An Implementation of POSTSCRIPT

Crispin A. A. Goswell

Rutherford Appleton Laboratory
Chilton, Didcot,
OXON OX11 0QX.

ABSTRACT

This paper describes an implementation of POSTSCRIPT for previewing use on workstations with high resolution bitmapped displays.

It discusses implementation of storage management, area fill and line drawing, imaging, fonts and font caching. Treatment of bezier curves and dashing is explained.

Various hints are given on improving performance, including caching fonts on disk, treating thin lines specially and magnifying bitmaps.

Finally there is a brief discussion of porting experiences with the interpreter. It is written in the C language and runs on the UNIX† operating system.

1. INTRODUCTION

POSTSCRIPT has all the features required of a general purpose programming language[1] and, indeed, they can be separated from the graphics primitives to form a useful interpreted programming language.

POSTSCRIPT is simple to manipulate, as it is dynamically scoped and has a simple syntax.

It might be thought that a general purpose programming language would be overkill for such an application. It is observed, however, that the lack of a simple feature creates disproportionate problems for the user. For example, many graphics description languages have no general way of repeating a diagram in several guises. Also, some drawing algorithms need to know the device resolution in order to do rounding correctly. Without some kind of two-way communication, such algorithms could not be implemented portably. POSTSCRIPT solves this problem by putting the device dependent computation in the device.

The POSTSCRIPT language fits together neatly: all the features present are both necessary and sufficient. This approach of using the fewest number of concepts which provide the required functionality has proved successful on systems like UNIX. Our implementation was begun from the POSTSCRIPT Language Reference manual. We obtained an Apple LaserWriter which we have since used as a model implementation.

This paper assumes some programming knowledge of the POSTSCRIPT language.

2. THE POSTSCRIPT LANGUAGE

Figure 1 shows an architectural overview of our POSTSCRIPT interpreter and environment.

2.1. Postfix notation

POSTSCRIPT is a post-fix language, and while this is not the form that people prefer, it is by far the easiest for programs to generate. Almost all POSTSCRIPT source is machine generated, and the interpreter operates directly from source. This is so that there need be no worries about non-printing characters (most networks are not transparent to binary files).

---

† UNIX is a registered trademark of AT&T in the USA and other countries.

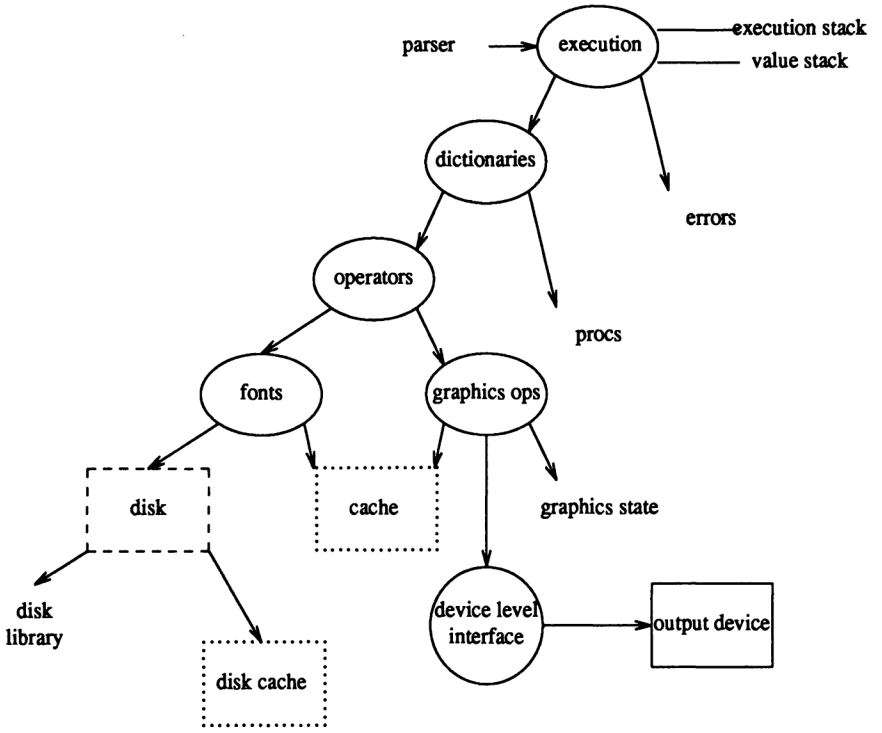

Fig 1. Architectural overview of our PostScript implementation.

2.2. Data types

POSTSCRIPT has a small fixed set of data types which describe objects in the system. Objects have a structure associated with them which contains type information, access control flags and a value.

