic Kah
LB Upper case Cyrillic El
MB Upper case Cyrillic Em
NB Upper case Cyrillic En
OB Upper case Cyrillic Oh
PB Upper case Cyrillic Peh
RB Upper case Cyrillic Err
SB Upper case Cyrillic Ess
TB Upper case Cyrillic Teh
UB Upper case Cyrillic Ooh

FB Upper case Cyrillic Ef
HB Upper case Cyrillic Kha
CB Upper case Cyrillic Tseh
2B Upper case Cyrillic Cheh
3B Upper case Cyrillic Shah
4B Upper case Cyrillic Shchah
QB Upper case Cyrillic Tvyordy Znak
YB Upper case Cyrillic Yery
5B Upper case Cyrillic Myakhki Znak
6B Upper case Cyrillic Eh Oborotnoye
WB Upper case Cyrillic Yoo
JB Upper case Cyrillic Yah

The Lower Case Cyrillic Alphabet:

AC Lower case Cyrillic Ah
BC Lower case Cyrillic Beh
VC Lower case Cyrillic Veh
GC Lower case Cyrillic Geh
DC Lower case Cyrillic Deh
EC Lower case Cyrillic Yeh
XC Lower case Cyrillic Zheh
ZC Lower case Cyrillic Zeh
IC Lower case Cyrillic Ee
1C Lower case Cyrillic Ee S Kratkoy
KC Lower case Cyrillic Kah
LC Lower case Cyrillic El
MC Lower case Cyrillic Em
NC Lower case Cyrillic En
OC Lower case Cyrillic Oh
PC Lower case Cyrillic Peh
RC Lower case Cyrillic Err
SC Lower case Cyrillic Ess
TC Lower case Cyrillic Teh
UC Lower case Cyrillic Ooh
FC Lower case Cyrillic Ef
HC Lower case Cyrillic Kha
CC Lower case Cyrillic Tseh
2C Lower case Cyrillic Cheh
3C Lower case Cyrillic Shah
4C Lower case Cyrillic Shchah
QC Lower case Cyrillic Tvyordy Znak
YC Lower case Cyrillic Yery
5C Lower case Cyrillic Myakhki Znak
6C Lower case Cyrillic Eh Oborotnoye

WC Lower case Cyrillic Yoo
JC Lower case Cyrillic Yah

The Hebrew Alphabet:

AH Hebrew Aleph
BH Hebrew Beth
GH Hebrew Gimel
DH Hebrew Daleth
HH Hebrew He
VH Hebrew Vav
ZH Hebrew Zayin
CH Hebrew Cheth
OH Hebrew Teth
YH Hebrew Yod
KH Hebrew Kaph
LH Hebrew Lamed
MH Hebrew Mem
NH Hebrew Nun
SH Hebrew Sameth
XH Hebrew Ayin
PH Hebrew Pe
EH Hebrew Sadhe
QH Hebrew Koph
RH Hebrew Resh
WH Hebrew Sin/Shin
TH Hebrew Tav
1H Hebrew Kaph (end of word)
2H Hebrew Men (end of word)
3H Hebrew Nun (end of word)
4H Hebrew Pe (end of word)
5H Hebrew Sadhe (end of word)

The Numerals:

0␣ Numeral 0
1␣ Numeral 1
2␣ Numeral 2
3␣ Numeral 3
4␣ Numeral 4
5␣ Numeral 5
6␣ Numeral 6
7␣ Numeral 7
8␣ Numeral 8
9␣ Numeral 9

Common Special Symbols:

␣␣ Blank
+␣ Plus sign
-␣ Minus sign
*␣ Asterisk
/␣ Slash mark
=␣ Equal sign
.␣ Period
,␣ Comma
(␣ Left parenthesis
)␣ Right parenthesis

Special Symbols for Punctuation:

.P Colon
,P Semi-colon
EP Exclamation mark
UP Question mark
IP Interrobang
FP Inverted exclamation
VP Inverted question
AP Apostrophe
QP Quotation marks
0P Single left quote
1P Single right quote
2P Double left quote
3P Double right quote
SP New section
PP New paragraph or Pilcrow sign
DP Dagger
RP Double dagger

Additional Special Symbols:

DS Dollar sign
CS Cent sign
SS British Sterling
YS Japanese Yen
QS International currency symbol
+S Ampersand
PS Pound sign
AS At sign
OS Copyright
GS Registered
0S Percent sign
1S Per thousand sign
VS Vertical line

IS Broken vertical line
WS Double vertical line
US Underline
NS Not sign
/S Backwards slash
(S Left bracket
)S Right bracket
LS Left brace
RS Right brace
BS Left angle bracket
ES Right angle bracket
XS Accent mark
TS Caret mark

Mathematical Special Symbols:

.M Dot product
XM Cross product
/M Division sign
PM Group plus
*M Group multiply
+M Plus or minus
-M Minus or plus
AM And
VM Or
UM Therefore
WM Since
LM Less than
GM Greater than
MM Less than or equal
HM Greater than or equal
3M Much less than
4M Much greater than
NM Not equal
=M Identically equal
KM Approximately equal
CM Congruent to
SM Similar to
FM Approximate
RM Proportional to
TM Perpendicular to
2M Surd
DM Degrees
IM Integral sign
JM Line integral

YM Partial derivative
ZM Del
(M Left floor bracket
)M Right floor bracket
BM Left ceiling bracket
EM Right ceiling bracket
OM Infinity

Set Theoretic Special Symbols:
ET Existential quantifier
AT Universal quantifier
MT Membership symbol
NT Membership negation
IT Intersection
UT Union
LT Contained in
GT Contains
KT Contained in or equals
FT Contains or equals

Physics Special Symbols:
HK H-bar
LK Lambda-bar

Astronomical Special Symbols:
HA Sun
MA Mercury
VA Venus
EA Earth
WA Mars
JA Jupiter
SA Saturn
UA Uranus
NA Neptune
PA Pluto
DA Moon
CA Comet
*A Star
XA Ascending node
YA Descending node
KA Conjunction
QA Quadrature
TA Opposition
OA Aries

1A Taurus
2A Gemini
3A Cancer
4A Leo
5A Virgo
6A Libra
7A Scorpius
8A Sagittarius
9A Capricornus
AA Aquarius
BA Pisces

Drawing Symbols, Arrows, and Pointers:

0W Underscore
1W Midscore
2W Overscore
UW Up arrow
DW Down arrow
LW Left arrow
RW Right arrow
BW Left/right arrow

Diacritical Marks:

GD Grave accent
AD Acute accent
HD Hat or circumflex
TD Tilde or squiggle
MD Macron or bar
BD Breve accent
DD Dot accent
UD Umlaut or dieresis
RD Ring or circle
VD Caron, hacek, or check
LD Long Hungarian umlaut
WD Over arrow
CD Cedilla accent
-D Under bar
.D Under dot
,D Under dots
PD Prime

Horizontal and Vertical Movement Control:

␣U Null
0U Backwards blank

1U Half blank
2U Half backwards blank
3U Third blank
4U Third backwards blank
5U Sixth blank
6U Sixth backwards blank
1V Half up movement
2V Half down movement
3V Third up movement
4V Third down movement
5V Sixth up movement
6V Sixth down movement

Subscript and Superscript Control:
0X Enter subscript mode
1X Leave subscript mode
2X Enter superscript mode
3X Leave superscript mode

Character Size Control:
0Y Increase size by one-half
1Y Decrease size by one-third
2Y Increase size by one-third
3Y Decrease size by one-fourth
4Y Increase size by one-sixth
5Y Decrease size by one-seventh

Position Control:
0Z Put current state in first save area
1Z Restore state from first save area
2Z Put current state in second save area
3Z Restore state from second save area
4Z Put current state in third save area
5Z Restore state from third save area
6Z Put current state in fourth save area
7Z Restore state from fourth save area

In addition to the primary and secondary character pairs shown above, most of the printable characters in the ASCII character set as described in *American National Standard for Information Systems: Coded Character Sets, 7-bit American National Standard Code for Information Interchange (7-bit ASCII)* [ANS86] will be produced with a secondary character of blank. Thus, if the primary character is a lower case Roman letter and the secondary character is a blank, then the proper character will be produced. The user, however, is encouraged to use the character

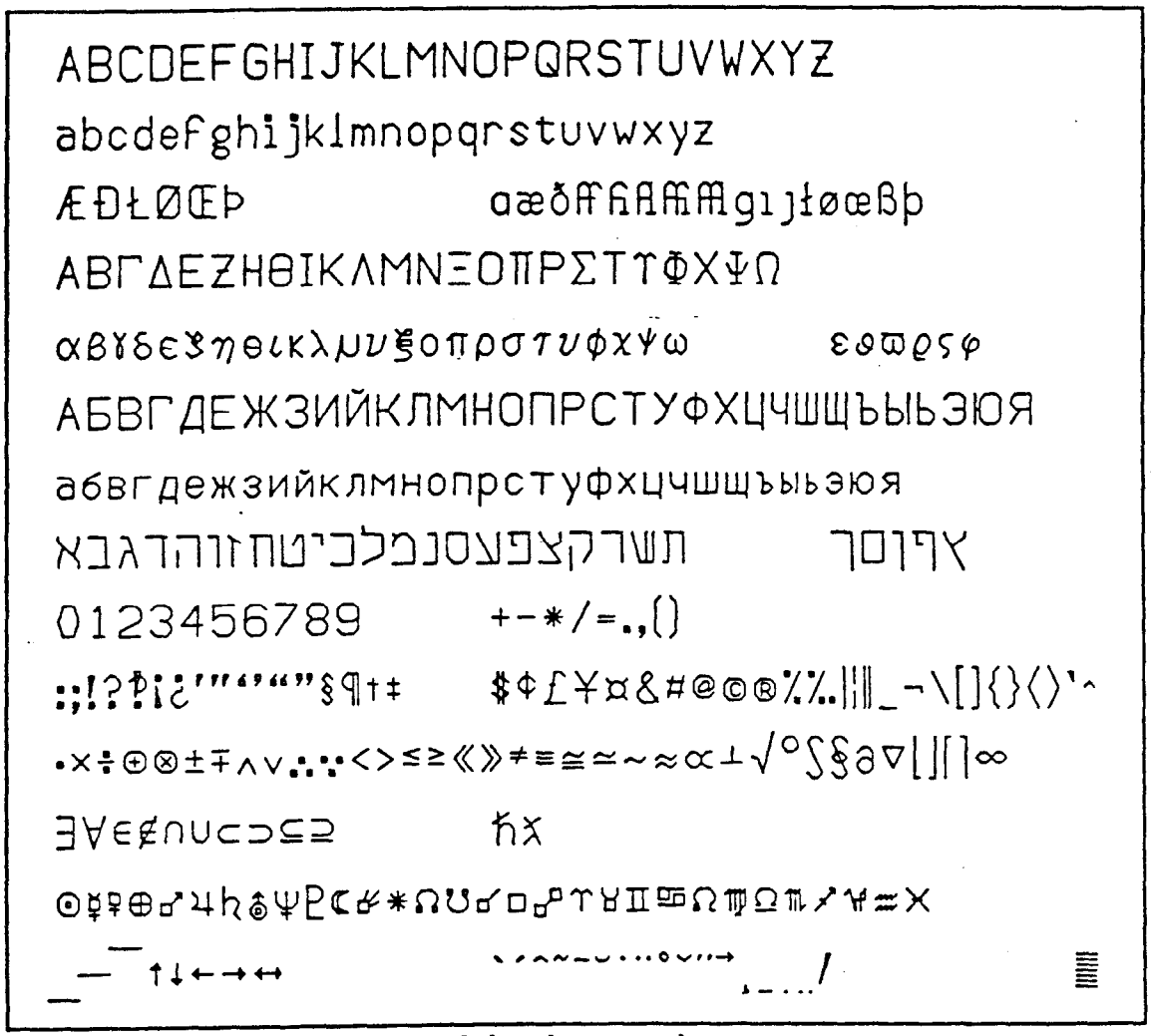

Figure 2.1. The simplex font of the alternate character set

pairs given in the above tables. The use of these character pairs will enhance the portability of the application program to non-ASCII computers.

The underscore, midscore, and overscore characters in the above table have some special properties. The purpose of these characters is to allow the programmer to draw lines under or over a line of text. Two consecutive underscore characters, for example, will join together into a single line (this is not true of the underline character). Thus the programmer, with some difficulty, can generate such things as fractions. The overscore will also join properly with the surd character to form a full radical sign.

The diacritical marks may be used immediately following any drawn character or a full sized blank. When this is done, the mark will attach itself to the preceding character and will be centered on that character. The prime mark is different than the others. The prime is normally used as a superscript on another symbol. More than one prime may be used in a superscript and the spacing will be appropriately close. However, this may mean that a partial space will have to be inserted if

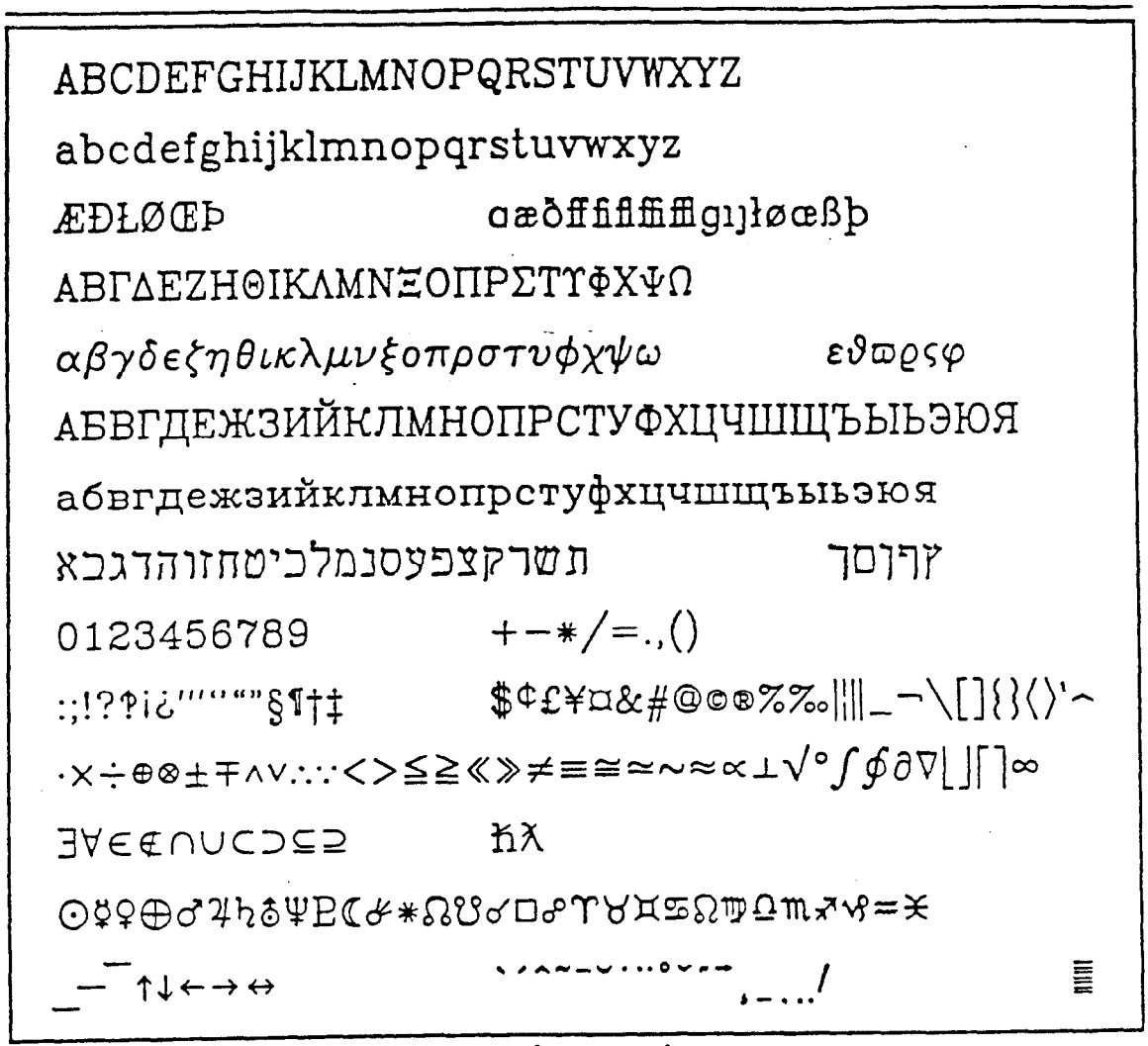

Figure 2.2. The duplex font of the alternate character set

something follows a prime.

After a character is drawn, it is always followed by a short blank space before the next character is drawn. When the character is a full blank, it produces a space representing the blank and then the blank space that follows all characters. The fractional blanks refer only to the space that represents the space itself. The backwards blanks cause exactly enough movement to eliminate the space representing the blank and its following space. Thus, a "third blank" followed by a "third backwards blank" will exactly cancel each other.

The alternate character generators usually produce characters of differing widths; thus the upper case letter "M" is about twice as wide as the upper case "I", and most lower case letters are about three-fourths as wide as most upper case letters. This results in a more pleasing appearance, but also causes some problems. If, for example, a letter is to carry both a superscript and subscript, something equivalent to a backspace would be necessary, but the amount backspaced would depend on the characters in the superscript (or subscript). To overcome this prob-

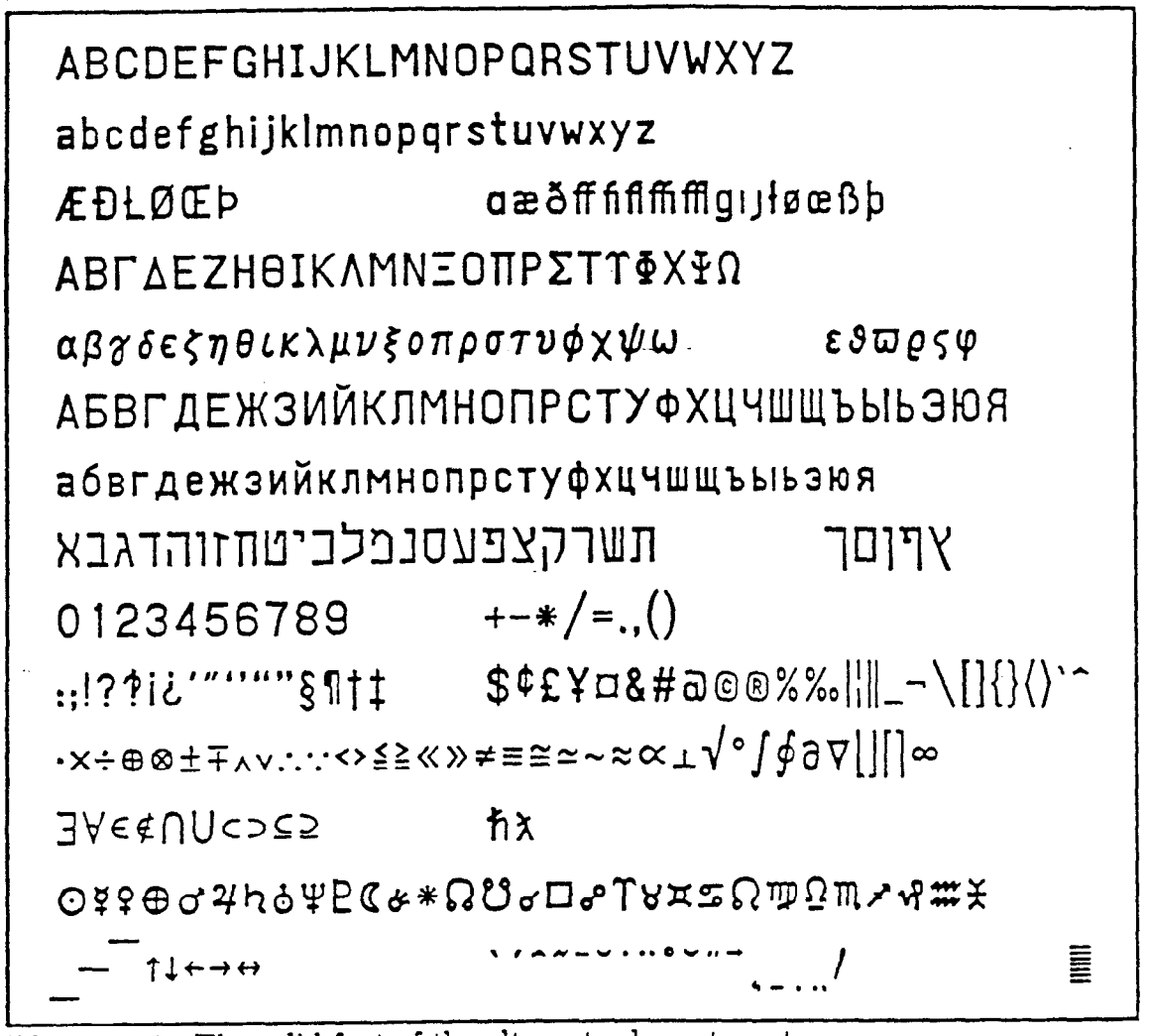

Figure 2.3. The solid font of the alternate character set

lem, a group of position control characters have been introduced which cause the stroke generator to save its current position and state. Another control character in a later part of the string can cause the earlier state of the stroke generator to be restored. There are four independent save-restore control character pairs available. The scope of these save-restore pairs is a single call to subroutine GZTX or GZTXS. That is, you cannot save a position in one call to one of these subroutines and try to use it in a later call. If you try to use a position without saving it in an earlier part of the string, you will obtain the position of the beginning of the string.

