Simple Ideas That Changed Printing and Publishing

Article *in* Proceedings of the American Philosophical Society · December 2012

CITATIONS
3

READS
70

1 author:

John Warnock
Adobe Inc.
18 PUBLICATIONS **401** CITATIONS

SEE PROFILE

Three Ideas that Changed Printing and Publishing

John E. Warnock

Traditional, paper-based printing and publishing evolved into its current state over 5.5 centuries (depending on where you start); transforming page composition into a wholly computer-based process for printing and publishing took a mere 30 years. Several technological advances occurring somewhat concurrently allowed this transformation to take hold and become globally pervasive. I would like to explain these technological innovations and the historical environment that made modern publishing possible.

This is a timely discussion because of the uncertain future of print-based enterprises. The businesses around newspapers, books, and magazines are changing on a daily basis; even still, global electronic communication over the Internet is systematically replacing print media as staples of the written word. This has undoubtably been influenced by the fact that the final form of most print publications is now produced on the computer.

WHAT HAPPENED AND WHERE

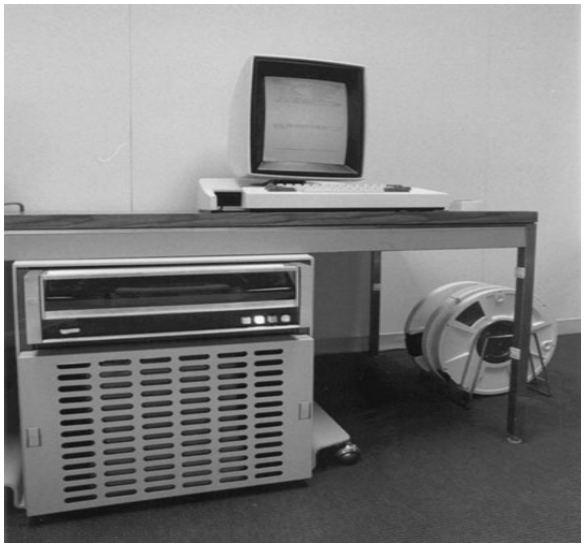
Although typesetter manufacturers took some steps toward electronic publishing, it is safe to say that much of the credit leading to the technical innovations around publishing began at the Xerox Palo Alto Research Center (PARC) in California. The personal computer with a graphical user interface was born at PARC, the Ethernet was invented there, and black-and-white and color laser printers were developed at PARC. These key components led to the computerization of publishing.

When I started to work at PARC in 1978, an enormous amount of groundwork had already been laid. We each had our own personal computer (the Alto) with a black-and-white page-sized raster display, a removable disk drive, and a keyboard and mouse. All our computers were connected via the Ethernet, which linked them to laser raster printers. All of these components were developed at PARC.

The predecessor of modern word processors (Bravo) was developed in this work environment by Charles Simonyi (among others), who later joined Microsoft and developed Microsoft Word. We also used an e-mail system (Laurel) that was remarkably like the e-mail we use today.

Sites around the country, including PARC, were connected via the ARPANET (which later became the Internet). When it was founded in 1972, PARC management's mandate was to invent the "office of the future." For all practical purposes, the researchers at PARC accomplished their task.

In 1978 the offices at PARC had all of the basic components of a typical workplace today—all in place before any commercial personal computers or networks were available. It is, however, important to note that computers were vastly less powerful. Speeds of the processors were hundreds of times slower, computer memories were tens of thousands times smaller, and external storage was hundreds of thousands of times smaller.



Alto computer at PARC, 1973.

Xerox was very secretive about the technologies at PARC and few people knew what existed there. Over time the ideas at PARC escaped and were exploited and developed to become the computing environment we use today. Steve Jobs borrowed the ideas in the graphic user interface for the Lisa computer and later for the Macintosh. Bill Gates borrowed the same ideas for Windows.

There was one aspect of PARC that was unique among computer environments of the time. Because Xerox's business was based on the quality of its copiers, researchers at PARC were predisposed to care about the quality of the printed page.

About a year after I had been at Xerox, I was asked to join the Imaging Sciences Lab, a new laboratory led

by Chuck Geschke. The charter of the Imaging Sciences Lab was to explore techniques for making high-level graphic programming interfaces for both displays and printers. This included figuring out how to represent the printed page in a resolution-independent way.

