High-resolution Computer Graphics Using FORTRAN 77

Ian O. Angell and
Gareth Griffith

High-resolution Computer Graphics Using FORTRAN 77

Ian O. Angell and
Gareth Griffith

Department of Information Systems
London School of Economics
University of London
Houghton Street, London WC2A 2AE

MACMILLAN

First published 1987
Reprinted 1989

Published by
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

Printed in Hong Kong

British Library Cataloguing in Publication Data
Angell, Ian O.
High-resolution computer graphics using Fortran 77.—(Macmillan computer studies series)
1. Computer graphics 2. FORTRAN (Computer program language)
I. Title II. Griffith, Gareth
006.6'76 T385

ISBN 0-333-40398-3
ISBN 0-333-40399-1 Pbk

Preface

Until recently, all but the most trivial computer graphics was the province of specialised research groups. Now with the introduction of inexpensive microcomputers and 'graphics-boards', the subject will reach many more users and its full potential can be realised. Computer-generated pictures involving smooth shading, shadows and transparent surfaces, for example, have made a major impact in television advertising. The 'mysterious' techniques for producing such pictures have gained a (false) reputation of complexity for computer graphics.

This book gives a practical description of these ideas and, after studying the contents and implementing the examples and exercises, the reader will be ready to attempt most tasks in graphics.

It is assumed that the reader has an elementary knowledge of the FORTRAN 77 programming language, and of Cartesian co-ordinate geometry. This knowledge will be used to produce simple diagrams, and to create the basic programming tools and routines for the more advanced colour pictures. Then, hopefully, the reader will be inspired to seek a greater understanding of geometry and also to read the more advanced journals (such as *SIGGRAPH* and *ACM Transactions*) describing recent research developments in computer graphics. We give a number of relevant references throughout the text, but for a more comprehensive bibliography readers are advised to refer to Newman and Sproull (1973) and Foley and Van Dam (1981).

The only way to understand any branch of applied computing is to study and write a large number of programs; this is why the format of this book is that of understanding through program listings and worked examples. The chapters are centred around numerous examples and the ideas that lead from them. Many students readily understand the theory behind graphics, but they have great difficulty in implementing the ideas. Hence great emphasis is placed on the program listings; well over a hundred routines are given – some quite substantial. Total understanding of the theory given in this book will be achieved only by running these programs and experimenting with them. The programs can be thought of as an embryonic graphics package, but most importantly they are a means of describing the algorithms required in the solution of the given problems. They are readily translatable into other computer languages such as Basic, C and Pascal. The routines given all relate to a small number of graphics primitives, which are necessarily device or package dependent. Examples of these primitives

are given for the Tektronix 4100 range, for the G.K.S. standard, and the GINO and sample microfilm packages in an appendix.

On occasions, efficiency has been sacrificed in favour of clarity in the description of algorithms. The programs are written in a modular form, which makes the interchanging of program functions relatively straightforward. A unique name is given to any routine that solves a given problem, but we will give more than one example of such a routine if different, perhaps more general, solutions are required. For example, routine FACFIL can be used to draw a polygon in a fixed colour, but other manifestations may include smooth shading of that polygon and even various textures.

The main purpose of this book, which is essentially a third year undergraduate and M.Sc. course at the University of London, is to set the groundwork of computer graphics. Some of the figures given in this book were produced by past students of the course. Figure 5.1 was produced by Hilary Green, figure 9.11 by Colin Ching, figure 12.5 by Andrew Pullen and figure 18.3 by Paul Mclean Thorne. All of the figures can be produced by using the listings in this book (with some extensions). We would also like to thank Digital Arts Production Ltd and Digital Arts Software Ltd for the use of their computing facilities. The programs given are NOT the only way of solving the problems of computer graphics: they reflect the teaching and research interests of the authors. They do, however, provide a general strategy for gaining a thorough understanding of computer graphics and should lead the reader to research level in the subject. With this advanced groundwork done, readers can reorganise the approach with their own specific interests in mind. The listings use a restricted form of FORTRAN 77 in order to make the programs applicable to many of the FORTRAN subsets now available on microcomputers.

