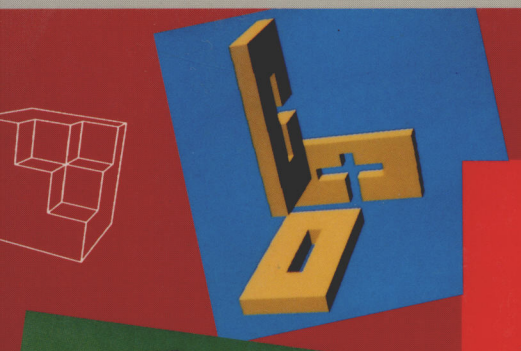
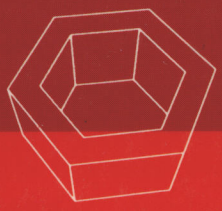
WILEY

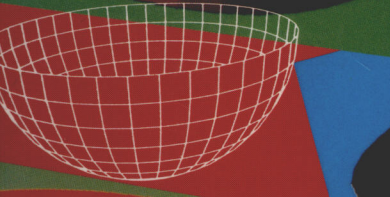
WILEY PROFESSIONAL COMPUTING

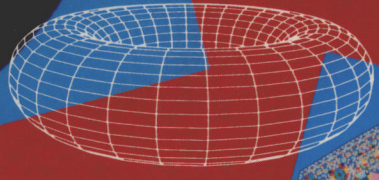




PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS

Second Edition





Leendert Ammeraal

DISK AVAILABLE

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS

Second Edition

Leendert Ammeraal
Hogeschool Utrecht, The Netherlands

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, G.P.O. Box 859, Brisbane,
Queensland 4001, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04,
Block B, Union Industrial Building, Singapore 2057

British Library Cataloguing in Publication Data

A catalogue record for this book is available
from the British Library

ISBN 0 471 93128 4

Printed and bound in Great Britain by Courier International Ltd, East Kilbride

AVAILABLE

The programs described in this book are available on 3½″ disk* for your IBM PC (and most compatibles). They can be compiled by the Turbo C++ and Borland C++ compilers. Your computer will also need a VGA graphics adaptor.

Order the Program Disk today, priced £15.00 (includes VAT)/$23.00 from your computer store, bookseller, or by using the order form below. (Prices correct at time of going to press.)

*5¼″ disks are available on request

Ammeraal: Programming Principles in Computer Graphics, Second Edition — Program Disk

Please send me ..... copies of the **Ammeraal: Programming Principles in Computer Graphics, Second Edition — Program Disk** at £15.00 (includes VAT)/$23.00 each.

0 471 93129 2

POSTAGE AND HANDLING FREE FOR CASH WITH ORDER OR PAYMENT BY CREDIT CARD

☐ Remittance enclosed ..... Allow approx. 14 days for delivery

☐ Please charge this order to my credit card (All orders subject to credit approval)

Delete as necessary:—AMERICAN EXPRESS, DINERS CLUB, BARCLAYCARD/VISA, ACCESS/MASTERCARD

CARD NUMBER ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiry date .....

☐ Please send me an invoice for prepayment. A small postage and handling charge will be made.

Software purchased for professional purposes is generally recognized as tax deductible.

NAME/ADDRESS .....

.....

.....

OFFICIAL ORDER NUMBER ..... SIGNATURE .....

If you have any queries please contact:

Helen Ramsey
John Wiley & Sons Limited
Baffins Lane
Chichester
West Sussex
PO19 1UD
England

Affix
stamp
here

Customer Service Department
John Wiley & Sons Limited
Shripney Road
Bognor Regis
West Sussex
PO22 9SA
England

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS *Second Edition*

Preface

This book is about mathematical and programming aspects of computer graphics. It is primarily intended for those who want to write graphics programs themselves and, in contrast to most other books on computer graphics, it is based on the C++ language. A sympathetic aspect of C++ is that it has so many good programming facilities in common with C; in other words, when programming in C++ we can deviate from C as much or as little as we like. The more spectacular new aspects of C++ are not always used in this book. Being unfamiliar with C++ will therefore not be a serious handicap, provided you can read C programs.