THE PROBLEMS TO BE SOLVED

From our perspective there were two major problems: how to build a computer representation, in a resolution-independent way, of *any* printed page; and, how to represent text, and typefaces, that are compatible with a solution to the first problem.

Since the time of Gutenberg, text and graphics have been treated separately, and this was also true at Xerox. The laser printers at Xerox were 240 dots per inch (dpi) while the computer screens were 72 dpi. Both are raster devices, which form images by coloring selected, closely spaced dots (called pixels)—the number of the dots per inch determines the resolution of the device.

To make high-quality pages, each letterform was designed as an array of dots with one array, called a bitmap, for each letter size for each device. It was commonly believed that a separate design for *each* size of *each* letter was necessary to preserve quality on raster devices, especially if they were low or medium resolution. For example, the following image illustrates enlarged characters for the 12-point size of the Times Roman font for 72 dpi and 240 dpi devices.

The researchers at PARC had laboriously crafted type designs for each font size for both the 72 dpi screen and the 240 dpi laser printers. This approach to representing character shapes was highly limiting because complete type libraries would have to be constructed for every new, different-resolution device that might be invented. Also, if a model was wanted that would allow for the scaling and rotation of page elements, then the PARC scheme would not be flexible enough. This limited representation of typefaces highly constrained the way printed pages could be depicted, but at the time, it was thought to be necessary.

At Xerox, Chuck Geschke and I worked with Butler Lampson, Bob Sproull, Brian Reed, and Gerry Mendleson on a printer protocol called Interpress, which was to be a standard for document representation for all Xerox printers. The design of Interpress incorporated some programming aspects; unfortunately, its design embodied the limitations in type representation that had become part of the Xerox culture.

In some sense, even with the above problems, Interpress was successful in that Xerox adopted it as a standard. Unfortunately, management would not expose its design to the public until all Xerox printers conformed to its implementation, which might never occur. This decision was very frustrating to say the least: we had all worked for two years on Interpress only to have it hidden by Xerox. I approached Chuck in May of 1982 and suggested we form our own company. In November of 1982 we left Xerox and started Adobe Systems.

THE LANGUAGE

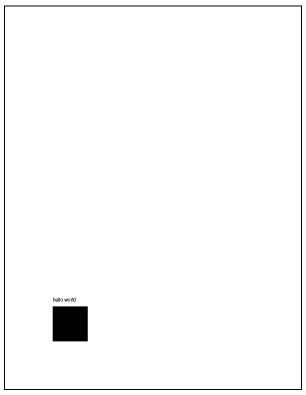
When I first joined PARC in 1978, I needed some tools to help me do graphics research. Before PARC, while working at Evans & Sutherland, John Gaffney and I developed a simple interpretive language called the Design System. I re-implemented the language at Xerox with the help of Martin Newell and Doug Wyatt, and called it JaM (for John and Martin). At Adobe, Chuck Geschke, Doug Brotz, Ed Taft, and Bill Paxton implemented the language yet again; we called it PostScript.

In general, at that time in the industry, printer protocols were encoded static-data structures—or mark-up languages—which defined a page. Any application program wanting to output to a printer had to build the encoded structure, or language, and deliver it to the printer. If the internal model of the application program differed from the model of the representation required by the printer, then building the interface to the printer drivers was problematic.

At Adobe we had a radical new idea: we would make programs written in PostScript the representation of a page. The PostScript interpreter would execute the program to build the raster array needed for the printer. As a result, a single PostScript program would produce exactly the same image when sent to any device equipped with a PostScript interpreter. For example, to draw a

black one-inch square with the text “hello world” on top of it in PostScript, one sent the following program to the interpreter:

```
0 setgray    %set the color to black
100 100 moveto    %go to lower left corner
100 172 lineto    %draw the left edge
172 172 lineto    %draw the top edge
172 100 lineto    %draw the right edge
closepath         %close the path
fill            %fill the path with current color
/Helvetica findfont %find the Helvetica font
10 scalefont      %make the font 10 points
setfont          %set current font to this font
100 184 moveto    %move to 12 pts above the left
                   %edge of the box
(hello world) show %show the text on the page
showpage        %print the page
```