The package developed in this book is for teaching purposes only. Although the authors place no copyright restrictions on the use of the listings in a lecture course, no commercial exploitation is permitted without prior agreement with the authors.

Computer graphics is fun! We hope that readers will make this discovery and spend many enjoyable and productive hours in front of their graphics console.

Ian O. Angell and Gareth Griffith

Contents

1 *Familiarisation with Graphics Devices and Primitives*

Computer graphics devices come in all shapes and sizes: storage tubes, raster devices, vector refresh displays, flat-bed plotters etc., which is why in recent years so much effort has been put into graphics software standards (such as G.K.S. (Hopgood *et al.*, 1983)) as well as into the portability of graphics packages (GINO, CalComp etc.). This book will concentrate on the techniques of *modelling* and *rendering* (that is, drawing, colouring, shading etc.) two-dimensional and three-dimensional objects, which surprisingly require only a small part of the above systems. Rather than restrict ourselves to one software system, and in order to make this book relevant to as many different graphics devices as possible, we will identify a general model for a graphics device together with a few (nine) elementary routines (*primitives*) for manipulating that model. *From the outset is must be realised that we are using the word 'primitive' in the literal sense of describing the basic level at which the programs in this book communicate with graphics devices*; the word has different meanings in other graphics environments, such as G.K.S. The FORTRAN 77 programs that follow will use only these primitives for drawing on this basic model (apart from a few very exceptional cases). Since even the most complex programs given in this book interface with the model device through relatively few primitive routines, the graphics package we create is readily portable. All that is needed is for users to write their own device-specific primitives, which relate to their particular graphics device or package! Later in this chapter we give ideas of how such primitives may be written, and in the appendix there are example listings of primitives suitable for some of the more popular graphics devices and standards.

The Model Graphics Device

We assume that the display of the graphics device contains a *viewport* (or *frame*) made up from a rectangular array of points (or *pixels*). This matrix is NXPIX pixels horizontally by NYPIX pixels vertically. The values of NXPIX and NYPIX are stored in COMMON block

```
COMMON/VIEWPT/NXPIX, NYPIX
```

An individual pixel in the viewport can be accessed by referring to its ***pixel co-ordinates***, a pair of integers, which give the position of the pixel relative to the ***bottom left-hand corner*** of the viewport. The pixel (IXPIX, IYPIX) is IXPIX pixels horizontally and IYPIX pixels vertically from the bottom left-hand corner (which naturally has pixel co-ordinates (0, 0)). Note that for all pixels, $0 \leqslant$ IXPIX $<$ NXPIX and $0 \leqslant$ IYPIX $<$ NYPIX, and the top right corner is (NXPIX $-$ 1, NYPIX $-$ 1). See figure 1.1. There are a few commercial graphics systems which use top left as (0,0) and bottom right as (NXPIX $-$ 1, NYPIX $-$ 1), but this can be compensated for in the primitives we construct and will not require a major rewrite of the larger programs.

Colour television and *RGB colour monitors* work on the principle of a colour being a mixture of red, green and blue components. Each pixel is made up of three tiny dots, one each of red, green and blue, and different colours are produced by varying the relative brightness of the dots. Red is given by a bright red dot with no green or blue; yellow is produced by bright red and green dots with no blue, and so on (see chapter 15 for a more detailed description). For this reason most graphics devices define colours in terms of red, green and blue components. We assume that our graphics device has a ***colour look-up table*** which contains the definitions in this form of NCOL colours, each accessed by an integer value between 0 and NCOL $-$ 1. Such an integer value is called a ***logical colour*** while the entries in the look-up table are referred to as ***actual colours***. The entries in the look-up table may be redefined by the user, but initially we assume the entries take ***default values***. We assume that the display on the model graphics device is based upon a ***bit-map***: associated with every pixel there is an integer value (representing a logical colour) and the pixel is displayed in the corresponding actual colour.

We imagine a cursor that moves about the viewport; the pixel co-ordinate of this cursor at any time is called its ***current position***. Objects are drawn by moving the cursor around the viewport and resetting the value in the bit-map at the current position to the required logical colour.

