;login:
The USENIX Association Newsletter

Volume 10, Number 4  October/November 1985

CONTENTS

USENIX Winter '86 Conference ......................................................................................... 3
Schedule of Events ............................................................................................................. 3
Registration Fees .............................................................................................................. 4
Pre-Registration Mailing ................................................................................................. 4
Technical Session A – Window Environments and UNIX .................................................. 5
Wednesday's Tutorials ...................................................................................................... 6
Technical Session B – UNIX on Big Iron .......................................................................... 8
Thursday's Tutorials .......................................................................................................... 8
Technical Session C – Ada and the UNIX System ............................................................. 10
Friday's Tutorials ............................................................................................................. 11
Second Workshop on Computer Graphics ...................................................................... 13
Future Meetings and Workshops ..................................................................................... 14
Nominations for Election of Officers and Directors of USENIX Association .................. 16
Report on the IEEE P1003 Portable Operating Systems Environment Committee ....... 16
Announcing Third Printing of USENIX 4.2BSD Manuals .................................................. 18
4.2BSD Manual Reproduction Authorization and Order Form ....................................... 19
Local User Groups ......................................................................................................... 20
Publications Available .................................................................................................... 21
USENIX Services ............................................................................................................ 22

Papers from the 1984 USENIX Graphics Workshop ....................................................... 22
Window Managers are Operating Systems ..................................................................... 23
  S. McGeady
Software Architecture for Animation Systems ............................................................... 35
  Roy Hall
A UNIX Image Production Pipeline .................................................................................. 40
  Julian E. Gomez
Patchwork: A Dataflow Model for Efficient Graphics Programming ............................ 43
  Ronen Barzel and David Salesin
The Alpha_1 Computer-Aided Geometric Design System in the UNIX Environment ........ 54
  Spencer W. Thomas
Principles of Structured Font Design for the Personal Workstation .............................. 65
  Charles Bigelow

The closing date for submissions for the next issue of ;login: is November 27, 1985
NOTICE

;login: is the official newsletter of the USENIX Association, and is sent free of charge to all members of the Association.

The USENIX Association is an organization of AT&T licensees, sub-licensees, and other persons formed for the purpose of exchanging information and ideas about UNIX\(^1\) and UNIX-like operating systems and the C programming language. It is a non-profit corporation incorporated under the laws of the State of Delaware. The officers of the Association are:

- President: Alan G. Nemeth
- Vice-President: Deborah K. Scherrer
- Secretary: Lewis A. Law
- Treasurer: Waldo M. Wedel
- Directors: Thomas Ferrin, Steve C. Johnson, Lou Katz, Michael D. Tilson
- Executive Director: James E. Ferguson

The editorial staff of ;login: is:

- Publisher: James E. Ferguson
- Copy Editor: Michelle Peetz
- Managing Editor: Tom Strong
- Editorial Director: Lou Katz

Member Services

Member services are provided through the Association office. Membership information can be obtained from the office:

USENIX Association
P.O. Box 7
El Cerrito, CA 94530
(415) 528-8649
{ucbvax,decvax}usenix|office

Contributions Solicited

Members of the UNIX community are heartily encouraged to contribute articles and suggestions for ;login:. Your contributions may be sent to the editors electronically at the addresses above or through the U.S. mail to the Association office. The USENIX Association reserves the right to edit submitted material.

;login: is produced on UNIX systems using troff and a variation of the –me macros. We appreciate receiving your contributions in n/troff input format, using any macro package. If you contribute hardcopy articles please leave left and right margins of 1½" and a top margin of 1½" and a bottom margin of 1¼". Hardcopy output from a line printer or most dot-matrix printers is not reproducible.

Acknowledgments

The Association uses a VAX\(^2\) 11/730 donated by the Digital Equipment Corporation for support of office and membership functions, preparation of ;login:, and other association activities. It runs 4.2BSD, which was contributed, installed, and is maintained by mt Xinu. The VAX uses a sixteen line VMZ-32 terminal multiplexer donated by Able Computer of Irvine, California.

Connected to the VAX is a QMS Lasergrafix\(^*\) 800 Printer System donated by Quality Micro Systems of Mobile, Alabama. It is used for general printing and draft production of ;login: with ditroff software provided by mt Xinu.

This newsletter may contain information covered by one or more licenses, copyrights, and/or non-disclosure agreements. No reproduction of this newsletter in its entirety or in part may be made without written permission of the Association.

\(^1\)UNIX is a trademark of AT&T Bell Laboratories.
\(^2\)VAX is a trademark of Digital Equipment Corporation.
\(^*\)Lasergrafix is a trademark of Quality Micro Systems.
USENIX Winter '86 Conference
January 15-16-17, 1986
Marriott Hotel – City Center
Denver, Colorado

The USENIX Winter '86 Conference will consist of three workshop-oriented technical sessions accompanied by fourteen full-day tutorials. Each of the technical sessions will be a full day devoted to only one topic. For each topic area, there will be related tutorials on adjacent days, concurrent with the other technical sessions. There will also be several tutorials of general interest. The topics of the technical sessions are:

- Window Environments and UNIX – Wednesday, January 15
- UNIX on Big Iron – Thursday, January 16
- Ada® and the UNIX System – Friday, January 17

You may attend the conference for one, two, or all three days, but only one technical sessions or tutorial per day. A full description of each technical session and tutorial appears below.

The Conference Host for this meeting is the Computer Science Department of the University of Colorado at Boulder. Evi Nemeth is the local coordinator. All events are being held at the Marriott Hotel – City Center in Denver.

There is excellent downhill and cross-country skiing available near Denver. Public transportation is available to most areas. Arrangements are being considered for both day and weekend trips, both before and after the meeting. Further information will be posted to net.usenix and net.ski and will be available at the meeting.

Schedule of Events

Wednesday, January 15, 1986, 9am–5pm

<table>
<thead>
<tr>
<th>Technical Session A</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial # 1</td>
<td>Window Environments and UNIX</td>
</tr>
<tr>
<td>Tutorial # 2</td>
<td>Design Considerations for SNA Communications Under UNIX</td>
</tr>
<tr>
<td>Tutorial # 3</td>
<td>UNIX Device Driver Design (4.2BSD)</td>
</tr>
<tr>
<td>Tutorial # 4</td>
<td>System V Interprocess Communication Application Programming</td>
</tr>
<tr>
<td>Tutorial # 5</td>
<td>Ada – From The Top; An Introduction</td>
</tr>
<tr>
<td>Tutorial # 6</td>
<td>UNIX System V Internals</td>
</tr>
</tbody>
</table>

Thursday, January 16, 1986, 9am–5pm

<table>
<thead>
<tr>
<th>Technical Session B</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial # 6</td>
<td>UNIX on Big Iron</td>
</tr>
<tr>
<td>Tutorial # 7</td>
<td>Introduction to 4.2BSD Internals</td>
</tr>
<tr>
<td>Tutorial # 8</td>
<td>Windowing System Implementations</td>
</tr>
<tr>
<td>Tutorial # 9</td>
<td>Language Construction Tools on the UNIX System</td>
</tr>
<tr>
<td>Tutorial # 10</td>
<td>Advanced C Programming</td>
</tr>
</tbody>
</table>

Friday, January 17, 1986, 9am–5pm

<table>
<thead>
<tr>
<th>Technical Session C</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial # 10</td>
<td>Ada and the UNIX System</td>
</tr>
<tr>
<td>Tutorial # 11</td>
<td>Advanced Topics on 4.3BSD Internals</td>
</tr>
<tr>
<td>Tutorial # 12</td>
<td>UNIX Networking</td>
</tr>
<tr>
<td>Tutorial # 13</td>
<td>Managing a Local Area Network</td>
</tr>
<tr>
<td>Tutorial # 14</td>
<td>Introduction to UNIX Systems Administration</td>
</tr>
<tr>
<td>Tutorial # 15</td>
<td>Writing Portable C Programs</td>
</tr>
</tbody>
</table>
Registration Fees

Registration fees are based on a per day rate. You may attend the conference for one, two, or three days, choosing either a technical session or a tutorial class on each day. It is not possible to attend more than one topic per day. A copy of the conference proceedings will be given to each person registering for at least one technical session. The proceedings will also be for sale at the conference.

<table>
<thead>
<tr>
<th>Membership Status</th>
<th>Number of Days Attending Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>one day</td>
</tr>
<tr>
<td>USENIX Member</td>
<td></td>
</tr>
<tr>
<td>Pre-registered – postmarked no later than 12/27/85</td>
<td>$150</td>
</tr>
<tr>
<td>On-site or postmarked after 12/27/85</td>
<td>$200</td>
</tr>
<tr>
<td>Non-Member</td>
<td></td>
</tr>
<tr>
<td>Pre-registered – postmarked no later than 12/27/85</td>
<td>$180</td>
</tr>
<tr>
<td>On-site or postmarked after 12/27/85</td>
<td>$230</td>
</tr>
<tr>
<td>Student</td>
<td></td>
</tr>
<tr>
<td>Pre-registered or on-site (Must enclose photocopy of current student ID card with registration)</td>
<td>$75</td>
</tr>
</tbody>
</table>

For pre-registration rates, registration forms must be received with full payment and postmarked no later than December 27, 1985. Visa, Mastercard, and American Express are accepted for registration.

Pre-Registration Mailing

Pre-registration materials containing detailed conference information along with registration and hotel reservation forms will be mailed in the middle of November to USENIX members and previous conference attendees. If you would like to be added to the mailing list, please contact:

USENIX Conference Office
P.O. Box 385
Sunset Beach, CA 90742
(213) 592-3243 or 592-1381

Hotel Registration Deadline – December 24, 1985
Pre-Registration Deadline – December 27, 1985
Winter '86 Conference Schedule – Wednesday, January 15

Technical Session A – Window Environments and UNIX

Program Committee: Sam Leffler, Lucasfilm, Chair
Mike Hawley, The Droid Works, Co-Chair
Jim Gettys, Digital Equipment Corp.
James Gosling, Sun Microsystems
Rob Pike, Bell Laboratories

This meeting will explore the design and integration of UNIX-based window systems and their applications. Three sessions and a panel discussion have been organized.

The following presentations are scheduled.

Hardware and Hardware Issues:
Galadriel: A Display List-Based Window Manager
Bob Lewis, Tektronix

Next Generation Hardware for Windowed Displays
Steven McGeady

Real-Time Resource Sharing for Graphics Workstations
Mark Grossman & Glen Williams, Silicon Graphics, Inc.

Applications:
Third Level Software Tools, CDEC Enhancements to Andrew
Thomas Neundorff, Carnegie-Mellon University

A Workstation-Based In-patient Clinical System in the Johns Hopkins Hospital
Stephen N. Kahane, et al, Johns Hopkins Hospital

The Feel of Pi
T. A. Cargill, AT&T Bell Laboratories

Systems and System Issues:
FLAMINGO: A UNIX-Based, Object-Oriented User Interface Management System

A Proposal for Interwindow Communication and Translation Facilities
D. P. Gill, Exxon Research and Engineering

Problems While Implementing Window Systems in UNIX
Jim Gettys & Bob Scheifler, Massachusetts Institute of Technology

A Distributed and Extensible Window System
James Gosling, Sun Microsystems

Panel Discussion:
Color? Do we need it? How can we use it? How do we deal with it? . . .
Wednesday’s Tutorials

Tutorial #1: Design Considerations for SNA Communications Under UNIX
Instructor: Daniel Fisher
System Strategies, Inc.

One of the major design considerations a developer must address when integrating SNA or other communications protocols into a UNIX-based system is the distribution of software system components. This presentation will define the three basic options open to the developer for placement of software components: utilization of user space; integration at the kernel level; and board level integration. The relative advantages and disadvantages of each alternative will be discussed in detail. Sample implementations of a SNA/3270 emulation and a X.25 emulation on UNIX-based systems will then be presented, followed by a discussion of the concept of streams implementation currently under development at AT&T Information Systems and its future implications as they relate to UNIX users.

System Strategies, Inc., has developed several major communication packages that provide compatibility with IBM's standard network protocols, SNA and BSC. It provides portable 3270 SNA and BSC packages under UNIX. Daniel Fisher is a Senior Software Engineer in the product development group at Systems Strategies. He has developed and implemented several UNIX-based communications systems.

Tutorial #2: UNIX Device Driver Design (4.2BSD)
Instructor: Daniel Klein
Consultant

4.2BSD SOURCE LICENSE REQUIRED FOR THIS TUTORIAL

This course is designed for people who wish to become familiar with the fundamentals of designing UNIX device drivers. A knowledge of the major structures and internals of 4.2BSD UNIX is a desirable prerequisite to this tutorial, although a specific knowledge of the finer details is not required. This seminar will cover the major aspects of driver design, implementation, and device integration. Both DMA and programmed I/O device drivers will be covered, as well as block and character (buffered and unbuffered) interfaces. We will outline the design and implementation of structured I/O devices (i.e. disk drives), and semi-structured devices (i.e. tape drives and serial communication links). This course will also discuss all aspects of adding a new device to the kernel (i.e. autoconfiguration, special files, device tables, and debugging). The intended audience for this course is systems programmers who will be actively engaged in the maintenance or design and implementation of UNIX device drivers. Although this course will be geared towards 4.2BSD, a comparison between the Berkeley and Bell Labs approaches will be offered. Users of System III or System V will therefore also find this course to be informative.

Daniel Klein has been involved with UNIX since the original university distribution of Version 6 in 1976, including writing device drivers, utility programs, applications systems, and enhancements to the kernel. A graduate of Carnegie-Mellon University, Mr. Klein was manager of software systems at Mellon Institute for six years. He is presently engaged in teaching UNIX internals and developing an on-line educational system for UNIX, as well as developing a multi-processor simulation system.
Tutorial # 3: System V Interprocess Communication Application Programming
Instructor: Jon H. LaBadie
AUXCO

This tutorial will be targeted toward application programmers who wish to know how and why
to use the interprocess communication (IPC) facilities described in the UNIX System V Interface
Specification. Syntax and examples of the use of the seven types of IPC facilities will be discussed.
The facilities to be covered include: semaphores, message queues, shared memory, named and
unnamed pipes, signals, and process waiting and exit status.

Jon LaBadie’s UNIX involvement spans the past seven years during which time he has designed
and implemented a number of graphics and database applications using UNIX and C. For the past
three years, he has served as a consultant to the UNIX training group at AT&T where he has
developed and taught numerous UNIX and C courses. He holds a B.S. degree in biology from Drexel
University, and a Ph.D. in Biochemistry from Pennsylvania State University.

Tutorial # 4: Ada – From The Top; An Introduction
Instructor: Putnam P. Texel
Texel & Company

This tutorial is designed for those individuals familiar with a procedure oriented high order
language, but do not have much familiarity with Ada. The level of the tutorial is applicable for
managers who need to understand Ada concepts and software engineers taking their first look at this
most powerful and expressive language.

Texel & Company specializes in Ada consulting, Ada education, and Ada R&D with clients in
government and industry. Ms. Texel has presented tutorials at both SIGAda and AdaUG confer-
cences, has taught Ada at the graduate level at Monmouth College, and has been an invited panelist on
several panels dealing with Ada education. She is Chairperson of the Education Committee of ACM
SIGAda and Chairperson of the ACM Princeton Chapter Local SIGAda.

Tutorial # 5: UNIX System V Internals
Instructor: Maury Bach & Steve Buroff
AT&T Information Systems

SYSTEM V SOURCE LICENSE REQUIRED FOR THIS TUTORIAL

This tutorial is a survey of the internal structure of AT&T’s UNIX System V, and it is intended
for people who maintain, modify, or port UNIX systems. The tutorial will discuss existing UNIX ker-
nel concepts such as I/O system, file system, and process and memory management, as well as new
features to be included in the next release of System V, such as the file system switch, streams, remote
file sharing, and shared libraries. Attendees should have a good working knowledge of the UNIX
system; basic kernel knowledge is recommended.

Maury Bach and Steve Buroff are members of the development staff at AT&T Information Sys-
tems. Maury Bach has worked on multi-processor UNIX development, and Steve Buroff has worked
on paging virtual memory implementation. Bach also has taught a multi-week UNIX internals course
within Bell Labs in order to train Bell Labs systems programmers. This one day tutorial draws upon
this work.
Technical Session B – UNIX on Big Iron

Program Committee: Peter Capek, IBM Research, Chair
Jim Lipkis, New York University, Courant Institute
Eugene Miya, NASA Ames Research Center

During this session the use of UNIX on two new classes of systems – very large single processor machines such as the Cray-1 and systems with many processors such as the Alliant – will be discussed. Topics which will be included are:

- Implementation of UNIX in these environments
- Operational and performance issues
- Interaction between the basic design of UNIX and these environments
- Using the UNIX environment on large systems for application production

The following presentations are scheduled.
Concentrix-UNIX for the Alliant Multi-processor
   Jack Test, Alliant Computer Systems
Experience Porting System V to the Cray-2
   Timothy W. Hoel, Cray Research
Implementation of 4.2BSD on a High Performance Multi-processor
   Dave Frobert, et al, Culler Scientific Systems Corp.
Porting UNIX to the System/370 Extended Architecture
   Joseph R. Eykholt, Amdahl Corp.

Thursday’s Tutorials

Tutorial # 6: Introduction to 4.2BSD Internals
Instructor: Thomas W. Doeppner, Jr.
Brown University

4.2BSD SOURCE LICENSE REQUIRED FOR THIS TUTORIAL

This tutorial is geared to the programmer with a good knowledge of UNIX programming in C, but with little or no experience with UNIX internals. The course will cover process management, high-level I/O (including the file system), low-level I/O (i.e., device drivers), virtual memory, interprocess communication, and networking. After taking the tutorial, the individual will have a basic knowledge of the structure of 4.2BSD and should be able make his or her way through kernel code.

Thomas W. Doeppner, Jr. received his Ph.D. in Computer Science from Princeton University in 1977 and has been on the faculty at Brown University since 1976. He has lectured extensively on UNIX internals over the past two years for the Institute for Advanced Professional Studies.

Tutorial # 7: Windowing System Implementations
Instructor: David Rosenthal
Sun Microsystems, Inc.

The tutorial is intended for developers of UNIX window manager applications. It will survey the range of current window systems, concentrating on the programmer's interface, the imaging model, and their support for interaction techniques. Familiarity with C and the UNIX programming environment will be assumed.
Winter '86 Conference Schedule – Thursday, January 16  (Continued)

David Rosenthal has been researching interactive graphics and user interfaces since 1968. He co-chaired the technical review of GKS, and until recently was Associate Director of the Information Technology Center at Carnegie-Mellon University. He was one of the developers of ITC's Andrew portable window system.

Tutorial # 8: Language Construction Tools on the UNIX System
Instructor:  Stephen C. Johnson
AT&T Bell Laboratories

This tutorial is intended for C programmers who want to become familiar with the language development tools available on the UNIX system. The course will be directed towards application designers who may wish to use these tools to make front ends for their applications, rather than towards "traditional" compiler writing. Specific topics covered include: designing a language recognizer, the lex and yacc programs, symbol table issues, error reporting and recovery, strong type checking, and testing. Several in-class exercises will be given to lead the students through the construction of a simple front end.

Steve Johnson received his Ph.D. degree in pure mathematics from Columbia University in 1968. In 1967, he joined Bell Laboratories, Murray Hill, N.J., where he worked in psychometrics, computer music, and the computation center before joining the Computer Science Research Department. As a researcher, he worked on computer algebra, wrote the yacc parser generator, contributed to complexity theory and the theory of code generation and parsing, wrote the Portable C Compiler, and, for several years, was involved in experimental VLSI design and silicon compilation. He is now head of the Language Development Department in AT&T Information Systems.

Tutorial # 9: Advanced C Programming
Instructor:  William C. Steward
AUXCO

This seminar will be directed toward applications programmers with at least six months experience in the C language. Its focus is on the theories behind C language syntax, which will be illustrated with programming examples. The topics for discussion include: multi-dimensional arrays, pointers to arrays, structures and pointers to structures, pointers to functions, and dynamic memory allocation and linked lists.

William Steward has developed and taught introductory and advanced topics in UNIX and C programming. Formerly with a major research institute, Mr. Steward has extensive experience in presenting UNIX related seminars.
Technical Session C – Ada and the UNIX System

Program Committee:  Charles Wetherell, AT&T Information Systems, Chair
                     Donn Milton, Verdix Corp.
                     Tucker Taft, Intermetrics
                     Larry Yelowitz, Ford Aerospace

The Ada and UNIX session will educate and inform UNIX users about the new programming language Ada, and Ada mavens about the UNIX system. The preliminary agenda includes an introduction to the world of Ada with projections about Ada’s future growth, a demonstration of a large real-time Ada application running under UNIX, and a variety of technical papers on the use of Ada with UNIX systems. No prior experience with Ada is required; talks and presentations will help you understand Ada itself and its relation to UNIX programming and environments.

The technical papers will include talks comparing C and Ada under UNIX systems, approaches to providing the standard Ada system interfaces to UNIX systems, Ada runtime systems implemented in UNIX environments, revision control and library management systems specially adjusted for the problems of Ada programs, and several common UNIX facilities re-implemented in Ada. Both European and American authors will be represented. Papers will stress the problems and opportunities UNIX systems provide for Ada environments and may help you see some UNIX capabilities.

The following presentations are scheduled.

Keynote Address
Speaker to be announced

Real-Time Demonstration
Details to be announced

UNIX, C and Ada
Herman Fischer, Mark V Business Systems

Managing Separate Compilation in the AT&T Ada Translator System
G. W. Elsesser, M. S. Safran & T. Tieger, AT&T Information Systems

Revision Control Tools and the Ada Program Library
Dick Schefstrom, TeleLOGIC AB

UNIX System and CAIS
Rebecca Bowerman, Mitre Corp.

SVID as an Interim Basis for CAIS
Herman Fischer, Mark V Business Systems

Implemented Curses in Ada
Karl Nyberg, Verdiix Corp.

An Overview of the Ada Shell
Lisa Campbell & Mark Campbell, NCR Corp.

