Technical Report

UCAM-CL-TR-87
ISSN 1476-2986

Number 87

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Computer-aided type face design

Kathleen Anne Carter

May 1986

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 1986 Kathleen Anne Carter

This technical report is based on a dissertation submitted November 1985 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

This thesis tackles the problems encountered when trying to carry out a creative and intuitive task, such as type face design, on a computer. A brief history of printing and type design sets the scene for a discussion of digital type. Existing methods for generating and handling digital type are presented and their relative merits are discussed. Consideration is also given to the nature of designing, independent of the tools used. The importance of intuition and experience in such a task is brought out. Any new tools must allow the designer to exercise his skills of hand and eye and to judge the results visually. The different abstractions that can be used to represent a typeface in a computer are discussed with respect to the manner of working that they force upon the designer.

In the light of this discussion some proposals are made for a new system for computer-aided type face design. This system must be highly interactive, providing rapid visual feedback in response to the designer's actions. Designing is a very unstructured task, frequently with a number of activities being pursued at once. Hence, the system must also be able to support multiple activities, with the user free to move between them at any time.

The characteristics of various types of interactive graphical environment are then considered. This discussion leads on to proposals for an environment suitable for supporting type face design. The proposed environment is based on the provision of a number of windows on the screen, each supporting a different activity. A mouse, graphics tablet and keyboard are all continuously available for interaction with the system. The rest of the thesis discusses the implementation of this interactive graphical environment and the type face design system that makes use of it. The final chapter evaluates the success of both the underlying software and of the type face design system itself.

Contents

Index of figures

Foreword

Intuition and Automation in Type Design

The design of type faces has a long tradition going back some five hundred years. Many of the problems and questions faced today have been encountered in some form in the past. Printing is a technological process and throughout its history it has had to assimilate and exploit new techniques. As Hermann Zapf says in his notes on type design [1970]:

> "The type form must subordinate itself to technical requirements, and be attentive to the increased demands for legibility from the reader; and it remains the type designer's task to be watchful that in modern mass production the letter's beauty be not lost."

It may seem that the computer has finally removed printing from the hands of skilled craftsmen and reduced it to an automatic process where intuition and aesthetics have no place. There is frequently tension between the engineer's pursuit of automation and speed and the designer's desire for beauty. Engineers have been taking decisions that typographers are trained to take and, on the other hand, typographers assume that the results produced by the engineers are the best that can be done. In fact the use of computers in printing provides a new freedom which, if used sensitively, opens up the possibility of mass produced printing of a quality never seen before. If this potential is to be realised the type designer must get involved and apply his skills to produce letter forms that best make use of the opportunities provided by the new technology, trusting the judgement of his own eyes to produce something beautiful and legible. On the other side, engineers must accept that intuitive judgements in type design are as valid as automatically calculated answers. This means producing systems that provide the freedom for designers to exercise their visual judgement. This potential clash between mechanisation and traditional skills is not peculiar to computers, to quote Daniel Berkeley Updike [1937]:

> "... it seems to be the eye and the hand that determine the excellence of the product of a machine, and it is only when the machine is as flexible as the hand that it is as good as the hand."

He was referring to punch cutting machines where the results were greatly improved when the operators learnt to trust the judgement of their eyes and the experience of their hands as they worked. Similarly, with computers the best results will be achieved if the designer's eyes and hands are allowed to be the final arbiters. Digital printing is a new medium with its own advantages and disadvantages and it will be put to best use as designers develop new skills and gain experience with it.

This thesis is about the provision of tools to enable designers to work in the digital medium and so to produce type faces that are beautiful and appropriate to this new medium. There are two strands to the thesis, that of type face design and that of interactive computer graphics. They come together in an interactive system for computer-aided type face design. The first part of the thesis covers type face

design, both the task in general and the specific questions that arise when computers are used to help in this task. A brief historical summary of printing techniques provides the background to modern trends and to a discussion of designing in general. Existing systems for handling digital type are discussed and then some proposals made for a new system to handle the whole design process. Any computer system to support a visual and creative process like type face design must make use of interactive graphics. The second part of the thesis surveys interactive computer graphics, culminating in proposals for the facilities needed to support the type face design system. The actual details of the facilities implemented are then presented. The third part then presents Imp, the proposed system for type face design as it has actually been implemented by the author. The final chapter evaluates the success of the work that has been done in the light of the original proposals and also presents some ideas for future developments. Two appendices are included to further illustrate the work. The first is a user manual for Imp, written by Lynn Ruggles, which complements the more technical discussion in the main body of the thesis. The second appendix is a video tape which shows what Imp is like in action.

1. Type Face Design

This chapter provides the background needed before any attempt can be made to produce tools for type face design. First a brief history of printing sets the scene and then the task of type face design itself is discussed. Much of the history has been taken from "Five Hundred Years of Printing" by Steinberg [1955] and further details can be found there.

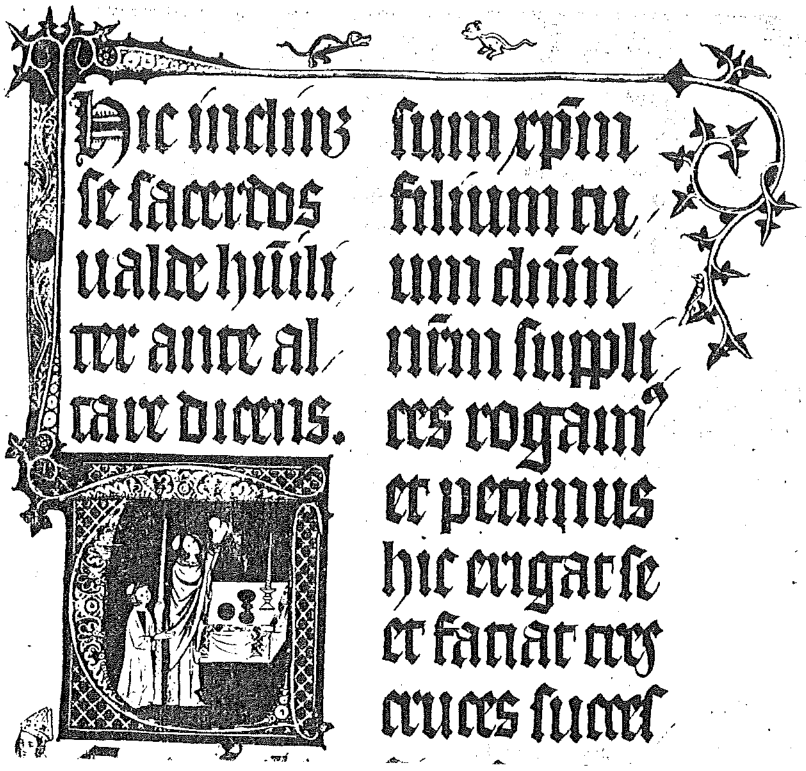
1.1 A Brief History of Printing

The Early Days
The history of printing from movable metal types is generally considered to have started with Johannes Gutenberg around the year 1455. Although it is probable that others were using metal types before this time, it was Gutenberg who made it a practical and profitable proposition. His process consisted of several stages for the production of each character, and these stages are still in use today for metal type production. First the character shape was carved on the end of a steel bar, called the punch. This punch was then pressed into a bar of copper to form a matrix. The matrix was fitted into a mould and then the type was formed by pouring in molten metal. Once the type had been cast it could be assembled into lines and then pages for printing.

The first generation of printers set out to imitate the handwritten manuscripts familiar to their readers and did not see printing as a new departure. They attempted to make the shapes of the type as close as possible to the pen written shapes, in spite of the difficulties of carving such shapes into metal. The scribes used large numbers of ligatures and abbreviations to speed up the production of manuscripts and these were taken over wholesale by the early printers (figure 1.1). Gutenberg used more than 250 ligatures in his work. As Updike [1937] says:

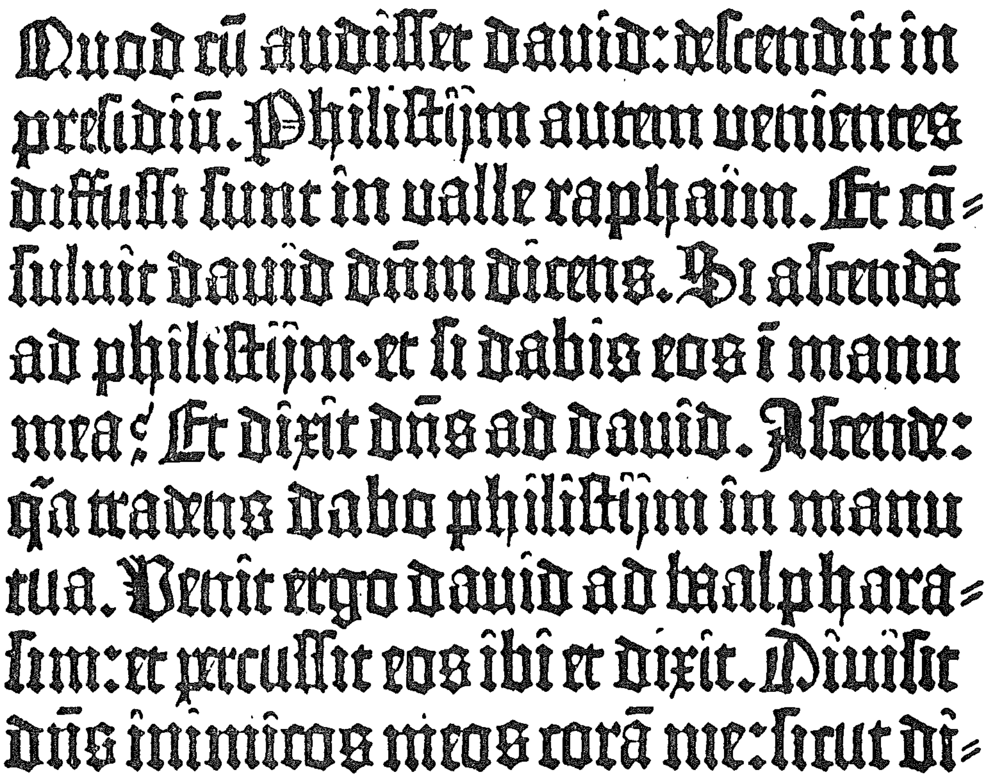
> "Intent upon imitating manuscripts, they felt obliged to reproduce the kind of letters that a reader had been accustomed to in volumes written by hand ... In other words, to the first type-cutters printing was merely an evolution, and did not appear a new invention in the sense that it obliged them to decide what forms of letter were best adapted to the new medium they had to employ. If these craftsmen had but thought of the whole subject from a fresh standpoint ... Instead of a long series of endeavours which have not yet entirely adjusted type-forms to the medium in which the type-cutter has to work, we should then have had characters designed with closer relation to the material from which they were fashioned."

The next generation of printers were able to break free from the scribes and began to develop typography as a craft in its own right. The most notable of these was Nicolas Jenson who, around 1470, designed a roman type face based on humanist scripts. These scripts were derived from carved Roman inscriptions and the forms were much better suited for the new technology of punch cutting. The reduction in the number of ligatures and special types also made the fount more manageable



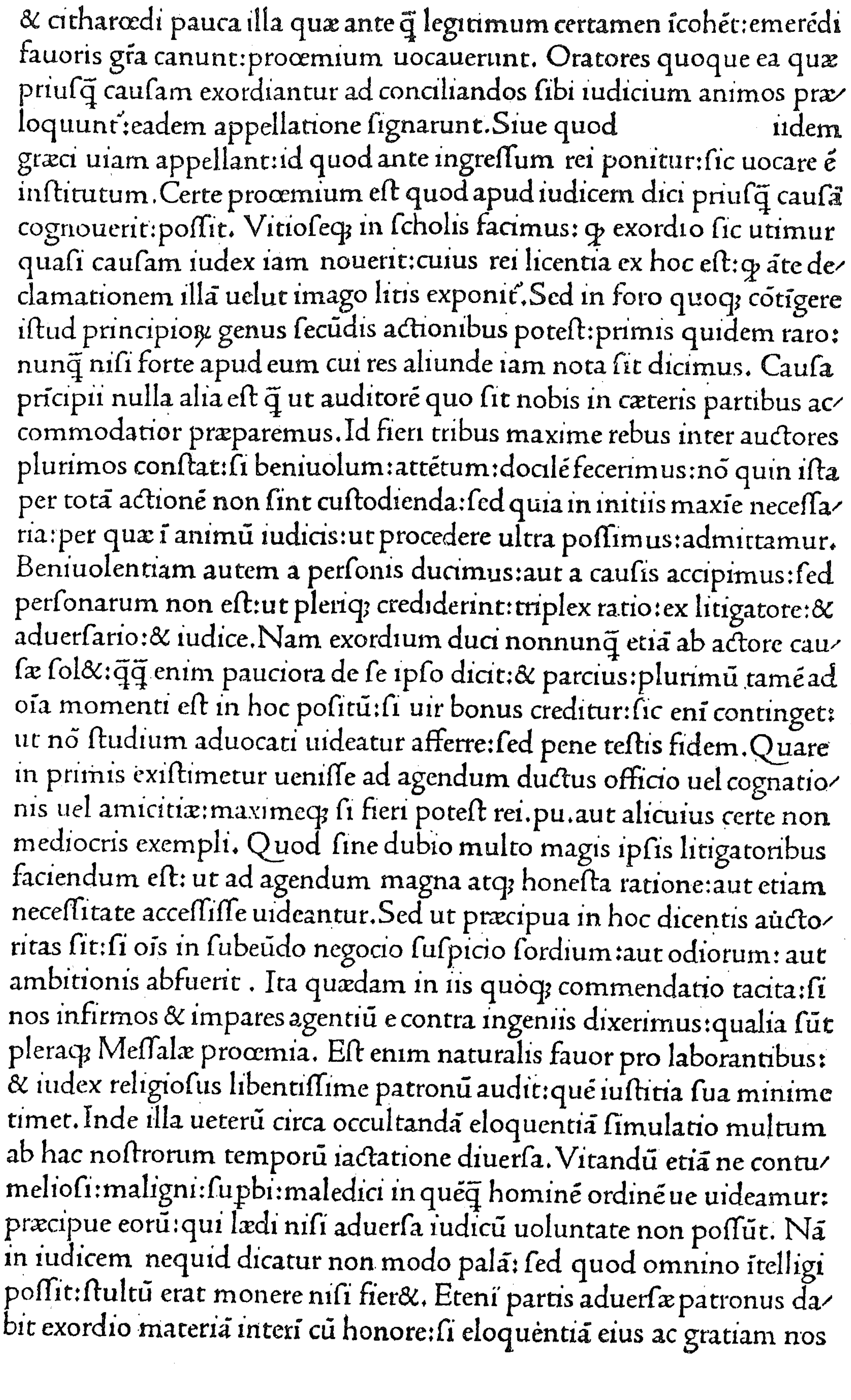
Hic inclinat
ſe ſacerdos
ualde hūili
ter ante al
tare dicens.

ſuum xp̄m
filium eius
um dn̄m
nr̄m ſuppli
ces rogam⁹
et petimus
hic erigat ſe
et ſignat cru
ces ſucceſ



Quod cū audiſſet dauid: deſcendit in
preſidiū. Philiſtijm autem uenientes
diffuſi ſunt in ualle raphaim. Et cō=
ſuluit dauid dn̄m dicens. Si aſcendā
ad philiſtijm· et ſi dabis eos ī manu
mea? Et dixit dn̄s ad dauid. Aſcende:
q̄a tradens dabo philiſtijm in manu
tua. Venit ergo dauid ad baalphara=
ſim: et percuſſit eos ibi et dixit. Diuiſit
dn̄s inimicos meos corā me: ſicut di=

Figure 1.1
A sample from a manuscript written in Germany in the 14th or 15th Century (top) and from Gutenberg's 42-line Bible printed about 1455 (bottom).



& citharœdi pauca illa quæ ante q̃ legitimum certamen ĩcohẽt:emerẽdi
fauoris grã canunt:proœmium uocauerunt. Oratores quoque ea quæ
priuſq̃ cauſam exordiantur ad conciliandos ſibi iudicium animos præ/
loquunt̃:eadem appellatione ſignarunt.Siue quod iidem
græci uiam appellant:id quod ante ingreſſum rei ponitur:ſic uocare ẽ
inſtitutum.Certe proœmium eſt quod apud iudicem dici priuſq̃ cauſã
cognouerit:poſſit. Vitioſeq; in ſcholis facimus: ꝙ exordio ſic utimur
quaſi cauſam iudex iam nouerit:cuius rei licentia ex hoc eſt:ꝙ ãte de/
clamationem illã uelut imago litis exponit̃.Sed in foro quoq; cõtigere
iſtud principioꝝ genus ſecũdis actionibus poteſt:primis quidem raro:
nunq̃ niſi forte apud eum cui res aliunde iam nota ſit dicimus. Cauſa
prĩcipii nulla alia eſt q̃ ut auditorẽ quo ſit nobis in cæteris partibus ac/
commodatior præparemus.Id fieri tribus maxime rebus inter auctores
plurimos conſtat:ſi beniuolum:attẽtum:docilẽ fecerimus:nõ quin iſta
per totã actionẽ non ſint cuſtodienda:ſed quia in initiis maxĩe neceſſa/
ria:per quæ ĩ animũ iudicis:ut procedere ultra poſſimus:admittamur.
Beniuolentiam autem a perſonis ducimus:aut a cauſis accipimus:ſed
perſonarum non eſt:ut pleriq; crediderint:triplex ratio:ex litigatore:&
aduerſario:& iudice.Nam exordium duci nonnunq̃ etiã ab actore cau/
ſæ ſol&:q̃q̃ enim pauciora de ſe ipſo dicit:& parcius:plurimũ tamẽ ad
oĩa momenti eſt in hoc poſitũ:ſi uir bonus creditur:ſic enĩ continget:
ut nõ ſtudium aduocati uideatur afferre:ſed pene teſtis fidem.Quare
in primis exiſtimetur ueniſſe ad agendum ductus officio uel cognatio/
nis uel amicitiæ:maximeq; ſi fieri poteſt rei.pu.aut alicuius certe non
mediocris exempli. Quod ſine dubio multo magis ipſis litigatoribus
faciendum eſt: ut ad agendum magna atq; honeſta ratione:aut etiam
neceſſitate acceſſiſſe uideantur.Sed ut præcipua in hoc dicentis aucto/
ritas ſit:ſi oĩs in ſubeũdo negocio ſuſpicio ſordium:aut odiorum: aut
ambitionis abfuerit . Ita quædam in iis quoq; commendatio tacita:ſi
nos infirmos & impares agentiũ e contra ingeniis dixerimus:qualia ſũt
pleraq; Meſſalæ proœmia. Eſt enim naturalis fauor pro laborantibus:
& iudex religioſus libentiſſime patronũ audit:quẽ iuſtitia ſua minime
timet.Inde illa ueterũ circa occultandã eloquentiã ſimulatio multum
ab hac noſtrorum temporũ iactatione diuerſa.Vitandũ etiã ne contu/
melioſi:maligni:ſupbi:maledici in quẽq̃ hominẽ ordinẽue uideamur:
præcipue eorũ:qui lædi niſi aduerſa iudicũ uoluntate non poſſũt. Nã
in iudicem nequid dicatur non modo palã: ſed quod omnino ĩtelligi
poſſit:ſtultũ erat monere niſi fier&. Etenĩ partis aduerſæ patronus da/
bit exordio materiã interĩ cũ honore:ſi eloquẽtiã eius ac gratiam nos

Figure 1.2
An example of Jenson's Roman type of 1471

(figure 1.2). In spite of these advantages it did not seem to be the aesthetic or practical considerations that first caused the adoption of roman type faces. At this time much humanist literature was being produced, stimulated by a rediscovery of classical texts. A new type face was desired that would better express this new learning and make a clean break with the black letter script associated with theological writings. The roman and the associated italic faces soon spread throughout Europe along with the new humanist literature. This desire to use a new type face to spread new ideas has continued throughout the history of printing and perhaps today it is most clearly seen in advertising typography.

The use of rectangular metal types presented certain difficulties which had to be overcome. Letters have to be placed on the body of the type in such a way that they appear evenly spaced when set together. For example, curved shapes have to be placed closer than straight shapes to appear the same distance apart (figure 1.3). Some letters, particularly in italic founts, actually need to overlap their neighbours, a process known as kerning (figure 1.4). An area where punch cutters developed special skills was in the production of type in a range of sizes. Identical designs reproduced in different sizes do not actually appear the same because of the way the human visual system works. Larger sizes need longer ascenders and descenders and thinner strokes to retain the spirit of the design (figure 1.5). The punch cutter, trusting the judgement of his eyes would automatically make these changes to a single original design as he produced the different sizes.

In the early days of printing it was often the same person who designed, cut and cast the type, chose the books to be published and carried out the setting and printing. These tasks fairly rapidly differentiated. Claude Garamond, who worked in the early sixteenth century, could be considered as the first specialist type designer and cutter. Eventually the production of type was completely separated from printing and publishing, with the establishment of type foundries specialising in the design and casting of types for many different printers. Within type production itself the design of type was increasingly separated from the cutting of punches. The designer had to work closely with the punch cutter if his designs were to be realised as he intended.

Until the nineteenth century there was little further change in the industry beyond a consolidation and refinement of techniques. Better quality paper and more accurate presses meant that clearer impressions of the type could be obtained. This encouraged the production of designs with finer details as these were no longer lost in the printing process. For example delicate serifs and fine hairlines could be used, first seen in the types of Baskerville and later in those of Bodoni and Didot. The design of type faces became more systematic with the production of families of related faces of different sizes and attempts to produce related roman and italic faces. A further systematization was attempted in the reign of Louis XIV in France. He commissioned a set of type faces, called "Romain du roi", which were designed by a committee of the Academy of Sciences. The design was specified to fit to a grid but in fact Philippe Grandjean, who actually cut the type, produced a much more mellow and free design. There were various other attempts to reduce type face designs to mathematical constructions but never with any success. Ultimately the intuitions and experience of the punch cutters determined the designs.

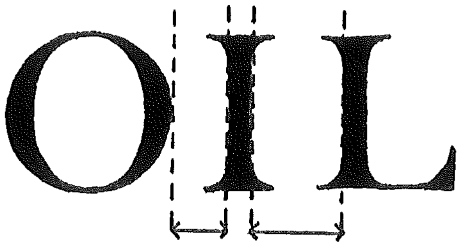


Figure 1.3
This shows the different distance needed between curved and straight strokes to get the appearance of even spacing.

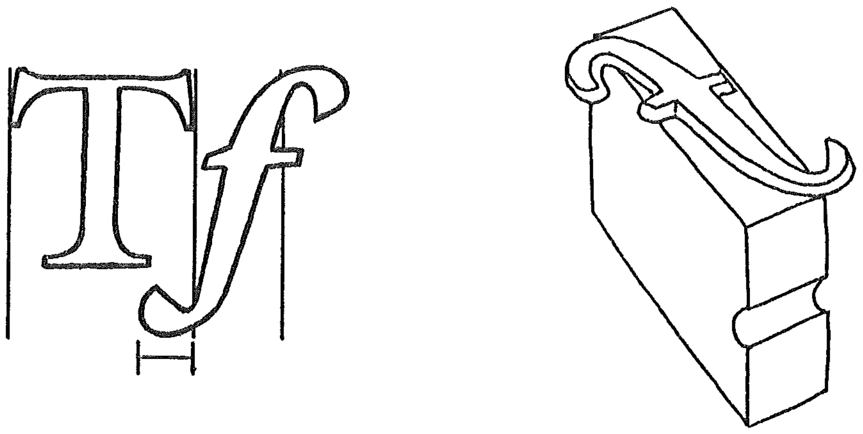


Figure 1.4
Kerned type.

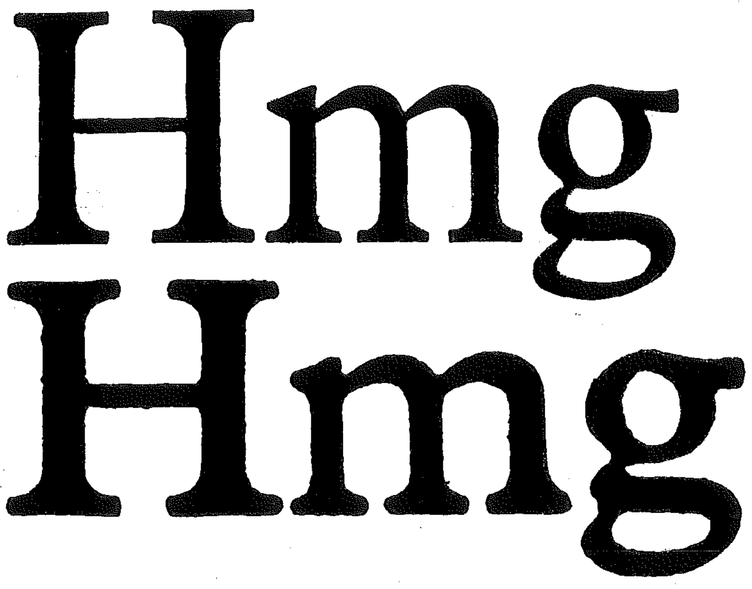


Figure 1.5
Monotype Plantin 24 point (top) and 10 point (bottom), both enlarged to the same size.

Modern Times

The nineteenth century saw the first radical changes in printing technology since Gutenberg's day. Until this time all printing was done with hand presses, essentially unchanged for centuries. The first power-driven cylinder presses were installed in the early years of the century for the production of newspapers. With the introduction of various mechanised techniques there was a vast increase in the quantity of material printed, although unfortunately the quality of design deteriorated. The type faces themselves were still cut and cast by hand until 1884 when a punch cutting machine was introduced by Linn Boyd Benton. This made use of a pantograph to transfer a large scale design to the punches at various sizes and removed the need for highly skilled punch cutters.

By 1900 the introduction of the Monotype and Linotype systems had revolutionised the casting and setting of type. The Linotype system has a single operator who types in the text; as he types the matrices are assembled and when a line is complete the whole line is cast as a single piece of metal. The resulting "slugs" are easy to handle but if there are any mistakes the whole line must be reset. The Monotype system casts and composes individual types similar to those cast by hand. It is done in two stages, one operator typing in the text and producing paper tape with the text and spacing information and the other feeding the tape into a casting machine. It is slower than Linotype but it is typographically superior and corrections are more easily done. These two systems rapidly became the standard for books, magazines and newspapers until the advent of filmsetting. The implications for type face design were serious. In order to make different founts available within one line, the Linotype matrices each contained two characters, often a roman and an italic. As a result, the italic designs had to be distorted to fit the same bodies as their corresponding roman letters (figure 1.6). With the Monotype system characters had to be designed so that their widths were a multiple of a fixed unit size, with 18 units to the em. Types were often produced for use on both sorts of machine and so had to fit the limitations of both.

Filmsetting

The past twenty years or so have seen dramatic changes in printing technology, starting with the introduction of filmsetting and now digital systems. Printing has been freed from the physical limitations imposed by the use of metal types. The type can be arranged in any way, even touching or overlapping, and it can also be distorted in various ways. In filmsetting the type is stored as black and white images, for example on a transparent disc. The image can then be projected onto the photographic film that will be used for producing the actual metal plate for printing. By using appropriate optics it is possible to produce all sizes of a face from one master design and also to stretch, shrink or slant the letters.

Many of the type faces used for filmsetting were copied directly from metal faces with little consideration of the effects of the new methods on the appearance of the type. A particular problem is the production of different sizes from the same master. When types were cast in metal each size had to be cut separately and so subtle adjustments could be made to cope with certain visual effects. By using the same master for all sizes the results are very different, and usually less attractive, than equivalent metal setting. Some manufacturers have taken this into account and produced masters that are not the same as any one metal size but look reasonable at

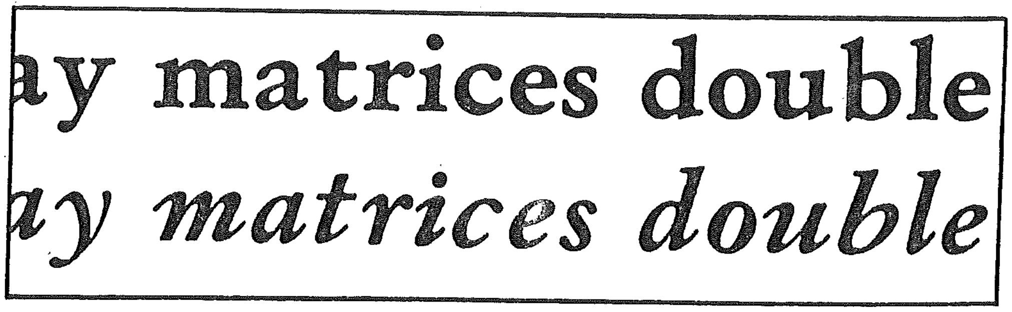


Figure 1.6
Italic and Roman letters fitted to the same bodies for the Linotype machine. The italic spacing is too wide and the letters do not have a uniform slant.

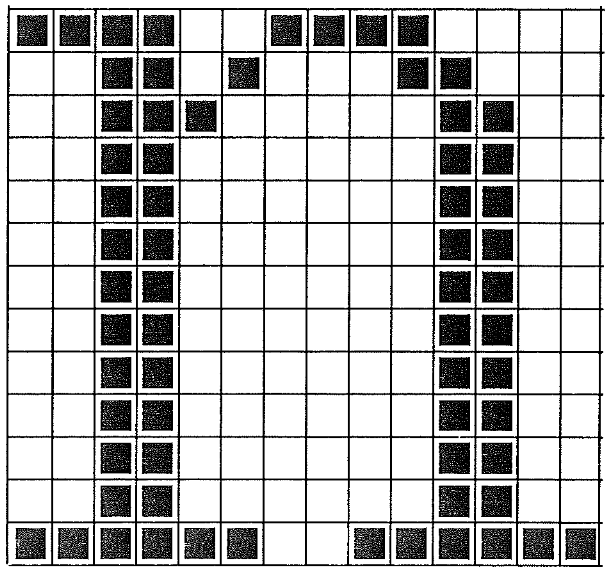


Figure 1.7
A simple digital letter

all sizes—another case of adapting the design to fit the technology. Further improvements are achieved by providing a number of masters, each one covering a small range of sizes. The designs also need adjustment for the lack of inkspread which makes the type appear lighter than its metal equivalent. An interesting example of a design made specially for this new technology is "Trinité" by Bram deDoes [1985]. This type face provides versions with three different lengths of ascenders and descenders, three different weights, and condensed, wide, italic and small capital founts. The user can then choose the particular combination of features that best suit the size, type of paper, intended reading conditions and the sort of text being set.

Now that type is no longer constrained to fit onto rectangular blocks of metal it is possible to achieve a much higher standard of setting. There is now no physical reason why letters should not overlap and so there is no need to distort letters to avoid kerning. With a computer to control the filmsetter it is possible to store kerning information and produce correctly kerned setting without taking a large amount of time or effort. Computers can also be used to calculate the kerning information, for example with the Logos program [Kindersley and Wiseman, 1978], so helping the type face designer to make best use of the new freedom. Unfortunately many systems still retain the old metal spacing units and so fail to exploit the flexibility of filmsetting.

Digital Printing

The use of digital printing introduces a form of type that is totally different from anything that has been used before. Digital type is stored in the form of a description of the image rather than as the image itself. A computer must interpret this description and generate the image on the film. At the lowest level digital type must define a bitmap, a pattern for the presence or absence of ink on a grid (figure 1.7). Many digital printers take letters defined in this way directly or, more usually, encoded to save space. Space can also be saved by storing the letters as a higher level description such as some form of outline encoding and there are some machines which will interpret such an encoding directly.

The use of a computer, along with abstract descriptions of the letters, opens up a whole range of new possibilities for manipulating letter shapes. As with optical film setting the letters can be stretched, shrunk and slanted but over much wider ranges. In addition, depending on the representation chosen for the shapes, a whole number of options may be available. The next chapter discusses these in some detail and so no more need be said here.

Perhaps we now have a situation similar to that when printing was first introduced. Just as the first printers copied their letter forms from manuscripts, we now have digital printing systems taking their letter forms from traditional type. These type faces were developed for cutting into metal with its particular characteristics and limitations and there is no reason why they should be suitable for digital reproduction. If the new medium is to realise its full potential there will be a need for type faces specially designed to exploit the new technology. There is now no need to stick to the limitations imposed by the use of metal type. Although these new types could be designed in traditional ways, the use of computers to aid the design process as well as the printing would provide a valuable step forward. The designers would gain firsthand experience of the digital medium and so be better able to take advantage of it.

1.2 The Process of Type Face Design

What is designing?
We must understand what the task is before we can begin to provide suitable tools, computerised or otherwise. In type face design, as in many other areas of designing, the process leading to a finished design is vitally important. Generally designs do not appear fully formed in a designer's mind, waiting only for a suitable medium for expression. Rather, a design is arrived at by a process of experimentation and refinement. The designer will probably start with some vague idea which can then be looked at and played with to give rise to further ideas. Throughout it is his trained eye that assesses what has been done and his intuitions that suggest what should be tried next. Anything might happen, and it is this air of the unexpected that gives life to a design. Although there may be rules and guidelines that can be applied, it is often as they are broken by an experienced designer that the best designs are produced. Central to the whole process are the designer's experience and intuitions and any aids or tools for designing must allow for this.

The traditional tools of a designer are the drawing board and implements such as pens, pencils and brushes. His draughting skills allow him to draw any shape he can visualise. He will probably use rulers and gauges to guide the proportions of letters and maybe French curves to construct curved portions but it is the skills of hand and eye that determine the shapes. The important feature of all these tools is that they can be used directly and visually, with immediate feedback of the results. Ultimately they can become effectively invisible, with the designer using them without thinking about it. This invisibility helps establish a direct link between what is in the designer's mind and the image he produces, in whatever medium he chooses. There is no need for him to think about how to produce the image, he just "does" it. Any new tool, if it is to extend and encourage a designer's creativity needs to provide this immediacy and potential invisibility.

Aspects of type face design
Looking at type face design specifically we see a number of different aspects that must be considered. Underlying everything is the fact that individual letter shapes do not exist in isolation; they exist as part of a complete fount of letters which belong together. Their proportions and shapes are in some way harmonious and distinguish this type face from any other. Some features can be easily isolated, for example the shapes of serifs or the relationship between thick and thin strokes in a letter, but others, such as the overall feel of the type face are much harder to pin down. When a new type face is being designed decisions must be made about these various aspects of the design, either at the outset or as the design progresses.

As a designer works he must consider the type face at a number of different levels, related to the different aspects that characterise the type. At the finest level there are details of serif shape and subtle curves within a letter to be worked on. Moving up, we must decide on the relationship between x-height and cap-height and ascenders and descenders, and also the relative widths and heights of letters. Individual letters do not generally stand alone and so at the next level we must look at the interactions between letters in order to set up the spacing. Ultimately we see whole pages of text set using the type—the final product of the design process. Decisions made at one level affect all the others, and so the designer must bear them all in mind as he works.

Intuition and automation

When we consider using computers to aid in the design process we find two opposing forces at work: the technologist's push towards automation against the designer's desire to use his intuitions and experience. The technologist's tendency is to analyse the results required and then to specify them in some abstract form. They can be manipulated, controlled and proved and algorithms constructed for producing the results. On the other hand, for a graphic designer what matters is whether the results look right, regardless of theoretical considerations. In its place the analytical approach is perfectly valid—for example a bridge must be structurally sound and this can be determined by calculation. It is not enough that the bridge look right, although even here a person can develop intuitions about what will work, providing a valuable double check against calculations that might go wrong. On the other hand, in graphic design the appearance is foremost and any calculation or algorithm is ultimately subordinate to the designer's visual judgement. For example, certain strokes in a fount may be intended to appear vertical, but optical effects can cause such strokes to appear slanted if they are drawn exactly vertically. The designer must trust his eyes and adjust the slant until the stroke appears vertical, rather than believing a calculation that says that the line is vertical. There are many similar optical illusions that affect type face designs and so it is very important to make room for intuitive, visual judgements.

Linked with the central role of intuition in designing is the air of the unexpected that was mentioned above. Perhaps it can best be described as playfulness. It is a fundamentally human characteristic that cannot be automatically programmed or planned for. Any computer system intended to support designing should provide the freedom for unstructured working that allows experimentation. Computers can provide many facilities that do not exist at a drawing board and playing with these can be the best way to learn how to exploit them. It may be that such facilities will be used in ways that could not be anticipated by the builders of the system.

An area where automation can be very valuable is that of organising the whole design task. A type face consists of a large number of letters and symbols and these must all be designed to fit together. As has already been mentioned, the designs must be considered at a number of levels and a computer system can provide the ideal environment for this. With the letters stored in digital form they can be displayed magnified to work on details, placed with a few other letters to assess spacing and also be set to form complete pages, all within one environment. It is even possible to have all the levels visible simultaneously. Such an environment could provide a great stimulus to inventive new designing as little investment is needed before even the wildest of ideas can be seen in print. The computer can organise the storage and retrieval of large numbers of designs that would be unwieldy, if not totally unmanageable, if drawn on paper.

2. Producing Digital Type

When we consider producing digital type there are two different points to be looked at. The first is how the type is to be stored for use and the second, obviously related, point is how the digital type is to be generated. These issues are discussed in general in the first section of this chapter and then the next section presents some existing systems for producing digital type faces. The final section makes some proposals for a new digital type face design system, drawing on the discussion of this and the previous chapter. Imp, the system described later in the thesis, was implemented on the basis of these proposals.

2.1 Handling digital type

Digital type is always an abstraction in that it is not stored as images but as instructions for constructing the images. At the lowest level each letter is made up of ink spots on a grid, with the positions of these spots forming the definition of the letter shape. For printing, these definitions must be stored compactly so that a large number of founts can be provided easily. On the other hand, various automatic manipulations of the letter shapes within the type setter can be very useful and so the representation chosen may need to take this into account. The generation of digital type can be seen as two different tasks, either the digitization of existing designs or the origination of new designs. In the former case the main aims are speed and accuracy, with adjustment of the shapes once digitized being minimal. In the latter case, as was emphasised in the previous chapter, the reworking and adjustment of the shapes is a central part of the process. Although I have made a distinction between the two tasks it can really be said that there is no such thing as straightforward digitization. Any design transferred to a new medium needs reworking to take account of its different characteristics. Saying that a system is for digitization only can allow certain simplifications but ultimately the best digitization systems will also be good design systems. What follows discusses the three main representations of digital type with respect to these considerations.

Bit maps

The simplest, but most bulky, way of storing a digitized letter is as a bit map. An array of bits represents the grid of ink spots (pixels) with a one representing presence of ink and a zero its absence. Such an array can be generated easily and directly by taking a large print of the letter and scanning it optically. A television camera can be used but better results are obtained with specially built scanning equipment. Unfortunately the results are often very noisy and so need hand editing to make them usable (figure 2.1). Bit maps can also be produced by using a digital painting system which allows the user to "paint" into the bit map using an electronic stylus. This can feel very close to conventional designing at a drawing board as it only requires draughting skills. There is no need to analyse what is being done, as is the case with more structured representations. On the other hand the lack of structure means that it is hard to exploit the power of the computer to carry out automatic manipulations. The bit maps are fixed for one size and resolution of device and would need to be either regenerated from scratch or considerably

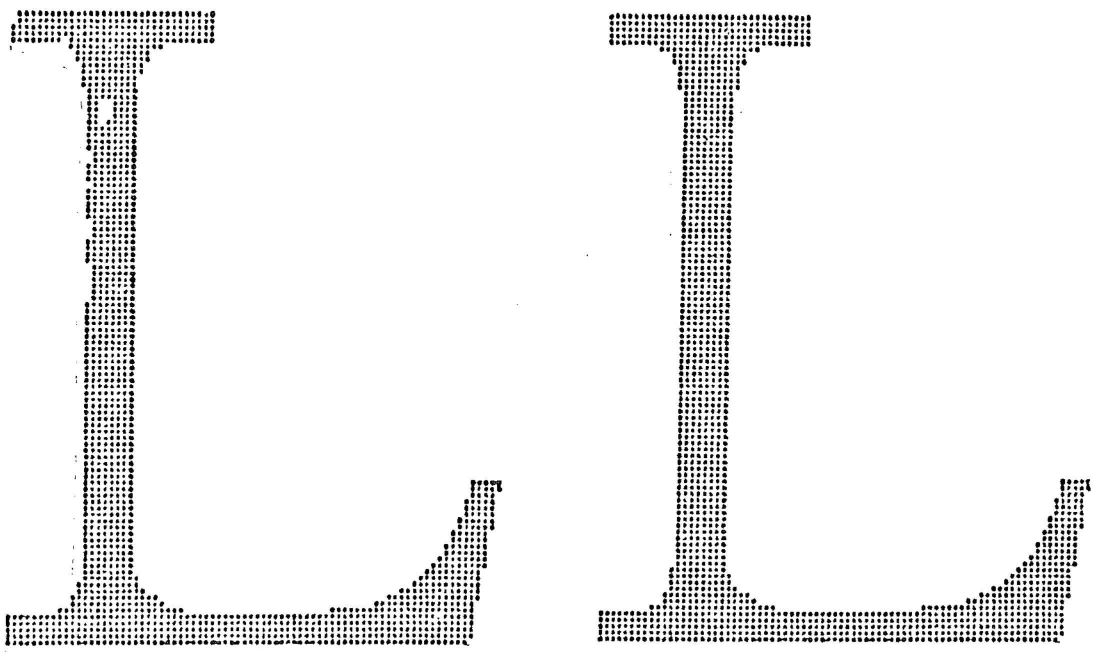


Figure 2.1
Bitmap for an optically scanned character before and after cleaning.

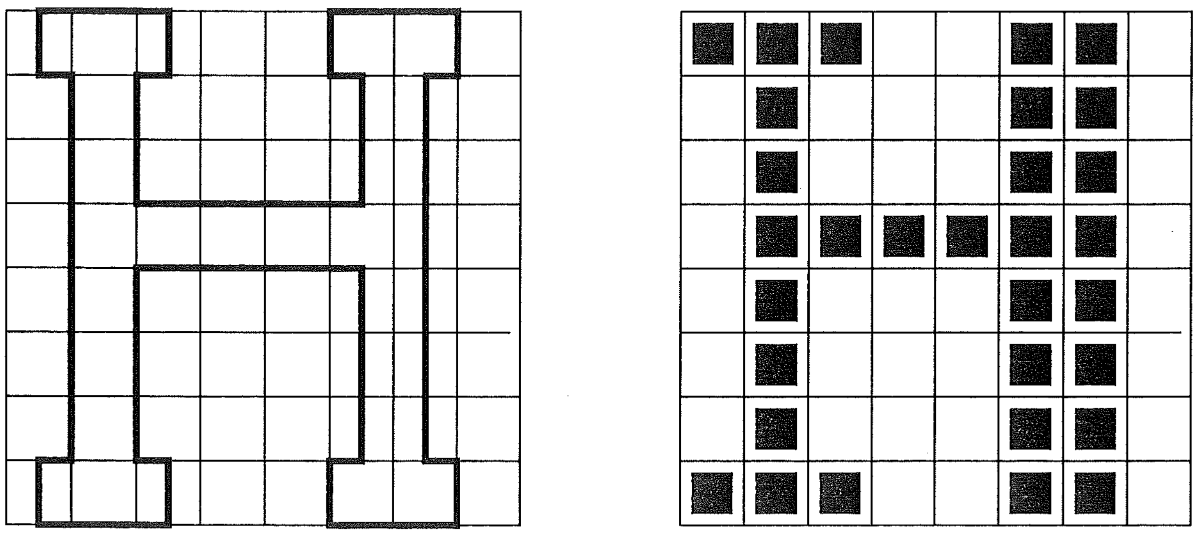


Figure 2.2
This shows a problem that arises with scan conversion. Pixels are filled in if half or more of their area falls within the outline. If the outline does not fall exactly on pixel boundaries we can get a symetrical shape appearing assymetrical when scan converted.

reworked for new sizes or devices.

A straightforward bit map representation wastes a lot of space—for example space is taken to represent all the blank area surrounding the letter. Various encodings can be used to save this space, with the encoding used depending on the capabilities of the decoding device. The storage of founts in the type setter is not a concern of this thesis and, because it is such a large topic, no more will be said here. Obviously any design system that uses bit maps may need to compress them but the problem is less serious than with a type setter as only a few letters need be available at one time. The more important consideration is that the letter being worked on can be easily modified, suggesting that if a bit map is to be edited it is best stored in a straightforward manner.

Outlines

As has already been mentioned, it is hard to manipulate letters stored as bit maps and so other, more structured, representations are also used. Even the simple transformations of scaling, stretching and slanting are not easily done with bit map representations. It is much easier to transform an outline and then convert it to a bit map. Unfortunately the process of converting an outline to a bit map (scan conversion) seldom works perfectly. For example, depending on how the outlines register with the grid, vertical stems may have different widths (figure 2.2). If the letter is to be stored as a bit map then these problems can be corrected in the stored version, whereas if the bit map is generated within the type setter there is nothing that can be done. With a high-resolution type setter this is not too much of a problem as the percentage error is very small. The difficulties are most pronounced for the lower resolution devices that are beginning to become widespread in offices. Techniques are being developed to get round such problems, for example the Postscript system produced by Adobe claims to print high-quality bit maps from an outline encoding. The techniques they use are closely guarded commercial secrets and so no more can be said here.

As well as making transformations easier, the use of an outline encoding makes interpolation between shapes easy. This is a very useful technique, for example to generate a letter of an intermediate size from two masters at opposite extremes of the size range rather than using the same outline for all sizes. Interpolation can also be used to generate new designs that are combinations of two different designs. To do any interpolation it is most convenient if the two master letters have the same number of points and corresponding points on the two letters are known. An intermediate letter is then generated by calculating the position of points some constant fraction of the distance between the corresponding points on the two outlines, as in figure 2.3.

The simplest way to store an outline is as a line chain, that is, as a series of points joined by straight lines. Curves are represented by a sufficiently large number of short straight segments to prevent the shape appearing polygonal. Unfortunately, there is still a danger of the letter appearing polygonal at very large sizes unless very large numbers of points are used. It is possible to overcome this problem and reduce the number of points needed by joining the points with curves rather than straight lines. A high order polynomial fitted through a series of points will tend to oscillate and so does not produce a very attractive result. A better solution is to fit a series of lower order polynomials through successive groups of points. Various techniques can

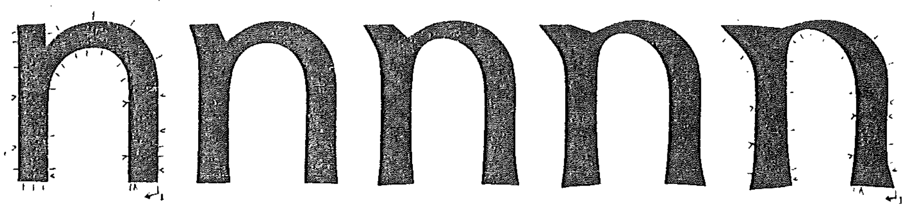


Figure 2.3
Interpolating between a modern sans serif type (left) and a medieval half-uncial bookhand (right). Based on drawings by Kris Holmes in [Bigelow, 1982] .

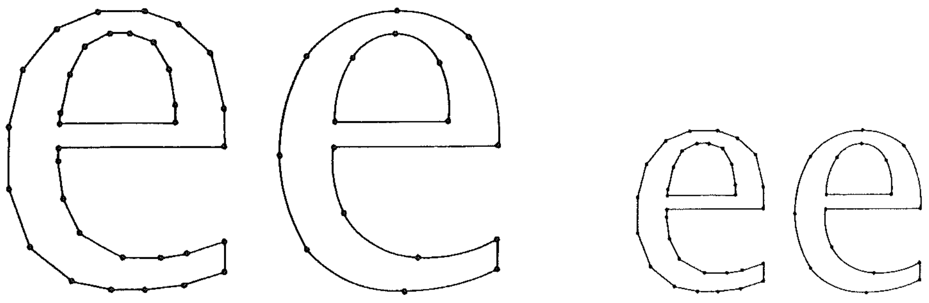


Figure 2.4
The same letter, in two different sizes, in a line chain representation (left in each pair) and spline representation (right).

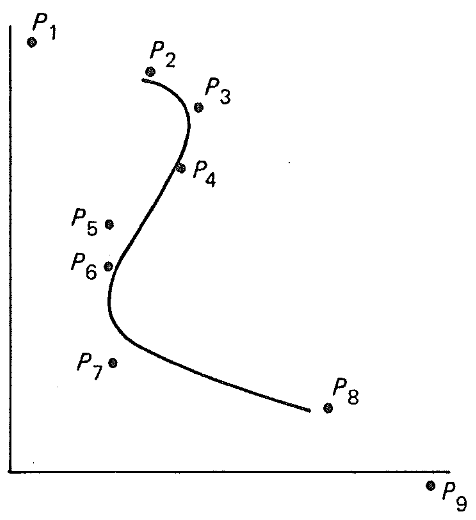


Figure 2.5
A B-spline cubic through nine points.

be employed to ensure continuity of curvature between one segment and the next. These curves, known as splines, can be used to generate attractive curves from a small number of control points. The shapes generated will always appear smoothly curved however much the original shape is enlarged (figure 2.4).

Creating an outline using splines takes some specialised skill as the shape produced by a particular set of control points is not necessarily what would be expected intuitively (figure 2.5). This is not so much of a problem where a finished design is being digitized as a curve need only be fitted once to a static shape. When a new design is being created the use of a spline representation becomes very difficult. Not only is the control of a spline frequently counter-intuitive but the calculations can be so slow that interactive control is not really possible. A variety of splines with different behaviours are available, some of which do not pass through their control points and so are not very suitable for defining letter outlines. Splines which do pass through their control points, known as interpolating splines, have a much more acceptable behaviour. Barsky and Beatty [1983] describe interpolating splines that have control parameters that can be understood physically. These Beta splines with "bias" and "tension" parameters have the potential for intuitive, interactive control but speed is very much a problem. Hobby [1985] describes a different system of interpolating splines with the emphasis on producing aesthetically pleasing curves. Leitch and Smith [1984] specifically tackle the problem of the speed of calculation needed for interactive control of splines. Once rapid interactive control is available it will provide designers with the opportunity of developing an intuitive feel for splines, which can then enhance rather than hinder their work.

Other curves are also used for defining outlines, for example the LIP system described below uses arcs of circles for curved portions. Purdy and McIntosh [1980] describe the use of a digital spiral to represent the outlines of scanned characters. The designer uses different segments of the spiral to fit parts of the outline with different curvatures. References to these curve segments can be stored and then used to reconstruct the outline.

Parametric representations

A very different approach from the systems described above is that of defining letters in some abstract mathematical form. The aim of such abstraction would be to enable certain parameters to be specified and then have the whole alphabet generated automatically. Attempts have been going on for centuries, for example Durer [1535] attempted to describe the roman alphabet in terms of geometric constructions on a grid. The size of the grid was the main adjustable parameter and the sizes, shapes and proportions of the characters were all related to this grid. In fact the alphabet did not prove amenable to complete description and frequently the artist using the constructions was instructed to fill in certain parts by eye.

Recent mathematical developments and the use of computers has now made possible the accurate description of the curves making up letter forms. The outline representations using splines that have already been mentioned are exploiting these developments, but just in terms of representing a single fixed outline. A further possibility with the use of computers is to construct whole alphabets based on a few parameters. When different type faces are examined there is obviously some relationship between all the A's, all the B's and so on. Also, within one type face there is a set of relationships between the different characters, defining the style of the face. This suggests the production of abstract descriptions of the essence of each character which can be converted into a particular instance of the character by providing parameters to the description. A new face could be generated by giving the same parameters to all the character descriptions, thus producing automatically a set of related characters. The first attempt to do this by computer was done by Mergler and Vargo [1968] who produced a very simple parametric fount. The results were not very beautiful but showed that such a technique might be possible. More recently Coueignoux [1975] produced a grammar for roman faces, describing them in terms of parameterised primitives. New roman faces could be generated by providing new sets of parameters. The Metafont language [Knuth, 1979], described below, is a more fully developed attempt to implement these ideas.