The graphics model[1] supported by PostScript is really quite simple. There are two kinds of graphic objects: the paths composed of curves and straight lines, and rectangular arrays of sampled values representing images (a pixel-based image format, such as TIFF). The paths can be open or closed, and multiple paths can form a complex path. Paths can be filled with any color or stroked with a line type of any thickness, and with a variety of joins and end caps, and with any color. Paths can also act as clipping boundaries for images and other paths, i.e., an image can be displayed only in the interior of a path, and masked on the exterior. Images can be any rectangular image of any resolution (black and white or color). Images can also act as masks for painting operations.

Our goal was to implement a full programming language that provided all the graphics operators required to build complex pages. To achieve this, PostScript implements about 400 operators. An abbreviated example set of operators includes:

Math operators: add, sub, mul, div, sqrt, exp, log, sin, cos, tan, arctan, abs, floor, ceiling, etc.

Control operators: if, ifelse, for, forall, loop, repeat, exec, etc.

Graphic operators: moveto, lineto, curveto, arcto, closepath, fill, stroke, image, imagemask, clippath, show, etc.

Data construction operators: array, string, dict, etc.

File operators: file, read, readline, readhexstring, readstring, write, writehexstring, writestring, closefile, etc.

Graphic state operators: setgray, setrgbcolor, sethsbcolor, setlinewidth, setlinejoin, rotate, translate, scale, etc.

The list goes on, but the important thing was that even the most complex pages found in any book or magazine could be described with PostScript and the extensive set of operators.[2] Choosing a program to represent a document had several theoretical disadvantages:

1. The program could go into a loop and never stop.

2. Examining the program did not always tell you how many pages (if any) were produced.

3. The program dictated the order in which pages were produced.

Even with these drawbacks, selecting a program as the page’s representation had some major advantages:

1. Because each operator in the language could be redefined, it was possible to fix or extend the language’s capabilities without changing the interpreter.

2. Programs in the language could interpret existing printer protocols and emulate their behavior (this is how the first Apple Laserwriter became compatible with all the existing applications on the Macintosh).

3. Complex character encodings could be implemented programmatically, e.g., Japanese, Chinese, Farsi, Unicode. This capability allowed PostScript to handle multiple encodings and ultimately become an international standard.

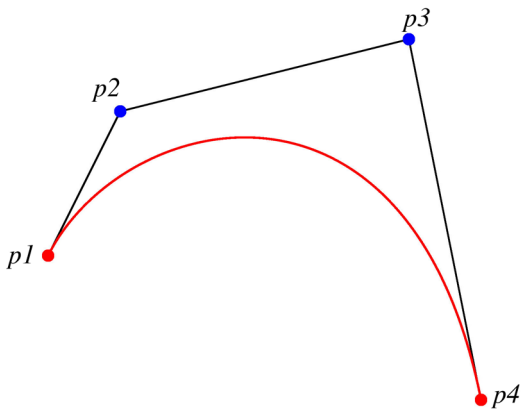
4. Subroutines in the language could efficiently represent complex sub pictures in a page.

THE REPRESENTATION OF CURVES AND CHARACTERS

So far nothing has been said about how curves, characters or typefaces were handled in PostScript. If we were to remain true to our goal of device-independent representations, then characters would have to be defined in the same way as any graphic shapes: by a collection of mathematical curves and straight lines (paths).

Curves in PostScript are defined as third-order Bézier curves (after the French mathematician Pierre Bézier).[3] We chose these curves because they are simple to describe, easy to render on a computer, and are flexible enough, when put together in combination, to describe the most complex shapes found in the graphic arts industry.

The interesting thing about Bézier curves is the very simplicity of the geometric construction that determines their points—any ordered four points (*p1*, *p2*, *p3*, and *p4*) on a plane define a third-order Bézier curve. For example, the four points below define the curve.