Targeting Ada to 68000/UNIX
Mitchell Gart, Alsins Inc.
Friday's Tutorials

Tutorial # 10: Advanced Topics on 4.3BSD Internals
Instructor: Mike Karels & Marshall Kirk McKusick
University of California, Berkeley

4.2BSD SOURCE LICENSE REQUIRED FOR THIS TUTORIAL

This tutorial is directed to systems programmers who have taken a course on 4.2BSD internals or who have had at least a year of experience working on the 4.2BSD kernel. The tutorial will cover the performance work done for 4.3BSD and will also discuss recent and planned changes to the kernel interfaces and facilities. The intent of the tutorial is to present a wide variety of material at a descriptive level. Presentations will emphasize code organization, data structures, and algorithms.

Mike Karels received his B.S. in Microbiology at the University of Notre Dame. While a graduate student at the University of California, he was the major contributor to the 2.9BSD release of the Berkeley Software Distribution for PDP-11's. He currently is the Principal Programmer at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley UNIX.

Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4.2BSD fast file system and was involved in implementing the Berkeley Pascal system. He currently is the Research Computer Scientist at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley UNIX.

Tutorial # 11: UNIX Networking
Instructor: Bruce Borden
Silicon Graphics, Inc.

Local area networks (LANs) are slowly finding mass acceptance with the entry of IBM into the competition. In October or November, industry analysts expect IBM to announce yet another LAN, probably based on the 802.5 standard but with unknown protocol software. This tutorial will cover the 802.X standards and the most common protocols in use today and expected for the future. Sun's NFS and AT&T's distributed file systems will be compared. Berkeley's Socket implementation will be compared with AT&T's Streams. This tutorial is directed at experienced UNIX users and developers with knowledge of Ethernet (802.3) and IP/TCP. It is not an introductory tutorial, and will not teach attendees how to program Berkeley 4.X networks.

Bruce Borden is Director of Engineering for Silicon Graphics, Inc., developing very high performance 3-D color graphics workstations running UNIX System V. Prior to SGI, Bruce was a founder of 3Com, developed the Excelan TCP/IP front-end protocol package, and authored the Rand MH mail handling System.

Tutorial # 12: Managing a Local Area Network
Instructor: Evi Nemeth & Andy Rudoff
University of Colorado, Boulder

This tutorial is a summary of all the things we (and many others) have learned over the past couple of years in managing a growing local area network. It is intended for system administrators and others involved in planning, configuring, installing, and maintaining a networked UNIX facility. The tutorial emphasizes 4.2/4.3BSD networks, yet includes issues that are global to all networks.
Topics to be covered are: building the network including hardware/software installation; global management schemes including source code management, logir management, resource management; distributed tools to make these chores easier; security; accounting; heterogeneous hardware (IBMs and others); other protocols (non TCP/IP).

Evi Nemeth is on the Computer Science faculty at the University of Colorado and has led the growth of the University's Engineering Research Computing Facility from a single VAX 11/780 to its present complement of 20 machines that include six different manufacturers' hardware.

Andy Rudoff is a Computer Science student who has been involved with the systems work concerning this network's growth (hoeing, harvesting, weeding, etc.) for the past four years.

Tutorial #13: Introduction to UNIX Systems Administration
Instructor:  Ed Gould
mt Xinu

The basics of administering a UNIX system will be covered. The tutorial will be oriented mainly towards Berkeley VAX UNIX systems, but the principles, and some of the examples, will apply to System V, 2.9BSD, and other systems. Topics covered will include system startup and shutdown, resource management, performance and tuning, the UNIX file system, and security, as well as others. The tutorial is designed for system administrators, not for systems programmers. A rudimentary knowledge of UNIX is assumed.

Ed Gould has been working with UNIX since 1976. At the Computer Center at the University of California in Berkeley he was involved with the management and administration of several systems that were used for general purpose timesharing for the campus. In 1983, along with Vance Vaughan and Bob Kridle, he founded mt Xinu, a company dedicated to the support and enhancement of technically advanced UNIX systems.

Tutorial #14: Writing Portable C Programs
Instructor:  Tom Plum
Plum Hall, Inc.

Today, the C programming language is widely used to implement portable applications programs. But there are many pitfalls for the unwary, some obvious, but some very subtle. If you are not aware of the issues, it is easy to write programs that will not operate correctly on another hardware architecture, or another UNIX version, or another version of the C compiler. It then becomes expensive to move the application to a new machine. This course will teach you to recognize the trouble spots and avoid these pitfalls. You will learn to write truly machine – and system – independent code, and to protect yourself when this is not possible. This course is intended for experienced C application developers. If you are involved in the development of software which is to be used or distributed on a variety of systems, you should take this course.

Tom Plum is chairman of Plum Hall, Inc., a publishing and training firm specializing in the C language. He is the author of two textbooks on C. He is also vice-chair of the ANSI X3J11 C Language Standards Committee.
Second Workshop on Computer Graphics

December 12-13, 1985
Doubletree Hotel
Monterey, California

Program Committee: Reidar Bornholdt, Columbia University, Chair
Tom Duff, AT&T Bell Laboratories
Lou Katz, Networked Picture Systems
Peter Langston, Bell Communications Research

USENIX is sponsoring a limited enrollment workshop on current and future developments in interactive computer graphics. The workshop will be structured to facilitate in-depth discussions of technical issues, and will have presentations in a number of formats, with ample time for questions and responses. It is expected that attendees will participate actively in the program. There will be an evening computer graphics film and video presentation.

The following presentations are scheduled.

A Quick and Dirty Algorithm for Animations
Jon Bentley, AT&T Bell Laboratories

Dogzilla: A Battle Computer for Dogfight
Greg Chesson, Silicon Graphics, Inc.

If the Earth is Round, Why is the Sun Square
James Gosling, Sun Microsystems, Inc.

MacMix: Mixing Music with a Mouse
Adrian Freed

A Dataflow Environment for Interactive Graphics
Paul Haebeler, Silicon Graphics, Inc.

Scattered Thoughts on Color
Roy Hall

Algorithms for Anti-aliased Rendering
Tom Duff, AT&T Bell Laboratories

Constructing Uniform Polyhedra
Andrew Hume, AT&T Bell Laboratories

A Modular Rendering and Modeling System
Carlo H. Sequin, Univ. of California at Berkeley

A Low Cost Graphics Workstation
Spencer Thomas, Univ. of Utah

Image Synthesis on Personal Computers
Turner Whitted, Univ. of North Carolina

The Aegis System
Douglas Blewett, AT&T Bell Laboratories

For further workshop details, contact:
Reidar J. Bornholdt
Room 7-444
College of Physicians & Surgeons
Columbia University
630 West 168 Street
New York, NY 10032
212-305-3411

{harpo|cmc12|lcucard|reidar
{decvax|ucbvax|usenix!reidar

For registration details, contact:
USENIX Conference Office
P.O. Box 385
Sunset Beach, CA 90742
213-592-1381
213-592-3243

The registration fee is $200, payable in advance. Mastercard, Visa and American Express are accepted for registration.

The registration deadline is December 4, 1985.
Future Meetings and Workshops

Second Graphics Workshop – December 12-13, 1985, Monterey, California

USENIX is sponsoring the second annual workshop on current and future developments in computer graphics in the UNIX environment or with UNIX tools and/or philosophy. Please see the "Second Workshop on Computer Graphics" article in this issue of :login: for more information.

USENIX Meeting – January 15-17, 1986, Denver, Colorado

The emphasis will be on a series of intensive workshop seminars and tutorials. Please see the "USENIX Winter '86 Conference" article in this issue of :login: contain more information.

To avoid conflicting with the 1986 UniForum in Anaheim, there will be no formal vendor exhibition.

UniForum – February 4-7, 1986, Anaheim, California

The 1986 UniForum trade show and conference will be held in Anaheim on February 4-7, 1986. One day will be devoted to 15 day-long tutorials on various topics for UNIX users, programmers, managers, and sales and marketing people. Subsequent days will have general panel sessions oriented towards marketers and managers, and technical sessions with paper presentations. The exhibitor trade show will run for three days.

USENIX members are being provided a $25 discount coupon good towards the registration fee. The coupon is included with this issue of :login:.

For further information, please contact the /usr/group office at 408-986-8840.

AUUG Meeting – February 10-11, 1986, Perth, Australia

The 1986 Summer Meeting of the Australian UNIX systems Users' Group will be held on the 10th and 11th of February at the University of Western Australia, Perth, Western Australia.

Papers on subjects related to the UNIX system of UNIX-like systems are called for this meeting. Abstracts should arrive at the University no later than last mail Friday, 13 December. Indication of intention by phone, or mail is desirable as early as possible. Abstracts and papers should be sent to:

AUUG Summer Conference
Attention: Chris McDonald
Department of Computer Science
University of Western Australia
Nedlands, Western Australia, 6009 AUSTRALIA

or (preferably) by electronic mail to:

ACSnet: auug@wacsvax.oz
CSNET: auug@wacsvax.oz
UUCP: seismo!munnari!wacsvax.oz!auug
ARPA: auug%wacsvax.oz@seismo.css.gov

For further information, please contact:

Glenn Huxtable
Department of Computer Science
University of Western Australia
Nedlands, Western Australia, 6009 AUSTRALIA
Phone: +61 9 380 2878
ACSnet: glenn@wacsvax.oz
CSNET: glenn@wacsvax.oz
UUCP: seismo!munnari!wacsvax.oz!glenn
ARPA: glenn%wacsvax.oz@seismo.css.gov
EUUG Meeting – April 21-24, 1986, Florence, Italy

The April EUUG Conference and Exhibition will have technical conferences, tutorials, industrial sessions, and an exhibition.

The technical conference will run for three days, beginning April 21. The main theme will be real applications of the UNIX system. This is a direct follow on from the X/OPEN group announcement made at the Copenhagen Conference. Notwithstanding this, papers on all subjects relating to the UNIX system will be considered.

The EUUG has decided to run two conferences per year, one of which will host the major European UNIX exhibition. At the Florence conference, there will be a three-day exhibition opening on April 21.

There will be at least one day of tutorials addressing advanced features of UNIX.

On the first day of the Conference, before the technical program, there will be industrial sessions. These talks are intended to be commercial and not necessarily technical in nature.

Call for Papers and Topics

If you are willing to present a paper at any of the sessions mentioned above, or if you would like to suggest a speaker or topic of interest to you or your organization, please express this to:

Mrs. Helen Gibbons
EUUG
Owles Hall
Buntingford, Herts. SG9 9PL, United Kingdom
+44 763 73039

If you would like to receive booking information when it is available, please contact the EUUG immediately at the address above. Pre-conference bookings will be offered at a significant discount.

The Programme Chair for the Florence conference is Nigel Martin of The Instruction Set.

Deadlines for submissions are:
Suggested topics: immediately
Formal abstracts: December 1, 1985
Complete papers: February 20, 1986

USENIX Meeting – June 10-13, 1986, Atlanta, Georgia

The USENIX Summer ’86 Conference will be held in Atlanta on June 10-13, 1986. There will be a conference, tutorials, and vendor exhibits.

USENIX Meeting – June 9-12, 1987, Phoenix, Arizona

The USENIX Summer ’87 Conference will be held in Phoenix on June 9-12, 1987. There will be a conference, tutorials, and vendor exhibits.
Nominations for Election of Officers and Directors of USENIX Association

The Board of Directors has appointed Randall L. Frank as chairman of the 1986 Nominating Committee. Members desiring to recommend someone to the Nominating Committee should submit their recommendations to him, addressed:

Randall L. Frank
USENIX Nominating Committee Chairman
University of Utah
Computer Science Department
3160 MEB
Salt Lake City, UT 84112
seismoulah-cs!frank
801-581-5591

The Association Bylaws state that “Nominations for each Office and Directorship may also be made by any five members.” (Article 7.1). Members are encouraged to give serious consideration to the representation desired on the Board – as a body it sets the policies and direction of the Association. Should a group wish to nominate a member for election, please send a letter in writing – no electronic mail to arrive before Sunday, December 1, 1985, to:

Lewis Law
USENIX Association Secretary
Harvard University
Science Center
1 Oxford Street
Cambridge, MA 02138

The letter should contain the name of the nominee, a statement that he/she has accepted nomination, and the signatures of at least five members.

Report on the IEEE P1003 Portable Operating Systems Environment Committee

The IEEE P1003 Portable Operating Systems Environment Committee (formerly known as the UNIX Standards Committee) is the descendent of the /usr/group Standards Committee, which produced the /usr/group standard. Many of the same people are on the IEEE/P1003 committee, and the purposes are similar, such as promoting the portability of programs by providing a commonly implemented operating system environment.

The IEEE/P1003 working group is nearing a consensus on a Trial Use Standard to be voted on by the balloting group. If the Trial Use Standard is approved, it will be in effect for a period of time (such as a year) during which it is hoped that an actual standard can be produced, based on experience with the Trial Use Standard.

Here is the schedule for recent and currently planned IEEE P1003 committee meetings and related events.

10-12 Sep 1985 Met in Tyson's Corner, Virginia (just outside of Washington D.C.). Progress was made in several areas, such as relations with the X3J11 C Standards Committee. The draft used in this meeting was P1003/D4.
14 Oct  Revised document, P1003/D5, containing changes from the D.C. meeting, available online, and received in the IEEE office. Start of document review, mass mailing to working group.

1 Nov  Technical review – Steering Committee meets in Dallas for editing session.

4 Nov  Master copy of document to IEEE.

8 Nov  Mail document for balloting. Technical review and ballot resolution begins.

11 Dec  End of balloting. Ballot resolution continues.

13-15 Jan 1986  P1003 working group meeting in Denver (in conjunction with the USENIX Winter Conference) to review comments and proposed responses.

21-31 Jan  Window during which ballot responses can be changed.


22 Mar  IEEE Standards Board meeting.


9-11 Jun  Proposed P1003 working group meeting in Atlanta in conjunction with the USENIX Summer Conference.

This schedule may slip if any form of the document is not ready at its particular deadline; however, everything is currently on schedule.

Some P1003 committee members will be attending upcoming X3J11 committee meetings to help promote coordination between the two committees.

Comments may be submitted to the working group by several means. There is a paper mailing list by which interested parties may get copies of the drafts. The address for requests to get on that list and for comments on drafts is:

James Isaak  
Chairperson, IEEE/CS P1003  
Charles River Data Systems  
983 Concord St.  
Framingham, MA 01701

dercax!frog@lim

Copies of the drafts may also be obtained by electronic means from several hosts on the ARPA Internet and the UUCP network, as announced in the newsgroup mod.std.unix. Articles posted in that newsgroup will be relayed to the Steering Committee by me (the moderator of the newsgroup) as the USENIX delegate.

Please key all comments by draft number and section number (and possibly section name), not by page number. There may be many printed forms of the draft, each with different page numberings.

Though it is rather late to do so, one may still join the working group, by coming to the January meeting in Denver. While there is evidently a procedural rule which could be invoked by anyone at any meeting to limit participation in actual decisions only to people who have attended two of the previous three meetings, this has never been done.

To join the Balloting Group and vote on the standard is more difficult. It is subdivided into two groups. To be a member of the quorum group, you must be a member of IEEE or the IEEE Computer Society, and must return a vote. To be in the other subgroup, you need not be a member of anything and your vote does not count towards the number of ballot returns required for draft approval. However, comments or objections from either subgroup receive equal consideration and visibility. Objections to a draft must be for specific reasons and should provide an alternative. To join either subgroup, mail a request to the above address of the Chair, but do it soon.
As a somewhat more indirect method, comments sent to me will be taken into account in deciding the USENIX ballot, though the actual ballot will be determined by the USENIX Board of Directors.

Comments

Here are some comments on the content and intent of the P1003 draft standard. They are my opinion, and not the official position of IEEE, P1003, or anyone else. While I think I understand the things I'm writing about here, I've only been to two committee meetings. I trust that other, more experienced members will correct me if I stray too far from the consensus.

Many people have the impression that the P1003 standard will be almost exclusively based on System V. This is not really true. The draft standard is probably closer to System V than to any other variant of UNIX, and the System V Interface Description is a constant reference at committee meetings. However, committee members often express concern about not outlawing features of hosted systems (emulations on top of other operating systems), networked systems, or distributed systems. (The standard does not explicitly address most of these issues, it just carefully does not make them impossible or hard to do.) Also, many of the committee members run non-System V-based software on their own systems at work. 4.2BSD features, in particular, are frequently mentioned.

Thus, while some System V-specific features like FIFOs appear in the standard, the mechanisms provided for reading directories are the 4.2BSD opendir/readdir/closedir functions, and the data interchange format is tar, not cpio.

Another major concern of the P1003 committee is compatibility with the X3J11 C standard. This led to major modifications to P1003.D4.

An issue which has not been addressed thus far to any great extent is internationalization. There is no mention of character sets other than ASCII, for instance.

And another issue which is explicitly not addressed is binary compatibility: the standard is intended to facilitate the writing of programs whose source may then be moved from one conforming implementation to another with minimal changes.

While the P1003 committee wishes to produce a standard which is inclusive enough to be of use, it is necessary to start with a small trial use standard and include other issues in later drafts.

John Quarterman
USENIX Delegate, IEEE/P1003
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
jsq@sally.UTEXAS.EDU
{gatech,harvard,ihnp4,seismo}@ut-sally!jsq

Announcing Third Printing of USENIX 4.2BSD Manuals

Since inventories of the second USENIX-sponsored printing of 4.2BSD manuals were "sold out" as of July 19, 1985, and since the office has continued to receive orders since that date, USENIX has decided to make a third manual printing. Manual production is now in progress, with shipments to begin by the end of October.

The third printing will be much the same as the previous two. Prices remain unchanged, and procedures for ordering the manuals will remain as before. A Manual Authorization and Order Form is on the following page of this newsletter. It is important to fill out this form carefully, include a valid purchase order number, and have an authorized person sign the form. As before, only current USENIX Institutional and Supporting members can order manuals. Failure to follow the procedures detailed on the order form has been the cause of delays in many institution's previous orders, so be sure to include all the information asked for.
4.2BSD Manual Reproduction Authorization and Order Form

USENIX Member No.: 
Purchase Order No.: 
Date: 

As the representative of a USENIX Association Institutional or Supporting Member in good standing, and as a bona fide license holder of both a 4.2BSD software license from the Regents of the University of California and a UNIXtm/32V or System III or System V license or sublicense from Western Electric Company, and pursuant to the copyright notice as found on the rear of the cover page of the UNIX/32V Programmer's Manual stating that

"Holders of a UNIXtm/32V software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included,"

I hereby appoint the USENIX Association as my agent, to act on my behalf to duplicate and provide me with such copies of the Berkeley 4.2BSD Manuals as I may request.

Signed: ________________________________________________
Institution: ________________________________________________

Ship to: 
Name: ____________________________________________
__________________________________________
__________________________________________
__________________________________________

Billing address, if different: 
Name: ____________________________________________
__________________________________________
__________________________________________
__________________________________________

Phone: ____________________________ Phone: ____________________________

The prices below do not include shipping and handling charges or state or local taxes. All payments must be in US dollars drawn on a US bank.

4.2BSD User's Manual ______ at $18.50 each = $___________
4.2BSD Programmer's Manual ______ at $17.00 each = $___________
4.2BSD System Manager's Manual ______ at $ 9.50 each = $___________
Total ______ $___________

[ ] Purchase order enclosed; invoice required (P.O. # ____________)
(Purchase orders must be enclosed with this order form.)

[ ] Check enclosed for the manuals: $______________
(Howard Press will send an invoice for the shipping and handling charges and applicable taxes.)

Make your check or purchase order out to Howard Press and mail it with this order form to:

Howard Press
c/o USENIX Association
P.O. Box 7
El Cerrito, CA 94530

for office use: I.v.: _______ check no.: _______ amt. rec'd: _______
Local User Groups

The USENIX Association will support local user groups in the United States and Canada in the following ways:

- Assisting the formation of a local user group by doing an initial mailing for the group. This mailing may consist of a list supplied by the group, or may be derived from the USENIX membership list for the geographical area involved. At least one member of the organizing group must be a current member of the USENIX Association. Membership in the group must be open to the public.
- Publishing information on local user groups in ;login: giving the name, address, phone number, net address, time and location of meetings, etc. Announcements of special events are welcome; send them to the editor at the USENIX office.

Please contact the USENIX office if you need assistance in either of the above matters. Our current list of local groups follows.

In the Boulder Colorado area a group meets about every two months at different sites for informal discussions.

Front Range Users Group
N.B.I., Inc.
P.O. Box 9001
Boulder, CO 80301
Steve Gaede (303) 444-5710
haolnbires!gaede

Dallas/Fort Worth UNIX User’s Group
Advanced Computer Seminars
2915 L.B.J. Freeway, Suite 161
Dallas, TX 75234
Irv Wardlow (214) 484-UNIX

In the Washington, D.C., area there is an umbrella organization called Capitol Shell. It consists of commercial, government, educational, and individual UNIX enthusiasts. For information and a newsletter write:

Capitol Shell
8375 Leesburg Pike, #277
Vienna, Virginia 22180
seismol!rgvax,umcp-cs,andi]!cal-unilcapish

In the New York City area there is a non-profit organization for users and vendors of products and services for UNIX systems.

Unigroup of New York
G.P.O. Box 1931
New York, NY 10116

In Minnesota a group meets on the first Wednesday of each month. For information contact:

UNIX Users of Minnesota
Carolyn Downey (612) 934-1199

In the Atlanta area there is a group for people with interest in UNIX or UNIX-like systems:

Atlanta UNIX Users Group
P.O. Box 12241
Atlanta, GA 30355-2241
Marc Merlin (404) 255-2848
Mark Landry (404) 874 6037

In the Seattle area there is a group with over 150 members, a monthly newsletter and meetings the fourth Tuesday of each month.