A parameterised fount has the potential to be a very compact way of storing information in a type setting machine. For example, a lot of space could be saved if shapes that are common to several letters are only stored once. Also the ability to generate different weights or styles from one set of definitions would save further space. Unfortunately letter shapes are not proving very amenable to such treatment. The relationships between letters of the same style are very subtle and the essence of a style cannot be easily extracted or defined. Descriptions that yield a harmonious series of letters for one set of values of the parameters may well result in total disaster for another set. Figure 2.6 shows a simple example taken from Metafont where different parameters are given to the same definition. With high boldness but low width rather strange results are produced, even though many of the other curves produced by the same definition are acceptable. This example is pushing the parameters to extremes but it does show the problems involved in specifying how different strokes should relate.

For designing, a parameterised fount sounds wonderful—changing one letter could cause all the rest to change, with no need for the tedious process of changing them all by hand. When one alphabet is complete then all of the different weights and sizes could be generated automatically. On the other hand there are big problems with the fact that such founts must be described rather than drawn. The definition rather than the picture is central, and so the definition must be updated to make changes rather than tweaking the picture until it looks right. Having to consciously describe what is to be done tends to get in the way of intuition and may even endanger the whole creative process.

2.2 Existing Systems

The previous section outlined the main approaches to the generation and representation of digital type. Various combinations of these approaches are embodied in the systems described in this section. A discussion of these systems will illustrate their different strengths and weaknesses.

Ikarus

The first practical system using outline coding for letters was Ikarus [Karow et al, 1979], developed by Peter Karow of URW in Hamburg. It is now quite extensively used for digitizing libraries of existing faces but it is not intended for designing new faces from scratch. Large prints of the letters to be digitized are marked up by artists to indicate points to be linked by straight lines, points on curves and tangent points where curves turn into straight lines. These points, along with codes

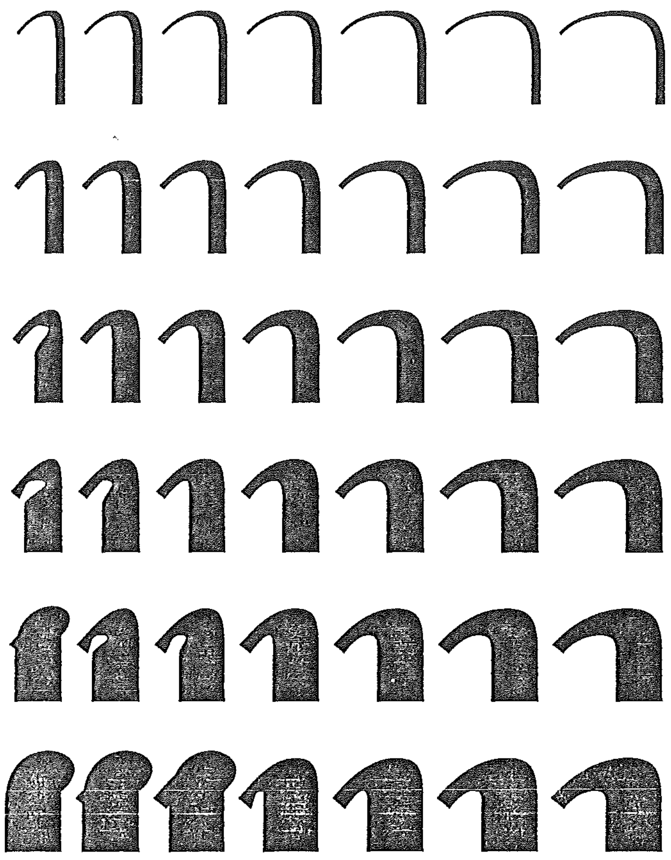


Figure 2.6
A strange effect with a meta-font - the width parameter increases from left to right and the boldness from top to bottom. Note the interaction between high boldness and low width. (example provided by Neenie Billawala, Stanford.)

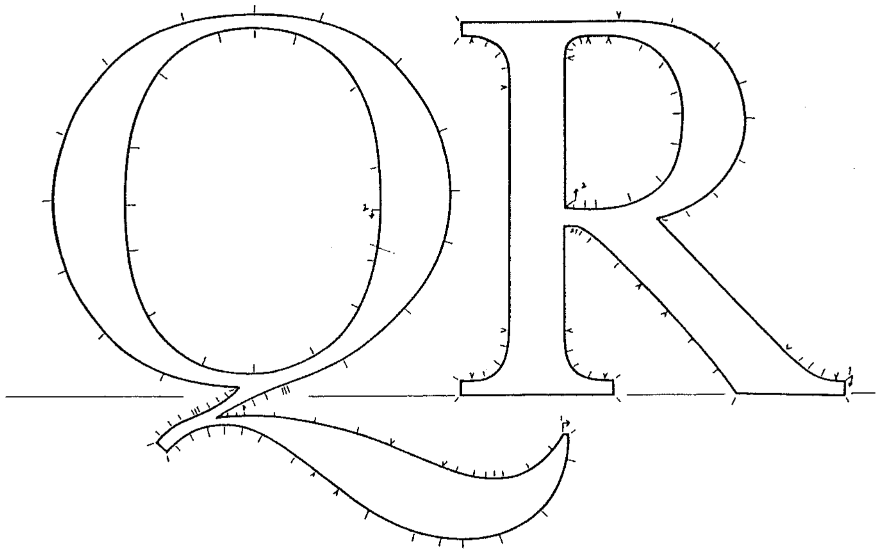


Figure 2.7
A letter marked up for digitizing by the Ikarus system (drawing by Kris Holmes in [Bigelow, 1982]).

indicating their type, are then entered into the computer by hand using a digitizing tablet (figure 2.7). Work is also being done on capturing the outlines automatically from bit maps produced by a scanning device. Further structure can be specified to make explicit the relationships between various letters, for example making all letters use the same set of standard serifs or indicating vertical stems that should have the same width. This makes it possible to change all the serifs together, or to expand or condense the face. Changes such as expanding or condensing require structure over and above simple outlines because the stroke widths do not change in the same way as the spaces between them, making simple stretching inadequate.

Proofs are produced of newly digitized outlines and these can be marked up for correction. It is not an interactive system and points are repositioned by typing coordinates rather than positioning them visually. This makes the production of the digital type faces slower and more laborious than it need be. Once a correct outline has been produced various transformations and interpolations are available. Such facilities are intended for minor changes such as producing different sizes or weights of the same face but it is also possible to generate new faces by interpolating between two different designs.

The completed outlines are device independent and can be used for generating founts for various sorts of type setters. This might mean simply converting the outline to the the format used by a particular machine, if it uses outlines directly. The outlines might also be scan converted to bit map form for devices that cannot do the conversion themselves. The structure incorporated into the letter definitions allows more accurate scan conversion to be done, for example identical serifs can be ensured and stems can be given the same widths automatically. The resulting bit maps can also be edited by hand if that is needed.

LIP

A more recent system using outline encoding is the Letter Input Processor (LIP) [Flowers, 1984] developed by Camex and Bitstream. Like Ikarus it defines letter outlines in terms of curves and line segments but the major design consideration from the start was for the system to be highly interactive and visual. As was touched upon in the previous chapter, designing is an experimental and interactive process and so any serious design system should support this way of working. In the case of type face design the results must be assessed visually and so visual feedback is needed. The user of LIP communicates with the system by way of a menu of commands on a digitizing tablet which can be selected by moving a puck over them. A shape is initially entered into the system by digitizing an existing piece of artwork, using the puck to input points along the outline. The letter being worked on is displayed on a screen and the user can zoom in on particular areas for detailed work or zoom out to see the whole letter. Objects on the screen can be selected by moving the cursor over them and pressing a button. The selected object can then be dragged around the screen by moving the puck over the tablet. Parts of characters, such as serifs, can be isolated and then copied, rotated, scaled or mirrored. These parts can be stored and then incorporated into other letters as an aid to consistency throughout the face. An electronic ruler is provided to ensure that the relative sizes of objects is correct. A limited programming ability is available in that sequences of actions can be stored and recalled later. This enables the designer himself to extend the system.

The result of the design process is a set of letter outlines in the form of points joined by straight lines or arcs of circles. As with Ikarus, the outlines are device independent and can be used in various ways to generate founts for different type setters. Scan conversion and editing of the resulting bit maps does not form part of LIP, but is done on other machines.

Although LIP is much closer to a designer's natural method of working than Ikarus, it still does not provide for the origination of new type faces entirely within the digital medium. Its facilities make for rapid and accurate digitization of existing faces but do not provide the freedom needed for experimenting with new ideas. Shapes must be presented to the system as a well thought out set of line and curve sections. There is no facility for freehand drawing.

Elf

Elf [Kindersley and Wiseman, 1979] was an interactive system that addressed directly the problem of producing and experimenting with new ideas. It ran on a vector display which is now obsolete and so it is no longer being developed or used. It made use of a light pen for interaction, with commands being issued either from the keyboard or by pointing at light buttons on the screen. A letter shape was generated by drawing directly on the screen, leaving a trail of points. These points defined the path of a pen with a user-defined width and angle and the filled letter was generated by building trapesia on the points along the path. Points could be selected and repositioned and the pen width and angle at any point could be changed. Several letters could be viewed together to assess the results, with the spacing being calculated automatically. Facilities were provided for moving parts from one letter to another and for interpolating between designs. Transformations such as rotation and scaling could be carried out.

The most important feature of Elf was the provision of freehand drawing and interactive editing. This made it possible to sketch very rough ideas and then work on them to produce the finished letter forms. The definition of letters as pen strokes allowed initial ideas to be roughed out very rapidly but became a hindrance at later stages. Some type faces are calligraphic in form but very many are not and so a lot of fiddling was needed in Elf to achieve many commonplace effects. Figure 2.8 shows the structure of some letters in Elf format.

Metafont

Metafont provides for the description of letters by mathematical formulae using a set of common parameters. By changing the parameters new faces of related letters can be produced. The original form of Metafont, like Elf, defined the letter in terms of the path of a pen of specified dimensions and so encountered the problems already mentioned. A new version of Metafont is being produced which allows the definition of the outline of the letter, which can then be filled.

Shapes in Metafont are defined in terms of a small number of points which are then linked by curves or straight lines. The line produced can be varied by specifying the directions at the end points as well as changing the pen width and angle. Once the definition has been written the shape can be displayed. Any changes needed are made to the written definition and the result can then be redisplayed. Figure 2.9 shows a Metafont definition and the resulting letter.

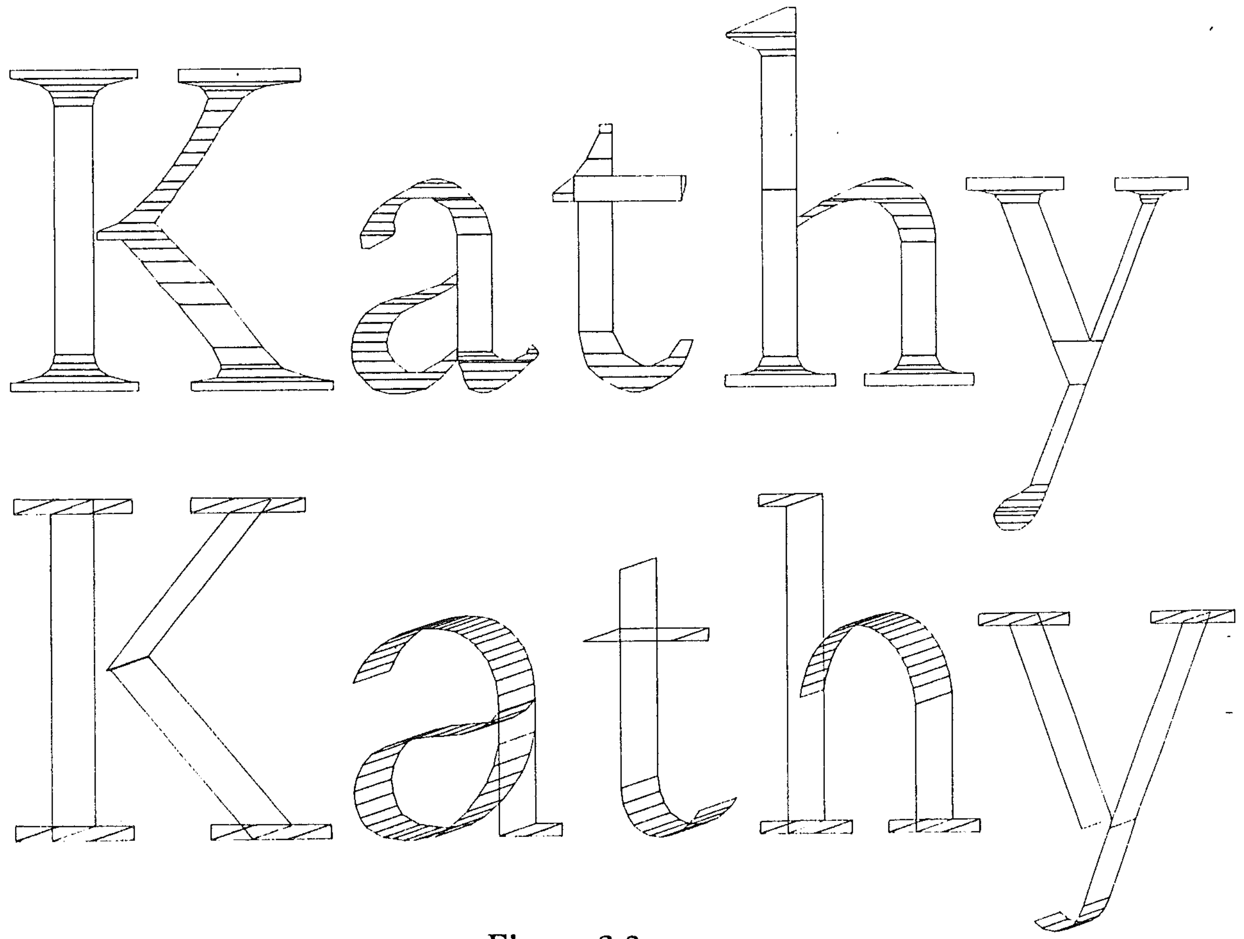


Figure 2.8
Structure of some letters designed using Elf.

```
"Cyrillic letter Eprime";
call charbegin('037+3codeoffset,11,0,0,phh,0,rbowl);
cpen; top3y2=.3hh;
if bot3y2<.1hh: new y2; bot3y2=.1hh;
fi;
lft3x2=round .75u; w3 draw 2;                  % lower bulb
hpen; lft0x3=round 1.25u; y3=good6 .8hh; lft0x4=lft3x2; y4=y2;
x20=x3; rt4x3=rt0x1; y1=y3; top0y20=hh;
x5=.5[x3,x7]-u; x6=.5[x4,x7]; top0y5=hh+o; bot0y6=-o;
rt1x7=round(r-u); y7=y6;
if ucs$0: w0 ddraw 1..20, 3..20;               % upper serif
        rpen#; w4 draw 3{0,1}..5{1,0};         % erase spurious part
fi;
hpen; w0 draw 3{0,1}..5{1,0};                  % shoulder
call 'a darc(5,7,w5);                          % bowl
y9=y10=.52hh; x9=2u; x10=lft5x7;
w0 draw 9..10;                                 % bar
w0 draw 6{-1,0}..4{0,1}.                       % tail
```

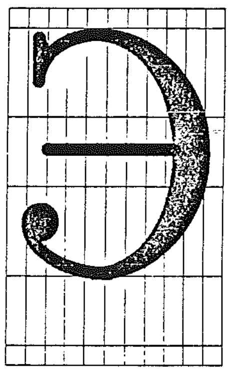


Figure 2.9
A Metafont program and the resulting character shape.

The Metafont definitions are device-independent and incorporate features that are intended to adjust the designs for devices of very different resolutions. The bit maps are produced directly from the definitions and it is assumed that there will be no need for any hand editing. The shapes are drawn using specially digitized pens that are claimed to remove the various problems that usually arise in bit map generation.

The main weakness of Metafont is that the letters are described in an abstract form rather than being interactively constructed. The designer is unable to exploit the hand-eye coordination that plays such an important role in any graphic designing. Obviously the use of new tools will involve the learning of new skills but it can be argued that the skills of abstraction and programming are inappropriate for a visual and intuitive task.

Another problem with Metafont is that the number of parameters required to produce adequate results keeps increasing. The nature of letter shapes is still poorly understood and maybe will never be described in a convenient and manageable mathematical form. The way that the parameters interact is hard to predict and so the production of new faces with Metafont can often be a case of accident rather than design.

Other systems

A number of personal workstations with bit mapped displays provide simple fount editors. These generally provide a grid on which the design is produced by filling it in, dot by dot. Such systems are not very suitable for serious fount production because the process is so tedious. A system running on the Lilith workstation [Kohen, 1985] provides a more structured environment for fount production. The characters are entered as outlines which are then edited as necessary. The outlines can be scan converted to rasters at various sizes and resolutions and the resulting bit maps can be edited. The system also provides guidelines for such things as ascenders, descenders and x-height. The resulting founts are intended for medium resolution devices in the range of 100 to 500 pixels per inch.

2.3 Proposals for a new type face design system

We have looked at various approaches to the task of generating digital type and now it is time to consider what the way forward may be. The aim is to produce a system that can support the whole process of type design from the earliest ideas through to the finished product.

A recurring theme through these first two chapters has been the importance of intuition and practical experience. Ultimately a good design works because it looks right, not because it fulfils some theoretical criteria. Designers should be free to follow the judgement of their hands and eyes rather than having to stand back and think abstractly about what they are doing. On the other hand we have seen the advantages that structure and abstraction can bring. It would be sensible to try and exploit such things as the similarities between letters in the same face while at the same time not imposing limitations on the designer.

This section introduces the main ideas and design decisions underlying Imp, the new system. The actual structure of Imp is described in more detail in chapter 5, after the computing environment it is built on has been presented.

Abstractions and interaction

Any type design system that uses a computer must have an abstract representation of the letter shapes. The decision that must be made is what sort of abstraction is most suitable and how it will be constructed and modified. Having given the designer's visual and manual skills a central place we must provide a system that works visually and interactively. All of the designer's actions must evoke an immediate response so that unconscious links between actions and responses can be built in his mind. Provided that this immediacy and directness is available it will probably not matter whether the tools available are exact analogues of traditional tools. Indeed, we may be able to make much better tools.

Complex abstractions bring a lot of power but they can also be very constraining. For example, an abstraction that defines everything in terms of pen strokes makes it hard to produce a non-calligraphic shape. When everything must be explicitly defined it becomes difficult to simply make something look right. We can go to the other extreme of treating everything as plain bit maps. By allowing the designer to paint or erase bits in the bit map any shape can be produced. This gives great freedom but makes the provision of any automated transformations difficult. It is also too specific to one resolution and size of type and so makes it hard to re-use design work.

It seems that the most suitable representation of letter shapes is some form of outline encoding. The designer can just draw an outline freehand without having to think about it. This outline can be edited by interacting with it directly on a display screen or various automatic procedures could be applied to it. If the designer is given the freedom to either work directly or by way of procedures then he need never be unnecessarily constrained. When the result produced automatically is not quite right it can then be edited directly until it looks acceptable. This new design system will not store the outlines in the form of splines. Firstly, storage space is not a problem and so curves can be represented by as many short line segments as are needed. Secondly, the difficulty of handling splines interactively is a very powerful argument against using them in the stored outline. Work currently in progress may well change this but for the moment line chains are the most suitable representation. On the other hand, splines are available for generating these line chains if the designer wishes.

Shared parts such as serifs can be designed as part of one letter and then copied to other letters. By providing measuring gauges and grids it is possible to ensure that stems are the correct widths. The disadvantage of this method is that the system has no knowledge of the relationships between the different letters. Hence, if the serif of one letter is changed none of the others that use this serif will change. The obvious step on from being able to copy parts from one letter to another is to be able instead to set up relationships indicating a shared part or equal sizes. Some form of graphical language is needed for specifying such relationships and setting up structured definitions.

Although the outlines are the main objects handled by the system, a bit map editor is also needed. This is to allow for the fact that scan conversion seldom works perfectly and so there is a need for some editing at this stage. It is assumed that most work will be done on master designs in the form of outlines and that the bit map editor will only be used for tidying up completed designs.

The structure of the task

The system must take account of the structure of the design task if it is to support it successfully. A designer will generally work in an unstructured manner, moving from one thing to another at frequent intervals, perhaps with several different tasks going on in parallel. This suggests that the system must be able to support a number of simultaneous tasks, with no particular structure or ordering imposed on the user. A computer screen can be divided up into a number of areas, each one dealing with a different task. The user can switch his attention from one area to another at any time to move to another task. The areas on the screen can be made a bit like sheets of paper that can be moved around, placed on top of one another or filed away for later use.

Several different aspects of the type design task were picked out at the end of the previous chapter and these aspects can be distributed across the different areas on the screen. One area can be available for the design of letter shapes including setting up grids, playing with rough ideas and working on fine details. Another can provide space for placing letters side by side to set up and examine spacing. By having both areas visible at one time the designer can be working on details of one letter whilst still keeping in mind its relationship to others. The outlines can be converted to bit maps and another area can be used to display a page of text using these letters, providing an even broader view of the role of the letter being worked on. Other areas can be provided to store letters not currently being worked on. By organising the system in this way the designer is able to keep track of the effects of any design decision at any level.

Another feature of designing that has a big influence on the way the system is implemented is the balance between interaction and automation. If a lot of lengthy calculations were being done to carry out some task automatically then it would be useful if the designer could get on with something else in the meantime. On the other hand, if the emphasis is always on interactive modification then there is no need for more than one thing to be happening at one time as the designer will only be working actively on one thing on the screen. The parallelism mentioned above is within the designer's head rather than within the program itself. He needs to be kept aware of the state of all his current tasks but will work on only one at a time. If it can be done this way then the underlying computer system can be greatly simplified at the expense of the user occasionally having to wait for a calculation to finish.

The user interface

The proposed system is highly interactive and hence the user interface is vitally important. For many designers the computer will be an alien machine and so they must be made to feel safe and confident. When designs are stored in some abstract internal form it is easy to get worried that they may be lost. After all, they cannot be seen or touched in the way that designs on paper can. Hence the system must make things visible as far as is possible. There are a number of commonsense considerations that can make a system feel safe and controllable, such as making it very difficult to destroy things accidentally. If the user can undo any action then he is much more likely to experiment with new and unfamiliar features. Playing around with the system will teach him much more than theoretical explanations. A consistent style of interaction and the use of menus or prompts will help the user to learn about the system and prevent confusion.

The user interface makes the abstractions that represent the letter forms in the computer visible and accessible. As he uses the system, the user builds up a conceptual model that enables him to understand how the underlying programs work [Moran, 1981]. This understanding can then provide him with ideas about new ways of using the system. An appropriate user interface will allow someone to work effectively and imaginatively.

Because type face design is a visual task a good graphics system is needed to support the user interface. It must provide for rapid interaction and feedback and also provide a good interface to the programs that make up Imp. The following two chapters present graphical computer environments in general and the environment implemented to support Imp in particular.

3. Interactive computing environments

In the previous chapter the design considerations for Imp, an interactive system for type face design were introduced. Any such interactive system requires special tools for constructing and running it. This chapter gives an overview of various types of interactive environment described in the literature. It then goes on to discuss the hardware and software environment available to me in Cambridge. The next chapter presents the environment that I have implemented to support Imp.

The tools currently available for managing interactive environments can usefully be divided into three groups, although the divisions are by no means clear or rigid. The first group is interactive graphics systems which provide facilities for handling graphical input and output. These systems are usually built around a stored model of some object with which the user can interact. Facilities are provided for displaying and modifying the model in various ways. Secondly we have User Interface Management Systems (UIMS) which have grown up both in graphical and in non-graphical areas of computer science. The main emphasis in these systems is on specification and management of the interactive dialogue. Although there are a number of non-graphical UIMS the discussion here is restricted to the graphical systems as these illustrate the main ideas and are of more relevance to the rest of the discussion. Finally the concept of "windows" is presented, a way of working that exploits graphical facilities for general computing environments. The systems that make use of windows usually try to provide a fully integrated programming environment with consistent techniques being applied across all applications. Many of the applications are not strictly graphical but the methods employed are of use in any application area.

3.1 Interactive graphics systems

The physical environment

Any hardware for interactive graphics must provide a display surface that can be rapidly updated along with tools that can be used for selecting objects on display and for specifying coordinates. Most displays currently in use are based round cathode ray tubes (CRTs), using either vector or raster scan techniques. The application program, usually by way of a library of graphics routines, furnishes a description of the image to be displayed. The display processor controlling the CRT interprets this description and provides suitable control signals to the beam of the CRT. In a vector display the image is constructed by drawing the line segments that make up the picture. The stored description of the image is this set of line segments, possibly with higher level structures allowed for grouping lines into objects and so on. In a raster scan display the beam draws the image in successive horizontal sweeps across the screen. The image is stored as an array of intensity values for each spot (pixel) on the screen. Raster scan displays have become very cheap and are now more widely used than vector displays.

The user of an interactive graphics system must have tools that enable him to select and manipulate the objects being displayed. In general the input tools used yield numeric values which can then be interpreted as the program requires. One device, the light pen, actually interacts directly with the image. The light pen has a light detector in its tip which signals when the beam of the CRT crosses its field of view. When used with a vector display with a structured image description it is possible to determine directly what part of the image the pen is pointing at. This type of input is known as a "pick" and is found in many graphics input libraries. The problem with the light pen is that it is tiring to use as it must be held up to the screen. It also obscures the screen contents beneath it. As it is not so useful with raster scan displays anyway, it is being replaced by other tools. Any device that can be made to yield varying numeric values can be used as a graphics input device. These include the graphics tablet, which senses the position of a special stylus; the joystick, which senses the amount and direction of deflection from the vertical; and the trackerball, which detects the amount and direction of rotation. Another device that has recently become widely used is the mouse. It is essentially a small box that can be held in one hand and moved over a surface. This movement is detected, either by mechanical or optical means, and converted to numerical data. It has a number of buttons on top, usually two or three, which can be used to signal events. Different devices are suitable for different applications and different users and ideally one workstation should provide a choice. Further details of these devices can be found in Foley and Van Dam [1982].

Software

In any interactive graphical application we have a user observing an image on a screen and using various input tools to manipulate this image. The graphics program must collect and interpret the user's input and update the image accordingly. A classic example of such a program is Sketchpad [Sutherland, 1963] which allowed the user to construct pictures using a light pen and function buttons to describe shapes of and relationships between displayed objects. The image on the screen was a visual representation of a model stored in the computer's memory that recorded these shapes and relationships. By interacting with the image the user was updating this stored model. The maintenance of relationships between objects in the model meant that the image on the screen behaved as more than a simple sketch. For example, if a point was moved then all lines that terminated at that point would automatically be repositioned. This pattern of the image being the visual representation of a highly structured stored model has provided the basis for many subsequent graphics systems.

Sketchpad was a specially written, "once off" piece of software but from the earliest days there have been attempts to simplify the writing of such programs by providing libraries of routines that are of general use. This can greatly reduce the amount of code that must be written for a new application. On the output side this usually means providing routines for drawing and transforming simple shapes and perhaps for identifying and grouping objects. The particular facilities provided do not have a great influence on the nature of the interactive programs that can be written although they do influence the amount of work that must be done in the application program. The handling of graphical input is a much more difficult area with a wide variety of very different tools with different dynamics. There are a number of different models of input tools and of the structure of interactive programs. The model chosen has a great influence on the way an interactive application is written and so the correct decision at this level is very important.

An early model for the handling of input tools was the Reaction Handler [Newman, 1968] which used a finite state machine to model the interactive application. Such a machine is always in one of a finite set of states and it shifts between states when particular events happen, each state having a fixed set of events that it recognises. In the context of handling interactive input the events are the user's actions and each state has associated with it a response that occurs when that state is entered. A simple example of a dialogue is shown in figure 3.1. The Reaction Handler provided a language for defining the states of such a machine and the movements between them. This definition language just specified the flow of the dialogue and was quite separate from the language used for writing the procedures that were the responses of the various states. The separation of dialogue specification from the actual procedures of the application foreshadows the recent developments in UIMS which provide special languages for specifying dialogues. This allows the two different functions of controlling the dialogue and doing application-specific calculations to be written in languages appropriate to the nature of the tasks.

Another feature of the Reaction Handler was the attempt to keep the programs device independent. Input was divided up into categories, for example, "pen movement" may come from the movement of a light pen or from the typing of coordinates at a keyboard. Again this foreshadows later developments in the use of "virtual tools". In particular, attempts at defining graphics input standards have concentrated on the definition of an appropriate set of virtual tools that will allow applications to be moved between different sets of hardware.

The idea of device independence was first formalized by Wallace [1976] and Wiseman and Robinson [1977], who both specified sets of virtual tools sufficient for any task. The program is implemented in terms of these virtual devices and can then be moved from one environment to another by binding different physical devices to the virtual devices. Wiseman and Robinson proposed a keyboard yielding a string of characters, a locator yielding coordinates and a picking device yielding identifiers of picture components. One physical tool can feed input to more than one device, for example a light pen could provide coordinates by moving a tracking symbol, strings of characters by pointing it at a light button and picture components by pointing at them. Also, one virtual device can receive input from more than one physical device, for example the virtual keyboard could receive characters from both the light pen and the physical keyboard. Wallace also includes a valuator, yielding a single numeric value, and a button device yielding a button identifier. He suggests that these cannot be implemented in terms of the other devices, but it can be argued that a valuator is a one-dimensional locator and that buttons yield characters for the keyboard stream. Others, for example Rosenthal [1981], suggest that all input should be unified into a single type. The general idea of virtual devices is now firmly established although argument continues on the nature of the primitive devices needed. GKS (see below), a draft standard, uses five devices similar to those proposed by Wallace and in addition has a stroke device that yields sequences of coordinates. The nature of the primitive devices settled for may well reflect the capabilities of the physical devices that people are used to rather than any genuine abstract device independence.

Newman's Reaction Handler, discussed above, used a single finite state machine to model the interactive dialogue. At each stage a specific set of inputs are valid and the user is led, step by step, through a highly structured dialogue. This approach is

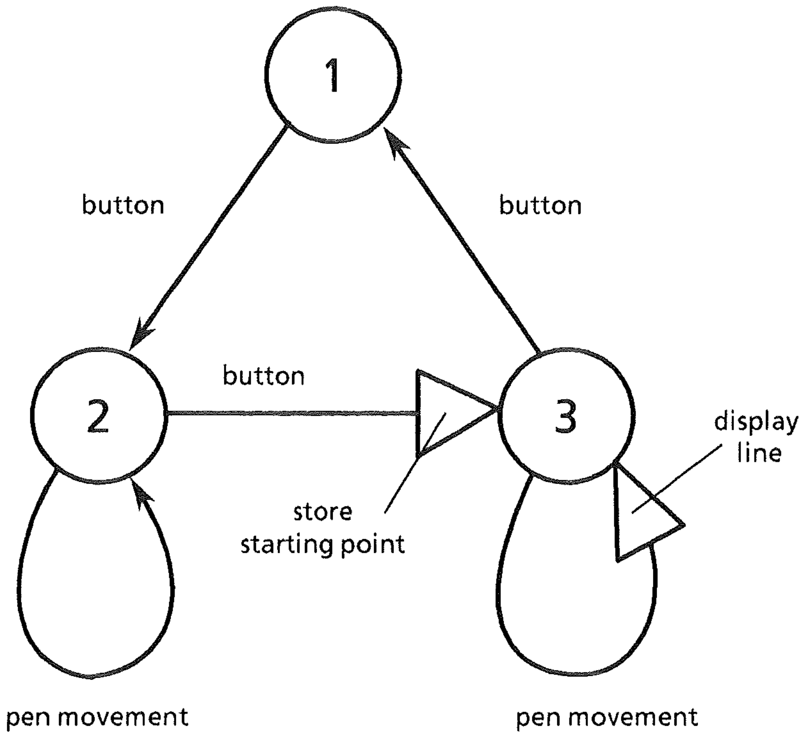
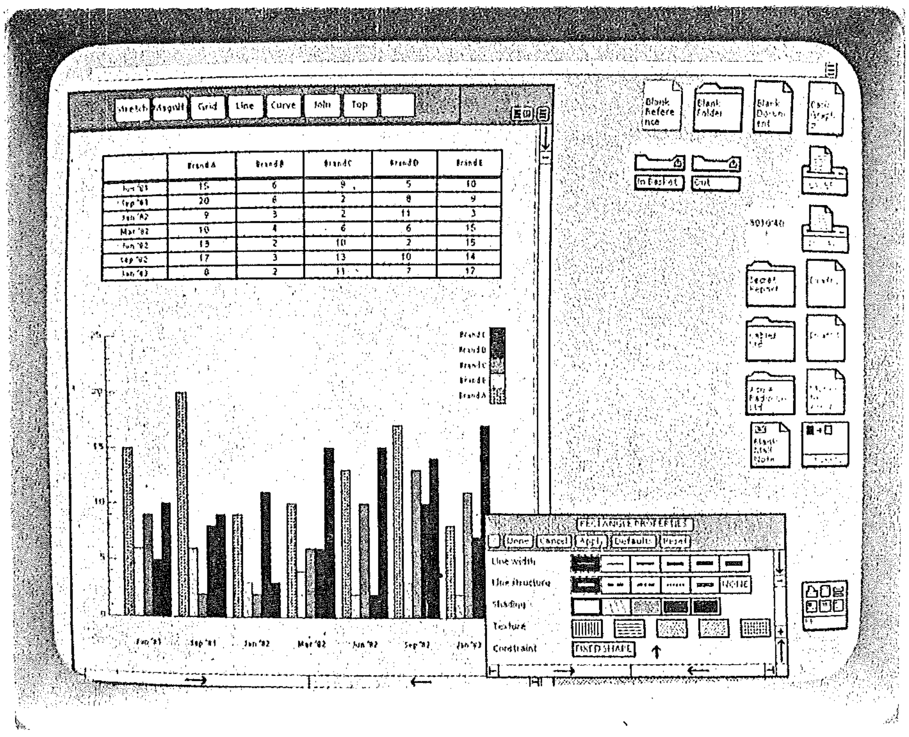


Figure 3.1
A state-diagram representing rubber-band line drawing.
(copied from [Newman, 1968])



	Brand A	Brand B	Brand C	Brand D	Brand E
Jun '81	15	6	9	5	10
Sep '81	20	6	2	8	9
Jan '82	9	3	2	11	3
Mar '82	10	4	6	6	15
Jun '82	13	2	10	2	15
Sep '82	17	3	13	10	14
Jan '83	8	2	11	7	17

Figure 3.2
A view of the screen during a typical session on the Star workstation.

actually not very well suited to the way people really work. Someone may be working on several different things at once, moving between them at intervals. Part way through one task he may decide that he does not want to complete it or that he wants to do something else for a while. The approach taken in the Reaction Handler, although widely used in subsequent systems, makes such unstructured working very hard to handle. The user is unable to change course or discard a partly completed dialogue unless such an action has been explicitly provided for. The structure of the dialogue would soon become very complicated if all possible changes of direction were catered for. A more appropriate approach is to provide a multi-tasking environment where several tasks can be active at any time and where all the input tools are always available for use. This is the approach taken by Wiseman and Robinson, where tasks handling each of the input tools are running continuously so that the system never ignores the user's actions. For example, a separate task can handle requests for cancellation of activities. The code for each activity does not then have to include frequent tests for the user cancelling it. Instead, the cancellation task waits for this event and can kill the activity from outside. This approach has obvious parallels with the use of multi-tasking to improve responsiveness in operating systems. A more recent paper [Beach et al, 1982] also describes the use of multiple processes within a single program to aid in structuring and controlling an interactive application. Although similar programs can be written using a single threaded approach they are likely to become very tangled and hard to understand.

Standardisation

The foregoing discussion has illustrated some different approaches to the handling of input devices but obviously there is pressure for standardisation. This has culminated in proposals for various standards including the Siggraph Core proposals [GSPC, 1977] and later GKS [GKS, 1984] which cover both graphics input and output. I will confine this discussion to GKS as it has largely superceded the Core proposals.

GKS provides a library of subroutines for input and output which can be called from the application program. Input is provided by way of six virtual devices: string, locator, pick, valuator, button and stroke. These virtual devices can then be mapped onto whatever physical devices are available. The application can sample these devices, wait for a user action on a particular device or can have significant events stored in an event queue. The most appropriate form of input handling can be selected by the application programmer. GKS handles all echos and prompts for the input devices itself rather than leaving it to the application program. This has the advantage of relieving the application programmer of this task but on the other hand it constrains the variety of feedback that the application can provide. For example, a valuator could be shown as a straight forward dial or digital display or it could be shown in some highly esoteric application dependent form. Although GKS provides a variety of standard echos and prompts it cannot hope to cover all possible requirements. There does not appear to be room for the application programmer to provide his own feedback in place of what the library provides. Another problem area is the control of the physical devices and relating them to the input mode selected by the application. It is very unclear from the GKS specification how triggers for various devices are set up—for example, is it possible for a change in coordinates to trigger a locator device or must the user depress some button? What does it mean to sample a choice device where a choice is a discrete event? There also seems to be no way to control low level behaviour of devices, such as the sampling rate of a coordinate device, which can have a significant effect at the application level.

GKS is implemented as a subroutine library, leaving all flow of control in the hands of the application program. If the application is to be efficient and responsive on a variety of equipment it must behave differently when different physical devices are present. This means that either the application must provide many options selected on the basis of the devices actually available or else it will settle for handling some devices inappropriately. An alternative approach to device independent input is presented by Rosenthal [1981] which effectively turns the idea of graphics subroutine libraries on its head. He suggests a scheme where all flow of control is handled in the graphics library and the application is called as subroutines, rather than vice-versa. The library can contain all the code needed to handle each type of physical device in an appropriate manner so leaving the application truly device independent. This approach is also taken by most of the UIMS discussed below.

Device independent output is more easily handled and the details are not really relevant to this discussion on interactive dialogues. The main problem area is the inability of GKS to deal with simultaneous output from multiple processes, whether within the same machine or distributed over a network. GKS stores a current state for such things as pen colour and line style that applies to any drawing commands that are issued. Unpredictable results will occur if changes of state and drawing commands from two different processes become interleaved. This sort of problem has arisen because GKS is based on well established techniques and equipment and did not look to a future where multi-processor distributed systems might become the norm. Arnold [1981] and Kilgour [1981] provide further discussion of these problems in relation to future directions for graphics standards.

3.2 User Interface Management Systems

Recently the phrase "user interface management system" (UIMS) has been used of a variety of systems for interactive work, graphical and otherwise. The philosophy behind these systems is to reduce the work required to construct a new interactive application and to provide some sort of consistency in the interfaces to different programs. It is hard to predict in advance how well a particular interface is going to work and so the necessary experimentation will be encouraged if it is quick and easy to put together a number of different schemes. There is a continuum between the graphics systems already discussed and the graphical UIMSs described in this section. Perhaps the most useful distinction to make is that the graphics systems described above tend not to provide tools specifically for describing the structure of interactive dialogues whereas a UIMS is geared especially for that task. It should also be pointed out that not all UIMS provide graphical facilities although I will limit discussion here to graphical systems.

A UIMS provides facilities for specifying the flow of an interactive dialogue and for relating the dialogue to the actions of the application program. A run-time system forms part of the UIMS and is responsible for interpreting or running the dialogue specification and calling the application routines. This run-time system handles input and output and usually provides similar facilities to the interactive graphics systems described in the previous section. Usually the language for specifying the dialogue is quite separate from the language used for writing the application program. This separation is advantageous because a language most suitable for describing dialogues may be totally unsuitable for the sorts of functions that are required in the application. A UIMS that provides total freedom in the choice of application language is particularly valuable although there are many problems involved in this approach. Currently the interfaces between dialogue specifications, application programs and the run-time system are not well enough defined or understood to enable this complete separation to be maintained.

One widely used way of dealing with interactive dialogues is in terms of a grammar specifying the allowable sequences of inputs. An obvious forerunner of this approach was Newman's reaction handler, described above, which used a FSM description. The dialogue consisted of a number of states, with transitions between the states being controlled by the user's input. Each state had an associated procedure that was called on entry to the state. The transitions of such an FSM can be described by a formal grammar. If an interactive dialogue can be reduced to such a grammar then it is possible to produce a FSM implementation automatically. In fact more complex grammars can also be handled in a similar way, using other types of automata. A recent UIMS following this line is SYNGRAPH [Olsen and Dempsey, 1983] which takes as input a specification of the grammar for the dialogue. It generates the menus, prompts and echos needed and also deals with device management, error handling and backtracking. Closely related to this approach are those systems that use menu trees to define the interaction, for example Tiger [Kasik, 1982]. Tiger allows the application programmer to write routines in any language and then specify how the routines are to be assembled into menus and what arguments must be collected before a given routine is called. Menulay [Buxton et al, 1983] is also used for creating menu-based systems from a collection of previously written routines. An important feature of Menulay is its use of an interactive, graphical system for creating the dialogue specifications. Hence the dialogue designer is working in the same environment as the ultimate user of the system. Flair [Wong and Reid, 1982] is similar in its use of interactive graphics for specifying dialogues, but it would appear not to provide the ability to import application routines written in other languages.

A problem with this approach is the large number of states needed to define a realistic dialogue. It is easy for the user to either get completely lost or to want to move temporarily to another part of the menu tree. This latter problem is sometimes addressed by the use of independent sub-trees that can be called from a variety of places. When the user is finished with the dialogue defined by the sub-tree, control then returns to the menu that the sub-tree was called from. The system described by Apperley and Spence [1983] tackles the problem of getting lost by providing a continuous display of the route taken so far and allows backtracking by pointing at earlier parts of the route. Default routes are remembered to allow the user to move faster through the menus.

A completely different approach is to describe the dialogue in terms of objects ("event handlers") that the user interacts with. An example of such a system is described by Strubbe [1983] where the application program constructs a display tree describing what is to appear on the screen and in what relationship. The nodes of the tree have routines attached to them which are called whenever a user action occurs with the cursor within the node's designated area on the screen. It is then up to the routine to decide what action to take as a result of a particular input event. The run-time system can support a number of independent display trees associated with different applications running in parallel. This system is similar in philosophy to the windowing systems described below. Another interesting example is Anson's device model of interaction [Anson, 1982] which provides a notation for describing devices in terms of the values and events seen outside. These devices can be interconnected to produce composite devices with complex behaviours. This model provides one language for specifying the whole system down to the lowest level device handlers.

Describing a dialogue in terms of a formal grammar encourages a very structured and constrained environment. The system is in control and the only actions available to the user at a particular stage are those specified in the grammar. As has already been mentioned, it is quite possible that someone will want to work on several tasks in parallel. Various mechanisms can be used to allow diversions from the main thread of the dialogue, but the underlying idea is of a single dialogue being followed through from beginning to end. Using event handlers to construct an interactive system supports a multi-threaded approach in a much more direct manner. The overall view of the system is of a series of independent dialogues proceeding in parallel, rather than a single thread with diversions. The difference between the two approaches is more in terms of the style of dialogue that results rather than in any actual difference in what can ultimately be done with the resulting system. For a highly interactive system this difference is very important as the style of dialogue has a great effect on how efficiently and comfortably a person is able to work.

3.3 Windows, icons and mice

This section describes a paradigm that has developed outside mainstream computer graphics. It is epitomized by a personal computer with a black and white bit-mapped display and a mouse and keyboard for input. Typically the user has several programs running, each using a different region of the screen for input and output. These regions, known as "windows", can overlap one another and can be rearranged on the screen as the user wishes. The programs running in each window usually have no control over where their window appears on the screen. This style of working has recently become very familiar in the Xerox Star workstation [Smith et al, 1982a] and the Apple Lisa and Macintosh personal computers but in fact has a much longer history.

This approach grew out of work done at Stanford Research Institute (SRI) during the sixties on using computer systems to extend people's intellectual abilities. Engelbart and English [1968] describe the workstations developed at SRI to support software development and documentation. Considerable research was done to investigate the suitability of various input tools for text-based applications. As well as a keyboard, the resulting workstation had a five key handset and a mouse. The handset allowed one-handed typing, freeing the other hand to manipulate the mouse for selecting and pointing operations. The workstations were used for browsing and manipulating structured text or programs. The screen could be split, with the top half being used to display certain selected portions of text and the bottom half remaining available for further text manipulation.

Displays making use of multiple overlapping regions originated in the FLEX system [Kay, 1969] and the "Dynabook" [Kay and Goldberg, 1977]. These systems were envisaged for extending and encouraging people's creative abilities by providing a versatile and dynamic new medium applicable to a whole range of tasks. Availability to all sorts of people, especially children, was an important emphasis and so the user interface became a major area of research. In particular an "object oriented" approach to programming was developed, embodied in the Smalltalk language [Ingalls, 1981; Tesler, 1981]. The system is made up from objects with various behaviours which interact with one another and with the user. All objects in the system have some sort of visible representation so that nothing is hidden away. An important feature is its use of multiple windows on the screen displaying information about different objects or contexts. The user can move from one context to another by moving the cursor into another window, while all contexts remain active.

Smalltalk has been followed by a variety of different window-based environments, mainly aimed at providing rich and flexible programming environments. Some of their important features and the issues that they raise are discussed below. Before moving onto this general discussion of windowing systems the Xerox Star workstation is described in some detail as it is a useful illustration of the windows paradigm. It is of particular interest here as it implements a non-programming application and so provides some indication of approaches that might be successful in non-programming graphical applications. This is followed by a brief discussion of the underlying hardware, to the extent to which it influences the design of windowing systems. The scene is then set for the discussion of more general issues in windowing systems that are of relevance to graphical design applications.

The Star Workstation

The Star workstation is a commercial product that arose out of the work on Smalltalk and windowing systems. It was designed for secretarial tasks and a considerable amount of time was spent on choosing the conceptual model to underlie the user interface before any programs were written. An object oriented approach is used where everything in the system has a visible representation. Communication is done by way of the mouse, with the keyboard available for typing text if the user wishes.

As the workstation was intended for use by secretaries, a desk top model was chosen for the user interface. The screen displays the user's "desk top" with various objects such as files, documents, in- and out-trays upon it. The objects are displayed as "icons", which are small visual representations of the objects. The user interacts with objects by pointing at them with a cursor controlled by a mouse. All user actions have some immediate visual effect to help reinforce the reality of the model being presented.

The system attempts to build on the user's intuitions about the world so, for example, mail is sent to someone by labelling the document with the recipient's name and then placing it on the out tray icon. A document is printed by moving it to the printer icon. Hence, the system is operated by actually manipulating objects rather than issuing abstract commands. It is possible to move from one activity to another by shifting the cursor from one object to another, without any need to terminate the first activity. The user may wish to search through a set of files for information in the middle of editing a piece of text. In many systems this can only be done by closing down the first activity before going on to the second. On the Star workstation, as on a real physical desk the document being edited can be put on one side and the reference file consulted with no problems.

As far as possible modes have been avoided in the Star system. In other words, the same key has the same effect in every situation, or if it does not it should be visibly obvious that the system is in a strange state. This approach, discussed in [Smith et al, 1982b] is closely related to the Smalltalk environment [Tesler, 1981]. There is a set of special keys for such things as "open", "move" and "copy" which act on the currently selected object. They will work whether the object is an icon for a file, a chunk of text within the file or a single character. There is a "properties" key which allows the user to examine and update the properties of an object, for example the type face used to print a paragraph of text. Other commands, which are selected from menus, also act on the currently selected object. If a command takes more than one argument then the user must fill in the fields of a "form" to specify the arguments. Such a form will contain useful default values, or the values the user provided last time.

The use of small size icons to represent objects allows everything to remain on the screen without it filling up too fast. The icons for such objects as documents can be "opened" and displayed as a large window covering much of the screen. The user can then interact with whatever is "inside" the object, for example edit the text inside a textual document. All the other objects remain on the "desk top" but are temporarily invisible as would be the case if they were covered with paper on a physical desk. Menus of commands are not visible all the time but appear ("pop-up") after certain user actions and disappear once the command has been issued. Similarly, forms to be filled in only appear when they are needed. Figure 3.2, on page 30, shows a typical screen during a session with the Star workstation.

The Hardware

Windowing systems are generally based around black and white bit mapped displays, although simple systems can run on character based displays. A bit mapped display can support high quality graphics which often forms an important part of a window-based user interface. As well as a keyboard these systems also tend to use a mouse, for pointing and selecting on the screen. The Alto personal computer [Thacker et al, 1981] was the first such machine and it has provided the model for many later systems.

The Alto, and other workstations modelled on it, make use of special functions for copying rasters around in order to implement windows. The screen image is taken from a fixed area of memory and new images must be copied in to appear on the screen. Hence, when one window is obscured by another care must be taken that the image in the obscured part is not lost when it is overwritten by the other window. If the obscured part is ever uncovered the original image must be restored. How this is handled by the software is one important point of difference between windowing systems. Some systems store all the obscured regions whereas others require the application to redraw regions that have been obscured. Most workstations in general use are monochrome because the use of colour means a vast increase in the amount of data to be copied around. This either results in very unsatisfactory screen dynamics or else the replication of special hardware for copying operations.

An alternative approach is to use dynamic mapping of graphics memory to the screen, so that the screen image is made up from a mosaic of rasters from different parts of memory. The Rainbow Workstation [Wilkes et al, 1984] was built to experiment with this approach by providing special hardware for combining arbitrary rectangular areas. Every window in the system occupies a separate region of memory and so there is no problem with saving and restoring obscured regions. The fact that no data is being copied around means that full colour windows can be supported with no time penalty, improving the quality of the user interfaces that can be implemented. The Rainbow workstation provided the hardware substrate for my own work and is described further in the next section.

Window Managers

A window manager mediates between the application program and the hardware and there are probably as many approaches to this task as there are window managers. This section highlights the relevant issues rather than providing an exhaustive survey. In particular, many window managers support a program development environment which must provide facilities that are not needed for the more limited graphical applications that I am tackling in this thesis.

The first point to look at when considering window managers is what task a particular window manager is intended for. Many systems provide an interface to a multi-tasking operating system, with the possibility of a number of tasks running concurrently in different windows. This facility is almost essential in a system where lengthy compilations or calculations are being carried out, although it brings with it problems of sharing resources safely. On the other hand, if the applications for the system are highly interactive there is less need for concurrency as the user will only be interacting with one task at a time. The tasks in other windows need only save their state rather than continuing to run in the background. Obviously the user will be held up if one window starts some lengthy process as he cannot then move to another window to do something else. On the other hand, sharing of resources is made much easier.

The window manager provides the interface to the hardware and so is ultimately in control of the screen layout. Both the application ("client") programs and the user of the system have an interest in this layout. How this possible clash of interests is resolved can have important effects on the user interface to the system. The original window managers always supported overlapping windows, reflecting the idea of windows being much like pieces of paper. Windows could be placed anywhere on screen and frequently the user had to specify the location and size of a window before it could be created. More recently some managers, for example Viewers [McGregor, 1983] have adopted "tiling", in which the windows completely cover the screen and do not overlap. Here the system plays a much more active role in arranging the screen layout, making use of information provided by the client programs about their preferred size and position of window. The shapes of windows are constrained as they must always completely cover the screen. New windows cannot overlap existing ones and so various heuristics are automatically employed to determine where a new window can be placed and what reorganisation is required. The user is then free to change the layout to suit himself. This approach relieves the user of the necessity of making frequent decisions about window placement while still leaving him in control of the layout. On the other hand the unpredictability of window placement can be disturbing.