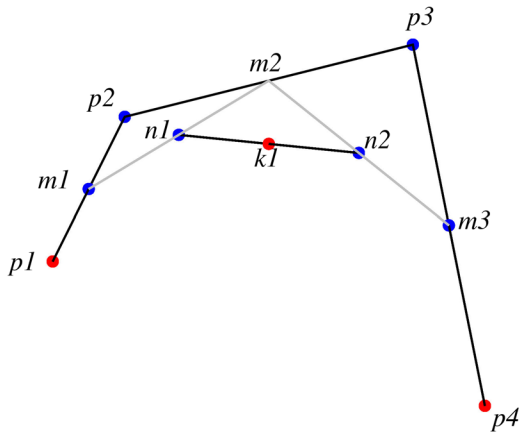
1. The first and last Bézier control points (*p1* and *p4*) are always on the curve.

2. To construct another point on the curve, find the midpoints of the segments between *p1* and *p2* (*m1*), between *p2* and *p3* (m2), and between *p3* and *p4* (*m3*).

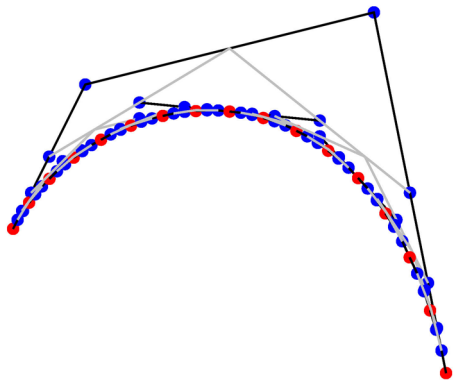
3. Find the midpoints of the segments between *m1* and *m2* (*n1*), and between *m2* and *m3* (*n2*). Connect *n1* and *n2* with a segment.

4. Find the midpoint (*k1*) of the segment between *n1* and *n2*.

5. The point *k1* is on the Bézier curve, the lefthand side of the curve has the Bézier control points *p1*, *m1*, *n1*, and *k1*. The righthand side of the curve has the Bézier control points *k1*, *n2*, *m3*, and *p4*.

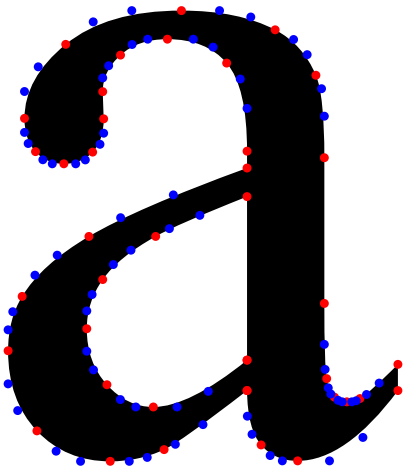


When this construction process is repeated recursively, points are generated all along the line. The process stops when all the control points for a segment are within an error tolerance for the resolution of the device, i.e., the process stops when the straight-line segment connecting the control points approximate the curve needed for the raster device. The result might be:



Here the red points are on the curve. Armed with this curve technology, it was now possible to define device-independent characters.

As an example, consider how to define the character *a* in the Times Roman typeface. The following image shows the Bézier control points that define the letter. The red dots are points on the curve. The pairs of blue dots together with the adjacent red dots are Beziér control points for that curve segment.



The program that defined this character in PostScript consisted of "moveto" "lineto" "curveto" and "closepath" commands. Tens of thousands of typefaces across all written languages are now defined in this way.

THE FONT PROBLEM

In 1982, it was widely believed that the only way to make acceptable fonts for low- and medium-resolution devices was to design the bitmap representations by hand. And there was a good reason for this: if one takes the outline description of a character and turns on the pixels where the character outline touches or covers a pixel, the results look horrible.

For instance, converting outlines to arrays of black dots looks like this:

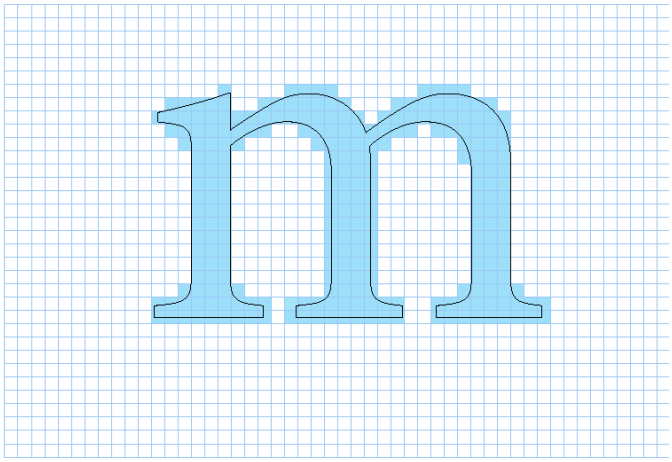
At Xerox, and elsewhere, a number of attempts were made to design an algorithm that figured out which pixels to turn on and which to turn off based on information from the outline of a character. None of these solutions worked when implemented over a wide variety of typefaces.

At Adobe we knew that if PostScript was going to be successful we had to solve this problem for printer-resolution devices. We had to find a way to generate high-quality bitmap representations of outlined characters automatically.

A VERY SIMPLE IDEA

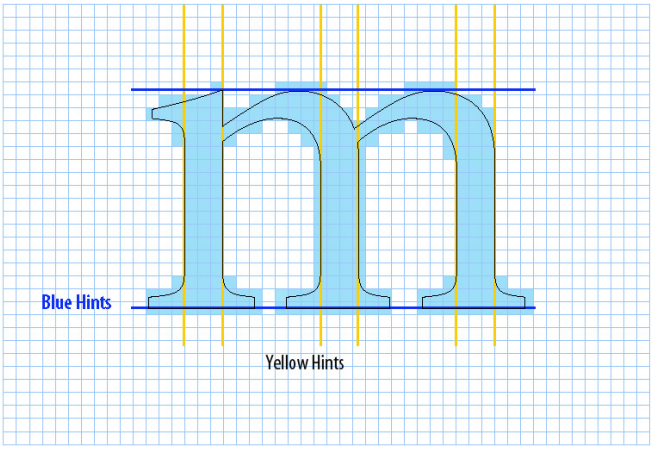
At one point, it occurred to me that we had been looking at the problem from the wrong point of view. Rather than trying to figure which bits to turn on based on the outline, one should change the outlines (in a minor way) at the time of raster conversion for a specific size of the character so that high-quality characters would be generated with a straightforward algorithm. Doug Brotz, Bill Paxton, and I worked on solving this problem.

To illustrate: After scaling the outline to the desired size, if the bitmap is generated with no change to the outline, then the result is:



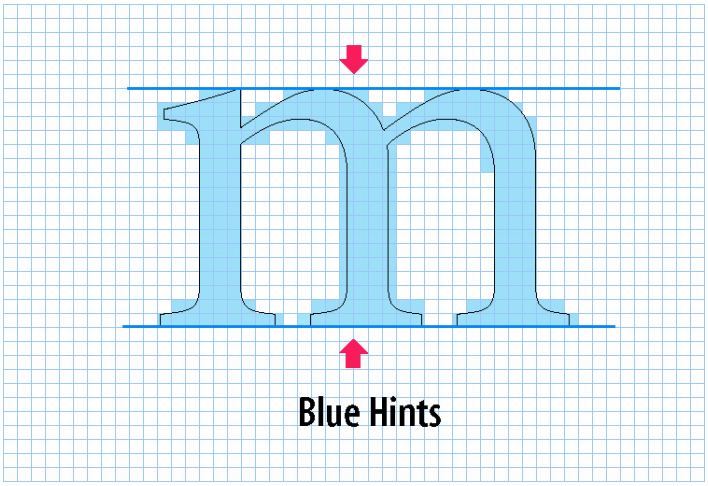
Notice that the center stem is four pixels wide, whereas the end stems are three pixels wide. This kind of error makes the characters look inconsistent and rough.

If you mark the vertical stems and/or curved parts that should be consistent, then you know how to align the curves with the dot matrix. If you also mark the baseline of the font family and the x-height, then the characters heights can be made to be consistent relative to the dot matrix and to each other.