Seattle/UNIX Group
P.O. 58852
Seattle, WA 98188
Irene Pasternack (206) FOR-UNIX
uw-beaver!tikal!scc!slug

An informal group is starting in the St. Louis area:

St. Louis UNIX Users Group
Plus Five Computer Services
765 Westwood, 10A
Clayton, MO 63105
Eric Kiebler (314) 725-9492
ihnp4!plus5!sluug
In the northern New England area is a group that meets monthly at different sites. Contact one of the following for information:

Emily Bryant  
Kiewit Computation Center  
Dartmouth College  
Hanover, NH 03755  
decvax!dartvax!emilyb

A UNIX/C language users group has been formed in Tulsa. For current information on meetings, etc. contact:

Pete Rourke  
$USR  
7340 East 25th Place  
Tulsa, OK 74129

David Marston  
(603) 883-3556  
Daniel Webster College  
University Drive  
Nashua, NH 03063

---

Publications Available

The following publications are available from the Association Office or the source indicated. Prices and overseas postage charges are per copy. California residents please add applicable sales tax. Payments must be enclosed with the order and must be in US dollars payable on a US bank.

USENIX Conference Proceedings

<table>
<thead>
<tr>
<th>Meeting</th>
<th>Location</th>
<th>Date</th>
<th>Price</th>
<th>Overseas Mail</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>USENIX</td>
<td>Portland</td>
<td>Summer '85</td>
<td>$25</td>
<td>$25</td>
<td>$5</td>
</tr>
<tr>
<td>USENIX</td>
<td>Dallas</td>
<td>Winter '85</td>
<td>$20</td>
<td>$25</td>
<td>$5</td>
</tr>
<tr>
<td>USENIX</td>
<td>Salt Lake</td>
<td>Summer '84</td>
<td>$25</td>
<td>$25</td>
<td>$5</td>
</tr>
<tr>
<td>UniForum</td>
<td>Wash. DC</td>
<td>Winter '84</td>
<td>$30</td>
<td>$20</td>
<td>/usr/group</td>
</tr>
<tr>
<td>UNICOM</td>
<td>San Diego</td>
<td>Winter '83</td>
<td>$15</td>
<td>$25</td>
<td>$5</td>
</tr>
</tbody>
</table>

USENIX Association  
P.O. Box 7  
El Cerrito, CA 94530
/usr/group

4655 Old Ironsides Dr., #200  
Santa Clara, CA 95050

EUUG Publications

The following EUUG publications may be ordered from the USENIX Association office.

The EUUG Newsletter, which is published four times a year, is available for $4 per copy or $16 for a full-year subscription. The earliest issue available is Volume 3, Number 4 (Winter 1983).

The July 1983 edition of the EUUG Micros Catalog is available for $8 per copy.
USENIX Services

Steve Johnson
Director

USENIX offers a number of services to its membership. Some are familiar, such as login: and the sponsorship of the USENIX conferences.

However, we also offer some non-traditional services, and would be interested in expanding these roles in the future.

We are prepared to sponsor workshops on topics of interest to our membership. We would define a workshop as a focused conference with limited attendance (~100). If you would like to organize a workshop, write the Board a letter with your proposal (usenix/board gets to the board of directors). If we like it, we can offer:

- Help with the hotel, food, and travel arrangements
- Financial underwriting the conference (could be $20,000!)
- Help with registration, mailings, etc.
- Hints, suggestions, and tons of advice

The model here is that you worry about the technical side, and we will deal with all the details. We have run several of these workshops now, and would be happy to run more.

Another effort we would be willing to sponsor is research or development into areas of interest to our membership. In response to a flood of concern about Netnews, we sponsored the uucp mapping program, and also have sponsored the Stargate experiment.

We would be quite prepared to hear additional suggestions or proposals for how to spend more money on our members in non-traditional ways. Generally speaking, we would look most favorably on projects with modest beginnings (the total cost for the mapping and Stargate projects was only a couple of thousand dollars) – just at present, we aren’t in the position to finance a total rewrite of UNIX in Modula II, for example. Here again, what we offer is some expense money and a chance to take care of some of the non-technical details while you worry about the technical problems.

So if you would like to have our help in setting up a nationwide face server, or a distributed uucp autorouter, or anything else you think you understand technically, usenix/board would like to hear from you.

Papers from the 1984 USENIX Graphics Workshop

The papers presented at the USENIX-sponsored “UNIX and Computer Graphics Workshop” on December 13-14, 1984, in Monterey, California, are reprinted on the following pages.

Each of the following papers is © Copyright 1985 by the author(s) – All Rights Reserved. They may be reproduced only by USENIX Association members for personal or in-house, non-commercial use. None may be reproduced or distributed in any form for any other use without permission of the author.
Window Managers are Operating Systems:

Software for a Distributed Graphics System

S. McGeady
Ann Arbor Terminals

ABSTRACT

Window Management is the control of multiple display contexts represented by overlapping rectangular regions on a graphics device, typically a bitmap. Contrary to popular belief, window management is not a graphics issue. While Window (or Display) Management uses certain graphics tools and hardware to implement its underlying structure, the main difficulties lie in traditional operating system areas: process scheduling, I/O multiplexing, memory allocation, and overall resource management.

This paper consists of two parts: the first discusses the parallels between various operating system functions and the analogous functions in a window manager; and the second part focuses on a software architecture for a distributed display device implementing these principles.

This paper is organized into two sections. The first section discusses the parallels between traditional multiprocessing operating systems and the author's conception of the functions of a window manager. The second section reviews a software architecture which implements the principles discussed in the first section.

The Window Manager as Operating System

1. Introduction

It is by now trite to say that the fields of bitmapped graphics and graphical user-interface design have increased tremendously in importance in recent years. It is less obvious that in the recent rash of development in the graphics field, contributions that could have been made from adjoining fields in computer science have been overlooked. In particular, well-understood principles of operating system and distributed system design are often ignored in the development of many modern commercial and research display systems.

That window management is a facet of operating system design may seem, for some to be obvious, while for others, those who are more at home with CORE and GKS than UNIX, this may appear absurd. Window Management has been looked on by many as an issue to be resolved with the application of traditional graphics tools: clipping algorithms, rasterization techniques, and generic models from the graphics world. These techniques are well understood, and are usually applied correctly in graphics systems. However, the step beyond graphics to window management is often never made, or made in the wrong direction, because the traditional graphics viewpoint does not lead in
the right direction. In other words, sophisticated graphics techniques exist in a windowing graphics system above and below an operating system, and that operating system is different in important ways from a standard host OS.

Thus, the intent here is to show that window managers can and should be built using principles learned in operating system development. In addition, I hope to show that windowed display devices are not only well suited to distributed processing, but demand it when performance and responsiveness of user interface is important.

The statement that window managers are operating systems should not be misconstrued to mean that a window manager is or should be a part of a traditional operating system, or that it is trivially identical to such a system. Rather, it appears that window managers are analogous to operating systems in several crucial ways. The most important implication is that window managers can be built with the same tools and with the same base of experience as operating systems. Another way of saying this is that window management falls into the same problem domain as operating systems, but differs in the details of specific solutions to some problems.

2. Definitions

Before we proceed, some definitions need to be made.

1) Window — The word window is bandied about by many, and is often used to mean different things. It means one thing in traditional graphics textbooks, and another in modern usage regarding bitmapped displays. In this context, windows are Xerox-style [Tes81] windows, that is, overlapping rectangular regions represented on a bitmapped screen, that are concurrently updated, and that represent multiple input/output paths. These windows represent views onto some logical process on a host processor, e.g. an editor (graphical, text, font, etc), a file-viewing utility, or a programming environment.

2) Frame Buffer — A region of memory that is mapped, unit-for-unit, onto the pixels of a CRT. In most cases, the pixels are represented by single bits, but in color systems, they may be represented by as many as 24 bits.

3) Bitmap — A (conceptually) rectangular region of memory containing a displayable image. Contrasted with the frame buffer, bitmaps may occur anywhere in display or host memory, as well as in the frame buffer.

4) Display Device — In this context, a unit consisting of a bitmapped frame buffer being driven by a dedicated general-purpose processor. A display device will usually have user interaction devices such as keyboards as graphics input (GIN) devices attached.

5) Host Processor — The computer system in which an application that is directing the display is running. The use of the singular is not intended to exclude multiprocessor systems: indeed, the architecture described below is well-suited to multiprocessor systems.

3. Operating System Functions Applied to Window Management

A traditional operating system has several responsibilities, in particular: memory allocation, memory protection, memory mapping, I/O mapping, I/O multiplexing, CPU management, and common services.
3.1 Memory Allocation

Memory allocation consists of the division a scarce resource, main (primary) memory, between many competing processes, usually when there isn't enough to go around. Grossly simplified, the rules are that the currently active process gets as much main memory as it needs, and lower priority processes get correspondingly less. There is usually some feedback relating the total volume of memory consumed by a process to its priority. A large number of techniques and heuristics have been developed for this task, virtual memory techniques being the most common. These techniques allow a conceptually straightforward way of handling and addressing memory that is not currently represented in primary memory. Secondary and tertiary memory hold what cannot fit in the available memory.

In a graphics system the scarcest resource is the frame buffer — the screen memory. The rules for dividing it are different, but analogous. Overlapping windows provide a user interface and a graphic representation of the processes (windows, in this case) to which the frame buffer is allocated. Unobscured windows have intrinsically higher priority over screen memory, and obscured windows must have their pieces of the frame buffer held elsewhere. Many graphical techniques for maintaining the obscured sections have been tried, most often involving the maintenance of a display list that represents everything displayed in the window [Meyr80]. This technique has several drawbacks that are discussed in [Pike83], along with another technique, the layers approach to frame buffer management. Layers allows raster operations on bitmaps that are partially represented in on-screen memory (frame buffer) and partially represented in off-screen memory. An extension of this technique that allows parts of windows in virtual off-screen bitmaps (in semiconductor memory or on disk) and virtual memory display lists is described later in this paper.

A graphics device that provides computational capabilities must also manage normal per-process memory in the traditional way.

3.2 Memory and I/O Mapping

Memory mapping is closely linked with memory allocation and protection. An operating system provides the illusion of a virtual machine to its processes: each can safely assume that it is the only process running on that machine, and that unless otherwise arranged for, it controls its own address space. Operating system software and processor hardware map virtual addresses onto noncontiguous and possibly non-resident pages in physical memory. They also map physical addresses so that the process’s virtual address space begins at a constant, known address at all times, so that relocation is not required.

Similarly, the operating system provides a virtual I/O system for the process: filenames are mapped from the conceptual file structure to directory blocks, and file blocks are mapped into physical disk blocks.

The window manager maps the client’s virtual frame buffer onto a physical bitmap. The client process has the image of an unobscured, fully resident screen image, although little or none of it may be actually represented in the available physical frame buffer. Pieces that are not represented in the frame buffer are handled using the layers software, which manages pieces of these buffered in other memory, as discussed above. All operations function on this virtual frame buffer as though it were contiguously represented on the screen. The window manager arranges so that the client’s virtual frame buffer always starts at a constant address, (0,0), so the client need not perform a new set of
transformations whenever the window is moved around the screen.

3.3 Memory Protection

Another function of multi-tasking and multi-user computing environments is the protection of one process and the system itself from other processes. It is never acceptable for a marauding process to write over the memory image of another process. Thus, programs are limited by the operating systems to certain bounds, accesses outside of which are denied and flagged as errors. This memory protection is supported by hardware on most processors, and actively managed by the operating system.

It is less widely accepted, but no less true, that a display process should not be able to write outside its own window on a bitmapped screen. A display device that supports asynchronously active windows must treat these as though they are virtual terminals or virtual frame buffers, entirely distinct from one another. While some systems (e.g. the Smalltalk-80 environment [Ing81]) allow arbitrary access to the screen bitmap, they do not support concurrency or multi-tasking for displaying processes. Thus, the window manager must flag out-of-bounds bitmap references and deny them. This function typically falls out of memory mapping.

3.4 I/O Multiplexing and Demultiplexing

An operating system multiplexes output directed from processes onto single shared devices such as disks and communications devices, and demultiplexes the responses back to the originally requesting process.

As in an operating system, there may be many processes attempting to communicate with the (shared) display device simultaneously. Typically there is a single logical and/or physical communication path between the host processor and the display device over which all communications must pass. A window manager demultiplexes input from this path and controls the multiplexing of this input onto the shared screen. It also demultiplexes operator-generated responses from input devices to the appropriate clients.

3.5 Scheduling (CPU Management)

Along with memory, CPU cycles are among the scarcest resources in multiprocessing systems, and thus are managed most actively. Numerous algorithms exist for scheduling processes according to various heuristics. A typical scheduling algorithm may break processes down into lists of those that are ready and waiting to run, those that are waiting for virtual memory page-ins, those waiting for fast I/O (disk) to complete, and those waiting on slow I/O (tty lines). An operating system may also schedule access to special-purpose computing hardware, such as a floating point processor or an array processor.

A window manager manages display update in much the same way. It schedules windows, according to its own set of rules: those that are active for user input typically get the highest priority, followed by those that are wholly unobscured on the screen, those that are only partly obscured, those that are wholly obscured, and those that are deactivated.

In many systems, the window manager must also schedule access to special-purpose hardware such as special rasterizers, bltblt engines, scrolling hardware, or other devices.
3.6 Common Services

An operating system typically provides a set of common services for the system as a whole other than those already mentioned. These are things that are either universally used, or are inefficient or impossible to do in client processes, or that need some sort of special protection for security or other considerations.

There is a similar set of operations that a window manager might perform for its clients, including, but not limited to: graphics input (GIN) tracking, and interface to special hardware. In particular, to implement the protection and mapping functions outlined above, the Window Manager must control access to the bitblt primitive.

4. Summary

There are certainly functions of operating system which are not represented in a window manager, and vice versa. However, there is enough similarity in the function of the two that they can be implemented using the same tools. Let it be clear, however, that the parallels between operating systems and window managers do not mean that they are one in the same. Combining them makes about as much sense as combining a screwdriver and a hammer: they both make holes in wood, but they do it in much different ways.

Software Architecture for a Distributed Display

5. Introduction

When building a high-performance graphics workstation, the first reaction many designers have is to closely couple the central processor with the display device, as a way of ensuring a high-bandwidth link between the two [Levy84]. This is usually seen as guaranteeing good performance for the system. This simplistic approach maps a bit-mapped frame buffer into the central processors data space, with the notion that the CPU can then access it as fast as possible, maximizing system throughput. This, however, is not nearly enough to ensure speed, and in fact, often destroys it. The maintenance of a windowed display consumes a large amount of processing resources. Bitblt and rasterization operations are CPU intensive, and handling GIN and keyboard devices associated with a display can be expensive. Some react to this by attaching special-purpose widgets to the display to perform scrolling or bit moving operations, but the complexity of interfacing to these often outweighs their advantages. It is usually discovered that the standard operating system is hopelessly inadequate for the task of managing a windowed display. It has the wrong scheduler for the job, the wrong I/O structure, and the wrong memory management system. (It has been pointed out, anecdotally, that UNIX’s cat on the SUN workstation is CPU bound.)

Distributed systems are the answer to this problem. We should let a general-purpose computing server do what it does best: manage multiple concurrent processes, handle disks and databases, provide services, and so forth, and design a graphics device that handles the necessary details of the display. It should become clear that a graphics display is a logically separate entity from a host (or application) processor. There are certain operations, for example, bitblt and polygon fill, that are either done directly by display hardware or need intimate access to it. Further, you may wish to place appropriate parts of certain applications in the display, leaving the remainder in the host. The host, for example, supports file system management and databases, runs compute-bound tasks, and networks to other machines. The display would update a window, create and
maintain display lists, control the user interface, and perform sundry imaging operations. As an example, a graphical editor might consist of a large section running on a display processor, handling user interaction and displaying the result, while design-rule checking and database handling would be shunted to the host. What is needed is an environment for the display that is a "mirror image", in a sense, of the operating system environment – one that is capable of running a class of display processes and providing sensible support for them.

When we assign a general-purpose processor to the display device, we must be careful to design an interface to this device that will not overburden the central processor with management details. Thus, the host and display should meet over the narrowest (i.e. lowest bandwidth) possible conceptual channel. Furthermore, if one wants to design a display device that is not beholden to a particular central processor (a terminal), then one is strongly enjoined from using a high-bandwidth physical interface.

This leads us to the need for an architecture to support this style of interface and address the problems of running a display, and this architecture, as we have seen in the previous section, looks like an operating system. In particular, if the operating system running on the host processor is UNIX, the window manager operating system will bear a startling resemblance to UNIX, without being identical.

6. Display Hardware

The software architecture presented here is intended as a general example of the application of operating system principles to display software design, and is not beholden to a particular set of hardware. Nevertheless, a little must be said about the minimal hardware requirements of a system like this, and about the hardware for which this architecture was originally conceived.

1) The display requires a general purpose processor, preferably with memory management capabilities. In a dedicated workstation environment it is desirable for the display processor to be similar or identical to the host processor – this simplifies the implementation and testing of the software. In any case, the display processor should be one for which software is easily generated.

2) It is desirable, but not essential, that the processor have memory management capabilities. This aids debugging and increases the robustness of the system as a whole.

3) Communication to the display may be by any means, but should generally be made to appear as a stream of messages.

This architecture was originally designed for a workstation in which the host processor was one or more National Semiconductor 32016 processors, the display processor was an exact copy of the host processor board, the the host processors communicate with the display via a message-passing protocol implemented in shared memory. The display processor supported virtual memory, and communicated directly with a processor that controlled secondary storage, paging onto a dedicated disk partition. All memory in the display except the physical screen memory was pageable. The display processor was augmented by a second board containing a high-speed special purpose rasterization processor that implemented traversal of display lists in virtual memory and the execution of general-purpose display primitives, including blt, blt, cine-drawing, and fill algorithms. Versions of this architecture are currently being implemented on this system, and on a remote bitmapped terminal, lacking special display hardware, memory management, secondary storage, and having only a serial communication link.
7. Display Software Architecture

The software running on the display device can be divided into several major sub-
sections: a Screen Manager, a Scheduler, a Memory Manager, an I/O Manager, that
together comprise the Display Operating System, and a set of processes controlling the
semantics of the windows on the screen, known as Per-Window Processes (PWPs) or sim-
ply Window Processes.

Figure 1 (toward the end of this article) portrays the overall design of the system.
The upper half of the drawing depicts the software that runs in the general-purpose
display processor, and the lower half depicts functions that can be off-loaded to a special
processor, if one is available.

7.1 The Per-Window Processes

The semantics of each window represented on the screen are controlled by a set of
Per-Window Processes (PWPs). There is, as the name implies, one of these for each win-
dow on the screen. These would be called "user mode" processes in a normal operating
system. These processes receive input from a cooperating process on the host and from the
display devices input devices, and manages a virtual frame buffer. The PWP knows
only about the content of this frame buffer, and nothing about where the window is on
the screen as a whole. As far as it is concerned, it alone is writing to a single frame
buffer.

Examples of typical Per-Window Processes are:

1) A combination UNIX TTY driver and ANSI X3.64 (VT100-like) terminal emulator,
   implementing a programmable keyboard, pop-up menus that act like function keys,
   and other amenities;

2) The back-end to a Graphics Kernel System (GKS) or CORE device driver, imple-
   menting standard graphics primitives such as line-drawing, polygon fill, segment
   management, and some coordinate transformations;

3) The front-end of a graphics editor, implementing the command interface and the
   graphics functions; and

4) The user interface of a text processing system.

A Per-Window Process generates a list of commands that are pending for the
display. In the typical case, this may be a simple list of bitblt commands. On systems
that support special-purpose rasterization hardware, this display list may consist of com-
mands to draw lines, polygons, or fill areas. Bitblt commands may address virtual frame
buffer memory (whether screen-resident or not), and memory within the Per-Window
Process itself.

Per-Window Processes are run in User Mode (protected mode) on the Display Pro-
cessor. This has the advantage that a rogue PWP cannot crash the entire display system
— only the window in which they are running. This property helps tremendously when
debugging the system, and when downloadable user-written display routines are sup-
ported.

The interface between the PWP and the Display Operating System consists of many
system calls, looking much like a portion of the UNIX [Ritc74] operating system inter-
face, but lacking any semblance of a filesystem, and with the addition of several special-
purpose calls.
7.2 The Screen Manager

The Screen Manager is responsible for the positioning of each window on the screen (or in the physical bitmap). The Screen Manager does not deal at all with the contents of a window. It is only concerned about the mapping between each per-window process' virtual frame buffer and the physical frame buffer. Thus, the Screen Manager is the memory management controller for the physical bitmap memory. It maintains a set of data structures that record the location of various pieces of each virtual frame buffer: pieces may be actually represented on the screen or in a pool of virtual frame buffer pieces, or un rasterized display commands may be held in a per-window process's command display list.

When a window is moved into a position where it obscures part of another window, the obscured region of the window is copied into a buffer pool of frame buffer pieces. Using *layers*, these pieces of frame buffer can be acted on by *bitblt* and line-drawing primitives as though they were on the screen. Thus, these undisplayed bitmaps are kept up-to-date, and are ready for redisplay when windows get shuffled again. One difficulty is that when the screen display becomes complex, with many overlapping windows, there may not be enough physical memory to hold the non-displayed frame buffer pieces. There are two potential solutions to this problem. If a secondary storage device (e.g. disk or bubble memory) is available, the frame buffer pieces can be paged off onto this device, either explicitly or using hardware virtual memory support. The least recently used pieces of frame buffer could be selected for paging, or a more complex algorithm involving the relative priorities of the windows involved could be used. Alternatively, if no secondary store is available, the frame buffer pieces can be destroyed, and the per-window process command list can be maintained to back-up the window. This command list is much denser than a bitmap, and thus substantially smaller for large bitmaps.

The Screen Manager also manages access to special rasterization hardware, if it is available. Ideally, the rasterizer will interpret both the Screen Manager's window data structures and execute the per-window process' command list, directing the resulting bitmaps into either the physical frame buffer or a virtual frame buffer element in off-screen memory. Since the rasterizer has no access to memory management hardware, in a virtual memory environment the Screen Manager must page-lock pages that the rasterizer will need.

7.3 Scheduler

Since there are many per-window processes running in the system, some scheduling must be done to ensure that the highest priority windows are updated most frequently. Scheduling rules for per-window processes are simple but much different from those for a traditional operating system. Windows are classed as follows, with decreasing priority:

1) Active – The active window is the one with which the user is interacting. This is usually the window to which the keyboard is attached. It may or may not be wholly unobscured (see below). Giving highest priority to this window helps ensure maximum responsiveness to user input. This is crucial for applications such as paint programs where the per-window process must track a GIN device in real-time, "painting" bits on the screen with *bitblt* as it moves.