Besides switching from C to C++, there are some other new points in this edition. The important Bresenham algorithms for line and circle drawing are now included, and so is the subject of polygon filling. This is related to hidden-face elimination, which is also new in this edition. Emphasis is now more on *reusable program modules*. Most programs consist of an application module and several implementation modules, with corresponding header files as interfaces.

The file format for 3D objects used in this book is the same as that in *Interactive 3D Computer Graphics*, which is mainly about one program, D3D. The present book, on the other hand, is primarily intended as a textbook and therefore more general in two respects: it deals also with 2D graphics and, except for Appendix C, it is machine independent. Its first edition did very well in the first two or three years after its publication, but it badly needed a revision because all programs in it were in the old C style. Instead of only switching to the new style, recommended by ANSI and mandatory in C++, it seemed a good idea to me to add also some new subjects, such as those mentioned. This second edition will reduce the amount of paperwork involved in handing out supplementary lecture notes to my students. I hope that this will also apply to some colleagues. Any comments would be very welcome.

Leendert Ammeraal

Contents

1

Introduction

1.1 Graphics Programming and the C++ Language

It is hardly possible to find a subject that is more controversial than programming languages. Let us therefore consider some facts, rather than opinions. A discussion about programming languages used in computer graphics would be incomplete without mentioning Fortran, which was the dominant language for professionals for a very long time. In the eighties, many programmers switched from Fortran to the C language, which has many advantages, such as, for example, the possibility of recursion. About 1990, the C++ language became popular because of its facilities for object-oriented programming (OOP). Unlike some other OOP languages, C++ can also be used as a conventional language, that is, as 'a better C'. This is also the case with ANSI C, but this is still very tolerant with regard to old-style programming practice, such as calling functions that are not declared previously. By contrast, C++ requires the new style; it accepts calls to a function only if complete information about the parameters of that function is available. Now that very good C++ compilers are available, especially under Unix and on the IBM PC, it is to be expected that C++ will soon be widely regarded as the successor to C. Some convenient new aspects are

(1) operator definitions for user-defined types,
(2) constructors used to create objects of user-defined type,
(3) type-safe linkage.

Applying (1) and (2) can be regarded as extending the language to a new, more problem-oriented one. New types and operators are declared in header files, and they are implemented in separate modules. For example, using a properly defined vector type **vec**, we can write

```
vec v(2, 3), s;
s = v + vec(1, 0);  // s is equal to vec(3, 4)
```

to compute the sum of vector v and the unit vector (1, 0). This example illustrates the points (1) and (2): it shows that our own plus-operator can be applied to vectors and that a vector object can be easily created when its *x* and *y* components are given.

This example also shows a potential weakness of this approach. Computing the sum s of the two vectors v and (1, 0) will be more efficient if we write

```
s.x = v.x + 1;
s.y = v.y;
```

Not only does this prevent computing the sum v.y + 0, but it also avoids calling the *constructor* **vec** and copying 1 and 0 into a newly created **vec** object. We will therefore often use the more traditional (ANSI) C programming style.

Nevertheless, we will often benefit from the fact that we are using C++. For example, the good practice of declaring functions before they are used is (only) *recommended* in ANSI C, while it is *mandatory* in C++. As in C programs, such declarations of functions defined elsewhere are normally placed in header files, so that consistency is guaranteed. But even if we are inconsistent in this regard, we will obtain a linker error in C++ but not in C. For example, there is such an error if we declare and use function **f** as **void f(void)** in one module and define it as **void f(int)** in another. An error message from the linker is possible in C++ because of *type-safe linkage*, mentioned in point (3), which means that the linker is supplied with full information about parameter types.