The alternate character set in the simplex font is shown in Figure (2.1), the duplex font is shown in Figure (2.2), and Figure (2.3) shows the solid font. The order of the characters in the figures is the same as in the preceding table. The character in the lower right of these figures is produced when an invalid character pair is specified. The average number of polyline end points per character in the simplex font is 7.8 and the maximum number is 21 (the lower case Roman G and the lower case ligature AE). The average number of polyline end points per character

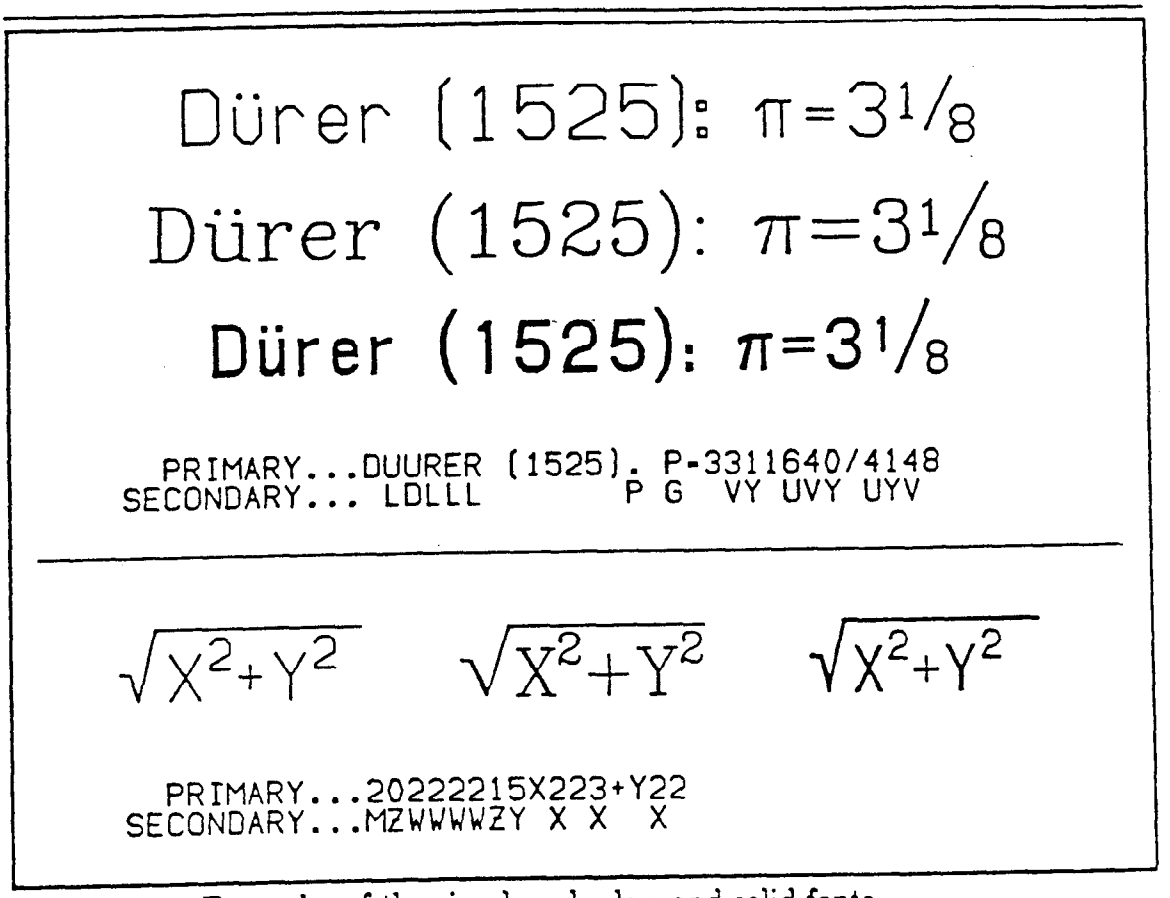

Figure 2.4. Examples of the simplex, duplex, and solid fonts

in the duplex font is 22.4 and the maximum number is 62 (the upper case Cyrillic Zheh). The average number of fill area vertex points per character in the solid font is 23.6 and the maximum number is 94 (the ascending and descending node symbols).

Many of the characters in the duplex font were designed by A. V. Hershey and are described by him in *Calligraphy for Computers* [Her67].

A large number of interesting constructions are possible with these character generators. Some examples are shown in Figure (2.4). In producing that figure, the primary and secondary characters were drawn with the simplex font in the mono-spaced mode. The other parts of the figure were done with the simplex, duplex, or solid fonts in the proportionally spaced mode.

Chapter 3

Projective Transformations

This chapter describes a group of subroutines that may be used to define projective transformations from two-dimensional or three-dimensional space into two-dimensional space. Subroutines are provided which generate the transformations and encode them as a matrix. Other subroutines are then provided that take a point, in two-dimensional or three-dimensional space, and project them into two-dimensional space. The mathematical derivation of all of these projective transformation algorithms is given in *An Introduction to the Curves and Surfaces of Computer-Aided Design* [Bea91].

One use of the two-dimensions to two-dimensions transformation is in digitizing photographs. If the photograph contains a figure of known dimensions then the transformation from real two-dimensional space to the coordinate system of the photograph can often be determined. A projective transformation is also the physically correct transformation if the optical system of the camera approximates a pinhole camera.

The three-dimensions to two-dimensions transformations are useful whenever two-dimensional images of three dimensional objects are required.

These transformations have many desirable properties. One of the most important is that they transform straight lines into straight lines. Another advantage is that neither the generation of the transformation nor the projection of a point is computationally expensive.

If one of the transformation generating subroutines determines that the transformation does not exist, it sets an error indicator and returns to the caller. The subroutines that project a point should always work unless they are supplied with extremely large coordinates.

3.1. Two-dimensions to Two-dimensions Projective Transformations

This section describes a means of generating and using a projective transformation from two-dimensional space to two-dimensional space. The transformation is defined by giving four points in the source coordinate system and the corresponding four points in the target coordinate system. The resulting projective transformation will always be computable provided no three of the points lie on a straight line in either coordinate system.

There is, however, a problem with points that transform into a *point at infinity*. To understand this problem, refer to Figure (3.1). In this figure, the four points on the irregular quadrilateral, $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, and $\mathbf{P}_4$, are to be transformed into the

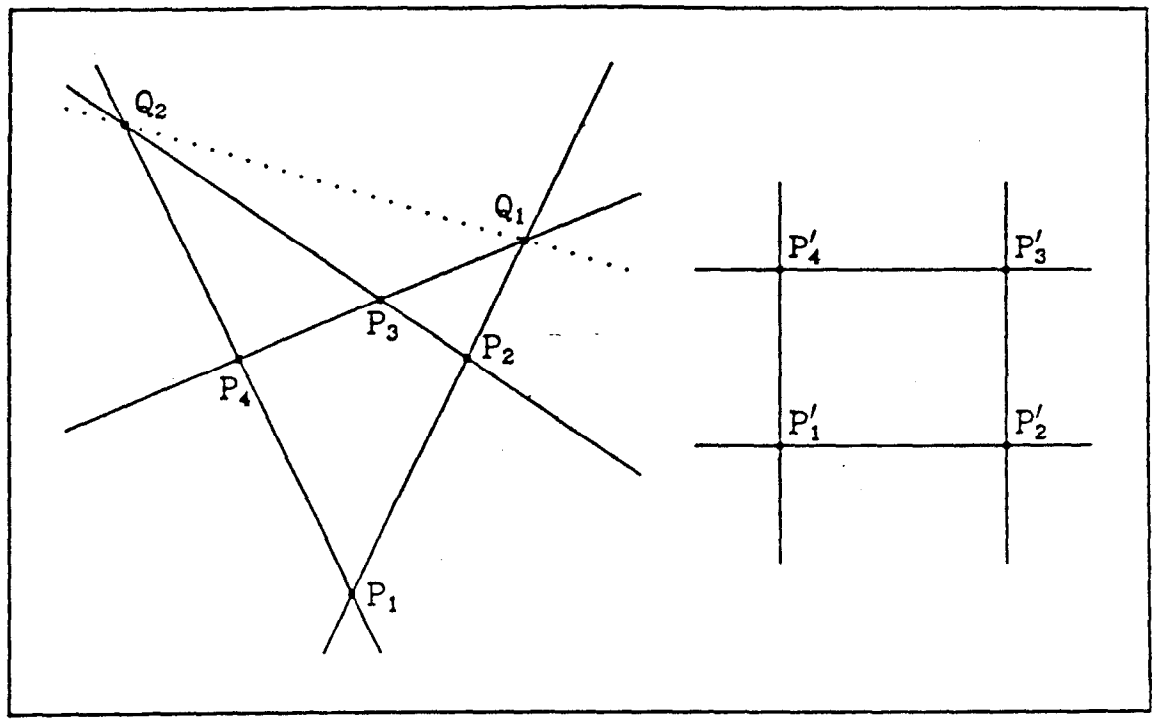

Figure 3.1. A two-dimensions to two-dimensions projective transformation

rectangle described by $\mathbf{P}'_1$, $\mathbf{P}'_2$, $\mathbf{P}'_3$, and $\mathbf{P}'_4$. The line through the points $\mathbf{P}_1$ and $\mathbf{P}_2$ intersects the line through the points $\mathbf{P}_3$ and $\mathbf{P}_4$ at $\mathbf{Q}_1$. The lines through the corresponding $\mathbf{P}'_i$ points do not intersect, or rather, they intersect at infinity. The point $\mathbf{Q}_1$ therefore transforms into a point at infinity. The point $\mathbf{Q}_2$ similarly transforms into a point at infinity. Since straight lines are preserved under the transformation, all of the points on the dotted line through $\mathbf{Q}_1$ and $\mathbf{Q}_2$ transform into points at infinity. The subroutine that transforms a point from one coordinate system to another will determine if the given point transforms into a point at infinity and warn the caller.

3.1.1. Subroutine GZ22PJ: Generate a Transformation

This subroutine may be used to generate a two-dimensions to two-dimensions projective transformation that carries four given points into four given points.

The calling sequence is:

```
CALL GZ22PJ(PXAS,PYAS,PXAT,PYAT,IERR,PTRN)
```

The input parameters are:

PXAS — A real array of dimension 4 containing the x coordinates of the source points.

PYAS — A real array of dimension 4 containing the y coordinates of the source points.

PXAT A real array of dimension 4 containing the x coordinates of the target points.

PYAT A real array of dimension 4 containing the y coordinates of the target points.

The output parameters are:

IERR An integer giving an error flag. A nonzero value means the transformation could not be computed.

PTRN A real array of dimension (3,3) containing the projective transformation.

3.1.2. Subroutine GZ22TR: Transform a Point

This subroutine may be used to transform a point using a two-dimensions to two-dimensions projective transformation. A flag indicates if the projected point is a finite point or a point at infinity.

The calling sequence is:

```
CALL GZ22TR(PTRN,PAS,PAP,FLAG)
```

The input parameters are:

PTRN A real array of dimension (3,3) containing the projective transformation.

PAS A real array of dimension 2 containing the source point.

The output parameters are:

PAP A real array of dimension 2 containing the projected point.

FLAG A real value that indicates whether a finite point or a point at infinity has been computed. If this value is nonzero, PAP contains the finite coordinates of the projected point. If this value is zero, PAP is a unit vector pointing in the direction of the point at infinity.

3.2. Three-dimensions to Two-dimensions Projective Transformations

This section describes a number of ways to generate a three-dimensions to two-dimensions projective transformation.

In the first case the projection of a point in three-dimensional space is defined by an eye point and a projection plane as shown in Figure (3.2). The plane is defined by an origin point, **O**, on the plane, and two direction vectors, **H** and **V**. **H** is the "horizontal" direction and **V** is the "vertical" direction. These two direction vectors will often be perpendicular to each other. A point on the plane, **Q**, is found by starting at **O**, and moving parallel to **H** the necessary distance and then parallel to **V** the necessary distance. Thus, **Q** is represented as

$$\mathbf{Q} = \mathbf{O} + \xi\mathbf{H} + \eta\mathbf{V}.$$

Thus the vectors **H** and **V** impose a coordinate system on the plane. The projection of a point **P** onto the plane is then obtained by drawing a straight line through

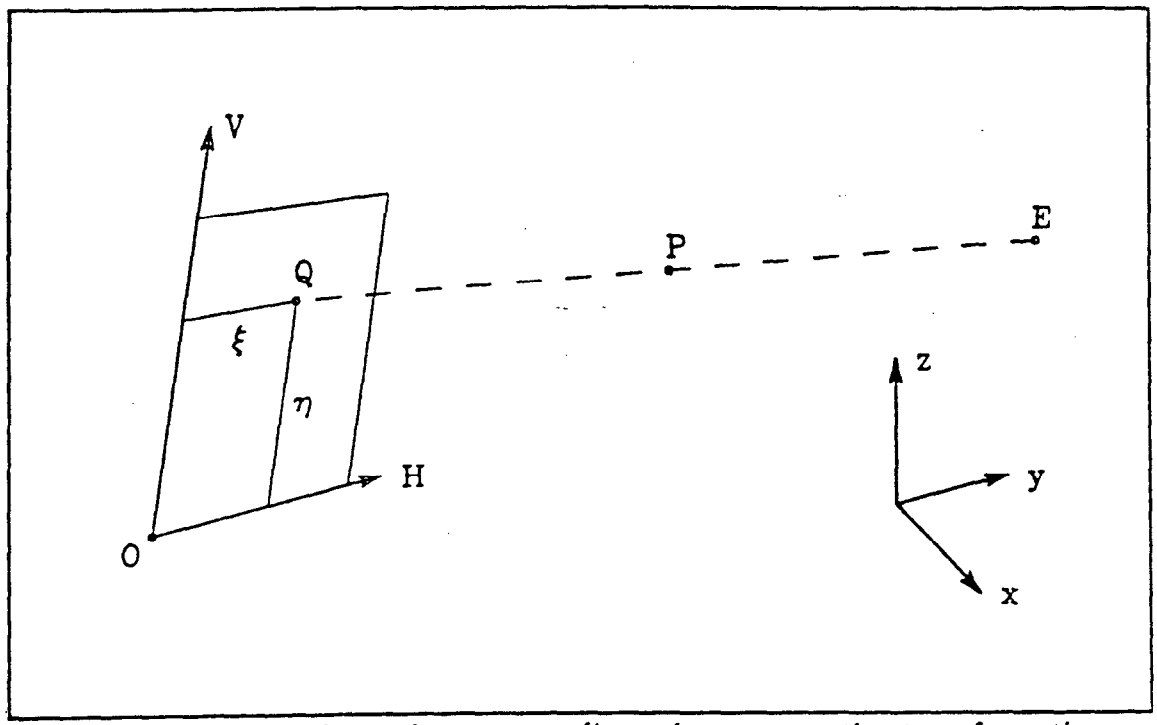

Figure 3.2. A three-dimensions to two-dimensions perspective transformation

the eye point, E, and the point P until it meets the plane. The ξ and η values of the intersection point are the coordinates of the projected point in two-dimensional space. In many applications H and V are perpendicular and the vector from E to O is perpendicular to both H and V but that is not necessary in these subroutines. This type of transformation is known as a *perspective transformation*. These transformations are best understood by imagining a viewer at the eye point, looking toward the origin point.

The second type of three-dimensions to two-dimensions transformation that is described here is known as a *parallel transformation*. It is formed by projecting a given point, P, parallel to a fixed direction, D, as shown in Figure (3.3). It is again common to have H and V perpendicular and to have D perpendicular to both H and V.

In the case of a perspective transformation, the horizontal and vertical directions must be distinct and neither may point at the eye point. In a parallel transformation, the horizontal, vertical, and projection directions must all be distinct.

The preceding scheme is very general but is not very easy to use. The problem is that the origin point is not easy to determine. For this reason, a second way to define the projection plane is provided. In this second scheme, the projection plane is defined by selecting a rectangular area on the projection plane and thinking of it as the "projection screen." The projection screen is orientated so that one set of parallel sides is parallel to the x-y plane. The projection screen is defined by giving the center point of the screen, C, and its horizontal and vertical size, h and v. In the case of a perspective transformation, the projection plane is perpendicular to

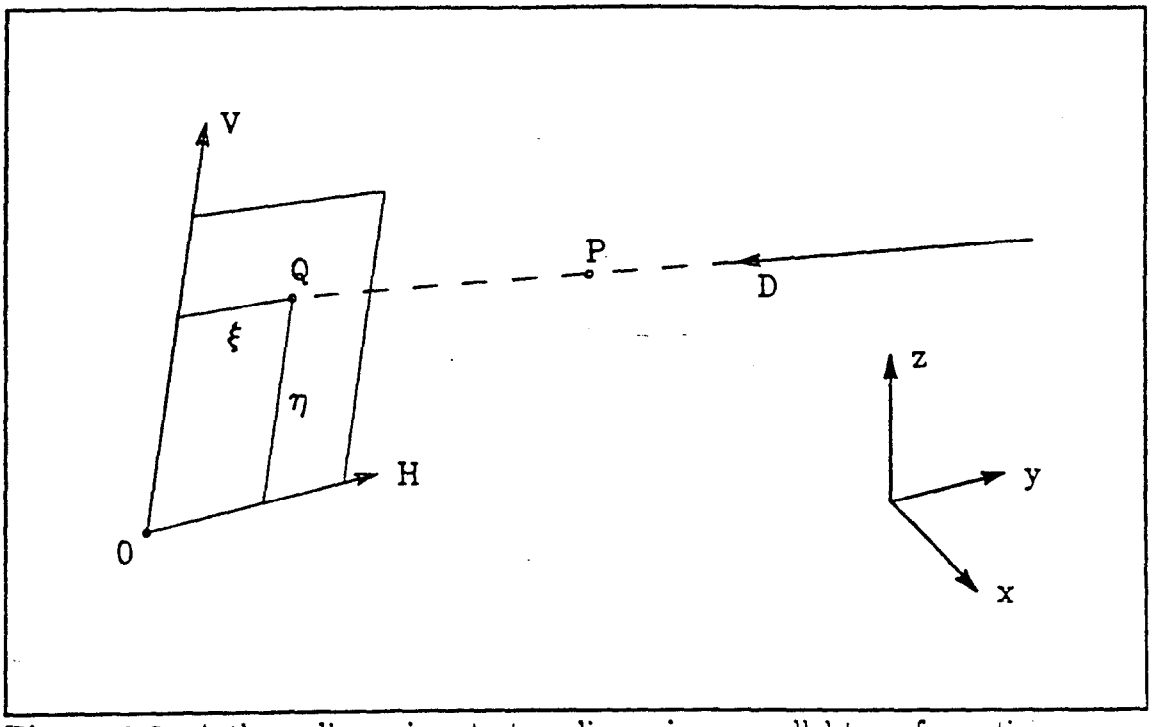

Figure 3.3. A three-dimensions to two-dimensions parallel transformation

the vector from E to C; in the case of a parallel transformation, it is perpendicular to D. To define the coordinate system on the projection screen, the maximum and minimum values of ξ and η are given. This information is all shown in Figure (3.4).