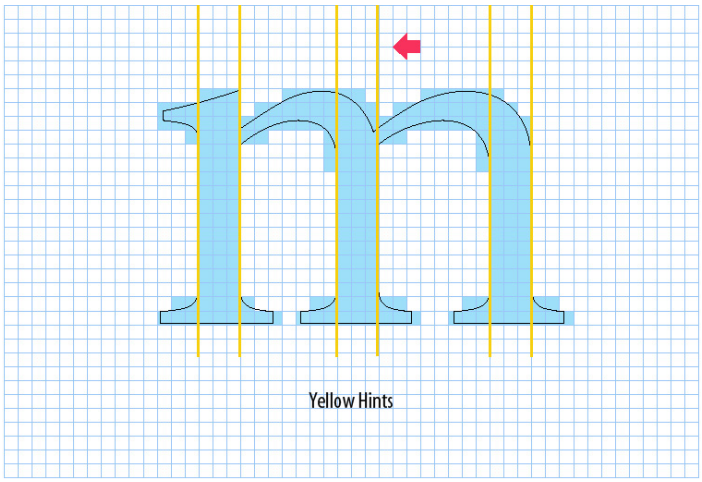


We marked the vertical strokes that were to be con-

sistent with pairs of vertical lines (we called these "yellow hints"). We marked the baselines and x-heights with two lines that applied to the whole alphabet ("blue hints"). We then stretched or compressed parts of the curves by moving the control points so that the yellow and blue hints were consistent relative to the dot matrix. In the case of the *m* (as with all characters of the font), the blue hints were moved to the closest raster boundary, and the curves expanded or condensed to correspond to those lines.

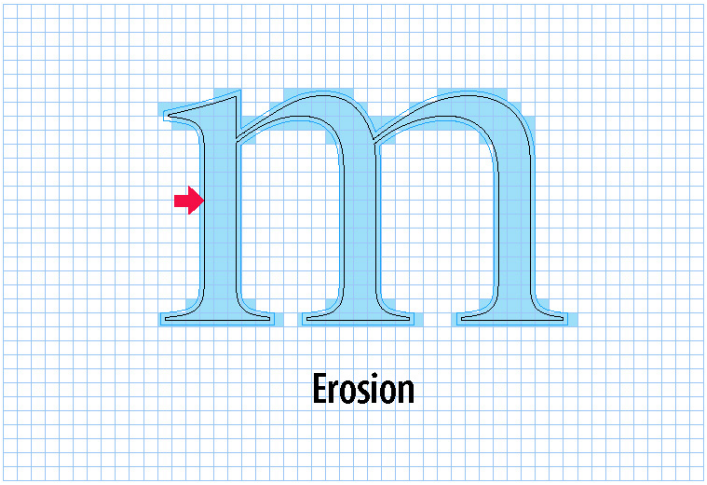


Next we added the yellow hints for each character. Pairs of yellow hints are put on curves or strokes where it is important that the stroke weights are consistent in the final character. In the case of the *m* the hint lines are put at the edges of the vertical staffs. Pairs of yellow hint lines are translated to the nearest raster boundaries. The control points of the curves inside the pairs of yellow hint lines are only translated, whereas the points outside the yellow hint lines are linearly interpolated depending on how much the movement of the hints compressed or expanded the region it defined. This transforms the curves accordingly: the center stem of the *m* shifts to the left to follow the hint.

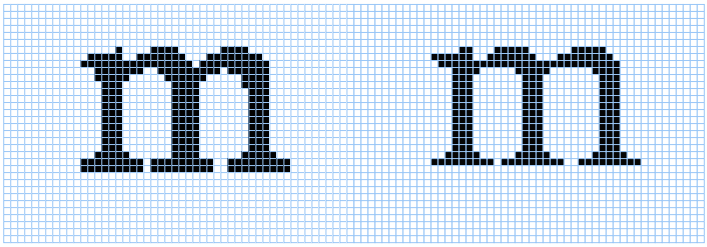


Although these changes to the curve worked extremely well when generating character bitmaps, the resulting fonts still looked heavy, i.e., too many bits were being turned on. In particular, diagonal lines looked much too heavy.