The Nine Primitives

The viewport may need some preparatory work done before it can be used for graphical display. We assume that this is achieved by the primitive call

```
CALL PREPIT
```

After pictures have been drawn some 'housekeeping' may be needed to finish the frame (see the section on the command code method later in this chapter for an explanation of buffers), and this is done by the primitive call

```
CALL FINISH
```

Only one logical colour can be used at a time, so to change the *current colour* to logical colour ICOL, $0 \leqslant \text{ICOL} < \text{NCOL}$, we use the call

```
CALL SETCOL (ICOL)
```

We can erase all the pixels in the viewport with the current colour by

```
CALL ERASE
```

If we are using microfilm then ERASE can be used to move onto the next frame of the film.

We can colour the current pixel (IXPIX, IYPIX) in the current colour by

```
CALL SETPIX (IXPIX, IYPIX)
```

The graphics cursor can be moved about the viewport to its current (pixel) position (IXPIX, IYPIX) without changing the colour by the primitive call

```
CALL MOVPIX (IXPIX, IYPIX)
```

Or we can draw a line in the current colour from the current position to a new position (IXPIX, IYPIX)

```
CALL LINPIX (IXPIX, IYPIX)
```

(IXPIX, IYPIX) then becomes the current position.

We can fill in a polygon whose vertices are defined by N pixel vectors (IXP(i), IYP(i)), $i = 1, \ldots, N$ taken in order, by the call

```
CALL POLPIX (N, IXP, IYP)
```

Finally, we need a primitive which defines the actual colours in the colour look-up table. There are several methods for dealing with such definitions (Ostwald, 1931; Smith, 1978; Foley and Van Dam, 1981), but we assume that the table entry referred to by logical colour I is made up of red (R), green (G) and blue (B) components which may be set by

```
CALL RGBLOG (I, R, G, B)
```

The intensity of each component is a value between zero and one: zero means no component of that colour is present, one means the full colour intensity. For example, black has RGB components 0, 0, 0, white has 1, 1, 1, red has 1, 0, 0, while cyan is 0, 1, 1. These colours can be 'darkened' by reducing the intensities from one to a fractional value. Initially we shall use just eight default actual colours, comprising black (logical 0), red (1), green (2), yellow (3), blue (4), magenta (5), cyan (6) and white (7). Note the three bits of the binary representation of the numbers 0 to 7 give the presence (1) or absence (0) of the three component colours. The default background and foreground logical colours may be set by the user, we assume 0 and 7 respectively, although for the purpose of diagrams in this book we use black foreground and white background for obvious reasons.

These primitives are by no means the last word. Users of special-purpose graphics devices should extend the list of primitives in order to make full use of the potential of their particular device. For example, many raster devices have different styles of line drawing; thus a line need not simply be drawn in a given (numerical) colour, each pixel along the line may be coloured by a bit-wise boolean binary operation (such as exclusive OR) on the value of the present colour of that pixel and the current drawing colour. A line could be dashed! We shall introduce a new (tenth) primitive in chapter 5 for introducing different ***line styles***. Another possible primitive would be the window manager referred to below. In this book we concentrate on geometric modelling; we do not consider the whole area of computer graphics relating to the construction and manipulation of two-dimensional objects which are defined as groups of pixels (***user-defined characters*** (Angell, 1985), ***icons*** and ***sprites***). You could introduce your own primitives for manipulating these objects should your particular graphics application need them.

Implementing the Primitives

We consider two different ways of writing the primitive routines. The first is applicable to users who have access to a two-dimensional graphics package (either in software or hardware), in which case all communication between the primitives and the graphics display will be made via that package. The second is for users of a device for which all manipulation of the display is done by sending a sequence of graphics commands, each command being an *escape character*, followed by a *command code*, possibly followed by a list of integers referring to pixels and/or colours.

The graphics package method

Many graphics packages will have their own routines similar to our nine primitives. Do *not* go through the program listings given in this book, replacing all references to our nine primitives with the names of the equivalent package routines. It is far more efficient to write individual subroutines for our nine primitives, each of which simply calls the corresponding package routine. You must, however, be aware of any peculiarities or restrictions of your package, in order to ensure that your use of the package corresponds exactly to the definition of the nine primitives above.