An issue arising out of user control of window size and placement is what information, if any, is passed back to the client program. In some systems, the client program must redraw its window if its size is changed or an obscuring window is moved. In these cases the client must obviously be informed of any changes. Such an approach may seem unsatisfactory as it places an extra load on the client programmer but it has been claimed not to be a major problem. A different approach is that of providing a virtual screen to the client in which they can do what they wish. The window manager handles all placement and obscuring problems by storing the image on the virtual screen and the client need have no knowledge of the window's position or size. The user is responsible for ensuring that the window is of an appropriate size and shape for displaying its contents. Some systems take the view that the client should know about the size of its window on screen so that it can rearrange the window contents. For example, text may be reformatted to fill a wider window or a picture may be magnified or reduced to fit. The user interface toolkit described by Gosling [1984] provides the opportunity for client programs to reformat their windows when the size changes.

The use of icons is a common feature of windowing systems. One use is as graphical representations of the commands available. Rather than providing a menu of textual items describing the commands a menu of icons is used. The intention is that these icons are rapidly and easily understood, for example an icon depicting a circle means "draw a circle". Unfortunately there is little consistency between systems in the use of particular icons. It is also difficult to think up distinctive and informative icons for more than a small number of commands. Another use of icons is as a small-size representation of some object that would otherwise take up a lot of screen space. This is done in the Star system where objects are represented by icons whose appearance indicates the type of object. The objects can be manipulated by manipulating their icons. An icon can be opened to reveal its contents, with the resulting window obscuring a large area of the screen. A very different use of icons is found in the Sapphire system [Myers, 1984]. Again the icons are related to objects within the system but they exist in addition to the object rather than being an alternative representation. The icons are always visible on screen whether the associated windows are off-screen, obscured or actually visible on screen. These icons provide various items of status information about their associated window, for example progress bars show what proportion of the current activity has been completed. Flags in the icon show whether the task is waiting for user input or whether an error has occurred. They can also be used to signal that events have happened, such as the arrival of mail in an icon associated with the mail program.

Any windowing system tends to make use of menus on screen from which commands can be selected. To save screen space these menus often pop-up when they are required and then disappear when the user has finished. On the other hand static menus that are continuously on screen are quicker to use as the user can more easily memorise where frequently used commands are. Particular movements can become automatic and the user does not have to waste time scanning the menu. Some users may even prefer to use the keyboard if single key strokes can be used to evoke commands. There is always a balance to be maintained here between novice users who need prompting and reassurance and experts who want speed.

In many systems visual feedback is important and the cursor usually has an important role. The cursor shape can be used to indicate what state the system is in. Many window systems make provision for changing the cursor shape as the cursor moves from window to window. It can also indicate if the cursor has entered a sensitive region, such as a window boundary used for scrolling or changing the window size.

The final important issue is how the user communicates with the window manager, both to deal with screen layout and also to select a window to receive input. A simple and obvious way to select a window is by cursor position. Any event is automatically passed to the window in which the cursor is positioned. This generally works well, but when the user is typing text into a window it is the text input position that is of interest, rather than the cursor position. If the cursor was accidentally misplaced the text might suddenly start appearing in another window. As a result, some window managers require an explicit action, such as pressing a special button or issuing a command, to select a new window for input. Rearrangement of windows can be achieved in a number of different ways. Some systems have special menus of window management commands which must be issued to change the size or position of a window. A more direct approach is to allow the user to interact directly with the window without having to issue a command. This can be done by such things as having a special button set aside or having sensitive regions around the edges of windows that invoke window management.

Relationship to other graphical systems

This use of the word "window" is quite different from the usage in most graphics libraries such as GKS. In GKS a window specifies a region of an image that is to be mapped to a viewport, and thence onto the screen. There is no concept of support for multiple programs or for organising input through windows. GKS could support a single program within one window but it would be difficult to implement a complete window based environment. Lantz and Nowicki [1984] describes a graphics library in some ways similar to GKS but intended for a window based environment, especially where multiple distributed processors are being used. The application programs, possibly running on remote processors, create structured display files describing what they want displayed and the workstation processor then handles all screen update. The user specifies the size, position and magnification of the viewports onto the display files rather than this being left in the hands of the application programs.

3.4 The Rainbow workstation

The Rainbow Workstation was designed and built in the Cambridge University Computer Laboratory to evaluate the support of windowing by dynamically mapping graphics memory to video. Its architecture is described in Wilkes et al [1984]. Because windows are supported by dynamic mapping of memory rather than bit copying operations, full colour windows can be supported with no speed penalty. This particular workstation and its associated hardware provided the substrate for a graphical environment to support Imp.

The operating environment
The Rainbow workstation is used within the Cambridge distributed system [Needham and Herbert, 1982]. The core of this distributed system is a processor bank consisting of a large number of single user machines and a central file server. A terminal connected to the network can run sessions on one or a number of these machines. The central file server means that a user is not limited to using just one particular machine with its own local disc as all files are accessible from all machines. The Rainbow Workstation is based on a Motorola 68000 system similar to the standard processor bank machines. Like the processor bank machines it runs the Tripos operating system [Richards et al, 1979] and can be used from any terminal on the network. This allowed the workstation to be used before the software to make it into a terminal in its own right was available. Figure 3.3 shows the basic structure of this environment.

Tripos is an operating system for mini-computers with support for multi-tasking and message passing. The messages that are passed between tasks are referred to as packets and are simply vectors of words in memory containing the information to be communicated. Tripos is essentially a single-language system, being written in BCPL [Richards, 1969] and until recently supporting BCPL as the only high level language. As a result all the display software has been written in either BCPL or assembler language. BCPL has only a single data type, the machine word. Operations are provided that treat an item as an integer, a bit-pattern or a memory address and any item can be treated in any way. Hence there is nothing to stop the programmer from assigning a machine address to a variable and then doing integer arithmetic on it. The result can still be used to address memory, whether it is valid or not. Complex data-structures can be built up by using vectors of words in memory, although the lack of checking can make it hard to debug structures of the complexity required by the Rainbow Workstation software. Constant values can be given names by declaring these names as manifest constants with the required value. Values such as lengths of vectors and offsets into them can be set up in this way. By changing the values associated with the names it is possible to change the layout of data-structures and have the program remain unchanged. It also makes it easier to write and read code for handling data-structures if mnemonic names rather than numbers are used.

A BCPL program can be written as any number of separately compiled sections. Access to routines and variables in different sections is provided by the global vector, which is simply a vector of BCPL values. At the beginning of each section names are associated with specific offsets in this vector. These names can be used anywhere within the section and provide access to the value stored at that offset in the global vector. If the name is declared as a routine then its entry point is automatically placed in the global vector. Any number of sections can access the same location in this vector and so communicate with one another.

One limitation imposed to simplify the run-time environment for BCPL is not to allow access to dynamic free variables. In other words, variables declared in an outer block cannot be accessed from within routines declared inside the block, even though the name is in scope. Any variables to be accessed from within a series of routines have either to be global variables or else must be declared as static variables. Static variables are implemented as specific locations associated with the compiled code of the section and they retain their value even when they are out of scope. All

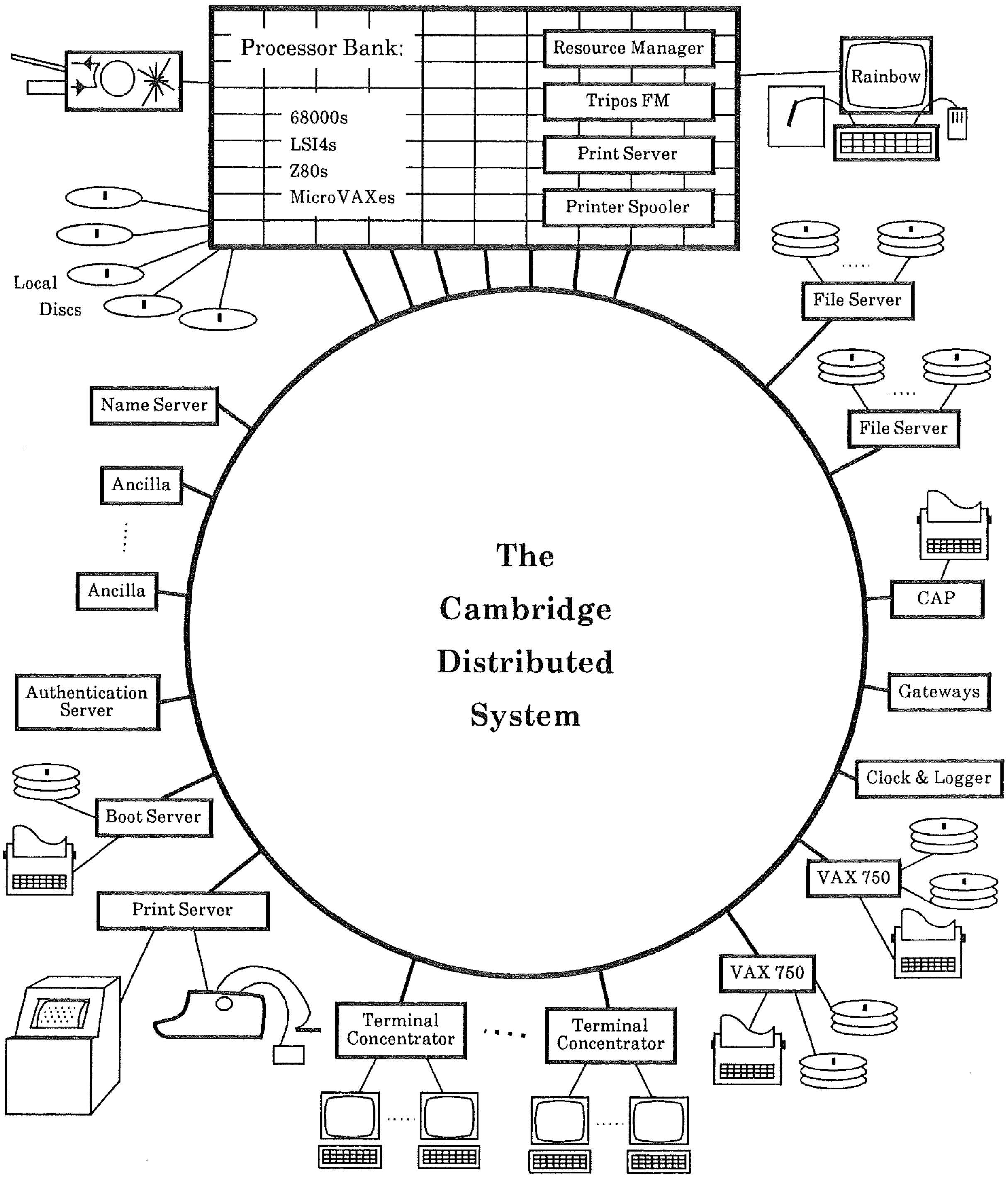


Figure 3.3
The ring environment (courtesy of Dan Craft).

procedure names are automatically declared as static variables. Static variables are declared with an initial value and this, coupled with the retention of the value means that they can be used for such things as counting the number of times a particular piece of code has been called. This feature is used extensively in the window manager described in the next chapter.

BCPL provides a simple and efficient coroutine mechanism [Moody and Richards, 1980] with coroutines implemented as procedures with their own local stack and program counter. A coroutine is created by:

```
coroutine := CreateCo( procedure, stackSize )
```

All flow of control between coroutines is explicit, by way of the procedures `CallCo` and `CoWait`. The current coroutine is suspended and another one started executing by the call:

```
result := CallCo( coroutine, value )
```

When this second coroutine calls `CoWait` the call to `CallCo` returns and the original coroutine continues executing:

```
result := CoWait( value )
```

The value passed in the call to `CoWait` appears as the result of the original `CallCo`. If the original coroutine then calls the second coroutine again, the call to `CoWait` returns and the second coroutine continues. The value passed in `CallCo` appears as the result of `CoWait`. When a coroutine is called for the first time this value appears as the argument to the procedure that formed the coroutine. If the coroutine's procedure returns rather than calling `CoWait` then the result appears as the result of `CallCo` and the next call to the coroutine will start again as if it had never been called. As well as passing values in this way, coroutines can communicate by way of the global vector, which is shared between all coroutines in the same task.

At various points in this thesis the text is illustrated by small pieces of program. These programs are presented in what is essentially BCPL, but with some parts in straight-forward English where this is clearer and more concise. The symbols `$(` and `$)` serve to begin and end blocks and `LET` and `AND` introduce declarations.

Display software

The image that appears on the screen of the Rainbow Workstation is made up of a mosaic of rectangular images drawn from different areas of the graphics memory. The video processor interprets a low-level screen description, called the band structure, which specifies which areas of graphics memory are to appear where on the screen. The graphics memory is arranged in eight planes and so up to eight bits of image data can be displayed in one pixel. Each area in the band structure also has up to eight bits of context information associated with it. These bits are combined with the bits from graphics memory and the resulting twelve bit value is used to index a lookup table of 4096 entries. Different lookup tables can be used for different parts of the screen image by assigning different context values. The bits that make up a pixel can be taken from different offsets in each plane of graphics memory, allowing images to be combined and moved freely. For example, an anti-aliased object can be moved over a multi-coloured background in real time with the correct shading being produced by the lookup table. Without the specialised hardware a large amount of computation would have to be done and the real time effect would not be easy to achieve. These and a variety of other combining and recolouring effects are referred to by the general name of "transparency". More details can be found in Glauert and Wiseman [1985]. A more straight-forward use of the hardware is to support overlapping windows, where one object obscures, rather than combines with, another. This is the way the workstation has been used for the system described in this thesis.

The client is provided with a high-level screen description in the form of a rooted acyclic directed graph which is compiled into band structure every time something changes. The root node of this tree represents the whole screen. The leaf nodes of this structure are actual areas of graphics memory and are referred to as "pads". The internal nodes are referred to as "clusters" and these serve to organise the objects mapped into them. The cluster has a specified size and other clusters or pads are mapped into it relative to its origin. These objects are clipped to the boundary of the parent cluster. The arc linking an object to its parent cluster has associated with it the object's position in the cluster and its priority. This arc can also be turned on or off to make the object visible or invisible. There is no limit on the number of arcs associated with an object, enabling the same object to appear many times on the screen. If several objects mapped into the same cluster overlap then that with the highest priority arc will be visible. If two or more arcs have the same priority then the resulting image is a combination of all the objects involved, giving rise to the various transparency effects mentioned above. Procedures are provided for setting up lookup tables, which are associated with individual pads or combinations of overlapping transparent pads. Figure 3.4 shows a simple screen layout and the graph structure that the client must set up to achieve it. Styne et al [1985] describes this software and what follows is a summary of the various procedure calls provided.

A library of procedures is provided for creating and destroying pads and clusters and for building and manipulating the graph describing the screen. A pad is created by the call:

```
pad := CN.create( IsPad, xmax, ymax, planes, allocType, startPlane )
```

The size of the pad in pixels is defined by `xmax` and `ymax` and `planes` specifies the number of bits per pixel. `AllocType` and `startPlane` can be used to specify where in graphics memory the planes making up a pad will be located. If this is not important then both values can be given as 0 and the software allocates the planes in the next available spaces. The specific planes must be given when transparency is to be used as the interacting pads must be on disjoint planes. If a pad is created with 0 planes it takes up no space in graphics memory and can be used to provide an area of plain colour. Such a pad is called a "virtual" pad. Clusters are also created by the procedure `CN.create`:

```
cluster := CN.create( IsCluster, xmax, ymax )
```

No graphics memory is associated with a cluster but its boundaries, as defined by `xmax` and `ymax`, serve to clip any object mapped into it. Its size can be changed at any time:

```
CN.size( cluster, xmax, ymax )
```

An arc is created by inserting a padad or cluster into another cluster, specifying the

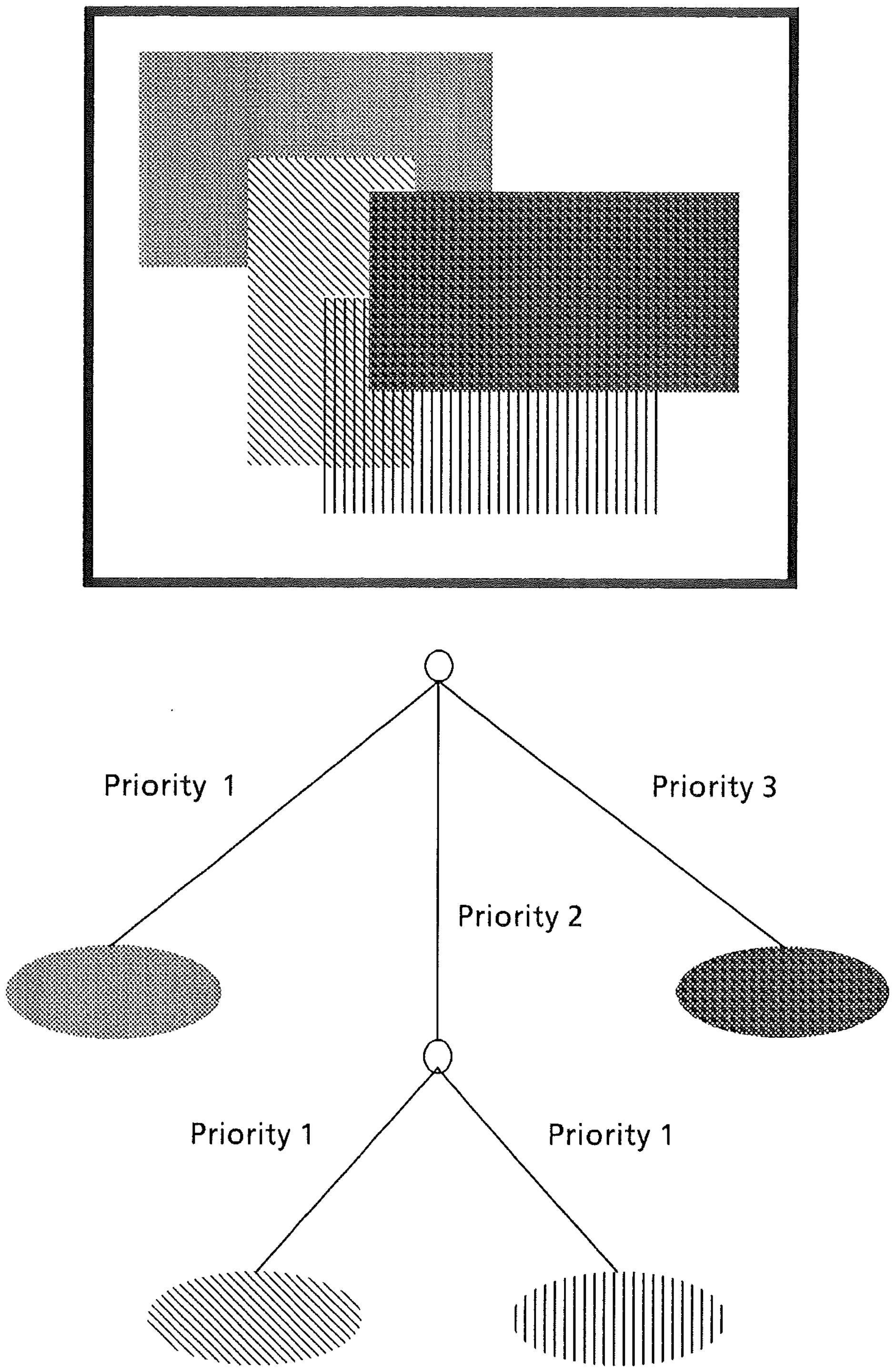


Figure 3.4
A sample screen and the associated pad structure

position and priority:

```
  arc := CN.insert( object, cluster, x, y, priority )
```

This arc can then be passed to various procedures to modify it:

```
  CN.move( arc, x, y )
  CN.priority( arc, priority )
  CN.turnOn( arc )
  CN.turnOff( arc )
```

Before a pad can be displayed it must be provided with a lookup table. A region of lookup table is set aside by:

```
  region := LU.getRegion( planes )
```

`planes` should be the number of planes in the pad or pads that will use this region. The region is set up by:

```
  LU.setRegion( region, vector )
```

`vector` contains the colour values for the different pixel values. Any number of pads can make use of the region, through the call:

```
  CN.setLURegion( pad, region )
```

A procedure is also provided that takes a number of pads, plus colour vectors for each pad and sets up the tables for all combinations of these pads. The pads must all be mapped into the same cluster at the same priority. Special transparent and translucent colour values are provided. When these are included in the colour vectors the appropriate colour mixes occur when the pads interact. The effects of any of the procedures that change the graph structure or lookup tables are not seen until `CN.display` is called. This allows a large number of changes to appear instantaneously on the screen.

Another library of procedures is provided for drawing images in pads. These do not build any sort of display file but just set particular pixels in the pad to a specified value. Single pixels, lines, filled rectangles and arbitrary filled polygons can be drawn. A state-free interface is provided where there is no notion of such things as current pen position or current colour. All this information is specified in each call, for example:

```
  CN.drawLine( pad, x1,y1, x2,y2, value, planeMask )
```

The two end points are specified by `x1,y1` and `x2,y2` . The combination of `value` and `planeMask` specifies the bit pattern to be written to the pixels. Where a bit in the plane mask is 0, the bits in that plane will remain unchanged. A 1 allows that bit from the value to be written into the plane. There are a large number of parameters for each call, but this approach allows calls from different asynchronous processes to interleave without problems.

These display procedures provide good support for windowing. The model of clusters and pads maps very directly onto sets of windows overlapping according to priority. The procedures can be used as subroutines within a single Tripos task which takes complete control of the display. Alternatively the display procedures can be provided in a separate task, where they can be called from several different tasks, or even from remote machines on the network. This method of working allows the Rainbow workstation to support simultaneous sessions on a number of machines and so can provide the basis for a network terminal.

Tools handling
The Rainbow workstation is capable of supporting a number of different tools and currently a keyboard, mouse and graphics tablet are available. At the lowest level each device is handled by a Tripos device driver which deals with interrupts and sends data to other Tripos tasks when requested. The high-level interface used to support my own work has been provided by Bruce Styne and is described further in [Styne, 1985].

This high-level interface runs as a Tripos task and takes care of communication with the device drivers. It maintains a queue of events, labelled with the originating device and whether they are switch or coordinate events. A client program, running as another task or even on another machine, can set up one or more channels to this tools task:

```
channel := OpenChannel()
```

The types of events and devices to be returned on this channel can now be specified:

```
SwitchChannel( channel, device, eventType )
```

The same channel can be quoted in more than one call to this procedure so that, for example, switch events from several devices can be returned on this channel. Information about the next event on a channel is obtained by the call:

```
ReadChannel( channel, packet )
```

`packet` is a vector into which information about the event will be copied. If no event is waiting then the client program is suspended until one occurs. Alternatively the client can poll the channel to see if there are any events before attempting to read from it:

```
result := TestChannel( channel )
```

If the result is `TRUE` then a call to `ReadChannel` will return immediately. The tools handler buffers up all switch events but only retains the latest coordinate event. This means that if the client does not request events for some time then no significant actions are lost, although the coordinates may jump.

Coordinate events are generated by devices such as the mouse and tablet and occur either after a specified timeout has elapsed or if the coordinates have changed by more than a specified increment. In addition special events can be generated to mark the beginning and end of a continuous movement (a stroke). These occur when the coordinates first begin to change and then when they have not changed for a specified period. The mode of the device determines which particular behaviour the device exhibits. In mode 0 events occur in response to significant changes of coordinates but not after a timeout. Hence, if the device is stationary no events are generated. In mode 1 the beginning and end of stroke events are generated as well, and the timeout value is used to determine the time to elapse before the end of stroke event is generated. In mode 2 an event is always generated when the timeout expires and also when the coordinates change significantly, if this is sooner. The behaviour of a device is set up by:

```
ResetDevice( channel, device, mode, deltaT, xOrigin,yOrigin, deltaXY )
```

deltaT is the timeout and deltaXY the change in coordinates that is to be considered significant. xOrigin and yOrigin give the x and y value corresponding to the current position of the device.

The packet describing a coordinate event has the following fields:

```
1. Type - coordinates
2. Device identifier
3. x coordinate
4. y coordinate
5. mode-specific
```

The mode-specific field has the value 0 if the event is in response to a change in coordinates, 3 if in response to a timeout, 1 for a start of stroke and 2 for end of stroke.

Switch events occur whenever a switch, button or key is pressed. Some devices, such as the mouse or tablet stylus, produce "raw" switch input which consists of the index number of the switch on the device and an indication of whether the switch is being depressed or released. Other devices, such as the keyboard, produce ASCII data which consists of an event giving the ASCII code for each key depression. No event occurs when an ASCII key is released. A key table is provided which specifies which ASCII code is returned by a key, what value is to be returned if the shift and/or control keys are held down simultaneously and whether the key auto-repeats when held down. The default key table provides standard keyboard behaviour, for example shift+alphabetic key produces the upper case character, but the client can provide any other key table he wishes. The fields of a switch event packet are:

```
1. Type - switches
2. Device identifier
3. Data type - raw or ASCII
4. Switch number or ASCII code
5. TRUE if switch is going down
```

The library can be set up to produce a range of different behaviours. In particular, it is useful to be able to experiment with the response characteristics of coordinate devices. Being able to change the mappings of physical keys to numeric codes is also a very valuable facility.

4. The environment for Imp

This chapter describes an interactive graphical environment implemented on the Rainbow Workstation. It was constructed specifically to support Imp and this is reflected in a number of the design decisions. The requirements and the resulting decisions are outlined in the first section of this chapter. The rest of the chapter goes on to discuss the actual implementation details.

4.1 A specification

The previous chapter introduced various issues in interactive graphics and presented different approaches to dealing with them. Here we tackle points that are of particular relevance to a type face design system. The resulting model is also influenced by the particular abilities of the Rainbow workstation.

Requirements
Any form of graphic design, including type face design, is a highly interactive and visual process. The results are approached by many small steps guided directly by the designer rather than by long, automatic calculation. Hence, any program to support such designing must be interaction-driven and the underlying environment must provide facilities to make the handling of interaction straightforward. In graphic design a task may be tackled from a number of directions or at different levels. The designer may wish to move freely between different aspects of the same design, pursuing several tasks in parallel. Such a style of working is most easily supported by adopting an explicitly multi-threaded approach. Each task can be specified as an independent event handler, with the user free to move between them at any time. When the designer is working on one task, it can be very important that the state of other tasks is also visible. This method of working can be supported directly by the use of windows on the screen, each providing the visual interface to a particular task.

Different people may use different tools for the same task and almost certainly different tools will be needed when a different task is tackled. This suggests that the interactive environment should provide a number of input tools, any of which can be used at any time. Any devices that give variable numeric values to be used as coordinates, along with one or more switches to signal events, can be used.

A problem with the use of computers is the abstract nature of the objects being manipulated. The user can feel very insecure when objects are not actually visible. The use of a graphical user interface that makes everything visible goes a long way towards relieving this anxiety. Objects should be manipulated directly by "touching" them rather than by issuing a command to the system to do it. This helps to build a concrete model of what is happening and is much more akin to the way the physical world behaves. When more complex information must be communicated to the system it can still be done visually through menus or other devices on the screen.

Another feature that helps to build up the user's confidence is to make it easy to undo the effects of actions. This will encourage him to experiment and so more rapidly learn to exploit the system's facilities. Designing is particularly helped by such an experimental approach. Some actions, such as overwriting a file, cannot be reversed and so in these cases the system must make it difficult to do it by accident.

The view point of the client programs that will be implemented in this environment must also be considered. Such a program needs information about significant input events but should not need to deal with low level device handling. Generally the program needs to know when a switch has been depressed or released and what the current cursor position is. Usually the client programmer will have no desire to deal with updating the cursor position on the screen and so will not want to be continuously informed of cursor movements. For such a client, the underlying system should handle the cursor and merely inform the client about the significant events. On the other hand the client may wish to provide special feedback as the cursor moves, for example, to sketch a line. Hence, the system should allow a client program to override the default cursor feedback. The client may also wish to change the cursor shape.

The client program resides entirely within its own window and the programmer should not have to deal with anything to do with positioning the window on the screen. All window management functions and the interactions with the user to support them should be handled by the underlying system. Anything that the client program displays in its window should be automatically clipped to the window boundaries.

The job of the client programmer can be greatly eased by the provision of a "toolkit" containing various useful facilities that go beyond basic graphics. This might include support for constructing and using graphical menus, for handling messages to the user in a standard way and so on. Not only does this reduce the load on the programmer, it provides consistency across applications with the same things being done in the same way. The client is free to use the toolkit or not as he wishes so that he is not constrained by something that is not quite suitable for the task in hand.

The implementation

The model adopted, as indicated above, is that of writing an application as a series of event handlers, each running in its own window. A window manager handles the input from the tools and also deals with positioning the windows on the screen in response to the user's actions. The event handler associated with each window responds to events that happen within that window.

The event handlers are implemented as coroutines, that is procedures that can suspend themselves and then be resumed later without loss of local state. The coroutine for a particular window is called when an event occurs in the window. It processes this event and then suspends itself ready to receive the next event. There is no preemption and so control remains with this coroutine until it suspends itself. Conceptually the program is multi-threaded, but at any moment only one coroutine is active. This fits very well with the fact that the programs making up Imp are largely interaction-driven. At any time the user is only interacting with one task and all the others will most likely be suspended waiting for further input. By adopting this approach a multi-threaded model is mapped onto a single thread at run-time. This allows a considerable simplification of the implementation as there is no need to synchronise access to data shared between windows. Because the applications tend to consist of several different windows acting in different ways on the same data this could have become a major problem.

The input from the tools is divided into coordinate changes and switch events. The cursor position is updated by the window manager whenever a coordinate change is received from any device. If the activity associated with a window needs to make use of continuous feedback of the cursor position it can provide the window manager with a routine to be used in place of the default cursor update routine. This is then called whenever the coordinates change with the cursor within that window. When a switch event occurs the window manager finds the window that contains the cursor position on the screen and passes the event to the handler for that window. This event handler can then request the current cursor position with respect to its window origin if it is needed for processing the event.

As well as input from tools, events can be generated internally. An event handler can pass back an event to be sent to a specified handler, including itself. This facility can be used to provide communication between windows. It can also be used to do things such as making a menu selection appear like an input event. When the event handler detects a menu selection, rather than acting upon it directly it can send an event to itself describing the selection. This makes it easy to change the program later to use a direct input event to evoke the selected action. It is also possible to have both input events and menu selection evoke the same action. The cursor update routine can also give rise to events so, for example, it could signal to an event handler that a particular boundary within its window had been crossed.

Events are described by packets that contain information about the originating device, the type of event and the event data. Switch events are described by a switch identifier and an indication of whether the switch is going up or down. Coordinate events are described by the current coordinate values of that device. The window manager scales all coordinates to fall within the same range and converts relative coordinates to absolute values. Hence, the cursor update routines and the event handlers always receive coordinates in terms of screen position.

The user's interface to window management has been provided by reserving a button on the mouse. Pressing this button evokes various effects depending on where the cursor is on the screen. If the cursor is near the corner of a window then the size and shape of the window is changed as the cursor is moved. If it is near the middle of a window then the whole window is moved. One of the mouse buttons was sacrificed in order to keep window management separate from the other functions of the system. This approach also gives the user a very direct feel for the windows as objects because he can move a window by reaching for it with the cursor and then "grabbing" it by pressing a button. The window is dropped by releasing the button. Using menus or commands to control window size and position would have been less direct.

The window manager makes use of the display library, described in the previous chapter, that represents the screen as a tree structure with raster images as leaves. The arcs in the tree have a priority and position with respect to the parent node. Overlapping rasters within the same node obscure one another according to priority. The windows are top level nodes in the tree and the client is then free to create any structure within the node for a particular window. The display library automatically clips rasters to the boundaries of the parent node and so the client program does not need to be concerned with straying outside its screen area. The basic library of procedures for drawing in the rasters does not maintain any structured representation of the image. The client program must maintain any such structure itself, if it is needed. Procedures for handling menus and forms have been implemented to make certain standard features easier to program.

4.2 Some building blocks for an interactive environment

There are a variety of widely used techniques in graphical interfaces and the provision of standard procedures that implement them can be of great value in reducing the effort required to develop new applications. The use of standard procedures also helps to ensure some degree of consistency across applications using them. I have implemented libraries of routines to support graphical menus and forms. The menus are used for presenting options to the user and for obtaining a single response. The forms allow a user to fill in a series of fields to provide a number of items of information to the client program. These routines can be used in any program that makes use of the Rainbow workstation. In addition, the window manager itself provides a number of other facilities for use by its client programs. These are described in the next section.

Menus

A menu is some sort of array of symbols or text which indicate the choices open to the user at a given time. Selecting an item will invoke an action dependent on the item selected. The simplest way to select an item is to type an index number or letter but this technique is not of great interest in a graphical environment. In this case a selection can be made by moving a cursor to the item to be selected and then signifying selection by, for example, pressing a button. The cursor can be moved by any suitable input tool. Many graphics systems provide facilities for finding objects near the cursor when a particular event happens. With such facilities menus can be implemented easily. The Rainbow software does not provide any such facility and so when a selection event happens the client program must carry out the calculations needed to locate the item selected. Hence, it was decided to implement a library of routines for constructing and using menus made up of rectangular items of the same size. This is a considerable simplification of the general "picking" facility of most graphics systems but provides for the construction of a variety of useful menus. Some typical menus that can be implemented with these routines are shown in figure 4.1.

Such a menu is created by the call:

```
menu := Menu.create( rows, columns, planes, region,
                          itemWidth, itemHeight, itemsVec )
```

`rows` and `columns` gives the number of rows and columns that the menu is to have and `itemWidth` and `itemHeight` give the dimensions of a single item. The pad for the menu will be created with the given number of planes and region is the lookup table region to be used. `itemsVec` is a vector of specifications for the appearance of each item. The first word contains the number of items, which need not equal rows multiplied by columns. The menu is filled until either all spaces are filled or there are no more item descriptions. The specification for a single item contains the following fields:

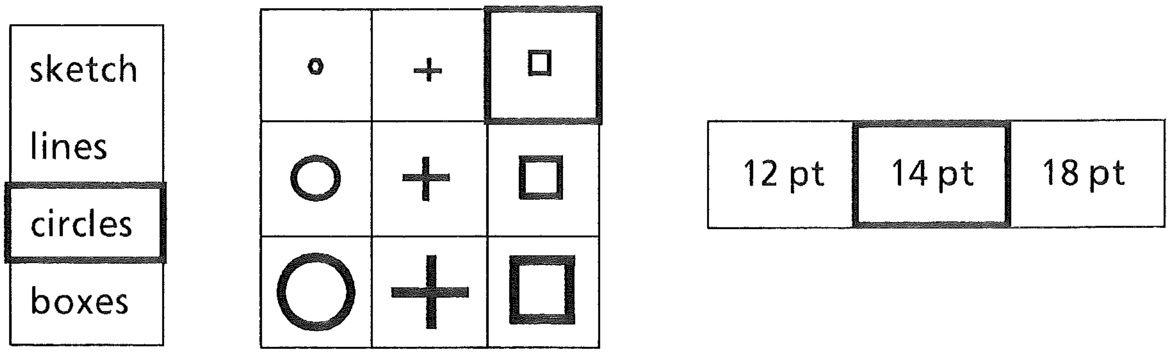
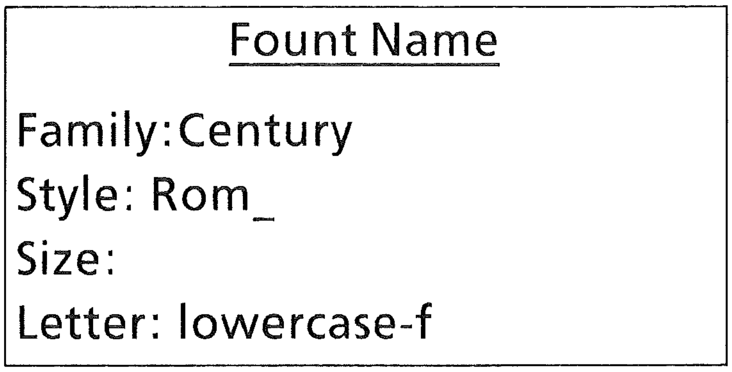


Figure 4.1
Some typical menus supported by the menu package: The heavy box indicates the current selection.



Fount Name

Family:Century

Style: Rom_

Size:

Letter: lowercase-f

Figure 4.2
A typical form: The fields can be filled in any order with the under-score showing the current typing position.

```
1. A procedure to draw this item
2. The first argument for the procedure
3. The second argument for the procedure
4. The value to return when this item is selected
```

When an instance of the menu is created the procedure for each item is called in turn, passing in the client's arguments along with the size and position of the item in the menu pad:

```
procedure( menuPad, x,y, width, height, arg1, arg2 )
```

If the client's procedure takes into account the size when drawing the item it is then possible to change the size of the menu without rewriting the code for each item. Unfortunately there are problems with the size of founts and so it is not practical to create menus of textual items whose size can be changed freely. The menu creation procedure returns a pointer to a "menu descriptor", which is a vector containing various information describing the menu.

Once created the menu can be mapped into clusters like any other pad, using a special procedure:

```
arc := Menu.Insert( menu, cluster, x, y, priority )
```

The resulting arc can be moved around and made visible or invisible using the standard display procedures.

A menu selection is made by calling the selection procedure and passing in the menu descriptor and the cursor coordinates relative to the bottom left corner of the menu:

```
value := Menu.select( menu, x, y )
```

The item that the cursor is over is highlighted by drawing a box around it in the most significant plane of the menu pad. The visual effect of this will depend on the lookup table that the client provided. If none of the menu items themselves make use of this plane then we have no problem with items being erased where the box overlays them. The value returned is the one that the client supplied for the selected item. If the cursor lies outside the menu, no item is outlined and the value `Nil` is returned. This routine can be called repeatedly as the cursor moves over the menu to give continuous feedback of the item to be selected. The client program would take the value returned by the final call as the actual selection, ignoring all the previously returned values.

Forms

Even in a graphical environment it is sometimes necessary to type in text from a keyboard. This is usually to provide textual names for things but can also be used for collecting numeric information for which a graphical representation is not necessary or convenient. A form is an object that appears on the screen with a number of fields that can be filled by typing. A marker in the form shows where characters will be put when the user types and this marker can be moved between fields by using the cursor keys. A form is more flexible than simply prompting for items sequentially as the user is free to move between the fields in any order. He can also go back and alter a field he has already filled in before signalling that the information is ready to be used. The information given in a previous use of the form can be left there to provide default responses the next time. A typical form is illustrated in figure 4.2.

A form is created by the call:

```
form := Form.create( planes, region, title, itemsVec )
```

`title` is a string to be displayed at the top of the form. The form pad is created with the specified number of planes and makes use of the given lookup table region. `itemsVec` is a vector of descriptions for each of the items in the form. The descriptions have the following fields:

```
1. The prompt string
2. A vector to contain the response string
3. The maximum number of characters allowed in the response
```

The prompt string is displayed in the form as a label for that field. Any characters typed with the marker in that field are placed in the response vector as well as appearing in the form on the screen. The first byte of the response vector will always contain the number of characters typed so far so that the vector can be treated as a BCPL string. The procedure returns a pointer to the form descriptor.

The client can display the form, like menus, where and when he wishes. The form is filled in by calling a special procedure whenever a switch event occurs:

```
Form.Call( form, packet )
```

If the packet is for an alphanumeric key then the character is added to the result vector of the current item if there is space. If the user has typed too many characters for this item then the cursor is flashed and a message written out. If the key is the up or down cursor key then the marker is moved to the previous or next field in the form respectively, wrapping round from top to bottom. The delete key causes the character before the marker to be deleted and the marker stepped back one position, unless the marker is already at the beginning of a line. Other switches are not understood and so just cause the cursor to be flashed.

Strings from elsewhere, for example initial default values, can be copied into the result vector. By calling:

```
Form.reset( form, itemNumber )
```

the new string for this item is displayed and the marker position is reset.

These routines do not impose any interpretation on the information typed into the result vectors. It is left entirely up to the client program. A small library of routines for converting between strings and numeric values has been provided to make interpretation easier. A further extension would be to include such interpretation in the form-filling routines. The client would merely indicate what type of value was expected and then the information typed by the user could be converted and checked automatically. In fact the lack of such a facility has not caused much inconvenience as most of the information collected by forms has been textual rather than numeric.

4.3 The window manager

The window manager provides the interface between the user and the client program. It exploits the facilities of the Rainbow Workstation hardware and

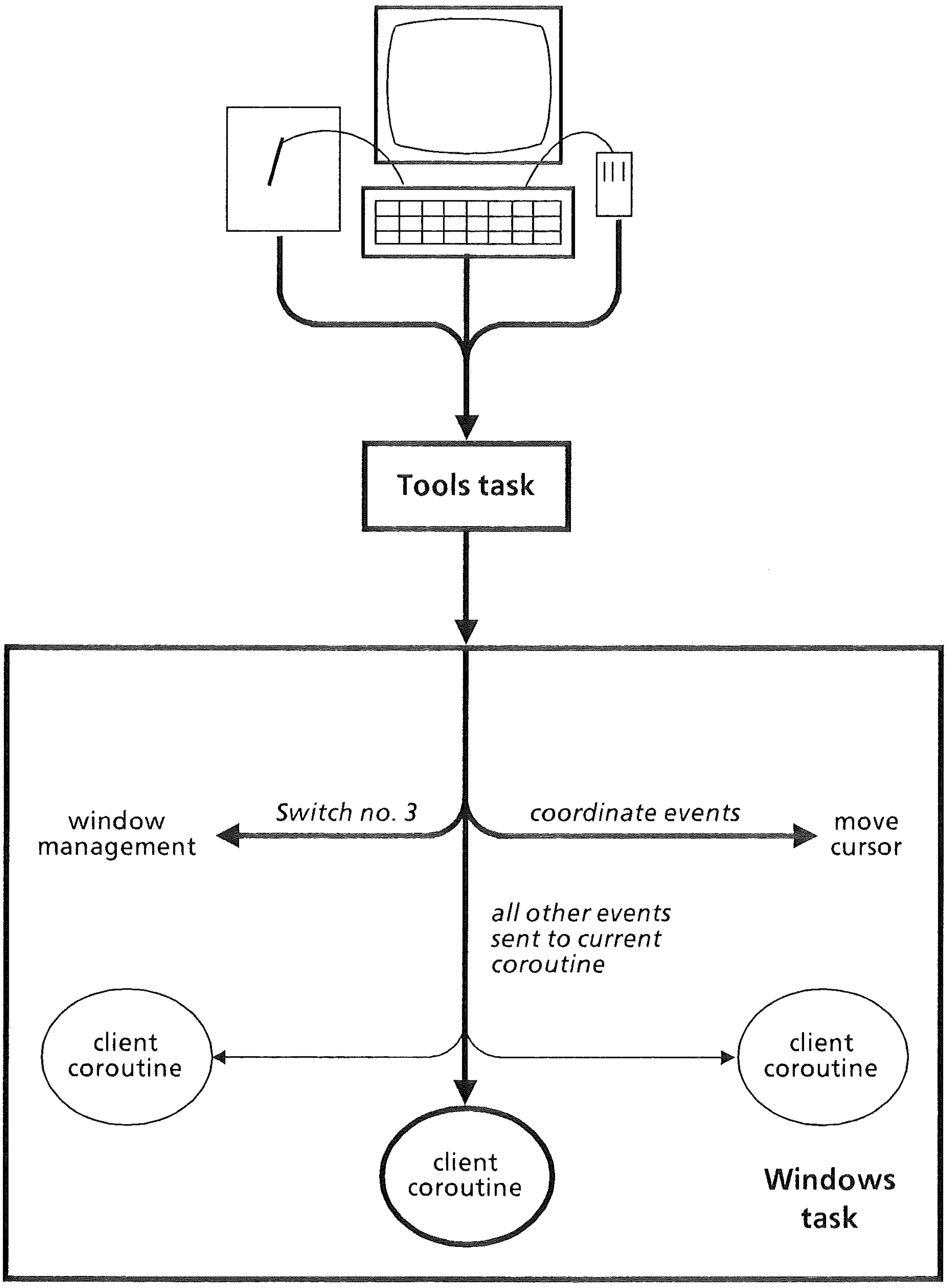


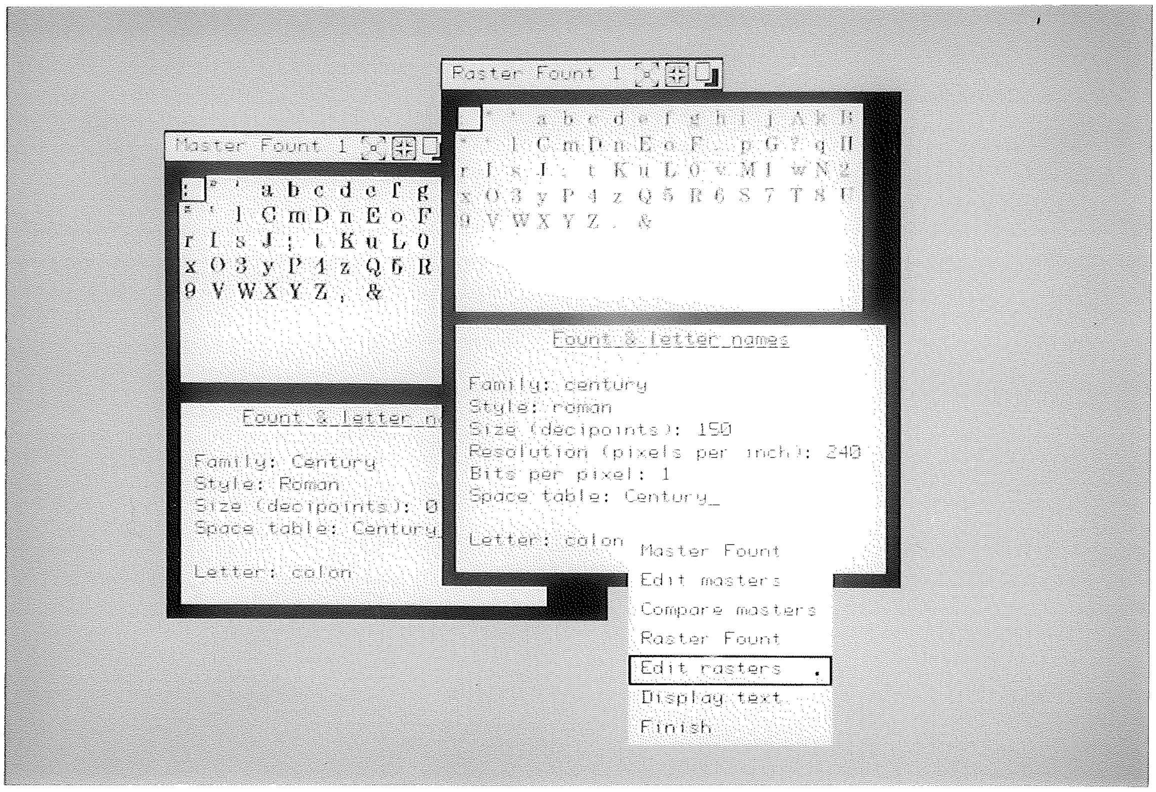
Figure 4.3
Structure of an application using the window manager.

software to support windowing in a direct and simple manner. The input tools are handled by the software described in the previous chapter, with input from all the tools being sent along a single channel to the window manager. The window manager handles cursor movement and also intercepts the third mouse button to provide window management functions for the user. The client program is written as a series of coroutines, each associated with a different window, and all other switch events are passed to the current coroutine. Figure 4.3 shows a summary of the overall structure of an application program. In what follows the window manager is first described from the view points of the user and of the client program. The internal structure of the manager is then described in more detail.

The user's view

When the system starts up the user is presented with a blank screen with a cursor that can be moved around the screen using the mouse or stylus on a graphics tablet. The mouse has three buttons and the tablet stylus is pressure sensitive. When the middle mouse button is depressed, a pop-up menu appears listing the types of windows that can be created, for example "Edit rasters" or "display text". The menu remains displayed as long as the button is held down, and as the cursor moves over the menu different items in it are outlined. When the button is released the outlined item is considered to have been selected. If the cursor has been moved out of the menu then nothing is outlined and so releasing the button selects nothing. If an item has been selected then the window manager creates a window of this type. The new window is created at the cursor position, or as near as will ensure that the whole window appears on the screen. All windows have a tag at the top left corner which gives the type and serial number of the window (the user may create further windows of the same type, hence the serial number). The tag also contains three "icons" which will be explained below. The contents of the window itself will depend on the client program, which is given complete freedom to display what it wishes in its own pads. If the user moves the cursor over the background and depresses the middle mouse button again he can create more windows of the same or different types. A new window always has the highest priority and so obscures any window that it overlaps (figure 4.4).

In order to communicate with the client routine associated with a particular window the user must move the cursor into the window. As soon as any action occurs (a switch being depressed or released) it is passed to the routine associated with the window. This window is automatically given the highest priority so that it is not obscured by any other. The effect of any action will depend on the client program. The user does not need to take any special action before leaving a window as the routine will just resume from where it left off when the user returns to it. This gives the user complete freedom to switch his attention from one window to another without the need for administrative action. The user can investigate the possible results of an action by depressing the "help" key on the keyboard. This sets the system into help mode where any action causes messages to be displayed describing what would happen, but without doing it. The sort of information provided depends on the client but the system provides a routine for displaying this information in a standard place to give some consistency across applications. A special cursor is displayed while the system is in help mode so that the user is aware that the system is in an unusual state. The system can be set back into the normal state by



Fount & letter names

Family: century
Style: roman
Size (decipoints): 150
Resolution (pixels per inch): 240
Bits per pixel: 1
Space table: Century_

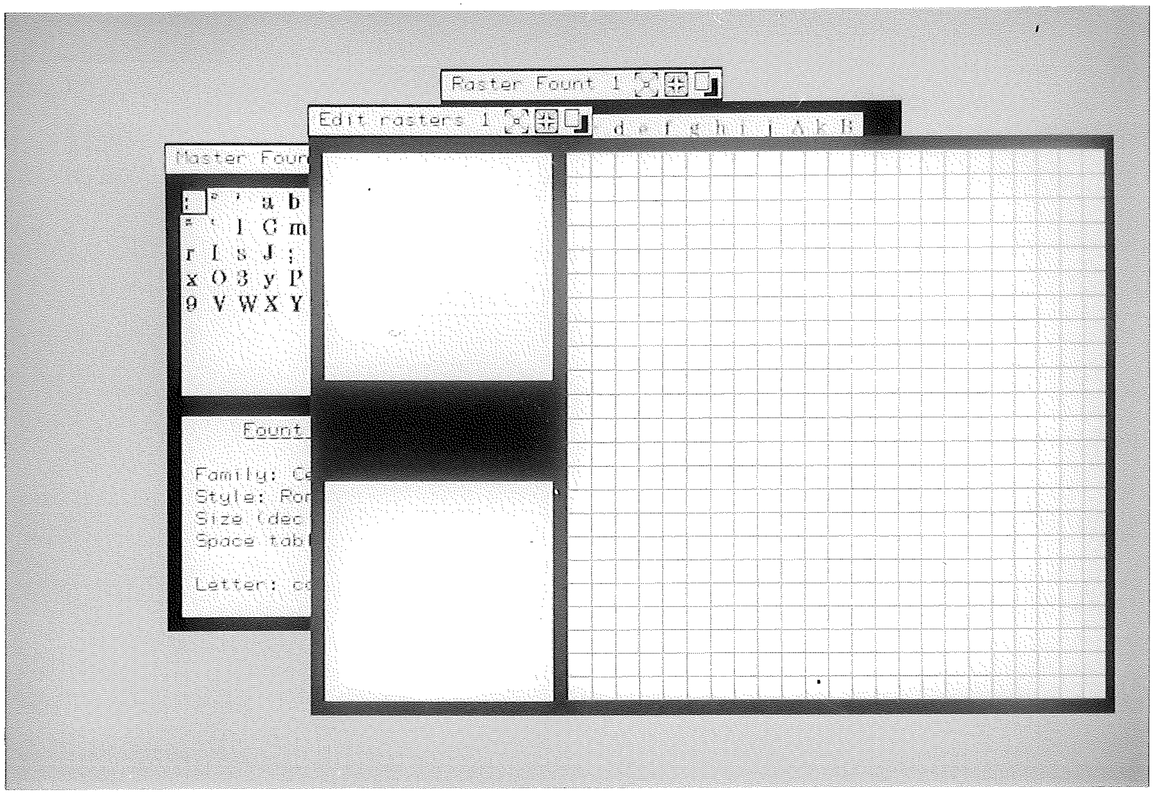
Letter: colon

Fount & letter n

Family: Century
Style: Roman
Size (decipoints): 0
Space table: Century_

Letter: colon

The window required has been selected in the main pop-up menu.