Dictionaries, *Names* and *Operators* form the essential parts of the POSTSCRIPT execution environment. The interpreter looks up any names (symbols or tokens) which it encounters on the input stream in the dictionary stack. This stack can be dynamically added to by the user for creating local scopes. Dictionaries may take (almost) any data type as a key or value. It was observed that there are no operators for removing items from *dictionaries* (except **save** and **restore**), they were implemented by hash tables with linear chaining. This is the simplest scheme which fits the requirement. Most names in the system dictionary at the bottom of the stack are POSTSCRIPT operators, which do the real work. Because the naming is dynamic, extra names can be used to replace existing POSTSCRIPT functions with modified alternatives. This makes POSTSCRIPT very easy to alter.

Names in POSTSCRIPT can be compared quickly so that they can be used efficiently as dictionary keys. This requires a unique storage pointer. The *name* table was thus implemented by using a simple unbalanced tree structure with no garbage collection. When a *string* is converted to a *name*, it is looked up in the tree. This usually happens during parsing, though there is a separate POSTSCRIPT operator, **token**, for this purpose[1]. The resulting *name* object contains a unique reference to a tree node.

Operators were implemented by using function pointers with added argument type information. We arranged for arguments to be pulled off the stack and type-checked before being passed to the

[1] **token** is one of many key operators in POSTSCRIPT whose presence have a controlling influence on an implementation.

implementing function. This was an important feature, as it localised the type-checking for all but the most polymorphic operators. No escape mechanisms were added for operators with strange arguments, such operators simply declare themselves to have fewer arguments and then use extra ones off the stack. Result types are not checked, since they are tested as arguments by everything which uses them, but the number of results may be declared so that stack overflow and underflow checks can be localised.

Files, *Arrays*, and *Strings* are all usefully executable: an executable *Array* is a procedure body – there are special array literals for expressing these. This relies on the fact that parameters are passed on the stack and that binding is dynamic, so procedure bodies don't also have to be function closures. *Strings* are executed by parsing them like input. It is notable that the string quotes are parentheses, which are thus nestable.

There are a few subtleties with *files* - certain file names refer to pseudo-files which return complete syntactic units to the interpreter for processing. We implemented this by reading into a *string* and making that executable.

Integers, *Reals*, and *Booleans* are traditional.

Marks are used to mark unbounded lists on stacks, and have no interesting operations. *Nulls* are similar.

FontIDs are a means of internally referencing fonts (described later).

SaveObjs refer to saved contexts, which are described in the section on Storage Management.

There is no real notion of assignment in POSTSCRIPT but it is possible to replace key/value pairs in dictionaries or place values in *arrays* and *strings*.

2.3. Polymorphism

A large number of POSTSCRIPT operators take arguments of differing types. The **get** operator, for example, can get a value from an *array*, *string* or *dictionary*. It is a fairly simple extension of this idea to allow arbitrary polymorphism.

Our implementation has an extra dictionary for each data type which contains a simple operator implementing a version of a polymorphic operator for that data type. Many operators in the system dictionary point at a generic operator for the kind of polymorphism in question, and that generic operator calls a specific operator from a type dictionary. This is similar to the "discriminator functions" in COMMONLOOPS[2] . This facility allows users to add or change the standard data types and operations. New operation/data type pairs can be added even in the POSTSCRIPT language itself.

It is not possible to construct new data types in POSTSCRIPT, as this involves changing data formats, but it is fairly easy to link new source modules into the interpreter with code for a new data type.

2.4. Error handling

POSTSCRIPT has an effective error handling mechanism, which is quite tidy and general when compared with many other systems.

When an error occurs, an error operator is called from a special dictionary. These operators are replaceable, like most other parts of the system. The change in control flow required by error handling is handled separately by two operators called **stop** and **stopped**. These implement "catch" and "throw", as found in some functional languages.

An executable array is passed to the **stopped** operator which calls it and normally returns **false**. If a **stop** is encountered, execution resumes immediately at the **stopped** operator, which then returns **true**. **stop** unwinds the "execution stack", which is directly accessible from POSTSCRIPT.

The above has a considerable influence on the design of the interpreter: Some operators need to take a POSTSCRIPT procedure as an argument, which they may call a number of times. The flow control operators work this way for example. When an operator needs to do this, it issues a "call-back". The obvious way to implement call-backs would be to call the interpreter recursively. To implement **stop** and **stopped** would then be difficult to do cleanly and portably (without manipulating the C runtime stack and compromising the reliability of the system). Instead of this, a virtual machine was built which executes

POSTSCRIPT objects, and allows POSTSCRIPT operators to explicitly alter the execution (return address) stack when necessary. This makes operators like **stopped** and **stop** fairly easy to implement, but makes call-backs into POSTSCRIPT code slightly more complicated. Call-backs are achieved by pushing a *continuation* operator on the execution stack, followed by the POSTSCRIPT object to be executed. When the operator finishes executing, the virtual machine will drop into the POSTSCRIPT code on the stack and when that finishes, the continuation function will be executed. In order to perform a loop, for example, the continuation would push the POSTSCRIPT object again, and another instance of the continuation function. Items such as the bounds of **for** loops and such are also placed on the execution stack, but they are removed by the continuation operators before the virtual machine attempts to execute them.