After much experimentation, we came up with the idea of "erosion." The basic idea was to shave the outlines as a function of angle (oblique lines are shaved more) to reduce the number of bits that are turned on. To accomplish this, the amount of erosion is a function

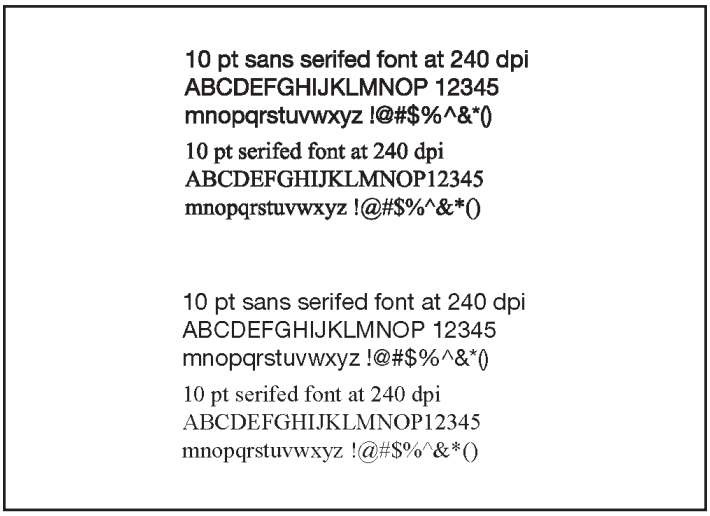


of the raster size and not the character size. After the erosion process, the bitmap for the character at that size was generated in a straightforward manner. All the raster squares that are touched are turned on. Below is a comparison of the "m" before hinting and erosion, and after.



The resulting bitmap of the "m" is remembered. All the lowercase *m*s of this size and orientation are identical, and can be used over and over as *m*s are required. In practice, for each size and orientation of a character, the character is built once and cached into a holding memory. This vastly reduces the amount of computing needed to generate a typical page.

Below is a comparison of type generated from outlines without hinting, and type generated from outlines with hinting.



It is interesting to note that the outline modification of the yellow and blue hints is applied only when the lines of text are parallel to either the *x* or *y* axis of the page. When fonts are rotated, there is no frequency in the raster that conflicts the horizontal and vertical strokes in the font. Even so, the outline erosion strategy is always applied. This combination of hints and erosion produced high-quality raster representations of the letters at medium and high resolution.

THE SUCCESSES OF POSTSCRIPT

In 1983 we began showing samples of PostScript output to a number of computer and computer printer suppliers as well as to some typesetter manufacturers. In particular, we met with Steve Jobs of Apple Computer who showed us the new Apple Macintosh about to be introduced in January of 1984. In mid 1983 we signed a contract to build a PostScript interpreter for the controller of the Apple Laserwriter, which was to be based on a Canon laserprinter. A little later that year, Linotype Corporation contracted with us to build a PostScript interpreter for a controller for their Linotronic 100 and 300 machines.

With the Apple Laserwriter, the Linotronic 100 and 300, the Apple Macintosh, Aldus' PageMaker program, and PostScript, desktop publishing was born.

The years following were very competitive: printer and computer manufacturers were faced with either supporting PostScript or introducing competing formats. In 1990, 31 companies were producing 120 different PostScript printers or imagesetters,[4] including Apple, IBM, Hewlett-Packard, Digital Equipment Corporation, Texas Instruments, Wang, Canon, NEC, Ricoh, Fujitsu, Matsushita, Compugraphic, Linotype, Autologic, Monotype, and Scitex. PostScript was becoming an international standard for printing and publishing.

In 1989 PostScript came under competitive pressure from Apple and Microsoft. They formed an alliance to introduce their own font format (called TrueType). In response, Bill Paxton, Steve Schiller, Tom Malloy, and Mike Byron improved the Adobe font technology so that high-quality fonts could be produced from Adobe's character outlines for low-resolution devices (screens).

This technology was called ATM (Adobe Type Manager), and allowed users to display high-quality type on display screens by scaling type within applications that ran on Macs and PCs.

THE NEXT INNOVATION

In 1990, large and medium size organizations increasingly began using local area networks. At the same time, almost all application programs for Macintosh computers and PCs produced PostScript files for printing. In 1991, it occurred to me that there was a way to use the ubiquity of PostScript to produce a file format that would reliably communicate documents between all different kinds of computers and operating systems.[5] PostScript itself was not a viable candidate for this purpose because:

1. PostScript files are programs that are not necessarily secure. It is generally dangerous to send programs between computers across networks.