Note that graphics commands for microcomputers, such as the IBM PC, also fall into this category. You should further note that some graphics systems (such as microfilm: see the appendix) use the concept of addressable points as opposed to pixels. If a dot is drawn at such an addressable point, then the area centred at that dot will contain a number (certainly tens, perhaps hundreds) of other addressable points. To use our system you will have to identify squares of addressable points with individual pixels.

A graphics package could give you a number of different ways of obtaining the effect of one primitive. The most obvious example is that of filling a polygonal area. Some devices give you a normal area fill (or perhaps a triangle fill), whereby the polygon defined by the pixel co-ordinates of its vertices is filled in the current colour; a *flood fill* which uses the current colour to fill in all pixels in the viewport connected to and of the same colour as an initially specified pixel (seed point); a *boundary fill* which starts at a given pixel, and colours all connected pixels out to a given boundary colour. Some give ***pie fills*** – that is, filling segments of circles. Others allow ***pattern filling***, where areas are filled not in single colours but with combinations of different coloured pixels. All of these can be included in your own specialised primitives should you have a need for them.

If you are working with a single-colour line-drawing package or one which does not give you an area fill, then you have to write your own area-fill primitive using sequences of parallel lines (see chapter 5).

Example primitives for the Graphical Kernel System (G.K.S.) and GINO, and sample Microfilm packages are given in the appendix.

The command code method

Many high-resolution raster display terminals fall into this category. They are normally connected to a host computer, with communication achieved via character string transfer along a pre-defined input/output channel. Graphical information is distinguished from ordinary text by preceding the string of graphics information with a special escape symbol, the strings being sent to the terminal by the usual FORTRAN WRITE statement. Since this character transfer process can be slow, many systems accept ***buffered and/or encoded information*** for increased efficiency where a section of memory is used to hold a number of commands, and only when the buffer is full is the information transferred to the display device. *Flushing* of a partially filled buffer on the completion of a drawing may be included in the FINISH primitive. Examples of primitives for the Tektronix 4100 series are also given in the appendix.

Listing 1.1

```
C     **************
      PROGRAM DEMO
C     **************

      DIMENSION IXP(3),IYP(3)
      COMMON/VIEWPT/ NXPIX,NYPIX
C     Prepare graphics viewport.
      CALL PREPIT
C     Define logical colour 8 to be grey.
      CALL RGBLOG(8,0.5,0.5,0.5)
C     Set current colour to grey.
      CALL SETCOL(8)
C     Define the vertices of a triangle.
      IXP(1)=0
      IYP(1)=0
```

```
      IXP(2)=NXPIX-1
      IYP(2)=0
      IXP(3)=0
      IYP(3)=NYPIX-1
C   Fill in this triangle in current colour.
      CALL POLPIX(3,IXP,IYP)
C   Define the vertices of a square centred in the viewport.
C   First the bottom left-hand corner.
      IX1=IFIX(FLOAT(NXPIX)*0.25+0.5)
      IY1=IFIX(FLOAT(NYPIX)*0.25+0.5)
C   Then the top right-hand corner.
      IX2=IFIX(FLOAT(NXPIX)*0.75+0.5)
      IY2=IFIX(FLOAT(NYPIX)*0.75+0.5)
C   Set current colour to white.
      CALL SETCOL(7)
C   Draw the outline of the square.
      CALL MOVPIX(IX1,IY1)
      CALL LINPIX(IX2,IY1)
      CALL LINPIX(IX2,IY2)
      CALL LINPIX(IX1,IY2)
      CALL LINPIX(IX1,IY1)
C   Draw white dot in the centre of the viewport.
      IXC=IFIX(FLOAT(NXPIX)*0.5+0.5)
      IYC=IFIX(FLOAT(NYPIX)*0.5+0.5)
      CALL SETPIX(IXC,IYC)
C   Call the end of frame routine.
      CALL FINISH
      STOP
      END
```

Example 1.1

In listing 1.1 we give a contrived program to draw a pattern of dots, lines and areas. It uses all nine primitives. ERASE is implicit in PREPIT.