The maximum and minimum values of ξ and η are given by a real array of dimension (2,2). The format of the data is

$$\texttt{SCRC} = \begin{pmatrix} \texttt{SCRC}(1,1) & \texttt{SCRC}(1,2) \\ \texttt{SCRC}(2,1) & \texttt{SCRC}(2,2) \end{pmatrix} = \begin{pmatrix} \xi_{min} & \eta_{min} \\ \xi_{max} & \eta_{max} \end{pmatrix}.$$

For most usage, the aspect ratio given by h and v should be the same as that defined by the maximum and minimum values of ξ and η. That is, the values should satisfy

$$\frac{\eta_{max} - \eta_{min}}{\xi_{max} - \xi_{min}} = \frac{v}{h}.$$

However, the subroutines do not enforce this constraint.

A perspective transformation can also produce points at infinity. All of the points on the plane through the eye point and parallel to the projection plane map into points at infinity except for the eye point itself. The eye point has no corresponding point. A parallel transformation never produces points at infinity.

3.2.1. Subroutine GZ32PT: Generate a Perspective Transformation (I)

This subroutine may be used to generate a three-dimensions to two-dimensions perspective transformation. The transformation is defined by giving the projection

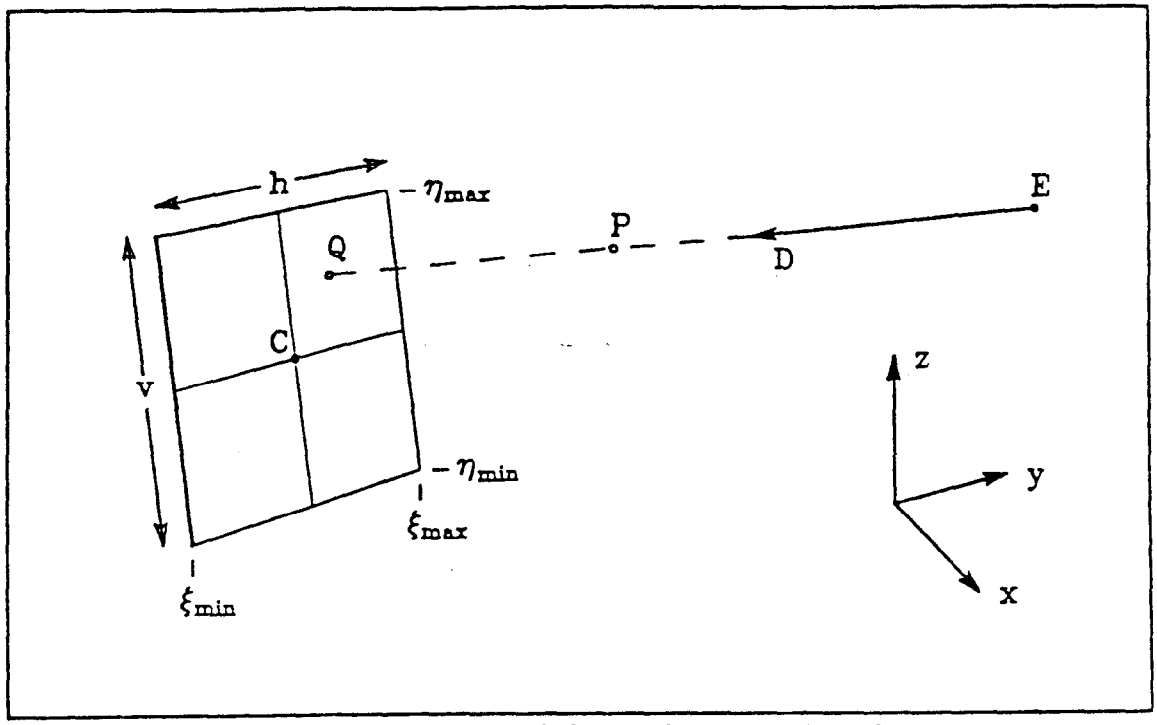

Figure 3.4. An alternate method of defining the projection plane

plane and an eye point. The projection plane is specified by giving a point on the plane and a horizontal and vertical direction within the plane.

The calling sequence is:

```
CALL GZ32PT(PO,HD,VD,PE,IERR,PTRN)
```

The input parameters are:

- PO — A real array of dimension 3 containing the origin point on the projection plane.
- HD — A real array of dimension 3 containing the horizontal direction in the projection plane.
- VD — A real array of dimension 3 containing the vertical direction in the projection plane.
- PE — A real array of dimension 3 containing the eye point.

The output parameters are:

- IERR — An integer giving an error flag. A nonzero value means the transformation could not be computed.
- PTRN — A real array of dimension (3,4) containing the projective transformation.

3.2.2. Subroutine GZ32AT: Generate a Perspective Transformation (II)

This subroutine provides an alternate way to generate a three-dimensions to two-dimensions perspective transformation. The transformation is defined by giving the

projection plane and an eye point. In this case, the projection plane is specified by giving the center point of a projection screen, its size, and the limits of the coordinates on the screen.

The calling sequence is:

```
CALL GZ32AT(PC,HZ,VZ,SCRC,PE,IERR,PTRN)
```

The input parameters are:

- PC — A real array of dimension 3 containing the center point on the projection plane.
- HZ — A real value giving the size of the screen in the horizontal direction.
- VZ — A real value giving the size of the screen in the vertical direction.
- SCRC — A real array of dimension (2,2) containing the limits of the coordinate system on the screen.
- PE — A real array of dimension 3 containing the eye point.

The output parameters are:

- IERR — An integer giving an error flag. A nonzero value means the transformation could not be computed.
- PTRN — A real array of dimension (3,4) containing the projective transformation.

3.2.3. Subroutine GZ32PL: Generate a Parallel Transformation (I)

This subroutine may be used to generate a three-dimensions to two-dimensions parallel transformation. The transformation is defined by giving the projection plane and a projection direction. The projection plane is specified by giving a point on the plane and a horizontal and vertical direction within the plane.

The calling sequence is:

```
CALL GZ32PL(PO,HD,VD,PD,IERR,PTRN)
```

The input parameters are:

- PO — A real array of dimension 3 containing the origin point on the projection plane.
- HD — A real array of dimension 3 containing the horizontal direction in the projection plane.
- VD — A real array of dimension 3 containing the vertical direction in the projection plane.
- PD — A real array of dimension 3 containing the projection direction.

The output parameters are:

- IERR — An integer giving an error flag. A nonzero value means the transformation could not be computed.
- PTRN — A real array of dimension (3,4) containing the projective transformation.

3.2.4. Subroutine GZ32AL: Generate a Parallel Transformation (II)

This subroutine provides an alternate way to generate a three-dimensions to two-dimensions parallel transformation. The transformation is defined by giving the projection plane and a projection direction. In this case, the projection plane is specified by giving the center point of a projection screen, its size, and the limits of the coordinates on the screen.

The calling sequence is:

```
CALL GZ32AL(PC,HZ,VZ,SCRC,PD,IERR,PTRN)
```

The input parameters are:

- PC — A real array of dimension 3 containing the center point on the projection plane.
- HZ — A real value giving the size of the screen in the horizontal direction.
- VZ — A real value giving the size of the screen in the vertical direction.
- SCRC — A real array of dimension (2,2) containing the limits of the coordinate system on the screen.
- PD — A real array of dimension 3 containing the projection direction.

The output parameters are:

- IERR — An integer giving an error flag. A nonzero value means the transformation could not be computed.
- PTRN — A real array of dimension (3,4) containing the projective transformation.

3.2.5. Subroutine GZ32TR: Transform a Point

This subroutine may be used to transform a point using a three-dimensions to two-dimensions projective transformation. A flag indicates if the projected point is a finite point or a point at infinity.

The calling sequence is:

```
CALL GZ32TR(PTRN,PAS,PAP,FLAG)
```

The input parameters are:

- PTRN — A real array of dimension (3,4) containing the projective transformation.
- PAS — A real array of dimension 3 containing the source point.

The output parameters are:

- PAP — A real array of dimension 2 containing the projected point.
- FLAG — A real value that indicates whether a finite point or a point at infinity has been computed. If this value is nonzero, PAP contains the finite coordinates of the projected point. For a perspective transformation, a positive value indicates the source point is in front of the viewer while a negative value indicates it is behind the viewer. In these cases, the magnitude of FLAG is proportional to the distance from the eye point to

the projected point; it can be used as the projected distance from the eye point to the source point. If this value is zero, PAP is a unit vector pointing in the direction of the point at infinity. If the source point is the eye point of a perspective transformation, both components of PAP and the value of FLAG will be zero.

Chapter 4

Curve Drawing Algorithms

This chapter describes a group of subroutines that may be used to draw smooth curves. The curves are defined by supplying control points and other control information to the subroutines. The curves are drawn by breaking them down into small straight line segments and then calling the GKS polyline subroutine, GPL. The user has control over the number of line segments generated. The mathematical derivation of all of these curve drawing algorithms is given in *An Introduction to the Curves and Surfaces of Computer-Aided Design* [Bea91].

Mathematically, all of these curves are defined *parametrically*, that is, the x and y coordinates are defined as functions of a parameter, t. In effect, a user may specify the parameter at each of the control points. Different assignments of the parameter values at the control points usually results in different curves. There are two schemes that are commonly used to define the values of the parameter associated with the given control points. These two schemes produce curves that are known as *uniform* and *nonuniform* curves. For uniform curves, the parameter is set to zero at the first point and increases by one for each succeeding point. For nonuniform curves, the parameter may be set to any increasing sequence of positive values.

The uniform scheme is very simple mathematically but often does not produce acceptable curves if the points are not nearly equally spaced. A nonuniform scheme that usually produces good results is based on accumulated chord length along the sequence of points. The parameter is set to zero for the first point and increases by an amount equal to the distance between consecutive points for each point. For later reference, we display the increments in the parameter for this nonuniform case

$$\begin{aligned} D_1 &= \text{distance from point 1 to point 2,} \\ D_2 &= \text{distance from point 2 to point 3,} \\ &\cdots \\ D_{N-2} &= \text{distance from point } (N-2) \text{ to point } (N-1), \\ D_{N-1} &= \text{distance from point } (N-1) \text{ to point } N, \end{aligned} \tag{4.1}$$

where N is the number of given control points. The subroutines described in this chapter all start the parameter at zero and expect the user to supply the increments in parameter value, explicitly or implicitly, along the curve.

In the following subroutines, the parameter values are supplied by two arguments; the first, NP, is an integer and the second, PA, is a real array. If NP is positive, the dimension of PA must be NP. The increments in parameter values are

then obtained from the PA array. If more parameter values are needed than are contained in PA, then they are obtained cyclically from PA. That is, the values PA(1), ..., PA(NP) are obtained and then this sequence is repeated. This makes it very easy to specify the uniform curve; NP is simply given an integer value of one while PA is given a real value of one. It is also easy to specify the nonuniform curve with the parameter value based on accumulated chord length. This is done by giving NP a value of zero. In this case, PA is ignored and the subroutine calculates the parameter internally.

Most of the algorithms described here produce curves by using concatenations of simple parametric polynomials. The parametric polynomials are usually of low degree (normally two or three). The points at which consecutive polynomials join are known as *knots*.

In addition to the simple polynomial form of these algorithms, some also have a *rational* form. The rational form consists of x and y being defined as quotients of polynomials. In certain applications, the rational form can be more useful. For example, the only conic the polynomial form can ever match exactly is the parabola. It is impossible for the polynomial form to exactly match a simple circle although it can come arbitrarily close. On the other hand, a rational parametric quadratic can exactly match any conic.

Two distinct types of curves, *interpolation curves* and *design curves*, may be produced by these subroutines. Interpolation curves pass through all of their control points while design curves do not necessarily do this.

The description of each subroutine will include figures showing examples of curves produced by the subroutines. In these figures, the given control points are joined by straight lines between consecutive points. This open polygon is known as the *control polygon*. The reader will notice that these figures do not display the coordinate axes. The reason for this is that all of the curves described here are *isotropic*, that is, they are independent of the coordinate system in which they are defined. In fact, the reader may draw a set of coordinate axes anywhere in these figures and label the axes in any units. The figures also do not label the points so the reader cannot tell which end of the curve corresponds to the first point. The reason for this is that most of these curve drawing algorithms are *symmetric*, that is, they do not depend on which end of the control polygon is the starting end.

If one of these subroutines detects an error in the data supplied to it, the subroutine prints an error message and returns without producing any graphic output.

4.1. Bessel's Method of Local Cubic Interpolation

Bessel's method is a cubic interpolation algorithm. Between each pair of points is a segment of a parametric cubic. Adjacent cubic segments join at the control points and have tangent vectors at those points which have the same direction. The method is also local in that a cubic segment is completely determined by four control points, the ones at its ends and the two on either side of it. In addition to the usual parameter values that are associated with the line segments in the control polygon,

there are additional parameters associated with the tangent vectors at the points. This combination of parameters gives the user a substantial amount of control over the final interpolation curve.

The two subroutines that are described here differ in the type of control that the user has over the ends of the curve. In the first subroutine, the user must supply and extra point beyond the actual ends of the curve. In the second subroutine, the user may specify the tangent direction at the end points or request that the curvature be zero. In this later case, the end conditions may be mixed, that is, the user may specify a tangent vector at one end and request zero curvature at the other.

4.1.1. Subroutine GZBESL: Draw a Parametric Bessel's Curve (I)

This subroutine may be used to draw a curve through a sequence of points using Bessel's method. In this scheme, the ends of the curve are controlled by an extra point. The actual curve, therefore, extends from the second control point to the second point from the end of the curve. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the line segment parameters on accumulated chord length is provided.

The calling sequence is:

```
CALL GZBESL(N,PXA,PYA,NP,PA,NT,TA,NS)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of (N − 1) values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.
NT	An integer giving the number of parameter values associated with tangent vectors at the interior points. This value must be positive and the values are selected cyclically from the next parameter. A total of (N − 2) values are needed.
TA	A real array of dimension NT containing the given parameter values associated with the tangent vectors.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.

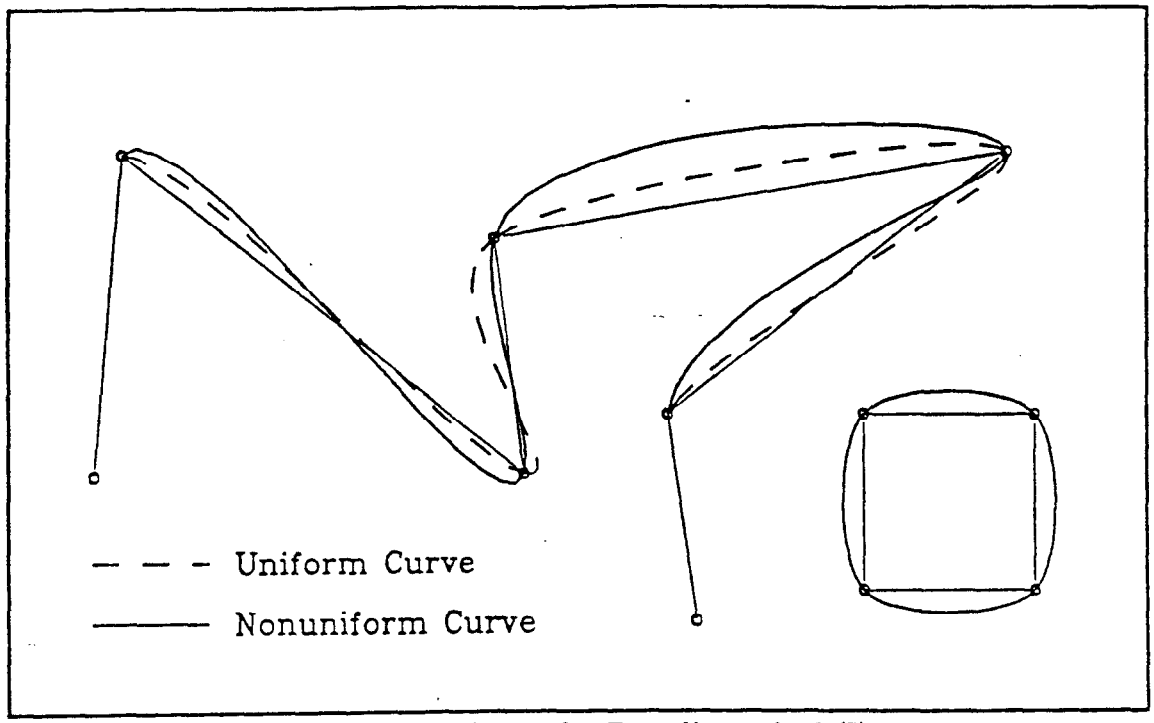

Figure 4.1. Examples of interpolation by Bessel's method (I)

Figure (4.1) includes an example of a nonuniform curve where accumulated chord length has been used as the parameter. The TA values have all been set to one. The circular curve at the lower right of Figure (4.1) was formed by specifying seven points at the corners of the square in sequence. Since the chord segments are equal, the uniform and nonuniform curves based on accumulated chord length are identical.

Figure (4.2) illustrates the affect the PA values have on the curve. The figure illustrates the manipulation the PA value associated with the central line segment of the control polygon. It shows that reducing the value of PA(3) causes the curve to move closer to the chord between the third and fourth points. In this case, the tangent vectors at the third and fourth points also rotate to become closer to the chord. Large values of PA(3) cause the curve to move away from the chord and a cusp or loop can form if it is made too large. Figure (4.2) also illustrates the local properties of the interpolation because all three composite curves are tangent to each other at their ends; any continuation of the curve beyond its current ends will not be affected by the change in the parameter.

Figure (4.3) illustrates the manipulation of the TA values. The natural value of the TA values is one. As TA(2) is reduced, the influence of the tangent vector at the middle point is reduced and the curve pulls away from the tangent vector and approaches the adjacent chords. However, in this case, the tangent direction at the middle point does not change. If TA(2) is made large, the influence of the tangent vector at the middle point becomes strong. This forces the interpolation curve to flatten and follow the direction of the tangent vector longer. In general, when the

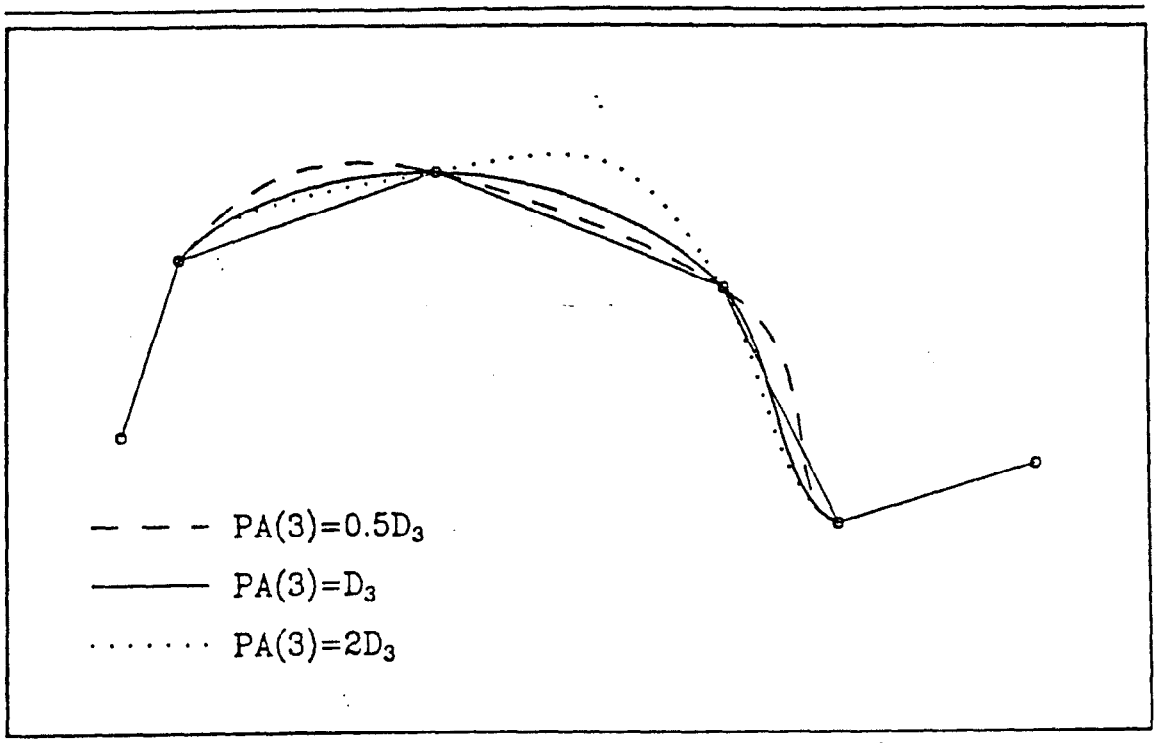

Figure 4.2. Examples of interpolation by Bessel's method (II)

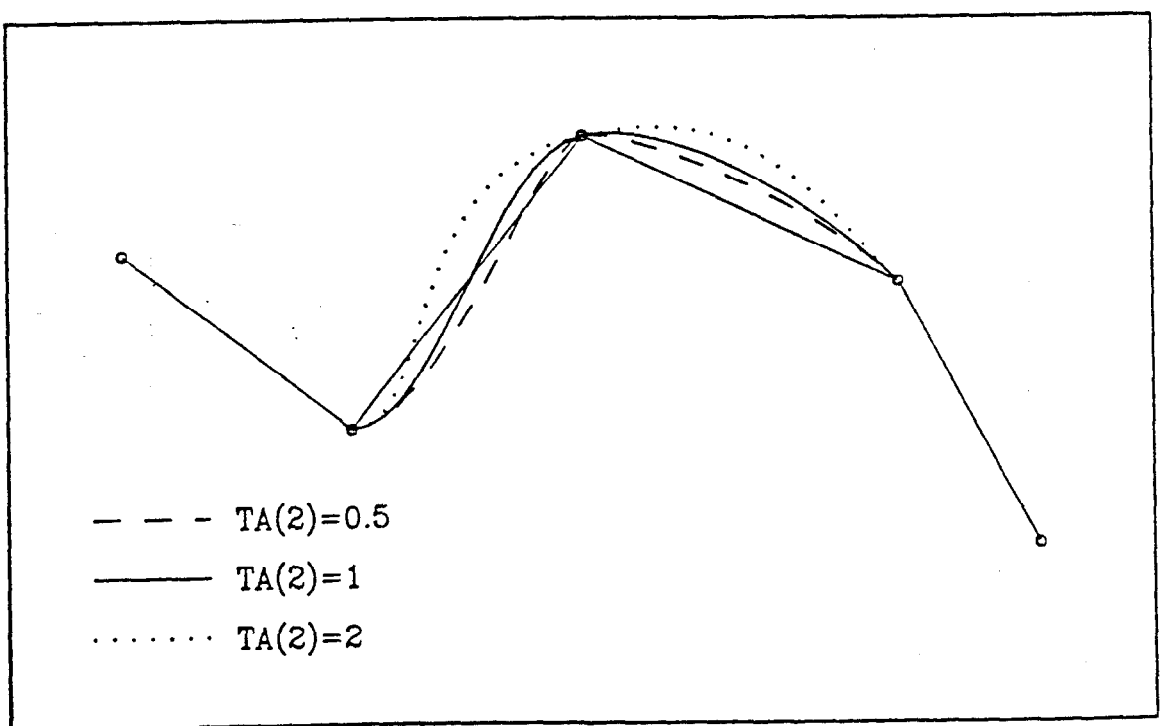

Figure 4.3. Examples of interpolation by Bessel's method (III)

TA values are reduced, the curve moves closer to the adjacent chords and becomes taut; increasing the TA values allows the curve to relax and bow out.

4.1.2. Subroutine GZBESE: Draw a Parametric Bessel's Curve (II)

This subroutine may be used to draw a curve through a sequence of points using Bessel's method. In this scheme, the ends of the curve are controlled by specifying the end tangents or by requesting zero curvature at the ends. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the line segment parameters on accumulated chord length is provided.

The calling sequence is:

