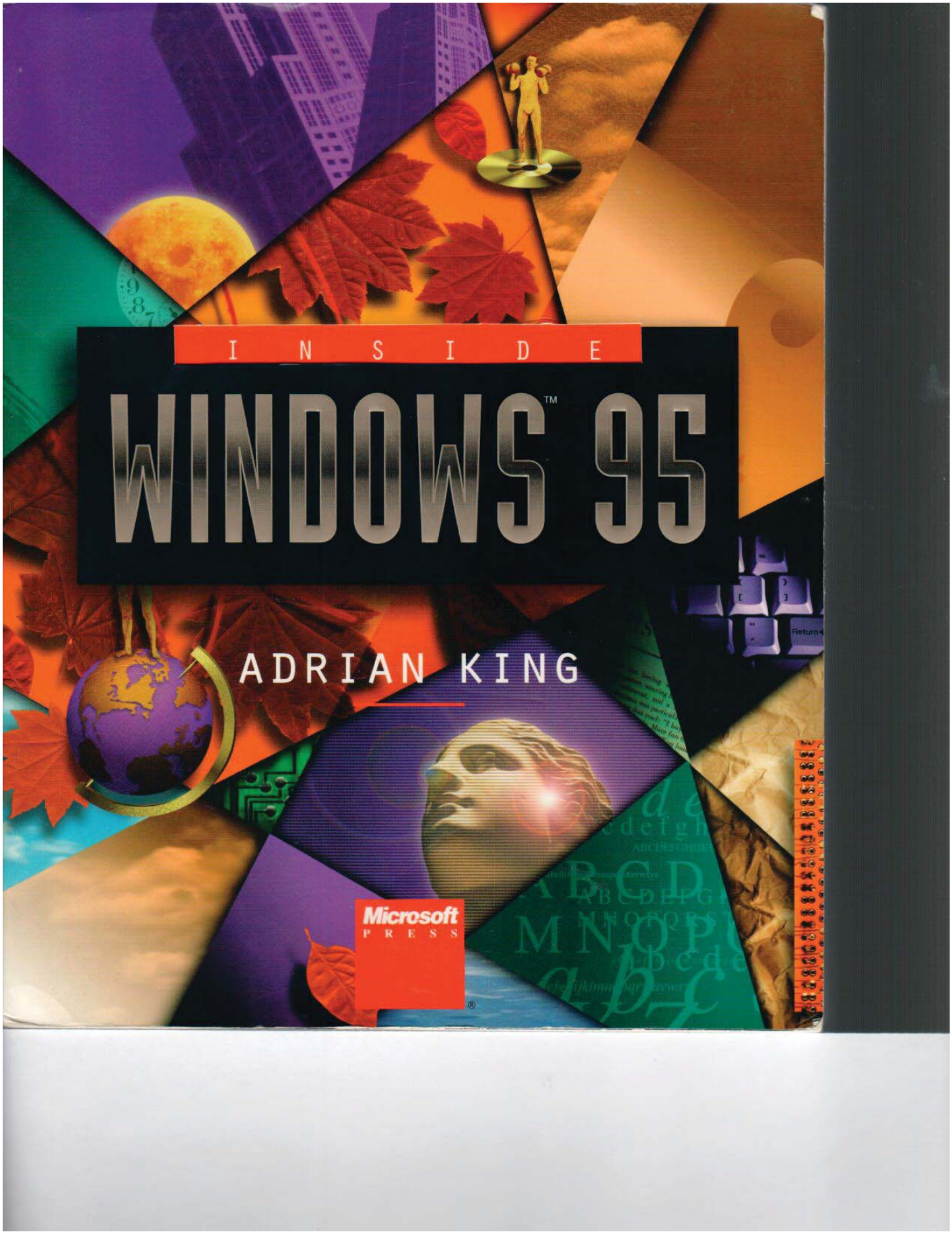

I N S I D E

WINDOWS™ 95

ADRIAN KING

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
King, Adrian, 1953-
Inside Windows 95 / Adrian King.
p. cm.
Includes index.
ISBN 1-55615-626-X
1. Windows (Computer programs) 2. Microsoft Windows (Computer file) I. Title.
QA76.76.W56K56 1994
005.4'469--dc20 93-48485
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 9 8 7 6 5 4

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

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

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

Acquisitions Editor: Mike Halvorson
Project Editor: Erin O'Connor
Technical Editors: Seth McEvoy and Dail Magee, Jr.

the event of a memory access fault when the code scans past the end of the allocated memory buffer (with the pattern not found). Neither example tests for this condition, and in the first case you'd get a program failure with little useful qualifying information.