1.2 Our First Graphics Programs

We will not discuss the C++ language here in a systematic way, so a book[1] on this language may be required if you are not yet familiar with it. On the other hand, many programs will be rather simple, so you may understand their meanings even if they contain some language constructs that are new to you. Here is our first C++ program. Except for the way comments are written, it is at the same time a C program:

```
// SQUARES: This program draws 50 squares inside
//          each other. To be linked with module GRSYS.
#include "grsys.h"

int main()
{  float xA, yA, xB, yB, xC, yC, xD, yD,
         xA1, yA1, xB1, yB1, xC1, yC1, xD1, yD1, p, q, r;
   int i;
```

[1] See *C++ for Programmers*, by the same author and from the same publisher as this book.

```
    q = 0.05; p = 1 - q; // q = lambda (see discussion below)
    initgr();
    r = 0.95 * r_max;
    xA = xD = x_center - r;
    xB = xC = x_center + r;
    yA = yB = y_center - r;
    yC = yD = y_center + r;
    for (i=0; i<50; i++)
    {  move(xA, yA);
       draw(xB, yB); draw(xC, yC);
       draw(xD, yD); draw(xA, yA);
       xA1=p*xA+q*xB; yA1=p*yA+q*yB;
       xB1=p*xB+q*xC; yB1=p*yB+q*yC;
       xC1=p*xC+q*xD; yC1=p*yC+q*yD;
       xD1=p*xD+q*xA; yD1=p*yD+q*yA;
       xA=xA1; xB=xB1; xC=xC1; xD=xD1;
       yA=yA1; yB=yB1; yC=yC1; yD=yD1;
    }
    endgr();
    return 0;
}
```

The output of this program is shown in Fig. 1.1. There are calls to four graphics functions, declared in the header file GRSYS.H:

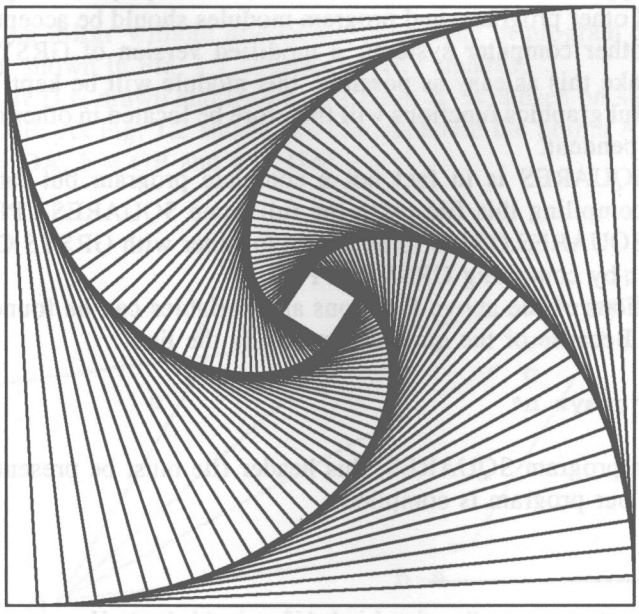


Fig. 1.1. Output of program SQUARES

initgr( )	initializes graphic output;
move(x, y)	moves a (real of fictitious) pen to point (x, y)[1];
draw(x, y)	draws a line segment from the current pen position to point (x, y);
endgr( )	performs any final actions (such as switching back from graphics mode to text mode) after a key has been pressed.

The following external variables are also declared in GRSYS.H; since function **initgr** assigns appropriate values to them, they should be used only after a call to that function. The first three of these seven variables are used in program SQUARES:

x_center, *y_center*	coordinates of screen center;
r_max	radius of largest possible circle on the screen;
x_min, *y_min*	coordinates of lower-left corner of the screen;
x_max, *y_max*	coordinates of upper-right corner of the screen.

A call to **initgr** is required before calls to the functions **move** and **draw**. Similarly, there must be a call to **endgr** after the final call to **move** or **draw**. The two calls **move(x, y)** and **draw(x, y)** have in common that they move a (real or fictitious) pen to point (x, y); with **move(x, y)** this pen is up and with **draw(x, y)** it is down.