```
CALL GZBESE(N,PXA,PYA,V1,V2,NP,PA,NT,TA,NS)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
V1	A real array of dimension 2 containing the given tangent vector at the initial end. This argument should usually be a unit vector or a zero vector. If it is a zero vector, then zero curvature is imposed at the end.
V2	A real array of dimension 2 containing the given tangent vector at the terminal end. This argument should usually be a unit vector or a zero vector. If it is a zero vector, then zero curvature is imposed at the end.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of (N − 1) values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.
NT	An integer giving the number of parameter values associated with tangent vectors at the points. This value must be positive and the values are selected cyclically from the next parameter. A total of N values are needed.
TA	A real array of dimension NT containing the given parameter values associated with the tangent vectors.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.

Figure (4.4) shows examples of interpolation by Bessel's method when tangents at the ends of the curve are supplied. In this case the curve is not, strictly speaking, symmetric. Since the tangents at the ends are supplied, they must point in the direction of the curve so this curve was drawn from the left to the right. To draw the curve in the other direction, the directions of the tangent vectors must be

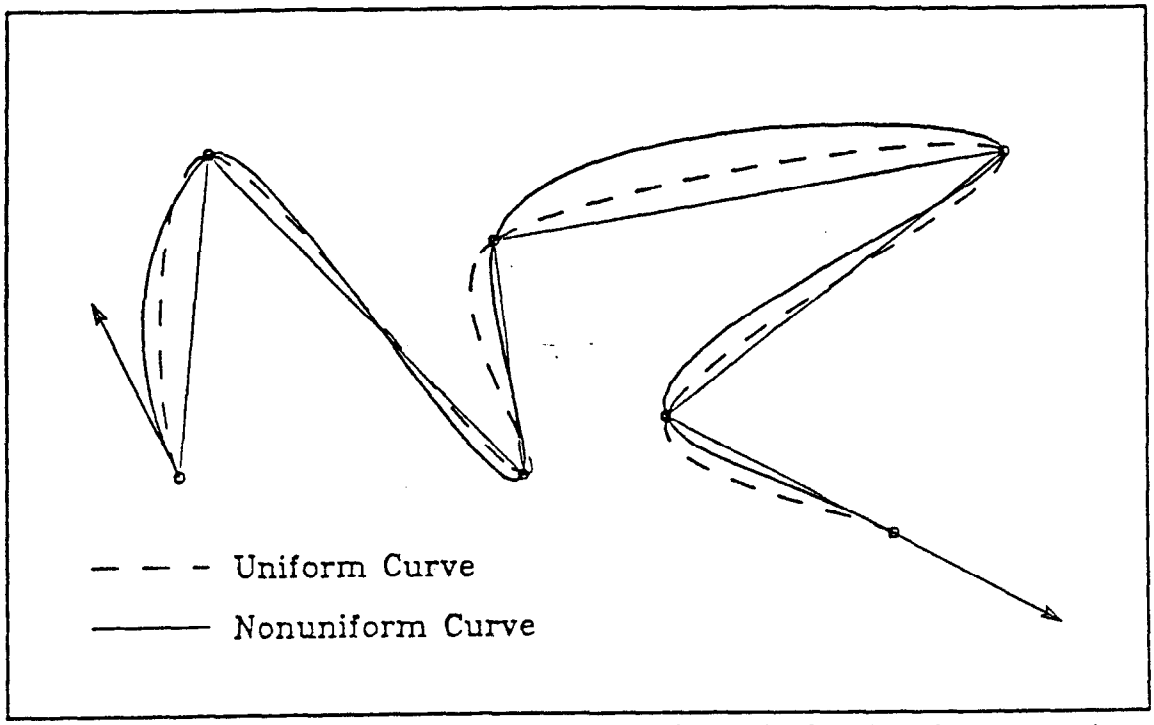

Figure 4.4. Examples of interpolation by Bessel's method with end tangents given

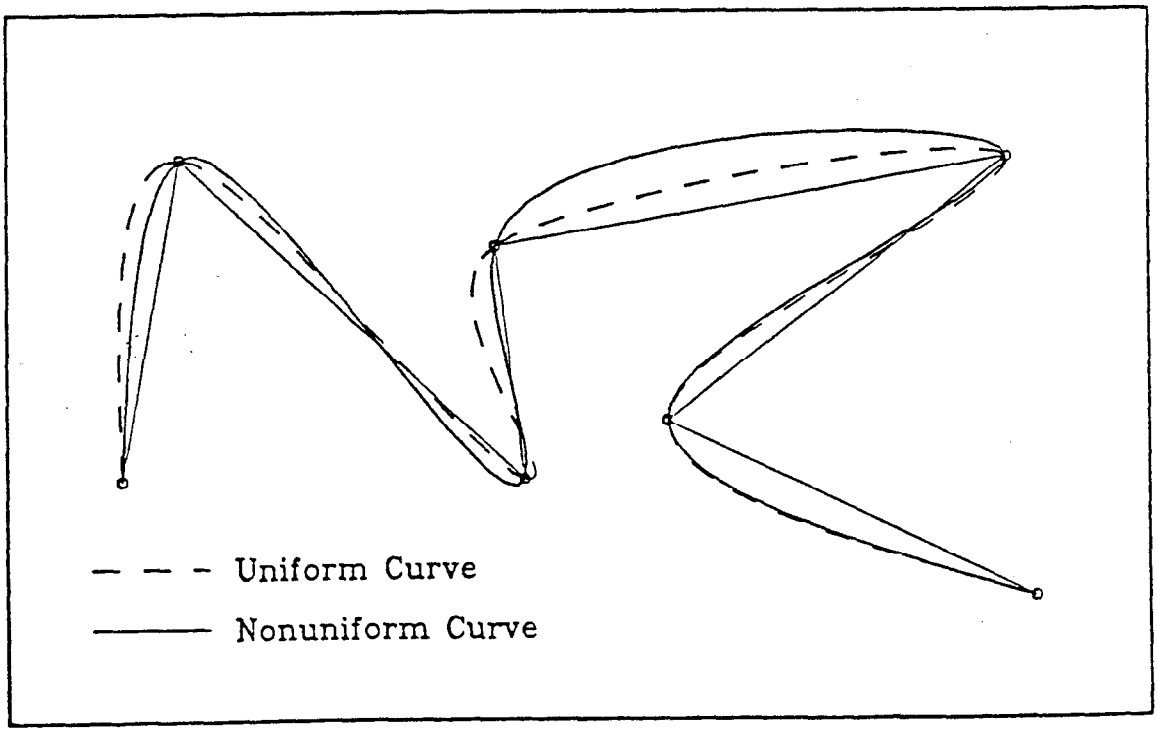

Figure 4.5. Examples of interpolation by Bessel's method with zero curvature at the ends

reversed. In the nonuniform curve, accumulated chord length has been used as the

parameter.

In Figure (4.5) the curvature at the end points has been constrained to be zero. The nonuniform curve again has accumulated chord length as its parameter.

4.2. Cubic Spline Interpolation

This section describes a subroutine that may be used to draw a parametric cubic spline. A cubic spline is an interpolation curve consisting of parametric cubic polynomial segments. The segments of the curve join at the knots with equal first and second derivatives. However, the curve is not local in nature; changing one control point modifies the entire curve.

There is a limit on the number of control points that may be supplied to this subroutine.

4.2.1. Subroutine GZSPLN: Draw a Parametric Cubic Spline

This subroutine may be used to draw a parametric cubic spline curve through a sequence of points. The ends of the curve are controlled by specifying the end tangents or by requesting zero curvature at the ends. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the parameter on accumulated chord length is provided.

The calling sequence is:

```
CALL GZSPLN(N,PXA,PYA,V1,V2,NP,PA,NS)
```

The input parameters are:

N	An integer giving the number of control points. The maximum number of points that are allowed is 32.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
V1	A real array of dimension 2 containing the given tangent vector at the initial end. This argument should usually be a unit vector or a zero vector. If it is a zero vector, then zero curvature is imposed at the end.
V2	A real array of dimension 2 containing the given tangent vector at the terminal end. This argument should usually be a unit vector or a zero vector. If it is a zero vector, then zero curvature is imposed at the end.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of (N − 1) values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.

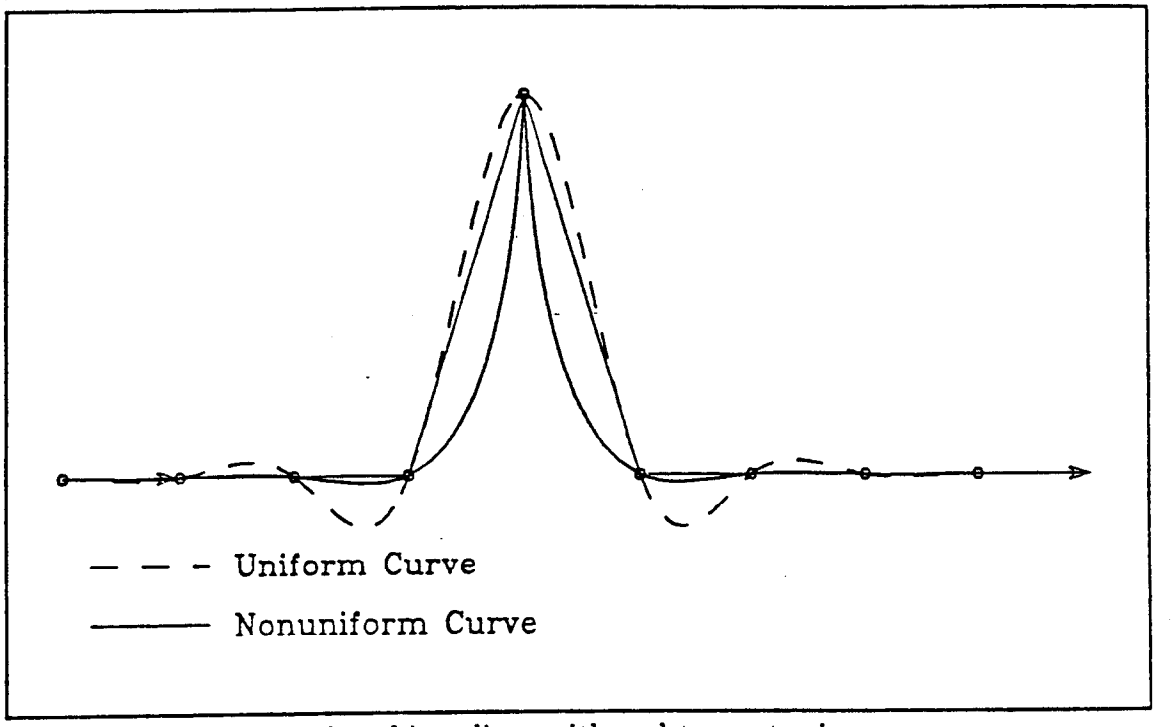

Figure 4.6. Parametric cubic splines with end tangents given

NS An integer giving the number of straight line segments into which each curve segment is to be divided.

Figure (4.6) shows examples of cubic splines with the tangents given at the end points. In this case the curve is not, strictly speaking, symmetric. Since the tangents at the ends are supplied, they must point in the direction of the curve so this curve was drawn from the left to the right. To draw the curve in the other direction, the directions of the tangent vectors must be reversed. The figure also shows the oscillatory behavior that is often a problem in spline curves.

Figures (4.7) and (4.8) were drawn with zero curvature at the end points. In Figure (4.7), the spacing of the points was deliberately chosen to have large variation in the chord lengths. As a result, the uniform curve exhibits oscillatory problems at the top center of the figure. Figure (4.8) illustrates how the PA values can be used to control the shape of the curve. In this case, chord lengths have been used for the parameters except that the PA value associated with the central line segment of the control polygon has been manipulated. As we have seen before, reducing a PA value causes the curve to move closer to the associated line segment. Figure (4.8) also shows that changes like these are not local; they affect the entire curve.

4.3. Bézier Curves

A Bézier curve is a design curve and not an interpolation curve. It does, however, pass through its first and last control points and is tangent to the first and last

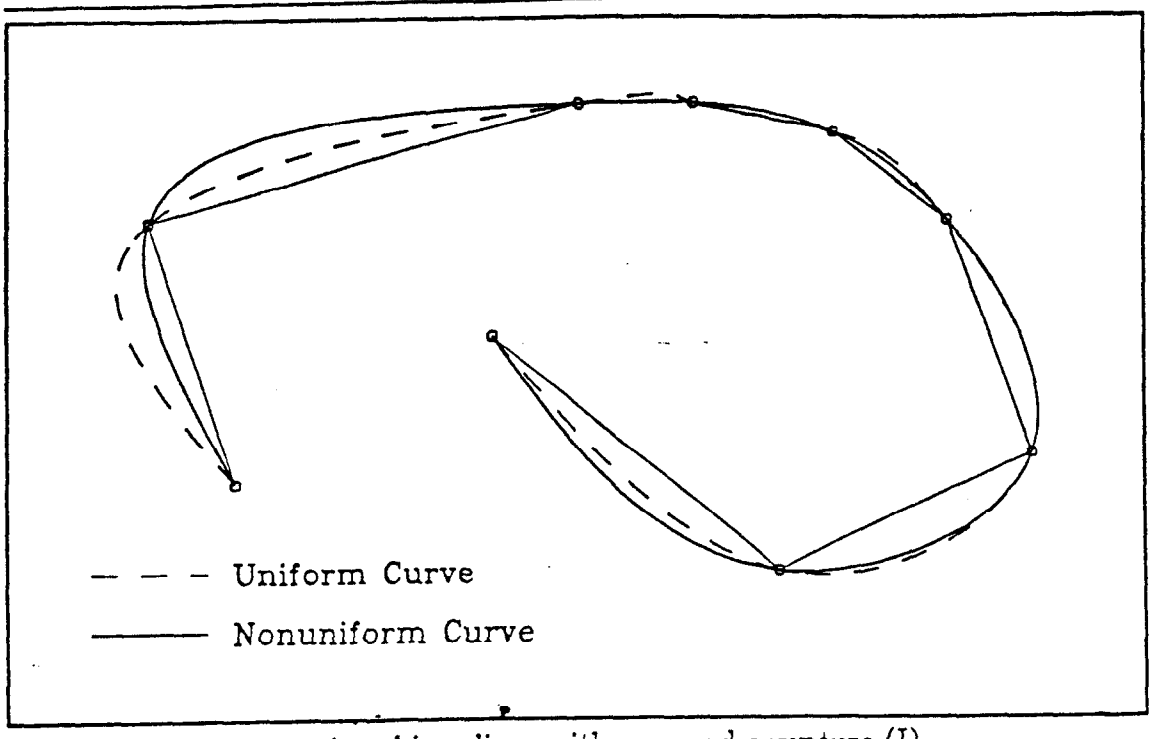

Figure 4.7. Parametric cubic splines with zero end curvature (I)

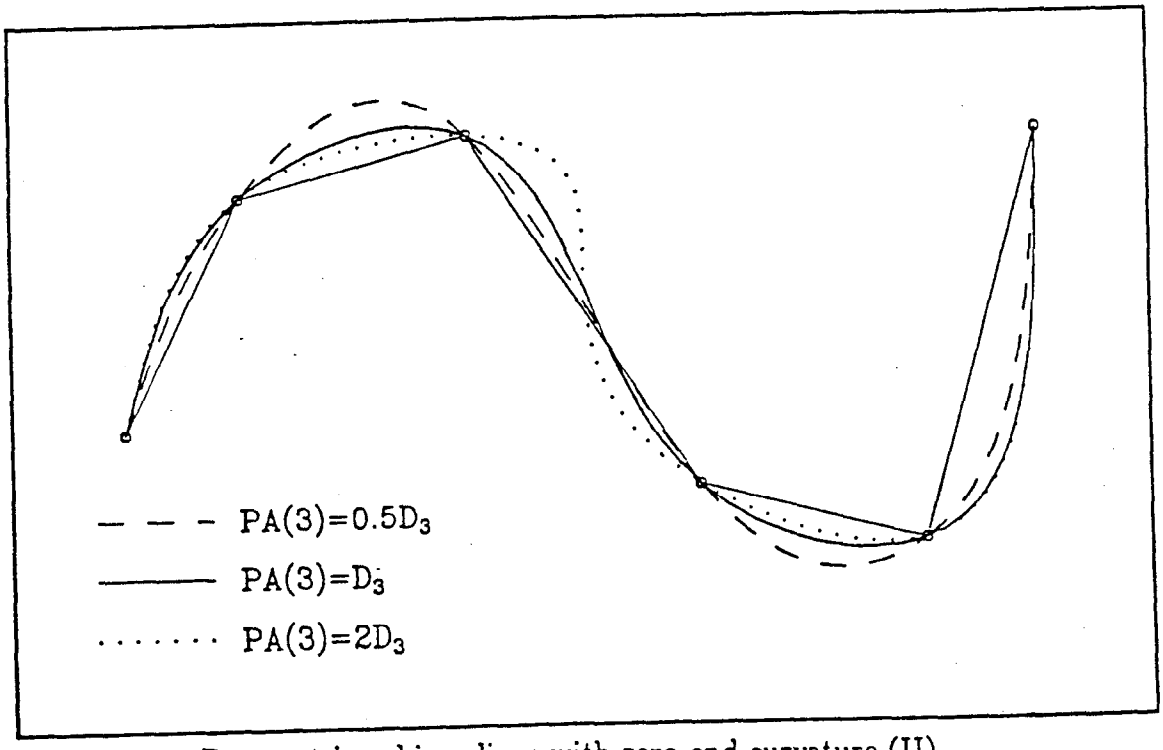

Figure 4.8. Parametric cubic splines with zero end curvature (II)

straight line segment in the control polygon. The Bézier curve is a parametric polynomial of large degree (in fact the degree is the number of control points minus

one). Although using polynomials of large degree is usually a dangerous thing to do, the Bézier curve is unusually well behaved.