Exercise 1.1

Many packages allow the construction of more than one viewport on the display whereas our routines refer to just one viewport, *the current viewport.*

Introduce your own routines which allow for multiple viewports. Assume that your display will hold NUMVPT ($\geqslant 1$) viewports. Change the /VIEWPT/ block to hold two variables NUMVPT and NOWVPT, and four arrays NXPIX, NYPIX, IXBASE and IYBASE. The i^{th} viewport is a rectangle of NXPIX(i) pixels by NYPIX(i) pixels, with the bottom left-hand corner of that viewport being a display pixel with co-ordinates (IXBASE(i), IYBASE(i)). Only one viewport is active at any given time, and the index of the current viewport is denoted by NOWVPT. You will have to change some of the above primitives accordingly.

Starting a Graphics Library: Routines that Map Continuous Space onto the Viewport

The use of pixel vectors for drawing (in particular) three-dimensional pictures is very limiting. The definition of objects using such discrete pairs of integers has very few real applications. We need to consider plotting views on the graphics

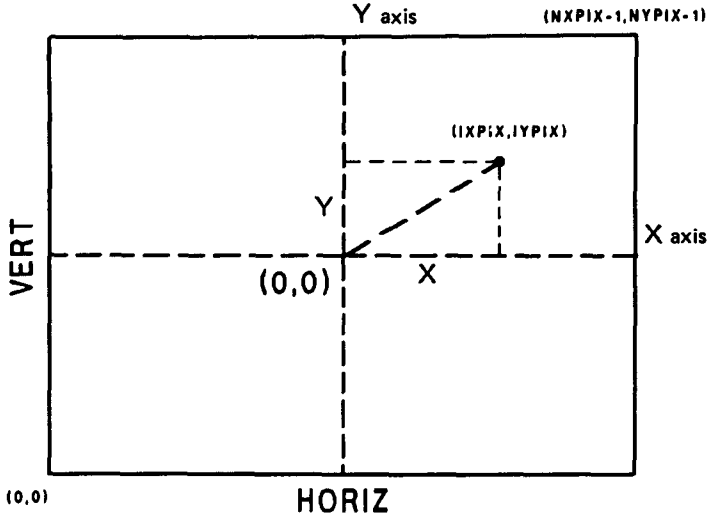

Figure 1.1

display, where the objects drawn are defined in terms of real units, whether they be millimetres or miles. Since our primitives draw using pixels, we have to consider a library of routines which relate ***real space*** with the pixels of our viewport. Before attempting this step we must first discuss ways of representing two-dimensional space by means of Cartesian co-ordinate geometry.

We may imagine two-dimensional space as the plane of this page, but extending to infinity in all directions. In order to specify the position of points on this plane uniquely, we have to impose a Cartesian co-ordinate system on the plane. We start by arbitrarily choosing a fixed point in this space, which is called the ***co-ordinate origin***, or origin for short. A line, that extends to infinity in both directions, is drawn through the origin – this is the ***x-axis***. The normal convention, which we follow, is to imagine that we are looking at the page so that the x-axis appears from left to right on the page (the horizontal). Another two-way infinite axis, the ***y-axis***, is drawn through the origin perpendicular to the x-axis; hence conventionally this is placed from the top to the bottom of the page (the vertical). We now draw a scale along each axis; unit distances need not be the same on both axes or even linearly distributed along the axes but this is

defined by using the y-axis. These two values, called a *co-ordinate pair* or *two-dimensional point vector*, are normally written in brackets thus: (X, Y), the x co-ordinate coming before the y co-ordinate. We shall usually refer to the pair as a vector – the dimension (in this case dimension two) will be understood from the context in which we use the term. A vector, as well as defining a point (X, Y) in two-dimensional space, may also be used to specify a direction, namely the direction that is parallel to the line that joins the origin to the point (X, Y) – but more of this (and other objects such as lines, curves and polygons) in chapter 3.

It must be realised that the co-ordinate values of a point in space are totally dependent on the choice of co-ordinate system. During our analysis of computer graphics algorithms we will be using a number of different co-ordinate systems to represent the same objects in space, and so a single point in space may have a number of different vector co-ordinate representations. For example, if we have two co-ordinate systems with parallel axes but different origins – say separated by a distance 1 in the x direction and 2 in the y direction. Then the point (0, 0) in one system (its origin) could be (1, 2) in the other: the same point in space but different vector co-ordinates. In order to clarify the relationships between different systems we introduce an arbitrary but fixed ABSOLUTE co-ordinate system, and ensure that all other systems can be defined in relation to it. This ABSOLUTE system, although arbitrarily chosen, remains fixed throughout our discussion of two-dimensional space. (Some authors call this the *World Co-ordinate System*.) Normally we will define the position and shape of objects in relation to this system.