Implementing the **exit** operator in this context requires a little thought: how far should this operator unwind the execution stack? In our implementation the flow control operators place a marker on the execution stack. **exit** searches down the stack until it encounters one, and removes everything above it. Execution will then pick up just after the control operator. The **stop** and **stopped** operators work in a similar way, except that they use a different marker, so that some checking is possible.

Arrays and *Strings* are executed by placing their tail (sub array starting after the first element) back on the execution stack and placing the first element above it. If the array is of zero length, nothing happens.

Files manage their own file pointer. A custom lexical/syntax analyser was built to interpret *files* and *strings*, partly because POSTSCRIPT is very simple to interpret and partly for the control obtainable by doing this. The parser was parameterised on the character input function to avoid code duplication.

2.5. Storage Management

POSTSCRIPT has a simple model of storage management. Objects are created on request, and removed when the user loses his last pointer to them. This would seem to imply a requirement for garbage collection, but for the existence of **save** and **restore**. **save** means "save a snap-shot of everything" and **restore** means "restore a previous snap-shot". There are two aspects to **save** and **restore**: one is that the interpreter state is saved and restored, and the other is that garbage collection can be performed when the **restore** occurs. There are some curious exceptions which are relevant: the contents of the stacks are not disturbed, although if they would cause dangling references, a **restore** operation generates an error. Also the contents of a *string* is not defined after a **restore**. It is implicit that file pointers do not get moved back either, though it is not stated anywhere. The POSTSCRIPT Reference Manual also states that **save** and **restore** are efficient enough that they may reasonably be used to save context around a number of assignments and restore it afterwards.

One possible implementation of **save** and **restore** would be to copy the entire data space maintained by POSTSCRIPT, but the requirement of efficiency precludes this as a practical choice.

Another implementation would be as follows:

Note when an element of an array or dictionary is altered, and preserve the old value if it has not changed since the last **save**. This means that there would have to be a record of the save level at the last assignment for each element of an array or dictionary, and checking code on the **put** operator in its various disguises (e.g. **def**). A *SaveObj* would then have a pointer to a list of old values for items which have been assigned since its creation (initially null). **restore** would search this list, and put back all the old values and save levels. It would then deallocate all objects created since the save object was created. This could be done either by moving back an allocation pointer, if new objects are allocated in consecutive store, or by following through a list of allocated objects if they are not.

Although our implementation does not yet support **save** and **restore**, we will use the latter approach, as we allocate many data structures which are not bounded by scope constraints, e.g. bitmaps for windows, I/O buffers, cache table entries and so on. An implementation could use fixed size tables for these items in a dedicated environment such as a printer, but it is simpler in a virtual storage environment to allocate things dynamically.

One modification we might make, rather than stringing all new objects together, would be to batch allocation of objects, starting a new "bucket" at each **save** or when one fills up. This simple technique would vastly reduce the work done by the standard storage allocator and bring it in line with the efficiency

of a contiguous allocation scheme.

Currently, our implementation does no garbage collection at all, and we rely on virtual memory to keep the interpreter running. Also, the only the thing our **save** and **restore** operators do is to save the graphics state. It turns out that this is sufficient for many purposes.

3. GRAPHICS

POSTSCRIPT distinguishes the generation of graphical shapes from their imaging. Shapes are called *paths*, and describe a set of possibly closed outlines. There is a variety of operators for adding lines, curves and arc segments to a path. Curiously a path is not a data type in POSTSCRIPT as such. The intention is that users should create procedures for generating paths and manipulate those instead.

A path is just one element of a fairly large *graphics state*, which can be altered prior to using an imaging operator. Much of it remains constant most of the time, so it forms an implicit parameter to these operators. In general very little of it needs to be changed frequently. The graphics state is also stacked so that changes made can be quickly undone with **gsave** and **grestore**.

There is also the notion of a current output page. This is the destination of the imaging operators and it accumulates the effects of them independently of **gsave** until a **showpage** or **erasepage** occurs.

3.1. Paths

POSTSCRIPT separates the generation of polygons from their imaging, so the same primitives produce shapes to draw and fill. This has the advantage that very complex shapes may be generated in a device independent manner and with no lack of generality.

Many operators in POSTSCRIPT do complicated things with paths, so a doubly linked list was used to implement them. This scheme proved very flexible and also simplified the storage management. A list of linked lists might have been better choice, since it would then have been easier to find "closepaths", however the use of a flat data structure kept the code simple. Most operations on paths require generating a new path from an old one, so very little juggling of path segments is necessary.