The new window is created when the switch is released. It overlays the existing windows.

Figure 4.4
Creating a new window with the window manager.

depressing the help key again.

One switch, the right hand mouse button, is reserved for use by the window manager. Each window can have any side or corner moved in or out and the whole window can be moved about the screen. Which of these actions occurs depends on where within the window the cursor is positioned when the right hand mouse button is depressed. It is as if windows are divided into three strips vertically and horizontally and a specific action is associated with each of the nine resulting areas. If the cursor is in one of the four corner areas, then the corner moves with the cursor until the button is released (this can adjust height and width simultaneously). In the four areas in the middle of the sides just the associated side is moved in and out (see figure 4.5). If the cursor is in the middle area then the whole window moves with the cursor. The window can also be moved by placing the cursor over the window name in the tag. The only limitation imposed is that no part of a window may be moved off the screen and that a window can only be expanded as far as some client-specified maximum. This gives the user great freedom in arranging for only the parts of windows currently of interest to be displayed. Windows can be shrunk down so that only their tags remain and because they can also be stacked on top of one another a large number of windows can be accommodated. Quite frequent actions are to expand a window to maximum size and to shrink it to the minimum, and special actions are provided for this. It was mentioned above that the tags on windows contained three icons—these are for expanding a window to maximum size, shrinking it down to the minimum and for flipping it to the bottom in the priority ordering. The action is carried out if a mouse button is depressed whilst the cursor is over the icon. All window management actions cause the window to be given the highest priority, except for the icon for flipping to the bottom.

The user can destroy windows at any time and can then close the system down when there are no windows left. An individual window is destroyed by the user communicating this desire to the client program within the window. The actual action required will depend on the client program but it is generally the selection of the "Finish" option in a pop-up menu. After destroying a window, if the next thing the user does is to hit the "cancel" key then the window is restored. This is to prevent the user from accidentally destroying a window. The whole system is closed down by selecting "Finish" in the background pop-up menu.

Although the above discussion has been in terms of the user manipulating a mouse the system works just as well with other input devices. The user can move the cursor around with either the tablet stylus or the mouse. All the input devices are active all the time although unpredictable results may be obtained if more than one device is manipulated simultaneously. Three keys on the keyboard return the same values as the mouse buttons and the tablet stylus switch returns the same value as the left mouse button. This flexibility allows the user to switch between tools as he wishes during the course of a session. It is also possible to add further different input tools if they are desired.

The client's view

The client's program consists of a collection of window specifications, which are handed to the window manager. The core of a client program can be summarised as follows:

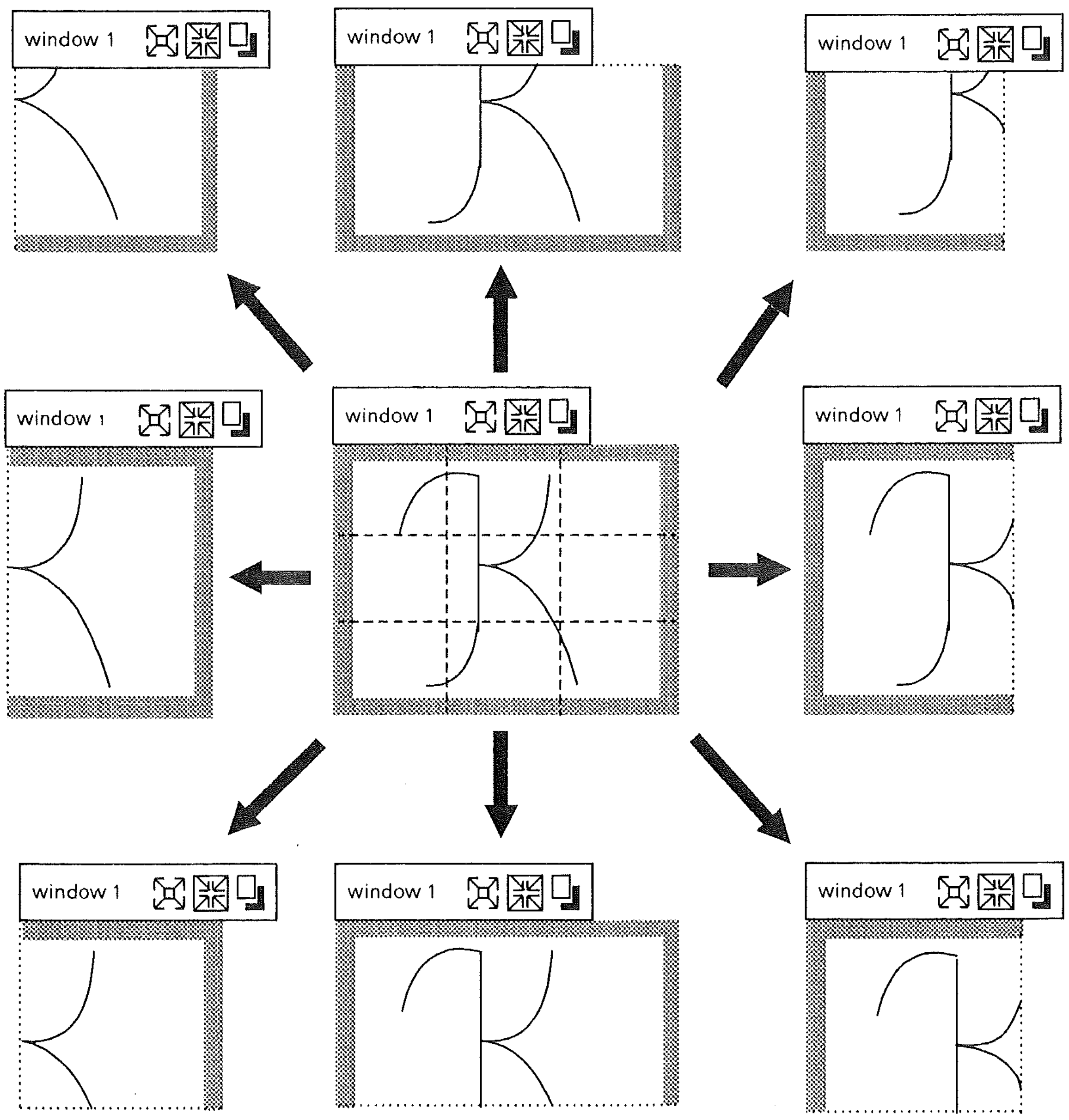


Figure 4.5
This shows the effect of clipping windows from the corners and edges.

```
LET Start() BE
$( set up window specifications S1 to Sn
   Windows( n, helpVec, S1 ... Sn )
   tidy up
$)
```

The call to the procedure `Windows` does not return until the user selects "Finish" in the window manager's menu. In the meantime the flow of control is in the hands of the user who creates and destroys windows and communicates with them as he wishes. `helpVec` is a vector of strings providing short descriptions of each window so that the system is able to provide some help for the user. A window specification is a vector with the fields:

```
1. width
2. height
3. colour
4. procedure
5. stack size
6. name
```

`width` and `height` define the maximum size that the window needs to be to show all that the client requires. The largest window allowed is the same size as the screen. The system provides a virtual background pad for all windows and it is set up with the given colour. This allows different types of windows to be distinguished by colour. `procedure` becomes the coroutine for the window, with a stack of the specified size. `name` appears in the window manager's menu and also in the tag of any window created to this specification.

When the manager creates a window in response to a user request it sets up a cluster containing the background pad and creates the client coroutine. This coroutine is then called with the cluster as argument so that it can carry out any initialisation needed. This will usually include creating clusters and pads to be mapped into the cluster it has been passed as argument. These clusters and pads will form the image that will appear in the window on the screen. It must then suspend itself by calling `CoWait`, thus passing control back to the manager. From then on it will be called whenever an event occurs within its screen area. The argument passed in is a packet describing the event which must be interpreted by the client coroutine. After dealing with the packet the client must return control to the manager by calling `CoWait` so that further input can be handled. A typical structure for such a coroutine is:

```
LET Procedure( cluster ) BE
$( initialise - create pads and menus etc.
   $( packet := CoWait( Nil )
      act on data in packet
      IF termination conditions
      THEN IF CoWait( Window.Kill )
      THEN BREAK  // jump out of repeat loop
   $) REPEAT
   terminate - destroy pads and menus etc.
$)
```

When the user asks for the window to be closed down, probably by a menu selection, the window manager must be informed so that it can tidy up the data structures involved. This is done by passing back the value Window.Kill in the call of CoWait. A safeguard has been provided, in that the manager can make the window invisible and then wait for the user's reaction. Nothing has yet been destroyed and so if the user hits the cancel key the window is made visible again, the result FALSE is returned to the client and the coroutine can continue as if nothing had happened. Otherwise the coroutine goes ahead and executes its termination code before returning control to the manager for the last time.

The system provides no restrictions on the structure of a coroutine, although the most useful form is that given above. A typical extension is to have two or more loops for the reading and interpretation of data packets. These loops would provide different interpretations of the same actions to be used at different times. The client can also put new values in the data packet and pass back a message to the manager to request that this packet be passed to another window, rather than new data being read from the tools channel. This allows windows to communicate with one another. This facility can also be used within a single window to cause menu selections to appear like switch events. By doing this it is possible to change the interface from switches to menus very easily, allowing experimentation to find the most suitable method. Making a switch and a menu selection generate the same value leaves this choice to the user when the program is running.

The client coroutine is responsible for dealing in an appropriate manner with the help mode when the user selects it. This mode is indicated by a global flag that is set by the window manager in response to the user pressing the help key. No particular behaviour is enforced but it is expected that the coroutine will put out messages explaining the results of an action, rather than carrying out the action. As will be described later the window manager provides routines for common activities such as pop-up menus which automatically handle help information and so relieve the client of some of this responsibility.

The window manager provides routines for the client coroutine to change the appearance of the cursor or to turn it off within its window. The procedure call:

```
Cursor.setShape( pad )
```

causes the given pad to be used as the cursor within this window. If the argument is Nil then the default system cursor is used. As well as the default cursor the manager also makes use of an hour-glass symbol when there is some sort of delay and input is not being responded to immediately. This symbol is available to the client as Cursor.WaitIcon and can be used when the client coroutine is carrying out some lengthy process that holds up input handling. The calls:

```
Cursor.turnOff()
Cursor.turnOn()
```

have the expected effects. If the cursor has been turned off the client should be providing some other form of feedback as coordinates change. This can be done by setting up a routine to be called by the window manager whenever a coordinate event occurs with this window current. Such a routine can be set up whether the cursor is turned on or off. The routine is set up by the following calls:

```
Cursor.setFn( procedure )
Cursor.setArg( argument )
```

```
// The main procedure ////////////////////////////////////////////////////

LET trails( cluster ) BE
$( // Initialise the cluster structure for this window
   LET pad = CN.Create( IsPad, width, height, 1, 0,0 )
   LET arc = CN.Insert( pad, cluster, 0,0, 2 )
   LET region = LU.getRegion( 1 )
   LET lookupTable = TABLE white, red
   LET packet = ?

   LU.setRegion( region, lookupTable ) // load lookup region
   CN.setLURegion( pad, region )       // set pad to use this region
   CN.display()                        // display it

   // The central loop for processing switch events ////////////////////
   $( packet := CoWait( Nil )          // wait for packet

      SWITCHON packet!pkt.switchNo INTO
      $( CASE 1:                       // this is switch number one
            TEST packet!pkt.goingDown  // is it going down?
            THEN                       // yes - follow the cursor
            $( Cursor.setFn( drawTrail )
               Cursor.setArg( pad )
            $)
            ELSE                       // no - stop following
            $( Cursor.setFn( Nil )
               Cursor.setArg( Nil )
            $)
         ENDCASE

         CASE 2:                       // this is switch number two
            IF CoWait( Window.Kill )   // did she really mean it?
            THEN BREAK                 // yes - close down the window
         ENDCASE

         DEFAULT:                      // any other switch
            W.unusedSwitch( packet )   // we can't do anything with it
      $)
   $) REPEAT ////////////////////////////////////////////////////////////

   // Terminate the cluster structure
   CN.remove( arc )
   LU.freeRegion( region )
   CN.destroy( pad )
   CN.display()
$)


// The cursor function ///////////////////////////////////////////////////

AND drawTrail( packet, pad ) BE
   CN.writePixel( pad, packet!pkt.x, packet!pkt.y, 1, allPlanes )
                                       // set a pixel at cursor position
```

Figure 4.6
A simple procedure suitable to be used as a coroutine within a window

The window manager calls the procedure whenever a coordinate event packet arrives, passing in the packet and the client's argument:

```
procedure( packet, argument )
```

packet contains the x and y coordinates relative to the origin of this window. argument can be a vector of values, allowing multiple values to be passed from the client coroutine. The contents of the packet can be changed by the routine, for example the type can be changed to switch data and a switch number stored in the data field. Provided the packet has been marked "internal" by the routine, the window manager will detect this and use the updated packet instead of reading a new packet from the channel. Hence particular coordinate changes, for example the crossing of a boundary within the window, can be made to appear as switch events to the client coroutine.

The client coroutine can request the cursor coordinates at any time:

```
Cursor.read( coords )
```

coords is a two-word vector in which the x and y values relative to the window origin will be placed. The client never has any knowledge of the window's actual position or size on the screen. It is left to the user to set up the screen so that he can see the things that are important to him.

A simple client coroutine will serve to illustrate the preceding discussion. Its window contains a single pad and when switch number 1 is pressed down a trail of dots is drawn in this pad as the cursor moves. The trail finishes when the switch is released. The window is closed down by pressing switch number 2 and all other switch events are ignored. The trail is drawn by setting up a cursor function whose argument is the pad to be drawn in. A dot is drawn in the pad for every coordinate received. If the cursor moves fast there may be gaps in the trail as coordinates are not buffered up. The code to achieve this is shown in figure 4.6. A continuous trail could be produced by keeping a record of the previous coordinates and drawing a line joining the previous position to the current position.

Extra facilities for the client program

A number of procedures are provided by the window manager for the use of the client coroutines. These implement a variety of common actions to encourage consistency and to reduce the time needed to implement simple applications. The use of these procedures also means that the way a particular action is carried out can be changed easily, for example to produce a different user interface.

The client can add an extra line of text to the system-provided tag on the window. This can be used to give more information about what is happening in a particular window:

```
W.createTag( string )
```

This is especially useful when the window is closed down to just its tag.

A procedure is provided for displaying messages in a standard form:

```
W.message( string, arg1, arg2, ... )
```

An arbitrary number of arguments can be provided for substitution into the string, as in the BCPL formatted write statement (WriteF). Currently these messages are displayed on a VDU communicating with Rainbow round the ring but it is intended that they should eventually appear on the Rainbow display itself, either in a single message window or in areas associated with each window. A heading line is automatically provided to identify the window that originated a particular message. The use of the VDU meant that existing software could be used to get this facility working immediately.

Another procedure:

```
W.unusedSwitch( packet )
```

can be used as the default action for switch events that do not have any function associated with them, as in the example program given above. A message is displayed and the cursor is flashed so that the user knows that the switch has been received.

A problem that arises with the use of coroutines is that when a long calculation is in progress all input is held up. Although there is no sensible way to handle general input while a process like this is running it is useful for the user to be able to stop the process, especially if it was started by accident. A procedure, `W.cancelled`, is provided that can be called from time to time within a lengthy procedure to check for the user having hit the “cancel” key. If it returns the value `TRUE` then the procedure should terminate. This routine reads any packets on the input channel and calls `W.unusedSwitch` until there are no more packets, or a packet for the “cancel” key is found. This has the unfortunate side effect that normally acceptable key strokes will be thrown away. The client can display `Cursor.waitIcon` when the user must wait and this coupled with the feedback of the cursor flashing when a key stroke is received makes this side effect less confusing. This is not a totally acceptable way of handling long calculations but the assumption is that this will not happen very often.

A set of procedures are provided to implement channels between windows to enable data to be sent in response to requests from other windows. The client program can create a channel that is available to a number of windows within the system:

```
channel := W.createChannel()
```

This channel is a list of packets waiting for responses (these packets have no connection with the packets that are received from the tools handler). The packet fields are:

```
1. pointer to the next packet on this channel
2. type - either "send" or "receive"
3. pointer to the coroutine that sent this packet
4. a flag indicating if the request was satisfied
5. data to send
6. result
```

Before making a request the client must set up the type field and provide the data to be sent. The type does not in fact indicate any directionality but is used for pairing corresponding packets. When a request is made:

```
result := W.makeRequest( packet, channel )
```

the channel handler searches the list of packets on the channel for a packet of the opposite type. If one is found then the data field of both packets is copied to the result field of the other and the client's call to the request routine returns. Obviously the packet found waiting on the channel corresponds to an earlier request where a packet of the opposite type was not found. The coroutine that made this request will have been suspended and must now be released. This is done by the coroutine that made the second request passing back the result of the request to the window manager. The result is a request to send an internal packet to the suspended coroutine, thus releasing it. The suspended coroutine's call to `W.makeRequest` now returns with the result `Nil`. For the channels to function correctly a coroutine must always pass any non-`Nil`result from `W.makeRequest` back to the manager. A suspended coroutine can also be released by hitting the cancel key with the cursor within its window. The client can determine whether the request was cancelled or satisfied by the value of the flag in the request packet. The cursor is set to the "wait" icon in a window that is suspended waiting for a request.

A common form of menu that the client may make use of is a pop-up menu consisting of a single column of textual items. The window manager provides a routine for creating such a menu, based on the menu routines described in the previous section. The client provides a string of characters to be displayed for each item and the value to be returned if the item is selected. The client can also provide text describing the result of picking each item, which is then used for helping the user.

```
menu := W.menuCreate( itemsVec, helpVec, priority )
```

`itemsVec` contains the information about each item in the menu with word 0 giving the number of items. `helpVec` is a vector of strings giving the text for helping the user. The menu is created with a standard lookup table giving a yellow background, black text and a red box drawn round the selected item. Because the menu only appears for a short time these bright, eye-catching colours were chosen in preference to gentler colours. This use of a standard colour is very important in reinforcing the user's (appropriate) feeling that all these pop-up menus behave in the same way. Once the menu is created it is inserted into the current window's cluster at the given priority, with the arc turned off.

These menus are assumed to appear when the middle mouse button (switch number 2) is depressed. The procedure:

```
result := W.menuSelect( menu )
```

should be called by the client coroutine when this event is detected. The menu is then displayed in the centre of the window and a cursor function is set up which highlights the item under the cursor whenever it moves. If the cursor is moved out of the menu then no item is highlighted. The routine returns the value associated with the item under the cursor and removes the menu from display when the middle mouse button is released. If the system is in help mode then the text provided by the client is written out and `Nil` is returned as the result when the user makes a menu selection.

Standard routines for handling forms are also provided, again based on the software described in the previous section. A form is created by the call:

```
form := W.formCreate( itemsVec, helpVec, title, priority )
```

itemsVec and title are passed straight through to the general form creation procedure, along with a standard lookup table region. This gives a pale blue background and dark green text, a combination that gives text that is easy to read and is gentle on the eyes. helpVec is a vector of strings describing the fields of the form and is used to provide help to the user. Once created, the form is inserted into the current window's cluster at the given priority with the arc turned off.

When the client wishes the form to be filled in the procedure:

```
result := W.formFill( form, x,y )
```

is called. The form is then made visible at the given position within the current window. This procedure handles all input, passing all ASCII switches and cursor control keys through to Form.call. When either the cancel key or carriage return is received the form is removed from display and the procedure returns, with the result being FALSE if cancel was hit and TRUE for carriage return. All other switch events are passed to W.unusedSwitch. A special cursor is displayed whilst control is within W.formFill so that the user realises that only form filling can be done. When the system is in help mode then any attempt to fill the form will instead result in the help information for the current field being displayed. When the client has finished with the information in the result vectors the form can be cleared by the call:

```
W.formClear( form )
```

This sets all the result vectors back to empty strings.

Inside the window manager

The window manager provides the interface between the input tools manipulated by the user and the client coroutines. The tools are handled by the asynchronous tools task described in the previous chapter. This provides a channel from which information about input events can be read by the window manager task. The key table has been set up so that three special keys on the keyboard return the same code values as the mouse buttons. The tablet stylus switch also returns the same value as the left mouse button and so the user can choose whichever device is most suitable at a given time. The tools handler buffers up switch actions but only retains the latest coordinate information. This means that if the window manager is not able to read from the channel for a while no switch actions are lost but the cursor position may jump because intermediate positions will have been lost. The coordinate devices are set up to return packets whenever the coordinates change, rather than after a timeout. Packets can also be passed to the manager from the client coroutines. These packets are referred to as "internal packets" and are passed on to the window specified by the originating coroutine. This allows different windows to communicate with one another.

The window manager always maintains a pointer to the current window and it is this window which receives switch packets and whose cursor function is called when a coordinate packet arrives. The current window is essentially the window in which the cursor is located. In fact a new current window is only found when a switch is pressed so that the search is not done every time a coordinate packet arrives. Hence, the cursor shape and procedure associated with a particular window are only used once a switch has been pressed with the cursor in the window. When a window is made current it is given the highest priority so that it overlays all the other windows.

After initialisation the manager enters a loop where a packet is read from the channel and then processed. The following is a summary of the main actions taken with the packets:

The packet contains coordinate data:

The cursor is moved to the new position. If the manager has provided a cursor function then this is called, otherwise if the client has provided a cursor function for the current window then this is called. Similarly the shape of the cursor is changed if the manager or current window have provided a new shape.

The packet contains switch data:

If the switch is going down, then the window in which the cursor is positioned is made current. If the switch is the right hand mouse button then window management is started, otherwise the coroutine for the current window is called with the packet as argument.
If the switch is going up a new current window is not found. The right hand mouse button terminates window management, and all other switches are passed to the coroutine of the previous current window. The result returned when the coroutine suspends itself determines what happens next.

The packet is an internal packet:

The window specified by the window that originated the packet is made current and its coroutine is called with the packet as argument. As above, the result returned determines what happens next.

In general when the coroutine returns the loop continues with the next packet being read. Exceptions to this occur when a special result is passed back to the manager by one of the coroutines. Two cases are dealt with at the moment, a request to kill the current window and a request to send an internal packet.

When a request is made to kill the current window the window manager makes it invisible and records that it is ready to be destroyed. The window is not destroyed immediately so that the user can cancel the request if it was a mistake. If the next switch event is the cancel key then the value `FALSE` is passed back to the coroutine that made the request. The window is redisplayed and the coroutine can continue as if nothing had happened. If any other switch event occurs then the coroutine is called with the value `TRUE`, indicating that it should go ahead and tidy up its display and data structures. When the coroutine returns the window manager destroys it and its associated window. The manager than proceeds to process the switch packet as usual.

If a request is made to send an internal packet then the type field of the packet is marked “internal”, rather than “switches” or “coordinates”. The window manager continues, using this internal packet rather than reading a new one from the channel. A cursor procedure can also cause an internal packet to be sent by returning the appropriate value. In both cases the global variable `W.result2` is assumed to contain the identifier of the recipient window.

Window management is carried out by setting up a system cursor function when the right hand mouse switch goes down. This function adjusts the size or position of the window as the cursor moves and fixes the window when the switch is released.

The function is selected on the basis of what part of the window the cursor was in when the switch was depressed, for example in the top right corner the top and right edges move to change the size and in the middle the whole window moves without changing size.

It should be noted that a new current window is not found when a switch is going up rather than down. The "up" events are not ignored altogether because it is a useful interaction technique to have a continuous function carried out for as long as a switch is depressed, rather than requiring a second switch depression to terminate it. If a new window was found for switch up events then the up event could be sent to a different window if the user had moved the cursor out of the original window. This would mean that the user must move the cursor into the window again with the switch depressed in order to get sensible behaviour. As a result it was decided that it would make the system easier to use if switch up events always went to the same window as the corresponding down events.

A number of special types of data packet are intercepted by the window manager, rather than being passed down to the client coroutine. The trapping of the third mouse button for window management has already been mentioned. If the switch data is for the key labelled "help" on the keyboard, then the value of a global flag is inverted to swap the system in and out of help mode. The "home" key causes the cursor to be moved to the middle of the screen and resets the standard appearance. This is provided because it can be easy to loose sight of the cursor when there is a lot of detail on the screen. The cancel key is passed through to the client coroutine unless a window is waiting to be destroyed.

When the window manager starts up, a background window is created that covers the whole screen. The associated coroutine provides a pop-up menu on the middle mouse button that lists all the types of windows that the client programmer has specified. When the user selects an item from this menu the appropriate window is created. There is a certain amount of delay while the manager creates the window data structures and also while the client coroutine is carrying out its initialisation. In order to reassure the user the "wait" cursor icon is displayed during this pause. In help mode the selection of an item from this menu causes a piece of text provided by the client to be displayed. This text is expected to describe the window that would be created.

Data structures

The state of the window manager is represented by a list of all the windows currently in existence, with a global variable pointing to the current window. There are also global variables indicating whether the system is in help state or not, and pointers to the current window manager cursor state.

Each window that exists has a window descriptor, which is a vector containing the following information about the window:

```
1. A pointer to the window's display structures
2. Its position on screen
3. Its size
4. Its priority
5. The client-specified maximum size
6. A pointer to its cursor state
7. A pointer to its coroutine
```

The display structures used for a window are described in the next section. The cursor state, both for the window manager and for individual windows, is represented by four items of information:

```
1. The cursor procedure
2. An argument for the procedure
3. A flag indicating whether the cursor is to be displayed
4. A pointer to the pad to be used as cursor
```

The window list is the central data structure and consists of a singly-linked list of window descriptors. This list is maintained in priority order with the highest priority window first. In order to find the current window it is a simple matter of scanning down the list comparing the cursor coordinates with each window's position and size until a "hit" occurs. The last item on the list is the background window which covers the whole screen so there is always a hit. The window list is reordered whenever a new window is given highest priority. This is simply a case of moving one item in the list to the beginning and so does not take any significant amount of time. The consequent gain in speed of finding the current window is very important because the list is searched every time a switch event happens.

Display structures

Each window in the system is represented by two levels of cluster, with the top cluster mapped into the screen. Two levels are required for arbitrary clipping because although the size of a cluster can be changed it is not simple to move the origin with respect to objects mapped into the cluster. These effects are best explained by the diagrams in figure 4.7. The client coroutine is handed the lower level cluster and can map any display objects into it. Clipping is handled completely within the window manager and the client has no need to know what is happening. The window is moved around by moving the top arc relative to the screen and clipping is done by moving the two clusters relative to each other. The client specified maximum size determines the upper limit and the manager prevents the window shrinking below its tag size, including the client-provided tag if one exists. The manager also prevents windows moving off the screen although the display software would allow this. This is to prevent windows getting into a position where they cannot be retrieved. In fact this effect could also be achieved by preventing just the tag from moving off screen.

The background of each window is of a fixed colour and so is implemented as a "virtual pad". These have an extent and colour but no associated raster so as to save on graphics memory. All windows of a given type share the same background pad and lookup table which produces savings in lookup table entries. In fact sharing of lookup tables occurs extensively throughout the system as lookup table entries have been found to be a scarce resource. Many things, such as all standard menus and forms as well as window tags, share lookup tables. This sharing can also be done in the client coroutines. The lookup table is stored in a static variable, which is a fixed location in the single instance of the code for each coroutine. A static variable is defined with an initial value and so the coroutine can test the value of this variable to see if the initial value has been changed. If not, then this is the first time the coroutine code has been called and the lookup table can be created and stored in the static variable. Later, independent calls to the coroutine will see this changed value and be able to make use of the ready-created lookup table.

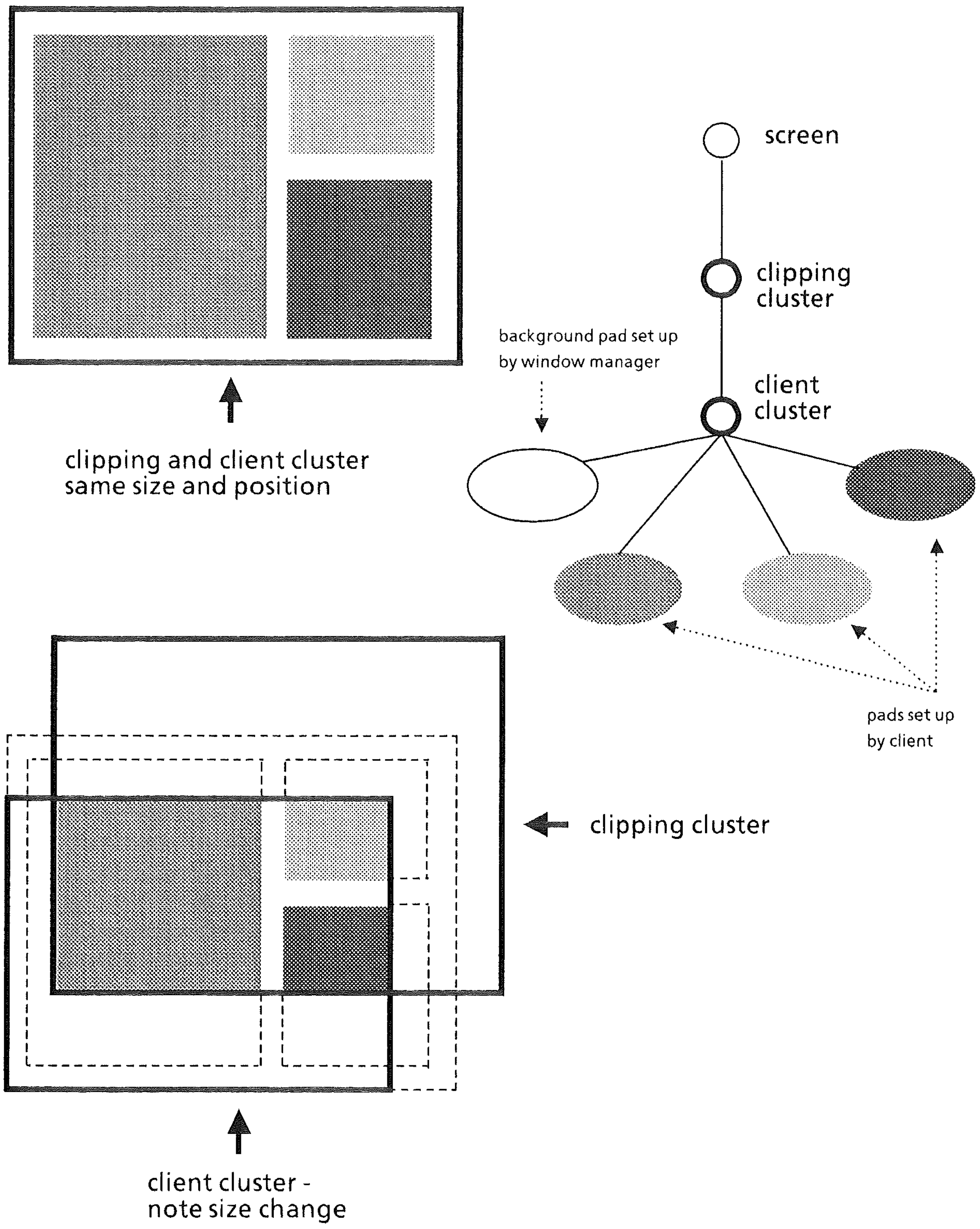

screen
clipping cluster
background pad set up by window manager
client cluster
pads set up by client
clipping and client cluster same size and position
clipping cluster
client cluster - note size change

Figure 4.7
Cluster structure of a window

The cursor is a small pad mapped into the screen node. Ideally it should have a transparent background to avoid obscuring the objects it passes over. Unfortunately with the Rainbow hardware and software this can only be done by allocating a whole plane of the graphics memory to the cursor so that it is always in a plane disjoint from every other object on the screen. This would be very wasteful as it would take one eighth of the total memory available to represent one very small object. Hence, a transparent cursor is not used and the standard system cursor is a small black square with a white centre which obscures whatever it passes over. If a particular window requires a transparent cursor then it is possible for it to provide its own cursor pad in a suitable plane. This cursor would only be used within this window and so there is no need for it to be transparent to the other windows.

5. Imp

Proposals for a new interactive system for type face design were made at the end of chapter two. Now that the environment for Imp has been presented we can expand on the proposals and describe what has actually been implemented. This chapter provides a brief overview of Imp as an introduction to the details of the following chapters. The first section below summarises the types of data and the programs that make up Imp. The second section contains some general observations about the way the window manager is used in Imp. A brief summary of Imp has been published in [Carter, 1984].

5.1 Overview

Imp is a type face design system that provides a framework for experimenting with interactive techniques. The emphasis is on unconstrained, non-algorithmic working which leaves the designer free to follow his imagination. More abstract methods of working can be introduced in such an environment to gradually extend the methods the designer is familiar with. Ultimately it is the designer's visual judgement that decides whether the results are acceptable and he must not be limited by complex algorithms that do not quite do what is required.

The final product of Imp is rasterized type faces for use on digital type setting equipment. In a manner analogous to designing for metal type these raster founts are derived from master designs rather than being designed directly in their final form. With metal type, subtle adjustments may be made to the design as it is being cut to take account of the nature of the metal type. Similarly, fine adjustments can be made to the characters in a raster fount to cope with problems arising from the conversion process.

Imp builds on the idea of there being several levels at which a type face design can be considered. The designer is able to work on fine details of a master character whilst seeing the details in the context of the whole character, and this character in association with others. Imp is made up of a series of programs that deal with these different aspects of the design process. Each program is implemented as a coroutine associated with a window, running under the window manager. Underlying Imp is a fount database which provides for the storage and retrieval of founts. When Imp is running, special fount windows provide a graphical interface to this database.

Master Founts

A master fount is the original design from which raster founts can be derived. There may be a single design for all sizes of the fount with it simply being scaled to fit. More likely the designer will wish to produce different designs for large and small sizes to compensate for various optical effects. When a metal type is cut the large sizes tend to be narrower and lighter than the small sizes to give a similar overall appearance. Similar results can be achieved with digital type by providing a number of master designs for different size ranges. There is also the possibility of interpolating between the different designs to produce genuinely intermediate designs.

The master founts are stored as line chain outlines rather than as filled areas defined by a large bit map. Outlines are more easily manipulated, for example by applying transforms such as rotation, and also take less space for storage. The outlines could have been encoded as splines in order to save further space and to reduce the risk of characters appearing polygonal at large sizes. As was discussed in chapter two, it is hard to manipulate a spline representation interactively and so this was rejected for the storage of outlines. Even so, splines can be used to generate the line chains if the designer wishes.

No attempt has been made to add further structure to the line chains, for example to identify serifs and strokes. Instead a variety of tools has been provided to make it easier to produce a set of related letters, in a manner similar to working at a drawing board. Grids can be set up to ensure that characters are the correct size and a measuring gauge is provided so that proportions can be checked. Pieces of line chain can be copied from one character to another allowing, for example, a standard serif to be designed and spliced into a series of characters. Each instance of the serif is independent of the others so that one can be changed without affecting the others. This allows fine adjustments to be made to fit the serif onto different shapes. On the other hand we have lost the ability to experiment rapidly with a variety of serifs as each one must be changed individually. Any point in a line chain can be repositioned by hand at any time and also groups of points can be moved by scaling or translation. The particular combination of features for manipulating the master outlines has been a constant source of experimentation and further details are provided in a later chapter.

If the whole design process is to be carried out on a computer then the early stages of playing around with rough ideas must be supported. It is unlikely that the designer will wish to commit himself to exact outlines from the beginning and so a rough sketching facility is provided as part of the outline editor program. This provides an edged pen of variable width and angle which can be used to draw calligraphic letters or other shapes. These sketches are completely unstructured and are just used as guides for producing outlines.

The characters being designed do not exist in isolation and so a program is provided which allows them to be viewed together. The interactions between the shapes of the characters can be examined and the spacing can be set up. This is as important ultimately to the appearance of a page as the shapes themselves. The spacing can be calculated automatically but in line with the overall philosophy it can be adjusted by hand.

Raster Founts

The raster founts are generated automatically by scan conversion from master founts. Different raster founts must be generated for devices of different resolutions and for different output sizes. Hence the fount database may contain many raster founts derived from one master.

Currently a fully successful scan conversion algorithm does not exist and so the raster founts need some editing before they can be used. For example, depending on where on the grid an outline registers a particular stroke may have a different width. Different characters containing this same stroke may have registered differently and so the uneveness must be corrected by hand. For a device such as a graphics screen which supports grey scale the characters may be anti-aliased by super-sampling.

These grey scaled characters can also be edited by hand if necessary.

For simplicity no attempt has been made to save space by using any complex encoding of the rasters. Run-length encoding can be used for raster founts stored on disc but generally the founts are stored simply as runs of pixel values for each scan line. In the workstation memory characters are stored as arrays of pixel values as it is easiest to interact with the data in this form.

A further aid to speed the design of type faces is a program that displays a page of text on the computer workstation screen. Obviously the final test for a type face is how it looks when used for a whole page. By using super-sampling techniques, described later, it is possible to display a page of text on the screen at the same size as it would appear on paper when printed. This can give a reasonable idea of the appearance of a page, although the final judgement can only be made when it is printed on paper.

5.2 Imp and the window manager

The overall organisation of Imp is in the hands of the window manager and this section discusses points that apply to all the programs making up Imp.

Conventions for interaction

The window management software makes few demands on the client to use particular actions for particular effects. The use of the third mouse button for window management, the help key and some uses of the cancel key are the only times that a particular effect is enforced. There are some procedures which, if used, enforce consistency in some areas but it is always possible to do things differently. The best user interfaces stick to certain conventions so that related effects in different contexts are evoked by suitably similar actions. This section outlines the conventions adopted in Imp.

One fundamental action is to point at something on the screen, for example to indicate that a function is to be applied to it. This is initiated by moving the cursor to the object and then depressing switch number 1. (In fact, switch number 1 is one of three switches, either the left mouse button, the switch in the tip of the tablet stylus or the left of three special keys on the keyboard.) The final selection is not made until the switch is released and moving the cursor around with the switch depressed causes different objects to be indicated. This makes it easier to select objects accurately. If some sort of continuous function is to be applied to the object, for example moving it to a new position, then this function is called whenever the cursor moves with the switch depressed to give constant feedback of the effect. In the description of Imp programs any mention of pointing, selecting or dragging refers to this particular action with continuous feedback.

Switch number 2 (the middle mouse button or the middle one of the three special keyboard keys) is used for pop-up menus. As with the use of switch number 1 depressing it initiates the action and releasing confirms the particular selection, with continuous feedback in between. In fact the menu routine provided by the window manager helps to enforce this particular convention as it makes a selection when switch number 2 is released. The client program would be expected to call this routine when the switch was depressed.

Another sort of menu used in Imp is one that is displayed continuously in a window. These menus are used for selecting a command that is then applied by pointing at objects. The command can be applied repeatedly without having to reselect it in the menu. Selection is done using switch number 1 to point at the appropriate item in the menu.

Some attempt has been made to use the cancel key consistently within Imp but the meaning of cancelling some actions is not totally clear. In general pressing the cancel key sets the system back to the state it was in before the latest menu selection. Hitting cancel again restores the state that was cancelled so that the user can switch repeatedly between the current and previous state.

Program structure

The fact that the different programs making up Imp are implemented as coroutines has certain consequences for their structure, as does the use of BCPL. Coroutines running within the same Tripos task share the same vector of global variables and so communication between them is simple. The piece of code that sets up the window descriptors and calls the window manager also initialises any shared globals. The two main global variables used in Imp provide lists of the master founts and raster founts that are in memory. Another two provide channels for synchronized communication with the fount windows. The rest of the variables are local to a particular coroutine.

The ideal structure of a coroutine would be to declare all the variables defining the local state and all the procedures that act on this state at the start of the code for the coroutine, for example:

```
LET coroutine( cluster ) BE
$( LET var1 = ?
   LET var2 = ?
     ...
   LET proc1() BE
   $( ...
      var1 := ...
      ...
   $)
   LET proc2() BE
   $( ...
   $)
   ...

   pkt := CoWait( Nil )
   $( SWITCHON pkt!pkt.switchNo INTO
      $( CASE 1: proc1()
                 ENDCASE
         CASE 2: proc2()
                 ENDCASE
         ...
      $)
      pkt := CoWait( result )
   $) REPEAT
$)
```

The local state may be made up of large numbers of variables, any of which might be updated from within the local procedures. Unfortunately, these variables are dynamic free variables with respect to the procedures and BCPL does not support access to them. A solution for non-reentrant code is to use static variables. These have a single fixed location with respect to the code and they can be accessed from anywhere within the block, including from within procedures. This is obviously inappropriate here as there may be several instances of the coroutine active at once and they would all share the same static locations.

One solution is to pass the addresses of the variables needed as arguments to the procedure, so that they can then be accessed indirectly. The problem is that this becomes very cumbersome when large numbers of variables are involved. It can also be a source of confusion if some variables are passed by address and others by value. Another solution is to incorporate the code of the procedure into the body of the coroutine, in place of the procedure call, and thus remove the need for dynamic free variables. This has been done in some places but the problem is that it begins to obscure the structure of the program and makes it hard to handle and debug.

The solution that has been most widely used in the Imp programs is to create a vector to contain all the local variables. Instead of accessing a variable directly, a manifest constant is used to index the vector. The address of this vector can be passed as an argument to the procedures, thus giving them access to all the local variables. This introduces an extra level of indirection in the main body of the coroutine but the resulting consistency throughout is a great advantage.

Although static variables are not suitable for storing the local state of a particular instance of a coroutine, they can be used for communication between the different instances of this coroutine. If the static variable is updated by one instance then all the other instances will see the change. This fact is exploited to allow the sharing of lookup table regions between instances of a coroutine. They will all be using the same lookup table and so it would be a waste of space for each instance to create a separate lookup table. The first instantiation of the coroutine creates the lookup table and then places its address in a static variable. Further instantiations of this coroutine will see the updated static variable and can then make use of the same lookup table. A static variable is initialised to a specific value in its declaration. By using a value that cannot possibly be a lookup table address a particular instance of the coroutine can tell whether or not the lookup table has yet been created.

The actions of a coroutine are determined by the switch numbers in the packets it receives. The actions are selected by using a SWITCHON command into a set of CASE statements labelled with switch numbers. These labels are usually manifest constants rather than explicit numbers so that by changing the values of the manifests the user interface can be changed. Switch event packets can either be generated by physical switch events or they can be generated internally. In the former case the switch number is that bound to the physical switch, usually a small integer. In the latter case the switch number field can contain any value that the client program places there. In Imp this value is usually the result of a menu selection which has been set up to generate an internal event. Hence a number of schemes are possible for evoking a particular action. By giving the manifest constant a value that is not associated with any switch the action can only be evoked by some internally generated event such as a menu selection. If the value can be generated by a switch then the action can be evoked by pressing the appropriate switch. There is no reason why this value should not also be generated by a menu selection, giving the user a choice. The actual physical switch used for an action can be changed by changing the value of the manifest.

Generally menu selections are used to generate internal events rather than evoking actions directly, in order to maintain a flexible user interface. On the other hand, on a few occasions a different approach has been taken. In these cases the result of a menu selection is assumed to be the address of a piece of code. This is called directly to carry out the action. Generally this has been used only when forced by the difficulties of constructing a large and complex coroutine spreading over several separately compiled sections.

When data is to be passed from one program to another, for example from a fount window to an editor, it is sent along one of the channels mentioned earlier. One channel is used for master fount data and the other for raster founts. Two actions are always required, one to send data from one end and the other to fetch the data. These two actions can be done in either order and the program making the first request is suspended until the second has been made.

Other features

The use of colour is an important feature of Imp which is unfortunately not conveyed by the monochrome illustrations in this thesis. The different windows making up Imp are distinguished by different coloured backgrounds. This is a valuable cue as the window tags themselves are quite small and not rapidly distinguished. It does not take much use of the system for the colour, along with the layout of the window contents, to be adequate to enable the user to find a particular window. The background colours chosen are all soft pastel colours as these are easier on the eyes than bright colours. Originally the colours used for drawing lines and text were chosen to differ from the background colours only in hue, not brightness. This reduces the perceived flicker of the lines considerably but they can be rather hard to see. Currently large adjacent areas tend to use colours of similar brightness but details that must be focussed on are displayed in more distinctive colours. The colours actually used are a somewhat adhoc choice that seem to work adequately and there is room for more experimentation here.

Another important feature, resulting from the structure of the window manager, is the availability of different input tools simultaneously. On many systems there is a choice of either mouse or tablet, and then a stylus or puck for use on the tablet. These devices are each suitable for different operations, all of which are found in Imp. By having both a mouse and tablet available the user can choose which is most suitable at a given time. This might even be a different tool for the same task at different times and almost certainly different users will have different preferences. There is also the possibility of incorporating other input tools later.

6. The Fount Database

Any fount design system needs facilities for the storage and retrieval of founts. When Imp is running a small number of founts can be stored in the workstation memory for immediate access by the various programs that make up Imp. Other founts that are not currently being worked on must be stored on disc and when Imp is closed down the founts stored in the workstation must also be written out to disc. The term "fount database" is used to describe this collection of fount data although this is really rather a grandiose name for it. The name is used as it is the most convenient way of referring to this "base of data". This chapter discusses the data that must be stored to describe a fount, the formats used in workstation memory and on disc, and the graphical interface from within Imp.

6.1 Fount Data

The fount database contains both master founts and the raster founts derived from the masters, along with the spacetables needed to set the founts. The master founts consist of high resolution outlines defining the character shapes. They are device-independent designs from which the resolution-dependent raster founts are derived. Raster founts from other sources may also be included in the database, in which case a corresponding master design will not exist. The space tables are kept separately from the character shapes because the same fount may be spaced in different ways. Keeping the spacing information separately makes experimentation with different spacing easier. When a fount is to be set by some device there must be a way of relating the character names used in the fount with the character codes used by the device. This information is provided by code tables, which map names to codes and vice versa. Any mapping can be set up, allowing the use of founts that do not fit a standard coding.

Fount names are composed of names describing the family and style, followed by the size, device resolution and bits per pixel when appropriate. Family and style names may be set up in any way that the user wishes, although it might be assumed that founts of the same family can be used together. Typical names might be family Times, style Roman or family Times, style Italic, with the implication that the italic fits with that roman. Character names can also be anything that the user chooses, provided a code table is set up to map the names to numeric codes. For standard roman alphabets forms such as Uppercase-A or Lowercase-b are generally used.

Master Founts

A letter in a master fount is represented by a set of line chains defining the outline of the letter. Each line chain ("strand") is a series of points connected by straight lines, with the last point connected to the first to give an outline that is always closed. As was discussed at the end of chapter two, this outline representation was chosen because it is easy to interact with. The use of splines, for example, could reduce the amount of data stored but at the expense of unexpected effects when points are moved. The strands of a letter are collected into "parts", the parts of a letter being scan converted separately and then superimposed to give the final form. This is done because the scan conversion algorithm does not cope correctly with

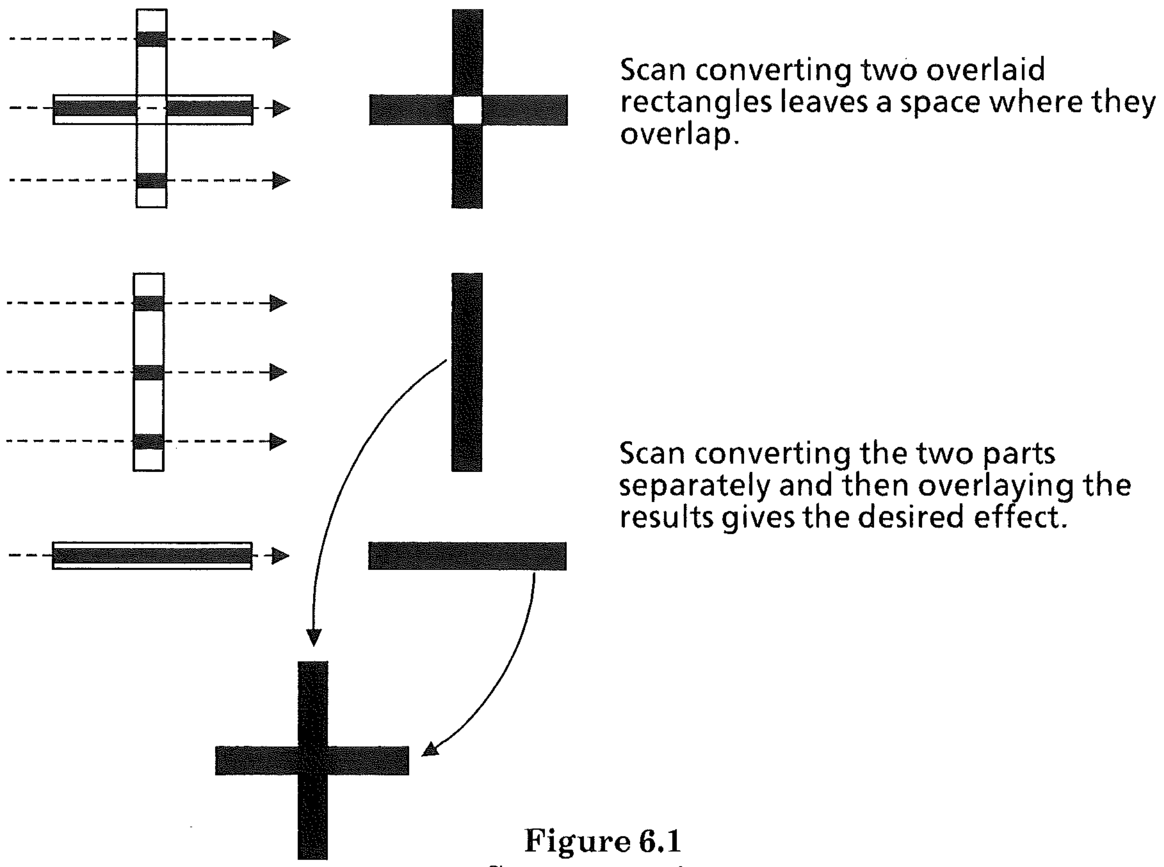
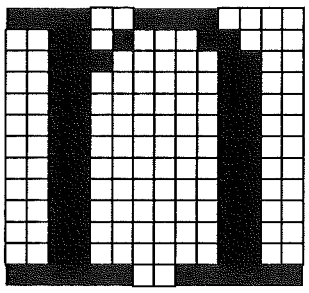

Scan converting two overlaid rectangles leaves a space where they overlap.
Scan converting the two parts separately and then overlaying the results gives the desired effect.