Having imposed this fixed origin and axes on two-dimensional space, we now isolate a rectangular area (or *window*) of size HORIZ by VERT units, which is also defined relative to the ABSOLUTE system. This window is to be identified with the viewport so that we can draw views of two-dimensional scenes on the model graphics device. We may wish to move the window about two-dimensional space taking different views of the same objects. To do this we create a new co-ordinate system, the WINDOW system, whose origin is the centre of the window, and whose axes are parallel to the edges of the window, are scaled equally in both x and y directions, and extend to infinity outside the window. Since we will be defining objects such as lines, polygons etc. in terms of the ABSOLUTE system, we have to know the relationship between the ABSOLUTE and WINDOW systems – that is, the relative positions of the origins and orientations of the respective axes. Having this information, we can relate the ABSOLUTE co-ordinates of points with their WINDOW co-ordinates and thence represent them as pixels in the viewport.

We begin our graphics package by assuming that the ABSOLUTE and WINDOW systems are identical, so that objects defined in the ABSOLUTE system have the same co-ordinates in the WINDOW system: in chapter 4 we will consider the more general case of the window moving around and about the ABSOLUTE system. We give routines that operate on points given as real co-ordinates in the WINDOW system, convert them to the equivalent pixels in the viewport, and

finally operate on these pixels with the graphics primitives mentioned earlier. Naturally these routines will then be *machine-independent*, and to transport the package between different computers and graphics displays all that is needed is a FORTRAN 77 compiler and the small number of *display specific* primitives. Programs dealing with the display of two- (and three-) dimensional scenes should rarely directly call the primitives: all communication to these primitives should be done indirectly using the routines below, which treat objects in terms of their real (rather than pixel) co-ordinates (listing 1.2).

We assume that the window is HORIZ units horizontally, hence the vertical side of the window (VERT) is HORIZ * NYPIX/NXPIX units, and we define the WINDOW co-ordinate origin to be at the centre of the window (figure 1.1). In order to identify the viewport with this window we must be able to find the pixel co-ordinates corresponding to any point within the window. The horizontal (and vertical) scaling factor relating window to viewport is XYSCAL = (NXPIX−1)/HORIZ and since the window origin is in the middle of the window we note that any point in space with WINDOW co-ordinates (X, Y) will be mapped into a pixel in the viewport with horizontal component IFIX (X * XYSCAL + (NXPIX−1) * 0.5 + 0.5) and vertical component IFIX (Y * XYSCAL + (NYPIX −1) * 0.5 + 0.5). Here IFIX is the FORTRAN function that *rounds down* – hence the final 0.5 for rounding to the nearest integer. These two components are programmed as two functions IFX and IFY, included in the library of routines in listing 1.2. All information needed about the dimensions of the window is stored in COMMON block /WINDOW/

```
COMMON/WINDOW/HORIZ, VERT, XYSCAL
```

Note if we do not assume that ABSOLUTE and WINDOW systems are identical and we have an object defined in ABSOLUTE system co-ordinates, then each point in the object must be transformed to its WINDOW co-ordinates before it can be drawn in the viewport – but more of this in chapter 4. The scaling factor (NXPIX−1)/HORIZ is chosen, and not NXPIX/HORIZ, to ensure that all points (±HORIZ/2, ±VERT/2) actually lie in the screen pixel area.

Listing 1.2

```
C     **********************
      SUBROUTINE START(H)
C     **********************

      COMMON/WINDOW/ HORIZ,VERT,XYSCAL
      COMMON/VIEWPT/ NXPIX,NYPIX
C   Set up viewport.
      CALL PREPIT
C   Set up window dimensions.
      HORIZ=H
      VERT=H*FLOAT(NYPIX)/FLOAT(NXPIX)
      XYSCAL=FLOAT(NXPIX-1)/HORIZ
      RETURN
      END
```