2. It is impossible to examine a PostScript program and determine how many pages it will generate without actually executing it. To generate a particular page, one first has to generate all the pages that preceded it (at great cost in time).

3. A given PostScript program can compute a lot and produce little or no output.

The trick was to execute a PostScript program but redefine the graphics and graphic state operators: setrgbcolor, setcmykcolor, moveto, lineto, curveto, closepath, fill, stroke, translate, rotate, scale, findfont, setfont, scalefont, show, showpage, etc., so rather than build up the page image, these operators write their arguments before their operator names to a new text file.

This distilled file is a PostScript file but has only the graphic commands. All loops; conditional statements and control statements are absent from the file. This file is secure, has well delineated pages and has minimal

computing associated with its execution. From this file it is straightforward to build a data file that represents the document while retaining device independence: it is no longer a program itself. We called this new file format the "Portable Document Format," or PDF.

In 1993 Adobe introduced Acrobat. This application allowed users to read, navigate, annotate, and output PDF files from any application. Adobe also announced Acrobat Reader, a free application, which made it possible for users to read, navigate, and annotate PDF files. Since then, there have been over one billion downloads of Acrobat Reader between 2006 and 2010. As of 2010 there are roughly 170 million PDF files on the Internet, and probably many times that within organizations and corporations. PDF is a recognized National Archives Standard and an ISO Standard.

Since the days of PostScript, PDF has been extended to encompass all manner of media, including video, animation, 3D, sound, and online collaboration. The technologies around PDF have begun to replace the millions of paper workflows that exist in and between organizations. We will probably never attain a "paperless society," but if trends continue, our use of paper for communication will be drastically diminished.

The three simple ideas—modifying a computer language, PostScript, so it could *be* the representation any complex page, no matter the combinations or kinds of text and graphics; making use of Bézier's mathematical algorithm to make any typeface readable on screens and on paprer; and then using PostScript again as the basis for PDF, making possible the safe and accurate transmittal of any page anyone might think to create on a computer—have together built the foundation of modern electronic publishing.

Of course, in addition to the above ideas, it takes considerable talent and a huge amount of engineering to make successful the dozens of innovative products to continue to advance the ever-broadening media landscape.

Our traditional definition of a document, a sequential collection of words, images and graphics printed on paper, is changing daily. Documents in the electronic world can contain all kinds of media, can be interactive, and can dynamically link to all manner of other material.

How we cope with this changing base of information and how we save it for future generations is yet to be discovered.

1. John Warnock and Douglas K. Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," in *Computer Graphics* 16, no. 3 (July 1982): 313–19.

2. Adobe Systems Incorporated, *PostScript Language Reference Manual* (Reading, Mass.: Addison-Wesley, 1985).

3. Pierre Étienne Bézier and Paul de Faget de Casteljau were working independently and almost simultaneously on a system that would enable automobile designers to express a curve mathematically. Faget de Casteljau (at Citroën) found the solution first in 1959, but the company kept the algorithm a secret. His two Technical Reports, *Outillages, méthodes, calcul* (1959) and *Courbes et surfaces à pôles* (1963), did not become known to the outside world until Wolfgang Boehm obtained copies in 1975. Bézier (at Renault) began his researches in 1960, and published his "Définition numérique des courbes et surfaces" in two parts in the journal *Automatique* (1966–67). In the division of the nomenclatural spoils, since Bézier had, in all innocence, already been commemorated in the curve, Faget de Casteljau was allotted the algorithm. "It is unfortunate that the name of Paul de [Faget de] Casteljau is less well known than that of Bézier, since the curves of the latter would be of little interest without the algorithm of the former." Yannis Haralambous, *Fonts and Encodings,* trans. P. Scott Horne (Sebastopol, Cal.: O'Reilly Media, 2007), G2.

4. *Adobe Systems Incorporated 1990, Annual Report* (Mountain View, Cal.: Adobe Systems Incorporated, 1991), 2.

5. John Warnock, "The Camelot Project" (internal document, Adobe Systems Incorporated, 1991). Available at: http://www.planetpdf.com/planetpdf/pdfs/warnock_camelot.pdf