2) Unobscured – These windows are the class of windows that are not overlapped by any other windows on the screen. We make the assumption that the user has arranged them this way for a reason, and that an effort should be made to keep these windows as up-to-date as possible. Within this class windows are scheduled
according to how much outstanding input from the host is pending: those with a large backlog are run most often.

3) Partially Obscured — These are windows that are partially overlapped by other windows, but have pieces still visible on the screen. In practice, these windows can be considered in the same class as unobscured windows, and scheduling priority can be determined according to the percentage of unobscured window area to total window area. Thus, a large window with a tiny portion obscured will run almost as often as an unobscured window.

4) Wholly Obscured — These are windows that are logically on the screen, but of which no part is visible. These can be considered to be in the background, and are run only when no other windows need to be serviced.

Additionally, window priority can be affected by the Screen Manager and the I/O system. The Screen Manager can stop a process by artificially lowering the priority of a window that has too many pending commands in its display command list, and the I/O Manager can stop a process that has too much pending I/O for the host system. In a virtual-memory environment, per-window processes are also stopped and started because of page faults.

The Display System as a whole affects the priorities of host processes indirectly: the host input buffer of a low-priority window will fill up, and the host will be instructed by the I/O system to cease sending data, and will, according to its own scheduling rules, eventually stop the sending host process.

7.4 I/O Manager

The I/O system is responsible for external communications between the display device and the host system, and between the display device and secondary store, if any is provided. The I/O system is also responsible for communication within the display system: between the Window Manager and the window processes; and between the Per-Window Processes and hardware devices such as the keyboard and GIN devices. The I/O system demultiplexes communications from the host and maintains input buffers for each potentially active window process and for the Window Manager, and it maintains output buffers for each of these clients, and multiplexes return communication to the host.

The I/O path to the host appears logically as a stream of packets, each consisting of a header and a body. All information needed by the I/O system is contained in the header. The I/O system makes no interpretation of the body of the packets.

In practice, the 4.2BSD Berkeley UNIX [UCB83] Interprocess Communication (IPC) Facility provides an adequate basis for this communication stream. Other mechanisms would probably function equally well.

7.5 Memory Manager

The Memory Manager functions identically to its counterpart in a normal operating system, except that it receives requests from the Screen Manager to lock certain pages that are being accessed by special hardware.
8. Summary

There are several major advantages to this architecture:

1) It allows each window to implement entirely different internal semantics, thus implementing a true virtual terminal on a windowed device, while allowing maximal performance from the system as a whole;

2) It provides a mechanism whereby graphics-oriented processes (or parts of them) can be off-loaded from the host processor, providing better responsiveness to user interaction, and lower overhead on the host processor;

3) It can be effectively used where only a slow communications path to the display is present, but is equally effective when a higher-speed path is available;

4) It provides simple and well-understood rules for managing peak display load.

5) It can effectively support the use of some special-purpose display hardware.

6) It provides a straightforward development environment for process described in 1), above;

7) It enforces a logical separation between display functions and host functions that will have a positive effect on software design on a system.

There are a few disadvantages to this approach:

1) The system requires a dedicated general-purpose processor for the display.

2) The layers software works best with a large amount of physical memory (2 to 4 times the overall frame buffer size).

   Both of these disadvantages will be offset by rapidly falling processor and memory parts cost in coming months.

Notes and Acknowledgements

Much of the work described in this paper is still in progress. The author welcomes written comments, which may be mailed to S. McGeady, Ann Arbor Terminals, 711 SW Alder, Suite 402, Portland, OR 97205. Electronic mail may be directed via cbosglaatapdx.lmcg, or ucbvax:tektronixpsu-cs1aapdx.lmcg.

Several other people contributed to the architecture discussed here. A precursor to this architecture was developed by the author and Jim Valerio, Bruce Cohen, Ken Rhodes, and Michael Squires while at Tektronix, Inc.

References


[Mank83] Mankins, D. and Franklin, D. A Simple Window Management Facility for the
USENIX Association.

Manager/Virtual Terminal Systems. ACM Eighth Symposium on Operating Systems
Principles 15.5 (December 1980).

[Pike83] Pike, R. Graphics in Overlapping Bitmap Layers. ACM Transactions on Graph-
ics. 2,2 (April 1983).

USENIX Summer Conference USENIX Association.

[Ritc74] Ritchie, D. M. and Thompson, K. The UNIX Timesharing System. Communica-
tions of the ACM 17.7 (1974).


[UCB83] University of California at Berkeley, The UNIX Programmer's Manual, 4.2BSD,
(1983).

Software Architecture for Animation Systems

Roy Hall
Robert Abel & Associates
953 North Highland Ave.
Hollywood, CA 91607

Introduction

An animation system provides the tools to facilitate creative graphic expression using a computer.

The basic components of an animation are relatively well defined. These components are geometric database generation (modeling), time-based or frame-based parameter control (scripting), image generation (rendering), image postprocessing (compositing), and image output to recording and/or viewing devices.

What's in an Animation System

Animation is a combination of movement and geometric alteration of objects. In any image generation system, whether single frame or multiple frame animation, one needs a method of composing or scripting a scene.

In the simplest sense, scripting is describing what objects are in a scene and where they are located. But scripting also adds detail and emotion. Elements of scripting include animation of color, lighting, motion, geometric alteration, transparency, fog, control of image postprocessing options like diffusion and windowing, etc. As we shall see, information generated by the scripting system may be used by any of the other components of the system. Thus, the scripting system becomes the key element in an animation system.

Another important consideration for the scripting system is the planning aspect of animation. Very subtle changes in animation can produce large changes in the feeling of animation, thus it is important to be able to quickly preview animation before spending large amounts of time rendering frames.

There are two basic types of animation. The first is the movement of rigid objects through translation, rotation, and scaling. I call this "easy" animation in the sense that the objects are merely positioned through the use of transformations. The second is when the objects are not rigid and can undergo stretch, twisting, bending and transformation into entirely different objects. I call this "hard" animation in the sense that the description of the object is altered.

Often used are simple, complex, and combined forms of the "easy" and "hard" animation types. Moving single objects through a scene is simple easy animation (Panasonic glider.) Moving an object with respect to another object, which in turn moves with respect to another object is complex easy animation (TRW whirligig). Generating frame by frame databases through the use of an easily controlled procedure is simple hard animation (TRW ripple). Complex hard animation occurs when there are no easily defined procedures for generating data and a frame-by-frame hand procedure is used to generate data (Chadwick chair and CBS flag). Combinations occur when both positional transformation occurs with procedures generating different databases for each frame (shadows and reflections in the TRW ripple).

The interactions of all of the components of the animation system must be carefully choreographed to allow the graphic designer to plan and execute animated sequences with the system. Poorly defined interactions can make even the simplest animation extremely difficult to execute.

Modeling consists of generating object data. Objects may be fixed geometries created by a modeling system or created through procedures. Objects may also be texture maps, light, sources,
and any other data manipulated by the scripting system and then given to the rendering system for image generation.

The rendering system takes the generated data as specified by the scripting system and generates images or image elements. The types of rendering used depends upon the types of objects and the types of effects to be created. It is not unreasonable to expect a scanline renderer, a ray tracer, and a Z-buffer renderer to be found in the rendering system. The rendering system should be capable of producing images in a variety of formats including ntc, square pixel, and multiple resolution.

The post-processing system is a digital analogy to optical post-processing. Image elements can be combined and special effects added in the post processor. An example is the TRW ripple where the ground plane, foreground elements, background elements, sky, shadows, and reflections were all computed as separate elements and composited together. Glows were added on some of the lighting, shadow and reflection densities were adjusted as required, et. Additionally, this means that the entire scene need not be fully choreographed and modeled before the rendering process starts.

Display and recording put the digital information onto a monitor, film, and/or videotape. The system must know about any differences in format required to support any of these output media.

The system components and data pathways between the components can be summarized as:

---

**Fundamental Considerations**

We observe that animation systems are complex software projects and that the demands placed on an animation system are not static. New output format requirements are generated by almost any acquisition of display or recording equipment. The need for and availability of new objects requires system update. The interface needs of users continually changes as the types of animation to be preformed changes.

It is desirable to minimize the extent of the alterations that are required to accommodate any of these changes or updates. Object-oriented programming is the simplest and most flexible way to accomplish this. To avoid confusion between object-orientation and geometric objects, lets call this entity-oriented programming.

We can consider a number of “primitive” operations needed for the manipulation of data by each of the components of the system. For each entity we can divide the primitives into two basic functional groups. The first group consists of data creating and data editing primitives. The second group consists of data retrieval primitives.

Data creation occurs during scripting, modeling, rendering, and compositing. Some types of data, such as image data, can be generated and/or edited by a number of components of the system. In this case the create and edit primitives need to be easily used and the functional interface well
defined. The creation of other types of data such as geometric objects presents complications because of the variety of possible object representations and the dissimilarities of the representations. This suggests that a set of unique create and edit primitives could exist for each member of the set of geometric objects. Additionally this suggests that the modeling component may consist of a number of different modeling programs.

Data retrieval occurs within all components of a system. The data required or expected can be very well defined leading to well defined functional interfaces. For example, in previewing objects a wireframe of the object must be drawn. The scripting system must open object databases, get a list of the polygons in each object, and then close the database. This set of primitives has identical functional requirements for any and all geometric objects and light sources.

An example of an easily defined set of create and retrieve primitives are the image file handling routines. These routines could exist in the form:

```c
image_pointer = IM_open (name, read/write, mask)
image_header = IM_gen_header (format, info as required)
IM_w_header (image_pointer, image_header)
IM_r_header (image_pointer, image_header)
IM_w_scan (image_pointer, scan)
IM_r_scan (image_pointer, scan)
IM_close (image_pointer)
```

The open routines opens an image file for read or write of information as described in the mask. The mask can specify what the file contains such as red, green, blue, matte, etc. The image header is an information block whose contents are dependent upon your needs. Critical information includes image resolution and format. Convenience information includes a description, job number, generation date, generating program, etc. The read and write header routines are used to read from or write to the file. The scan read and write routines write single scans into or read them from the files. The close routine closes an image file.

Additional primitives might be added to this set to handle seeking to some scan in a file, transferring a scan from one file to another, etc. In adding any database manipulators, the principle of restricting access to the database to the primitives is the implementation constraint.

Notice that the use of these primitives does not need to know whether the image is a single file or a set of files, does not need to know about the file encoding scheme, does not need to know specifics about image formats, etc. Notice also that if any of the above changes, it does not require changes in any of the code that uses the primitives, only in the primitives themselves.

Table 1 provides a summary of the primitive operations that are necessary for interaction with the devices and databases used in an animation system.

Conceptually, we can apply this technique to manipulating all of the information that is passed through the animation system. For example, in the case of a light source entity, we need to create light databases, we need to query the database to find out where the light is coming from and what intensity it is, we need to be able to write the database to a file and read it from a file, we may need to supply a list of polygons so that the light can be drawn for motion planning, etc.

It is necessary for all of the components of the system to query the script database. Modeling procedures will need to query the JavaScript database about parameters that control the procedure. The rendering system will need to query the script database for geometric objects and transformations, light sources and transformation, data generation procedures and transformations, viewing parameters, etc. Post-processing systems will need to query the scripting system about locations of layers (multi-planing) fade, glow, diffusion, diffraction, and other postprocessing options.

We may treat devices as entities much the same as other data is treated. For example, frame buffers may be treated generically. To access a frame buffer we need to open the device, get information about resolution, initialize the lookup tables, read and write pixels, etc. The use of an environment variable to describe which physical device is being used allows the same program to be run on the same machine supporting a number of dissimilar devices.
In summary, we need to generate a uniform procedural interface to the data that is manipulated by the system so that major components become independent of file handling and databasing for all but their primary task.

<table>
<thead>
<tr>
<th>Table 1 - Summary of Primitive Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>script database primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- add/delete parameter channels</td>
</tr>
<tr>
<td>- add/delete scenes</td>
</tr>
<tr>
<td>- add/delete frames</td>
</tr>
<tr>
<td>- change channel contents</td>
</tr>
<tr>
<td>- change channel assignment</td>
</tr>
<tr>
<td>- add/delete object references</td>
</tr>
<tr>
<td>- add/delete light references</td>
</tr>
<tr>
<td>- change camera parameters</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open database to some frame</td>
</tr>
<tr>
<td>- get next object and transformation</td>
</tr>
<tr>
<td>- get next light and transformation</td>
</tr>
<tr>
<td>- get object references</td>
</tr>
<tr>
<td>- get procedure variables</td>
</tr>
<tr>
<td>- rewind to list beginnings</td>
</tr>
<tr>
<td>- change camera parameters</td>
</tr>
<tr>
<td><strong>material database primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- create initial database</td>
</tr>
<tr>
<td>- input/change spectral curves</td>
</tr>
<tr>
<td>- change other parameters</td>
</tr>
<tr>
<td>- assign illumination model type</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open the database</td>
</tr>
<tr>
<td>- get sampled reflectance/transmittance</td>
</tr>
<tr>
<td>- get other parameters</td>
</tr>
<tr>
<td>- get illumination model type</td>
</tr>
<tr>
<td><strong>texture map database primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- create from an image</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open the database</td>
</tr>
<tr>
<td>- return mapped values</td>
</tr>
<tr>
<td><strong>geometric database primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- create initial database</td>
</tr>
<tr>
<td>- change geometric content</td>
</tr>
<tr>
<td>- add/delete material references</td>
</tr>
<tr>
<td>- add/delete texture references</td>
</tr>
<tr>
<td>- add/delete script animation references</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open and transform database</td>
</tr>
<tr>
<td>- get next polygon</td>
</tr>
<tr>
<td>- get bounding volume</td>
</tr>
<tr>
<td>- get bounding volume intersect</td>
</tr>
<tr>
<td>- get ray intersection</td>
</tr>
<tr>
<td><strong>image file primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- open file write header</td>
</tr>
<tr>
<td>- write scan</td>
</tr>
<tr>
<td>- close file</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open file read header</td>
</tr>
<tr>
<td>- read scan</td>
</tr>
<tr>
<td><strong>light source database primitives</strong></td>
</tr>
<tr>
<td>create/edit:</td>
</tr>
<tr>
<td>- create initial database</td>
</tr>
<tr>
<td>- change color and intensity</td>
</tr>
<tr>
<td>- change goniometric diagram</td>
</tr>
<tr>
<td>- add/delete script animation references</td>
</tr>
<tr>
<td>retrieve:</td>
</tr>
<tr>
<td>- open and transform database</td>
</tr>
<tr>
<td>- get illumination information for a point</td>
</tr>
<tr>
<td>- get next polygon (for wireframe draw)</td>
</tr>
<tr>
<td><strong>frame buffer</strong></td>
</tr>
<tr>
<td>open device</td>
</tr>
<tr>
<td>read/write pixel</td>
</tr>
<tr>
<td>read/write row</td>
</tr>
<tr>
<td>read/write color lookup tables</td>
</tr>
<tr>
<td>get screen info</td>
</tr>
<tr>
<td>draw vector</td>
</tr>
<tr>
<td>close device</td>
</tr>
</tbody>
</table>
Creating a System

The creation of a system is difficult because of the scope and complexity of the project. The generation of a database handling toolkit greatly simplifies the project.

All of the databases, with the exception of image files, can initially be implemented as ASCII files. While this is particularly clumsy, it does allow for the quick generation of an initial toolkit that will support the simultaneous development of the entire system. This implies that some type of language be initially developed for each of the databases, thus aiding in the definition of the ultimate form of the tool that will be used to generate the database.

In my experience, the definition and generation of this toolkit is by far the most important exercise because it shapes all future development. A well planned toolkit will encourage continual system improvement and ease the problems of maintaining the system. A poorly planned toolkit will lead to even the smallest changes requiring alteration of large bodies of code.

Suggested References


A UNIX Image Production Pipeline

Julian E. Gomez, Ph.D
Cranston/Csuri Productions, Inc.
1501 Neil Avenue
Columbus, OH 43201

The Pipeline

This talk describes the image production pipeline in use at Cranston/Csuri Productions. The image production process is viewed as a set of "pipe selections" which are fitted together to form a pipeline, through which the user's data flows on its journey from bits stored on disk to images on the screen. Software implementing the various states of this pipeline are loosely coupled through ASCII interfaces, and the configuration is controlled during run time by the user.

Unified data formats throughout the whole process help ensure a lack of confusion. The object file describes how various items of data are to be combined to make an object for a display program to draw. In this file are pointers to files containing the surface geometry, surface normals, vertex colors, texture maps, etc. The object file is written in ASCII in a tabular form so it can easily be edited by an animator. The detail files containing the geometry, color, etc. definitions are in binary to optimize I/O and image calculation.

The pipeline consists of four stages: date generation, animation, rendering, and postproduction.

Data Generation

Most of the data generation stage is handled by dg, a system implemented by Wayne Carlson. The major features of this system are lofting and surfaces of demi-revolution.

To loft a surface, the user first digitizes a number of cross sections defining the object. dg then connects points on one cross section to points on the next, thereby lofting a surface across the cross sections.

The user can also sketch a profile of some object and have dg revolve that object about its vertical axis to create a surface of revolution. To make things interesting, the user can sketch up to three profile curves which control the revolution process at different heights of the object. The profile is splined with a cubic curve, which is used for the revolution rather than a simple circle.

Animation

Most of the animation stage is handled by twixt, a system implemented by Julian Gomez. twixt uses event driven animation.

In event driven animation, values for particular display functions are seized at various times and splined to create animation. The union of the display parameter's value and the frame it belongs to is known as an event. As the animator collects events for that display parameter, a track of activity is defined. Tracks can be splined with a number of techniques, including linear combination move, parabolic, Overhauser, and the common cubic splines. Furthermore, tracks can be splined with one function as a whole, or different splining methods can be combined over the track's course.

Display functions include the conventional geometric transformations as well as more abstract functions such as surface geometry or orientation matrices. The parameters can be treated individually as well as members of arbitrarily sized groups. A regular expression evaluator as well as an alias mechanism are built into the name parser.

When the animator has created some animation, it is played back on whatever display device the animator is using. This is known as a pencil test. Real time playback is possible only on the
Picture System 300. After some amount of time modifying the script and viewing playbacks, the animator has \texttt{twist} write out a script to \texttt{scn assmblr} telling it how to calculate the animation.

Because the calculation step is so expensive, \texttt{twist} provides the animator with facilities to talk directly to \texttt{scn assmblr} about the current frame, or to prepare a color playback script where frames are computed in a poor resolution, which the frame buffer hardware will later zoom in on as it steps through all the subframes.

Once a track has been defined, it can be copied to any compatible track (e.g., a position vector track can be copied to a color vector track, but a surface geometry track cannot be copied to a position track). Also, any track can be passed through a geometric transform. Thus tracks are not absolute motion, but procedures that can be instanced as necessary.

\section*{Rendering}

The scene rendering process is overseen by \texttt{scn assmblr}, a system described by Crow\textsuperscript{2} and implemented early in 1982. \texttt{scn assmblr} inputs a scene description and analyzes it for \textit{clusters}, or areas where objects overlap on the screen. Each cluster is then described to the proper display program, which is an attribute of the data. Communication from \texttt{scn assmblr} to the display program is through a pipe using sockets.

While running \texttt{scn assmblr} the animator can change the object fields that were originally read in from the object file. This allows the animator to quickly change display programs for faster but lower quality images, or to see if one display program's bug is another's medicine. It also allows quick changes of object attributes to see how it looks displayed differently.

The commonly used rendering program is by Shaun Ho and uses a scanline oriented depth buffer approach. Through a shared scanline buffer, the display program renders the various surface qualities of an object such as its transparency, metallic appearance, reflection and refraction, and also fogginess through a technique known as environmental mapping. Another display algorithm produces (apparent) ray traced images in approximately the same time as the regular display program.

\section*{Postproduction}

The postproduction stage consists mainly of compositing separately computed layers, and of processing layers to add effects such as low pass filtering. Digitized images are mixed into the final images in this stage.

Technically, this state in not necessary, since the optical effects theoretically could be incorporated into the original scene calculation. Practically, however, not all the science needed to do so is available, and in any case, many calculation sequences are cheaper when done separately and composited later.

\section*{Summary}

The power of the image generation process comes from the flexibility of the system configuration combined with a number of programs implementing the different states of the pipeline. The flexibility in turn comes from the loose coupling of the modules and the easy access to descriptions of objects. Implementing this system requires an operating system capable of nice interprocess communication and hierarchical process relationship, along with good software development support; i.e., UNIX.
References


Patchwork: A Dataflow Model for Efficient Graphics Programming

Ronen Barzel and David Salesin
Computer Graphics Project
Computer Division
Lucasfilm Ltd.

ABSTRACT

We have built a system, Patchwork, that allows programs to be organized according to a dataflow model. In our implementation, application programs use Patchwork to assemble complex microcode programs for a graphics processor from a library of microcode modules. We describe a simple and efficient implementation for Patchwork, in which the only overhead incurred is a single extra level of indirection when invoking a module or when a module accesses inputs, outputs, or local storage. The implementation depends on being able to describe a distinct execution tree for the network, which obviates the need both for runtime monitoring of the execution and for movement of data. Thus, neither dataflow hardware nor a dataflow language is needed for the implementation. Patchwork supports flow-of-control constructs such as looping and branching, the assembly of complex modules from simpler ones, modules written in a variety of languages for a variety of different devices, the interleaved execution of several programs on a single processor, and the execution of a single program on a set of processors in parallel. An analysis showed that Patchwork contributed between 2 and 5 percent to the total running time of sample microcode programs.

Introduction

Patchwork addresses a class of problems in which a processing job is set up and then executed in a “batched” manner, without additional input or user intervention. These problems range from jobs that may take hours or days to run, like complicated image processing algorithms, to simple real-time operations, such as updating a “rubber-banded” line as a stylus is moved.

Programs within a given domain generally share common elements. For instance, both a rubber-banding program and a cursor-moving program require code to save, draw, and restore pixels. Thus, it is convenient to be able to construct programs from a set of processing elements.