Figure 6.1
Scan conversion



Pixel encoding:	**Run-length encoding:**
0 10 1 1 1 1 0 0 1 1 1 1	0 4 2 4 0 0
2 9 1 1 0 1 0 0 0 1 1	2 2 1 1 3 2 0 0
2 10 1 1 1 0 0 0 0 0 1 1	2 3 5 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
2 10 1 1 0 0 0 0 0 0 1 1	2 2 6 2 0 0
0 14 1 1 1 1 1 1 0 0 1 1 1 1 1 1	0 6 2 6 0 0

Figure 6.2
This shows a typical bitmap, with below, the two encodings that can be used in the fount database.

overlapping strands such as would occur if a cross symbol was created by overlaying vertical and horizontal bars. Scan conversion is done by following across a scan line and changing colour every time a boundary is crossed. As can be seen in figure 6.1, overlapping areas are not filled in. The algorithm used is described in more detail in chapter eight. For most designs the use of different parts is unnecessary but it is a helpful aid when composite symbols are created.

The coordinates of points are stored to a scale which has 10000 units between the baseline and the cap-height. This allows sufficient resolution for high quality designs but keeps the numbers small enough to be stored in 16 bits. When letters are being designed from scratch in Imp the baseline and cap-height line are shown so that the letter shapes can be correctly related to the scale. If letters from outside sources do not have a cap-height defined then the letters are scaled so that the height of uppercase X is 10000 units. Letters are positioned on the coordinate grid so that the minimum x coordinate is 0. Imp will accept founts with a non-zero minimum x coordinate but will shift the outline so that the minimum x is 0. The y coordinate is taken to be 0 on the baseline so that the minimum y coordinate will be less than 0 for many characters. Letters are usually designed to be displayed at some specific size so this internal coordinate system must be related in some way to the outside world. Traditionally the size of type was given as the size of the piece of metal that formed the type and thus determined the minimum inter-line spacing. The size of the metal type was given in points and these units are still in general use for specifying the size of type faces. Obviously the master outlines do not of themselves have any fixed physical size, so the designer must provide the information along with the design. This is done by specifying the inter-line spacing in internal units and then quoting the point size of the fount. These two values are actually for the same measurement and so provide a direct conversion from internal units to the outside world.

Raster Founts

A letter in a raster fount is represented by an array of pixel values which can be displayed to give the letter shape. Each pixel value can be up to 8 bits to allow for anti-aliased founts, although values are unlikely to be more than 2 or 3 bits. A pixel with a value of zero is white and non-zero values are grey or black. The most straightforward way to store a raster fount is to store the array of pixel values as a succession of scan lines but considerable savings of space can be made by encoding the data in various ways. Two different encodings are used in the fount database, one which only stores the sequences of non-zero pixels and run-length encoding, which can reduce the storage needed even further. In both cases the data is presented as a sequence of scan lines starting from the maximum y value and going down to the minimum. If the data is presented as pixel sequences the first item for the scan line is the number of white pixels at the left hand end of the line, counting from the minimum x value. The next number gives the number of pixel values that follow and then the pixel values themselves are given. This sequence of pixels must span all the non-white pixels in that line. For a run-length encoded fount a scan line consists of a sequence of pairs of numbers, the first one being the number of white pixels and the second being the number of black. The line is terminated by a pair whose second value is 0, and a blank line would be represented by the pair 0 0. Obviously this encoding will not work for anti-aliased founts but for 1 bit deep founts it can compress the data considerably. Examples of these formats are illustrated in figure 6.2.

Space tables

As well as the shapes of characters a fount definition must provide information about how to place the characters in relation to one another. When type is cast in metal the size of the piece of metal determines the spacing in that the pieces are simply butted up against one another. In the best metal setting individual pairs of characters may be adjusted by hand to improve the spacing, but this is a time-consuming process. When a computer is used to control the setting of type these special cases can be spotted and dealt with automatically to produce high quality setting without large time penalties. Furthermore, when type is no longer in the form of physical pieces of metal new methods of spacing can be easily experimented with.

The spacetables used with both the master and raster founts have been designed to support any method of spacing without undue overheads in terms of space or calculations. The most commonly used method of spacing is based directly on metal setting, where each character has a fixed width. This may be defined in terms of the position of the character shape on the coordinate grid, along with a width value. In this case the next character is positioned with its origin of coordinates offset from the previous character by the width of the previous character. Another, effectively equivalent, way is to provide an amount of space to be left before the character and the amount after. These values would be measured from the minimum and maximum x coordinate in the character. Some letters, for example A and V or T and o, will not fit very well so special kern values may be provided that correct these difficult cases. Figure 6.3 illustrates these forms of spacing.

A completely different method of spacing is based on optical letter spacing [Kindersley and Wiseman, 1978] which attempts to position all pairs of characters so that they appear balanced to the observer. This is done by finding what may best be called a centre of gravity for each character and measuring the spacing from this point. The amount of space to be left on either side of a character is calculated from the total weight of the character. Parts of the character further from the centre contribute proportionately more to the weight than parts close to the centre. Hence a wide character will be given more space than a narrow character that has the same area of black. A letter such as A has most of its weight at the bottom and so will have a low centre of gravity whereas a letter like V will have a high centre. Because the spacing is measured from centre to centre rather than along the baseline, when this pair of letters is set together they automatically move closer. This effect of centres with different heights helps to space many difficult pairs correctly without the need for special kerning values. Figure 6.4 shows how this form of spacing works. It can be represented in a more conventional manner by providing fixed widths for each character and large numbers of kern values to reproduce the effect of the different centre heights.

The format of the files of spacing information stored in the fount database has been chosen to support optical spacing without large overheads. Optical spacing could be done by providing large numbers of kerning values along with conventional widths to be measured along the baseline. On the other hand, by allowing centres as well as widths to be defined for all characters the amount of space required to store

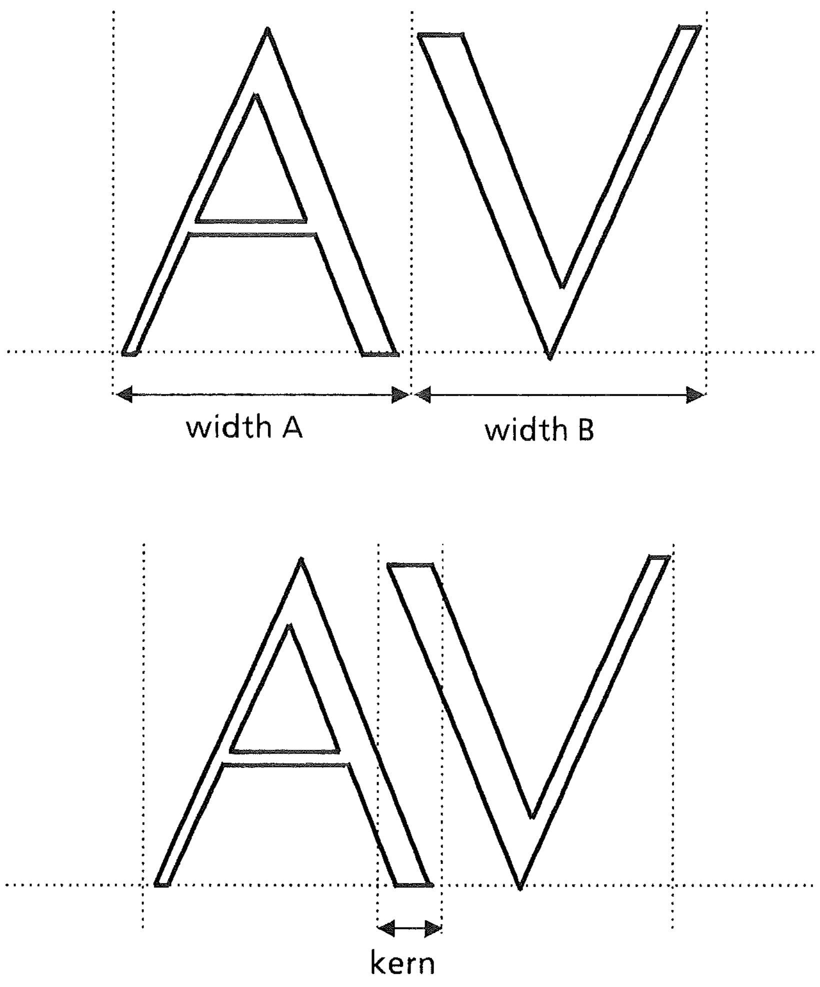


Figure 6.3
Typical "metal" spacing, showing the default results and the effect of subtracting a kern

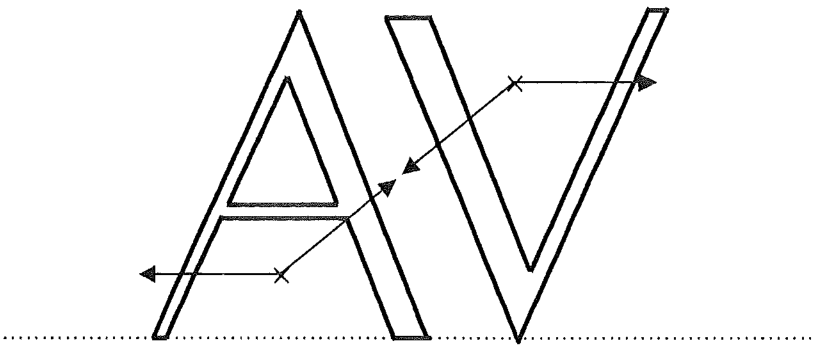


Figure 6.4
"Optical" spacing, with widths measured between the optical centres rather than along the baseline, giving the same effect as kerning in metal spacing.

a spacetable for optical spacing can be very dramatically reduced. Allowing centres to default to the origin of coordinates if they are not specified means that extra space is not required for spacing methods that do not use centres. Provision is also made for specifying adjustments to the spacing for particular pairs of letters.

Codetables

In the fount database characters are identified by names rather than numeric codes. Names are used as they are more meaningful to the designer producing the fount. They also provide more flexibility in that the fount is not limited to those characters that a specific coding scheme provides. Codetables provide these mappings from names to codes and vice versa.

6.2 File Formats

The files making up the fount database are ordinary text files. Although this is not the most compact representation, it does make the files easily transportable between machines as non-text files are often corrupted by transmission protocols. It also means that the files can be read, understood and edited directly. The data is stored as keyword followed by value so that all items in the file can be easily identified. This removes the need to maintain a strict ordering of items in the file. It also means that incorrect files are easily detected by program as they are being read. Comments, enclosed in square brackets, can be included in the file.

The keywords used all begin with the character `%` to distinguish them from other words which might occur in the file. The values can be strings, such as fount names, or numeric values, which must be integers. The string "NotSet" can be used in place of certain numeric values if a particular value is not wanted. For example, a master fount may have been designed to work successfully at any point size, in which case the size can be given as NotSet. Other founts designed for specific sizes will quote an actual numeric value for the size. Other values, for example the number of characters in the fount file, must always be specified exactly. When fractional values are required they are quoted as numerator and denominator so as to avoid problems with programming languages that cannot handle real numbers. Full details of the keywords and values that can occur in the different database files can be found in the appropriate sections below. The formats of the files and the procedures for lexical analysis described below are based on work done by John Wilkes [Wilkes, 1982].

Lexical analysis

A library of procedures is provided that can carry out lexical analysis of any text file. In addition, an initialisation procedure has been written that will set up this library for the analysis of fount database files.

The library recognises certain symbols by default, including numbers, names made up of sequences of certain characters and end-of-line and end-of-file characters. The client program can tailor the library by specifying the characters that make up names and by providing additional symbols. Also the characters that are ignored and those used to introduce comments can be changed by the client.

Each symbol that the library recognises has a type and a value associated with it. These are placed in the global variables `Lex.symb` and `Lex.value` respectively when the symbol is read by the call:

```
Lex.nextSymb()
```

For example, if the symbol is a number then **Lex.symb** takes the value **Lex.number** and **Lex.value** contains the actual value of the number. For names, **Lex.symb** takes the value **Lex.name** and **Lex.value** is a pointer to the string making up the name.

The client can add a symbol to the library by one of the following:

```
Lex.add( string, symbolType, value )
Lex.insert( string, symbolType, function, a,b...g )
```

In both cases, when the given string is read **Lex.symb** is set to **symbolType**. In the first case **Lex.value** takes on the given value but in the second case **Lex.value** takes the result of calling the given function with the arguments **a** to **g** (in BCPL these need not all be specified). **symbolType** can be any value the client wishes, including a built-in type.

Often a particular symbol is expected to be next in the input, for example a number after a keyword, and two procedures are provided to simplify the checking:

```
Lex.checkthis( n, symb, value )
Lex.checknext( n, symb, value )
```

The first checks the current symbol, that is the one defined by **Lex.symb** and **Lex.value**, whereas the second calls **Lex.nextSymb** before carrying out the check. If **n** is greater than 0 then **Lex.symb** is compared with **symb** and if **n** is greater than 1 then, in addition, **Lex.value** is compared with **value**. If **n** is 0 or less then no checking is done. If the symbol and value match as required then the procedure simply returns but if the match fails then the procedure **Lex.Error** is called with the expected and actual symbols as arguments. By default this procedure writes out a message to the terminal stating what was expected and what was actually found. It then terminates the program. More usefully, the client can provide a replacement procedure for **Lex.Error** which deals with the error in a manner appropriate to the particular program.

For the fount database files the keywords have been divided up into three different sorts of symbol: those that introduce sections, those that are attribute keywords and those that introduce actual data defining a character shape. Each individual keyword also has a special value associated with it to distinguish it from the other keywords of the same type. For example, the section keywords in a fount file are set up by:

```
Lex.add( "%Family", S.Section, V.Family )
Lex.add( "%Character", S.Section, V.Character )
```

Within a section various attribute keywords are found, for example:

```
Lex.add( "%Cap-height", S.Attribute, V.Cap.Height )
Lex.add( "%Xmin", S.Attribute, V.Xmin )
```

and the data keywords include:

```
Lex.add( "%Strand", S.Data, V.Strand )
Lex.add( "%Part", S.Data, V.Part )
```

All of these keywords are then followed by values that are either numbers or names.

Full details of these keywords and values can be found in the sections on the different types of file. The sections within a file are terminated by one of two keywords whose type is `S.End`:

```
Lex.add( "%End", S.End, V.End )
Lex.add( ".", S.End, V.End )
```

The client can choose whichever of the two keywords he prefers in a particular situation. In general `%End` is used to terminate a series of sections of the same type and `.` is used between sections of the same type. The decision has no significance in the lexical analysis of the files and so can be made on personal preference.

Fount keywords

The first section in a master or raster fount file, introduced by the section keyword `%Family`, contains various attributes of the fount as a whole. This includes the full name, the number of characters making up the fount and various details about the size. The keywords in this section are all of type `S.attribute` and the section is terminated by the keyword `%End`, of type `S.End`.

```
%Family <name>
%Style <name>
```

These first two keywords give the full name of fount, for example family Bembo, style BoldItalic. The `%Family` keyword must always be first as it introduces the section but the `%Style` keyword and all the other attributes can occur in any order or not at all.

```
%Characters <number>
%Size-num <number>
%Size-den <number>
```

The `%Characters` keyword introduces the number of characters in the fount and `%Size-num` and `%Size-den` introduce the point size. The size is given as numerator and denominator to allow for fractional point sizes. Commonly `%Size-num` is the size in decipoints and `%Size-den` is 10. The size can take the value NotSet for a master fount if it is being designed for an unspecified size.

```
%Xmin <number>
%Xmax <number>
%Ymin <number>
%Ymax <number>
```

These values give the maximum and minimum coordinate values fount in the fount. These values are given in internal units, which for a master fount is 10000 units to the cap-height and for a raster fount is in units of pixels.

```
%Inter-linespacing <number>
%Cap-height <number>
%X-height <number>
```

`%Inter-linespacing` gives the distance between lines in internal units and so gives the number of internal units that equals the point size. This is particularly important for a master fount as it is the only value that relates internal coordinates to measurements in the outside world. In a raster fount this value need not be given as it can be derived from the resolution and the point size. `%Cap-height` is in fact always 10000 for a master fount by definition but is included for completeness if a fount is transported elsewhere. For a raster fount this value can be of help in matching up different founts for setting together. `%X-height` can take the value NotSet or be omitted altogether. If it is present it can be used to provide a guideline for the design of further characters in this fount. Like the cap-height it can also be used to match up founts for setting together.

Three further values are provided for a raster fount:

```
%Resolution <number>
%Bitplanes <number>
%Representation "Pixels" or "Runlength"
```

The resolution must be given in pixels per inch and the fount will only be the correct point size if it is displayed at this resolution. The bitplanes value gives the number of bits required per pixel and one of the two representations must be selected for the whole fount file.

For a raster fount the maximum space that may be required to store the fount can be calculated from the number of characters and the maximum extent of the pixel array. For a master fount some further information must be provided for such calculations:

```
%Parts <number>
%Strands <number>
%Points <number>
%Strandsize <number>
```

The parts, strands and points values are the maximum total for a single character and strandsize is the maximum number of points found in a single strand. These values may be used, for example, to allocate a single datastructure big enough to accommodate any character in the fount.

```
%TotalParts <number>
%TotalStrands <number>
%TotalPoints <number>
```

The addition of these values allows the amount of space required for the whole fount to be estimated.

Character keywords

The rest of the fount file consists of a section for each character in the fount, introduced by the keyword `%Character`. The section is terminated by a symbol of type `S.End`.

```
%Character <name>
%Xmin <number>
%Xmax <number>
%Ymin <number>
%Ymax <number>
```

The `%Character` keyword introduces the name and the other four attributes give the bounding box of the character. Four further attributes are provided for a character in a master fount:

```
%Parts <number>
%Strands <number>
%Points <number>
%Strandsize <number>
```

These attributes give the total number of parts, strands and points in this character and strandsize is the largest number of points found in a single strand in this character. The data defining the actual outline is then given, introduced by keywords of type `S.data`:

```
%Part <number>
```

This keyword introduces a new part with the given number of strands.

```
%Strand <number>
```

This keyword introduces a new strand with the given number of points. It is followed by an (x, y) coordinate pair for each point in the strand.

For a raster fount the pixel data is given either as sequences of pixels or as run-lengths, depending on the format specified in the fount attributes section. No keywords are used to introduce this data.

Spacetable keywords

The files for spacing both master and raster founts are very similar, except that the units used are those of the particular type of fount. In other words master fount spacetables are given in the internal units of the master fount whereas raster fount tables are given in units of pixels. A spacetable file starts with the name of the spacetable followed by a section defining the fount to which this spacetable belongs:

```
%Spacetable <name>
%Family <name>
%Style <name>
%Characters <number>
%Size-num <number>
%Size-den <number>
%Resolution <number> [for a raster fount only]
%Bitplanes  <number> [for a raster fount only]
```

This information can be used to check that a given spacetable does indeed belong to the fount which is being set. In addition, for a raster fount the following may be included:

```
%Factor <number>
```

This factor is used for scaling the spacing values down when they are actually used. When a raster fount is derived from a master it will be scaled down, generally quite considerably. The spacing information must also be scaled down but the rounding errors introduced can have a serious cumulative effect. By providing this scaling factor the spacing information can be stored to a higher resolution and the space between characters can be calculated at this same high resolution if desired. Scaling down after calculating the spacing between characters reduces the effect of the rounding errors. This section is terminated by the keyword `%End`. The file continues with a section for each character:

```
%Character <name>
%Centre-x <number>
%Centre-y <number>
%Before <number>
%After <number>
%Between <number>
```

The before and after values are the distance to be left before and after the centre of the character. With optical spacing the before and after values are the same, being half the width but with other spacing methods these values may be different. The distance to be left between the centres of two letters is the sum of the after value for the first and the before value of the second. `%Between` gives the number of special adjustment values provided for this character. It is followed by that number of items with the format:

```
<number> : <name>
```

The number is the amount of extra space to be added (the space can be negative) when the current character is followed by the character with the given name. This extra space is added to the sum of after and before values. These values are referred to as "betweens" rather than kerns because they may be measured at an angle between centres, unlike conventional kerns which are measured along the baseline. As in the other fount database files the information about a character is terminated by a keyword of type `S.End`.

Codetable keywords
A codetable files starts with some general information:

```
%Code <name>
%MinCode <number>
%MaxCode <number>
```

This gives the name of the codetable and the range of code values that are used. For example the codetable might be ASCII and the range 0 to 127. The fount database could contain several different ASCII codetables (which must have distinctive names) if several different naming schemes for characters are used, or if some foreign fount is to be mapped to ASCII codes. After this introductory information the following information is provided for each code:

```
<number> <name>
```

This gives the name that is to be mapped to the given code. After the last pair of values the keyword `%End` is used.

File names
All the fount files have names that indicate what the contents of a file are. The files of the database are stored in various sub-directories of a directory with the name founts:. Founts:masterchars contains files of master outlines, founts:masterspace contains the spacing information for the master founts, founts:rasterchars contains the raster characters and founts:rasterspace contains the spacing information for the raster founts. Within each sub-directory the file names are composed of information that specifies exactly the fount to be found in that file. For a master fount this is:

```
<family>-<style>-<size>
```

`size` is a number giving the size in decipoints. If it is 0 then this fount was designed to be displayed at any size. `family` is a name used for grouping a set of related founts together and `style` distinguishes between these related founts. Examples of families are "Times" or "Century" and commonly used styles are "roman", "italic", "light" or "bold". There is no restriction on the names that can be used, although founts of the same family may be expected to look correct when used together. For example, one might use "times-roman-100" and "times-italic-100" together. For raster founts the number of bits per pixel and the resolution of the device in pixels per inch must also be specified:

```
<family>-<style>-<size>-<bits>-<resolution>
```

For spacing files the name of the spacetable must also be included:

```
<family>-<style>-<size>-<spacename>
<family>-<style>-<size>-<spacename>-<bits>-<resolution>
```

These file names can become very long but the structure of the names makes clear the relationships between the different founts in the database. It allows for the possibility of automatic fount selection, for example finding an italic to go with a particular roman or finding any bold fount of a particular size. Within Imp the file names are constructed automatically so that the user does not have to worry about the length of the names.

6.3 Organisation in the workstation

When fount data is stored in the memory of the workstation it is organised to be compact but rapidly accessible. The type of information stored, and the way that it is stored, is related very closely to the way that Imp makes use of the information. Each fount in memory has a fount descriptor giving general information about the fount as well as information about each character. Spacing tables are not stored separately but as part of the fount description. Hence the same character shapes with several different spacing methods are stored as several completely separate founts. Two lists are maintained, one of master fount descriptors and the other of raster founts. This enables Imp to determine rapidly which founts are in memory. Also a symbol table is provided to map character names to ASCII codes for use in displaying text.

Master founts

A master fount is represented by a vector of words in memory consisting of thirteen words of general fount information followed by information for each character. The general information starts with pointers to strings giving the family, style, size in decipoints and the name of the space table. These strings are the same strings that appear in the form in the master fount window (discussed in the next section) and so they can be changed directly by the user of Imp. Next the values of inter-line spacing, x-height and the minimum and maximum coordinates are given. These values can also be set up and changed interactively in Imp. If the fount has been read in from disc these values are initially those found in the fount file. When a fount is created from scratch within Imp these values are uninitialised until they are explicitly set up through Imp commands. For a fount in memory the maximum and minimum coordinates do not necessarily define the exact maximum and minimum values for the current set of outlines. Rather, the values are used to define the size of workspace for outlines to be created in, using the master outline editor. The size of the workspace can be changed in the editor and the new value saved in the fount vector for use later in the same session. This allows characters larger than the current bounding box to be created without problems. When the fount is written out to disc the maximum and minimum coordinates saved in the file are calculated from the outline data itself so that they define the minimum bounding box for any character in this fount.

The final two words of fount information are pointers to lists of vertical and horizontal grid lines for use in the master editor window. These lines have no particular meaning but serve as guidelines for the user when creating new outlines. The user sets up these lines using the "Grids" mode of the editor and can then send them to a fount window to be saved in the fount vector. These lines do not form part of the master fount data but they are written out to disc by Imp when a master fount is saved. When the fount is read back in again the grid lines are automatically retrieved at the same time.

Information about each character follows the general fount information. The fount vector is created with space for 128 characters corresponding to the 128 slots available in the fount window. Each character is described by seven words of information, the first being a pointer to a string giving the character's name, followed by four words giving the coordinate range for this character. The final two words are a pointer to the data structure representing the character's outline and a pointer to the spacing information. Any of the pointers may have the value Nil, indicating that there is no data for that item. A character with an outline but no name is given the name "Unknown" and if the character has no spacing information then simple default values are calculated from the character's width when this information is needed. The spacing information is given as a four word vector containing the coordinates of the centre and the space to be left before and after this centre.

The outline of the character is represented by a hierarchical data structure. Each level is represented by a vector of values, with the first word of the vector giving the number of values at that level. The first level is the part list, where each value is a pointer to a list of strands making up a part of the character. Each strand list is a vector of pointers to strands and the strands are vectors of points. The x and y coordinate of each point is packed into a single word as neither value should exceed 16 bits in size. This data structure is illustrated in figure 6.5.

Raster Founts

The vector representing a raster fount is very similar to that of a master fount, with twelve words of general information, followed by information about each character to a maximum of 128. The general information is somewhat different from that of a master fount, reflecting the different type of data being stored. As well as family, style, size and space table strings there are pointers to strings giving the resolution this fount is intended to be displayed at and the number of bits needed for a pixel value. All of these strings appear in the form in the raster fount window, where they can be updated by the user. One word defines the representation used for the pixel data, either "run-length" or "pixels". Currently the various programs making up Imp use only pixel representation. Another word gives the factor by which the

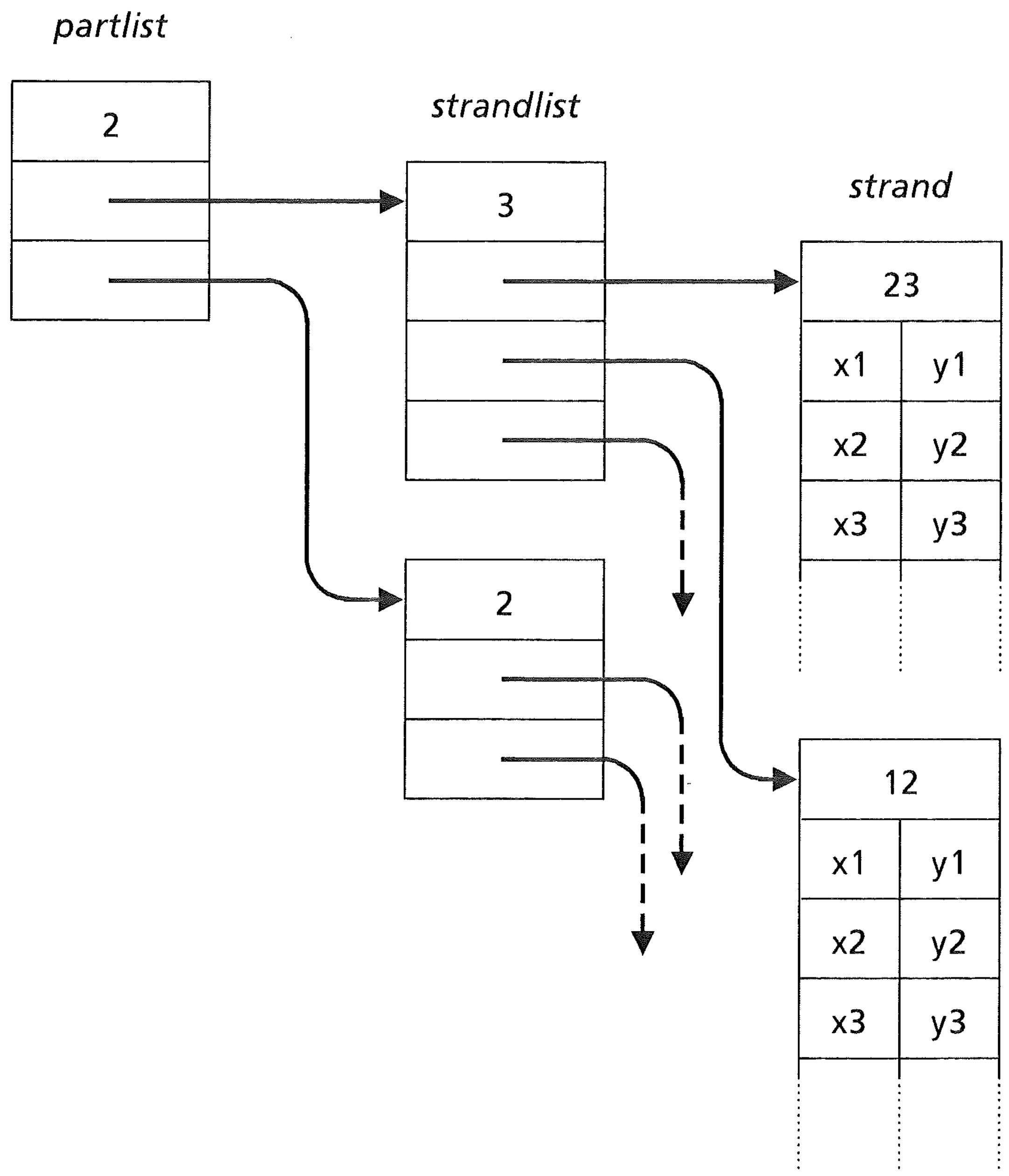


Figure 6.5
Representation of an outline in the workstation memory

spacing values must be divided to get pixel units. The final four values give the maximum and minimum coordinates. As in a master fount, these values are used in the allocation of working areas and they can be updated by the raster editor program. They are also recalculated when the fount is written out to a file. The current implementation does not include the x-height or cap-height values because Imp does not make use of these values. This omission should be corrected in the future because these values could well be of use outside Imp.

The information about each character is identical to that in a master fount except that the pointer to the character's outline is replaced by a pointer to a pixel array. This array has one byte for each pixel within the character's bounding box. The array is organised with the first byte being the pixel value at `(xmin, ymin)`, the next being `(xmin+1, ymin)` and so on to `(xmax, ymin)`. This pattern then repeats from `xmin` to `xmax` for `ymin+1` up to `ymax`. This makes quite extravagant use of space, both in storing all the pixels within the bounding box and in using a byte for each pixel. The pixel map could be stored in an encoded form so that large numbers of insignificant pixels were not stored. Also the absolute minimum amount of space could be used per pixel. Most founts use just one bit per pixel and few of the others use even as many as four, so this could result in a very considerable saving. Using either, or both methods, access to individual pixels would be slow because extra calculations would be required to translate the encoding and to extract values of less than a byte. In fact this would only be a problem in an editor window where rapid access is required. Here we are discussing formats for storage where speed of access is not a very significant factor. The reason why no attempt has yet been made to compress the data is that there has been no shortage of memory space and so there has been no justification for expending effort in this area.

Code tables

The characters making up a fount are identified by textual names and there are times when these names must be related to some numeric coding scheme. In Imp a fount window can contain up to 128 characters and these characters can be mapped onto the ASCII coding. Text is presented to Imp in ASCII coding and so a table must be provided with each fount to locate the character definition corresponding to the code in the text. This table takes the form of a vector indexed by ASCII code with each entry being a pointer to the character definition corresponding to that code. The position of a character within a fount window could be made to correspond to its code but this would require the designer to be familiar with the coding. The extra level of indirection allows the designer to organise the characters in any order. The table of pointers to the character definitions in a fount can then be set up from the textual names given to the characters.

A symbol table constructed from the information in a code table file is used to translate names to codes. A hash value is calculated from each name in the code table file and an entry containing the name and code number is made at that point in the table. Entries with the same hash value are linked together and are distinguished by the name stored in the entry. Several names can map to the same code value but within one fount only one of these names should be used. This allows the same code table to be used for all of the founts in Imp even if they contain completely different characters. When a character is added to a fount the hash value for the name is calculated to locate the appropriate symbol table entry. This entry gives the code and a pointer to the character definition can then be put in the code table vector at the appropriate point.

6.4 Master and Raster Fount Windows

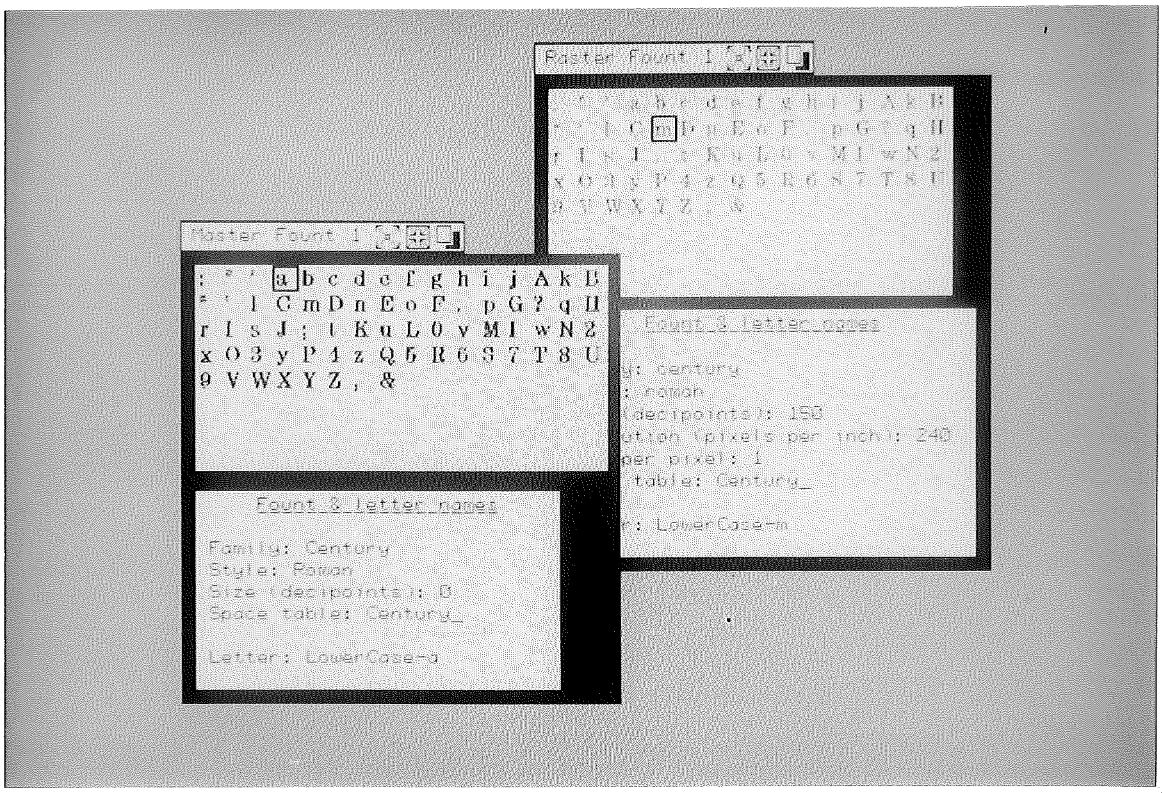
These windows provide a visual interface to the fount database. Both provide 128 "slots" to contain character data and a form which displays details of the name and size of the fount currently stored in the window. The fount data is stored as vector, as described above. The character stored in a slot is displayed there at a small size so that the character shapes making up a fount can be seen at a glance. It also allows the designer to select characters from the fount by shape rather than by name. One of the slots is always current and the name of the character in this slot is also displayed in the form. Figure 6.6 shows some fount windows.

The array of slots is implemented as a menu with eight rows and sixteen columns of items. Each item is displayed as a sixteen by sixteen pixel array. The routine for drawing an item is given a pointer to the character definition at the corresponding offset in the fount vector. The character shape is then drawn in the area allocated for the item. This pointer to the character definition is also the result that is returned when the item is selected. The character definition is augmented by an extra item of information, the index number of the character in the menu. This means that when the character definition is changed the item can be redrawn simply by calling `Menu.redraw` with the appropriate index number.

Characters from a master fount can be scaled down easily for drawing in the menu. Because all master founts are defined with a cap-height of 10000 units all the coordinates can be shifted right by ten places (ie. divided by 1024) to fall within a range of about sixteen pixels. The outlines are then drawn using straight forward line drawing routines. A lot of detail is obviously lost and better results might be obtained by using anti-aliased lines. The lines would then appear finer and the outline could be drawn to an apparently higher resolution. This has not in fact been done because the results of the straight forward method are good enough to allow different characters in the fount to be distinguished which is all that is really required. The details of the shapes can be examined in other windows and so the loss of detail here is not serious.

It is much less straight forward to scale characters from raster founts. This is done by super-sampling the high resolution pixel array to give a sixteen by sixteen pixel array of average pixel values which can then be displayed. The resulting pixel value is compressed into two bits, giving four levels of grey scale. The algorithm used is the same as that used to display text on screen and is described in detail in chapter eight. As with the master founts a good deal of detail is lost but the results are good enough to distinguish the characters. The characters in different raster founts are very different sizes, depending on the resolution of the intended output device and the point size of the fount. Hence, a single factor cannot be used to scale all the founts and so the factor is calculated individually for each fount.

When switch number one is depressed `Menu.select` is called to highlight items in the character menu as the cursor moves over them. A slot in the menu is selected when switch number one is released. The name of the character associated with this slot is displayed in the form, where it can then be updated, as discussed below. If there is no character in this slot or it does not yet have a name then the string



Family: Century
Style: Roman
Size (decipoints): 0
Space table: Century_

Letter: LowerCase-a

y: century
: roman
(decipoints): 150
ution (pixels per inch): 240
per pixel: 1
table: Century_

r: LowerCase-m

Figure 6.6
Master fount and raster fount windows .

"Unknown" is displayed in the form.

Any switch packets from the keyboard are passed to the form, allowing the fields to be updated at any time. The vectors used as the result vectors for the form are the same vectors as are used to store the strings associated with the fount. Hence, as the user types a new name it is automatically included in the fount data. It can be argued that the names of raster founts and their characters should not be changed because this destroys the relationship with the corresponding master fount. This is not in fact enforced as sometimes a raster fount may be extensively edited and so not correspond to any master fount anyway. It is left to the user to use this freedom if he wishes but generally it is hoped that very little work will be done on raster founts once they have been derived from the master designs, so making renaming unnecessary.

Both windows have a pop-up menu containing the following commands:

```
Read in from file
Copy out to file
Send data
Get data
Clear
Finish
```

In addition the raster fount window has the command:

```
Scan convert a master
```

Any of these commands can be issued at any time.

The command to read in from a file constructs file names for the fount and space table from the strings in the form. If the files exist their contents are read with the characters being placed in successive slots in the window. The previous contents of the window are saved so that if the user hits the cancel key at any time while a file is being read the reading is stopped and the previous contents restored. The saved data is discarded once the reading is completed. This gives sufficient time for the user to cancel the reading as reading a file takes a significant amount of time. Nothing happens if the files do not exist and the user is free to try another name or to create a fount of this name from scratch. As the characters are read their names are looked up in the symbol table and a codetable constructed to map from codes to the character definitions.

The command to copy out to a file also constructs the file names from the strings in the form. The data is not written straight out to these files but is written to temporary files. Once the writing is complete these files are renamed to take the constructed names. It is only at this point that the previous versions of these files, if any, are lost. The writing of a file can be cancelled at any time before this. As a further precaution, if previous versions are found to exist the user is asked if he really wants to overwrite them. This is done by displaying a form in the fount window into which he can type his reply. In particular, this picks up the frequent mistake of creating a new fount window, typing in the fount name and then selecting "Copy out to file" instead of "Read in from file". Copying the empty fount window out to a file happens so rapidly that the user has no chance to cancel it before the fount on disc is overwritten!

Imp provides two channels for communication between windows, created using `W.createChannel`. One is for passing raster data around and the other for master data. The command "Send data" causes a request of type Request.sendData, and "Get data" a request of type Request.getData, to be made on the appropriate channel. The data in the request packet is a two word vector with a pointer to the fount vector and a pointer to the currently selected slot. All activity in this window is suspended until either the user cancels the request or a request of the complementary type is made from elsewhere. The fount window program takes no action after a request is satisfied, all actions being left to the window that made the corresponding request. For example, the user may have designed a new master outline which needs to be stored. The command "Send character" is selected in the master editor window, which causes a request of type Request.SendData to be made. This request is satisfied by issuing the command "Get data" in the fount window. The master editor window then copies the data for this new character into the current slot of the master fount window. Menu.redraw is called to display this new character in the menu.

The command to clear the window removes all the fount data leaving the space clear to create a new fount. The data is not finally destroyed until another command is received, thus allowing the user to cancel this command if it was a mistake. The whole window can be closed down when it is no longer needed, and again this can be cancelled if it was issued by mistake.

A master fount can be created from scratch by designing characters in the master editor window and then storing them in an empty fount window. The only way to create a new raster fount is by scan converting a master fount. When the command is issued the list of master founts in memory is searched for one that matches the family, style, size and spacetable of the required raster fount. If one of the same size is not found then a search is made for one with size zero, indicating that the design is for any size. If none is found then the user is informed so that he can then amend the names in the form or read in a suitable master from disc. The form in the raster fount window gives the resolution and the number of bits to use per pixel and the scan conversion is carried out using the algorithm described in chapter eight. The scale factor to be applied is calculated from the point size of the fount (giving the inter-line spacing in fractions of an inch) and the inter-line spacing value given in internal units. The number of pixels spanning the inter-line space is derived from the resolution of the device.

More than one bit per pixel cannot be produced automatically from within Imp at the moment. The user must produce a fount that is one bit deep within Imp and then apply a super-sampling program on the resulting fount file. This program halves the linear dimensions of the fount and uses two bits to represent the average value in each pixel. Some fairly simple modifications would enable the program to be incorporated into Imp and produce any number of bits per pixel.

7. Master Founts

The production of master founts is the most important part of Imp and so extensive facilities are provided for creating and editing them. The first section of this chapter discusses some approaches that can be taken to the manipulation of the outlines defining the master characters. This provides some background to the description of the master editor window in Imp. It is important to be able to see characters in relation to one another and the final section describes the window that provides the facilities for doing this.

7.1 Approaches to outline manipulation

The master fount characters are stored in the form of line chains defining their outlines. We need to provide interactive tools for creating these line chains and for editing them once they are created. An initial approach was to try and mimic traditional ways of working very directly, keeping the nature of the line chain representation hidden from the designer. The line chains would be derived automatically in some way from the designer's sketches and they would be edited as the designer changed his sketches.

Keeping directly in line with traditional methods the designer could be provided with an electronic drawing board where he could sketch and paint in a very familiar manner. While the designer worked the picture would be stored simply as a bit map without any structure. Once the design was completed the outline of the shape could be extracted automatically from the bit map using image processing techniques. A disadvantage of this approach is that the design, when it is being worked on, is in a form that cannot easily be manipulated. It is hard to exploit much of the power of the computer in preparing the design. Encouraging the designer to work in terms of outlines from the earliest stages may be preferable.

When someone is sketching a rough picture they will often draw faintly at first and then gradually firm up the lines by repeated drawing. The final shape is seen where the darkest lines are drawn. This could be applied to the production of outlines, with the outline being defined where the darkest pixels are found. Each pass of the pen over a pixel would increment its value in imitation of sketching with a pencil. Again some sort of image processing would have to be used to extract the outline. The outline could be extracted at an early stage and then further sketching could be used to change the shape of the outline.

A more straight forward approach, avoiding the need for image processing, would be to create a line chain to exactly follow the path of the pen. The shape could then be changed by redrawing and moving the line chain onto the path of the pen. The underlying structure would still be implicit and hidden from the designer.

This last method was chosen but the simplicity of the idea was soon smothered by the difficulties of implementation. For example, is a particular pen stroke to define a new line or to change the shape of an existing line? A proximity test was used, so that if the pen was near an existing line then this line was moved, otherwise a new line was created. Similarly, the ends of line chains were joined if they were sufficiently close. This caused problems when lines were intended to be close together —the system made incorrect assumptions about what was wanted. The way round this is to provide commands to signal the interpretation required, for example "Create a line chain" or "Move a line chain". Once this is done, the initial simplicity of being able to sketch without thinking about structures has been lost. Rather than implement further algorithms, such as automatically introducing more points along curves, it was decided to make the structure completely explicit. The designer would be provided with tools for manipulating this structure directly.

Creating line chains in Imp

It is expected that the designer will work with line chains from a fairly early stage of the design but it is hard to create such outlines from nothing. Hence, a simple painting and sketching facility is provided so that rough shapes can be drawn without any need to worry about structure. Once the basic shape has been roughed out the designer picks out points around the outline himself rather than it being done automatically. Two different methods have been tried for this and which would be preferable has not yet become apparent. When the "Create line chain" command is selected a new line chain is started. As points are created they are linked onto the end of this chain until the command is terminated. One method is to create a new point at the cursor position each time switch number 1 is pressed. The position of this point is not fixed until the switch is released so that it can be dragged around into the correct position. This can prove rather laborious around curves where a lot of points are needed and so another approach is to create new points as the cursor moves with the switch held down. A point is created each time the cursor moves more than some specified amount. This is essentially the method described above where a line chain is created to follow the path of the pen.

The designer is obviously aware that the outline consists of points joined by lines and this is further emphasised by picking out the points in red as the outline is drawn. Links between points can be broken or created using another command, again giving a direct handle onto the data structure. The continuous visual feedback means that a feel for the structures involved is very rapidly developed.

Editing line chains

Once a line chain has been created it will undergo extensive modification as it is assumed that the initial version will be very rough. Two different methods for moving existing points have been tried out and again no decision as to which is better has been made. The first method is to select a point when the switch is pressed and then to drag it around as the cursor moves with the switch held down. As with creating line chains this can be very laborious and so an alternative was investigated. In this case a path is drawn as the cursor moves. When the switch is released the segment of line chain nearest the path is moved onto it. This is done by finding two points on the same line chain that are closest to the two ends of the path. Another point is then found that is nearest to the mid-point of the path to define which section of the line chain is to be moved. The points in this section are then moved to lie evenly spaced along the path. If the switch is pressed and released without moving the cursor then the effect is to move the nearest point to the cursor position. This means that the ability to reposition single points easily has not been lost but continuous feedback as a point moves is no longer possible.

The initial outlines will probably not contain a large number of points and so as they are refined more points will need to be introduced, especially for smoothing

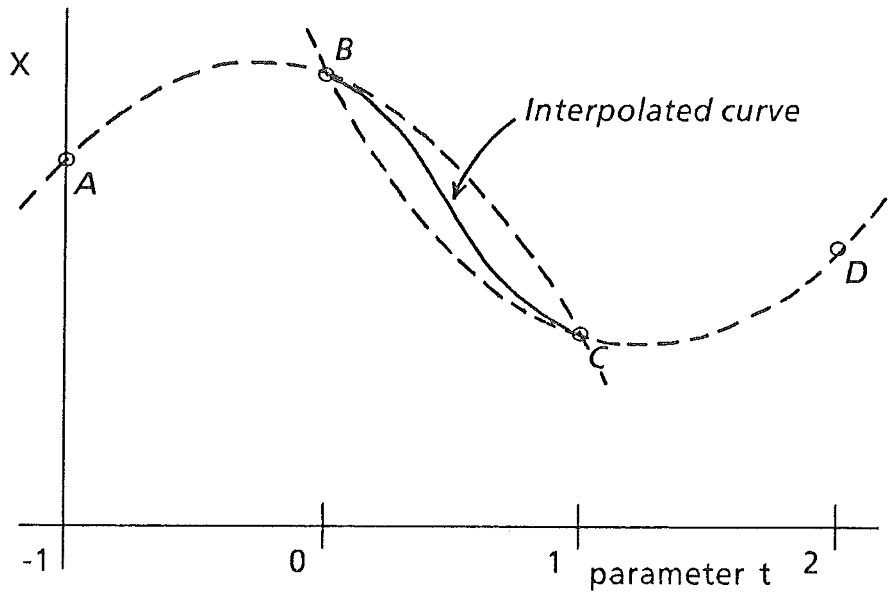
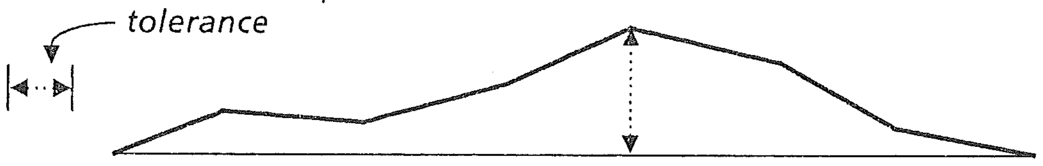
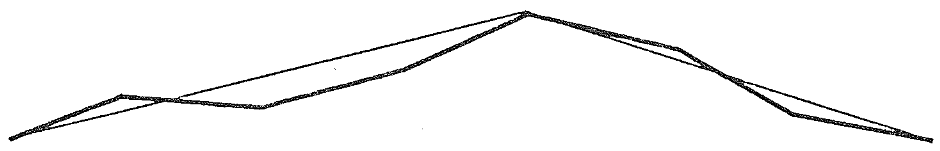


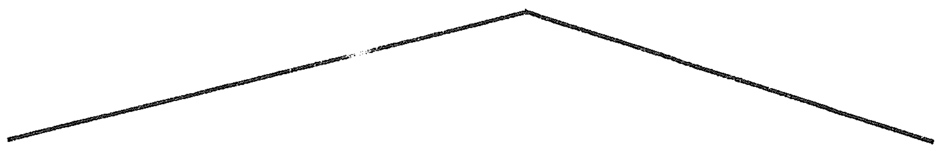
Figure 7.1
Generating the Overhauser curve between points *B* and *C*.



The furthest point in this section is further from the baseline than the tolerance and so the section must be subdivided.



The furthest points in both sections are now less than the tolerance and so these sections can be replaced by straight lines.



The filtered line chain.

Figure 7.2
The filtering algorithm

curved portions. There are a variety of methods for generating points along curves defined by a few control points and one method has been implemented in Imp. This makes use of Overhauser curves [Brewer and Anderson, 1977] which are blended polynomials that are easy to calculate and produce pleasing, interpolating curves. Given four points, A, B, C and D, an Overhauser curve can be fitted between points B and C. Quadratic curves are fitted through points A, B, C and points B, C, D and the curve between B and C is a weighted average of the two curves. At B it is 100% the first curve and at C it is 100% the second. In between it takes a proportion of each curve depending on the relative distance from B and C. This is done separately for the x and y coordinates, with the curves being expressed in terms of a third parameter, the distance along the curve. Figure 7.1 illustrates this calculation. Any number of points can be inserted between B and C but currently in Imp only one point is added for each application of the smoothing command.

It is also possible that there are too many points in the outline, for example if smoothing was applied a large number of times points would continue to be added whether or not they were necessary. A filtering algorithm [Douglas and Peucker, 1973] can be applied that removes all points that are unnecessary to represent the shape to a given tolerance. This is done by taking a section of line chain and working out the perpendicular distance of each point from a line joining the two end points. If the maximum such distance is less than the tolerance given then this section can be represented by the single line joining the two ends. All the other points can be discarded. If the maximum distance is greater than this tolerance then the line chain is subdivided at this maximum point and the process repeated on the two halves. This continues until we come down to adjacent pairs of points. Figure 7.2 illustrates this process.

Data structures

The data structure used for representing an outline in the fount windows is compact but is not easily edited. With the strands stored as vectors of points, the individual points can be moved easily but addition of a new point in the middle would involve moving all the data for the later points along the vector. A less compact, but more suitable, structure is a linked list of some sort. Points can be easily added or removed by updating pointers rather than physically moving data around. Each part, strand and point is represented by a descriptor which can be linked into a list.