The four functions just mentioned do not belong to the C++ language. They are external routines; after compilation of our program they are added to it by the linker. The definitions of the functions and variables mentioned above occur in a separate module, GRSYS.CPP, which is system dependent and therefore not included in this chapter. However, a version of this module for the IBM PC (and compatible machines) is included in Appendix C. This is intended to be the only system-dependent element in this book: all other programs and program modules should be accepted by any C++ compiler. For other computer systems, a modified version of GRSYS.CPP will be required. To make this as easy as possible, this module will be kept limited in size. Some other useful graphics functions will therefore be located in other modules, which are device-independent.

'Program' SQUARES is in fact not a complete program but rather a program *module*. After compiling this file (with the full name SQUARES.CPP), its resulting *object module* SQUARES.OBJ is to be linked together with GRSYS.OBJ, which was produced earlier by compiling GRSYS.CPP.

The declarations of the above functions and variables can be found in the header file GRSYS.H. Because of the quotation marks in the line

```
#include "grsys.h"
```

which occurs in program SQUARES, this header file must be present in the current directory when our program is compiled:

---

[1] Program fragments are normally printed in boldface in this book. However, program variables denoting numbers and standing on their own are printed in italics because they frequently also occur in mathematical expressions, in which boldface is used for vectors.

```
// GRSYS.H: Graphics primitives
extern float x_min, x_max, y_min, y_max, x_center, y_center,
             r_max;
void initgr(char *hpgfile=0);
void endgr(void);
void move(float x, float y);
void draw(float x, float y);
...
```

As you can see, function **initgr** has a default parameter **hpgfile**. We can use this to obtain our graphics output in a file, which can subsequently be imported by many text processors and desktop publishing packages. A suitable format for this file (used in the version of GRSYS.CPP listed in Appendix C) is HP-GL, originally designed for pen plotters but now also widely in use for other purposes. HP-GL files are *vector oriented*, which means that each line segment is identified by its two end points. Consequently, the quality of the final result on paper depends only on the printer or plotter that we are using, and not on the video display. We could instead have used a screen-capture utility (such as GRAB from WordPerfect), but then the file would be *bit oriented*, with a resolution based on the video display and resulting in a lower quality of the hard copy that we will eventually produce. If we want a file, say, SQUARES.HPG, of our set or squares, we can write

```
initgr("squares.hpg");
```

instead of calling **initgr** without arguments, as is done in program SQUARES. The graphics output shown in Fig. 1.1 consists of fifty squares.

A square ABCD is drawn and then a new point A′ is chosen on side AB such that AA′ = 0.05 × AB. As Fig. 1.2 shows, we can associate the points A, B and A′ with the vectors **a** = **OA**, **b** = **OB** and **a′** = **OA′**.

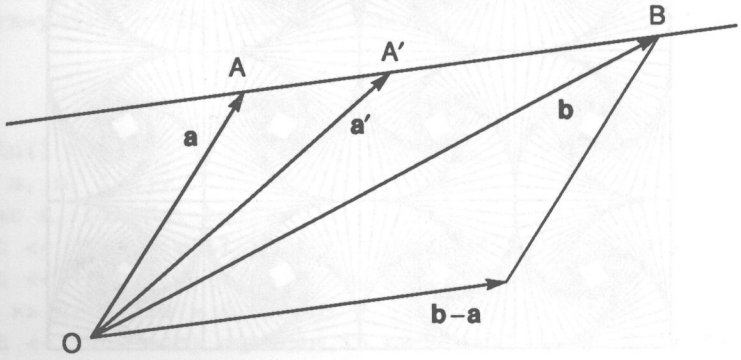


Fig. 1.2. Points and vectors

Then for any point A′ on the line AB the following vector equation applies:

$$\mathbf{a}' = \mathbf{a} + \lambda(\mathbf{b} - \mathbf{a})$$

which we can also write as

$$\mathbf{a}' = (1 - \lambda)\mathbf{a} + \lambda\mathbf{b}$$

In terms of coordinates, this is written as

$$x_{A'} = (1 - \lambda)x_A + \lambda x_B$$
$$y_{A'} = (1 - \lambda)y_A + \lambda y_B$$

Point A′ coincides with A if $\lambda = 0$, and with B if $\lambda = 1$. The value $\lambda = 0.05$ was used in program SQUARES to make point A′ lie near A on line segment AB. Points B′, C′ and D′ are chosen similarly on the sides BC, CD and DA, respectively. The procedure is then repeated with A′, B′, C′ and D′ as the new points A, B, C and D, respectively.