Patchwork gives the programmer a way to assign inputs and outputs to processing elements and to hook them together into batch programs according to a dataflow model. Although the organizations of the resulting programs are similar to dataflow networks, using Patchwork requires neither dataflow hardware nor a dataflow language. Instead, Patchwork merely provides a way of assembling ordinary structured programs from dataflow-like elements; it can be thought of as the “glue” that holds the elements together.
Motivation

Our original motivation for designing Patchwork was to facilitate microprogramming a high-speed graphics processor called the Chap [11] (Figure 1), which is the basis of the Pixar 2, a framebuffer and image processor [15]. Throughout the paper, graphics programs written for the Chap will be used as examples for motivating and explaining Patchwork. However, as we shall see, Patchwork is actually a general utility that is independent of any particular machine or set of problems.

![Figure 1: Pixar 2 Configuration](image)

The Pixar 2 is being used as a digital film printer that performs many of the functions of an optical film printer. To use the digital film printer, frames of film are scanned in by lasers and stored as digital images in framebuffer memory or on disk. The Chap can then process the digital images in a variety of ways, using a large scratchpad memory for temporary storage. Finally, finished frames are sent back through the lasers to be scanned out on film. Up to eight Chaps may be connected to a single host processor; our current configuration uses a 68000-based host processor which is programmed in C under UNIX.† The sort of processing that the Chap must perform on framebuffer images includes compositing images [13], extracting blue-screen mattes [12], resizing and rotating images [7], anti-aliased painting [19], and run-length encoding and decoding.

Let us take a closer look at some of the programs we wanted to put on the Chap (Figure 2). One thing we wanted to do was to send pictures from the host to the framebuffer. This involves first copying pixels from the host to the scratchpad, and then from the scratchpad to the framebuffer (Figure 2a). We also wanted to send an encoded picture from the host, using a program on the Chap to decode it in the scratchpad and place the result in the framebuffer (Figure 2b). In a still more complicated scenario, we wanted to composite encoded pictures as they were sent to the Chap and place only the composited image in the framebuffer (Figure 2c).

![Figure 2: Some Chap Programs](image)

A simple analysis of these programs made it clear that they could share common elements. For instance, they each required a module to copy pixels from scratchpad to framebuffer ("SF_COPY"), and two of the programs required a module to produce a scratchpad buffer of decoded pixels from a scratchpad buffer of encoded pixels ("DECODE"). Conceptually, the modules had inputs and outputs that could be connected. But how could we best construct programs out of common modules so that they would be easy to write and would run efficiently?

The simplest solution is to make a microcode program for each element and to invoke them in turn from the host. There are two disadvantages to this solution: first, it is inefficient, because of a large overhead in making calls from the host; and second, it ties up the host in making these calls, preventing the host from doing other processing in parallel.

† UNIX is a Trademark of Bell Laboratories.
A more efficient solution is to write a microcode subroutine for each element. A new microprogram can then be written for each operation using routines in the library. This relieves the burden on the host, but transfers it to the microprogrammer, who must now write or modify microcode every time even a slightly different operation is required.

To relieve the burden on the microprogrammer, we want to be able to use a library of subroutines without having to write a microcode driver for each operation. We want to assemble new microcode applications directly from a high level language. These programs should run without intervention from the host, and run as efficiently as if they had been coded explicitly.

The Dataflow Model

Let us examine the dataflow model by comparing it to the more familiar structured programming model, from which it differs in several ways (Figure 3). The basic unit of a structured programming model is the procedure. Each procedure has parameters and may return a value to its caller. A program is organized as a hierarchy of procedures: procedures make calls to lower-level procedures, which may call still lower-level procedures in turn (Figure 3a). To use a procedure, a higher-level procedure must be written to pass it parameters and recover returned values in appropriate ways.

Goals

We had a number of goals in designing Patchwork. First and foremost, we wanted Patchwork to run efficiently; we should not waste the raw speed of our graphics device by heaping software on it, nor should we tie up the host in orchestrating the microprograms. Second, we wanted a design and implementation that was not tied to any particular device; Patchwork should operate on the Chap, as well as on the host and on any other programmable device that may become available.

We wanted Patchwork to support a library of functions; to make it easy to write a new function and add it to the library; to provide a way to create macro functions from existing functions and add them to the library; and to allow application programs to use the library of functions directly and easily in order to assemble complex programs at runtime.

Finally, we wanted to support programs that could run across several processors simultaneously. Conversely, we wanted to allow several different programs to run at the same time on a single processor.

Figure 3: Structured and Dataflow Programming

The basic unit of a dataflow programming model is the module. Each module has inputs and outputs. A program is organized as a network of modules. The modules are linked together by the data connections between their outputs and inputs. A module is invoked when all its inputs are filled; once invoked, it may produce outputs, which in turn are used to fill the inputs of subsequent modules (Figure 3b). To use a collection of modules, the modules' outputs and inputs are connected to form a network, or patch. A single output may be connected to many different inputs. A module may be used more
than once in a patch by creating separate instances of the module. Note that a dataflow organization does not preclude the use of traditional structured programming in designing the dataflow modules.

For our purposes, dataflow is the more useful abstraction for several reasons. By requiring formal descriptions of each module’s inputs and outputs, dataflow allows us to place modules into a library and link them together in an automated way. Moreover, whereas all a procedure’s parameters must be provided by the one procedure that calls it, a module’s inputs may be supplied by any number of other modules. Finally, unlike procedures, which make explicit calls to lower-level procedures and are thus bound to them, modules have no explicit knowledge of each other and can therefore be mixed and matched freely.

Previous Work

Most research in dataflow has been concerned with dataflow as a model for parallel processing and with the design of languages [9] and machine architectures [10] to support it. The Evans and Sutherland PS300 [11] is an example of a machine that uses a dataflow architecture for setting up and performing transformations on graphical objects.

Some work has also been done in implementing a dataflow model as a useful programming abstraction, which is more like our work. Babb [5] and Yamano and Matsumoto [20] used dataflow as a model for software engineering. Cross addressed the issue of formally describing modules in terms of inputs and outputs, although not as part of a dataflow environment [8]. Abbott wrote a dataflow system, Fm, for orchestrating networks of digital signal processing modules on an audio signal processor [2].

In the graphics realm, Anson introduced a hierarchical input-output formalization for describing graphical input devices [3]. And van den Bos et. al. generalized the input-output formalization further, introducing activation expressions and if-then-else type constructions [6]. Vickers implemented a three-dimensional graphics pipeline as a network of connected modules [18]. And Strauss and Barzel generalized this system to support a dataflow model for general-purpose programming in a high-level language [17].

A fundamental difference between this previous work and our own is that the previous dataflow models really do implement a dataflow environment—modules are ultimately invoked whenever their inputs are filled—whereas our work uses dataflow as a programming abstraction but abandons it at the implementation level; as we shall see, our modules are actually invoked by run-time choices made by the code.

Concepts

We will use a program for the Chap as an example to help make some Patchwork concepts more concrete. A patch to combine two framebuffer images and place the result in a third framebuffer area would use three primitive modules: FS_COPY, which copies some number of pixels from a scanline to a scanline buffer in scratchpad (Figure 4a); SF_COPY, which copies pixels in the other direction (Figure 4b); and MERGE, which composites foreground and background scanlines in scratchpad, placing the result in a target scratchpad scanline (Figure 4c). Instances of these modules can be connected into a patch that composites two framebuffer scanlines and deposits the result into a third framebuffer scanline.

Each primitive module is implemented by a microcode routine. The routine has access to its inputs, outputs. It must be re-entrant in order to allow multiple instances of the module. A module is automatically called at the appropriate time; its routine does not have to worry about the high-level flow of control. Within a module, good structured programming techniques apply; typically, the routine that deals with the interface is separate from the routine that implements the module’s function, which it calls as a subroutine after setting up parameters. This subroutine is thus independent of the entire Patchwork system.

To create a patch, the application program
Figure 4: Some Primitive Modules

Figure 5: Macro Modules

Design Decisions

Execution Sequence

The use of a non-multitasking machine forces modules to be executed serially rather than in parallel; thus, the invocation of a patch can be represented by an execution sequence of modules. If we impose the restrictions that each input is tied to a single output, and that each module produces a single value for all of its outputs every time it is invoked, then the data movement within a patch becomes predetermined. If we impose the additional restriction that there are no circular data paths, then we can sort the modules based on the partial ordering implied by their data connections and produce an a priori execution sequence of modules in a patch. Note that this sequence is not necessarily unique; often, more than one sequence may exist which can satisfy the dataflow constraints (Figure 6).

The fact that there is an a priori execution sequence is key to an efficient and simple implementation. Patchwork can chain together all the microcode modules so that they can run in sequence on the Chap with minimal overhead in getting from one to the next, and without any interaction with the host.
Our restrictions have eliminated many of the subtleties that often accompany dataflow implementations. We do not need any runtime monitoring of the data movement [17], nor do we require methods that buffer or tag the data to handle such things as dataflow synchronization, looping, or recursion [4,16].

**Execution Phases**

In our application, a patch is typically used repetitively (e.g. once for each scanline). Thus, it is advantageous to separate initialization overhead and to perform it just once for the entire patch in a Setup phase, which precedes the repetitive Go phase, in which the patch’s function is executed.

During the Setup phase, modules may allocate buffers and squirrel away initial results for later use during the Go phase. Once the Go phase is complete, a third phase, the Cleanup phase, is executed in which modules may free any dynamically allocated storage. Each module is therefore implemented by three routines, one for each phase, which are called by Patchwork as needed.

**Data Movement**

Since all modules on a given processor can share memory, there is no need for physical data movement or for mechanisms to support it; instead, two modules can simply communicate through a shared data buffer. Moreover, since an a priori execution sequence makes it unnecessary to monitor the movement of data, it is unnecessary to notify any process when data is passed; thus, unmonitored communication through a shared data buffer is sufficient for our simplified dataflow implementation.

We want to be able to create a network of data connections by allocating these shared data buffers and pointing modules at them before the patch is executed. But the details of the connections may depend on run-time initialization parameters; for example, the size of the buffer might depend on a "scanline width" parameter.

Thus, it becomes Patchwork’s job to create a Setup network by allocating buffers in which initialization parameters can be passed. It is then the job of the modules’ Setup routines to use these parameters to construct a Go network by allocating the buffers used by the Go routines and by passing pointers to these buffers through the Setup network (Figure 7).

During the Go phase, modules communicate through the Go network. If the Go network
has to change at run-time, a Go module may send new parameters through the Setup network and re-invokes the Setup phase to reconstruct the Go network.

Explicit Execution Sequence

All sequences derived from the data connection network are satisfactory for executing the Setup phase and, generally, for executing the Go phase as well. But in some cases, additional constraints rule out some of the possible Go phase sequences. For instance, a patch that swaps two areas of framebuffer must be executed such that both areas are copied out of framebuffer before either is copied back in (Figure 8).

By default, Patchwork derives possible execution sequences from the Setup network, and chooses one from among them. However, it is possible for a patch description to override this process and specify a sequence explicitly.

Execution Tree

We originally postulated that modules would be able to produce all their outputs every time they were invoked, and this led to the existence of an execution sequence. But suppose there existed a module that could produce only one of several possible outputs, or none at all, every time it was invoked. The execution of such a module would need to be followed by one of several alternate execution sequences, depending on which output, if any, had been produced.

The construct of a single execution sequence branching out into several can be described by an execution tree. A module that chooses among subsequent execution sequences at run-time is called a branching module. The external description of a branching module names the alternate branches; the module’s routine chooses a branch by name when it exits. Note that a branching module still has no knowledge of the rest of the patch; the next module called when a branch is taken depends entirely on how the patch has been configured.

An example of a branching module is a module that decodes run-length encoded pixels a scanline at a time. If the module successfully completes a scanline, it can pass execution along the normal sequence. But if the module runs out of encoded data before completing a scanline, it must exit along an alternate execution sequence, which is typically configured to provide the module with more data (Figure 9).

Note that by allowing an execution branch to merge back into the tree, we have provided a looping construct. (Here, the “tree” is really a directed graph.) An ordinary “for-loop” can
be implemented by a module that is topologically equivalent to DECODE_HOST; the module exits along one branch each time it is invoked until a certain count is reached, then exits along the other.

**Implementation**

*The Instance Block*

At the heart of the implementation is a data structure called the instance block. For the Chap, the instance block resides in scratchpad memory. Each instance block corresponds to a single instance of a module. Instance blocks are linked together to form the Setup and Go execution trees. An instance block contains:

- pointers to the module's Setup, Go, and Cleanup routines;
- pointers to the next instance block (or blocks, for a branching module) in the Go and Setup execution trees;
- storage for a module's outputs;
- storage for a module's default inputs;
- pointers to a module's inputs;
- and storage for a module's local static data.

An input and output are plugged together by pointing a module's input directly to the storage allocated for the corresponding output in the instance block of the outputting module. Note that any number of inputs can point to the same output, but a given input can only point to a single output. If a module's input has not been plugged to any output, it will point to a default value stored in the same instance block.

Multiple instances of a single module are implemented by allocating multiple instance blocks, one for each instance, that all point to the same set of routines.

**Using Patchwork**

There are two steps in creating a Patchwork module: writing the external description, and writing the internal code.

The external description of a module is written in a simple description language [14]. This description includes the name of each of the module's inputs, outputs, and execution branches, as well as optional default values for inputs. Patchwork's parser reads the external description and produces a number of files which have the effect of adding a description of the module to the function library.

The code that implements a primitive module is written in the usual programming language for the module's machine. Patchwork provides programming macros for accessing the fields of the instance block symbolically. For instance, one macro provides access to the inputs (given an input name, it follows a pointer and returns the input data). Another macro provides access to the outputs (given an output name and a value, it places the data in the output storage location). A third macro is used to exit the routine (given the name of a branch, if there is one, it follows the pointer to the next instance block in the tree and returns the pointer to that instance's routine).

No code needs to be written to implement a macro module. Instead, the description language is used to describe the patch that makes up the macro, as well as mappings between the inputs and outputs of the macro, and those of its component modules.

Once a module is created, it is available to application programs through Patchwork's run-time package. When run-time calls are made to set up a patch, Patchwork will allocate an instance block for each module and point each block at its appropriate routines. Patchwork will also set up pointers in the instance blocks to create the execution trees for the Setup and Go phases and to create the Setup data network.

Another run-time call causes Patchwork to invoke the Setup, Go, and Cleanup phases in turn. A microcode sequencer is used in our
Chap implementation to chain between modules in the execution trees. The sequencer simply branches successively to locations returned by modules' exit macros. The use of a sequencer provides several advantages for debugging over chaining the modules together directly. For instance, the sequencer can be used to single-step through a patch a module at a time, and to test global error codes after each module returns.

**Additional Features**

*Multiple Machines:* Patchwork is written to support modules for any number of different devices. Each device has its own instance block structure and programming macros analogous to the ones described for the Chap. In addition, Patchwork allows modules to run on different devices within a single patch. At each device transition, Patchwork inserts a "placeholder" module (Figure 10). When invoked, a placeholder module halts execution on the current device, passes output data to inputs on the next module's device, and invokes the next module on its device. The example here shows execution proceeding from the Chap to the host and back again.

![Figure 10: Placeholder Modules](image)

*Host Modules:* One important device supported by Patchwork is the host machine itself. Routines for host modules are written in C. Being able to execute modules on the host allows modules that move blocks of data between the host and Chap, as well as modules that make use of operations (such as floating point) that are not supported in the Chap. In addition, it allows algorithms to be implemented and tested at a high level first, and later migrated to microcode transparently.

*Parallelism:* Although using a fixed execution tree prevents the kind of automatic parallelism usually associated with a dataflow machine, we can still take advantage of several other forms of parallelism. First, a patch that runs entirely within an auxiliary device will always be executed in parallel with a controlling process on the host; for instance, a patch that runs in the Chap to draw rubber-banded lines will automatically execute in parallel with tablet polling on the host. Second, in a configuration with more than one device, each device can be executing independent patches in parallel; meanwhile, the host can poll the devices and assign a new patch to each device as it becomes free. Third, patches can be pipelined across several devices; special data movement modules can be configured into the patch explicitly to pass data down the pipe.

*Interleaved Execution:* The execution of several patches can be interleaved on a single device at the same time. The trick is to make the sequencer itself instantiable, and to use one instance of the sequencer for each patch that is running (Figure 11). The sequencers are chained together. Each one exits after executing a single module of its patch, allowing the execution of all the patches to be interleaved on a module-by-module basis.

![Figure 11: Multiple Sequencers](image)

*Subroutine Patches:* In an extension to our execution model, we have found it useful to allow a module to be able to call an entire patch as a subroutine. For instance, a module that produces a stream of data when it is invoked needs to invoke the rest of the patch one time for each piece of data in its output stream. Such a module can make a subroutine call to a separate instance of the sequencer to execute the rest of the patch. Note that care must be taken in configuring subroutine patches; a subroutine patch may expect input only from its calling module.
Analysis

We have been able to implement a variety of applications—from compositing images to anti-aliased painting to run-length encoding and decoding—all within Patchwork's dataflow framework. Moreover, we feel that this dataflow scheme can be useful in many areas, both in and outside of computer graphics. Since rendering involves passing graphical primitives as data from one process to another (e.g. a hider, a shader, a texturer) we expect to be able to use Patchwork to implement the rendering pipeline for new 3D hardware that we are developing. The same dataflow scheme would also work well for a viewing pipeline; in fact, Patchwork can be thought of as a generalization of the viewing scheme presented in Vickers [18]. Patchwork might also work nicely for digital music processing. Currently, analog sound engineers work with large "patch panels," in which oscillators and filters are plugged together physically. Patchwork could perhaps be used, like Abbott's Fmx [2], to model this world directly, allowing logical connections between digital oscillators and filters.

Patchwork has met our goals well. Programs written with Patchwork have been assembled from a library of small modules that performed specific functions and were thus relatively easy to write. The implementations are efficient, as the only run-time overhead incurred is a single extra level of indirection when invoking a module or when a module accesses inputs, outputs, or local storage. Since the overwhelming proportion of the time is spent executing modules' routines rather than chaining between them, and since a module's routine can load its inputs and local variables into registers just once before it begins executing, this overhead is small.

We performed a simple analysis of the running times for a framebuffer-to-framebuffer copy (Figure 7) and a MERGE_FB macro (Figure 5) for sets of 1000X1000 framebuffer images. Our analysis revealed that the overheads for executing Patchwork-related code in our implementation were 5% and 2% of the total running times, respectively. In general, the proportion of the time spent in Patchwork decreases as the complexity of routines in a patch increases or as the amount of data being processed increases.

Conclusion

Patchwork does not implement a strict dataflow model. It is actually a hybrid system, incorporating ideas from structured programming as well as from dataflow. The organization of Patchwork programs is modeled after dataflow networks, which gives us the freedom to configure patches easily. Execution, on the other hand, is modeled after structured programming, with flow-of-control choices being made at run-time by the code. We traded the support for parallelism and asynchronicity of a data-driven execution method for the efficiency and higher degree of control of a predetermined execution sequence.

Although Patchwork was designed to implement dataflow constructs, we neither built new hardware nor implemented new programming languages to support it; conventional programming tools were all we used. Thus, Patchwork could be viewed as a utility for writing ordinary structured programs with dataflow constructs. A strength of this approach is that programmers can still write code in a programming language they are comfortable with, and that is efficient and well-understood; they do not have to learn a new language to use it.

Acknowledgements

Sam Leffler wrote all the software tools for working with the Chap. Tom Porter, Bob Drebin, and Steven Baraniuk wrote the microcode modules. H.B. Siegel implemented the code for the host-Chap interface and for Patchwork's Chap utilities. Finally, we also wish to thank Curtis Abbott, Rob Cook, and Mike Haulsey for their help on early drafts of this paper.
Bibliography


The Alpha_1 Computer-Aided Geometric Design System in the UNIX Environment

Spencer W. Thomas
University of Utah

ABSTRACT

The Alpha_1 system is a spline-based geometric modeler. Solid objects are modeled by representing their boundaries as B-splines. Objects may be combined with set operations into more complex objects. Several modeling paradigms are supported, including direct manipulation of the B-spline surface, sweeping operations for surface creation, creation and combination of primitive shapes, and high-level shape operators such as bend, twist, and warp. The single underlying mathematical formulation simplifies implementation, but is sufficiently powerful to represent a very broad class of shapes. The system is able to create images of the designed objects, to perform certain analysis functions on them, and to produce numerically-controlled machining information for manufacturing.

Alpha_1 was developed and runs under the UNIX operating system. Various features of UNIX aid both in the development and use of the system. For example, the various text manipulation tools available under UNIX made possible the development of sophisticated program generation aids. The command language of the Alpha_1 system is a set of C-shell aliases – no special command parser is necessary. The use of standard input and output streams by UNIX programs, and the easy process creation possible under UNIX let us run our geometric editor as a subprocess of the text editor Emacs, thus automatically providing a record of program execution, editing facilities for the command input, and a history mechanism. We have found UNIX to be a near-ideal system for our application.

Introduction

The Alpha_1 project started in 1980 with the goal of building a spline based solid modeling system [1]. Most commercially available geometric design systems at the time were little more than glorified drafting tables. Some had the capability of drafting in three dimensions, producing “wire frame” drawings. Drawings are, however, inherently ambiguous; Figure 1-1 illustrates this with an example of a wire frame drawing that can be fleshed out into a solid object in several different ways.
Solid Modeling

At the time, research was well advanced at several locations into modeling solid objects as solid objects, rather than as pictures. Ian Braid and his colleagues at Cambridge University developed the Build system in the late 70's [2,3]. The Production Automation Project at the University of Rochester, headed by Herbert Voelcker and Ari Requicha had developed the PADL system by this time [4,5]. Both of these systems modeled solids as a combination, by means of the set operations union, intersection, and difference, of certain primitive shapes, such as "boxes," cylinders, spheres, and so on. This style of modeling is commonly called constructive solid geometry (CSG), and is able, according to Requicha [6], to model "95% of conventional, unsculptured parts."