3.2. Painting model

An important feature of POSTSCRIPT is the graphical imaging model chosen. Because it was designed for printer devices, POSTSCRIPT uses a paint model for imaging (any output colour replaces any colour previously there). A path describes an outline which can be stroked with a pen or filled with colour. A path may also be used as a clipping region, which is like filling a region and then using it as a stencil.

3.3. Filling and Clipping

Filling regions needs a certain amount of thought. The intention is that filling should behave like paint, and cover anything previously painted. How then does one arrange to fill a self-intersecting polygon in such a way that it behaves like paint? Many systems use the so-called even/odd rule: starting at the outside of the region, count edges and paint when the count is odd and not when it is even. One problem with this is that if a region is self-intersecting, parts that are intersected do not always get filled.

POSTSCRIPT provides an alternative *non-zero winding number rule*, which counts the difference in the number of passes an edge makes across a line from any point to the outside. See figure 2. Clockwise passes count one way and anticlockwise passes count the other way. In this way, a self intersecting region gets completely filled where the difference is non-zero.

Implementing area fill in POSTSCRIPT is far from simple. Not only can one fill a possibly overlapping and self-intersecting path, but one can use it as a clip boundary to clip further output. Clipping is cumulative (i.e. each **clip** operation makes the clip area smaller). One of the most intriguing POSTSCRIPT operators is **clippath**, which gives you back a path description of the current clip boundary, taking multiple **clip** and **eoclip** operations into account.[2]

The first approach used to implement area fill was to scan the bounding box one raster scan line at a

[2] **clippath** is another example of a key POSTSCRIPT operator.

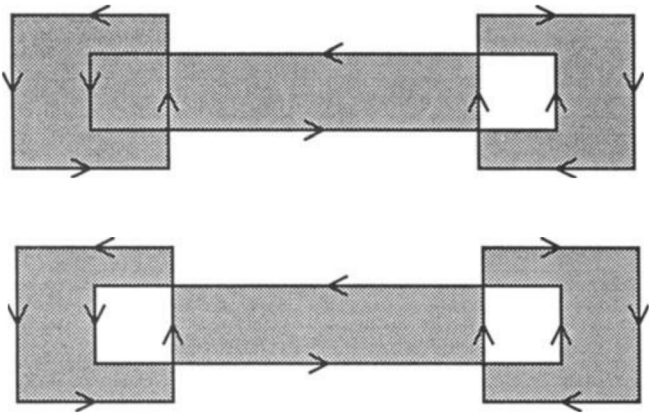

Fig 2. A comparison of the two rules for area fill.
Non-zero winding number above and Even/Odd below.

time. At each scan line, all line segments in the path which crossed the scan line were intersected with it (noting the direction of crossing) to establish a list of cross points. This was then sorted to produce raster line segments which could be filled according to the filling rule required (winding or even/odd). This algorithm is plainly inefficient, since it looks at every scanline, while most images drawn have interesting points which are widely spaced. Clipping could be achieved with this algorithm, but the clippaths produced would contain rectangular polygons describing raster line segments which would be unnecessarily large and slow to process.

A better way to implement **fill** and **clip** is to look only at scan lines (Y coordinates) where something interesting happens, such as a line starting or ending, or crossing another line. It is possible to scan in this fashion, generating trapezoid shapes at each interesting Y coordinate. This is described in a paper by Newell and Sequin[3] . See figure 3.

One reason why this algorithm has not yet been implemented is the problem of intersecting two sets of trapezoids to achieve clipping. It turns out that this is not the best way to think about the problem: clipping should be factored into the trapezoid generation so that the same code which does trapezoid decomposition also does clipping. From this, it is apparent that the clippath only needs to be reduced to trapezoids if it is intersected with another clip area, since the new clip area is getting clipped. This explains a feature of the Apple LaserWriter that puzzled us for sometime, which was that clip paths did not always seem to be reduced to trapezoids. This clue eventually hinted at the solution to the trapezoid intersection problem, above.

Some care is necessary in rendering trapezoids on an output device. With a scan line algorithm such as the one we are currently using, any deviations due to rounding are relatively minor, because they are localised. If two trapezoids share a vertical edge, the coordinates may be rounded so that the lines have slightly different gradients. If the trapezoids form the edge of a single line from the point of view of the user, he will be particularly disturbed if it doesn't look straight. One solution to this problem is to use floating point coordinates even at the output stage. A possible alternative is to define a trapezoid by the complete lines from which its vertical edges were formed, clipped by the interesting Y coordinates. If the vertical line segments are rounded, the minor differences in gradient are much less likely to be noticeable, especially since the line segments will be longer.

Fig 3. Trapezoid decomposition in action.