A generalized program for squares

Program SQUARES does not read any input data and can produce only one picture. It is normally desirable for graphics programs to be more general, so that we can use

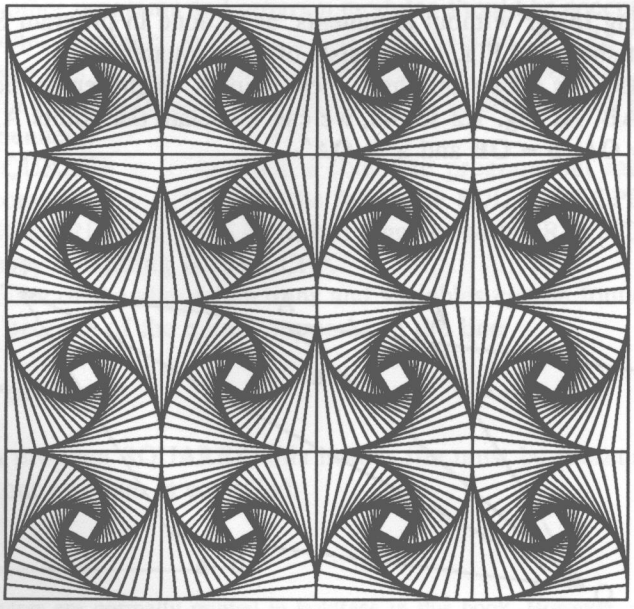


Fig. 1.3. Sample output of program MANYSQ

input data to specify details about what we want. In our example, we may as well draw more than one set of squares, say, $n \times n$ of them, arranged in a larger square, like the squares of a chessboard. We also want to use a variable number m of squares in each set, rather than exactly 50. Finally, we want to supply λ as input data, instead of always using $\lambda = 0.05$. Program MANYSQ is such a generalized version. Figure 1.3 shows an example of its output. It was obtained by using the input data $n = 4$, $m = 20$, and $\lambda = 0.9$. Actually, the $n \times n$ squares are divided into two types, like the black and white squares of a chessboard, by alternate use of the values λ and $1 - \lambda$ for **p** (each time **q** being equal to $1 - \mathbf{p}$).

If you are not yet familiar with C++, you may also find program MANYSQ instructive in that it shows how to display text on the screen and to read input data from the keyboard:

```
/* MANYSQ: This program draws n x n sets of squares, arranged
           as on a chessboard. To be linked with module GRSYS.
*/
#include <iostream.h>
#include "grsys.h"
void set_of_squares(float xA, float yA, int m,
                    float p, float a)
{  float xB=xA+a, yB=yA, xC=xB, yC=yA+a, xD=xA, yD=yC,
   xA1, yA1, xB1, yB1, xC1, yC1, xD1, yD1, q=1-p;
   int i;
   for (i=0; i<m; i++)
   {  move(xA, yA);
      draw(xB, yB); draw(xC, yC);
      draw(xD, yD); draw(xA, yA);
      xA1=p*xA+q*xB; yA1=p*yA+q*yB;
      xB1=p*xB+q*xC; yB1=p*yB+q*yC;
      xC1=p*xC+q*xD; yC1=p*yC+q*yD;
      xD1=p*xD+q*xA; yD1=p*yD+q*yA;
      xA=xA1; xB=xB1; xC=xC1; xD=xD1;
      yA=yA1; yB=yB1; yC=yC1; yD=yD1;
   }
}

int main()
{  int m, n, i, j;
   float a, lambda, halfn;
   cout << "There will be n x n sets of squares.\n";
   cout << "Enter n (e.g. 8 for a chessboard): ";
   cin >> n; halfn = 0.5 * n;
   cout << "How many squares in each set? (e.g. 10): ";
   cin >> m;
   cout << "Enter interpolation factor between 0 and 1" ;
```