The Build and PADL systems differ, however, in their methods for representing solids. Build uses a boundary representation, while PADL uses a volume representation. That is, Build stores the boundaries of the modeled solids, while PADL stores a representation of the sets of points which make up the objects. Both representations have their advantages and disadvantages. Advantages of a volume representation include:

- It is easier to ensure that the model represents a manufacturable object, and
- It is easier to perform set operations.

Disadvantages of a volume representation include:

- It is more difficult to make images of a model,
- The model must typically be kept in an unevaluated form, as primitives combined by set operations, and
- Modeling complex shapes is difficult or impossible.

Advantages of a boundary representation include:

- It is easy to make pictures, since the boundary is what is seen,
- The boundary of the result of a set operation may be calculated explicitly, so the model may be kept in an evaluated form, and
- The availability of a wide variety of surface description schemes make it possible to model complex, sculptured objects.
Disadvantages of a boundary representation include:

- Calculating the result of a set operation is difficult, and
- Ensuring that a boundary really bounds a manufacturable object is difficult.

**Goals of Alpha_1**

The major goal of the Alpha_1 project was to produce a spline based, boundary representation solid model that could be used to model the large class of sculptured mechanical parts which were not representable by a CSG model. Objects which fall into this category\(^1\) include turbine blades, airplanes, helicopters, automobiles, sewing machine parts, shoes, automobile engine blocks, silverware, and so on. Even parts that have, at first glance, a simple geometry cannot be accurately modeled by a CSG approach. A good example of a part in this class is a bolt. Although it is basically a cylinder surmounted by a hexagonal prism, proper modeling of the threads requires a complex surface shape. Of course, the first representation is probably suitable, except when stress or failure analysis of the bolt is desired.

Alpha_1 also proposed to demonstrate the utility of high quality graphics in the mechanical design process, particularly when dealing with sculptured surfaces. Proper perception of a complex shape is not possible from a line drawing or rough shaded rendering. Almost all perceived information about the shape of a sculptured object comes from the highlights caused by specular reflections from the surface. Without an accurate lighting model and a high resolution image, the highlights, and thus the perceived shape, will be distorted. Surface ripples that might be invisible in a line drawing show up immediately in a shaded raster image, and errors in design can be corrected before they propagate to the costly machining stage. A good set of pictures of a design can sometimes substitute for expensive scale mock-ups, again saving time and money.

Another goal of the Alpha_1 project was to incorporate “computer science principles” in the software design. This includes, but is not limited to, use of a modern, structured language, and some adherence to the concepts of data abstraction, object oriented programming, modular design, extensibility, and portability.

The end result of the initial Alpha_1 effort was to be, not a finished computer-aided geometric design system, but an experimental testbed on which new modeling ideas and techniques could be easily mounted and evaluated. It should be, for example, easy to insert new surface representations into the system as they are developed. On the other hand, it is not expected that it would be possible to test a volume representation under Alpha_1 without a fair amount of work.

**B-Splines in Alpha_1**

The initial surface representation upon which the Alpha_1 system was built, and the only surface type supported by the entire system, is the rational, nonuniform, parametric tensor product B-spline surface. These have the mathematical form

\[
F(u,v) = \frac{\sum P_{i,j} B_{i,n} (u) B_{j,m} (v)}{\sum w_{i,j} B_{i,n} (u) B_{j,m} (v)}
\]

where \( B_{i,m} \) is the \( j^{\text{th}} \) B-spline basis function of order \( m \), and is piecewise polynomial. The shape of a B-spline surface is controlled by the position of the control points \( P_{i,j} \) and by the weighting factors \( w_{i,j} \) [7].

B-splines have a number of desirable properties for use in computer-aided geometric design. As a spline, a B-spline curve\(^2\) will exhibit a specified degree of continuity; for example, cubic B-splines are \( C^2 \). This ensures that a curve which should be smooth is indeed smooth. On the other hand, by properly manipulating the knot vector, discontinuities of specified degree may be inserted in the curve.

---

\(^1\)A number of examples of such objects are illustrated in the slides.

\(^2\)Although the following discussion deals with B-spline curves, all the same properties hold for B-spline surfaces.
B-splines have the convex hull property: the curve always lies within the convex hull of the control points. It is therefore easy to bound the curve, simplifying the calculations such as those required for producing an image or for intersecting it with a ray or another curve.

Another feature of B-splines that is useful in a design system is that they provide local control. Changing the location of a single control point modifies the shape of only a portion of the curve; the rest of the curve remains unchanged. This allows, for example, a designer to shape one region of a curve without affecting other regions that may be already complete.

The use of rational B-splines allows the exact representation of a large class of shapes, including the common quadric surfaces such as spheres, cylinders and cones. A non-rational B-spline, since it is a piecewise parametric polynomial curve or surface, can reproduce a circular or elliptical shape only approximately. By adding a weight or homogeneous component at each control point, it is possible to model these shapes exactly. Since many common mechanical parts have circular cross sections, it is critical that a design system be able to reproduce such shapes.

A B-spline may be evaluated by computing the above summation at any given point. Often, however, an approximation to the entire curve or surface is desired. This may be found via a subdivision or refinement process. The Oslo Algorithm [8] provides a method to take a B-spline and compute a new B-spline, point for point identical to the original, but with more knots, and therefore more control points and more polynomial spans. It can be used to split, or subdivide, a B-spline into two pieces that, together, make up the original. The new control polygon, formed by joining the control points in order, more closely approximates the curve than did the original. Thus, by adding enough new knots, the control polygon itself may be used as an approximation to the curve.

A Short Description of the Alpha_1 System

The Alpha_1 system provides a number of capabilities. Some of these will be described in more detail below. A short listing of the features of Alpha_1 includes:

- The ability to perform set operations to combine objects into more complex assemblies,
- High quality graphical renderings of object models,
- An interactive, extensible geometric editor,
- Drafting style geometric construction operators,
- High-level shape modification and design operators,
- Automatic creation of primitive objects, and
- Computation of simple mass properties of modeled objects.

In addition some capabilities are partially implemented; their complete implementation is a matter of current research. These include:

- An interface to finite element analysis,
- Generation of numerically controlled machining information, and
- Inclusion of non-geometric data in the model.

Set Operations

The ability to perform set operations, usually union, intersection, and difference, on models is recognized as one of the characteristics of a true solid modeling system. Set operations perform a number of useful functions in the modeling process. They can be used to combine simpler objects into more complex ones. It is not necessary to design an entire object in one piece; it may be broken down into simpler components that can be designed independently. Thus, set operations provide a facility for modular design. Set operations may be used to model certain common machining operations. For example, set difference models material removal processes such as drilling or milling. Set union may be used to model gluing or welding type operations.
In general, different parts of an object must perform different functions, and may be designed by different techniques. The regions where the different parts meet are not explicitly designed, but arise from the interaction between the independently designed pieces. Using set operations to combine the parts produces the interface region automatically. A good example of this interaction occurs in a turbine blade, where the airfoil portion of the blade must be designed to meet certain aerodynamic or hydrodynamic constraints. The root of the blade is designed to hold the blade into the turbine during operation, and must meet an entirely different set of mechanical constraints. The airfoil must, however, be attached to the root, and a set operation is the ideal way to model the merger of the two.

Performing set operations in a boundary modeling system requires computing the boundary of the result of the set operation. The boundary of the result must be composed of pieces of the boundaries of the operands, and these pieces must be bounded by intersection curves between the boundaries of the operands [9,10]. Unfortunately, computing the intersection curve between two B-splines in a closed form is currently an intractable problem; the intersection curve between two bicubic surfaces is an implicit polynomial of degree 324. The intersection can, however, be approximated to any desired precision.

The Alpha_1 system has the ability to perform set operations on partially bounded sets. As long as they satisfy certain loose criteria, boundaries of objects to be combined by set operations need not be closed. This is a nice feature for the designer, who is not arbitrarily forced to close object boundaries just so that set operations may be performed. A good example of this is seen in the design of a turbine blade, where none of the boundaries of the subpieces completely enclose a volume, but the final model does have a closed boundary.

Graphics

A key concept put forth by the Alpha_1 project is that design of mechanical parts and, in particular, of sculptured mechanical parts, is greatly aided by high quality graphic rendering of the model. The shape of the model can be perceived more accurately and mistakes in the design found sooner if the designer can better visualize the model being created.

Towards this end, the Alpha_1 system incorporates a scanline rendering program that can produce images of B-spline surfaces. It adaptively subdivides surfaces until the pieces are flat, to a given resolution, producing polyhedral approximations. These are then rendered traditionally. A simple supersampled anti-aliasing scheme is used. The program has many options to control the rendering process, the lighting model, surface characteristics, and output form. It is possible to specify transparent or semi-transparent surfaces, allowing the designer to see interior details as well. Control over lighting is also provided, to allow careful placement of highlights to give maximum shape information.

Several support programs have been created to aid in creating good images. The view program uses an interactive line drawing display to properly position objects for later rendering. The user moves the image of the object around by manipulating knobs. When the desired view is achieved, the program outputs a transformation matrix that the rendering program will use to create the same view in the shaded image.

Another program lets the user manipulate lighting and surface parameters using a specially computed raster image. The position of a light may be varied using knobs or the data tablet, or by indicating the desired position of a highlight. Several lights may be simultaneously positioned, since a single light rarely provides a satisfactory rendering of a complex shape. The reflectance characteristics of the object surface may also be varied dynamically.

Recent acquisition of a raster display with a built in Z-buffer prompted the creation of an incremental rendering program. Surfaces are displayed and redisplayed with successively greater degrees of refinement. This provides a quick-look capability; a user need only run the program until the desired accuracy level is reached.

An experimental ray tracing program is being implemented. Ray tracing provides the ultimate in realisting imaging, but it is not clear yet whether the gain is worth the extra cost in this application.

---

3The slide set contains a picture showing the pieces that go into the design of the root of the turbine blade.
Further experience with ray traced versus scanline images is necessary.

Experience of members of the Alpha_1 project and others has indicated that the incorporation of quality graphics into the design process is desirable. In one case, an expensive design mistake that was not discovered until the first test machining run could have been averted if a good, shaded rendering of the model had been available. Complex shapes that would have been virtually impossible to recognize in a line drawing are easily viewed in a realistic image. For industries, such as the automobile industry, where surface reflections are an integral part of the design, such renderings are invaluables.

**Geometric Editor**

The Alpha_1 geometric editor, *shape_edit*, is currently implemented in PSL.\textsuperscript{4} Lisp provides a rich interpretive environment in which arbitrary and extensible data structures may easily be created. The embedding of the editor in a language leads easily to the concept of the program as the model, rather than the output created by the program. It is also easy, therefore, to create a parametric model, one that can represent a family of parts by changing a set of parameters.

*Shape_edit* extends PSL to understand a several different graphics devices, and how to display geometric data on them. An assignment operation is provided that automatically displays the result on the graphics screen. A multi-window environment is created on the screen, and graphical interaction is supported on those devices on which it is available. It currently runs on such diverse devices as Apollo workstations, a Megatek line drawing system and the E&S PS300.

The only geometric editing environment currently supported is the programming environment. To build a model, one must write a program or set of functions that create it. While this may be suitable for computer scientists, it is not an interface usable by most designers. Research is in progress on a menu driven, graphical editing environment. Obviously, some flexibility is lost, and it is possible to revert to the programming environment if necessary. Or, since the graphical interface creates a *shape_edit* program, it may be used in an initial “sketching” phase. The program thus created can then be edited to get the desired result, or to produce a parametric model.

*Shape_edit* provides many tools and operations for shape definition; these are discussed in more detail below.

**Drafting Operations**

*Shape_edit* supports the drafting paradigm of design by providing a number of drafting type operations that deal with points, lines, and arcs in the plane. Points can be constructed explicitly, by supplying their coordinates. A point can also be determined by intersecting two lines, as an offset from another point, as a blend of several other points, and so on. Lines can be constructed parallel to the coordinate axes, through a pair of points, through a point parallel to a given direction, and so on. There are many ways to create circular arcs. Three points determine an arc, as do two endpoints and a center point.\textsuperscript{5} Arcs may be constructed tangent to three lines, to two lines with a given radius, or to a circle and a line, with a given radius. The total selection of constructor functions is limited only by the imagination of their creators.

Arcs and line segments (determined by a pair of points) may be chained together into a B-spline curve. Curves can be used to construct surfaces in many ways. The simplest, but the one requiring the most direct knowledge of B-splines, is to define the surface control points directly from the control points of several curves. A more useful operator sweeps one curve along another to form a surface, or revolves a curve about an axis for a surface of revolution. Or, several curves may be used as the outline of a surface, with the interior filled in by a boolean sum operation.

Points, arcs, curves, and surfaces may be imported into other coordinate systems after being defined. This makes it possible to always draft in the X-Y plane, and to put the result in the correct orientation afterwards. This models standard drafting practice.

\textsuperscript{4}Portable Standard Lisp

\textsuperscript{5}Actually, this over-determines a circular arc, but correctly determines an arc of an ellipse.
Shape Modification Operators

To simplify the task of working with B-spline surfaces, high level shape modification and design operators have been created. For example, the *bend* operator takes a previously defined surface and bends it. A thickening operation starts with a single surface, computes a surface that is offset from it by a given amount, and joins the edges of the two to form an object shaped like the original surface.

An interesting shape modification operator is *warp*. It is used to put bumps in surfaces. The basic warp operator puts a round bump of a specified size into a surface. The shape of the bump can be varied from square (parallel to the parametric directions) through round to a diamond shape, and the cross section from a box shape to a narrow spike. Another variation, called *skeleton warp*, essentially uses a curve to emboss a bump into a surface.

Other shape modifiers include *twist*, *flatten*, *taper*, and *stretch*.

Primitive Objects

To accommodate models created by CSG systems, or for those cases when primitive shapes are desired, *shape_edit* contains routines for creating some basic objects. These include boxes and pyramids with various numbers of faces, spheres and ellipsoids, cylinders and cones, and tori. The surfaces of the primitives are represented as B-splines, so any of the aforementioned shape modification operators can be applied to them.

Real objects often have rounded corners, instead of the ideally sharp angles on the primitive shapes. Therefore, provision has been made to create versions of some of the primitives with rounded corners. In particular, cuboidal shapes with rounded edges and corners are supported. The radii need not be identical on all edges, and the faces need not be at right angles. This capability is especially useful when modeling molded parts, since they typically have rounded edges, and must have nonparallel sides so they can be extracted from the mold.

Finally, special purpose routines can be written to model any desired family of shapes. An example of this might be screw threads. Screw threads must follow certain standards for their cross-sectional shape, and for their spacing. Metric and American threads differ in shape as well as sizes. However, all the necessary knowledge can be built into functions, and the user need only ask for, for example, a 2 inch 1/4-20 bolt.

Alpha_1 and UNIX

The Alpha_1 project has gained immensely from the use of the UNIX operating system. The availability of C-shell aliases made it unnecessary to write a separate command parser. The text processing tools provided by UNIX made building source modification and control tools relatively easy. Parser generation tools were used to create a parser to read data files, and to write programs that automatically generate code from an object description. Pipes made it easy to plumb together separate programs for performing various tasks. And later on, (Gosling) *Emacs*, with it subprocess control, provided a perfect editing front end for *shape_edit*. In the near future, networking support will allow Alpha_1 applications to be distributed over several machines. But, in the end, is UNIX really necessary for Alpha_1?

Aliases as a Command Language

The command language that drives all the Alpha_1 programs except *shape_edit* is written as a set of C-shell aliases. This is both a blessing, and a curse. It is a curse because it takes forever to read all the aliases into the C-shell,\(^6\) and a blessing because it was not necessary to write a command parser, nor to remember all the different flags that each program takes.

The command for running the render program is typical. A top-level command *render* invokes a data conversion program *conv* and pipes the result into the *render* command. The first argument is passed to *conv*, and the rest to the *render* program. Typically, though, only one argument, a list of input

\(^6\)This is partially circumvented by defining the important aliases to load the files containing their real definition.
files, is supplied, because all the flag arguments are stored in C-shell variables and passed to the render program by the alias. Note that both the directory on which the render program lives, and the name of the program itself are easily modifiable. This aids greatly in testing new or experimental versions of the program.

```
render =>
  conv !:1 | rendercmd !:2--*
rendercmd =>
  $rndir/$renderprg $shader $hilightflg $shademode $ovlyflg $bufflg $cullflg $lightflg $flipflg $diskzflg $bkgndflg $vwportflg $transflg $opacityflg $quiltflg $dblresflg $visflg $segsflg $aspect $filterflg $clrinterpflg $highlightmode
```

The conv alias is somewhat interesting; it demonstrates a method of passing several files as one argument, and also of being able to pick up the files from a preselected data directory. The variable datadir is set to the name of the directory where the data files are located, with a trailing '/'. To get files from the current directory, or from several directories, datadir is set to the null string. A list of files separated by commas as the first argument to conv will then be properly expanded as arguments to cat.

```
conv =>
  cat $datadir(!:1) | convcmd !:2--*
```

All flag setting is done through other commands. For example, the aliases below allow the user to set some of the lighting options for the rendering program. The variable names that start with 'X' are set to a human readable string describing the value of the particular attribute. The variables without an 'X' are set to flag values to be passed to the program. A state command echoes the values of the 'X' variables, so the user can see the flag settings. The default value echoes nothing, so a show_defaults command is provided.

```
blinn     set shader=""; set Xshader='blinn'
cosine    set shader='-C'; set Xshader='cosine'
cornell   set shader='-I'; set Xshader
white     set hilightflg='-w'; set Xhilightflg='white'
ownwhite  set hilightflg='"'; set Xhilightflg
smooth    set shademode='g'; set Xshademode='smooth'
flat      set shademode='"'; set Xshademode='flat'
normal    set shademode='-n'; set Xshademode
dynamic   set shademode='N'; set Xshademode='dynamic';
```

This mode of operation has worked quite well, but is beginning to become unwieldy. It was recently extended to execute commands on remote machines, over the network. At first, it seemed that the only change necessary was to set the execution directory string for a command to be

```
  rsh remotemachine bin-directory
```

However, when using this scheme with multiple stages in a pipeline, the pipe snakes back and forth from the remote machine to the local machine and back to the remote machine for each stage. The compromise adopted was to run only the major commands on the remote machine; they are the CPU hogs, and there is usually only one per pipeline.

Another problem with this approach is that it eats up the command name space. If a useful program named, say, smooth, was released, it would be necessary to change either its name, or the name of the alias for the rendering command. An older version of render was called scan, a name that conflicts with a program in the MH mail reading package. The MH program was renamed mkscan. Furthermore, the profusion of variables makes the output of the C-shell set command almost useless. Still, the use of aliases (or ksh functions?) to provide a command interpreter is a good technique; and the name space problem is a general problem that is merely exacerbated by the profusion of aliases.
Text Tools

The presence of text processing tools such as *awk, sed, grep, join*, etc. encouraged the production of shell scripts to do many useful tasks within the system. A source code control system was built that provides a central database and audit trail, and prevents hasty modification of working sources. Lately, it was extended to provide automatic change distribution to remotes sites running Alpha_1 software. All changes made at the development site are automatically sent to all dependent sites. Any changes made at a dependent site are sent back to the development site for approval before it is actually installed.

An *awk* script reads a list of source directories to produce a shell script that will perform any one of a number of actions to every directory. This is run every night to *make* anything that might have been changed during the day. The results of the *make*, if unusual, are mailed to a list of people responsible for the directory in question.

A shell script for automatically generating makefile dependency lines, heavily adapted from the *make depend* entry in the 4BSD system makefile, is now being rewritten in C, with greatly expanded functionality. *make* has been, of course, an invaluable tool for system development.

*yacc, lex, and Program Generation*

*yacc* and *lex* were the obvious choice to write the parser for the data files. The *yacc* portion of the parser has about 170 rules and is over 1500 lines long. Writing it by hand would have been out of the question.

A much more interesting application, using *yacc* and *lex*, and of the same flavor as those programs, is one that reads data structure descriptions and produces subroutines designed to manipulate those data structures. The Alpha_1 applications that are written in C use an object oriented programming style. That is, all significant data structures are tagged with their type, and can therefore be dealt with in a generic way. For example, all objects subscribe to the storage management protocol. For any given object type, there is a function that creates a new object, that makes a copy of an object, including subobjects to which it may have pointers, and that frees an object of that type. All these functions are generated automatically. Of course, a `.h` file describing the data structures that make up an object type is also generated. Approximately 45% of the lines of code in the main Alpha_1 library are generated by a program.

*Interprocess Communication*

The Alpha_1 project started life on a PDP-11, running V6 UNIX. The only interprocess communication medium was the pipe, and it was sorely needed, to fit the necessary functions into a PDP-11 address space. Pipes are still useful for plumbing, to hook together various data filters instead of intermediate data files.

Using an interactive programming language that does not have a built-in editor can be a very frustrating experience. Whatever one types directly to the language processor is lost, but popping in and out of an editor to modify an input file and retry it defeats much of the purpose of the interactive language. On the other hand, if one is used to a particular editor, and the language processor has a built-in editor of a different flavor, the situation can be almost as bad.

The ideal solution, then, is to run the language processor as a subprocess of the editor. Then it is necessary to learn only one editor, and the language processor can be smaller, or its data and program space bigger, by the size of the editor it no longer needs to incorporate. With the advent of Gosling's *Emacs*, and its subprocess facility, this mode of operation became possible. An interface was written in MLisp that can feed an expression at a time to the *shape_edit* geometric editor. Thus, one can easily directly execute statements from any *shape_edit* program that has been read into an editor buffer. This automatically provides a history mechanism, remembering previous commands, and allows the user to incrementally build and test a model. A transcript of the entire editing session is kept in the output window of the *shape_edit* process.

---

2In fact, before the first VAX came, it was necessary to implement a crude virtual memory scheme.
Networking and Distributed Processing

Preliminary investigations have begun into the best ways to use a distributed facility in a design context. It seems clear that use of a server for computationally intensive processes will help to keep interactive response to a maximum on the local machine. A number of workstations are now available on which it should be feasible to implement some portion of the geometric editor; presumably much of the interaction and display generation could be done locally on the workstation. The computations necessary for many of the modeling operations would probably be done on a mainframe, however.