There are difficulties with clipping: how does one efficiently render text through a complex clipping boundary? One possible solution is to use a shadow mask bitmap containing a fill of the clip area. It may also be noted that bounding boxes on clip and fill areas may eliminate a large amount of work.

There are a number of techniques available here, but so far only simple area fill with no clipping has been implemented.

3.4. Strokes

Strokes (lines) in POSTSCRIPT are fairly complicated: most graphics systems provide line drawing, and some have facilities to allow thick lines to be drawn essentially by using a line drawing algorithm to drag a pen shape along a path and drawing it at every pixel. This generates poor results at corners and ends when the pen shape is large, as the Smalltalk book[4] demonstrates: it is fine for draft use, but not really good enough for printed copy.

POSTSCRIPT provides several options for finishing strokes at ends and corners to prevent ugly joins. Round, Mitered and Bevelled corners are filled in by describing the shapes which fill the gaps between the rectangles that form the line bodies.

POSTSCRIPT has a **strokepath** operator, which replaces the current path by a path which describes its outline, including the line ends and joins. The definition of this operator is that if the resulting path is filled, it will look the same as the original path if stroked, so **stroke** can be implemented by **strokepath fill**.[3] See figure 4. Working out exactly where to place the polygons which describe an arbitrary stroked path is simplified by transforming the coordinate system to make the stroke lie along an axis in a conventional direction. There are certain difficulties with closed paths, because the algorithm has to look to see if the path is closed before deciding whether to add end points.

We made one simple optimisation to the **stroke** operator to using native line-drawing for thin strokes. This made an enormous difference to performance, and also produced much better results for previewing output from programs such as *pic*(1). It is probably worthwhile treating two to five pixel lines specially also, but we haven't done this yet.

3.4.1. Curves

POSTSCRIPT paths can contain line and curve segments. In order to make the imaging and path transformation algorithms easier, an algorithm is used to convert curve segments into enough line segments to approximate the curve to the required accuracy.[4] POSTSCRIPT has a parameter in the graphics state which states by how many device pixels the flattened curve is allowed to deviate. A simple algorithm for Bezier curve flattening is recursive bisection[5]. The recursion is not to a fixed depth, since Bezier curves can have widely different curvatures along their length, but stops when the curve is sufficiently flat. See

[3] **strokepath** is another key POSTSCRIPT operator.

[4] There is another key POSTSCRIPT operator called **flattenpath** which does curve flattening explicitly.

Fig 4. This illuminating example demonstrates how miters are formed.

figure 5.

Circular arcs can be approximated be dividing them into Bezier curves of small enough curvature. The Apple LaserWriter uses 90 degree segments, so our implementation does the same.

The use of **flattenpath** inside the interpreter has the nice property that no other operator needs to know about curved lines.

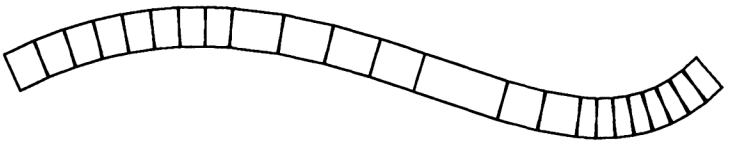

Fig 5. Curves can be flattened to line segments.

3.4.2. Dashing

POSTSCRIPT supports dashing of lines in a very general way. Dashes are expected to follow around corners with the correct length, though no attempt is made to force them to appear at corners. The reason for this is so that curves may be flattened before the result is dashed. See figure 6.

In our implementation, a small amount of path juggling was necessary to get closed paths to connect up properly when they begin and end with the visible part of a dash. Since several connected line segments may form one curved dash segment immediately preceding the end of the path, a piece of the dash pattern may have to be chopped out and joined to the piece at the beginning of the path. This is one of the few cases where the new path has to be altered in situ as it is being generated.

strokepath calls **flattenpath**, then the dashing routine, and then it generates the path that would draw around the resulting path. This is so that the dashing routine doesn't need to know how to follow around curve segments. It is curious to note that there is no operator which returns a dash description in a manner similar to **flattenpath**.

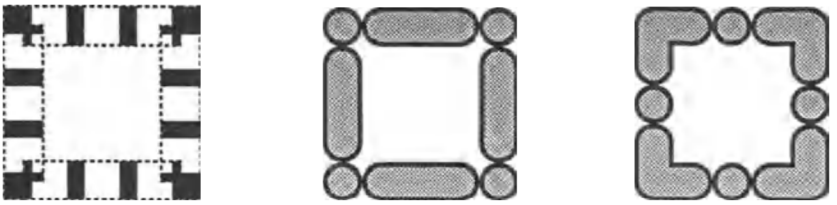

Fig 6. Getting dashes to follow around corners is hard.