The lowest level object in the data structure is the point. The descriptor for a point has the following fields:

```
1. A pointer to the next point in the strand
2. A pointer to the previous point
3. Fixed or free?
4. x coordinate
5. y coordinate
```

These fields are largely self-explanatory except for the third. This is a flag indicating whether the user has selected this point to be marked as fixed. This information is used by the editor program to determine which points can be moved by the editing commands. The user can fix certain points that have been carefully positioned so that they are not later moved accidentally. These fixed points are also used to delimit sections of line chain for certain commands. The information about which points are fixed and which free is specific to just the editor and it is lost once

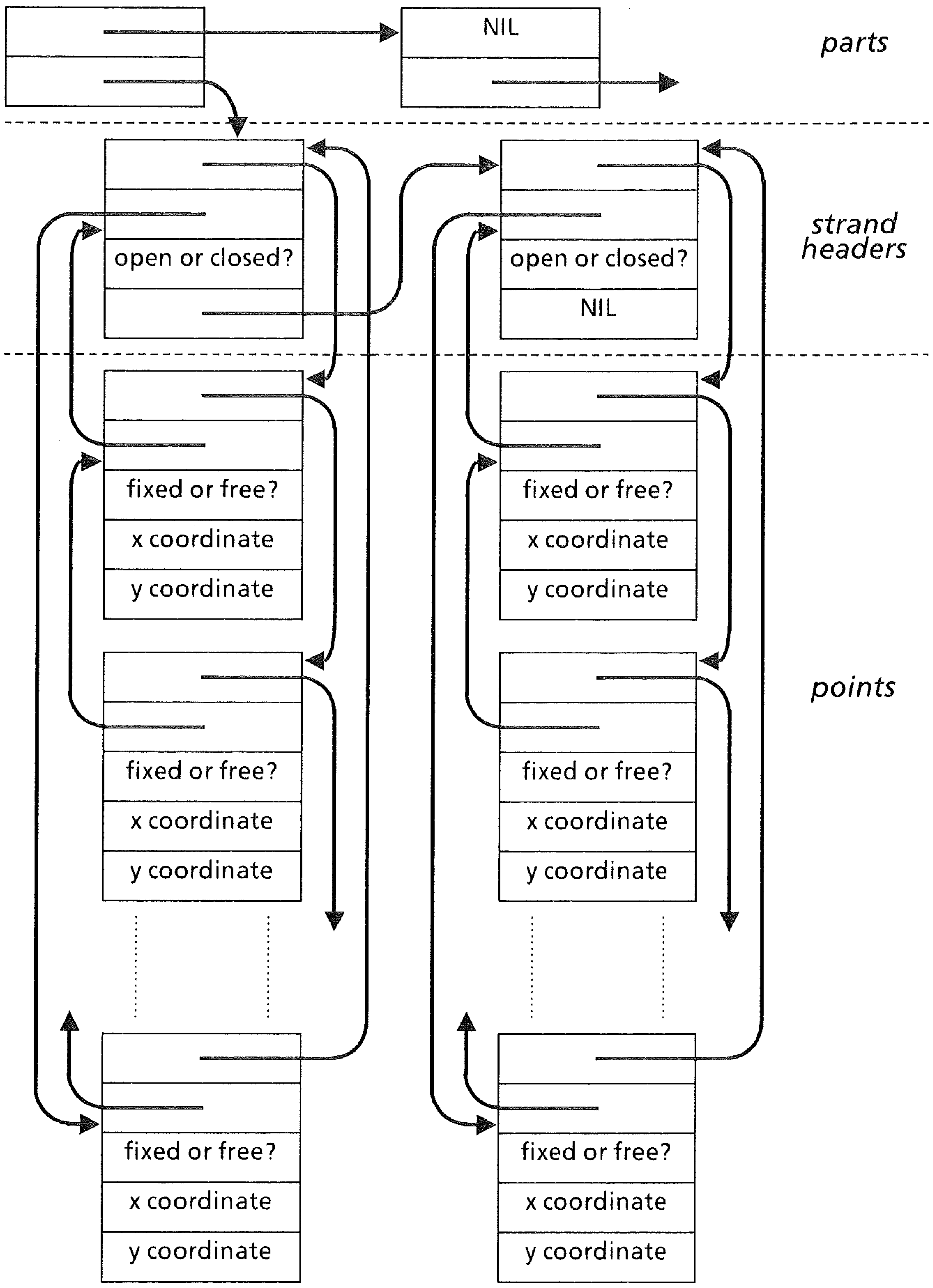


Figure 7.3
Representation of an outline as used in the outline editor.

outlines have been stored in a fount window.

It can be seen that the list of points making up a strand is a doubly-linked list. This was done to speed up the frequently required operation of finding both neighbours of a point. With a singly-linked list a search would have to be made of the list, keeping a pointer to the previous and current points until the required point was found. By having pointers in both directions both neighbours can be found immediately. This operation is not just needed when points are inserted or deleted but also when a point is repositioned. In this case, the lines linking the point to both its neighbours must be redrawn.

A strand is described by a strand header block, which actually forms part of the list of points. This was done so that, given a point, it is easy to find which strand it is in by following round the list of points to the strand header. A more direct, and possibly tidier, method would be to have a pointer to the strand header in every point descriptor. This was rejected as being excessively wasteful of space, especially as it is not a very frequently used operation. The fields of a strand header are:

```
1. A pointer to the first point
2. A pointer to the last point
3. Open or closed?
4. A pointer to the next strand in this part
```

The first two fields correspond to the next and previous pointers in the point descriptor and they serve to close the circle in both directions. The third field can be used to distinguish a strand header from a point descriptor as the range of values it can take are different from the third field of the point descriptor. The value of this field indicates whether the last point of the strand is linked to the first. In the fount database this is always true by definition but during editing line chains may be incomplete and hence it would be inappropriate to link the two ends. If two points in a closed line chain are unlinked then the line chain is reorganised such that these two points become the first and last points. If a line chain that is already open is split again by unlinking another pair of points then a new strand is automatically created. When two separate line chains are linked together care must be taken to ensure that the pointers continue round in the same direction in the two chains. If this is not the case then one of the line chains is automatically reversed before they are linked.

The strands of an outline are collected into parts, each part being a singly-linked list of strands. The fourth field of the strand header is a pointer to the next strand in this part. The list is not doubly-linked as it is not modified as frequently or as rapidly as the lists of points making up strands. No noticeable delays result from this decision and so no need has been felt to use a more complex structure. A part is described by:

```
1. A pointer to the next part
2. A pointer to the first strand of this part
```

The list of parts making up an outline is also a singly-linked list as updating of this list is infrequent. Figure 7.3 shows the structure of an outline in this format.

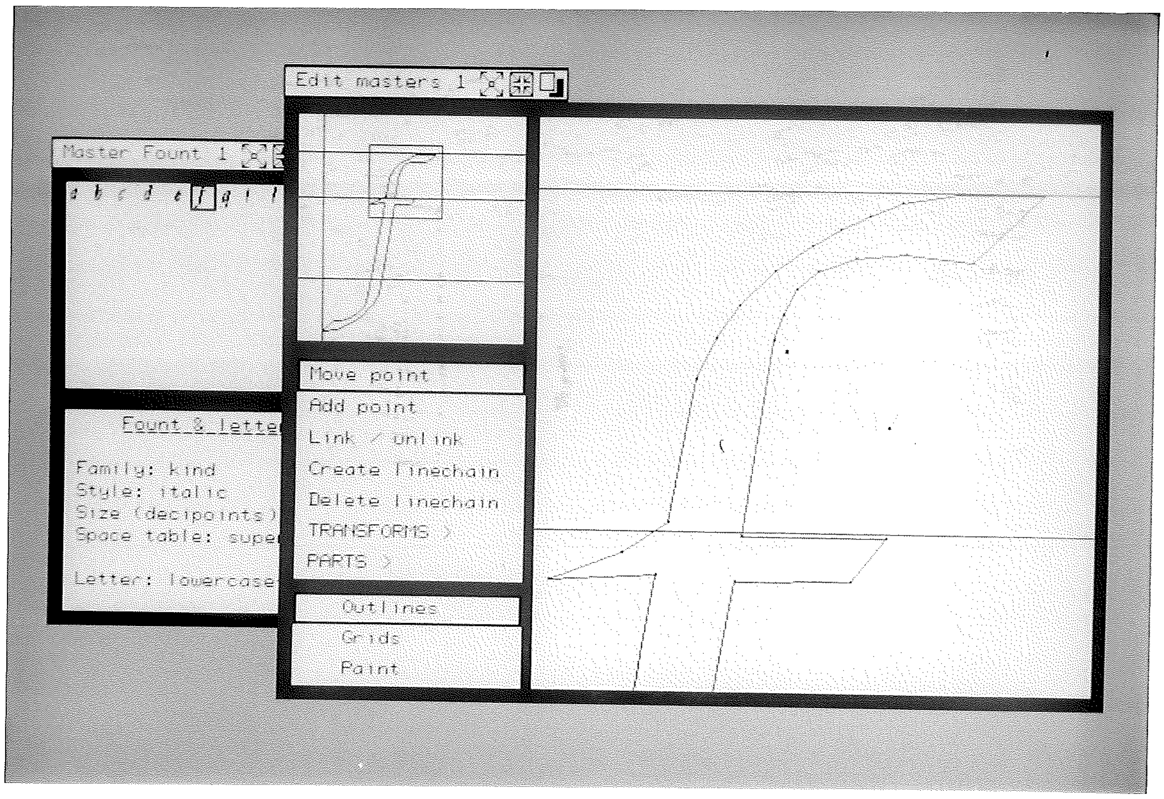
7.2 Master Editor Window

This is the window in which the master outlines of letters are created and edited. The outline to be worked on can be retrieved from a fount database window or created from scratch. At any time the outline can be stored back into the fount database, or further outlines retrieved and added to the outline being worked on. Pieces of outline can be cut out and spliced in elsewhere so, for example, a standard serif could be designed, stored in the database and then spliced into each letter as it is created. Each fount has an associated set of grid lines which can be displayed in this window to ensure that all the letters of the fount are in the correct proportions. The grid includes x-height and cap-height lines plus any others that the designer wishes to set up. The distance between the cap-height and baseline is defined to be 10000 units and all coordinates are stored in this unit system. This window also provides facilities for free-hand sketching. No data is extracted from the sketching by the system but the sketches can serve as a guide to the designer for creating outlines or grids. The design of this window has raised many interesting problems in the area of the user interface and experimentation is still going on. This is reflected in the discussion below which includes details of ideas already tried out and rejected as well as ideas for the future.

Overall organisation

This window is divided into five different areas, as is illustrated in figure 7.4. The interpretation of switch actions depends on which of these areas within the window the cursor is in. A list is maintained of the size and position of each area and an associated function to call when events occur in each area. The layout of the areas can be changed to be more suitable for a right-handed person by adjusting a single value at the head of the program. The central data structure in this window is a character outline in linked-list form along with a set of horizontal and vertical grid lines. The outline is the final product of this window and everything done in this window is ultimately geared towards creating and manipulating this outline.

The reference area displays the outline and grid lines in the whole of the available coordinate range whereas the working area displays some portion of this area. All editing of the outline is carried out in the working area but the changes are seen immediately in the reference area, giving an overall context for what is being done. The particular portion displayed in the working area is selected in the reference area by manipulating a box that is displayed there to outline the portion. In an attempt to be consistent a method similar to that used by the window manager is employed to adjust the box. When switch number one is depressed the whole box is moved if the cursor is in the middle and corners are dragged in and out if the cursor is in a corner. Because the working area is square the box in the reference area is constrained to remain square. Unfortunately this causes the analogy with the window manager to break down and so makes the use of this technique less useful, and maybe even confusing, for the user. It might be preferable to change to a very different technique and not encourage the user to draw parallels with the window manager. For example, the user could indicate the position of one corner by pressing switch number one, move the cursor and give the position of the opposite corner by releasing the switch. The resulting box could be continually updated as the cursor is moved to provide feedback. On the other hand, being able to move a fixed size box is useful as it makes it easy to work on a number of areas at the same magnification.



Fount & lette

Family: kind
Style: italic
Size (decipoints)
Space table: supe

Letter: lowercase

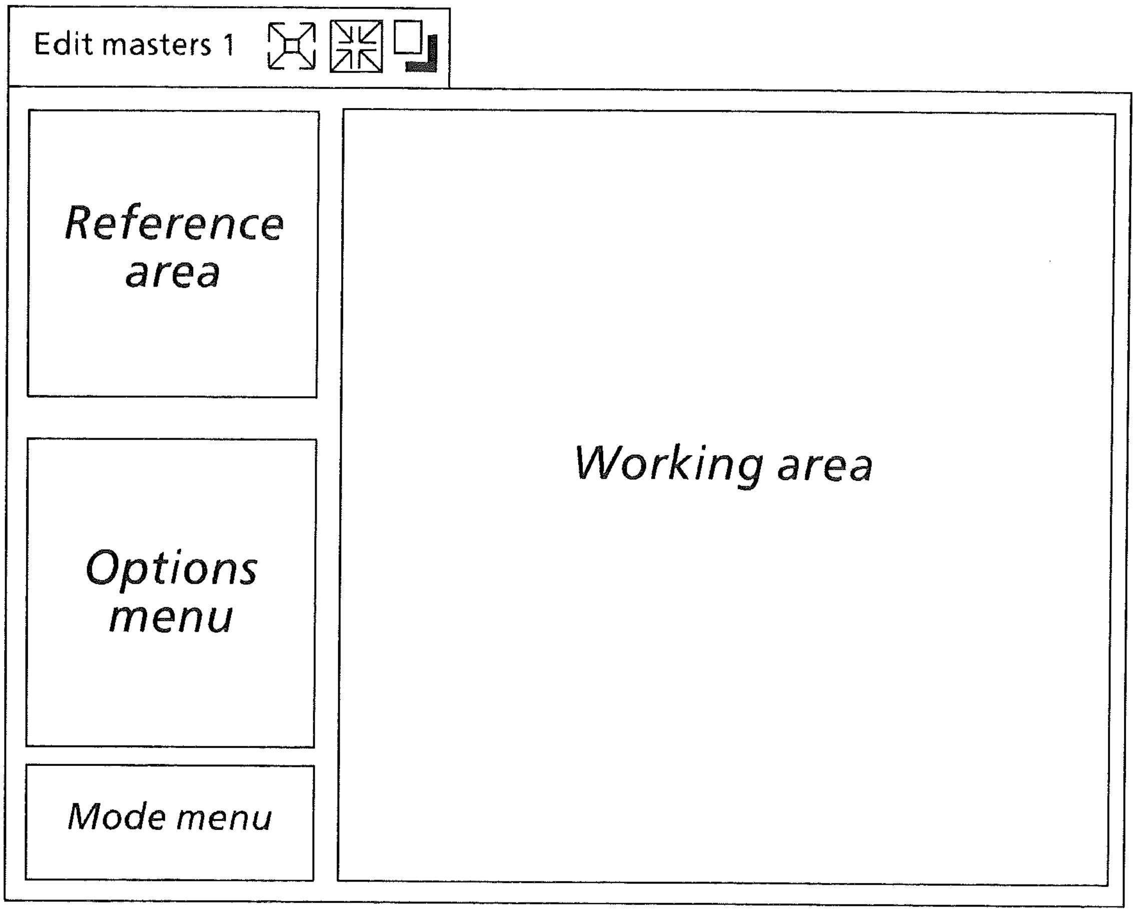


Figure 7.4
The master editor window

It is also the same action as is used in the raster editor for selecting a new area. This is a good illustration of the problems encountered in trying to achieve total consistency across a number of different programs.

The main work of this window is carried out in the working area. This area is made up of three layers, the bottom layer for painting, the next for setting up grids and the top layer for creating and editing the outlines. The top two layers behave something like tracing paper so that everything underneath can still be seen. Hence the outlines are seen superimposed on the grids and sketches. These layers correspond to the three modes displayed in the mode menu. This menu is permanently displayed and a mode is selected by pointing in the menu. Each mode has an associated set of commands which are displayed in the mode-specific menu. This menu is changed each time a new mode is selected. In general, these mode-specific commands are selected in the menu and then applied to specific objects by pointing at them. The command remains selected until another replaces it and so it can be repeatedly applied by pointing at more objects. Details of the actual commands are given in the sections below on each of the modes. There are a number of commands concerned with storage and retrieval of outlines and grids and for changing certain parameters associated with the window. These commands are provided by way of a pop-up menu, which is the same in every mode.

Every time a menu selection is made the state of the master editor is saved. If the user makes a mistake, or the command has an unexpected effect, he can hit the cancel key and retrieve this previous state. This means that it is safe to experiment with unfamiliar commands. If the cancel key is hit a second time, the effect of the previous cancel is reversed and the new state restored. The user can swap repeatedly between the new and previous states, providing the opportunity to look closely at the effect of the command just issued.

Painting

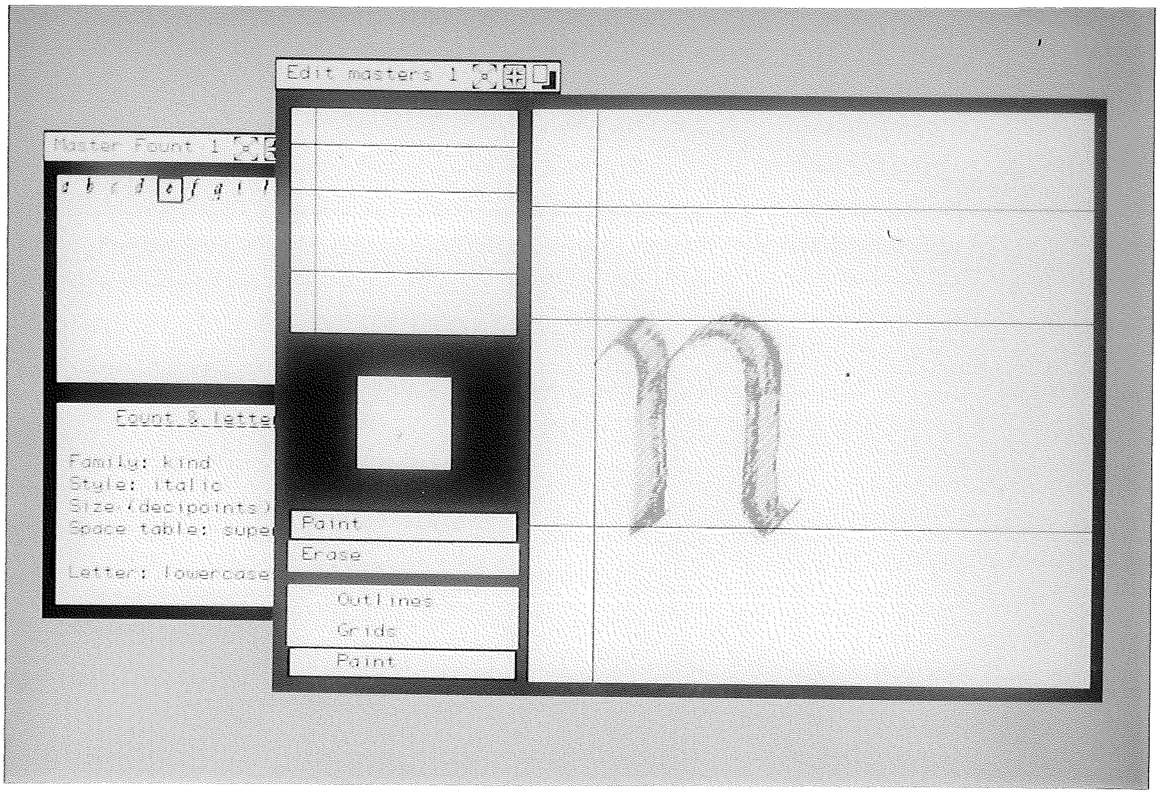
When this mode is selected the user can sketch freely in the working area. As long as switch number one is depressed a trail is drawn as the cursor moves in this area. Because switch number one corresponds to the switch in the tip of the tablet stylus as well as to the left mouse button a natural drawing action is possible. The stylus is pressed down on the tablet to start drawing in a manner similar to pressing a pencil on paper. It is at times like this that the freedom to move between tools is so valuable. The mouse is generally more useful for setting up and manipulating windows and for selecting commands but the much finer and more fluid movements possible with a stylus make it invaluable for sketching subtle shapes.

The menu specific for this mode contains the following commands:

```
Paint
Erase
```

These two commands determine what happens when the user starts sketching. If the first is selected then a trail of colour is left behind the cursor. With the second a trail of background colour is left, giving the effect of an eraser.

The trail is drawn as if the cursor was an edged pen with a fixed width and angle. Originally a command "Change pen" was used to set up a different width and angle. When this command was selected a small pad appeared in the menu area. The user then indicated the new pen shape by depressing switch number one in the pad to



Fount & lette

Family: kind
Style: italic
Size (decipoints)
Space table: supe

Letter: lowercase

Figure 7.5
A rough sketch - the current pen is very narrow, for refining the sketch. The pen shape can be seen in the pen pad, above the options menu.

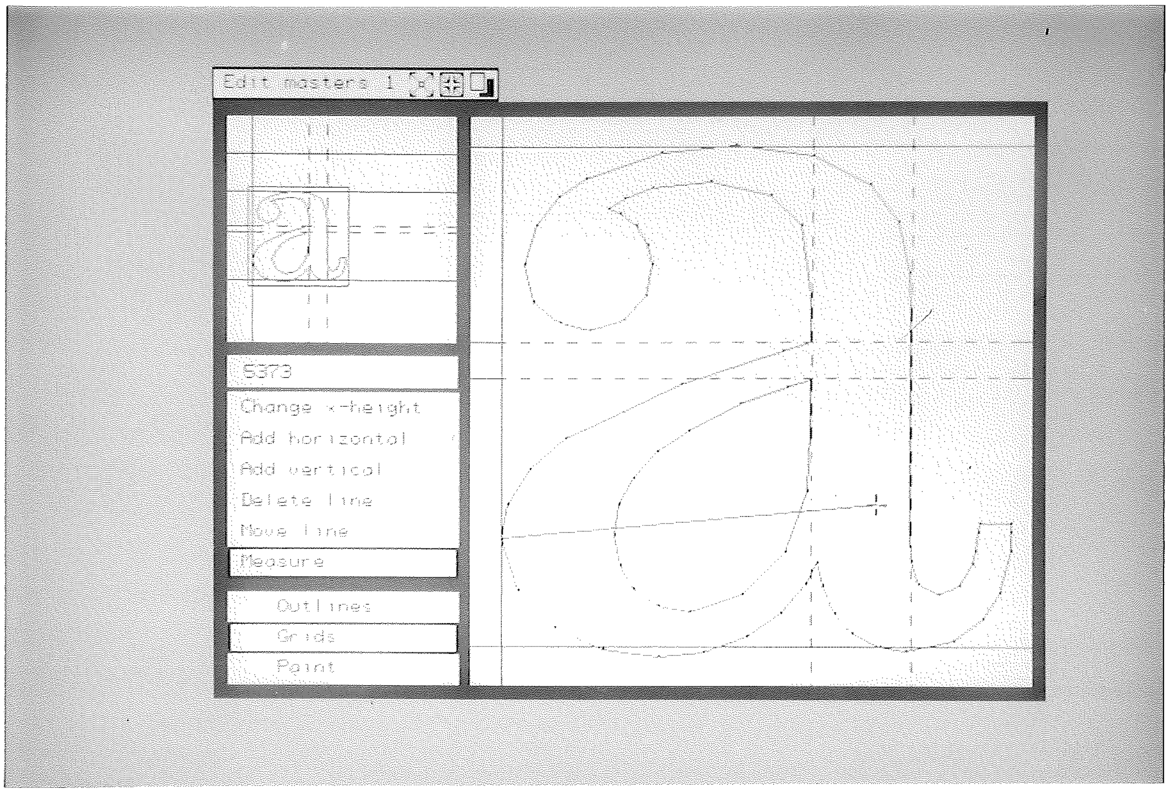


Figure 7.6
Measuring in grids mode.

indicate one end and then moving to the other end before releasing the switch. A rubber-band line was drawn as long as the switch was held down to show the pen that would be formed if the switch was released at that moment. Once the switch was released the pad was removed from the screen. One problem with this implementation was that the current pen shape could not be seen until the user started drawing. This need for some form of feedback led to the introduction of a special pen pad, showing the current pen, that is displayed whenever this mode is selected (figure 7.5). Instead of the "Change pen" command the pen shape is changed by pointing in this pad directly. Another way to provide feedback about the pen shape would be to make the cursor take on the shape of the pen. This could be done by a special transparent cursor for this window on which the pen shape was drawn. A further considerable improvement would be to provide pens of arbitrary shape, for example circles or ellipses. Although calligraphic forms are important in type face design they are not the only ones used. The provision of other pen shapes would make it easier to draw non-calligraphic forms. On the other hand, a fine edged pen provides a good deal of freedom and so the extra effort has not yet been expended in this area.

There are currently no fast pixel copying operations on the Rainbow workstation and so the trail of the pen is formed by repeatedly drawing lines of the appropriate angle and length. An unexpected and interesting effect of the slowness of this operation is that the trail is not solid colour but appears faint because of the gaps between the strokes making it up. If the user then keeps working over one particular area the colour becomes more dense, as it would in a pencil sketch. The drawing of other pen shapes would be even slower but until some experiments are tried it is hard to anticipate the effects.

Grids

This layer provides grids to be used as guidelines in sketching or creating outlines. Cap-height, baseline and left margins are always provided and the user can add other horizontal or vertical guidelines to these. In particular the user will almost certainly provide an x-height line, and this can be stored in a fount database window for use with other letters of the same fount. The other guidelines can also be stored if the user wishes.

The mode-specific menu contains the following commands:

```
Measure
Change x-height
Add vertical
Add horizontal
Move line
Delete line
```

When "Add vertical" or "Add horizontal" is selected a new line is created every time switch number one is depressed in the working area. The position of the line is not fixed until the switch is released, allowing it to be moved around for accurate positioning. These guidelines are distinguished from the main grid lines by being drawn with dotted lines.

Initially an x-height line is not provided. The "Change x-height" command is provided for setting it up so that it can be distinguished from other horizontal guidelines. If it is not already present, an x-height line is created when switch number one is depressed in the working area. The line is fixed in position when the switch is released. If the line already exists it is simply moved to the new position. Within this window this line serves merely as another guideline but the x-height forms part of the fount data and so must be kept separate from the other grids. It is drawn with a solid line rather than a dotted line to show this distinction.

The guidelines can be moved or deleted using the appropriate commands from the menu. When the switch is pressed the nearest line is moved or deleted. If the line is being moved it is not finally fixed in position until the switch is released.

When "Measure" is selected the distance, in internal units, between where switch number one was pressed and where it was released is displayed in a special pad. The value is constantly updated as the cursor moves with the switch held down. A rubber-band line joining the two points is also displayed. This remains on the screen when the measuring is finished so that it can be used to guide sketching or creation of outlines. The line and the display pad are removed when another grid mode command is selected. The photograph in figure 7.6 shows a typical grid, with measuring in progress.

Outlines

This layer provides the facilities for creating and manipulating outlines, the main product of this window. The outlines making up a letter are stored as linked lists, as described in the previous section. One part is always current and only strands in this part can be worked on and any new strands created are added to this part. The non-current parts are distinguished by being displayed in a less bright colour than the current part. The description that follows is of the current state but experimentation is still going on. When the system has been more extensively used it is quite possible that a different set of commands may be considered more suitable.

The menu for this mode currently contains the following commands:

```
Move Points
Add Point
Link / unlink
Create line chain
Delete line chain
TRANSFORMS >
PARTS >
```

The last two items are in fact pointers to secondary menus as there is not enough space for all the commands in one menu. The most frequently used commands for creating and manipulating line chains have been kept in the first menu and the rest of the commands have been divided into two groups for the secondary menus. These are commands to carry out simple transformations:

```
Fix / free
Filter section
Smooth section
Translate
Scale
< BACK
```

and commands for reorganising the part structure of the outline:

```
Next part
Create part
Delete part
Float line
Drop line
< BACK
```

The command BACK in these two menus brings back the initial menu. In order to give a fairly connected account the commands in the three menus will be described in the order in which they might be used.

To start a new line chain the command "Create line chain" is selected. When switch number one is pressed in the working area a new point is created. When the first point of the line chain is created then the strand header block is created and linked into the list of strands. If there are not yet any strands or parts in this outline then a part descriptor is created and the new strand becomes the first strand of this part. When the switch is held down, as was described earlier, movement of the cursor can either result in the creation of more points or else in the repositioning of the point just created. Both methods are provided in Imp and the user swaps between them by pressing the "c" key on the keyboard. This is a temporary measure to allow experimentation and eventually either one version will be abandoned, or else a less cryptic method of selecting the method will be provided. As long as this command remains selected any new points are added to the same line chain and are displayed linked together. When all the points have been created another command, or no command, is selected to signal the termination of the line chain.

The first point of a new line chain is not automatically linked to the last. Instead the command "Link / unlink" is provided which allows points to be explicitly linked or unlinked as desired. Pairs of points are selected by pressing the switch down with the cursor near one and releasing it with the cursor near the other. If the first point is an end point then as the cursor approaches other end points a line is drawn linking the two points. If the cursor is released with the linking line present then the data structure is modified to link the two points. This may be the closing of a line chain or else the linking of two separate line chains into one. The structure only allows end points to be linked and so moving the cursor towards other points has no effect. If the first point is not an end point then the second point can only be a neighbouring point. In this case the linking line is removed and if the switch is released then the link is removed in the data structure. If an unlinking operation results in a line chain containing only a single point then this line chain is deleted as it is meaningless to have a line chain of only one point. This is the quickest way to delete single points from the structure. Unlinking points can also result in the creation of two line chains from one.

When "Delete line chain" is selected the line chain nearest the cursor is deleted when the switch is pressed. The quickest way to delete a number of adjacent points from a line chain is to unlink the section and then delete the resulting line chain. If the last line chain in the current part is deleted then the part descriptor is deleted and the next part, if one exists, becomes current.

The points making up a line chain will probably need to be moved around to get the desired shape. The command "Move points" is selected and then the cursor is used to select and reposition points. As was discussed in the first section of this chapter two methods have been used in Imp, and as with creating line chains, neither has proved overwhelmingly superior. The first is to press the switch near a point and then to drag the point around until the position is correct. The other method repositions a whole series of points along a path drawn by holding the switch down and moving the cursor. As with creating line chains, both methods are provided and the user swaps by pressing the "m" key on the keyboard. Again, a decision needs to be made to either abandon one method or to integrate them both into the graphical interface.

The initial version of an outline is usually rather coarse and so extra points must be added to refine the details. The command "Add point" can be used for adding single points in specific places. When it is selected, a new point is added between the nearest pair of points whenever the switch is pressed. This new point is created at the cursor position. The smoothing command, described below, automatically adds extra points along a curve fitted through a series of existing points.

The command "Fix / free", in the TRANSFORMS menu, is provided to mark points as immovable and to act as delimiters. When this command is current any points selected are marked as fixed if they were free or free if they were fixed. Fixed points are shown by a larger marker than free points and are also highlighted in the reference area. Fixed points cannot be moved by any commands, including transforms applied to the whole outline. This command can be applied to single points by placing the cursor near the point and pressing and releasing the switch. A sequence of adjacent points can be selected in one movement by pressing the switch with the cursor at one end of the sequence, and with the switch held down moving the cursor near the second point of the sequence to give the direction. The switch is then released with the cursor by the final point. The points that would be selected by releasing the switch at any time are highlighted to aid in the making of the correct selection. When a whole sequence of points is selected they all take on the new property of the first point, regardless of their own current state.

The TRANSFORMS menu provides several commands, not all strictly transformations, but with more global effects than the commands discussed so far. The command "Smooth section" can be applied to a section of line chain delimited by fixed points. A new point is added between each pair of points in this section, positioned on the Overhauser curve through its two neighbours on either side. For the end segments the end points are used twice rather than bringing in neighbouring points outside the section. This is done because the ends of sections will quite often coincide with discontinuities in the shape and very strange results would be obtained by using points outside the section. An even better result may well be obtained by creating end points that lie on a linear extrapolation of the last section, but this has not yet been tried.

Superfluous points can be removed from a section by applying "Filter section". Again the section is delimited by fixed points. The section is filtered to the current tolerance, which can be changed by the user through the "Change parameters" command in the general menu (see below).

"Translate" and "Scale" both apply to all the free points of the current part. Once selected, these commands are applied by drawing a vector in the working area. For translation, all the points are moved by the amount and in the direction given by the vector. For scaling, the points are scaled from the origin of coordinates such that the starting point of the vector is moved to the position of the end point. If the vector crosses one of the axes of coordinates the effect is to reflect the outline about this axis as well as scale it.

All of the previous commands only work within the current part but the commands in the PARTS menu provide for moving between and reorganising the parts of an outline. The command "Next part" differs from the commands discussed so far in that it is applied when it is selected rather than waiting for any user action. The next part in the part list is made current, going back to the first part when the end of the list is reached. This method of selecting a part for working on should not be too laborious as there are generally very few parts in an outline. Similarly "Delete part" is applied immediately, deleting the current part and making the next part current. A part is also deleted if all its line chains are deleted.

It is possible to extract line chains from the outline data structure, in order to reorganise the structure. These line chains are referred to as floating line chains and they are drawn in red to mark them out. The command "Float line" is applied to line chains by pointing at them using the cursor. These line chains can be gathered from more than one part as desired. One way to incorporate these line chains back into the structure is to use "Drop line". When this command is current, any floating line chain that is selected is linked into the current part and is removed from the list of floating line chains. Hence line chains can be moved freely among existing parts. Another way to use these floating line chains is to select "Create part". This is another command that is applied immediately, creating a new part from the floating line chains. This becomes the current part.

As is described below, when an outline is fetched from a master fount window its parts are added to those already in the editor window. This means that the PARTS commands have an important part to play when a feature is to be extracted from one outline and incorporated into another. A piece of line chain can be cut out of one outline, floated and then dropped into another. The unwanted line chains from the original outline can be removed by deleting the parts containing them. It should also by pointed out that the general command described below to destroy the outline does not apply to any floating line chains. Hence, pieces of line chain can be extracted from an outline and then the rest of the outline can be cleared out of the way by destroying it rather than explicitly deleting each part. The floating line chains can then be incorporated into another existing line chain or can be used as the basis of a new outline.

General commands

These are provided in a pop-up menu and consist of "once-off" commands rather than ones for continuous application:

```
Get outline
Send outline
Fill outline
Destroy outline
Get grid
Send grid
Destroy grid
Copy painting to file
Get painting from file
Clear painting
Copy pen to file
Get pen from file
Change parameters
Finish
```

The commands to get and send outlines and grids provide communication with master fount windows. Both commands to get data cause a request of type `Request.getData` to be made on the master fount channel. These requests are satisfied by data, sent from a master fount window, consisting of a pointer to the fount vector and a pointer to the current slot in the fount window. If the command was "Get outline" then the outline from the current slot is added to any outlines already present in the editor window, after being converted into the linked list format. The last part to be added becomes the current part. If the command was "Get grid" then the grids stored in the fount vector replace any currently associated with the editor. The grid information includes the x-height, total range of x and y coordinates allowed and vertical and horizontal guidelines.

The commands to send data cause a request of type `Request.sendData` to be made on the master fount channel. It is satisfied by a request to get data made in a master fount window and again, the data from the satisfying request is a pointer to the fount vector and a pointer to the current slot in the fount window. For "Send outline" the outline in the editor window is copied into the current slot of the fount window, replacing the previous contents. The menu item corresponding to this slot is redrawn to display the new contents. For "Send grid" the grids in the editor window are copied into the fount vector, replacing any previous grids.

The commands "Destroy outline" and "Destroy grid" have the expected effects, although the former does not destroy floating line chains, these not being part of the main outline structure. Both these commands can be cancelled so that important data is not easily deleted accidentally.

"Fill outline" causes the outline to be scan converted and the resulting filled areas are displayed in the paint pad. This is very useful for checking a design as ultimately it will be used as a filled area rather than an outline. The command "Clear painting" can be used to clear the painting pad, either after filling an outline or after doing some painting by hand.

Both paintings and pen shapes can be copied to files for use later. These commands display a form in which the user specifies the name of the painting or pen. Imp then automatically constructs the file names and either reads or writes the appropriate file.

"Change parameters" causes a form to be displayed with certain parameters associated with this window. The user can type in this form to change the parameters and then press the return key to signal that he is satisfied. The parameters are then applied. Currently the parameters are the minimum and maximum x and y coordinates that can be accommodated in this window and the tolerance value. By changing the maximum and minimum coordinates the amount of space available for ascenders and descenders or for abnormally wide characters can be changed. Initially these values are set up to accommodate a standard roman fount, suitable for most purposes. These values form part of the grid data for this window. The tolerance is a value expressed in internal units that is used to indicate what is considered to be a significant distance. For example, in the discussion above the phrase "near a point" was used on a number of occasions. This means being within the distance given by the tolerance. When the nearest object to the cursor is to be found it is only selected if it is within the tolerance to prevent funny effects if the user presses buttons with the cursor apparently in the middle of nowhere. The other use of the tolerance value is in filtering out excess points from a line chain. Because the tolerance value is in internal coordinates the actual physical distance on the screen will vary with the magnification of the object being worked on.

The command "Finish" is issued to close down this window.

7.3 Compare Masters Window

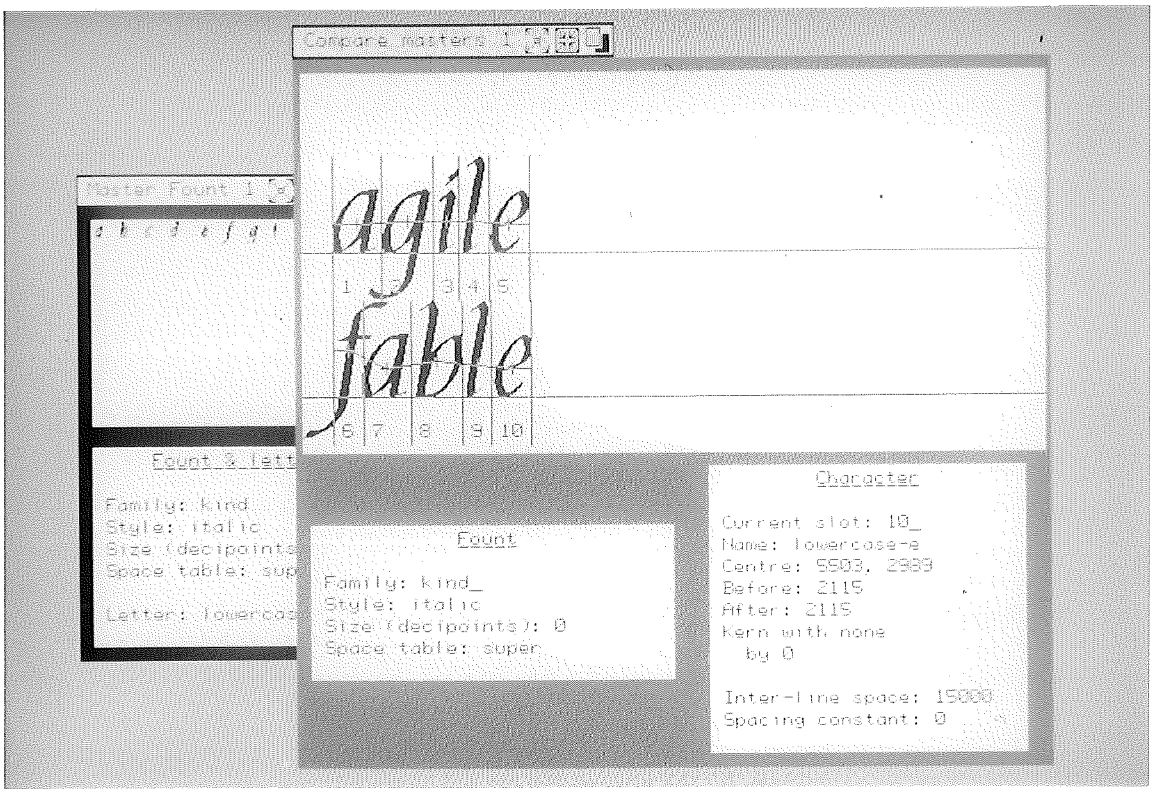
While a letter outline is being worked on it must be considered in combination with other letters because the space formed around a letter is as important as the shape of the letter itself. This window allows up to ten letters to be retrieved from the fount database and placed side by side, spaced according to the information provided in the database. The letters are displayed in two lines of five so that the inter-line spacing can be seen too.

Overall organisation

This window has two main areas, an area for displaying characters at the top and an area containing two forms at the bottom. The display area shows the visual appearance of the spacing and the forms give the numeric values of the parameters as well as the names of the fount and characters. The characters are stored in numbered slots, with slots one to five forming the top line in the display area and slots six to ten forming the bottom. One slot is always current and the spacing information for the character in this slot can be updated.

The left hand form displays the name of the fount, which is obtained from a master fount window and cannot be changed. The right hand form displays the current slot number and the information associated with the character in this slot. The name of the character is fixed but all the other information can be changed. The spacing information for a character is given in the form of the coordinates of a centre and the distance to be left before and after this centre. Also kerning values can be given to be added when this character is followed by other specific characters. This form also displays the inter-line spacing value and the spacing constant to be added between all the characters in this window. The spacing constant can be used to experiment with overall tightening or loosening of the setting.

The original version of this window made extensive use of the special facilities of the Rainbow Workstation for blending images. Each letter consists of a filled


Compare masters 1

Master Fount 1

Fount & lett

Family: kind
Style: italic
Size (decipoints
Space table: sup

Letter: lowercas

Fount

Family: kind_
Style: italic
Size (decipoints): 0
Space table: super

Character

Current slot: 10_
Name: lowercase-e
Centre: 5503, 2989
Before: 2115
After: 2115
Kern with none
  by 0

Inter-line space: 15000
Spacing constant: 0

Figure 7.7
"Compare masters" window.

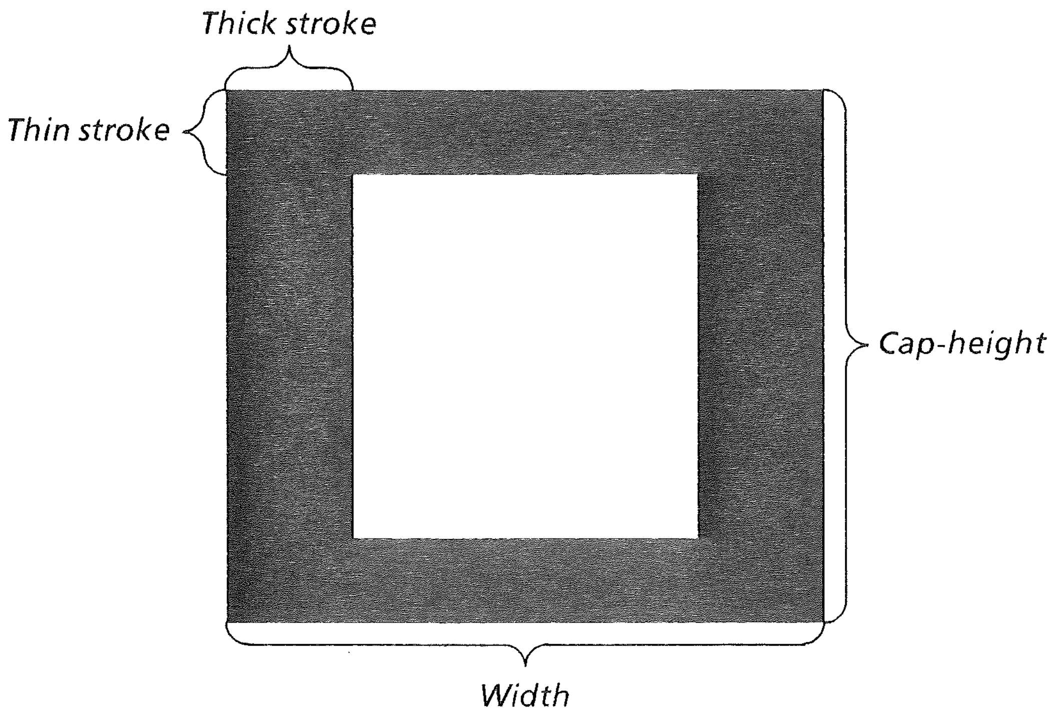


Figure 7.8
An uppercase canonical letter.

polygon which takes a significant amount of time to draw and so a considerable delay would occur if the letters were repositioned by redrawing them. Instead the letters were drawn once, each on a separate transparent pad. These pads were placed over a white background and letters were repositioned by moving the pads, a relatively fast operation. The letters must be on transparent pads rather than ordinary opaque pads because letters are often positioned so that their bounding boxes overlap and parts of adjacent letters may be obscured. The whole area was overlaid with a translucent grid showing the baselines, centres and before and after distances for each letter. Unfortunately the resulting high number of pad boundaries along a scan line stretched the Rainbow hardware and microcode to the limit. In theory the system should degrade elegantly but in practice the behaviour was unreliable to the extent of making this window unusable. Hence the ideal of rapid, interactive repositioning was abandoned and the characters are now all drawn in one ordinary pad and are repositioned by redrawing. The translucent grid is drawn in an extra plane of the pad containing the characters. The photograph in figure 7.7 shows this window in use.

Setting up the spacing

The user sets up or changes the spacing by typing in the fields of the right hand form. The character name fields cannot be changed as these are fixed in the master fount but all the other fields can be updated at any time. The results of any changes are not seen until carriage return is hit, when the characters and grid are redrawn. Once the "current slot" field has been updated no other field can be changed until carriage return is hit and the information about the character in the new current slot has been displayed. The "Kern with" field is filled in by the system if there is a character after that in the current slot. The value to kern by can be changed by the user and this value (which can be negative) is added to the spacing between the two characters.

An intention was to try using cursor movement to update the numeric fields, preferably with continuous feedback of the character positions. The display structures (see above) were initially set up to allow rapid repositioning of the characters. Unfortunately the calculation of the positions has turned out to be a rather slow process and so the feedback would not be very continuous. The current implementation of the calculation is very little optimised and so there is much room for improvement here. On the other hand the problems encountered with the display structures have ruled out interaction with continuous feedback, at least for the time being. Even without feedback it would still be useful to use the cursor to indicate distances on the displayed characters rather than typing in numbers. Hitting carriage return could be retained as the signal to recalculate the positions. This fairly simple addition may well have been done by the time the thesis is finished.

Calculating the spacing

The spacing can also be calculated automatically using a calculation based on the Logos system, introduced in section 6.1, which sets up optical letter spacing. This calculates an "optical" centre and a width for the character by a method analogous to finding the centre of gravity and weight of an object. The results of using this calculation are often very good and where the calculation does not work so well the user can override the results.

The outline is scan converted into an array of pixel values, to a resolution of 25 pixels to the cap-height. A higher resolution gives more accurate results but makes the calculation very slow. This particular resolution seems a suitable compromise, with the possibility of tweaking values by hand afterwards if the results look bad. The resulting pixel array is then passed to a procedure `Space.calculate`:

```
Space.calculate( pixels, guessWidth, spaceData, height, thick, thin )
```

All the parameters and results are given in units of pixels in the array passed as the first argument. `guessWidth` is an initial value for the character width, used for starting the iteration. A suitable value would be the width of the pixel array. `spaceData` is a pointer to a vector which will contain the results. The final three arguments give the dimensions of the canonical character, whose role is described below. The `spaceData` vector must be created six words long. The values returned are the coordinates of the centre and the width of the character, in floating point:

```
1. Two words for x coordinate of centre
2. Two words for y coordinate
3. Two words for the width
```

A special software floating point package, written by Alan Greggains [1985] is used as BCPL itself does not support floating point arithmetic. A procedure `Space.normalise` is provided that converts a floating point number to an integer, after multiplying by a specified factor:

```
integer := Space.normalise( n1, n2, factor )
```

The arguments `n1` and `n2` are the values of the two words making up the floating point value and `factor` is the multiplication factor. This can be used to convert the results back into master fount units.

The calculation is based around finding the "moment" of each pixel in the array with respect to a potential centre. Only pixels that fall between the baseline and the given height are included in the calculation. The position of the centre is repeatedly adjusted until the moments side to side and top to bottom balance. For a non-zero pixel in the array its moment is the sum of its distance in x and its distance in y from the potential centre, raised to the fourth power. The y distance is multiplied by a ratio of thirty percent, making the contribution of sideways displacement from the centre greater than that of vertical. The initial guess for the centre is taken as half the width and half the height from the origin of the pixel array. On each iteration the moments to the left and right of the centre are compared and the x coordinate changed appropriately; the moments to the top and bottom are then calculated and compared and the y coordinate changed. The amount and direction of the change in coordinates is calculated by linear interpolation from the current and previous values of the imbalance. The new potential centre is calculated such that the imbalance would be zero, given what has happened in this and the previous iteration.

Once the centre has been calculated the width is found by adjusting a canonical character until it has the same total moment as the original character. The canonical character is constructed from two vertical strokes whose thickness is given by the thick parameter and two horizontal strokes whose thickness is given by the thin parameter. The bottom edge of one horizontal stroke rests on the baseline and the top edge of the other reaches the height line. The width is measured between the outer edges of the two thick strokes, as is illustrated in figure 7.8. Because the canonical character is symmetrical the centre can be derived from its dimensions. The width is found by calculating the total moment and then adjusting the width in an appropriate direction to converge on the moment of the original character. This width is then taken as the width of the character.

General commands

The pop-up menu provides the following commands:

```
Get character + spacing
Copy character + spacing
Send character spacing
Get inter-line spacing
Send inter-line spacing
Get fount data
Calculate spacing
Clear
Finish
```

When the command to get a character and its spacing parameters is selected a request of type `Request.getData` is made on the master fount channel. This request can be satisfied by sending data from a master fount window. The character is placed in the current slot, it is displayed in the appropriate position in the lines of characters and the numeric values are displayed in the form. When the command to send the character spacing is given a request of type `Request.sendData` is made, which is satisfied by a request to get data made in a master fount window. The spacing information is copied into the slot in the master fount window. Because these spacing values have no meaning separate from the character shape, a check is made that the slot receiving the information is the one from which the character originally came. If it is not the same then the data is not saved and the user is informed of the fact. This constraint could easily be removed if it ever appeared that the users would prefer to be able to store the spacing information in any slot, regardless of the character it was originally associated with. It would then be in the user's hands to make sensible use of this freedom.

As with the characters, the command to get the inter-line spacing sets up a request of type `Request.getData` on the the master fount channel. This request is satisfied by sending data from a master fount window, but this time the fount information rather than the character information is used. When the inter-line spacing is sent back to a master fount window, a check is made to see if this is the fount whose spacing is currently being handled. If not, the user is informed and the inter-line spacing is not copied into the fount window. As with saving character spacing, this restriction could easily be removed if users requested it.

If the user is setting up inter-line spacing or intends to calculate the spacing automatically then some information is needed about which fount is being dealt with. The command "Get fount data" makes a request on the master fount channel. This is satisfied by sending data from a master fount window and the fields in the left hand form are then automatically filled in to show which fount has been selected.

The command to calculate the spacing carries out the calculation described above on the character in the current slot. The user is first requested to get the fount data if he has not already done so. This is because values such as the x-height are needed in the calculation. A form is then displayed in which the parameters required can be filled in by the user. These values are the height and the widths of the thick and thin strokes of the canonical character. The user quotes them in the standard internal units and the system converts them to pixel array units before starting the calculation. The user must also indicate whether an uppercase or lowercase canonical letter is to be used. The previous values given are always shown in this form, making it easy to apply exactly the same calculation to a series of letters. When the correct values are in the form the calculation is started by hitting the return key. If the user does not want the calculation to be carried out after all then the cancel key can be hit. The results are displayed in the character's form, converted back into internal units. The character display is also updated.