Research is also ongoing in the area of implementing some central algorithms in hardware, perhaps making a stand-alone workstation configuration possible. Experiments with distributed processing on traditional computers may indicate appropriate separation points between hardware and software components of the system.

Must it be UNIX?

Although development of Alpha_1 has taken place under UNIX, and the current implementation runs only under UNIX, it is not clear that UNIX is necessary to the Alpha_1 system.

It would be very difficult to move the development effort to another system, since many of the tools are strongly tied to UNIX. The experience of the Alpha_1 project is that UNIX is, indeed, a wonderful system for program development.

The programs that form the underpinning of the system have few or no UNIX dependencies. They are, except the geometric editor, written in the C language, which is now available for many computers, some running operating systems other than UNIX. The only system calls used by most of them are read and write, and whatever malloc calls.

The glue that ties the various programs together is more closely tied to UNIX. A few operating systems provide a facility similar to the UNIX pipe, but in most cases pipes can be replaced with intermediate files with little loss of functionality. More systems provide some capability that can emulate C-shell aliases. For example, DCL scripts could be used on VMS, or PCL procedures on TOPS-20. Failing this, a special purpose command processor could be written.

Interestingly enough, the component of the system that may be most portable is the geometric editor, shape_edit. Since it is written in PSL, and PSL has been ported to a number of machines, it moves fairly easily. Shape_edit is the only Alpha_1 program which has currently been ported to a non-UNIX environment on Apollo workstations.

With a few notable exceptions, many of the new computers being developed and sold run some variant of the UNIX operating system. The issue of porting the Alpha_1 system to another version of UNIX has received some attention. Recently, a series of benchmarks was completed, each of which consisted of bringing up a significant portion of Alpha_1 on the target machine and running some test cases through the code. Approximately 2 man-months work went into finding non-portable constructs in the C code, and another 2 man-months removing dependencies on the local UNIX environment. It was then possible to bring up part of the system and run significant benchmarks in one working day.

It is therefore conceivable that the Alpha_1 system, in some form, could be ported to a non-UNIX system. It would not be a small undertaking, though. More to the point, the system could easily have been developed for a different operating system, even if it could not easily have been developed on another operating system.
References


PRINCIPLES OF STRUCTURED FONT DESIGN
FOR THE PERSONAL WORKSTATION

by Charles Bigelow

*Portions of this paper previously appeared in Byte. Reprinted with permission from the January 1985 issue of BYTE magazine. Copyright © 1985 by McGraw-Hill, Inc., New York 10020. All rights reserved.*

The personal workstation is the new tool of the knowledge worker. The appeal of the personal workstation derives from the directness and immediacy of the relationship between person and machine. The visible expression of this relationship is a dynamic sequence of graphical images produced by an interactive dialogue between the computer and the user. Most of the images of this dialogue (the "user-interface") are composed of text, and the text is composed of letters. Our word "literacy" comes from the Latin word for letter. Today, our modern "computer-literacy" is much like traditional literacy - it is mostly reading and writing. Legibility of text is therefore a crucial factor in the usefulness of a personal workstation. Computer manufacturers, system designers, programmers, and users are taking a growing interest in the design of legible, high quality digital fonts.

Communication is the ultimate purpose of most computer systems. In our literate culture, this communication (whether person-to-person or person-to-computer) is achieved through text - written language. If the text written out by a system cannot be read, or can be read only with difficulty, the system is rendered ineffective, not by hardware or software problems, but by illegibility.

In discussing font designs and their composition into text, it is useful to define three levels of legibility. The highest level is usually called "readable", which means that the text is easy and visually pleasurable to read. The middle level is called "legible", which usually means that the text can be read without much difficulty, though the reading sensation may not be especially pleasant. The lowest level is often called "decipherable", which means that the text can be read only with difficulty and some degree of perceptual unpleasantness.

The most "readable" text font designs are transparent to the reader. Such fonts are designed to present the text in the clearest way possible, while remaining essentially invisible themselves. Thus, the best font designs are never noticed as such by the reader. Their qualities are transferred to the reading experience. The visual pleasure that comes from easily reading a text composed in a readable font is perceived as part of the pleasure of the text. The contrary is also true: the visual unpleasantness that comes from straining over a text in a barely decipherable font is perceived as a part of the difficulty of the text. This transference can be extended to the reading environment. A workstation with merely deci-
ipherable fonts may seem inferior to another system with effectively legible fonts, even though the former workstation may have some superior hardware and software features. Similarly, the difficulty of deciphering poorly designed fonts for extended periods may be manifested in other ways - as eye strain, headaches, or diffuse dissatisfaction with the work environment.

Until recently, there were few remedies to the problems caused by poor font designs for personal computers. The "character generator" technology which is still used to produce letterforms on most computer terminals usually provides only a single size of a single design style of coarse-resolution dot-matrix letters. Character generator technology usually permits no user modifications of the fonts, since the character images are contained in hardware or firmware. Similarly, the fonts provided on many dot-matrix printers have also been limited in size and style and low in resolution. Moreover, the designs of such fonts have too often been the work of people untrained in letterform design, while traditional lettering artists have rejected such systems because the tools and output media of digital typography have been so clumsy and crude in comparison to the responsive pens and brushes used in graceful calligraphy and precise lettering art. The result is that most existing dot-matrix fonts are not designed for optimum legibility within the limitations and constraints of the technology.

"Computer literacy" has therefore been a good deal less pleasant and less productive for the reader than traditional scribal and typographic literacy. The "hackerish" look of barely decipherable dot-matrix fonts on screens and printers has partially prevented full acceptance of computers as literate tools by a tradition minded literate public. Readers are conservative about the shapes of letters because literacy requires an expensive investment of time and effort, both on the part of the individual and on the part of society. Any change in the appearance of letters that makes them less legible wastes expensive human resources.

Today, however, the look of computer literacy is undergoing a major change which will alter the perceptions of the traditionalists. The newer raster-based technologies of bitmap display and non-impact printing offer potential solutions to many of the technical and aesthetic problems with digital fonts. We are beginning to see profound changes in the typographic appearance of documents produced by personal computer workstations. The new technologies also pose new problems, and the methods used to solve these new problems will determine whether the personal workstation will be a worthy successor to the traditional literate media.

**Imitation vs. Origination**

The newer raster-based font technologies can produce digital font images which more closely resemble traditional analog typefaces, such as the printing types used in the office world. This has begun an "imitative" phases of digital typography, in which manufacturers attempt to
copy traditional printing types and typewriter types directly in the digital medium. However, the analog shapes of traditional typography cannot be directly transported to digital media. Each technology has its own unique way of rendering letterforms; printing types originated for the molten lead casting techniques of Linotype mechanical composition cannot be faithfully and accurately reconstituted as pixel fonts on a digital raster display screen. Therefore, typographic traditionalists are disappointed by the degrading effects that digitization has on traditional typefaces. Text on display screens and on printer output is obviously inferior to text in a well printed book. Yet, users of workstations are dissatisfied with the common run of dot-matrix fonts, and desire something more readable for intensive use of computer systems.

Granted that we need legible and readable fonts on the new generation of computer workstations and printers, how can we produce them without being caught up in the intractable problem of imitating traditional analog fonts in a digital medium? The answer is to design digital fonts which adhere to the principles of readability found in traditional letterform designs, while tuning the features and details of the design to the digital medium. By focusing on solutions to the typographic problems of today and tomorrow, rather than vainly copying obsolete answers to yesterday's problems, we can initiate a "creative" phase of digital typography.

This pattern of imitation followed by origination is a well-known cycle in the history of literate technology. Every major change in the technique of rendering letterforms has begun with a relatively brief imitative phase in which obsolescent forms are imitated because of reader conservatism, followed by reader dissatisfaction with the poor quality of the imitations (due to the inherent limitations of the new technology), followed by the development of new creative methods for rendering original letters designs in the new technology. Once designers realize that the real problem is to optimize legibility within an imaging technology, a longer, richer, and more creative phase of letterform design begins. This is the phase that is just now beginning with digital typography.

Let us look at the principles of this approach to digital letterforms. First, it is helpful to understand the difference between a typeface and a font. A typeface is a group of characters (which may be alphabetic, numeric, or other signs and symbols) whose forms are shaped in accordance with a particular set of design principles and which share certain design features. A typeface design is thus an abstract work of art which attempts to create a certain kind of visual image in the mind of the reader. A font, on the other hand, is a concrete rendering of a typeface in a limited character set for a limited size-range for a particular imaging system. A font is thus a crafted product which implements a design in a specific device or system. Fonts are device dependent, whereas typefaces are technology dependent. The principles of legibility, however, are independent both of technology and of particular devices.
The basic letters of the Latin alphabet have had a long history in which many of the underlying forms have been remarkably well preserved since ancient times. Few of the shapes of artifacts in common use today are as unchanged as the shapes of letters. Our capital letters are based on imperial Roman capitals used for public inscriptions in the time of the Caesars, at the beginning of the Christian era. Our lower case alphabet is derived from a writing style called Carolingian minuscule (= "small letter") developed during the 8th and 9th century Frankish empire of Charlemagne. The elegantly readable handwriting of the Humanist scribes of the 15th century Italian Renaissance was based on a revival of the Roman capitals and Carolingian minuscule. The Humanist letterforms were in turn the models for the roman and italic printing types perfected in Italy and France during the 15th and 16th centuries. Those typefaces were the direct ancestors of most common printing types used today.

But, during the Renaissance transition from script to print, the early printing types were never successful copies of Humanist handwriting. A skilled scribe writing with an edged pen on vellum could produce a sharper, clearer, more lively and more rhythmic page of text than could an early printer confronted by problems of rough, hand-made papers, easily worn lead-alloy types, uncertain formulae for printing inks, and imprecise methods of printing. Types that attempted to imitate handwriting were inevitably of inferior quality. Some book collectors refused to allow printed books into their library, because printing was so inferior to professional handwriting. Faced with such criticism, it did not take long for type designers and printers to realize that types had to be designed and cut for optimum legibility in the printing medium. Imitation of handwriting was abandoned as unworkable and ultimately unfashionable, before the end of the 15th century. A new kind of letterform, based on the precise sculpting of refined contours rather than the real-time traces of a moving pen became the dominant look of literacy. Scribal letterforms, such as handwriting and calligraphy, are called "ductal", because their essential shapes are traced out as the path of the dynamic pen. (The sequence and pattern of pen (or brush) strokes is called "ductus" by lettering historians, from the Latin word "to lead".) Typographic letterforms, such as printing types, are called "glyptal", because their essential shapes are contours cut by an engraving process. (The term was first used by a Renaissance printer, from the Greek word "to carve".) Both ductal and glyptal letterforms are analog - produced by smoothly varying changes - though their technology and resultant forms are distinctly different.

Digital letterforms, such as raster-based screen and printer fonts, can be called "pictal", because their essential shapes are composed of patterns of discrete "pixels" (picture elements). (The term comes from the Latin word "to paint, to tattoo".) In the lower resolutions, pictal letters appear to have closer affinities with mosaics, tile patterns, and pointillist paintings than with handwriting or traditional letterpress printing. The
specifically pictal details of digital letters tend to obscure their basic functional qualities, and therefore it is important to look beyond the surface features to the deeper structure of the letterforms in order to design readable letters for digital rasters. Since the raster is a virtual design medium without tactile or material characteristics, the designer of pictal letters cannot rely upon traditional tools such as pen, brush, or graver to help guide the resulting shapes. The pictal letter requires a rational analysis and orderly methodology for the resulting text image to be useful and readable. One well-understood solution to a related set of problems in software development is called "structured programming", which emphasizes logical problem analysis, systematic construction, and conceptual structuring of design solutions. These same principles can be effectively applied to the problems of digital letterform design. We call these techniques "structured font design."

Letterforms have a perceptual structure as well as a conceptual structure; attempts to design fonts by logic and mathematics alone have been failures. Text must be perceived to be read, and therefore effective digital font design must include study of the nature of visual perception and the mechanisms of the human visual system. The traditional lettering artist learned principles of visual perception as one part of a long apprenticeship that emphasized practical tool use and intuition guided by practice. Our task today is to rationalize much of this traditional knowledge so that the digital computer tools developed for the design and reproduction of pictal letters will produce the same degree of readability as traditional analog technology and craftsmanship.

It is easiest to demonstrate the principles of structured font design by showing actual examples. We will first demonstrate the issues involved in font design for bitmap screen displays, and then the related issues for medium resolution laser printers.

**Screen Fonts**

The design of screen fonts is difficult because so little information is available for each letterform. The resolutions of common bitmap CRT displays range from 60 to 100 lines per inch, with approximately 72 lines per inch being average. This is approximately one-tenth the resolution of average quality digital typesetters used in the graphic arts and publishing industry. This much resolution is necessary for adequate rendering of analog typefaces because of the sensitivity of the human visual system. It is a truism that communication of information is most effective and most economical when the characteristics of the transmitter are matched to the characteristics of the receiver. In literate communication, the transmitter is the system that produces the visual text image, and the receiver is the human visual system. Thus, the digital text image must contain as much resolution as the eye and brain are capable of receiving and interpreting, but need not contain more information than that.

There is a way of estimating the theoretical minimum resolution for
good quality digital text. Experiments by psychophysicists and perceptual psychologists suggest that the visual system cannot detect spatial frequencies greater than 60 cycles per degree of visual angle. That is, a bar grating of regularly spaced black and white lines will be perceived as a solid gray, rather than a grating pattern, if the spacing is so fine that more than 60 black and white line pairs are imaged in one degree of visual angle, as measured at the retina. This provides a measure of the upper limit of the visual system’s ability to resolve the kind of detail produced by a digital raster. At a reading distance of approximately 12 inches, 60 cycles per degree of visual angle is equivalent to a resolution of 300 cycles per inch at the screen.

We can now estimate the minimum resolution necessary for good quality digital text by using a well-known principle of digital signal processing theory developed by Harry Nyquist at Bell Laboratories in the 1920’s. It states that a signal can be digitally sampled and reconstructed without loss or distortion if the sampling rate is at least twice the rate of the highest frequency in the original signal. This minimum sampling rate is today known as the Nyquist limit, or the Nyquist rate. Sampling below the Nyquist limit introduces “aliasing,” in which the high frequency components of the original signal are erroneously reproduced as spurious lower frequency components of the reconstructed signal. In digital typography, one form of these aliases is the well-known “jaggies”—the jagged stair-step patterns that fringe the edges of digital type.

To faithfully sample and reconstruct a signal of 300 cycles per inch, a minimum sampling rate of 600 lines per inch is necessary. In fact, resolutions in the range from 600 to 720 lines per inch were used for many of the common digital typesetting devices developed during the late 1960’s and the 1970’s. A decade of practical typographic experience with these machines showed that this resolution range was adequate for low and medium quality printing such as newspapers, telephone books, and magazines, but was not acceptable for the highest quality typesetting and printing, which required digital resolutions of 1200 or more lines per inch for optimal rendering of traditional analog typefaces.

The Nyquist limit is only a theoretical minimum, and the difficulty of quantifying the mechanisms of visual perception means that for high quality letter images, real-world sampling rates often have to be higher than the theoretical minimum. The practical evidence suggests that today’s screen resolutions of 72 lines per inch are at least one and possibly two decimal orders of magnitude too low to produce text of optimum visual quality. The lesson here is that traditional analog typefaces cannot be imitated and jaggies cannot be eliminated on display screens. The only practical solution that can produce functional legibility on display screens is to design the screen fonts within the limitations of the available raster system, and to optimize the features of the font to the mechanisms of the human visual system and ensure their conformation to the familiar historical principles of letter design.
Size
Type size is an important factor in legibility. Up to about 18 point (a printer’s point being approximately 1/72 of an inch), the larger sizes are easier to read than the smaller, but a need for economy tends to reduce the type size used in many applications. The most commonly used type sizes are a compromise between legibility and economy. Text for books, newspapers, magazines, and other documents is generally composed in type sizes ranging from 9 to 12 point. The most common text type size is 10 point, and this is also the approximate body size of a 10 pitch typewriter font.

The screens of computer workstations are generally viewed at a somewhat greater distance than books and printed documents. A reader can easily adjust a hand-held book to the ideal reading distance, but a computer terminal is more like a piece of furniture or a heavy appliance that requires a certain amount of surrounding space on a desk or work surface. Additional paraphernalia such as a keyboard, mouse, or graphics tablet also tend to intervene between reader and screen. The ideal screen viewing distance depends on the legibility of the main text font and the characteristics of the display. Some ergonomic guidelines recommend a viewing distance range of 16 to 28 inches; other guidelines, a range of 13 to 20 inches. As a rough estimate, screen fonts should be about 1.2 to 2 times as large as printed text fonts.

This measure must be corrected for the fact that the apparent size of text in English and other languages that use the Latin alphabet is dependent more on the size of the lower-case letters than on the body size of the whole font. Lower-case is measured by its “x-height,” which is the vertical distance from the baseline, on which all the letters appear to sit or stand, to the top of the lower-case ‘x’. The body size of a font is the vertical distance from the bottom of a “descender,” such as the descending stem of a ‘p,’ to the top of an “ascender,” such as the ascending stem of a “b.” The x-heights of common text types range from about 40% to 60% of the type’s body size. A type with an x-height of 50% of the body is considered to be a face of medium-large appearance. A popular 10 point book face might have an x-height of about 5 points. If we assume a display screen with resolution of 72 lines per inch (one pixel per printer’s point), then a screen font should have an x-height of 7 to 10 pixels, to adjust for the greater average reading distance.

Weight
The weight of a typeface is its relative density, or proportion of black image to white background. Weight can be measured as the ratio between the thickness of a straight vertical stem (such as the stem of an ‘l’) and the x-height. The greater the stem thickness in proportion to the x-height, the heavier the weight, and the darker the text image appears. Conversely, the smaller the stem thickness in proportion to the x-height, the lighter the face appears. For text types, the optimum weight ratio ranges from 5
to 6 stems per x-height. Weight ratios lower than 5:1 generally make the face appear too dark for easy reading, and ratios greater than 6:1 make the face appear too light. This presents a difficult and sometimes insoluble problem for the screen font designer. For example, given an x-height of 7 pixels, a stem weight of one pixel will be too light, but a stem weight of two pixels will be too heavy. The digital raster cannot permit non-integer stem weights, and thus an optimum ratio seems unachievable.

However, on a CRT display the perceived stem thickness is almost always different than the nominal thickness computed from the specified raster resolution. Physical factors which influence perceived stem weight include: the size and intensity contour of the writing spot, the amount of spot overlap, the speed with which the writing beam turns from on-to-off vs. the speed from off-to-on, the characteristics of the phosphor, and the brightness and contrast of the display. When the letterforms are black and the screen background is white, these factors usually combine to erode away a significant portion of the perceived stem weight. If this erosion is around 20% total, not an unusual amount, the perceived weight ratio of a font with x-height of 7 pixels and stems of 2 pixels would be approximately 4.4:1. While rather dark, this would be preferable to a one pixel stem, which would produce a weight ratio of 8.7:1 - far too light and spindly. A larger x-height of 9 pixels with a stem of 2 pixels would, under the same conditions, yield a perceived weight ratio of 5.6:1, within the optimum range. Thus, there is an interaction between font size, as measured by x-height, and stem thickness which makes some size/stem combinations significantly more legible than others. The matrix of all possible low-resolution digital fonts must be filtered to pass only those of acceptable weight ratios.

Contrast

In traditional text typefaces and lettering based on the Latin alphabet, vertical letter elements are almost always thicker than horizontal elements. The stems of an 'n' are thicker than the serifs or the connecting arch; the vertical bowls of of an 'o' are thicker than the horizontal hairlines. This noticeable difference between vertical and horizontal features is called contrast. Faces with high contrast have a brilliant, glittery look, and faces with low contrast have a stolid, monotonous look, but all Latin-based text letterforms have some contrast. Non-Latin scripts, such as Hebrew, Arabic, and Devanagari (used for several languages of India) also have contrast, but the horizontal elements are thicker than the vertical. Originally a result of the way the scribal writing tool was held and manipulated, contrast was preserved in typefaces because it aids recognition and discrimination of letterforms. For screen fonts to have some of the legibility of traditional typefaces, the traditional contrast must be preserved. Fonts in which the vertical and horizontal elements are the same thickness have an unfamiliar texture; this unfamiliarity impairs legibility. When both horizontals and verticals are only one pixel in thickness of a
CRT display, and the letters are black on an illuminated background, the problem is exacerbated by the erosion of vertical stems which become even thinner than horizontals, contrary to all visual expectations of the reader. Such fonts not only appear weak and spindly, they seem unclear and ill-defined, as though the reader's vision were blurred or something were misadjusted on the display screen. What is actually blurred and misadjusted is the design of the font. However, thicker stems require a larger x-height to maintain the proper font weight, so there is a lower limit to the size at which contrast can be implemented on a screen font.

*Spatial Frequency and Letter Fitting*

We have discussed visual spatial frequency as a means of estimating the limits of sensitivity of the visual system, but the lower spatial frequencies in text are of even more importance. Many psycho-physical experiments suggest that the human visual system is most sensitive to spatial frequencies in the range from 2 to 6 cycles per degree of visual angle. A line of text is composed of multiple spatial frequencies, of which the fundamental is the regular alternation of black vertical stems with intervening white counters (the space inside a letter like ‘n’ or ‘o’) or inter-letter spaces. Estimates of the fundamental spatial frequencies of printing types at text sizes show a range from 4 to 6 cycles of degree of visual angle – within the range of peak sensitivity of the visual system. When large text sizes, such as those used in luxury books where typographic economy is not a factor, are included in the estimates, the range expands to include 2 and 3 cycles per degree, the remaining area of peak sensitivity. The sizes and spacings of type are not arbitrary; they have been carefully tuned to the mechanisms of the visual system, not by rational analysis, but by centuries, even millennia, of careful experimentation. Screen fonts should also be tuned to this band of fundamental frequencies. Tuning does not mean just being within the range of peak sensitivity, it means establishing and maintaining a regular frequency for the text image as well.