The Bézier curve is available in both a simple polynomial and a rational form. The polynomial form does not have any user control except for the positioning of the control points. The rational form has control variables called *weights*. The weights may be any positive values. If the weights are all equal, the polynomial form of the Bézier curve is produced.

There is a limit on the number of control points that may be supplied to these subroutines.

4.3.1. Subroutine GZBEZR: Draw a Bézier Curve

This subroutine may be used to draw a Bézier curve of arbitrary degree determined by a sequence of points.

The calling sequence is:

```
CALL GZBEZR(N,PXA,PYA,NS)
```

The input parameters are:

N	An integer giving the number of control points. The maximum number of points that are allowed is 32.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NS	An integer giving the number of straight line segments into which the curve is to be divided.

Figures (4.9) and (4.10) show some examples of Bézier curves. Figure (4.10) illustrates the effect of moving a single control point.

4.3.2. Subroutine GZRBEZ: Draw a Rational Bézier Curve

This subroutine may be used to draw a rational Bézier curve of arbitrary degree determined by a sequence of points.

The calling sequence is:

```
CALL GZRBEZ(N,PXA,PYA,NW,WA,NS)
```

The input parameters are:

N	An integer giving the number of control points. The maximum number of points that are allowed is 32.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.

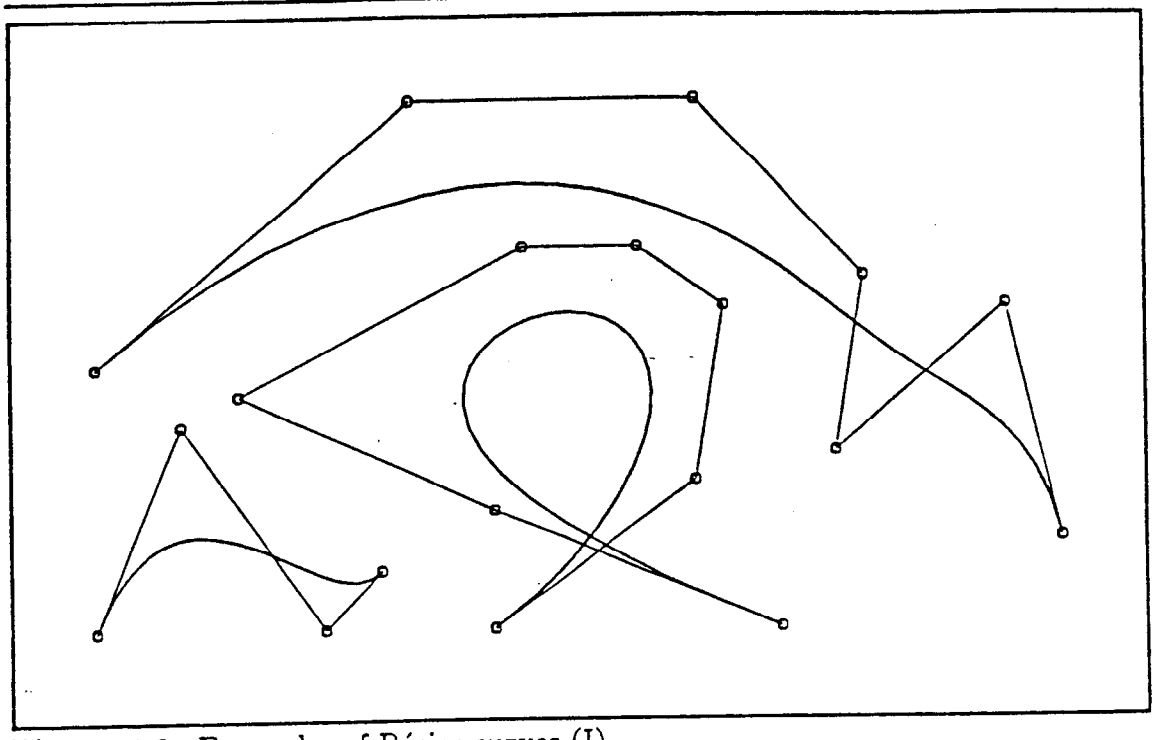

Figure 4.9. Examples of Bézier curves (I)

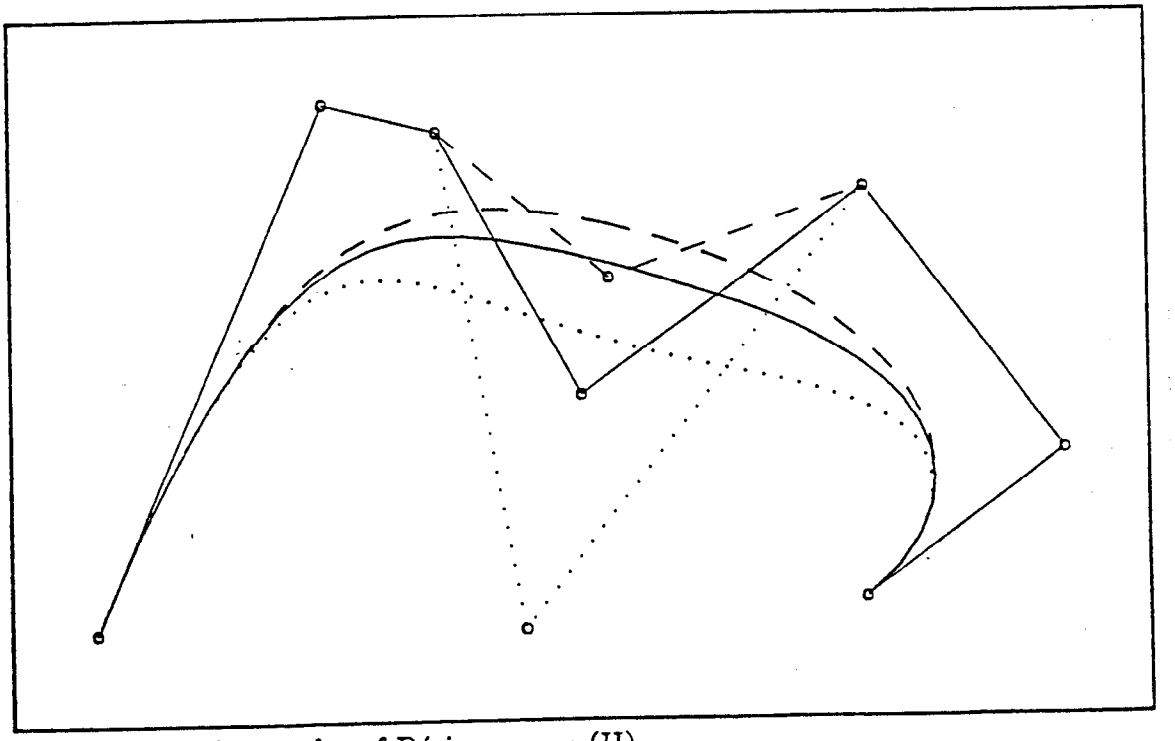

Figure 4.10. Examples of Bézier curves (II)

NW An integer giving the number of weights associated with the control points. This value must be positive and the weights are selected cycli-

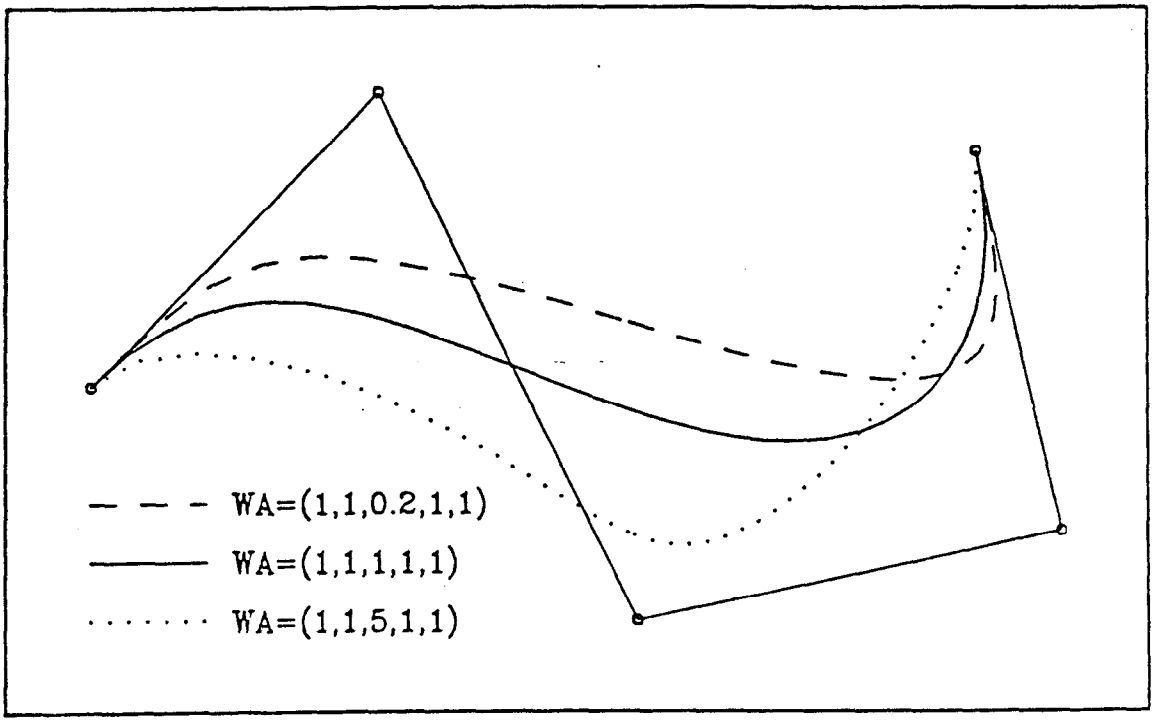

Figure 4.11. Examples of rational Bézier curves

cally from the next parameter. A total of N values are needed.

WA — A real array of dimension NW containing the given weights.

NS — An integer giving the number of straight line segments into which the curve is to be divided.

Figure (4.11) illustrates how the weights may be used to control the shape of a rational Bézier curve. In the figure, larger values of the weights cause the curve to move closer to its associated point while allowing the curve to pull away from neighboring points.

4.4. B-spline Curves

A B-spline curve is a pure design curve; it normally does not pass through any of its control points. The subroutines described here make the B-spline available in both the polynomial and rational forms in either quadratic or cubic degree. The segments of a quadratic B-spline match at the knots in ordinate and first derivative. The segments of a cubic B-spline match in ordinate, and first and second derivative. The curve also is local in nature; changing a single control point only affects a small number of curve segments.

Since the knots would not otherwise be known to the user, a facility is provided whereby the knots my be marked. This is done by calling the GKS polymarker subroutine, GPM. All of the figures in this section have had the knots marked with markers that are slightly smaller than those used for the control points.

The B-spline is actually a generalization of the Bézier curve. The proper selection of the parameter values can cause the subroutines described in this section to produce a Bézier curve.

4.4.1. Subroutine GZBSP2: Draw a Quadratic B-spline Curve

This subroutine may be used to draw a quadratic B-spline curve that is controlled by a sequence of points. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the parameter on accumulated chord length is provided. In addition to drawing the curve, the knots may also be marked.

The calling sequence is:

```
CALL GZBSP2(N,PXA,PYA,NP,PA,NS,MFLG)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of N values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.
MFLG	An integer flag that indicates if the knots are to be marked. Any nonzero value will cause them to be marked.

There is, however, a problem with the generation of the PA array when accumulated chord length is used to produce it. The problem is that there are (N − 1) distances available but N values are needed. An appropriate scheme, and the one used within subroutine GZBSP2, is

$$\begin{aligned} \texttt{PA}(1) &= D_1, \\ \texttt{PA}(2) &= \tfrac{1}{2}(D_1 + D_2), \\ \texttt{PA}(3) &= \tfrac{1}{2}(D_2 + D_3), \\ &\cdots \\ \texttt{PA}(\texttt{N}-1) &= \tfrac{1}{2}(D_{N-2} + D_{N-1}), \\ \texttt{PA}(\texttt{N}) &= D_{N-1}. \end{aligned} \tag{4.2}$$

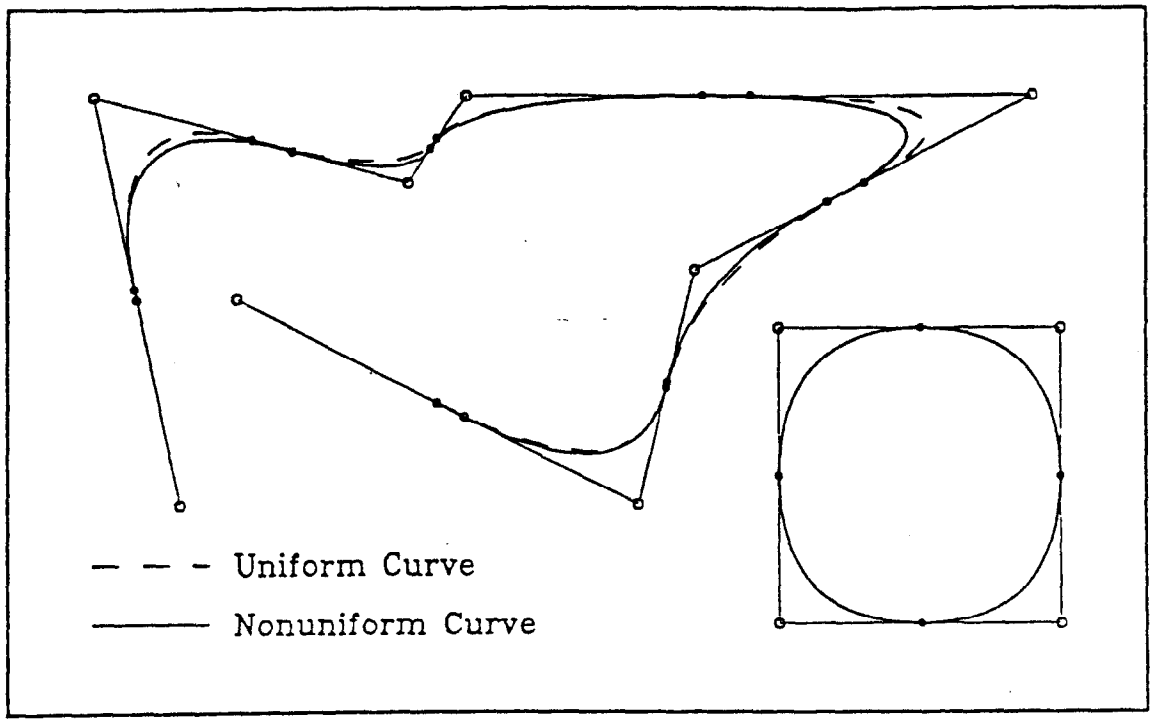

Figure 4.12. Examples of quadratic B-splines (I)

The D_i values are determined by Equations (4.1).

Figure (4.12) shows examples of uniform and nonuniform quadratic B-splines. The nearly circular curve at the lower right was formed by specifying six consecutive corner points around the square. The uniform and nonuniform curve based on chord length are equal in this case. In the other nonuniform curve, the PA values were determined from chord distances and Equations (4.2).

For the quadratic B-spline, the knots always lie on the control polygon and the curve is tangent to the control polygon at the knots. In the uniform case, the knots are at the midpoints of the line segments in the control polygon.

Figure (4.13) shows examples of how a modification of the PA values changes the curve. In this case, the PA's were also determined from Equations (4.2) and only the central one was modified. Notice how small values of this parameter cause the points of tangency on the control polygon to move closer to the associated point on the control polygon.

There is a fairly popular alternative to Equations (4.2). That alternative sets

$$\begin{aligned} \texttt{PA(1)} &= 0.0, \\ \texttt{PA(N)} &= 0.0, \end{aligned}$$

with the other values set by Equations (4.2). The advantage of this scheme is that the curve now passes through the first and last control points and is tangent to the control polygon at those points. The interior of the curve has the properties described above. The problem with this formulation is that it does not reduce to the usual uniform approach.

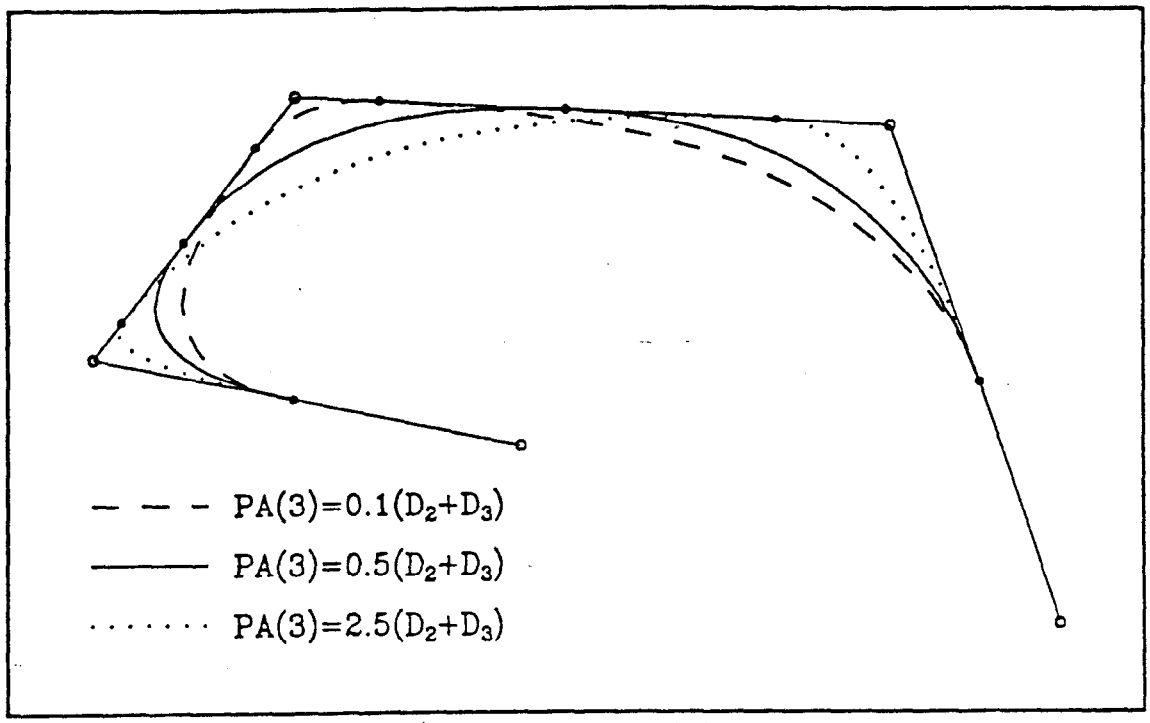

Figure 4.13. Examples of quadratic B-splines (II)

4.4.2. Subroutine GZRBS2: Draw a Rational Quadratic B-spline Curve

This subroutine may be used to draw a rational quadratic B-spline curve that is controlled by a sequence of points. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the line segment parameters on accumulated chord length is provided. In addition to drawing the curve, the knots may also be marked.

The calling sequence is:

```
CALL GZRBS2(N,PXA,PYA,NP,PA,NW,WA,NS,MFLG)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of N values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.