Exception handlers are frame based, meaning that their scopes nest just as declaration scopes do, so it's possible to handle errors on either a global or a local basis. There are also facilities for specifying the context in which the exception is handled.[24] The structured exception handling feature also allows a program to initiate an exception (the *RaiseException()* API) and specifies the protocol for interacting with a debugging tool if one is in use. Within an *except* block, you can determine the cause of an exception so that you can carry out appropriate error recovery. You shouldn't replace every error test in your code with an exception sequence, but it is a great way to manage a multitude of possible error conditions diligently and efficiently. After all, how many times do you test for every possible error in your code?

The Graphics Device Interface

GDI is the heart of the Windows graphics capabilities. All of the drawing functions for lines and shapes are in GDI as well as the color management and font handling functions. Many aspects of Windows performance are tied closely to GDI performance, and a lot of the GDI code is handcrafted 386 assembly language. At the application level, Windows provides logical objects known as *device contexts* (DCs) that describe the current state of a particular GDI drawing target. A DC can describe any output device or representation of a device. An application will obtain a DC for printer output or for completely memory-based operations, for example. Applications manage DCs by means of Win32 APIs only. The actual DC data structure is always hidden from the application. At any instant a DC contains information about objects such as the current pen (for drawing lines), the current brush (for filling regions), the color selection, and the location and dimensions of the logical drawing target.

The key to the use of Windows and Windows applications on a widely disparate range of target hardware is the device independence

24. Reminiscent of, but much better than, the C language *setjmp()/longjmp()* facility.

embodied in the Windows API. An application uses DCs and other logical objects when calling GDI functions. It never writes data directly to an output device. GDI itself manages the process of transforming the data into a format suitable for use by a particular device driver, and the driver handles the task of placing a representation of the request on the output device. For example, an application may call the system asking for its main window to be repainted. During the repainting operation, among many other requests, GDI may tell the driver "on the screen draw a one-pixel-wide black line from position (0, 48) to position (639, 48)." If the device—a dot matrix printer, say—can't perform operations such as line drawing, GDI will break the request down into simpler operations. The device driver will receive a series of calls telling it to draw individual dots, for example. This architecture frees applications from device-dependent problems and allows Windows to make use of even the simplest hardware as an output device.

With this device-independent capability come several problems. In addition to simply choosing and managing an appropriate device-independent representation of all the graphics objects, you need to have a plethora of device drivers available to interface GDI to the target hardware. Issues such as handling complex fonts through a range of point sizes and then being able to draw the font legibly on both a 1024 by 768 pixel display screen and a simple dot matrix printer involve many complex algorithms and a lot of very clever code.

Over successive releases of Windows, the capabilities of GDI have improved considerably, and the underlying structure of the system has adapted to the experience gained from earlier versions and to the prevailing market forces. The vast majority of Windows users nowadays tend to have fairly capable hardware: VGA displays and laser or high-resolution dot matrix printers. The hardware will probably get even more powerful, with higher resolution and color-capable devices abounding. It's therefore important to get the best possible performance out of a few core components rather than expend effort on hundreds of device drivers, each with a limited installed base. It has also been important to look ahead at the likely effects of hardware trends. Two of the major changes in the Windows 95 GDI subsystem reflect hardware trends: the *device-independent bitmap* (DIB) engine and the *image color matching* (ICM) subsystem.

Windows 3.1 successfully introduced the concept of the *universal printer driver*—a device driver that does much of the work for all the other system printer drivers. The so called *printer mini-drivers* support

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.