3.5. Images

POSTSCRIPT has support for bitmapped images, but in a device independent way. The model is that images are a short-hand for a grid of coloured squares. Two variations are supported: **imagemask** which allows a square to be either of the current colour, or absent; and **image** which allows a grayscale image of a range of pixel depths for the squares. The grid always occupies the unit square in user coordinates, so that the size of the definition does not affect the shape of resulting image. The unit square is transformed using the normal coordinate transformations to make the picture large enough to see.

Images can clearly be implemented by scanning the incoming data and generating small square paths, then filling them with colour. Our first implementation used this method, which is adequate for small images and large magnifications.

It is simple to detect when an image happens to correspond with the device resolution and orientation and use a display Raster Operation to render it quickly. An algorithm for simple magnification by integer amounts is described in Rob Pike's SIGGRAPH course notes on bitmapped graphics[6], and also in the Smalltalk book. See figure 7.

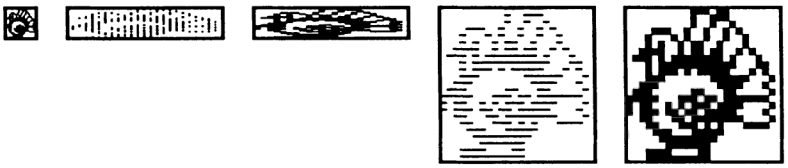

Fig 7. Integer magnification.

Magnification by non-integer amounts was not difficult to implement: essentially the difference is that there are two spacings between spread columns and rows, which together reach the correct width. The algorithm was adapted from the method of padding characters to a fixed width in text justification. The smear operation which follows the spread may in fact make some "pixels" too large by one pixel. A slower, but more accurate reverse sampling method would prevent this, but has not yet been done on our implementation.

Rotating an image by generalising reverse sampling should be fairly easy, but again this hasn't been done yet. Our code currently looks for fast simple cases and drops through into more general code when necessary.

Oddly, although there are facilities in the rest of POSTSCRIPT for supporting colour, there is no support for colour images. This is a curious and unfortunate omission.

3.6. Half-toning

POSTSCRIPT was specifically designed for use with bi-level raster devices, so half-toning is used to approximate gray shades. This is a familiar feature to users of bitmapped displays: typically a range of small speckle patterns is designed. They are replicated (tiled) over the area to be shaded, and clipped to its outline. An important feature of half-toning is that the tile shape should be aligned with the picture frame and not with the area being half-toned. If this is not done, seams will appear if two adjacent areas are tiled independently.

3.6.1. The half-tone screen

POSTSCRIPT adds a new twist to half-toning. Most bitmapped graphics systems do not actually provide a mapping between gray values and a suitable half-tone pattern. The user is expected to design each pattern himself, and that design is typically device dependent. POSTSCRIPT allows the user to specify a gray level intensity, and the interpreter maps this to an appropriate pattern. The user is permitted to specify a half-tone pattern by giving a solid spot function in POSTSCRIPT with a scaling size and rotation to an operator called **setscreen**. The spot function takes X and Y coordinates and returns a height (Z coordinate) at that point. The highest points are blackened first for the lightest grays and the lowest points last for the darkest grays. See figure 8.

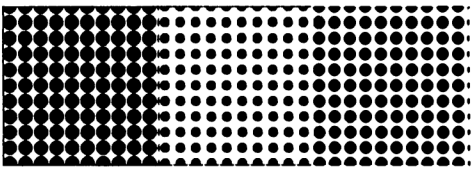

Fig 8. Half-tone screens have a solid spot function.

Half-toning has been implemented partly in POSTSCRIPT and partly in the interpreter. Extra operators were added to allow POSTSCRIPT code to request a set of coordinate values at which to sample the spot function. This is then applied, and the resulting set of heights is passed back to the interpreter. The half-tone machinery then sorts the sample values by height and uses them to turn on bits in a half-tone pattern. Any gray value requested is scaled and used as an index into the array of sample coordinates. All coordinates above the scaled value have a bit turned on in the resulting half-tone. In practice, all the possible half-tone patterns are generated in advance in this way for efficiency.

Because we use bitmapped displays, half-toning is done by RasterOp code, so the tile patterns cannot be rotated; our implementation currently ignores the rotation parameter. It is possible to replicate rotated bitmaps, but a fast algorithm for replicating with $\log_2 n + \log_2 m$ RasterOps instead of $n \times m$ RasterOps (essentially by copying the area replicated so far at each stage) covers a large number of pixels more than once when used rotated. It is also harder because of the unusual clipping requirements, so we haven't implemented this yet.

If a half-toning (three-way) RasterOp function is not available, it is possible to simulate tiling by using the source bitmap as a clip mask and copying the tile through that with replication. Since memory is relatively cheap, we have used a shadow bitmap to retain the replicated tile and thus do half-toning in a constant number of RasterOps.