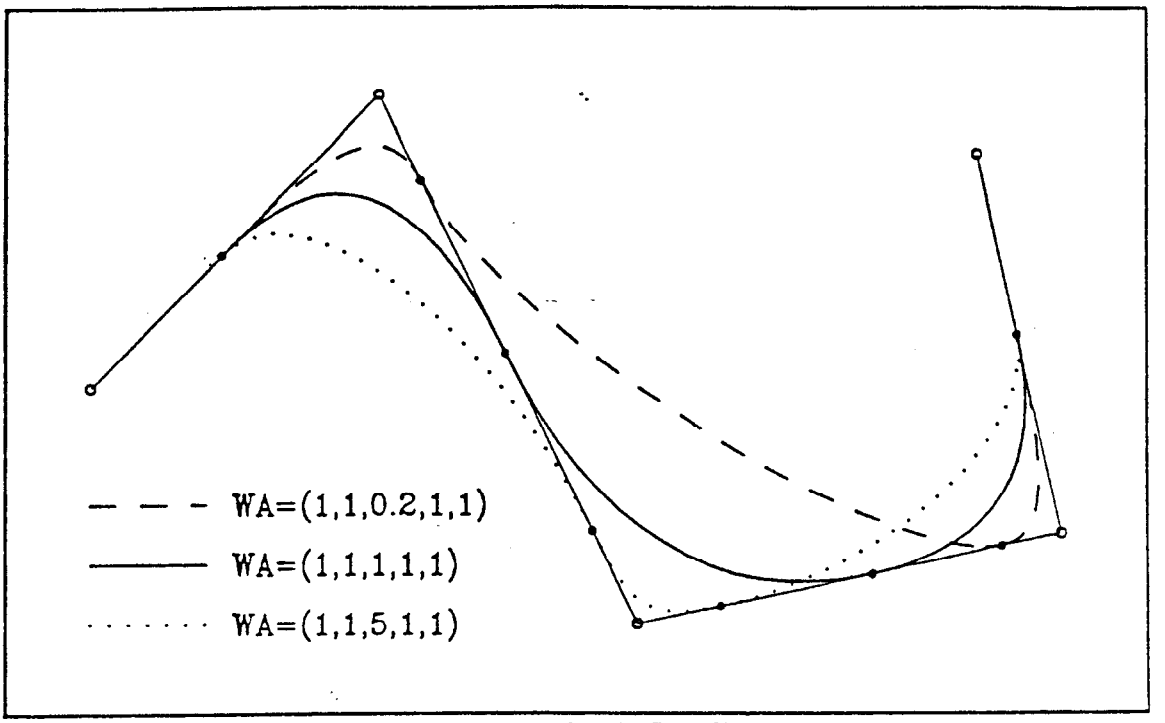

Figure 4.14. Examples of rational quadratic B-spline curves

NW	An integer giving the number of weights associated with the control points. This value must be positive and the weights are selected cyclically from the next parameter. A total of N values are needed.
WA	A real array of dimension NW containing the given weights.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.
MFLG	An integer flag that indicates if the knots are to be marked. Any nonzero value will cause them to be marked.

Figure (4.14) illustrates how the weights may be used to control the shape of a rational quadratic B-spline curve. The PA values were determined by Equations (4.2). In the figure, larger values of the weights cause the curve to move closer to its associated point while allowing the curve to pull away from neighboring points.

4.4.3. Subroutine GZBSP3: Draw a Cubic B-spline Curve

This subroutine may be used to draw a cubic B-spline curve that is controlled by a sequence of points. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the parameter on accumulated chord length is provided. In addition to drawing the curve, the knots may also be marked.

The calling sequence is:

```
CALL GZBSP3(N,PXA,PYA,NP,PA,NS,MFLG)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of (N + 1) values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.
MFLG	An integer flag that indicates if the knots are to be marked. Any nonzero value will cause them to be marked.

There is again a problem with the PA array when accumulated chord length is used to produce it. In this case there are (N − 1) distances available but (N + 1) values are needed. An appropriate scheme, and the one used within subroutine GZBSP3, is

$$\begin{aligned} \texttt{PA}(1) &= D_1, \\ \texttt{PA}(2) &= D_1, \\ \texttt{PA}(3) &= D_2, \\ &\cdots \\ \texttt{PA}(\texttt{N}-1) &= D_{\texttt{N}-2}, \\ \texttt{PA}(\texttt{N}) &= D_{\texttt{N}-1}, \\ \texttt{PA}(\texttt{N}+1) &= D_{\texttt{N}-1}. \end{aligned} \tag{4.3}$$

The D_i values are again determined by Equations (4.1).

Figure (4.15) shows examples of uniform and nonuniform cubic B-splines. The nearly circular curve at the lower right was formed by specifying seven consecutive corner points around the square. The uniform and nonuniform curve based on chord length are equal in this case. In the other nonuniform curve, the PA values were determined from chord distances and Equations (4.3).

Figure (4.16) shows examples of how a modification of the PA values changes the curve. In this case, the PA's were also determined from Equations (4.3) and only the central one was modified. Small values of this parameter cause the central curve segment to shrink and move closer to the associated segment of the control polygon.

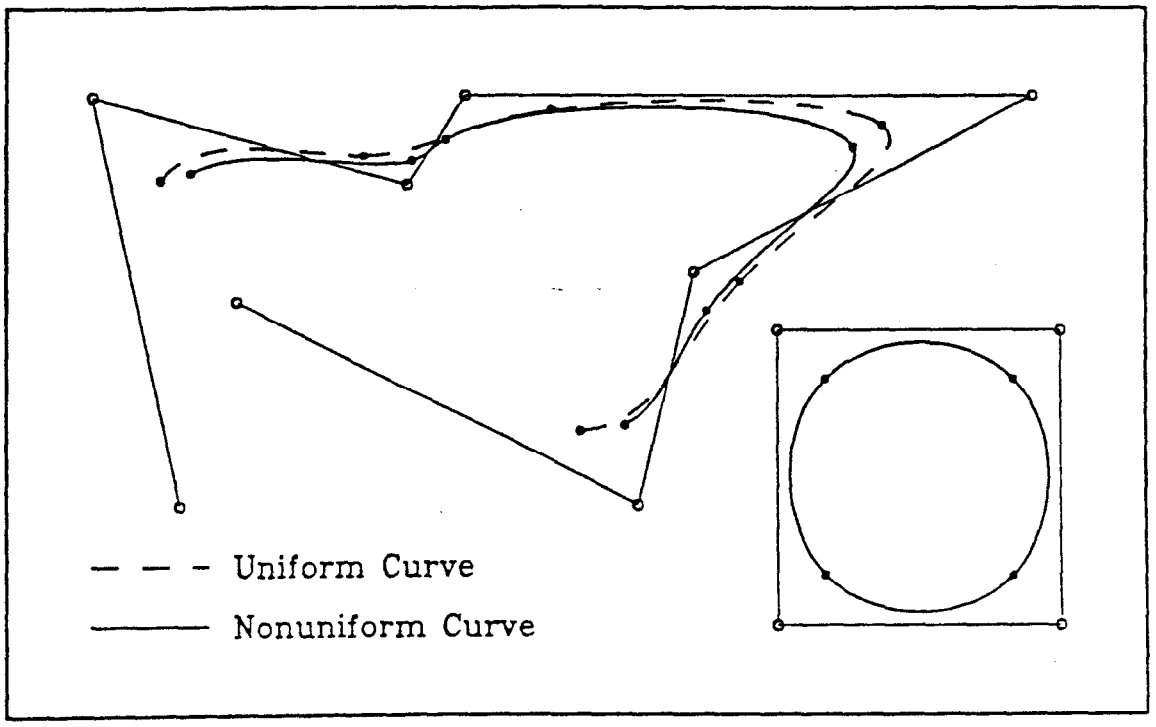

Figure 4.15. Examples of cubic B-splines (I)

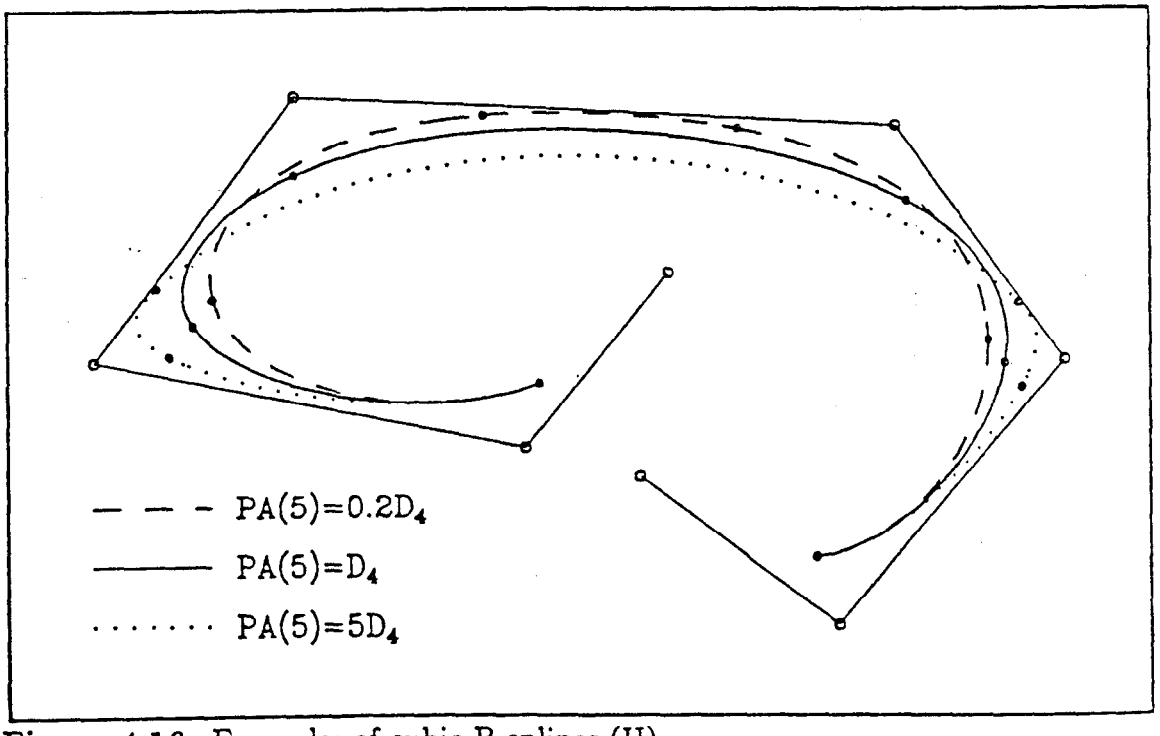

Figure 4.16. Examples of cubic B-splines (II)

As in the quadratic case, there is a popular alternative to Equations (4.3). That

alternative sets

$$\begin{aligned} \texttt{PA}(1) &= 0.0, \\ \texttt{PA}(2) &= 0.0, \\ \texttt{PA}(\texttt{N}) &= 0.0, \\ \texttt{PA}(\texttt{N}+1) &= 0.0. \end{aligned}$$

This scheme again forces the curve to pass through the first and last control points and makes it tangent to the control polygon at those points.

4.4.4. Subroutine GZRBS3: Draw a Rational Cubic B-spline Curve

This subroutine may be used to draw a rational cubic B-spline curve that is controlled by a sequence of points. Either a uniform curve, or a nonuniform curve may be drawn. In the case of a nonuniform curve, a simple means to base the parameter on accumulated chord length is provided. In addition to drawing the curve, the knots may also be marked.

The calling sequence is:

```
CALL GZRBS3(N,PXA,PYA,NP,PA,NW,WA,NS,MFLG)
```

The input parameters are:

N	An integer giving the number of control points.
PXA	A real array of dimension N containing the x coordinates of the control points.
PYA	A real array of dimension N containing the y coordinates of the control points.
NP	An integer giving the number of parameter values associated with line segments in the control polygon. If this value is not positive, accumulated chord length will be used to generate the parameter. If this parameter is positive, values are selected cyclically from the next parameter. In this case, a total of (N + 1) values are needed.
PA	If NP is positive, this is a real array of dimension NP containing the given parameter values associated with the line segments.
NW	An integer giving the number of weights associated with the control points. This value must be positive and the weights are selected cyclically from the next parameter. A total of N values are needed.
WA	A real array of dimension NW containing the given weights.
NS	An integer giving the number of straight line segments into which each curve segment is to be divided.
MFLG	An integer flag that indicates if the knots are to be marked. Any nonzero value will cause them to be marked.

Figure (4.17) illustrates how the weights may be used to control the shape of a rational cubic B-spline curve. The PA values were determined by Equations (4.3).

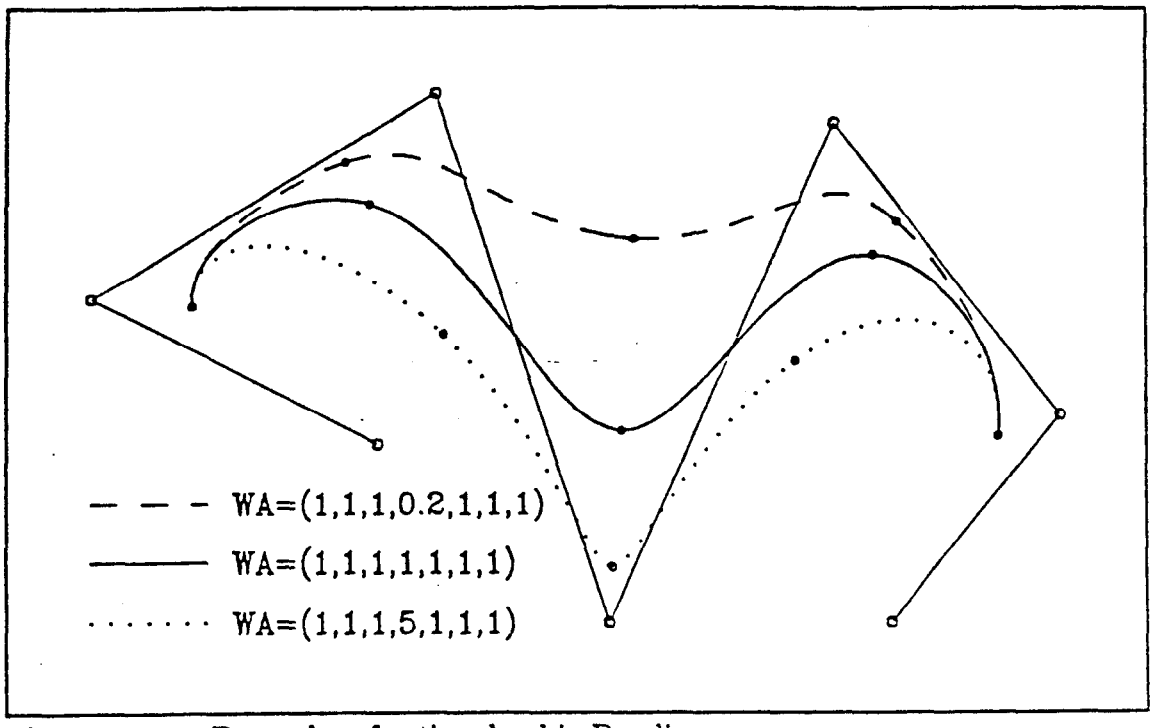

Figure 4.17. Examples of rational cubic B-spline curves

In the figure, larger values of the weights cause the curve to move closer to its associated point while allowing the curve to pull away from neighboring points.

Chapter 5

Surface Drawing Algorithms

This chapter describes a group of subroutines that may be used to draw pictures of the surfaces of solid objects. The surfaces are defined by supplying control points and other control information to the subroutines. The surfaces are drawn by breaking them down into simple polygons, eliminating those polygons that face away from the viewer, sorting the remainder so that the ones farthest away are first on the list, and then calling the GKS fill area subroutine, GFA, to write the polygons to the active workstations in the sorted order. Closer polygons, therefore, overlay the farther ones.

This method is quite fast but does have its problems. It is possible, especially when polygons of vastly differing sizes are involved, to have a large polygon determined to be "closer" than a small one even though the small one actually hides part of the larger. The subroutine in this chapter that deals with generalized polyhedral solids is especially vulnerable to this problem, particularly if non-convex polygons are supplied. It is also important that the polygons do not intersect each other; none of the algorithms described here can handle that problem.

There are a number of ways that the polygons may be drawn so that useful pictures are produced. In the simplest method, the polygons are drawn as fill areas, usually in the background color, and then outlined by a polyline. When this is done the pictures look like line drawn figures. A second way is to apply a light source and reflection model to obtain fairly realistic pictures. This method will only be successful on workstations that can produce a large number of colors. If the workstation only supports a small number of colors, the fill areas will all blend together and the picture will be unintelligible. In addition to these two general modes, some algorithms will supply other options.

To understand the light source and reflection model used in these subroutines, consider Figure (5.1). This figure shows a point, **P**, on the surface and the light source and eye point. **N** is the surface normal at **P**, **L** is a vector pointing from **P** to the light source, and **E** is a vector pointing toward the eye position. **R** represents a light ray that starts at the light source and reflects off the surface. The vectors **L**, **N**, and **R** are coplanar and **L** and **R** make the same angle, θ, with **N**. The vector **E** makes an angle of α with **R**. Notice that **E** is not necessarily coplanar with **L**, **N**, and **R**.

The light source and reflection model used is

$$I = I_0 + \frac{K_d \cos\theta + K_s \cos^n \alpha}{d + K}$$

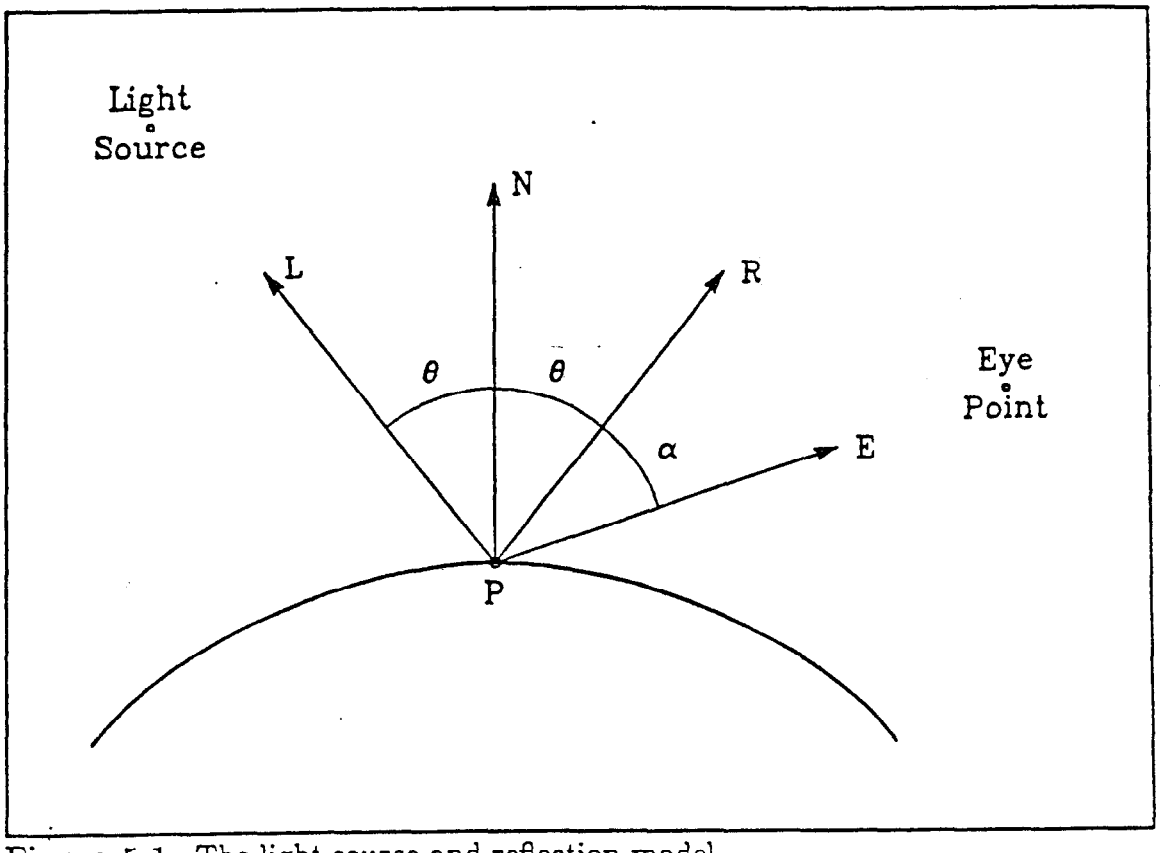

Figure 5.1. The light source and reflection model

where I_0, n, K_d, K_s, and K are parameters that the user may set. The computed value, I, is known as the *shading function* for the point. The value d is the projected distance from the eye point to the surface; its value is one on the projection plane and zero at the eye point. K is a distance adjustment constant. I_0 is the ambient illumination for the scene. K_d is the diffuse reflection constant and K_s is the specular reflection constant. Highlights on a shiny object are caused by large values of the specular reflection constant. Large values of n also cause an object to appear shiny. A complete derivation of the model is given in Section 5.2 of *Procedural Elements for Computer Graphics* [Rog85].

The computed shading function for a polygonal surface is mapped to a sequence of GKS color indices or attribute bundle indices. The calling program must set up this sequence of indices and supply the subroutine with the smallest and largest index and the value of the shading function that it corresponds to. Linear interpolation is used for intermediate values. Values of the shading function outside the given limits are mapped to the appropriate extreme index.

The parameters of the light source and reflection model are given in a real array, `CCA`, that is common to all of the subroutines. The description of the array in this case is as follows:

`CCA(1)` To specify the light source and reflection model, this value should contain a real value of one.

`CCA(2)` A real value giving the x component of a vector in the direction of the light rays. That is, the direction is from the light source toward the object.

`CCA(3)` A real value giving the y component of a vector in the direction of the light rays.

`CCA(4)` A real value giving the z component of a vector in the direction of the light rays.

`CCA(4)` A real value giving the ambient illumination, I_0.

`CCA(6)` A real value giving the specular reflection exponent, n.

`CCA(7)` A real value giving the diffuse reflection constant, K_d.

`CCA(8)` A real value giving the specular reflection constant, K_s.

`CCA(9)` A real value giving the distance adjustment constant, K.

`CCA(10)` A real value of zero indicates that the indices given below are color indices. A real value of one indicates that the indices given below are attribute bundle indices.

`CCA(11)` A real value giving the minimum value of the shading function corresponding to the next index.

`CCA(12)` A real value giving the color index or the attribute bundle index to be used to draw the color associated with the minimum value of the shading function. This value will be converted to an integer before it is used.

`CCA(13)` A real value giving the maximum value of the shading function corresponding to the next index.

`CCA(14)` A real value giving the color index or the attribute bundle index to be used to draw the color associated with the maximum value of the shading function. This value will be converted to an integer before it is used.

Notice that the direction of the light rays as given by `CCA(2)`, ..., `CCA(4)` is the reverse of that shown by the vector **L** in Figure (5.1). It is important to get the direction correct; it is no help if the light is shining on the bottom of the model when you expected it on the top. The values of these parameters can be difficult to select. In the absence of other information, a good place to start is $I_0 = 0.1$, $n = 2.0$, $K_d = 1.5$, $K_s = 0.3$, and $K = 1.0$.

Each of the subroutines also needs a work array. This is a real array that is used to sort the polygons. The required size depends on the problem but a maximum value is usually easy to obtain.

From the above discussion, it is apparent the there are two things that are difficult to determine when these subroutines are used. The first of these problems is the coefficients of the shading function and its extreme values. The second is the size of the work array. To aid in the use of these subroutines, the maximum and minimum computed values of the shading function and the actual size of the work array that was needed are made available to the user. These results are put into a `COMMON` block whose declaration is