"Clear" wipes out all the data stored in this window, ready for a fresh start. The cancel key can be used if the user issues this command by accident. "Finish" closes the whole window down.

8. Raster Founts

The raster founts are derived from master founts rather than being created from scratch, hence the facilities for manipulating and editing the raster founts are much less extensive than those for the master founts. Raster founts are created by scan converting master outlines to a particular resolution of grid. I know of no automatic algorithm that does this completely correctly and so some editing in usually required to make the results usable. To help in assessing the founts a window is provided to display some text using a particular fount.

8.1 Creating raster founts

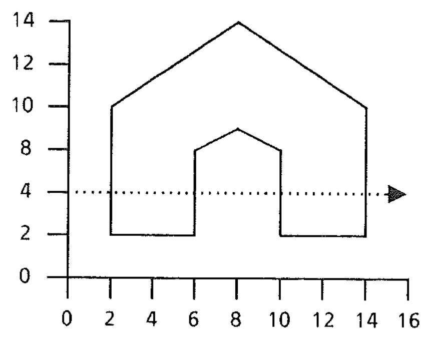
The algorithm used for scan converting a master outline is based on one written by John Wilkes. It copes with regions defined by any number of separate strands, determining the colour of a region by the number of strand crossings taken to reach it from infinity. Odd numbers of crossings give the foreground colour and even numbers the background.

Implementing the scan conversion algorithm

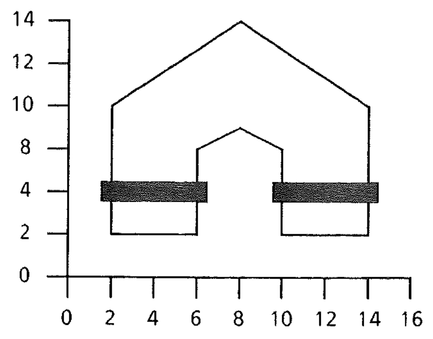
The strands making up an outline are each defined by a vector of coordinates. These coordinates define points to be joined by straight lines to form the required outline. The last point is always joined back to the first point to ensure that the outline is closed. For each scan line in the bit map, all the intersections with the lines making up the outline are found. These intersections are sorted into increasing x coordinate and then the pixels between each pair of intersections are filled. Figure 8.1 illustrates this basic algorithm.

A vector is created with one word for each y value in the bit map. Each word contains a pointer to a sorted list of the crossings of the particular scan line with the strands making up the outline. The beginning of the list is assumed to be outside the outline and so the first crossing introduces a transition from background colour to foreground. The crossings are located by using a simple digital differential analyser to generate the x coordinate corresponding to each y coordinate that the line passes through.

The algorithm used is not in fact as simple as implied in the previous paragraph. Problems arise at the vertices where points are shared between two lines. If both lines generated a point at the vertex there would be two crossings recorded here and the resulting parity of crossings would be wrong. Because the strands have direction it is possible to solve this problem by having each line generate a crossing at its last point but not at its first. Each vertex then only has one crossing recorded. Unfortunately this is not a complete solution, as figure 8.2 shows. Where the slope changes from up to down, or vice versa, a single crossing at the junction gives the wrong result. To overcome this, a test is done at the start of each line to see if the direction of slope has changed. If it has, a point is inserted at the beginning of this line to produce the necessary doubling of crossings (figure 8.3).

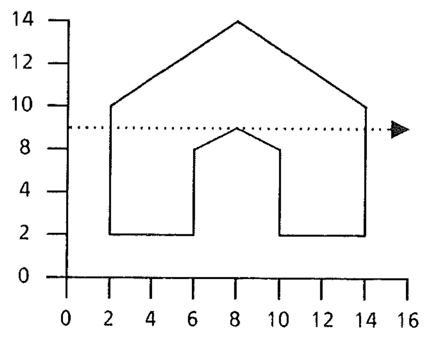


The list of intersections for this scan line is (2, 6, 10, 14).

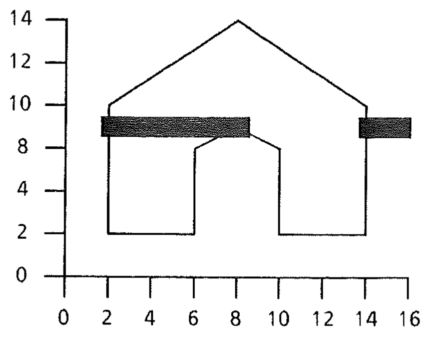


The pixels from 2 to 6 and 10 to 14 are filled.

Figure 8.1
The basic scan conversion algorithm

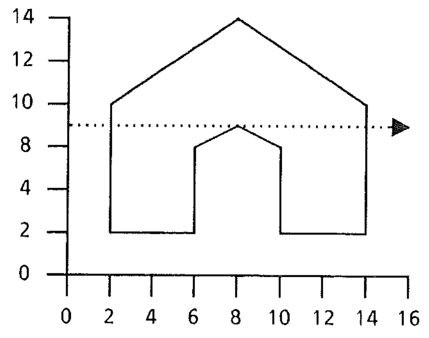


The list of intersections for this scan line is (2, 8, 14).

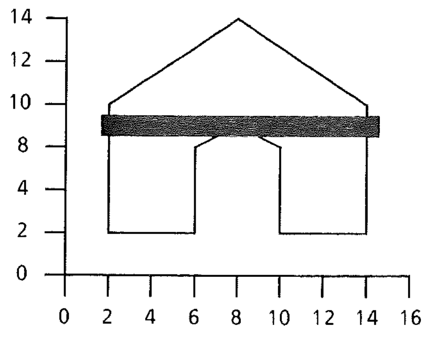


The pixels from 2 to 8 and 14 to the right border are filled.

Figure 8.2
The result obtained if one intersection is generated at each vertex



The list of intersections for this scan line is (2, 8, 8, 14).



The pixels from 2 to 8 and 8 to 14 are filled.

Figure 8.3
The correct result, obtained by doubling the intersection when the slope changes from one section to the next

Problems that arise
The scan conversion algorithm has no understanding built in to it of the structure of the characters. As has already been mentioned in chapter two, it is possible for features that are meant to be the same in different characters to come out differently. For example, vertical and horizontal strokes may differ in width depending on how they register on the pixel grid. This is particularly noticeable for lower resolutions and smaller sizes where a single pixel may form a substantial proportion of the width of a character. Another effect is for symmetrical features to come out assymmetrical. The raster editor is provided so that these errors can be corrected easily.

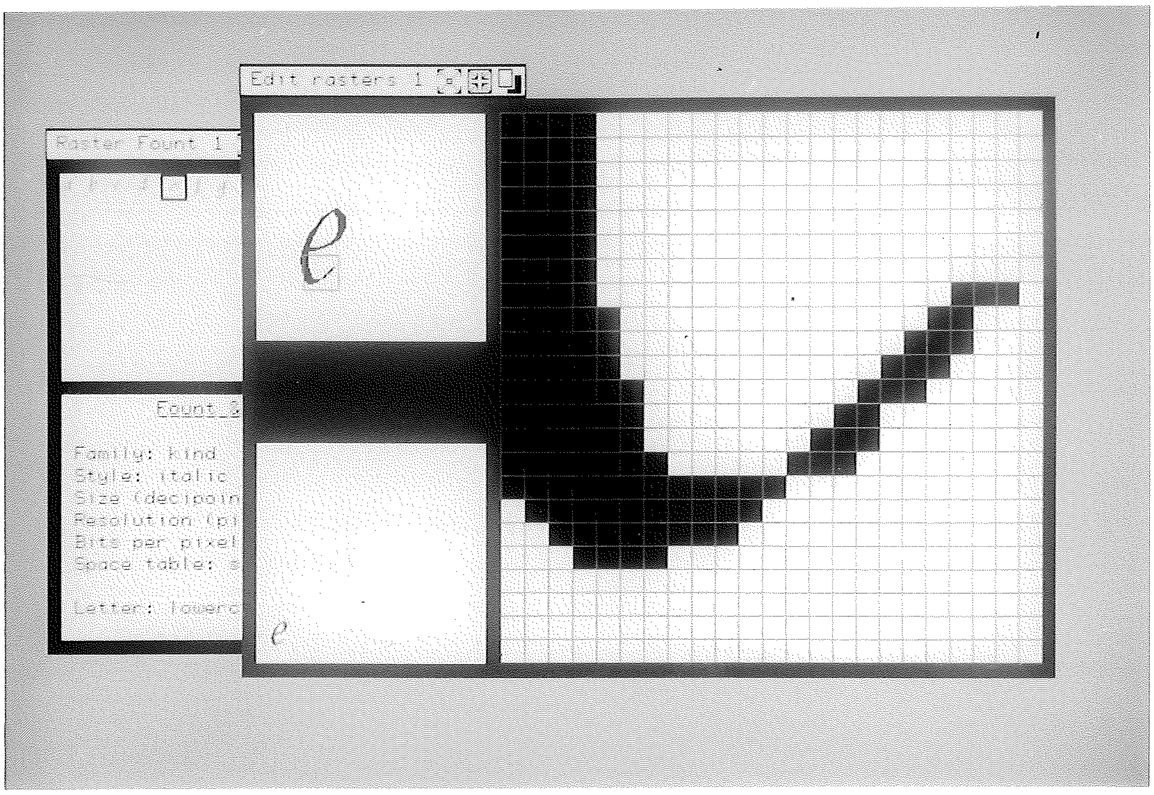
It is possible to get round these problems by such techniques as specifying what width the vertical and horizontal strokes are to be. The scan conversion algorithm can then be written so that it detects these strokes and fixes their width. Ideally such enhancements would be incorporated into the algorithm used in Imp, but lack of time has prevented this. Instead Imp relies on easy, interactive editing of the rasters to produce correct results.

8.2 Raster Editor

This window provides facilities for simple editing of raster characters to put such problems right. A photograph of this window can be found in figure 8.4.

Overall organisation
As in the master outline editor there is a small reference area where the whole character is shown and a working area where a part of the character is selected for working on. The region shown in the working area is selected by moving a box around in the reference area using the cursor. The working area displays a region 25 pixels square, with each pixel displayed as a 5mm square. The number of pixels that can be edited at one time could be increased by decreasing the size of the pixels in the working area. A 5mm square can be very easily selected with the cursor and so a smaller size would probably still be usable, although this experiment has not actually been done. The proportion of the character that the working area represents will depend on both point size of the fount and the resolution of the destination device. For a particular character this is obviously fixed and so the size of the box in the reference area cannot be changed. When the user points in the reference area the box is moved around centred on the cursor position until the user releases the switch. The new area is then displayed in the working area and pixels can be selected and updated.

The character is displayed in the reference area at the maximum size that can be accommodated so that areas for working can be easily and accurately selected. Sometimes more than one screen pixel will be used for one pixel in the character and for others one screen pixel represents several character pixels by using super-sampling. The algorithm used is the same as that used in the text display window and is described in section 8.3. An additional area below the reference area displays the character at a specified (integer) magnification of the size it will appear when printed on the destination device. By default this magnification is one but larger magnifications can be useful for small sizes of character. Again the super-sampling algorithm is used.

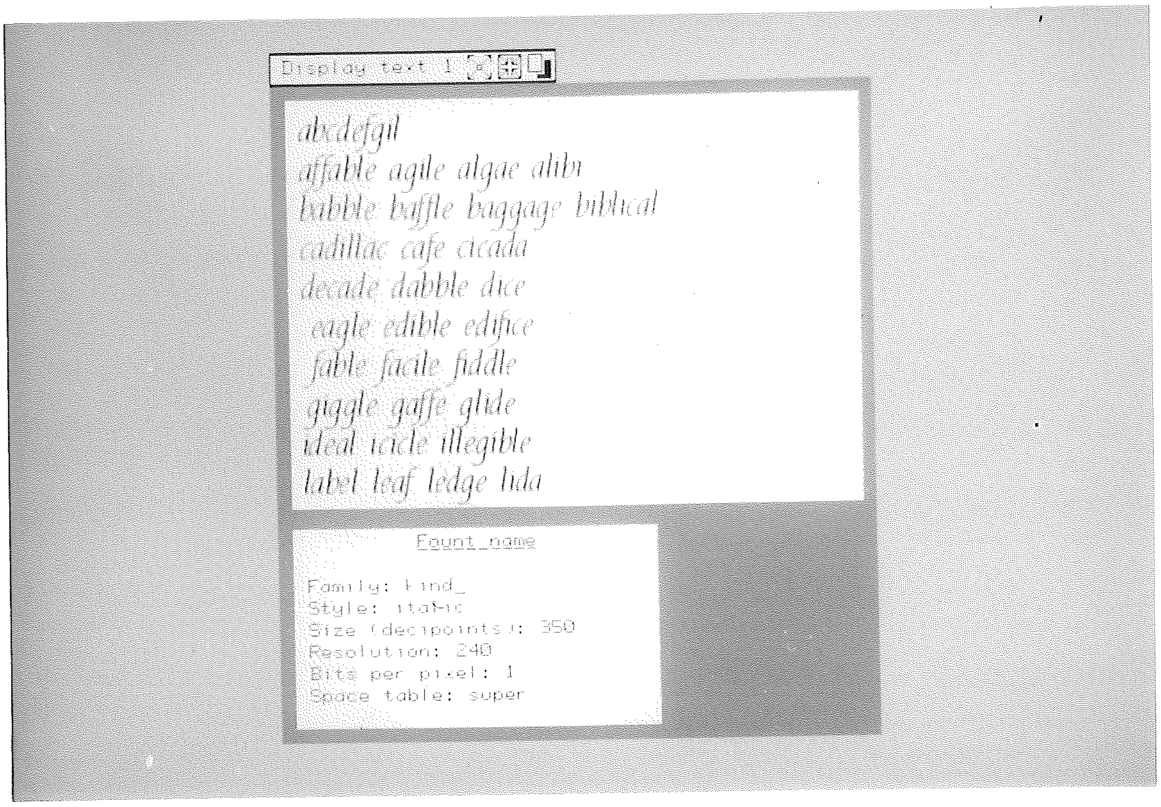


Fount_&

Family: kind
Style: italic
Size (decipoin
Resolution (pi
Bits per pixel
Space table: s

Letter: lowerc

Figure 8.4
Raster editor window.



abcdefgil
affable agile algae alibi
babble baffle baggage biblical
cadillac cafe cicada
decade dabble dice
eagle edible edifice
fable facile fiddle
giggle gaffe glide
ideal icicle illegible
label leaf ledge lida

Fount_name

Family: kind_
Style: italic
Size (decipoints): 350
Resolution: 240
Bits per pixel: 1
Space table: super

Figure 8.5
Text display window.

The working pad is two bits deep to allow for grey scaled characters with two bits per pixel. This could very simply be changed to three or four bits if it was desired but so far only founts with one or two bits per pixel have been edited. Allocating only two bits per pixel for the working pad saves a considerable amount of graphics memory. When a character is fetched for editing, a colour lookup table is set up depending on the number of bits per pixel for this character. The other two pads are four bits deep so that when a character is super-sampled to fit it can be displayed using sixteen levels of grey scale. This prevents the loss of too much information about the character shape, although two or three bits may prove adequate. If the character does not require super-sampling then the lookup table is set up to fit the actual number of bits per pixel. The grid in the working area and the box in the reference area are drawn in transparent overlay pads.

Editing a character

A character is edited by pointing at pixels in the working area. Every time a pixel is selected the value of the pixel is increased by 1 with the value wrapping round to 0 when the maximum is reached. For one bit deep pixels this means that the value simply alternates between black and white. With more than one bit the colour cycles through a grey scale. If the select button is held down and the cursor is moved the new colour is then swept into all the pixels that the cursor moves over.

An inconvenience with this method of updating pixels is that it is rather easy to overshoot the value being aimed at and then have to go through the whole cycle again. Obviously this is no problem for one bit deep pixels but if founts with three or four bits were being edited this could be a serious problem. A different interface was tried in order to address this specific problem. When switch number one was pressed down in a particular pixel the value of that pixel was changed as the cursor was moved horizontally across the screen, the value being fixed when the cursor was released. The value was repeatedly incremented by one as the cursor moved right and decremented as it moved left. Hence it was easy to move a pixel value backwards and forwards through the whole range but at the expense of no longer being able to sweep a colour through a large number of pixels. Each one would have to be individually adjusted. Because the overwhelming majority of raster founts are only one bit deep this interface was rejected as it made the editing of one bit deep founts unnecessarily tedious. The initial interface could be improved if large numbers of grey scale founts were to be edited by providing a command to reverse the direction a pixel value is changed on a switch depression.

When a character is fetched for editing its pixels are copied into an array big enough to accommodate the largest sized character in its fount. Areas for working on must always fall within this array otherwise there would be nowhere to store the pixel values. The procedure that repositions the box in the reference area does not allow the box to move beyond the extent of the array. The size of this array can be changed if desired by using the "Change parameters" command in the pop-up menu. Characters cannot be created from scratch and so no editing is allowed until a character has been fetched and the array set up.

General commands

The pop-up menu provides the following commands:

```
Get character
Send character
Clear
Change parameters
Finish
```

The command to get a character sets up a request of type `Request.getData` on the raster fount channel. This request can be satisfied by sending data from any raster fount window. A pixel array is created so that it can accommodate the maximum range of coordinates, as specified in the fount vector. The pixels for the chosen character are then copied into this array and the character is displayed in the reference and display areas. Initially an area in the bottom left corner of the pixel array is selected for display in the working area.

"Send character" is used to store the character back in a raster fount window. A request of type `Request.sendData` is made on the raster fount channel. The command "Get data" issued in a raster fount window will satisfy this request. The new pixel array replaces that in the slot specified in the reply and the maximum and minimum coordinates for the fount are updated. The menu item for the updated slot in the raster fount window is redrawn so that the new character can be seen there.

The pixel array can be discarded by selecting "Clear" and the whole window can be closed down by selecting "Finish". Selecting "Change parameters" results in a form being displayed showing the current maximum and minimum coordinates of the pixel array and the magnification being used for the display area. Any of these values can be updated and they are applied when carriage return is hit. If the user decides he does not wish to use the new values then the cancel key can be hit to discard the form.

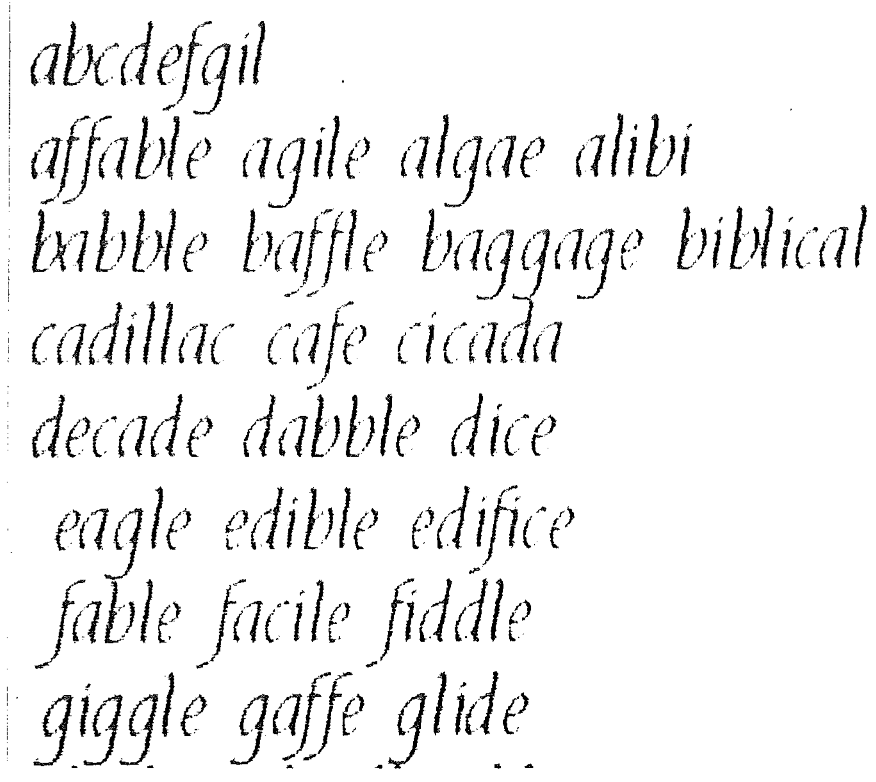
8.3 Text Display Window

The final test for any type face is how it looks when used for setting whole pages of text. This window gives a preview on the screen of a piece of text using any of the raster founts stored in the system. The text is displayed at the size it would appear on the final output device, or some specified multiple of this size. In general the resolution of the final output device will be much higher than that of the workstation screen and so super-sampling is employed to display the characters. Figure 8.5 shows an example of this window. Figure 8.6 shows some text photographed from the screen (about 60 dots per inch) and the same letters printed on a laser printer (240 dots per inch).

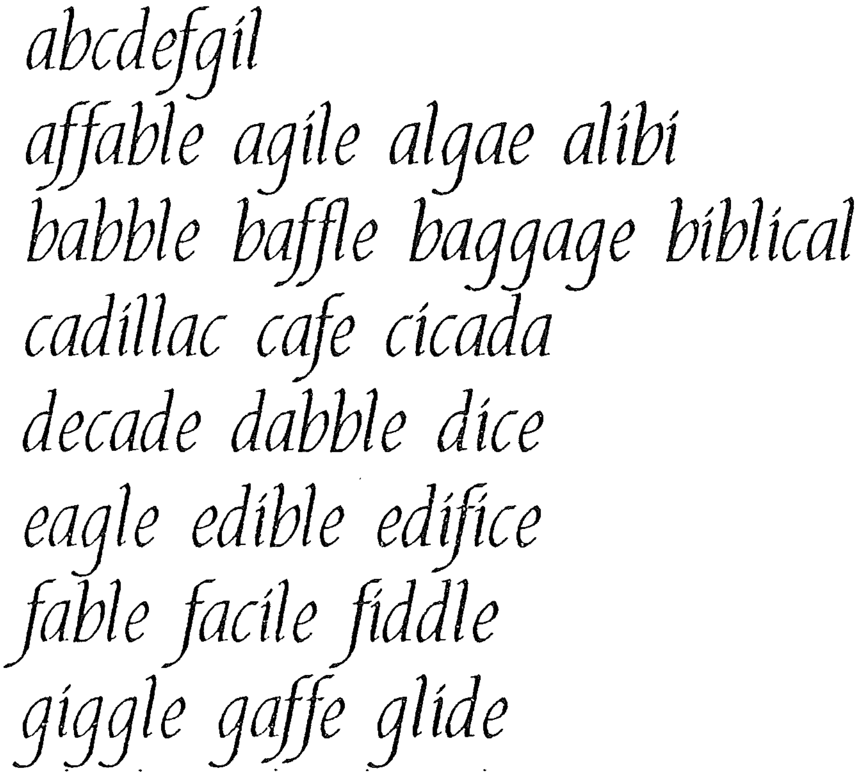
Overall organisation

The top half of the window contains the pad to display the text and the bottom half contains a form giving details about the fount, obtained from a raster fount window. The text is not formatted to fit the space available before being displayed as this seems somewhat unnecessary. All the space characters in the text file are given a fixed size and the line breaks are done where they occur in this file. If a line is longer than the width of the display pad it is simply truncated. The spacing between characters is calculated from the values given in the raster fount window.

The display pad is four bits deep, giving sixteen levels of grey for representing the super-sampled text. The super-sampling method used allows characters to be placed at arbitrary offsets with relation to the pixels of the display so that the spacing can



abcdefgil
affable agile algae alibi
babble baffle baggage biblical
cadillac cafe cicada
decade dabble dice
eagle edible edifice
fable facile fiddle
giggle gaffe glide



abcdefgil
affable agile algae alibi
babble baffle baggage biblical
cadillac cafe cicada
decade dabble dice
eagle edible edifice
fable facile fiddle
giggle gaffe glide

Figure 8.6
Some text set using letters designed by Lida Lopez Cardozo, using Imp.
Top - on screen; bottom - printed on paper.

be accurately shown. Because a character can appear at any offset with respect to the pixel grid it is not possible to super-sample the characters before displaying the text. All the calculations are done as the text is being displayed, making it slow but accurate.

The pop-up menu provides two commands:

```
Get fount data
Display file
Finish
```

Before a file can be displayed a fount must be obtained using the command "Get fount data". The fount must then be sent from a raster fount window. The command "Display file" presents the user with a form to be filled in with the name of a file to display, the magnification to be used and the spacing constant to be added between all the letters. The text is then written in the display pad. If the text file cannot be found then the user is informed and he can then take appropriate action n before trying again. "Finish" closes the whole window down.

The super-sampling algorithm

The basis of the super-sampling algorithm is shown in figure 8.7. A grey value corresponding to the percentage that are black of the character pixels falling within the screen pixel must be calculated. For each screen pixel, the values of the pixels in the original character that fall within it are summed to give **sum**. **sum** includes an appropriate fraction of the value of those pixels that do not lie entirely within this screen pixel. Two additional values are required before the value of the screen pixel can be calculated. The first is the theoretical maximum value for the sum of the pixels of the original character that fall within the screen pixel, **maxSum**. This value is calculated assuming every pixel to be black. The second value is the maximum value that the screen pixel can take, **maxPixel**, which is determined by the number of bits per pixel. The value of the screen pixel is then given by:

```
pixelValue := (sum / maxSum) * maxPixel
```

Essentially the range 0 to **maxSum** has been mapped onto the range 0 to **maxPixel**. This process gives rise to a certain amount of inaccuracy from rounding the value down to the nearest screen pixel value. **pixelValue** in fact represents a value for **sum** of:

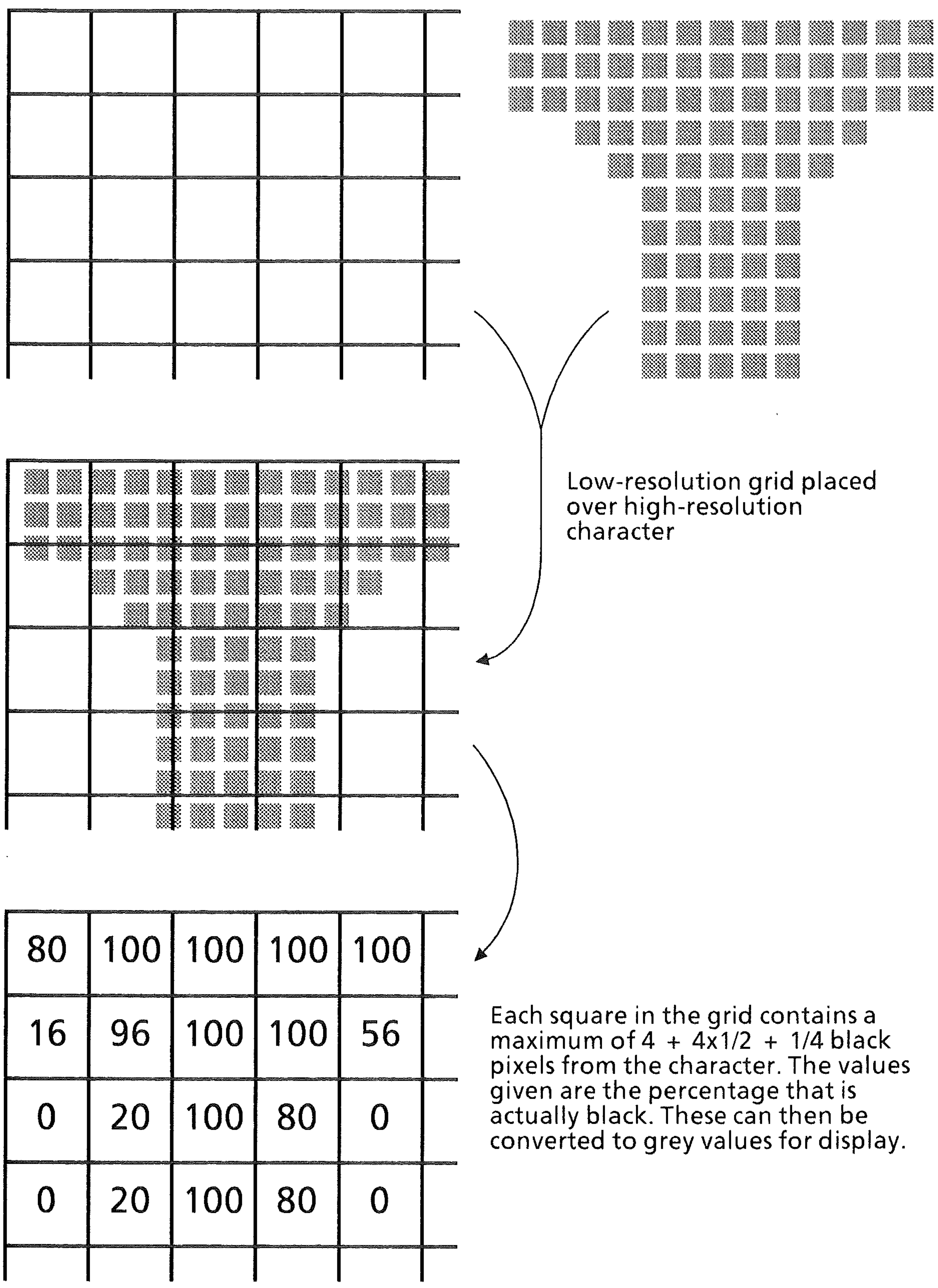
```
displayedSum := (pixelValue / maxPixel) * maxSum
```

giving rise to an error of:

```
remainder := sum - displayedSum
```

This error is referred to as the remainder and a process known as remaindering can be used to minimise its effects. This involves spreading the remainder in some pattern over adjacent screen pixels. In this implementation the remainder is simply added to the sum for the next pixel that is calculated. This is the next one along the row in horizontal sequence. No attempt is made to spread the remainder over other adjacent pixels because it complicates the algorithm considerably and the results obtained are acceptable anyway.

The procedure that implements this algorithm takes the scale factor to be used to map the given pixel map onto the screen and the screen offset at which to display the pixel map. Additional x and y offsets in character pixel units specify the offset in the pixel map to be placed at the screen position given. This allows sub-pixel control of the positioning of the pixel maps, which is very important for the correct display of the character spacing.



Low-resolution grid placed over high-resolution character

80	100	100	100	100
16	96	100	100	56
0	20	100	80	0
0	20	100	80	0

Each square in the grid contains a maximum of 4 + 4x1/2 + 1/4 black pixels from the character. The values given are the percentage that is actually black. These can then be converted to grey values for display.

Figure 8.7
Basis of the super-sampling algorithm

9. Evaluation and Conclusions

We started this thesis with the problem of designers confronted with new technology in printing. How can we make computers accessible to people who work in a way that is essentially alien to computer science? This discussion pointed out the need for computer systems that allow designers to use and extend their traditional skills of hand and eye. Before such a system could be built, an environment for constructing and running highly interactive programs was needed. Once this had been built, Imp, a system for type face designers could be constructed. This chapter seeks to evaluate how well both the interactive environment and Imp itself fulfil the requirements that were discussed at the beginning of this thesis.

The design of any interactive system needs to be an iterative process. Initial ideas are tried out in prototype systems to see how they behave in practice and the experience gained can be used in the design of subsequent systems. Before Imp was built a number of separate programs were written to experiment with the editing of letters in the form of outlines or rasters. Experience with these programs influenced the design of the programs that appear in the master and raster editing windows in Imp. Even so, further use of these programs in the context of Imp has revealed new problems and limitations. Often when problems are encountered the overall structure of Imp is such that changes have been easy to make and iterative design has continued without the need for a complete rewrite. What has been implemented is best considered as a framework that can now be extended. Many of the current limitations arise because of features that are missing rather than fundamental problems with the framework.

The window manager itself is really a first prototype and many problems with such systems were not appreciated until it was in use. It has been possible to make a number of changes with minimal rewriting of the window manager and with no need to change the application programs. This ease of making changes and extensions suggests that the approach adopted was a suitable one for the task being tackled. The simplicity of the window manager made it very usable and reliable, as well as making it easy to implement. It has lived up to its specification but in the light of experience it is now possible to envisage much richer environments for this type of work.

The rest of this chapter deals in more detail with experience so far and presents ideas for extensions to the existing system and for new approaches in future systems.

9.1 Imp in Action

Imp has been used fairly extensively by Lida Lopez Cardozo, a lettering artist with no computer science background. She used it to design the letters shown at the end of the previous chapter. Other designers have used the system for brief periods but not for long enough to give anything other than first impressions. A very interesting test would be for a designer who has worked with some other system to use Imp for an extended period. This would then give some more concrete information about how Imp compares with existing systems. What can be said is that Imp is the only currently available system that can support the design of new type faces from the very earliest stages to the final result within one environment. In what follows, some general observations arising from using Imp are presented and then more details of various specific aspects of the system are discussed.

Overall Impressions

The first impression of designers when confronted by Imp is that it is a surprisingly accessible and friendly system. There has been tendency to look for the hidden snag when it is actually as simple as it looks! The attempt at consistency across the system would seem to have worked, in that knowledge gained from working in one window enabled the designer to learn to use other windows much more rapidly. This expectation of consistency also enabled the users to point out areas where I had failed to be consistent. It was generally possible to put right such problems as soon as they were encountered.

One initial design decision was that the system should support all stages of designing from initial rough sketches through to the finished product. The designs that Lida produced were done entirely within the system, using calligraphic forms as the starting point. Once a few letters had been produced, new ones were more frequently produced by editing an existing letter rather than going back to sketching from scratch. A major problem that became apparent early on was the difficulty of drawing accurately with the tools provided. Although in some ways the stylus on the graphics tablet is similar to a pencil it requires a rather different action to use it. It must be held very upright and pressed down in a somewhat unnatural manner to activate the tip switch. As was expected, the mouse was even more awkward to use as a freehand drawing tool. This problem of being unable to draw exactly what was wanted induced a strong desire to go back to paper and pencil and then digitise the result! It seems probable that this is not an inherent problem with Imp but is a result of the particular hardware available. It is hoped that better quality hardware would overcome this problem.

A problem that designers anticipate when they come to a system like Imp is that they will be unable to work with their hand in one place and the result appearing somewhere else. In fact they rapidly find that they never really watched their hand anyway but used the result that was appearing on the paper to guide their movements. As long as the action connects directly with the result visible on screen most people have the instincts necessary to obtain the desired result. In fact it can be a positive advantage not to have your hand obscuring part of the image.

Handling master designs

The master designs, in the form of outlines defined by line chains, are the central items manipulated by Imp. All the major design decisions are made during the creation of these outlines. An important decision was not to use a spline representation for curves. A command to use a spline to generate a line chain seemed to provide all that was needed. The fact that the resulting line chain could then be edited by hand without mysterious and unexpected effects was a great advantage. The danger of polygonal letters does not seem to be sufficient reason to abandon the easily edited line chain representation. If the design is intended to be printed at a very large size it is left to the designer to insert a sufficient number of points so that the shape does not appear polygonal. The only factor to change this decision could be the availability of splines that can be easily manipulated interactively to give the sorts of shapes that designers need.

It was originally decided that the rough sketches should just provide some guidance to the designer in producing the master outlines, rather than being finely crafted in themselves. In practice a lot of work was often expended in drawing a good quality letter, only then to have to spend some time picking out the outline by hand. This was not only a tedious process but also less accurate than would have been the case if a drawing on paper had been digitised with appropriate equipment. What would have been useful is some automatic way of extracting the line chain defining the outline of a sketched shape. Image processing techniques exist for carrying out such a procedure and could easily be incorporated into the existing structure of Imp. In keeping with the overall philosophy this outline could then be worked on by hand to get the desired result.

It was hoped that the facility for moving a series of points onto a freehand line would make the editing of outlines as easy and direct as sketching. Unfortunately the difficulty found in drawing an accurate line means that this facility has not been much used. There has been demand for some means of working on the master outlines by sketching but it may well be that this facility, with better hardware, would fulfil this need. On the other hand it raises the possibility that it would have been more appropriate from the start to have made the sketch the central object and simply derive the outline automatically for storage or manipulation. The user would appear to be painting or erasing areas but internally it would be done by updating the outline. The problem with doing this, though, is that the structure of the object being manipulated is hidden and this can limit the user in what she can do with it.

Having decided to make the structure of the outlines explicit many of the commands were addressed directly to this structure, for example to add points, or insert or remove links in the line chains. Such a structure is initially unfamiliar to a designer but after some explanation and experimentation the structure could be easily manipulated. The immediate visual feedback and the ability to apply and cancel commands helped considerably in this learning process. The division of the strands making up an outline into separate parts was originally introduced to cope with the limitations of the scan conversion algorithm. As Imp was used to design a series of letters a new use was found for this facility. The editor window displayed all the non-current parts in a paler colour than the current part. Existing outlines could be brought into the window and displayed as separate parts to provide guidance in the design of a new letter. Their paler colour meant that the shapes could be seen well enough for guidance but did not intrude on the new design. Pieces of line chain could be extracted for inclusion in the new design and any unwanted pieces could be easily deleted. This is a good example of a user playing around and discovering that what was thought to be an obscure facility actually had a completely different and very valuable role to play.

The two simple transforms of translation and scaling proved to be very useful, especially in converting one letter into another. In fact scaling was not much used for size changes but for reflecting pieces of line chain, a side effect of the way scaling was implemented. The way reflection was obtained was rather obscure to non-computer scientists and would be better provided as a separate function. It was expected that rotation might be required but in practice all the effects needed seemed to be obtainable by reflections. On the other hand provision of a wide range of transformations may suggest new ways of working that would not be thought of in the absence of such facilities. Imp provides a sufficiently flexible framework for experimenting with such new extensions.

The "Compare masters" window was provided for viewing the master designs together, in particular for setting up the spacing. The possibility of setting up spacing by hand was available but in fact all spacing so far has been automatically calculated. Hence, this window has been used much more for confirming the appearance of the letters than for any extensive interactions. The original intention was to use this window as a background reminder of the context of a letter being worked on in the editor. Unfortunately the limitations of the Rainbow software, mentioned above, have prevented the window being used in this way. The window not currently in use could be present as a tag ready to be opened up from time to time but the immediacy of constant visibility was not available.

Handling rasters

There is less to be said about the handling of raster founts as the only work needed is to tidy up problems caused by scan conversion rather than any major designing. In particular, for high resolution printers there should be little work needed at this stage. Inevitably the task of cleaning up bit maps is somewhat tedious, but rather than better bit map editors what is needed is a better method of deriving the bit maps from outlines. It seems that some such procedures have been developed but they remain commercial secrets.

Currently the raster founts are handled quite separately from the masters. A raster character can be tidied up without any reference to the original master and it is easy to loose track of the original shape. An improvement would be to allow the outline to be displayed in the raster editor window, overlaying the actual raster. It would then be possible to edit the character to fit closely to the original. A more drastic change would be to integrate the outline and raster editors much more closely rather than treating them as separate facilities.

Text display

The display of text on the screen from within Imp provides a valuable check on the progress of a design. Although it is only a low-resolution approximation to the text as it would appear on paper, the use of anti-aliasing produces a remarkably good result. It is certainly adequate to give a feel for such things as the rhythm and density of a page, although further investigation is required to show how accurate this is in general.

At the moment the process of getting an image onto paper has to be done from outside Imp. The raster fount must be converted to the format required by the printer and the size and spacing information must be presented to the pagination program. It would be much better if the text displayed on screen could be transferred directly to a printer from within Imp. There is no reason why this should not be done although the time taken to transfer bit maps to a printer would cause a considerable delay when nothing else could be done. Even so, this could be preferable to stepping outside the environment of Imp.

The database

The structure of the database has influenced the structure of Imp and not always helpfully. The database is really just a set of files, each defining a complete fount. This limits the user of Imp to retrieving or storing whole founts, even when she is only working on a single character. It is often useful to have a whole fount present in memory but it gets very tedious if the whole fount must be written out to disc just to save a single character. In fact, it would be good to get rid of the distinction between founts on disc and founts in memory as this is a big source of confusion. It should be possible to achieve this if Imp were built on top of a proper database that provided rapid access and update to single characters in a fount. Any character that was copied into one of the database windows could be automatically saved on disc without the user taking any action. Currently it is left to the user to decide when to copy data to disc as it causes such a long delay. A properly structured database could also provide for the storage and retrieval of previous versions of a single character. This can only be done at the moment by giving the fount a new name so that it does not overwrite the previous version. Duplicating a whole fount in order to save a previous version of a single character is an unavoidable feature of the current database.

A further problem with the current version of Imp is that founts stored on disc but not in memory are invisible, thus violating the principle of total visibility. This could be put right by providing some sort of fount directory on screen from which founts on disc could be retrieved. A database of the sort described above could have everything equally visible and would just retrieve data from disc as the user accessed it.

The separation between master files and raster files has helped to maintain the distinction between working on master designs and the derived rasters. If the database explicitly kept the relationship between the derived characters and their original masters then experiments might have been done on integrating these two aspects of Imp. As it is now, it would be possible but not easy to locate the masters associated with particular rasters.

On the positive side, the use of human-readable text files for storing the fount data means that it is possible to edit the files by hand. Occasionally changes to be made to a fount are more easily done by editing the files directly rather than working by way of Imp. It is also easy to transfer the data files to other machines without corruption or to set up programs to read and manipulate the data. On the other hand, there is no reason why such text files should not be produced from a more complex database.

9.2 The User Interface

Imp has been constructed from a number of programs, each supporting different aspects of the design process. Each program is associated with a window on the screen through which the user communicates with it. As well as providing a useful separation of function it was intended that several different windows should be visible at once. This would enable the user to keep track of the different tasks being pursued. Unfortunately problems with the Rainbow software meant that it was unable to support the complex window structure required to have several windows visible. As a result it was not possible to investigate the utility of, for example, having groups of letters visible in the compare masters window whilst working on an outline in the master editor. The only reliable way of working was to have only the window currently in use fully visible and to have others closed down to just their tags. Even so, moving between different contexts was fairly rapid and easy and the constant environment for all the programs was very valuable.

The user communicates with the program in a particular window by simply moving the cursor into the window and pressing a switch. This method of selecting programs is easy to understand and avoids having special administrative actions that get in the way of moving between programs. It rapidly becomes a natural and instinctive action which the user does not have to think about. Because it is expected that the user will frequently move between programs this is a very important feature. The only problem encountered was that of a switch being depressed in one window and released in another. In order to prevent the release of the switch becoming separated from the intial depression, the switch release event does not cause a new window to be selected. After dealing with this problem, no further problems have arisen with this aspect of the window manager.

Presenting programs to the users as objects to interact with rather than commands to be issued probably played an important role in making the system accessible. Obviously this cannot be said conclusively as a version of Imp with a command rather than object oriented interface has not been tested. On the other hand it can be said that there appeared to be no problems with the concept of creating and using tools to carry out particular tasks. The fact that Lida fairly rapidly learnt to operate the system alone, including access to files stored outside the system suggests that the user interface has succeeded in making the facilities of the computer accessible.

Reserving the third button on a three button mouse for window management has also worked well. It makes for rapid and instinctive window manipulation without special modes or commands being apparent. The user simply "grabs" the window and moves it or pulls its edges around. An alternative interface that was not tried was to provide narrow, sensitive borders for the windows. Selecting these with the first mouse button or tablet stylus would allow the edge to be dragged in or out. A number of other systems in fact use this approach. The former approach emphasises that window management is separate from other functions of the system whereas the latter emphasises that all functions are of equivalent status. Although the former approach was perfectly usable, in the absence of extensive comparative testing it is not possible to say whether it is better than other approaches.

The fact that the window manager has been implemented on the Rainbow workstation makes the use of wide ranges of colours in windows a practical possibility. Imp, the main application implemented using this window manager, makes extensive use of colour even though type face design may be considered essentially a black and white application. Different colours distinguish different windows and make it much easier to see window boundaries. This gives the appearance of a much less cluttered screen than seems to be the case if all the windows are the same colour. Using muted colours for backgrounds is very restful on the eye and careful choice of foreground colour results in a picture that appears solid and steady. If bright colours are used then the flicker caused by refresh and interlace on the display becomes much more intrusive. The colours that have been used have been selected mostly by trial and error. Imp has served to show the utility of colour rather than to demonstrate the correct use of colour in any measureable way. Murch [1984] provides some guidelines, derived from psychophysical experiments, that could be made use of in future systems.

Input tools

A valuable feature of the window manager is the provision of multiple input tools in addition to the keyboard. This feature would seem to be absent in all other current window managers. It allows the user to choose the tool most suited to the current task, even changing over in mid-session. Other systems allow for a single device, generally providing a choice between a mouse or graphics tablet. Swapping devices is achieved by unplugging one and plugging in the other, hence discouraging frequent swapping. The window manager described in this thesis can support any number of devices with all of them contributing to a single stream of switch and coordinate events. The Rainbow workstation provides the necessary hardware support for arbitrary numbers of devices. The freedom to use any tool at any time means that the user need not be constrained by decisions made by other people. It should be pointed out that graphics libraries such as GKS can support any number of physical devices, but not integrated into a windowing environment.

A number of different ways of working with the tools arose, with different people favouring different combinations. In general the tablet stylus was used for drawing and the mouse for pointing and selecting objects. Even so, there were people who prefered the mouse for drawing or who used only the stylus and keyboard for everything. An interesting and unanticipated combination that arose was to use the stylus in one hand for moving the cursor and use the other hand for operating switches on the mouse. In this situation the mouse was being used simply as a three-button keyboard. It was interesting to observe the development of this method of working, which has certain parallels with the now little used mouse and handset mode of working. One hand is used for pointing and selecting objects on screen and the other is used for issuing commands. These experiences with the use of the different tools certainly served to confirm the belief that the user should be free to use any tools at any time.

An alternative use of multiple tools that has not been experimented with is to distinguish between switches and coordinates from the different devices and use the choice of physical device to determine the action of the program. For example, using the tablet stylus could automatically select sketching and using the mouse could select editing. Then again, different devices could select different coroutines, rather than using cursor position. In keeping with the general philosophy of giving the user freedom of choice these relationships between tools and tasks must be easily changeable.

The help system

The window manager has a help mode where any user action is supposed to evoke a message explaining what result this action would have. It is hard to assess the success of this approach as it relies on each application coroutine testing for help mode and reacting appropriately. A fair amount of effort and discipline on the part of the application programmer is needed to provide this information. Because the programs using the window manager have been experimental and changing this information has either been out of date or not provided at all. Provision of help would be encouraged by providing a way of entering the help text along with the code that carries out the action. An automatic system for generating the application code could easily handle this help text and set up appropriate responses in help mode. Hence even experimental programs could provide such information without an undue expenditure of effort.

Currently the information is displayed on the VDU connected to the workstation rather than on the graphics screen. A better approach would be to use a special text window somewhere on the graphics screen so that the user does not have to switch her attention elsewhere. The user would be free to close this window down to just its tag if it was not of interest, but it would always be available.

Help information provided within the system is easily accessible to the user but there are times when it is more convenient to read a printed document. Online help is perhaps best regarded as a reminder for users who are already fairly familiar with the system. More extensive information and introductory material is probably better provided in printed documents. For a system such as Imp, which is intended for people who may never have used a computer before, a manual in familiar printed form may help in introducing the new and unfamiliar environment.

Cancellation and backtracking

One important feature of any sort of design work is the need to experiment with different ideas. If the result is not suitable then the ability to move back to a previous design is extremely useful. Another valuable result of being able to backtrack is that it makes it much easier to learn how to use a computer system. A big stumbling block is often the fear of breaking something or losing the work that has already been done. When the user is confronted with some facility that has no parallel in her previous working environment the ability to experiment without harm will speed up her learning about that facility. For example mathematical transformations to shapes are hard to understand until they are seen in action.

The only provision the window manager itself makes is to wait for cancellation after the user has destroyed a window. If this happens then the window is restored unharmed. Any further provision must be made in the application coroutines, where it can be adapted to the nature of the particular application. In the master editor coroutine an experiment was carried out in which the total state was saved every time a menu selection was made. Only one previous state was stored, allowing the user to step back to the point where the current menu item was selected. Hitting cancel again restored the new state so the user could keep swapping between the current and one previous state. This provided a valuable facility for assessing the effect of the command just carried out as well as guarding against mistakes. Providing the ability to back track by just one step makes a big improvement to the user interface without using up large amounts of memory to store large numbers of previous states. This covers the most common mistake of either selecting the wrong command or applying it to the wrong object. Incorporating this facility into the window manager would require some way of defining the data that constitutes the state of a coroutine and defining the significant events. For example, in the master editor window only switch events that cause a menu selection are considered significant with regard to saving the state but under different circumstances other events might be significant. Beyond this, in Imp, the only back tracking available is to states which have been explicitly saved by the user.

This combination of the automatic saving of a single previous state with the explicit saving by the user of other significant states has worked very well. In fact this may be an ideal compromise in that large amounts of memory are not used up to store states that may never be of interest. The system merely provides a very valuable safeguard against momentary lapses or mistakes. A great improvement to the current system would be the provision of explicit support for multiple versions of objects in the database. At the moment the user must explicitly rename an object if a subsequent version is not to overwrite it. A simple first step would be for the system to add an automatically incremented version number to the file name. A visual interface to all the objects in the database would also be helpful.

9.3 The Window Manager

The window manager was implemented to provide an environment for constructing and running multi-threaded programs such as Imp. The application program is viewed as a series of event handlers, which feels very natural and results in a modular program that is relatively easy to modify and debug. The window manager deals with communication between the user and these event handlers and also takes care of cursor update. The user communicates with the window manager rather than the application program to rearrange windows on the screen. This means that the application program can be very simple as all the complexities of devices handling and screen organisation are dealt with elsewhere. The ease with which Imp was written and extended indicates that the window manager was successful in providing a suitable environment for a highly interactive program. In what follows a number of specific points relating to Imp and the window manager are discussed.

Coroutines vs multitasking

An important decision was to use coroutines rather than asynchronous tasks for the event handlers. This reflects the highly interactive nature of the application being supported. The resulting system is very responsive and there is no danger of background tasks affecting the response time. A few tasks, such as writing a fount out to a file or calculating spacing for a character, take a significant amount of time but even here the time taken is not enough to make the user want to do something else.

The use of coroutines rather than asynchronous tasks allowed a considerable simplification of the code for the application programs. For example, Imp makes extensive use of data structures that are shared between windows. If asynchronous tasks were used then special actions would have to be taken to synchronise access to the shared data. With coroutines the flow of control is explicit and so one coroutine can complete any manipulations before allowing any other coroutine to run. Some modern languages have facilities built in to support the definition and use of shared data, in which case multitasking would be no problem. In BCPL running under Tripos it is not so easy to handle shared data correctly and so there is a strong incentive to avoid multitasking if possible. In BCPL another advantage of coroutines over multiple tasks is that coroutines share the same global vector. This makes communication and sharing between coroutines very easy as they can access the same global variables.

Application program structure

As different programs have been written in different windows the need for certain common features in their structure has become apparent. For example, on many occasions a special cursor function is required whilst a particular switch is held down. This produces effects such as holding a switch down to drag an object or to sketch something. Given the identifier of the switch and the cursor function to be used it would be a simple matter to automate the production of code for such an interaction. Currently this is done by simply copying a standard piece of code from one coroutine to another.