Throughout the ages, scribes and type designers have painstakingly adjusted the spacing and fitting of letters to maintain a rhythmic and harmonious visual pattern in the line of text. This is equivalent to maintaining a regular fundamental frequency in the text image. Intentional interruptions in the basic frequency, such as those caused by word spaces, are therefore noticeable and significant. Accidental irregularities in the basic spatial frequency, such as dark tangles or light voids caused by poor letter spacing, also attract attention because they resemble significant patterns, but they impart no information. Irregular spacing is therefore a kind of noise that distracts the reader, interrupts the smooth flow of reading, and obscures the real textural information, thus impairing the legibility of the text.

A failing common to many screen fonts is letter spacing that is too tight in some combination, too loose in others, and generally irregular. This is a result of conceiving of a font as a collection of individual letters.
rather than as an organized system of figure and ground. The negative space of the background must be designed simultaneously with the positive shapes of the letters, and in many cases the designs of individual letters are shaped by the need to fine-tune the spacing of the entire font. Attempts to achieve tight inter-letter spacing prevent good overall spacing, because they create disparities between letters that can space closely, such as combinations like 'll', and those that must space widely, like 'vy'. Tight and irregular letter spacing is currently faddish in advertising typography, where it serves to attract attention to sales pitches that the reader would otherwise ignore, but analysis of several centuries of typographic texts demonstrates that open, rhythmic spacing is most readable for texts intended to inform and edify.

**Proportion**

Because the alphabet is a system, the proportions of the letters must be tuned to each other and to the overall proportions of the alphabet design. The widths of the letters must conform to three main criteria: the x-height of the alphabet design; the optimum spatial frequency of the text; and the historically evolved letter shapes. The average width of the letters in relation to the size of the font determines the fundamental spatial frequency of the font at a particular reading distance. This frequency should be within a certain range, as discussed above. Moreover, the different widths of the letters in relation to each other help the reader to discriminate their forms, as well as permitting a rhythmic spacing pattern. Proportionally spaced fonts are generally more legible than monospaced fonts because of the more finely tuned pattern of the text. When monospaced fonts are a necessity, great care must be taken to compensate for the irregular rhythm and distorted proportions, but such compensation is possible only up to a point. The limitations of mechanical typewriter technology created a need for monospaced fonts, but as these limitations are not necessary in digital typography, the less legible monospaced fonts can now be retired from most applications.

**Differentiation**

The alphabet is a semiotic system of graphic signs which refer to the phonetic elements of speech. In speech, these phonetic elements, sometimes called phonemes, are carefully differentiated from each other; therefore the letters of the alphabet must be similarly differentiated. A legible font has letterforms which are easily discriminated one from the other. The task of font design is then to ensure that the letterforms are unambiguous, discriminable, and distinguishable. That is, that they can be recognized easily, and that one can be told apart from another. Although resolution is limited for screen fonts, these goals can be achieved by concentrating on three areas: serifs; primitive elements; non-assimilation and asymmetry.

**Serifs.** Serifs act as 'flags' on the character shapes to aid in discrimination.
Note that in a sans-serif (= "without serifs") font, an 'r' followed by 'n' can easily be confused with an 'm', whereas the same combination in a serifed font is less easily confused with 'm'. Similar demonstrations can be made for other combinations. Therefore, while sans-serif fonts may seem more modern (and they are not all that modern, having been first designed in 1816), they are less legible for text because they lack these small but significant distinguishing elements.

**Primitive Elements.** Latin-based alphabetic characters are like molecules constructed from simpler atoms. These primitive atomic elements are called "strokes" because they were originally a single motion of a pen or brush. The various kinds of strokes include: verticals, horizontals, curves, and diagonals. The alphabet can be sub-divided into various groups of letters made up of particular primitive elements. For example, in the lower case, the letters 'n', 'm', 'h', 'r' form one group based on the vertical straight stem and arch; 'o', 'c', 'e' form a group based on the curved bowl; 'b', 'd', 'p', 'q', form a related group based on the curve plus straight; and 'v', 'w', 'x', 'y' form a group based on the diagonal stroke. These groups help the reader to discriminate and distinguish the letterforms. Faced with the problem of the jaggies, which are worst on curved and diagonal strokes, some screen font designers have succumbed to the temptation to reduce the letter shapes to straight vertical and horizontal elements. While this technique reduces the effect of the jaggies, it also destroys the legibility of the font by eliminating two of the three basic primitive elements and collapsing the form groups together. When every letter in the alphabet resembles every other letter, the basic principle of discrimination is lost and the alphabet degrades to decipherability at best. While the jaggies are indeed a problem, it is still preferable to maintain the traditional shape primitives and keep the letterforms unambiguous, even if the diagonals and curves show jaggies.

**Assimilation and asymmetry.** The principle of construction from primitive elements must not be applied in a simple-minded way. Traditional letterforms, while related by constructive principles, are neither overly assimilated nor overly symmetrized. In particular, it can be demonstrated that the upper portions of the lower-case characters are the most carefully differentiated parts of the alphabet design. It is as though the gaze of the reader focuses more on the "x-line" (the top of the lower-case) than on the base line. When the forms of the lower-case are strongly assimilated toward one basic shape, which has both vertical and horizontal mirror symmetry, the individual characters lose their identity. This can easily be seen in the set that includes 'a', 'b', 'd', 'g', 'p', and 'q'. The 'b', 'd', 'p', and 'q' should share many features, but they should not be strict mirror and rotational images of one another. Care must also be taken to prevent the 'a' and 'g' from being too closely assimilated to the others. Similar principles should be observed throughout the rest of the font design.

The foregoing principles have concentrated on designing screen fonts for optimum readability. Such optimization rests on the fundamental assumption that the screen is the place where the text will be read. However, text is also read as printer output on paper. The relation between screen text and printer text is the subject of intensive research, with many recent efforts attempting to integrate screen and printer in what are called "What You See Is What You Get" (WYSIWYG) editing and layout systems. The WYSIWYG principle is that the screen should show exactly how the printed document will look. WYSIWYG text editors and document formatters usually attempt to show different typeface styles in different sizes, spacings, and page organizations. The usual model for WYSIWYG systems is traditional typography, which offers so vast and complex a range of possibilities that present WYSIWYG systems can usually only offer a much reduced subset. In fact, it is easy to see that a restricted subset of typography is all that present screen technology can provide, since 72 lines per inch screens have only one-fourth the resolution of 300 line per inch printers, and one-tenth the resolution of 720 line per inch typesetters. The Nyquist limit on sampling rate shows that it is inevitable that typographic information will be lost or distorted in the lower resolutions. True WYSIWYG systems are impossible to achieve, and while the principle has been beneficial in focusing attention on typographic legibility and the structuring of documents, it can lead to serious error when strictly applied. Usual WYSIWYG implementations fall into one of two categories: "bottom-up" or "top-down".

Bottom-up WYSIWYG systems start with the screen resolutions and force the printer to conform to the limitations of the screen. In the simplest case, each screen pixel is mapped one-to-one onto the page of paper output by the printer. While this provides a certain cartesian satisfaction, since it can be logically demonstrated that the printer page is "exactly" like the screen display, the two images will actually appear very different. As we have discussed above, the screen characters are eroded by the characteristics of the display technology, whereas the printed characters are either emboldened, as by "ribbon-spread" on a dot-matrix printer or by toner effects on a "black-writing" laser printer, or at least not eroded to the same degree as the screen fonts, as by a "white-writing" laser printer. Thus, if a font is tuned to the optimum weight and contrast on the screen, it will appear too dark and too low in contrast on the printer output. Conversely, if the fonts are tuned to the printer, they will appear too light and too high in contrast on the screen. This is unavoidable. What you see is not what you will get, at the present level of display and printer technology.

A second problem with "bottom-up" WYSIWYG is exaggeration of jaggies on the printer output. Aliasing on the screen is somewhat ameliorated by the soft intensity contour of the CRT writing spot. The spot does not have sharp edges, nor is it square or rectangular; instead it is
blurry and round. The low-contrast edges of the pixels tend to soften the apparent jaggies. Printers, however, produce a high-contrast spot which clearly renders the edges of the jaggies. The jags become even apparent to the reader, since the human visual system tends to enhance edges. On a laser printer which has several times the resolution of the screen, several printer pixels will be used to render a single screen pixel. This emphasizes the rectangularity of the raster, and further enhances the jagginess of the digital artifacts. Printer fonts that are constrained to simulate screen resolutions look noticeably inferior to printer fonts that are optimized to the full resolution and imaging characteristics of the printer.

Top-down WYSIWYG systems store fonts as high resolution master images. These are usually outlines that can be scan-converted to raster images to represent arbitrary sizes at arbitrary resolutions on screens, printers, or typesetters. This "device-independent" method is intellectually appealing, since in some sense the "same" design is used to produce all actual raster "glyphs" at the writing resolution of each target device. However, the Nyquist sampling limit again prevents low-resolution and high-resolution fonts from being truly the same. In top-down systems, the fonts on low-resolution devices become the inferior ones, both in comparison to high-resolution versions, and in comparison to optimized low-resolution designs. The current generation of master-image data structures and associated scan-conversion algorithms can do good automatic rasterization at bitmap resolutions around 1200 lines per inch (equivalent to 200 x 200 pixels per em-square at 12 point size), and an acceptable job at 600 lines per inch (100 x 100 pixels per em), but only a mediocre to inadequate job at 300 lines per inch (50 x 50 pixels per em), an incompetent job at 150 lines per inch (25 pixels per em), and hopelessly botched hash at 75 lines per inch (12 x 12 pixels per em).

Since the current systems are the work of the best mathematicians working in the field of digital typography, it is unlikely that we will see major improvements in the near future. Letterform scan-conversion is a problem that is as much perceptual as mathematical. Letterforms must look right to the reader; their successful design requires knowledge of historical forms as well as understanding of the mechanisms of perception. At the present time, much of this understanding is intuitively perceived by artists, but not successfully analyzed by scientists. As Blaise Pascal, a great mathematician himself, wrote in his Pensees, "The reason why mathematicians are not intuitive is that they cannot see what is in front of them."

WYSIWYG spacing. Builders of WYSIWYG systems tend to conceive of the coordination of screen fonts with printer fonts as a numerical problem in matching spacing values, rather than perceiving it as parallel problems in optimizing legibility. Indeed, matching the letter spacing and fitting of a 72 line per inch screen font with a 300 line per inch printer font such that a given text will have the same words on each line, the same
line break and hyphenation, and occupy the same relative space on both screen and printer page is a difficult problem. It is, however, not the only problem. It is just one aspect of the general sampling and quantization problem of low-resolution fonts.

The "real" typeface is neither the screen font nor the printer font, nor even a typesetter font, nor even an artist's drawings. The "real" type is, as Parisian type designer Adrian Frutiger has said, "the image in the mind of the reader". Solving the numerical problems of matching letter spacing is not nearly enough. The text must also be readable. The spacing rhythms of the text, crucial for legibility, must not be tortured on a procrustean bed - stretched and truncated to fit an arbitrary numerical measure. Instead, both low and high resolution fonts must be developed in parallel - matched in spacing but optimized in legibility. This can be done most effectively when the typefaces are original designs, crafted for the digital media.

These critical evaluations assume that readability, the highest form of legibility, is the desired goal of digital font design. If a lower grade of legibility, say decipherability, is all that is wanted, then top-down automatic rasterization may be more adequate at the lower resolutions. However, we must not forget that fonts are for reading. Inferior fonts degrade the entire information system at the crucial human interface. There is nothing to be saved by wasting expensive hardware, software, and human time by attempting to make do with inferior, semi-legible fonts.

There is increasing suspicion that the automated office has not provided the increases in productivity promised by system vendors. One reason is that the fonts on such systems have not been as legible as the traditional typefaces familiar to the literate office worker. When a vendor claims that the fonts on a system are "pretty good", or "close enough", or "almost correspondence quality" or some other related euphemism, it should be apparent that this is the same as saying that the fonts are less than optimum, and that the reader has been short-changed on legibility. Considering the vast amount of time that is spent in reading digital fonts, the huge investment in literate education for those readers, and the immense expense for their salaries and wages, anything less than the highest possible font quality is tremendously counter-productive.

*Grayscaling.*

One technical response to the problem of low-resolution screen fonts is to increase the display information from one bit per pixel (the black & white bitmap display of current workstations) to several bits per pixel (the grayscaled display of some experimental and color workstations). Grayscaled fonts, because they contain more information, can better depict traditional letterforms, at least when viewed in isolated words and phrases. Also, the low-contrast edges of curves and diagonals reduce the visual effect of the jaggies. The letterforms appear smoother. Some re-
searchers have therefore suggested that grayscale text would be more readable than bitmap text. This is, so far, only an hypothesis. Our alphabet has evolved as a system of high-contrast edges. It is not yet certain whether the conservative eyes of readers will accept grayscale text, nor whether grayscale text is more difficult to read despite its less jagged appearance.

Grayscale fonts are also more expensive to display and more difficult to design. They require more bits of memory to store the gray value at each pixel, and more elaborate and stable CRT's and controller electronics. The shapes of grayscale letterforms are inherently more dependent on precise control of brightness and contrast on the CRT monitor.

The design of such characters is still problematic because as yet we have no common understanding of what kinds of digital filters will optimize legibility when creating grayscale low-resolution fonts from high-resolution master images, as in top-down grayscale systems. From the bottom-up direction, pixel-by-pixel construction by lettering artists would aid in the perceptual problem, since the artist could judge the designs by eye on the screen, but there are no effective pixel-editors for grayscale fonts as yet. We do not yet have a clear concept of how such an editor should function for an artist, since the choice of gray value for each pixel is influenced by the values of the neighboring pixels. Proper letter spacing also remains a serious problem with grayscale fonts. For economy, most systems will store each letter in only one grayscale version for a given font size, though many versions would be possible. Choosing which of the possible versions of a letter will best combine with which other possible versions of the other letters is an enormous undertaking.

Yet, if multiple versions are stored or automatically produced from high-level masters, there will be serious computational and memory burdens on the system. All in all, further research on grayscale fonts is required before these questions can be resolved.

If the goal of digital font design is to optimize legibility, then a combination of top-down and bottom-up design is necessary. At the lower resolutions of screens and dot-matrix printers, a skilled designer must construct the designs pixel by pixel. Our current technology offers no adequate substitute for the experienced eye and trained intuition of the lettering artist. At the medium resolutions, careful hand-tuning and editing of even the best automatically scanned fonts will provide significant improvement. Bit editing of digital fonts is regarded as a tedious process by artists and engineers alike. The need for bit editing results from a failure of our present technology to provide sufficient resolution such that the current crop of scan-conversion algorithms will function acceptably, or to provide scan-conversion algorithms that embody a more sensitive understanding of the structure of letterforms and the mechanisms of the human visual system. Although work continues on devising better conversion algorithms and data structures, more effort should also be spent on creating better bitmap editing systems that reduce tedium and im-
prove productivity for those situations where fonts must be edited for optimum legibility.

A compromise between the top-down and bottom-up methods means that a system can be optimized for legibility by allowing the storage and processing of both bitmap and outline font data. This kind of hybrid system need not be much more complicated or expensive. Designer-tuned bitmap fonts are need most at the lower resolutions and smaller sizes, which are relatively economical to store and process because the bitmaps are small. The larger sizes and higher resolutions can be stored as economical outlines and automatically processed by algorithm, a method which is effective at higher resolutions. In order to achieve maximum efficiency and economy, such a system must be designed with a model of typographic structure that optimizes the most frequency and important text usage. Not only should font designs be structured to optimize legibility, but typographic layouts should be structured to convey information content in the most effective way.

The Visual Editing of Text

Until recently, most authors were satisfied with the typewriter, which usually offers only a single size of a single style of type. More machines today offer interchangeable type styles, but style and size are seldom changed within a document. As laser-printers able to handle a variety of fonts become available, authors are becoming aware of greater typographic possibilities. Yet, authors are limited in their ability to express themselves typographically, because of lack of experience with the typographic medium. Experience with the traditional tools of typographic composition has until recently been the exclusive province of professional typographers, graphic designers, and printers.

A professional level of typographic expertise requires several years of training and practice. For most authors, the acquisition of such a high level of expertise would be an impractical distraction. Nevertheless, it is useful for the author to have awareness of and facility with the basic concepts of text organization. Fernand Baudin, a noted European typographic authority, calls this basic kind of typography “the visual editing of the text”. He rightly observes that every modern author should have at least some acquaintance with the principles and elements of typographic layout, in orde to make the written and printed document a more effective medium of communication.

While discussion of the principles of typographic layout would require a separate article, certain basic aspects are relevant to the design of digital fonts for workstations and printers.

Size

Experienced typographic designers and theorists have asserted that no more than 3 sizes of type are necessary for most page layouts, and that 5 sizes of type can handle almost any complete typographic document. A
basic text size, almost always in the range from 9 to 12 point, will compose 75% to 95% of the total text of most documents. A larger size is usually needed for "display", such as titles, headlines, and related functions, and a smaller "reference" size is often needed for footnotes, indexes, and similar material.

Study of how meaning is typographically structured in a text shows that absolute size is less important than relative size. The basic text size should be comfortably readable, while the display size should be noticeably larger and the reference size noticeably smaller than the basic text. In practice, a "noticeable" size difference is, on the average, a factor of about 1.5:1. Since type is two-dimensional, a linear increase by the square-root of two will double the actual area, and the intuitive, visual determinations of type size by designers roughly follow a square-root-of-two progression. For example, if the basic text size is ten point, many designers will choose a display size in the range from 14 to 16 point, and a reference size in the range from 7 to 8 point.

Thus, it is much more useful for a workstation and a printer to provide a functionally integrated set of three to five readable font sizes than it is for them to provide thirty to fifty fonts that have to be deciphered. Most workstations do not have to provide the professional typographic capabilities of industrial graphic arts equipment, but they do have to emit useful, readable, and comprehensible documents.

Style
By the "style" of a typeface, we do not mean fashion, but rather the thematic design principle of the letterforms, such as "roman" vs. "italic", "normal" vs. "bold" weight, and "serifed" vs. "sans-serif". Different alphabet styles can be integrated into a single functional typographic system. As with sizes, typographic authorities advise that no more than three styles of type are needed on an average page, and that no more than five styles of type are necessary in the average document. In fact, it is also generally believed and taught by professional typographers that too many type styles will confuse and obscure the information content of a document rather than enhance it.

A fundamental structured system of typeface styles has evolved throughout the past five centuries. The system is intuitively recognized and understood by most readers, even though it is seldom explained in schools. At one time, the capital alphabet was a style distinct from the lower-case, but both were amalgamated into one "duplex" alphabet by the Italian Humanist scribes during the 15th century. This richer alphabet allowed greater graphic nuances of expression. Roman and italic, distinct and competing styles in the 15th century, were mated into a single family by the 16th century French and Flemish printers who developed more complex book layouts. Bold face, an early 19th century English innovation, was adopted into the roman and italic family by the end of that century. Typography in the 20th century has begun to unite sans-serif with
serified type families to form "super-families" of designs for the complex
documents of the computer age.

It is remarkable that even though there are over a thousand typefaces
used to some degree in the printing industry, the basic level of functional,
informative typography is founded on a small, structured set of elementary
distinctions: capitals vs. lower-case; roman vs. italic; normal vs. bold;
serified vs. sans-serif. Since capitals and lower-case are routinely united
in a single font, only three dimensions of font variation are necessary
for most text purposes. These different styles are used to construct the
system of differences that graphically evoke the structure of a text. The
roman serified face is generally the basic text face. Difference from basic
text is graphically marked by italic. Difference plus emphasis is marked
by bold face. Difference from bold emphasis is marked by bold italic.
These distinctions form a group of four elements. Sans-serif creates a
second, related group of distinctions.

Informative texts, such as books, manuals, and other documents
make extensive use of these distinctions and relatively few others. Of
course, for specialized and technical applications, fonts of non-latin alphabets such as Greek, or non-alphabetical characters such as mathematical
signs and symbols, may be necessary. Also, monospaced variants of
certain fonts may be needed for software and hardware that cannot han-
dle the more legible but more complex proportional spacing. Even so,
the design of such fonts may be partly of fully integrated into the basic
structured set of typographic variations.

Harmonization

Typefaces work most effectively together when they are harmonized in
a family that shares common features and design principles. For exam-
ple, it is important that all the faces of a family appear to have the same
alignment of x-height, capital height, and ascender and descender length.
Also, the normal weight for roman, italic, and sans-serif should appear
to be the same for all alphabets of the family. Similarly, the bold ver-
sions should be adjusted to a single perceptual level of blackness. This
kind of harmonization emphasizes the significant differences between
the styles and enhances their meaningfulness as graphic signals. By elimi-
nating unpredictable, arbitrary, and insignificant typographic variations,
random graphic noise can be removed from the text image. Harmonized
typography communicates the message of the text more clearly and ef-
effectively. This kind of harmonization is standard for traditional, serified
typface families, but it has not been effectively applied to font design
for screens and printers, nor is it commonly achieved for combinations
of serified and sans-serif typefaces.

Digital typography should provide systematic methods of clearly or-
ganizing and graphically presenting informative texts. Variations in type
size, style, and spacing are most effective if they are used to clarify the
structure and meaning of the text. Typography is not an ornament to
make text attractive, it is the essential medium through which the text is communicated. Therefore, typography should be transparent to the reader. Typography is at its best when the reader is conscious neither of seeing the letters nor of reading the words, but only of understanding the author's meaning.

Conclusion
The personal workstation offers powerful tools to the knowledge worker, but these tools are dependent upon typography: legible fonts in effective arrangements. Digital technology is presently limited in its ability to reproduce analog letterforms. Traditional typefaces cannot be successfully reproduced at current display screen and printer resolutions. To optimize legibility, new fonts must be designed for the digital media. These fonts will be most effective if they take into account the nature of the human visual system, the logical and historical principles that shaped our present day alphabets, the characteristics of current digital imaging devices, and the conceptual structures underlying typographic variations and arrangements. The new technology requires a new typography, a typography that preserves the fundamental features of literacy, but expresses them with new clarity in a new medium.
Use of a logical variable with a non-numeric value

The logical variable should be assigned a numeric value before

It should be noted that the logical variable is used in a conditional

expression, and therefore requires a numeric value for evaluation.