```
C  COMMON BLOCK TO RETURN SURFACE INFORMATION.
```

```
      SAVE            /GZSINF/
      COMMON          /GZSINF/GZLMAX,GZSMIN,GZSMAX
C     MAXIMUM LENGTH OF THE WORK AREA THAT WAS USED.
      INTEGER         GZLMAX
C     MINIMUM AND MAXIMUM VALUES OF THE SHADING FUNCTION.
      REAL            GZSMIN,GZSMAX
```

The COMMON block is available after one of these subroutines has been called. If the light source and reflection model was not used, GZSMIN and GZSMAX will be zero.

The view of the surface is selected by specifying a three-dimensions to two-dimensions projective transformation. That transformation must be a perspective transformation; it cannot be a parallel transformation.

These subroutines are quite efficient when processing on the host computer only is considered. However, the amount of data that must be transmitted to the workstation can be quite large and many workstations require substantial amounts of time to process fill areas. In essence, these subroutines off-load much of the computation from the host computer to the graphic device itself.

If one of these subroutines detects an error in the data supplied to it, the subroutine prints an error message and returns, usually without producing any graphic output.

5.1. Two-dimensional Histograms

A two-dimensional histogram consists of a rectangular array of rectangular columns sitting on a common base. The height of the columns can be used to represent experimental or synthetic data. Pictures of this type are sometimes called *Lego* plots.

5.1.1. Subroutine GZ2DHG: Draw a Two-Dimensional Histogram

This subroutine may be used to draw a two-dimensional histogram.

The polygons that constitute the histogram may be drawn in one of three ways. In the first scheme, corresponding sides on each column are drawn in a distinct color. In the second scheme, the existing GKS settings are used to draw the polygons as fill areas and then outline the polygons using a polyline. The third scheme provides a light source and reflection model to color the polygons.

The calling sequence is:

```
CALL GZ2DHG(M,N,PXYZA,PTRN,CCA,L,WA)
```

The input parameters are:

M	An integer giving the first dimension of PXYZA.
N	An integer giving the second dimension of PXYZA.

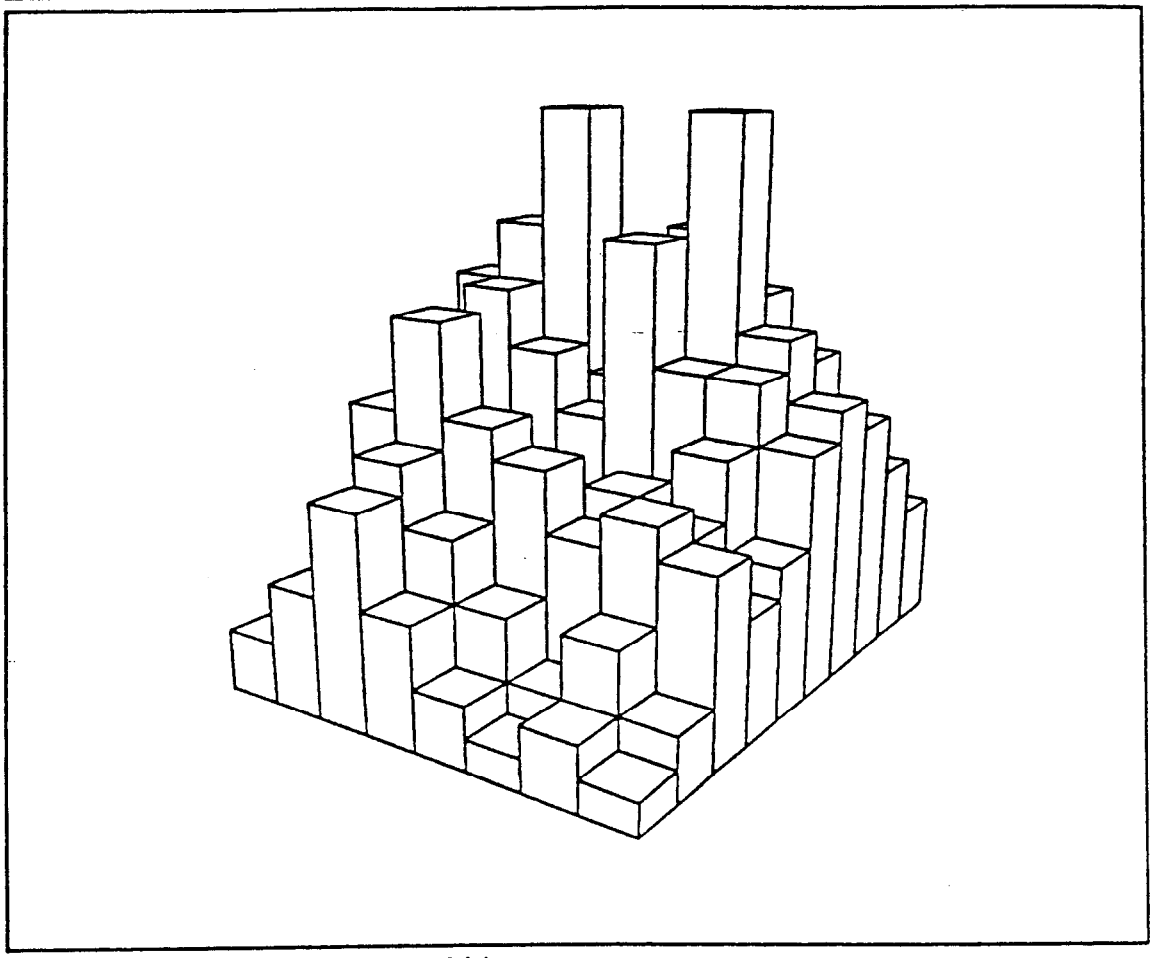

Figure 5.2. A two-dimensional histogram

PXYZA A real array of dimension (M,N) containing the x, y, and z coordinates of the two-dimensional histogram. The format of the array is

$$\begin{pmatrix} z_0 & x_1 & x_2 & \dots & x_{N-2} & x_{N-1} \\ y_1 & z_{1,1} & z_{2,1} & \dots & z_{N-2,1} & - \\ y_2 & z_{1,2} & z_{2,2} & \dots & z_{N-2,2} & - \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ y_{M-2} & z_{1,M-2} & z_{2,M-2} & \dots & z_{N-2,M-2} & - \\ y_{M-1} & - & - & \dots & - & - \end{pmatrix}.$$

The sequences $(x_1, x_2, \dots, x_{N-1})$ and $(y_1, y_2, \dots, y_{M-1})$ must be monotonically increasing but do not have to be equally spaced. The two-dimensional histogram consists of $(N-2)$ columns in the x direction and $(M-2)$ columns in the y direction. The value z_0 is the z coordinate of the base of the columns. The bounds of the (i,j)th column $(i = 1,\dots,(N-2); j = 1,\dots,(M-2))$ are x_i to x_{i+1} in x and y_j to y_{j+1} in y. This means that the last row and column of PXYZA are almost unused. These unused values are shown as dashes in the matrix. The

$z_{i,j}$ values give the z coordinates of the tops of the columns and should not be smaller than z_0.

PTRN A real array of dimension (3,4) containing the perspective transformation.

CCA A real array containing the color control for the two-dimensional histogram. The value of CCA(1) selects one of three possibilities:

- CCA(1)=-1.0 This means each side of a column is to be colored in a distinct color. Additional data is supplied in the array as described below.
- CCA(1)=0.0 This means that the existing GKS settings for fill areas and polylines is to be used to draw the polygons. No additional data is supplied in the array.
- CCA(1)=1.0 This means that the columns are to be colored using the light source and reflection model. Additional data is supplied in the array as described earlier.

L An integer giving the length of the work array.

WA A real array of dimension L that will be used as a work array. L should be at least $6(M-2)(N-2)$.

This subroutine allows the special coloring scheme defined by a value of CCA(1) equal to minus one. The description of the CCA array in this case is as follows:

CCA(1) To specify this option, this value should contain a real value of minus one.

CCA(2) A real value of zero indicates that the indices given below are color indices. A real value of one indicates that the indices given below are attribute bundle indices.

CCA(3) A real value which specifies the index to be used to draw the x_{min} side of the polygon. This value will be converted to an integer before it is used.

CCA(4) A real value which specifies the index to be used to draw the x_{max} side of the polygon. This value will be converted to an integer before it is used.

CCA(5) A real value which specifies the index to be used to draw the y_{min} side of the polygon. This value will be converted to an integer before it is used.

CCA(6) A real value which specifies the index to be used to draw the y_{max} side of the polygon. This value will be converted to an integer before it is used.

CCA(7) A real value which specifies the index to be used to draw the z_{min} side of the polygon. This value will be converted to an integer before it is used.

CCA(8) A real value which specifies the index to be used to draw the z_{max} side of the polygon. This value will be converted to an integer before it is used.

Figure (5.2) shows an example of a two-dimensional histogram. Like all of the examples in this chapter, it was produced by drawing the polygons in the background color and then outlining the polygons in the normal color.

5.2. Mesh Surfaces

A mesh surface consists of a rectangular sheet positioned above a rectangular area in the x-y plane. The sheet is divided into smaller rectangular or triangular patches. The height of the corners of the patches of the sheet can be used to represent experimental or synthetic data.

If the data suppled to the mesh surface subroutine is not relatively smooth, the resulting picture may be difficult to interpret. In this case, a two-dimensional histogram may be more appropriate.

5.2.1. Subroutine GZMESH: Draw a Mesh Surface

This subroutine may be used to draw a mesh surface. The mesh may be constructed by drawing rectangles or splitting each rectangle into a pair of triangles. Either the upper side, lower side, or both sides of the surface may be drawn. When only one side of the surface is drawn, a *skirt* is drawn around the base.

The polygons that constitute the surface may be drawn in one of two ways. In the first scheme, the existing GKS settings are used to draw the polygons as fill areas and then outline the polygons using a polyline. The second scheme provides a light source and reflection model to color the polygons.

The calling sequence is:

```
CALL GZMESH(M,N,PXYZA,SFLG,MFLG,PTRN,CCA,L,WA)
```

The input parameters are:

M — An integer giving the first dimension of PXYZA.

N — An integer giving the second dimension of PXYZA.

PXYZA — A real array of dimension (M,N) containing the x, y, and z coordinates of the mesh surface. The format of the array is

$$\begin{pmatrix} z_0 & x_1 & x_2 & \cdots & x_{N-2} & x_{N-1} \\ y_1 & z_{1,1} & z_{2,1} & \cdots & z_{N-2,1} & z_{N-1,1} \\ y_2 & z_{1,2} & z_{2,2} & \cdots & z_{N-2,2} & z_{N-1,2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ y_{M-2} & z_{1,M-2} & z_{2,M-2} & \cdots & z_{N-2,M-2} & z_{N-1,M-2} \\ y_{M-1} & z_{1,M-1} & z_{2,M-1} & \cdots & z_{N-2,M-1} & z_{N-1,M-1} \end{pmatrix}.$$

The sequences $(x_1, x_2, \ldots, x_{N-1})$ and $(y_1, y_2, \ldots, y_{M-1})$ must be monotonically increasing but do not have to be equally spaced. The mesh surface consists of $(N-2)$ surface elements in the x direction and $(M-2)$ surface elements in the y direction. The bounds of the (i,j)th surface

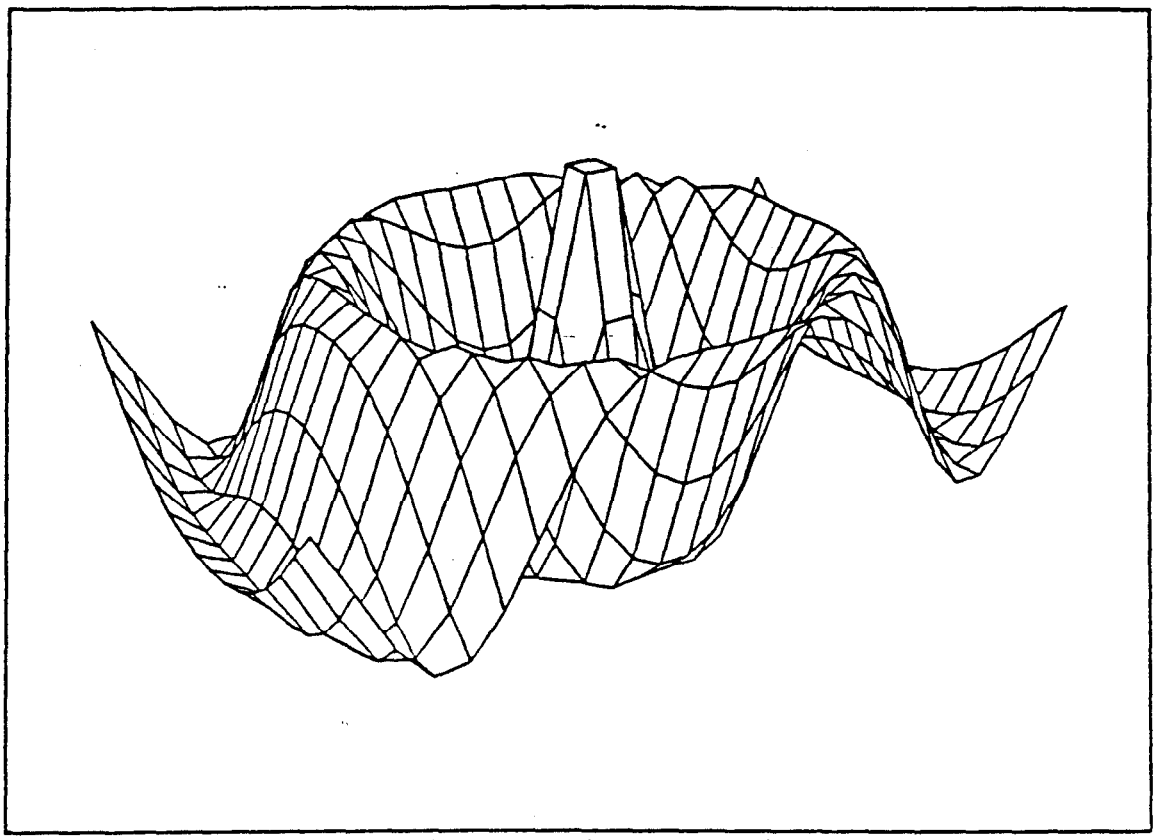

Figure 5.3. A mesh surface showing both upper and lower sides

element ($i = 1, \ldots, (\mathtt{N}-2)$; $j = 1, \ldots, (\mathtt{M}-2)$) are x_i to x_{i+1} in x and y_j to y_{j+1} in y. The $z_{i,j}$ values give the z coordinates of the corners of the rectangular surface elements. The value z_0 is the z coordinate of the base of the structure and is only used if a skirt is being drawn. When a skirt is drawn for the upper side of the surface, z_0 must not be greater than any of the $z_{i,j}$ values. When a skirt is drawn for the lower side of the surface, z_0 must not be smaller than any of the $z_{i,j}$ values.

SFLG An integer specifying which side of the surface is to be drawn. A positive value means the upper side is to be drawn while a negative value means the lower side. A zero value means both sides are to be drawn.

MFLG An integer specifying the type of mesh to be drawn. A zero value means rectangles are to be drawn while nonzero values mean triangles are to be drawn. A positive value means the dividing line for the triangles will pass through $z_{1,1}$ and a negative value means it will not.

PTRN A real array of dimension (3,4) containing the perspective transformation.

CCA A real array containing the color control for the mesh surface. The value of `CCA(1)` selects one of two possibilities:

`CCA(1)=0.0` This means that the existing GKS settings for fill

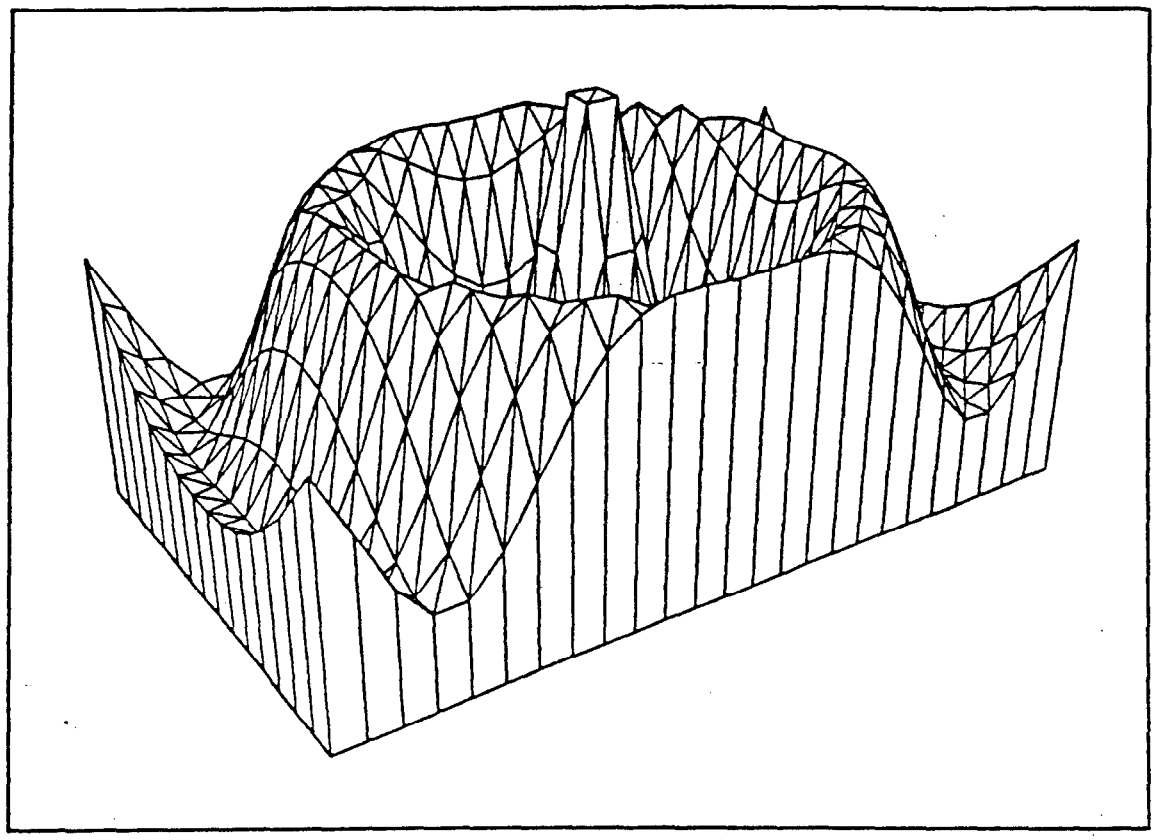

Figure 5.4. A mesh surface showing the upper side only

areas and polylines is to be used to draw the polygons. No additional data is supplied in the array.

`CCA(1)=1.0` This means that the surface elements are to be colored using the light source and reflection model. Additional data is supplied in the array as described earlier.

`L` An integer giving the length of the work array.

`WA` A real array of dimension `L` that will be used as a work array. The required size of `WA` is difficult to estimate. The maximum size is easier to compute. Start with $(M-2)(N-2)$. If triangles are being drawn, double that value. If both top and bottom are being drawn, double that value again. If only the top or bottom is being drawn, add $2(M-2)+(N-2)+1$. Finally, double that value. This value overestimates the number of words needed; the actual number will usually be about half this maximum number.

Figures (5.3), (5.4), and (5.5) all illustrate examples of mesh surfaces. In Figure (5.3) the mesh surface was broken down into rectangles, while the other two figures use triangles. In Figure (5.4) the triangular division goes through the $z_{1,1}$ point while in Figure (5.5) it does not.

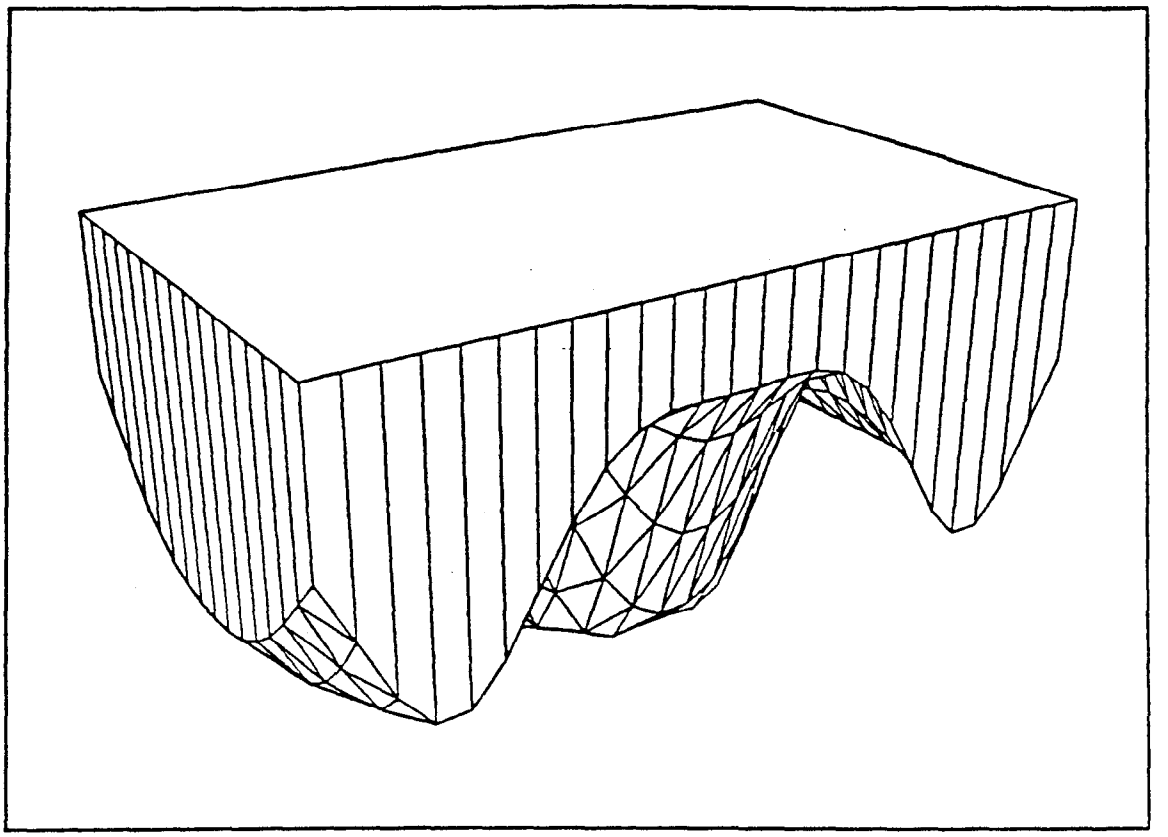

Figure 5.5. A mesh surface showing the lower side only

5.3. Generalized Polyhedral Solids

This section describes a subroutine that can be used to draw any figure that can be broken down into planar polygons. Normally the polygons should be organized into solid polyhedra because, at most, only one side of each polygon will be drawn.

5.3.1. Subroutine GZPOLY: Draw a Generalized Polyhedra

This subroutine may be used to draw a group of polyhedra. A polyhedron consists of a solid body bounded by polygonal faces. The polygons should be planar or very nearly so. The polygons must also be nonintersecting. The points on the boundary should be ordered in such a manner that the polygon is to the left as one traverses the outside of the surface in the given order of the bounding points.

The polygons that constitute the polyhedra may be drawn in one of two ways. In the first scheme, the existing GKS settings are used to draw the polygons as fill areas and then outline the polygons using a polyline. The second scheme provides a light source and reflection model to color the polygons.

The calling sequence is:

```
CALL GZPOLY(PXA,PYA,PZA,M,NPA,IPA,IXA,PTRN,CCA,L,WA)
```

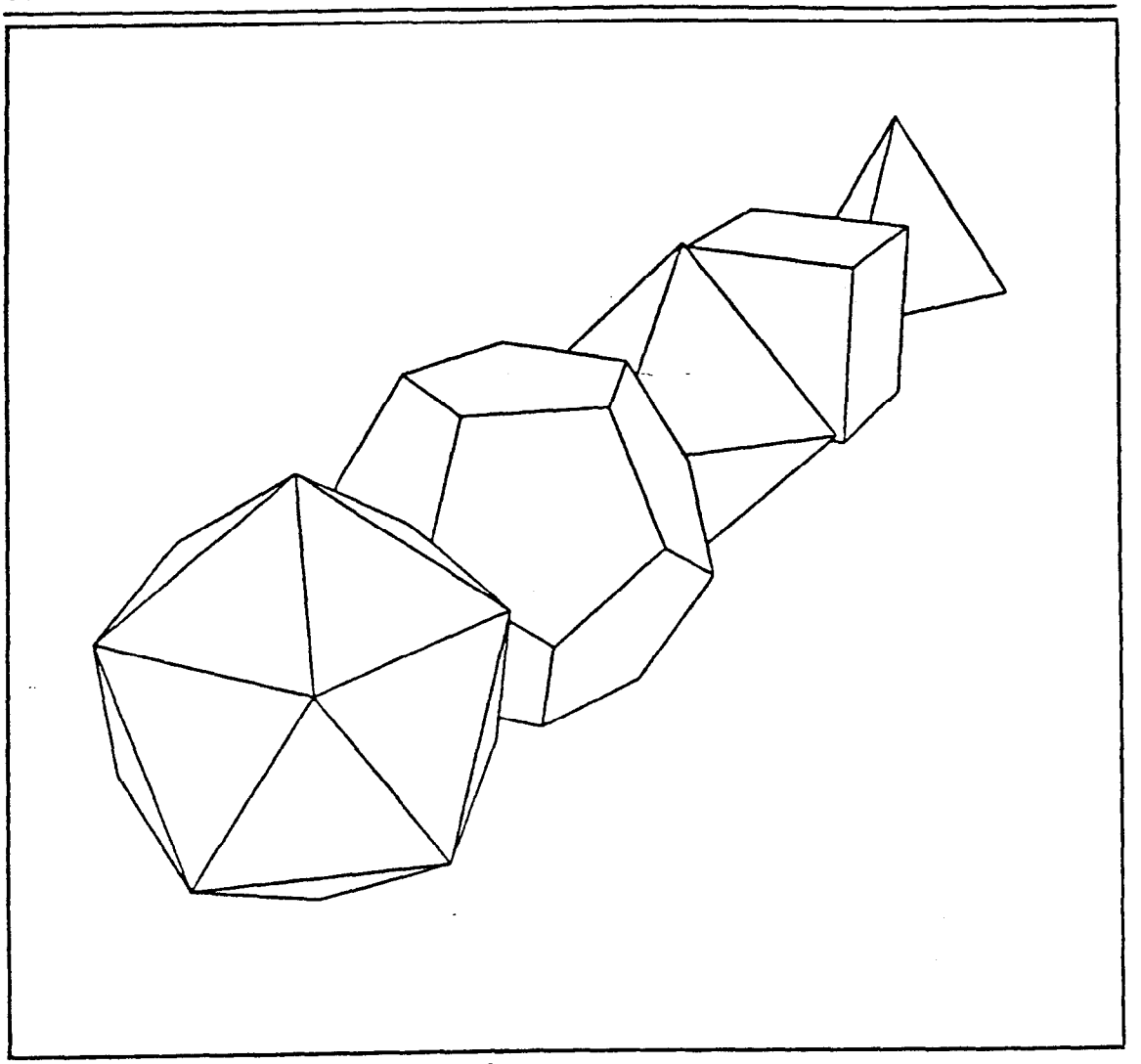

Figure 5.6. The five Platonic solids

The input parameters are:

PXA	A real array of containing the x coordinates of the points on the polyhedra.
PYA	A real array of containing the y coordinates of the points on the polyhedra.
PZA	A real array of containing the z coordinates of the points on the polyhedra.
M	An integer giving the number of polygons in the polyhedra.
NPA	An integer array of dimension M containing the number of points in each of the polygons. The maximum number of points allowed in each polygon is 16. However, there is no need to close the polygon; a triangular polygon may be defined by giving only three points.
IPA	An integer array of dimension M containing a pointer into the IXA array that gives the starting index of the indices pointing to the coordinates

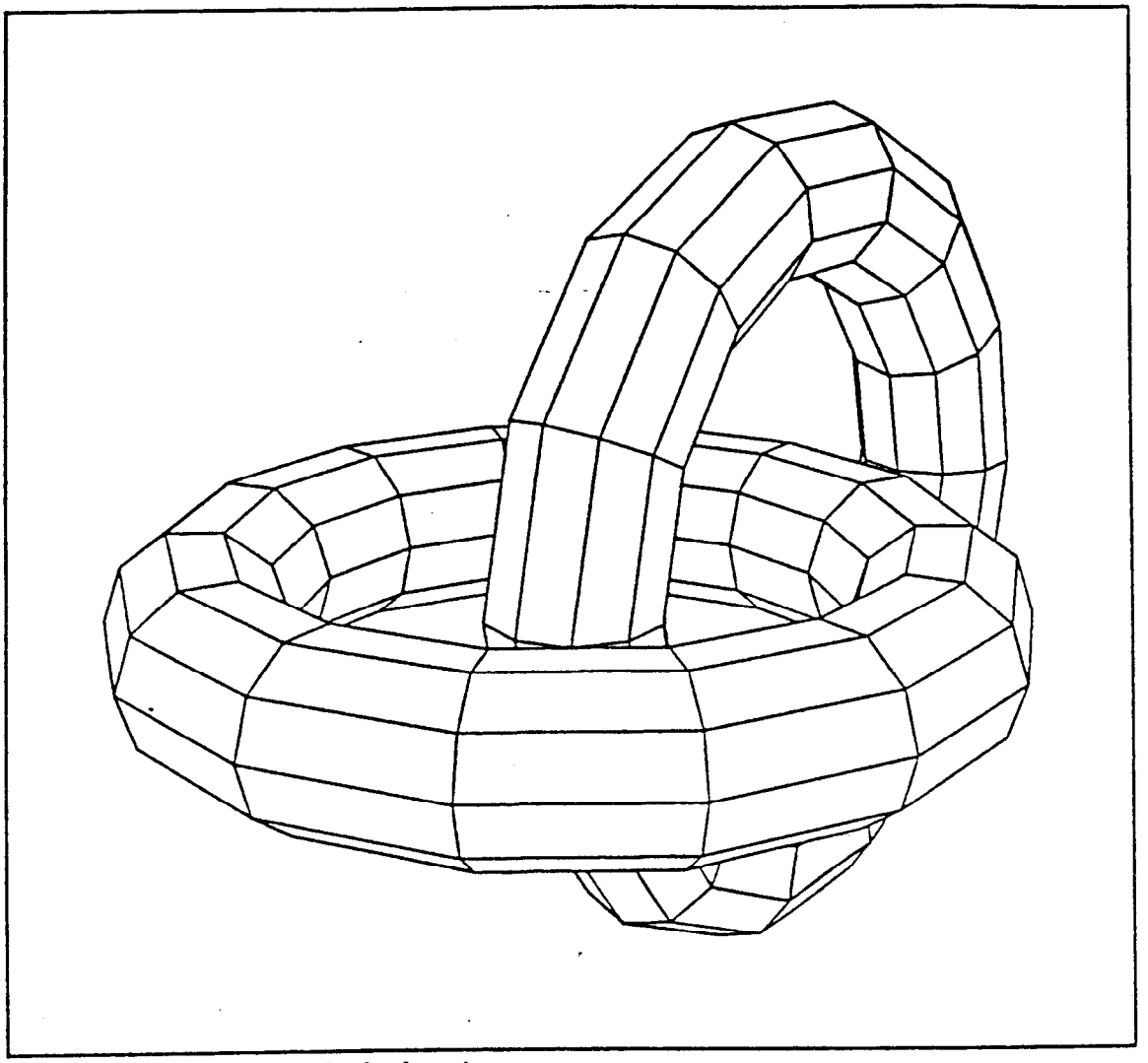

Figure 5.7. Two interlocked tori

of the points in the PXA, PYA, and PZA arrays bounding the polygon.

IXA An integer array containing the indices of the points bounding the polygons.

PTRN A real array of dimension (3,4) containing the perspective transformation.

CCA A real array containing the color control for the polyhedra. The value of CCA(1) selects one of two possibilities:

CCA(1)=0.0 This means that the existing GKS settings for fill areas and polylines is to be used to draw the polygons. No additional data is supplied in the array.

CCA(1)=1.0 This means that the polygons are to be colored using the light source and reflection model. Additional data is supplied in the array as described earlier.

L An integer giving the length of the work array.

WA A real array of dimension L that will be used as a work array. The maximum size that will ever be required is 2M. The actual number needed will usually be about half this number.

Figures (5.6) and (5.7) illustrate two applications of this subroutine. Notice that Figure (5.6) could have been produced in two distinct ways. In the first case, a single call could be made to subroutine GZPOLY supplying it with all of the data necessary to draw all five solids. A second way that the figure could have been produced is to draw each of the five solids in turn starting with the farthest from the viewer; first the tetrahedron, then the cube, octahedron, dodecahedron, and finally the icosahedron. Since the farther solids do not hide the nearer ones, either method will produce exactly the same result. This second method will be slightly more efficient because the subroutine always has smaller files to sort. However, the order that the polyhedra must be processed is dependent on the viewing direction. This shortcut will not work in producing Figure (5.7) because each of the tori hides part of the other; the entire figure must be produced in a single call to GZPOLY.

References

The following list contains more information about the books and reports that have been referenced in this document.

[ANS78] *American National Standard: Programming Language FORTRAN*, Document ANSI X3.9–1978, American National Standards Institute, Inc., New York, April 1978.

[ANS85a] *American National Standard for Information Systems: Computer Graphics – Graphical Kernel System (GKS) Functional Description*, Document ANSI X3.124–1985, American National Standards Institute, Inc., New York, June 1985.

[ANS85b] *American National Standard for Information Systems: Computer Graphics – Graphical Kernel System (GKS) FORTRAN Binding*, Document ANSI X3.124.1–1985, American National Standards Institute, Inc., New York, June 1985.

[ANS86] *American National Standard for Information Systems: Coded Character Sets, 7-bit American National Standard Code for Information Interchange (7-bit ASCII)*, Document ANSI X3.4–1986, American National Standards Institute, Inc., New York, March 1986.

[Bea91] Robert C. Beach, *An Introduction to the Curves and Surfaces of Computer-Aided Design*, Van Nostrand Reinhold, New York, 1991.

[Her67] A. V. Hershey, *Calligraphy for Computers*, Report Number 2101, United States Naval Weapons Laboratory, Dahlgren, Virginia, August 1967.

[Rog85] David F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.