3.6.2. Brightness Transfer Function

One other complication is that POSTSCRIPT provides a mapping function between user gray levels and device gray levels. The user can set this *transfer* function (normally the identity function), to any piece of POSTSCRIPT source with **settransfer**. When the setting is done, it is sampled in a similar way to the above in POSTSCRIPT, then a table of values is passed to the half-toning machinery.

4. FONTS

4.1. Fonts as graphical shapes

POSTSCRIPT fonts are treated as graphical shapes, so they may be stretched, scaled and rotated as much as desired. POSTSCRIPT has been criticised because it does not scale fonts according to the typographical convention, which involve changing the shape of the characters at different sizes. One answer to this is that there is nothing to prevent headline fonts being added to a printer. POSTSCRIPT is flexible enough that it would be trivial to down-load a replacement **scalefont** operator which uses different fonts at different sizes. The ability to scale and rotate a font arbitrarily is useful in itself, if only for special effects.

4.2. Simple font rendering

The POSTSCRIPT model is that the show operator sets up the Current Transformation Matrix according to the current point and Font Matrix of the current font. It then scans a string calling **BuildChar** with the font and the character code as argument. Each font has its own definition of **BuildChar** which does the appropriate thing for that font.

BuildChar is expected to use the character code as an index into an Encoding vector where it finds a character name. The character name is then used to find some representation of the character to draw (often a POSTSCRIPT procedure to draw it). Since the transformation matrix has already been set up, the character drawing is independent of size.

It turned out to be much easier to implement the most general form of **show**, which is **awidthshow**. The other more restricted forms (**show**, **ashow**, and **widthshow**) can all be implemented with almost no loss of efficiency using **awidthshow**. Implementing them all separately is tedious. The reason that *strings* are the primitive type and not characters is for efficiency and convenience. With the operators provided, a complete line of text can be shown with a single operator. See figure 9.

kshow, which allows kerning between every character shown, is best implemented in POSTSCRIPT as there is little performance advantage in calling back to POSTSCRIPT for every character over showing one character at a time from POSTSCRIPT.

stringwidth is currently implemented in our interpreter by using **nulldevice** to prevent output and showing the characters using **show**. It is then simple to get the position of the current point to compute the width. It would be a simple optimisation to compute the widths by examining the metrics in a font dictionary.

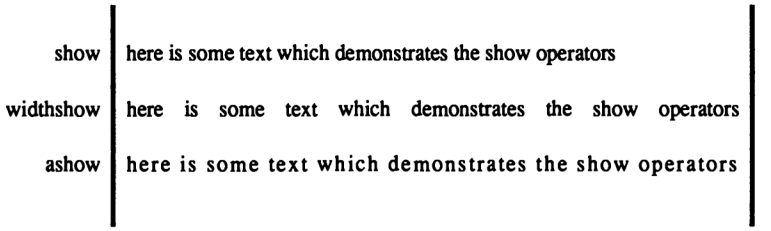

Fig 9. The various forms of the show operator.

Adobe Systems evidently have some ingenious algorithms for making algorithmically generated fonts look reasonably good at low resolutions, though they have not revealed how this is achieved. The Apple LaserWriter output demonstrates that there is room for improvement, however: the tight curves at the bottom of small letters such as 'a' reveal small pimples. Naturally, our implementation cannot approach Adobe's rendering quality. It is also difficult to judge how good a job they do because of the resolution difference between laser printers and bitmapped displays.

4.3. Caching

Drawing every character algorithmically is too slow to be practical, so the POSTSCRIPT language provides elaborate mechanisms for caching character images into bitmaps. Unfortunately the mechanism is partly visible and partly hidden, but the effect is that **BuildChar** needs to know how to set up the cache, or prevent its use.

The model is that the contents of the cache is a mask, which is painted onto the page in the current colour. POSTSCRIPT does not support multi-coloured fonts with caching (though if the caching is not used, the rest of the machinery still works in colour). It is interesting to see how the **imagemask** operator fits into this scenario.

The mechanism for caching characters is somewhat involved: there is an operator called **setcachedevice** which alters the device in the graphics state so that further output goes into a saved bitmap. This operator only works in the context of a **show**. One curiosity of this operator is that it takes an implicit argument which is the character name to cache. This is magically passed from **show** in the graphics state. It cannot be global, as POSTSCRIPT is expected to be able to use other characters in the generation of compound characters such as ligatures, so in our implementation there is a small amount of extra graphics state associated with **show**.

The parameters to the **cachestatus** and **setcachelimit** operators give clues as to how the caching machinery might be organised. There is a doubly linked list of font caches which is kept ordered on a Least Recently Used basis, so that when too many characters have been cached, complete caches can be thrown away to make space.