At a more global level the code for the coroutine in each window consists of initialisation, a loop for receiving and processing switches and then termination code which is entered when certain conditions arise in the central loop. The central loop consists of a series of separate routines for processing the different switch actions. This simple structure suggests that each coroutine could be specified in terms of its initialisation and termination code and a list of the actions to be evoked on particular switch events. These switch handlers run in the environment defined by the initialisation code. Given this specification, two possibilities are available. The code for each coroutine could be generated automatically and then run in the same way as hand-written coroutines. Alternatively the information describing each coroutine could be stored in tables for a new table-driven run-time system. Every time an event occurred the table for the current coroutine would be consulted and the appropriate response initiated. In fact life is not always so simple and some coroutines have more than one set of interpretations of the switch events. Each of these modes is specified by a complete new set of switch interpretations and the system must be capable of switching between them on particular events. With the present software the writing of new coroutines has been greatly simplified by this standard structure that has emerged even though it was not explicitly enforced.

Rather than using actual switch numbers in the code for the coroutines, manifest constants were used. The numbers bound to these names were specified in a header file. By changing these numbers and remaking the program the mapping of switches to actions could be changed. This provides a good deal of flexibility for experimenting with the user interface. There is a certain amount of consistency between coroutines in the use of switch actions. For example pointing and selecting, with continuous feedback while the switch is held down, are frequently used actions. An interesting extension of the current set up would be to write all the coroutines in terms of abstract events. For example, instead of waiting for switch number one to be pressed the coroutine would wait for the event "start selection". Tables provided for the run-time system would specify which physical events trigger which abstract ones. Different users could provide their own tables in order to tailor the system to their requirements. This has something in common with the flexibility provided by the use of virtual devices in graphics libraries.

Menus and forms

The provision of packages of routines for menus and forms has been valuable in speeding up the writing of application programs. It has been very easy to experiment with different formats and styles, particularly with menus. Currently the menus and forms are specified by vectors of information. The production of menus and forms could be greatly simplified by providing a program that converted a textual description of the menu or form into these vectors or the code for generating them. This would make it much easier to set up a new menu or form as well as eliminating a significant source of errors. On many occasions apparently obscure bugs have been found to be caused by such mistakes as changing the number of menu items without changing the length of the vector.

An interesting extension to forms would be the provision of support for different types of data. All of the information input into a form at the moment is treated as strings of characters. If it is ultimately to be interpreted as numeric data the client program must carry out the conversion and checking itself. It would be useful to be able to specify that a particular item was numeric rather than textual and have the checking and conversion done automatically. Taking this even further, different visual representations for the data could be used. For example, a numeric item could be specified by setting a dial or slider. A textual item with a limited number of options could display all the options for the user to select from by pointing. To make these facilities easy to use, various standard forms of feedback should be available but the client should always be able to specify some other.

9.4 Future directions

Imp has used straightforward line chains to represent letter shapes. An interesting area for future work is that of investigating the use of more structured representations. At the simplest level this might be fixing the angle or length of a single line or setting up relationships between letters so that the properties of particular lines in all of them were the same. A higher level of structure could divide letters up into parts such as serifs, stems and bowls. Letters could share these parts and so change their characteristics together when a part was changed. All sorts of problems start arising, such as how these structures are made visible to the user and how she interacts with them. These structures should always be built on top of the basic line chain structure, giving the user the freedom not to use the abstractions and simply adjust the details of individual letters by hand.

An area where some sort of "meta-ness" might be argued for is where a whole series of related faces is to be produced. The automatic production of bold, italic and light faces from one text design would ease what can be a rather tedious task. The problem with making a face bolder is that it is neither a simple stretching in one dimension, or just a thickening of the strokes. The strokes must be thickened and the letter widened. A meta-font could define how the different parts are to change for bolder or lighter faces but in fact it should be possible to go some way towards what is wanted without any explicit structuring. Image processing techniques can be used to extract a skeleton of the letter and find the vertical strokes. It should then be possible to thicken or thin these strokes automatically. The results would not necessarily be particularly good but they could then provide the basis for the heavier or lighter design. Rather than struggling with the definitions of how the different faces relate, it may be much more profitable to explore techniques that involve little or no structure but help the designer on her way. This would also avoid the danger of trapping the designer with structures that prevent her working intuitively whenever she wants to.

The discussion so far has been at two different levels—that of the user interacting with a graphical application and that of the environment supporting the application.

Imp consists of two distinct levels—the underlying graphical environment and the application program built on top. This application program is set up as a series of objects that will respond to the user's actions. A lot of effort has been expended in making sure that the structures provided for the user are appropriate and easy to manipulate. The user of Imp has no opportunity to create new types of structures with new behaviours. It has already been suggested that an environment that made it easy to define objects and the interactions with them would be a great help in constructing programs such as Imp. If the interface to this environment were suitably graphical, it might make it possible for the user of a program such as Imp to extend and modify it. There would be no need to provide elaborate facilities in the initial system. Instead, it would be left to the user to extend the system as needs arose. A type face design system that is of interest in this connection is described in Lynn Ruggles' forthcoming thesis. Like Imp, her system is intended to be highly interactive and is based on a windowing system. A feature of particular interest is the ability to set up various structures and relationships describing the letters. The user will make use of a graphical language for creating structures and also for extending the system to provide new facilities. It will be interesting to see how a higher level of abstraction combines with a visual, interactive approach.

In Conclusion ...

This thesis has explored ways of using a computer to support the process of type face design. I have tried to pay particular attention to the importance of intuitive decisions and visual judgement, with the skills and requirements of the person rather than the machine being the central preoccupation throughout. The computer has been treated as a tool that is very much under the constant control of the person using it. The fact that the design of new type faces has not been completely automated is not an admission of failure but rather a recognition of the nature of designing. Something vital would be destroyed if there was no room for the unexpected, individual touch. The system implemented in response to these ideas has indeed begun to reconcile intuition and automation. Perhaps in this small area we have gone some way towards answering the fears expressed by the Dutch economist and parliamentarian, Bob Goudzwaard:

> "In industry humans are adjusted to the machine and its tempo; the machine is not usually adjusted to human creativity and the rhythms of human life. Modern men and women feel more that technology controls them than that they control technology. Are these coincidences, or signs that technology occupies an exaggerated, perhaps idolatrous place in modern society?"

[Idols of Our Time, pg 23]

References

Anson E.D.
[1982] "A device model of interaction"
Computer Graphics; vol 16 no 3

Apperley M.D. & Spence R.
[1983] "Hierarchical dialogue structures in interactive computer systems"
Software—Practice and Experience, vol 13 pg 777

Arnold D.
[1981] "The requirement for process structured graphics systems"
Computer Graphics, vol 15 no 2

Barsky B.A. & Beatty J.C.
[1983] "Local control of bias and tension in Beta-splines"
ACM Transactions on Graphics, vol 2 No 2

Beach R., Beatty J., Booth K., Plebon D. & Fiume E.
[1982] "The message is the medium—multiprocess structuring of an interactive paint program"
Computer Graphics, vol 16 no 3

Bigelow C.
[1982] "Aesthetics vs Technology Pt 2"
Seybold Report, vol 11 no 11

Brewer J.A. & Anderson D.C.
[1977] "Visual interaction with Overhauser curves and surfaces"
Computer Graphics, vol 11 no 2

Buxton W., Lamb M.R., Sherman D. & Smith K.
[1983] "Towards a comprehensive user interface management system"
Computer Graphics, vol 17 no 3

Carter K.A.
[1984] "Imp—a system for computer-aided type face design"
Proceedings of Protext1, Boole Press

Coueignoux P.J.M.
[1975] "Generation of Roman printed fonts"
PhD Thesis, Massachusetts Institute of Technology

deDoes B.
[1985] "Trinité"
to appear in Visible Language, Vol XIX

Douglas DH, Peucker TK
[1973] "Algorithms for the reduction of the number of points required to represent a digitised line or its caricature"
Canadian Cartographer, vol 10 no 2

Durer A.
[1535] "Of the just shaping of letters"
Translated by R.T. Nichol; Dover publications Inc., New York, 1965;

Engelbart D. & English W.
[1968] "A research centre for augmenting human intellect"
Proceedings of AFIPS Fall Joint Computer Conference

Flowers J.
[1984] "Digital type manufacture: an interactive approach"
Computer, vol 17 no 5

Foley J.D. & Van Dam A.
[1982] "Fundamentals of Interactive Computer Graphics"
Addison Wesley

GKS
[1984] "Computer Graphics Special Issue"

Glauert T.H. & Wiseman N.E.
[1985] "Real-time Image Combination"
Proceedings of MICAD 85, Hermes

Gosling J.A.
[1984] "A User Interface Toolkit"
Proceedings of Protext 1, Boole Press

Goudzwaard B.
[1981] "Idols of Our Time"
InterVarsity Press; English Translation 1984; pg 23

Greggains A.
[1985] "A Structured Computing Environment"
PhD thesis, University of Cambridge

GSPC
[1977] "Status Report of the Graphics Standards Planning Committee of ACM-SIGGRAPH"
Computer Graphics, vol 11 no 3

Hobby J.
[1985] "Smooth, easy to compute interpolating splines"
Stanford University Computer Science Department, STAN-CS-85-1047

Ingalls D.H.H.
[1981] "Design principles behind Smalltalk"
Byte, August 1981

Karow P. et al.
[1979] "Ikarus-system: computer-controlled font production for CRT and Lasercomp"
URW Unternehmensberatung Karow Rubow Weber GmbH, Hamburg, Germany

Kasik D.J.
[1982] "A user interface management system"
Computer Graphics, vol 16 no 3

Kay A. & Goldberg A.
[1977] "Personal Dynamic Media"
Computer, vol 10 no 3

Kay A.
[1969] "The reactive engine"
PhD thesis, University of Utah

Kilgour A.C.
[1981] "A hierarchical model of a graphics system"
Computer Graphics, Vol 15 no 1

Kindersley D.G. & Wiseman N.E.
[1978] "Letter Spacing"
British Patent application number 37544/78; Patent number 2004502 granted 1982

Kindersley D. & Wiseman N.
[1979] "Computer-aided letter design"
Printing World, October 1979

Knuth D.E.
[1979] "Tex and Metafont: new directions in typesetting"
American Mathematical Society and Digital Press

Kohen E.
[1985] "An interactive method for middle resolution font design on personal workstations"
ETH, Institut für Informatik, Zürich

Lantz K. & Nowicki W.I.
[1984] "Structured graphics for distributed systems"
ACM Transactions on Graphics, vol 3 no 1

Leitch S. & Smith F.J.
[1984] "Cubic splines in font design"
Proceedings of Protext 1, Boole Press

McGregor S.
[1983] "The Viewers Window Package"
In "The Cedar System: an Anthology of Documentation", Xerox PARC CSL-83-14

Mergler H.W. & Vargo P.M.
[1968] "One approach to computer assisted letter design"
Journal of Typographic Research (now Visible Language), vol 2 pg 299

Moody K. & Richards M.
[1980] "A Coroutine Mechanism for BCPL"
Software—Practice and Experience, vol 10 pg 765

Moran T.
[1981] "Guest editor's introduction: An applied psychology of the user"
Computing Surveys, vol 13 no 1

Murch G.M.
[1984] "Physiological Principles for the Effective Use of Color"
IEEE Computer Graphics and Applications, vol 4 no 11

Myers B.A.
[1984] "The User Interface for Sapphire"
IEEE Computer Graphics and Applications, vol 4 no 12

Needham R.M. & Herbert A.J.
[1982] "The Cambridge Distributed System"
Addison Wesley

Newman W.M.
[1968] "A system for interactive graphical programming"
Proceedings of AFIPS Spring Joint Computer Conference

Olsen D. & Dempsey E.
[1983] "SYNGRAPH: a graphical user interface generator"
Computer Graphics, vol 17 no 3

Purdy P. & McIntosh R.
[1980] "Forward Thinking"
British Printer

Richards M., Aylward A.R., Bond P., Evans R.D. & Knight B.J.
[1979] "TRIPOS - a portable operating system for mini-computers"
Software—Practice and Experience, vol 9 pg 513

Richards M.
[1969] "BCPL: a tool for compiler writing and system programming"
Proceedings of AFIPS Spring Joint Conference

Rosenthal D.S.H.
[1981] "Methodology in computer graphics reexamined"
Computer Graphics, vol 15 no 2

Ruggles L.
[1986] "Paragon - an Interactive, Extensible Typeface Design Environment"
Forthcoming PhD thesis, University of Massachusetts

Smith D.C., Harslem E., Irby C. & Kimball R.
[1982] "The Star user interface: an overview"
Proceedings of AFIPS National Computer Conference

Smith D.C., Irby C., Kimball R., Verplank B. & Harslem E.
[1982] "Designing the Star User Interface"
Byte, April 1982

Steinberg S.H.
[1955] "Five Hundred Years of Printing"

Penguin Books

Strubbe H.J.
[1983] "Kernel for a responsive and graphical user interface"
Software—Practice and Experience, vol 13 pg 1033

Styne B.A.
[1985] Title unknown
Forthcoming PhD thesis, University of Cambridge

Styne B.A., King T.R. & Wiseman N.E.
[1985] "Pad structures for the Rainbow Workstation"
Computer Journal, vol 28, no 1

Sutherland I.E.
[1963] "Sketchpad: a man-machine graphical communication system"
Proceedings of AFIPS Spring joint computer conference

Tesler L.
[1981] "The Smalltalk environment"
Byte, August 1981

Thacker C.P., McCreight E.M., Lampson B.W., Sproull R.F. & Boggs D.R.
[1981] "Alto—a personal computer"
In "Computer Structures—readings and examples", edited by Siewiorek, Bell and Newell; McGraw-Hill

Updike D.B.
[1937] "Printing Types—their history, forms and use"
Harvard University Press; pg 6, pg 13

Wallace V.L.
[1976] "The semantics of graphic input devices"
Sigraph/Sigplan symposium on graphic languages

Wilkes A.J., Singer D.W., Gibbons J.J., King T.R., Robinson P. & Wiseman N.E.
[1984] "The Rainbow Workstation"
Computer Journal, Vol 27 no 2

Wilkes A.J.
[1982] "Lexlib—a user-extensible lexical analyser"
Rainbow Group User Guide, April 1982

Wiseman N.E. & Robinson P.
[1977] "An operating system for interactive terminals"
Software—Practice and Experience, vol 7 pg 501

Wong P.C.S. & Reid E.R.
[1982] "Flair—a user interface dialog design tool"
Computer Graphics, vol 16 no 3

Zapf H.
[1970] "About alphabets—some marginal notes on type design"
MIT Press; pg 55

Appendix 1

Imp user manual

Imp User Manual

Lynn Ruggles — August 1985

1. Getting Started

To run the fount editor Imp, you will be using the Rainbow workstation. There are certain things that must be set up by a Tripos expert before you use Imp for the first time—Appendix B describes these. The workstation consists of a black and white monitor (or black and orange as it happens), a colour monitor, and two keyboards. The black and white keyboard will be used while you are typing commands to the black and white monitor and the coloured keyboard will be used to type information while you are using the colour monitor. The convention followed in the manual is that anything that you are supposed to type in will be displayed in **bold face** type.

To login to Rainbow, using the black and white monitor and keyboard, press the **Break** key (upper right hand corner of the keyboard), then type **crainbow**. You will then be asked for your user name and password.

user: **gutenberg**
password: **typo**

You will not actually see your password as you type it—this helps you to keep it secret. Eventually Rainbow will respond on the black and white screen and you must now tell it to run Imp.

`RAINBOW-1>` **imp**

A lot of information will appear on the screen finishing with a string of dots followed by the word *done* at which point a cursor should appear on the colour screen and you should switch to the coloured keyboard. If you wait a long time for the word *done* and nothing appears to be happening, the system has probably crashed; check the section in this manual on Troubleshooting for information on how to start things up again.
Note that at any time while you are using Imp, you can generally cancel the effects of the last command you gave by hitting the **cancel** key on the right side of the keyboard. This should return you to your previous state. The system only remembers the current state and the previous state so this only undoes the last command. It doesn't remember any further back than that. If you hit cancel twice, the first one cancels the command, and the second one cancels the cancel, putting you back into the state that you just cancelled.

Once Imp is running, switch to the coloured keyboard.

2. Cursor and Window Information

To work with Imp, you will be using either a mouse (a small round device with one wire coming out its side and three buttons on the front) or a stylus (looks like a pen but has a wire coming out the top).

2.1 Mouse

Movement of the mouse results in movement of the *cursor* on the screen. The cursor can be seen as a small black dot. If it changes to an hourglass and you can't move it, the system is processing some information and you will have to wait until it changes back into a dot before you can resume fount editing. If you can move the hourglass, the system is expecting you to do something before fount editing continues. This happens when you are moving information between windows—there'll be more about this later. There are three switch buttons on the mouse numbered 1-2-3, left to right. In this manual, *clicking* a button will mean pressing the button and releasing it, and *pressing* a button will mean pressing the button and holding it down.

Button 1 is used for pointing at something that is already on the screen. Move the cursor so that it is positioned over the item or location that is to be selected, then click or press button 1.

Button 2 is used to get pop-up menus. As long as the button is pressed, the menu appears on the screen. Movement of the cursor within the menu results in the item under the cursor being highlighted by a red box. If the button is released while an item is highlighted, that item is *selected.* Each window has a different pop-up menu as does the background screen.

Button 3 is used for window manipulation such as moving a window to a different position on the screen or changing its size. To move the window, hold the button down while the cursor is positioned in the centre of the window and move the cursor. The window will follow the cursor. You can also move the window by placing the cursor over the name tag at the top left corner of the window. To change the size of the window, hold the button down while the cursor is on the edge of the window and move the cursor so that the window grows larger or smaller. Note that each window has a maximum size beyond which it cannot grow any more.

2.2 Stylus

Instead of using the mouse, you might find it easier to use the stylus. Move the mouse out of the way and, if it is there, move the red pad off the top of the blue stylus tablet. The cursor should respond to movements of the stylus in the same way that it responded to movements of the mouse.

The stylus can be used to select objects in the same way that Button 1 was used on the mouse. When the cursor is positioned over an item or location that is to be selected, hold the stylus perpendicular to the tablet and press down on the tip.

Instead of buttons on the stylus as there were on the mouse, there are three buttons on the upper right corner of the coloured keyboard. To get pop-up menus, you must use the middle key, and to do window manipulation you must use the rightmost key. The leftmost key can be used as well as the stylus to select items on the screen.

You may instead chose to use the stylus as a drawing tool, and use the mouse buttons for selection. If you decide to do this, position the mouse so that it is on the table next to the tablet, and try not to move it when you press a button. If you try to move the cursor with the stylus and it seems to jump around, it is probably because you are also moving the mouse so that Imp is receiving input from both the mouse and the stylus and is trying to follow both of them. Take your hand off the mouse and hit the **Home** button on the right edge of the keyboard. The cursor should return to the center of the screen.

2.3 Windows

When you are running one of the editors, there will be several sub-windows within each large window on the screen. These sub-windows have several functions.

The Master Fount and the Raster Fount windows have two sub-windows. The top window displays the characters that are in the fount. Different characters can be selected by moving the red box to the appropriate character. Press Button 1 and move the cursor until the box is positioned over the character you wish to select. An empty space indicates that there is no information stored for that slot. The lower sub-window displays the labels for the various attributes of the fount. These are entered via the coloured keyboard.

The Edit Masters and the Edit Rasters windows have a large sub-window which will be referred to as the *working area* and the small window in the upper corner will be referred to as the *reference area*. This window contains a smaller version of what is displayed in the working area. Other windows will contain menu selections and will be referred to as *selection pads*.

There is a red box displayed in the reference area. Changing the dimensions of the box allows you to select which section of the character you want displayed in the working area (note that if the box extends to the edges of the reference area, it may be hard to see. If you aren't sure about it, try changing its size just to see that it is really there). If you move the cursor to the reference area, pressing Button 1 when it is at the edge of the red box and moving the cursor will result in changes in the size of the box. The box is always square. If you want to see the entire character, you should change the size of the box so that its edges extend to the edges of the small window. Pressing Button 1 while it is in the centre of the box will allow you to move the box until it is positioned over the section of the character that you want to look at. Note that in the Raster Editor, you can't change the size of the box but you can move it around.

Selection of some menu options will result in a small purple form popping up on the screen. The cursor will change to a miniature version of the form. At this point, the program waits for you to enter some data in the window. Type in the information, using the orange up and down arrow keys to move from one field in the form to another. After you have entered the data, press the **Return** key on the coloured keyboard and Imp will continue. If you wish to cancel the command and get rid of the menu, press the **Cancel** key.

2.4 Window Icons

Once you have a window on the screen you will notice that at the top of it is a small tag containing the name of the window and three icons. The leftmost icon is used to Open the window so that it is redisplayed on the screen (the icon looks like four arrows pointing out from the centre of a box), the middle icon is used to Close Down the window, so that only the tag is left on the screen (the icon looks like four arrows pointing into the centre of a box), and the rightmost icon is used to hide the window if it is obscuring another window. It is a good idea to Close Down one window before Opening another window as Rainbow seems to crash less often that way.

2.5 Moving data between windows

You will often need to move data between windows—for example to edit a character you must move it from a fount window to an editor window. Data is moved by sending it along *pipes*. You must send it from one end and get it from the other to complete the transfer (you can do these actions in either order). When you have sent data from one window but not yet got it in another, the cursor will appear as an hour glass in the first window. You can't do anything else in this window until the transfer is either completed elsewhere or cancelled in this window.

3. Files

You will be storing several different kinds of files while using Imp. A ***master fount file*** contains outlines of character shapes. A ***raster fount file*** contains raster characters, or characters shapes that are made up of pixels. Pixels are discrete, usually square, picture elements. The characters stored in this file are similar to ones that might have been drawn on a piece of graph paper. A painting is stored in a *painting* file, and spacing information for characters are stored in *spacing* files. You will be using yellow menu commands to save information in the master fount file, the raster fount file and the painting file. Spacing information is automatically saved when you save a master fount file.

Information that has been stored in a file can be retrieved at some future date, provided you remember the labels that you typed in when you stored the file. The file names are generated automatically by Imp and are derived from the labels you fill in when you save the file. It is a good idea to label founts with a name that represents the typeface so that it will be easier to recall it later. Note, however, that the name of the file does not convey any information to the program itself. If you choose to name a file Roman the program will not complain if the fount contains Italic characters. The name is simply a convenient way for you to remember and save what you have done; you could choose to name them after your friends or favourite colours.

The name of the characters within each file is important. The Display Text window looks for familiar character names in order to map them to real characters. The convention followed in the program is that characters are named by either 'Uppercase-' or 'lowercase-' followed by the character, e.g. Uppercase-C, lowercase-m. Numbers are mapped to the name 'digit-' followed by the digit, e.g. digit-4, and punctuation is mapped to a descriptive name of the symbol, e.g. SemiColon, QuestionMark. Characters which do not map to a printable Ascii code are labeled by 'c' followed by the ascii number, e.g. the character which maps to ascii code 12 would be labeled 'c12'. A complete list of mappings can be found at the end of this document.

4. Running Imp

First the workings of the system will be described, then some sample scenarios will be presented. If you aren't sure what you are doing, check the back section of this manual to see if there is an example that you can follow. Also, don't forget about the **cancel** key on the right side of the keyboard which will cancel the last command.

After Imp is started, the screen will be dark grey with a cursor displayed in it. Movement of the mouse or stylus will result in cursor movement on the screen. Movement of the cursor corresponds to the movement of the mouse or stylus so if you consider the top of the tablet to be the edge closest to the screen, and the bottom to be the edge closest to the edge of the table, and you pull the mouse or stylus towards you thus moving it toward the bottom of the tablet, the cursor moves toward the bottom of the screen. If you lose the cursor, press the orange button labeled **home** on the right edge of the keyboard. That should bring the cursor back to the centre of the screen. If it doesn't, the system has probably jammed and you will have to start over. Even if things seem to be working alright for the moment it may be a good idea to look at the section on Troubleshooting at the end of the manual to get some ideas about how to keep it that way!
To start working with the system, press the centre button to get the background menu displayed. Once you have some founts to edit, you may want to start with any of the menu selections, but to start from scratch, you should select **Edit Master.**

Each of the background menu selections will be discussed in a separate section, so to find out how to use any of them, consult the relevant section of this manual. The selections are:

Master Fount: shows what characters exist, allows you to select the name of the master fount.

Edit Masters: used for sketching or editing outline drawings of characters.

Compare Masters: used to look at characters together and to adjust the spacing.

Raster Fount: shows what raster characters exist, allows you to create a raster fount or select the name of an existing one.

Edit Rasters: used to edit raster versions of the characters.

Display Text: used to look at text patterns using the characters in a particular fount.

Finish: Terminate this session of Imp. The system won't let you do this if you still have any windows around—make sure you save everything you want to keep before finishing your windows.

5. Master Fount

The master fount window has labels where you are to fill in the name of your master fount. If you are starting from scratch, fill in names that seem appropriate. Use the coloured keyboard to enter text, and use the orange arrow keys on the keypad on the right side of the keyboard to move up and down in the window. The size can be 0 if you do not intend the design to be for a specific size. Note that the last item, Letter, will need to be typed in for each character. If you begin to edit a non-existent character, this field will contain the name *Unknown*. The rest of the fields will stay the same.

Master Fount items:

Family: **cambridge**
Style: **roman**
Size (decipoints): **100**
Space Table: **standard**
Letter: **lowercase-m**

5.1 Master Fount Menu

Read in from File: reads in the fount that you have specified. If it exists, each of the characters in the fount will be displayed in the top half of the window. If it doesn't exist, a message to that effect will be displayed on the black and white monitor. Check your fount name and try again.

Copy out to File: once you have created or edited some characters, use this command to save them in an external file. Unless you save them in a file before you finish Imp, they will disappear forever. If you have already saved fount information using this fount name, a form will pop up asking if you want to overwrite the old information. Answer Y(es) or N(o). When you save a file, all the current information for the fount will be saved. If you don't wish to overwrite your old file, you will have to change the name of the new fount and then select Copy out to File again.

Send Data: windows in Imp communicate by Sending and Getting data and characters. Several different kinds of information can be sent including fount data, grid information, or individual character data to the various windows. If you are sending general fount or grid data, it does not matter what character is selected by the red box. If you wish to send character data though, you will have to select the character by moving the red box before you select Send Data.

Get Data: after you have edited a character, you must Send it from the editing window, then you must Get Data in the Master Fount window. This will update the information for that character in the Master Fount. If you have created a new character, make sure the red box is positioned over an empty space before selecting Get Data on the menu. After getting the character, type in its name in the Letter field at the bottom of the window. Use descriptive names if possible. (For naming conventions used by this program, see the section on Files.)

Clear: clears the information in the Master Fount window.

Finish: terminates the Master Fount window.

6. Edit Masters

This window allows you to paint or sketch new characters, create or edit outline characters, and set up grids. There are three pads associated with this window: the paint pad, the grid pad and the outline pad. Each pad is transparent so that items drawn in one pad can be seen when working in the other pads. You can clear each pad separately without affecting what is displayed in the other two pads. If, for example, you first set up some grids using the grid pad, then sketch a character in the paint pad, then draw an outline around the character in the outline pad, you can erase the grid, the painting or the outline without erasing the others.

There are two selection areas in this window. The lower area allows you to select one of the pads, and the upper area then displays commands that you can select for that pad.

6.1 Edit Masters menu:

Get outline: gets the character that has been Sent from the Master Fount window. Note that Getting and Sending data can be done in any order. You can Send the data from the Master Fount window and then Get it in the Edit Masters window, or you can Get it first, then go back to the Master Fount window and Send it.

Send outline: after editing an outline, if you want to save it, you must send it back to the Master Fount window. You then have to Get it from within the Master Fount window.

Fill outline: fills in the outline in the window with a solid grey pattern. This colour is in the painting pad, so to erase it, you will need to select Clear Painting from the yellow menu. Also note, that since is it in a different pad, if you move the outline, the fill colour will not move with it.

Clear outline: erases the outline in the window.

Get grid: Gets grid information from the Master Fount window. You must Send Data from the Master Fount window.

Send grid: Sends grid information to the Master Fount window. You must Get Data in the Master Fount window.

Clear grid: erases the grid in the window.

Clear painting: erases the painting in the window.

Copy painting to file: copies a painting done with the paint tool to a file.

Get painting from file: gets a painting from a file.

Copy pen to file: copies the current pen to a file.

Get pen from file: gets a pen from a file.

Change parameters: displays a form for you to change the baseline, cap-height, left or right margin. The baseline to cap-height distance is always 0 to 10000. If you want a larger space above the cap-height, you must enter a number larger than 10000. If you want a larger space below the baseline, you must enter a value smaller than 0. You can also set a *sensitivity* value which says how close the cursor must be to a point in order for it to be selected. This value also affects the Filter command in the TRANSFORMS menu, described below.

Finish: terminates this window.

6.2 Outline

If Outline is selected, a menu of outline choices appears. By selecting various choices, you can edit the outline drawings. To create an outline, you should first select Create Linechain. Once there is a linechain in the window, you can edit it.

The menu selections are:

Move point: allows you to pick up a point and move it. Press Button 1 when you are on top of the point that you want to move and hold the button down until you have positioned the cursor to where you want the point. The point will follow the cursor. Then release the button.

There are currently two forms of Move Point. The first, which is what works when Imp is first started, allows you to move individual points. You can switch between the first and second form by pressing the character **M** on the keyboard. The second form allows you to select an existing point by pressing Button 1, then to draw a curve from that point with the button held down, and when the button is released, the linechain closest to the curve is moved so that the points lie along the curve. You must start near an existing point. If you are not close enough, nothing will happen. This form allows for more free-hand sketching of curves, but you must have an existing line chain for it to work. To get back to the first form, press the character M again.

Add point: Adds a point on a linechain between two existing points. Position the mouse over the position on the linechain where you want to add the point and click Button 1.

Link/Unlink: Links or unlinks two points together. Press Button 1 and position the cursor over the first point then hold the button down until you are over the second point. Once you have selected the first point, the points that can be linked to or unlinked from it will be highlighted with a red box when the cursor passes over them.
You can use this command to delete a single point. To do this the point has to be at the end of a linechain (that is, only connected to one other point), and you will have to *push* it off the end of the linechain. First select the point that is connected to the point you want to delete, then select the point you want to delete. The linechain connecting the two points will be unlinked and the end point will disappear.

Create linechain: creates a line of points connected by straight lines. Press Button 1 down when you want to put down a point and release it when the point is positioned correctly. Each point in a linechain will be connected. To make two separate linechains, create the first line chain, then select one of the other menu options, then select Create linechain again and create the second linechain.

There are currently two forms of Create Linechain. The first, which is what works when Imp is first started, allows you to position individual points. You can switch between the first and second form by pressing the character **C** on the keyboard. The second form allows you to draw a continuous line. Imp will automatically leave a trail of points along the line that you drew. This form allows for more free-hand sketching of curves. To get back to the first form, press the character C again.

Delete linechain: deletes a connected row of points. Position the cursor over one of the points in the linechain and press Button 1.

TRANSFORMS: switches to the TRANSFORMS menu (see below).

PARTS: switches to the PARTS menu (see below).

6.3 TRANSFORMS

Clicking this option results in a menu of transformation operations being displayed.

Fix/Free: either Fixes a point so that it can't be moved or Frees the point so that it can be moved. Position the mouse over the point that you want to select and press Button 1. A fixed point is marked by a larger dot than a free point in the working area. Fixed points are also marked by red dots in the reference area. If you want to filter or smooth a section of a linechain and don't want other points on the linechain to move, you can fix the begin and end points of the section you want to process. This provides a boundary to the filtering or smoothing operation.

You can also Fix or Free a series of points by pressing the button down and holding it while you sweep the cursor along an existing linechain. All the points close to the path of the cursor will be either Fixed or Freed depending on their prior state.

Filter Section: removes extra points from a curve section. The section can be delimited by Fixed points. Fix the beginning and end point of the section to be filtered, then select Filter Selection, then click one of the points in the section. If no points are Fixed, the entire linechain is filtered. The larger the sensitivity value given in the Change Parameters form, the more points are removed. Experiment with different values to get the effect you want.

Smooth Section: Adds extra points between existing points in order to create a smoother line. The section to be Smoothed can be delimited by Fixed points. Fix the beginning and end point of the section to be smoothed, then select Smooth Selection, then click one of the points in the section. If no points are Fixed, the entire linechain is smoothed. If there are too many points in the resulting section, use Filter Section to try to get rid of some of them. If you mistakenly smooth a linechain without fixing points, the **cancel** button will undo the smoothing.

Translate: moves a part in the direction indicated by a line drawn by clicking Button 1 twice. Position the cursor in the working area. Press the button and hold it down until you have positioned the initial mark. This mark becomes the endpoint of an elastic line. Press the button again and move the cursor until the line is equivalent to the distance and angle that you want to move the part. Then release the button.

This command, in conjunction with the Fix/Free command, can be used to stretch a character in a particular direction. If you fix several points on a character, then select Translate and move the character, the fixed points do not move, but the rest of the character does. This could be useful, for example, if you design a short stem with a serif at the top and bottom, you could then fix the points in the lower serif, then by translating the rest of the character make it taller, resulting in an ascender stem whose serifs match the smaller stem.

Scale: scales a part. Position the cursor in the working area and draw a line in the same way as for the Translate command. Move the cursor so that the line moves up or to the right to make the part larger, or so that it moves down or to the left to make the part smaller. The longer the line, the more scaling will be done. When you release the button, the scaling is performed. This is kind of arbitrary, so you will have to play around with it a little to figure out how it works.

BACK: returns to the main menu.

6.4 PARTS

The Part selection allows you to break up a design into a series of parts. Each part can be saved separately and then used to create new designs. For example, once you have designed the lowercase-n, you might want to use the stem or the arch in several other characters. You can break the n into two parts and save them. When you design the lowercase-m you might want to use the arch from the n as a model for the arches in the m. You can overlap outlines in different parts, whereas if they were in the same part the design would not be coloured in correctly when you filled it or scan converted it.
One part is always current and its line chains are drawn in black. The other parts are drawn in pale grey. All the outline and transforms commands only apply to the current part.

Next Part: makes the next part in the part list current. It automatically moves back to the first part when it reaches the end of the list.

Create Part: Add a new part to the parts list. You must have Floated some linechains before selecting this option.

Delete Part: remove the current part.

Float Line: floats a linechain so that it can be extracted from the character and made into a part. Click Button 1 over any of the points in the line. The entire line will turn red. More than one line can be floated at a time. Then select Create Part and all the red lines will be made into a separate part.

Drop Line : un-floats a connected linechain. Position the cursor over one of the points in the line and click Button 1. The line becomes part of the current part and so turns black.

BACK: returns to the main menu.

6.5 Grids

This selection allows you to draw guidelines on the window. You probably want to set your guidelines before you design any characters. To add a line to the screen, first select the type of line you want, then move the cursor into the working area and press Button 1. This will result in a line being drawn on the screen. If you hold the button down, you can move the line. When you release the button, the line stays put. The choices are:

Change x-height: draws a solid line. This line is different from the dashed lines as it has a special meaning for Imp. It is used for such things as calculating the spacing of lowercase characters in the Compare Masters window.

Add horizontal: draws a dashed horizontal line.

Add vertical: draws a dashed vertical line.

Delete line: move the cursor to the line you want to delete, click Button 1 and the line will disappear.

Move line: move the cursor to the line you want to move, press Button 1 and hold the button down. As you move the cursor the line will also move.

Measure: this can be used to measure distances between objects on the screen. Press Button 1 and hold it down until you have positioned the cursor at the starting point of your measurement, then release it. Press Button 1 again and hold it down until you have positioned the cursor at the end of the span you want to measure, then release the button. The distance between the start and end point will be displayed in a small white window below the reference area and above the selection area.

As you move the cursor, the distance is updated continuously, so you can also use this feature to measure a distance you haven't really set up yet; you can place one grid line, then measure a certain distance away from it with the measuring tool, then place a second grid line at the point indicated by the end of the measurement.

It is a good idea to measure the stem width of your characters and make a note of the value. When you get to the Compare Masters window where you calculate spacing for your characters, you will need to know what the width of the stems are in order to generate the correct spacing.

6.6 Paint

This selection allows you to draw shapes in the working area with an edged pen. You might prefer to sketch some character shapes rather than drawing outlines directly, so this pad allows you to do some preliminary drawing. After the shapes are defined, you can then trace around the outside of them with outlines. To save the painting in a file, select the command Save Painting from the yellow menu and when prompted, fill in a name for the file. To retrieve a painting that you have saved, use the Get Painting command from the yellow menu. You will also have to fill in the name of the file. There are two similar commands in the yellow menu to save and restore a pen.

Paint: draws with the selected pen nib. If you are drawing with the stylus, you will have to move it slowly to get a smooth connected line. The mouse produces a darker line but is harder to control.

Erase: erases with the selected pen nib. To erase the entire picture, use the Clear Painting command in the yellow pop-up window.

Pen Angle window: displays the current pen width and angle. You can change the pen by moving the cursor to this window and pressing Button 1. While holding the button down, move the cursor so that the line displayed has the width and angle that you want. Release the button and the pen is fixed.

7. Compare Masters

The Compare Masters window allows you to determine spacing for characters in a Master Fount. You can display up to 10 characters in two rows of 5 characters each.

You must have already created the characters using the Edit Masters window.

7.1 Compare Masters menu

Get character + spacing: gets a character with its spacing information from the Master Fount window. Before you get a character, be sure to change the current slot value if there is already a character in the current one. Otherwise the new character will overwrite the last one. (This may, however, be what you want to do.)

Copy character + spacing: copys a character to the current slot from the designated one. A small form will pop up with a space for you to indicate which slot you want the character copied from.

Save character spacing: save spacing information for this character. You will need to Get Data in the Master Fount window. If you try to save the spacing with the wrong character, an error message will be printed on the black and white monitor.

Get interline spacing: gets interline spacing information from Master Fount. You must Send Data from the Master Fount window.

Save interline spacing: saves interline spacing information in Master Fount. You must Get Data in the Master Fount window. If you try and save the interline spacing with the wrong fount, an error message will be printed on the black and white monitor.

Get Fount Data: gets the fount data from the Master Fount window. You will need to Send Data from the Master Fount.

Calculate spacing: calculates the spacing for the character in the current slot. This takes a while so it is a good idea to save the spacing after it is calculated in case the system crashes so you won't have to do it again. Before the calculation can be done you must fill in the form that appears. You must say whether the character is to be treated as uppercase or lowercase, and what the thicknesses of the thick and thin strokes are. Once you have filled in the form you must hit **Return** on the keyboard. If you don't want to do the calculation after all you can hit **Cancel** instead.

Clear : clears the characters on the screen.

Finish: terminates this window.

7.2 Compare Masters lower left window

This window gets its information from the Master Fount window. There is no way for you to type any text into this window. It is displayed for information purposes only. To get information into it, you must select Get Fount Data in this window, then select Send Data in the Master Fount window.

Family: **cambridge**
Style: **roman**
Size (decipoints): **100**
Space Table: **standard**
Interline Space: **15000**

Spacing Constant: **0**

7.3 Compare Masters lower right window

This window displays information calculated by the spacing routine. The top item in the list is the Current Slot. Change this value to change which slot the window is either reading, writing, or calculating spacing for. The value must be a number from 1-10 inclusive. The next item, the character's name, cannot be changed, but is included in this window since it is associated with the rest of the data for the character selected by the Current Slot. The other items are values related to the spacing for the character. They are calculated automatically by the spacing routine, so you probably don't want to change them, but you are allowed to if you don't like the spacing values calculated by the programme.

Current Slot: **1**
Character: **lowercase-n**
Name: **none**
Centre: **0**
Before: **0**
After: **0**
Kern with : **none**
by : **0**

8. Raster Fount

The Raster Fount window allows you to retrieve a saved raster fount or to create one by converting a master fount into a raster fount. Note that a raster fount requires that you specify the resolution of the output device that this design is meant for. If you don't have a particular output device in mind, a number in the range of 200-500 would give you a good idea of what the design will look like as a raster fount. The number of bits per pixel is set automatically to 1 as this is the value needed for founts for printers. This value can be changed if the fount is a grey scale fount with more than 1 bit per pixel.

Raster Fount items:

Family: **cambridge**
Style: **roman**
Size (decipoints): **100**
Resolution (pixels per inch): **240**
Bits per pixel: **1**
Space Table: **standard**
Letter: **lowercase-m**

8.1 Raster Fount Menu

Read in a file: read in a raster file from disk. The raster file must have already been created.

Scan convert from master: convert a master fount into a raster fount. Imp will look for a Master Fount window with a fount of the same name as the one you have given in the Raster Fount window. The size does not have to be the same and Imp automatically compensates for this. If you don't have a Master Fount window with a fount in it, you must create one, and read in the fount data from a fount file. You do not need to Send Data to the Raster Fount window, nor do you need to Get Data in the Raster Fount window.

Copy out to file: save a raster file on disk. Once you have created a raster fount or edited some raster characters, use this command to save them in an external file.

Get data: get data sent from the Edit Raster window. The data will need to be Sent from the Edit Raster window.

Send data: send data to the Edit Raster or Display Text window. If you are editing a raster, the character which has been selected by the small red box in the upper window will be sent to the Edit Raster window. You will need to Get Data in the Edit Raster window. If you are displaying text, all the character information will be send to the Display Text window (you do not need to send it character by character).

9. Edit Raster

The Edit Raster window allows you to edit the raster patterns created in a Raster Fount. Like the master editor window this window has a reference area and a working area. There is also a special display area at the bottom left corner which shows the character at the size it would be when printed on paper, or at a specified magnification of this size.

9.1 Edit Rasters menu

Get character: gets a character from the Raster Fount window. The character must be Sent from the Raster Fount window.

Send character: sends a character to the Raster Fount window. You must then Get the character in the Raster Fount window.

Clear: clears the window.

Change parameters: displays a form for you to change the margins and magnification. Changing the margins gives more or less space around the character for you to work in. Changing the Magnification changes the size of the character displayed in the small lower window. After you have changed the values, hit the **Return** key on the keyboard.

Finish: terminates the window.

9.2 Editing pixels

Once you have got a character for editing you can selected the area to work on by moving the box around in the reference area, using button 1. You can then change the pixels in the working area by moving the cursor to a pixel and clicking button 1. This steps the pixel colour to the next possible colour—for founts with one bit per pixel this means that the colour swaps between black and white each time the button is clicked. For grey scale founts with more bits per pixel the colour moves through a series of greys. If you hold the button down you can sweep the new colour through other pixels as you move the cursor.

10. Display Text:

The Display Text window allow you to display text from an external file using the characters in a raster fount. First, create the file containing the text you want to display (check with a system guru if you don't know how to do this, or for a quick and easy method, see the examples section). You must have a raster fount displayed in the Raster Fount window. The display text routine reads in the text file and tries to match up characters in the file with names of characters in the fount. A lowercase character will be matched with a character having the name 'lowercase-' followed by the character, e.g. 'lowercase-n' in the fount will match the character 'n' in the text file. Uppercase characters are specified by 'Uppercase-' followed by the character, e.g. Uppercase-C. Numerals are specified by 'Digit-' followed by the digit and punctuation is specified by a descriptive name for the punctuation sign, e.g. QuestionMark, Period.

10.1 Display Text lower window

This window gets its data from the Raster Fount window. You can't type anything into it—it is just for information.

Family: **cambridge**
Style: **roman**
Size (decipoints): **100**
Resolution (pixels per inch): **240**
Bits per Pixel: **1**
Space Table: **standard**

10.2 Display Text menu

Get Fount Data: gets fount information from Raster Fount window. You must Send the data from the Raster Fount window. This information is displayed in the white form at the bottom of the window.

Display file: pops up a form and requests the name of the file to be displayed, the magnification of the characters, and the spacing constant to be used. The spacing constant will be added between each of the characters.

Magnification: **2**
File name: **cambridge-test**
Spacing constant: **0**

Finish: terminates the window.

11. Finish

When you are done using Imp, you must first Finish each of the windows, then Finish the program. Each of the windows has a Finish selection in its pop-up yellow menu. If this is selected while in the window, the window and its icons will disappear from the screen. When all of the windows have been removed, select Finish in the background menu and Imp will terminate. If you try to Finish the program while there are still windows on the screen, an error message will be displayed on the black and white monitor.

12. Saving files

If you are working with the paint command, you can select an item from the yellow menu to save your painting in a file. You will be prompted on the screen for the name of the file. After you have entered the name, press the Return button on the keyboard.

If you are working with the Master Editor or the Raster Editor, the process of saving your files is slightly more complicated. You should, however, make a practice of doing this fairly often, for if Rainbow crashes, you will lose everything you have done since the last time you saved everything.

To save a file, select one of the Fount windows (either Master Fount or Raster Fount), make sure the labels are filled in, then select the menu item on the yellow menu that says Copy Out to File. This will take all the information that is in the Fount and save it permanently. If you have edited some characters, be sure to copy them to the Fount window before you save the file. See the section at the end of this document for an example on saving a file.

If you select Compare Masters, and calculate spacing values for your characters, you will have to save the spacing information in a similar manner to the way you saved the character information if you want it saved permanently. Unless you plan to make extensive changes to your characters, it is a good idea to save the spacing information once you have computed it, since it does takes quite a while to compute. You don't want to have to repeat the computation every time you want to look at the character spacing. To save the information, Send the character data from Compare Masters, then switch to the Master Fount window, Get the character data, and then Copy Out to File.

To delete a character, you can simply delete all the information for the character, specifically its name and outline or raster. You delete the outline or raster by sending a blank character from the editor window.

13. Troubleshooting

Rainbow can be rather grumpy and so this section tells you what to do when things go wrong. Be forewarned, things are likely to jam up at some point and you will need to refer to this section to get started again. There are also a few tips here on how to prevent Rainbow from getting jammed up, so even if you are just beginning, look them over to save yourself some trouble later on.

13.1 How to Prevent Problems

Rainbow cannot support a lot of windows on the screen at one time. If you are working in several windows at once, it is best to Close Down one window before opening another. Also, the system can get jammed if you try to use the Raster Editor and the Master Editor at the same time. You should Finish one of the windows before starting up the other one. Note that Finishing a window and Closing Down a window are not the same thing. To Close Down a window, you click either Button 1 or Button 3 in the icon in the window tag which has the arrows pointing inwards. The large part of the window will disappear. To get it back, you can click the icon which has the arrows pointing outwards. When you Finish a window, the window is completely removed from the display and cannot be retrieved (unless you hit the **cancel** key immediately). To start a new window, you must select it from the background menu.

If you have just edited a character and want to save your changes, there is a quick way to save what is on the screen without having to write all the character information out to a file. Press the **B** button on the keyboard. This will *backup* the screen information. If Rainbow crashes, once you have restarted it you can type **R** to get the display back to what it was when you last hit the B character. It is probably a good idea to backup the screen every few minutes if what you are doing is something you really want to save. If Rainbow crashes and you decide you don't want your backup restored to the screen, just don't hit the R character when Imp starts up again. Rainbow is also sometimes picky about switching between the mouse and the stylus. If you prefer to work with one or the other, it is best to start out using that device than to switch halfway through the session. If you decide to switch devices, it is best to save what you have done first.

13.2 Restarting Rainbow

If the cursor movement stops and hitting the Return key on the black and white keyboard doesn't result in any scrolling of the text on the black and white monitor, then the system is probably jammed. You will have to start over again. Note that the procedure for starting over is not the same as starting from scratch.

Using the black and white keyboard, press the **Break** key. In response to the prompt, type **csm.** Then type **sys rainbow -m rainbow.** You will then be prompted for your login name and your password. If you get an error message instead of the login prompt, type **R** which will repeat the sys rainbow command. After you have logged in, the system will automatically start up again.

14. Example Scenarios

14.1 Example 1: Editing and Saving files

Assuming you are working with an existing fount, the sequence will be something like the following with yellow menu commands shown in brackets. You can use a similar sequence for a raster fount too.

1. [Select Master Fount]
2. enter name of fount
3. [Read in from file]
4. select character to be edited (using Button 1 and moving the red box)
5. [Send Character information]
6. Close Down Master Fount (use icon in the tag)
7. [select Edit Masters]
8. [Get Character information]
9. edit the character
10. [Save Character information]
11. Close Down Edit Masters (use icon in the tag)
12. Open Master Fount (use icon in the tag)

13. [Get Character information]
14. [Copy Out to File]

You can send several characters before you copy them out to a file, but if you have done extensive editing, it is best to save your changes often.

If you are editing a new character, in step 4 select an empty space and skip steps 5 and 8 (you don't have to send any information since there isn't any, and you don't have to get any because you didn't send any).

If you are editing a new fount, you can do steps **7-11** first, then steps **1,2** and **4**, then steps 13-14 (or you can do them in the order shown, either sequence will have the same result). Either way, you should leave out step **3** as the new fount will not yet be in a file.

14.2 Example 2: Creating a Text file for use with Display Text

To create a text file, you need to go back to the black and white keyboard. First press the **CTRL** key and **S** at the same time, then type **08** This will result in the prompt Rainbow-8 on the screen. Then type **input test.** Then type the test characters you want to display (not more than 6-8 lines as more than that won't fit in the Display Text window), and finish with **/*** on a separate line. You have now created a file named *test*. You can go back to Imp and use this as the name of your input file.

Appendix A

```
  0 c0 1 c1 2 c2 3 c3 4 c4 5 c5 6 c6
  7 c7   8 c8   9 c9  10 c10 11 c11 12 c12
 13 c13 14 c14 15 c15 16 c16 17 c17 18 c18
 19 c19 20 c20 21 c21 22 c22 23 c23 24 c24
 25 c25 26 c26 27 c27 28 c28 29 c29 30 c30
 31 c31 32 c32
 33 Exclamation  34 OpenDoubleQuote  35 Hash [CloseDoubleQuote]
 36 Dollar  37 Percent   38 Ampersand
 39 CloseSingleQuote  40 OpenParenthesis  41 CloseParenthesis
 42 Asterisk  43 Plus   44 Comma
 45 Minus  46 Fullstop   47 Oblique
 48 Digit-0  49 Digit-1   50 Digit-2
 51 Digit-3  52 Digit-4   53 Digit-5
 54 Digit-6  55 Digit-7   56 Digit-8
 57 Digit-9
 58 Colon  59 Semicolon   60 OpenAngleBracket
 61 Equal  62 CloseAngleBracket  63 QuestionMark
 64 At
 65 Uppercase-A  66 Uppercase-B  67 Uppercase-C  68 Uppercase-D
 69 Uppercase-E  70 Uppercase-F  71 Uppercase-G  72 Uppercase-H
 73 Uppercase-I  74 Uppercase-J  75 Uppercase-K  76 Uppercase-L
 77 Uppercase-M  78 Uppercase-N  79 Uppercase-O  80 Uppercase-P
 81 Uppercase-Q  82 Uppercase-R  83 Uppercase-S  84 Uppercase-T
 85 Uppercase-U  86 Uppercase-V  87 Uppercase-W  88 Uppercase-X
 89 Uppercase-Y  90 Uppercase-Z
 91 OpenSquareBracket   92 Backslash
 93 CloseSquareBracket  94 Circumflex
 95 Underline  96 Grave
 97 Lowercase-a  98 Lowercase-b  99 Lowercase-c 100 Lowercase-d
101 Lowercase-e 102 Lowercase-f 103 Lowercase-g 104 Lowercase-h
105 Lowercase-i 106 Lowercase-j 107 Lowercase-k 108 Lowercase-l
109 Lowercase-m 110 Lowercase-n 111 Lowercase-o 112 Lowercase-p
113 Lowercase-q 114 Lowercase-r 115 Lowercase-s 116 Lowercase-t
117 Lowercase-u 118 Lowercase-v 119 Lowercase-w 120 Lowercase-x
121 Lowercase-y 122 Lowercase-z
123 OpenCurlyBracket  124 VerticalBar
125 CloseCurlyBracket
126 Tilde
127 c127
```