There is a separate hash table of characters. When this fills up, the least recently used font cache and all associated characters are thrown away. There is also a maximum size beyond which characters are not cached at all (since characters of that size are used rarely enough to make caching them a waste of time and space). When a character is being drawn into the cache, the current colour is ignored.

All this accords with the description in the POSTSCRIPT Language Reference Manual, although some aspects of the Adobe implementation have recently changed.

4.3.1. Disk caching

In order to improve the start-up performance of our previewer, machinery was added for reading cached fonts from disk. These are keyed on the font name and final transformation matrix. It is not possible or even desirable to decide which cache to use until the actual **show** operator is executed, since this is dependent on the graphics state (i.e. current transformation matrix and font). At that point, the local cache is searched; if the font was present at the current transformation and scale, **show** will go on to render characters from the cache. If the cache was not found and the font bounding box is small enough, a new cache is generated. At this stage, the interpreter will attempt to add to the new cache from a disk copy. If it cannot find a disk cache, it simply continues. **show** will then attempt to render characters from the cache; if this fails at some point, it will drop out to perform a call-back into POSTSCRIPT code to execute **BuildChar** from the current font. When that returns, it will attempt to render from the cache again: if **BuildChar** added to the cache, this will succeed and show the character. If **BuildChar** defeated caching and drew the character itself, then **show** will fail to render from the cache and continue.

The nice feature of all this is that **show** is very fast as long as the characters it needs to show are cached. When they are not, it slows up briefly to add to the cache and then runs quickly for all subsequent renderings of that character. Apart from this, it never needs to return to interpreted code. Disk caches may be incomplete or even absent, but the interpreter always makes the best use of the information available.

Caches are not saved to disk automatically: if this were done, the disk would soon fill up with fonts

in strange sizes and orientations. Instead, we have added a new operator, called **savecurrentfont** which writes the current font, as cached so far, to the appropriate disk cache directory.

4.3.2. Font library

Our font descriptions in POSTSCRIPT also do a certain amount of lazy evaluation: the **BuildChar** procedure which is initially loaded is in fact a stub which loads the complete font description when it is called. It then replaces itself by the real **BuildChar**, which does rendering. The advantage of this, is that as long as cached font sizes are used, the large font descriptions need never be loaded.

Even the font descriptions are loaded lazily by **findfont**. The internal **findfont** only knows how to search the *FontDirectory* within the interpreter, so a POSTSCRIPT version is wrapped around it which knows how to search a font library on disk. The first thing that does is to attempt to find the font in **FontDirectory**. If that fails, it goes to disk. An extra directory mapping font names to file names has been added so that fonts can be renamed. POSTSCRIPT documents tend to have names like "/Times-Roman" wired into them, so the ability to map these to the fonts available through a separate mechanism adds much flexibility.

When the interpreter was ported to a machine with a smaller display, an A4 page would not quite fit on the screen, so the default transformation matrix was adjusted to allow the page to be less than actual size. When this occurred, the interpreter automatically picked up slightly smaller cache entries for ordinary font sizes without any adjustment.

5. PORTING EXPERIENCES

Our implementation was designed for high performance, high resolution bitmapped graphical workstations. It was originally written on an ICL Perq 2 running PNX3 and later ported to PNX5 with virtual memory.

It was realised fairly early on that the interpreter would have to be portable, so some effort was expended to ensure that the graphics device dependencies were well isolated. To assist with this, a device interface was designed which could be re-implemented for each new display with minimal effort and size. The current system runs with device modules which each take less than 4% of the total code size. The first implementations used a locally written portable graphics library called ww: the interface for this is about 400 lines of C, the rest of the interpreter is about 10000 lines of C. The ww version runs unmodified on Perq 2s, Sun3s and Whitechapel MG1s. There is no conditional compilation, only an exchangeable object module.

A port was done to the High Level Hardware Orion workstation, which supports gray levels. This was done initially by porting enough of the ww library, and then by using the resident graphics system (when the latter was enhanced to support grayscale/colour graphics). All the half-toning work is done below the device independent interface, so half-toning algorithms were simply replaced by code set brightness levels. The first Orion port took a week.

A version has also been written for the X window manager, although X lacks certain essential features, such as the ability to half-tone a bitmap before drawing it, or perform off-screen RasterOps. X windows does not support general purpose RasterOp functions on or between off-screen bitmaps, so POSTSCRIPT has great difficulty caching fonts on X. There does not appear to be any reasonable way of getting drawn characters into an X font, even if the latter could support the requirements of POSTSCRIPT fonts.

5.1. Colour ports

Although the Orion and X windows both support colour, a full colour port has not been done yet, mostly because we cannot decide how best to deal with colour maps. POSTSCRIPT deals with colour in the abstract: it asks for a particular intensity of Red, Green and Blue or Hue, Saturation and Brightness, without regard to the abilities of the display. A reference to the colour model used is in the POSTSCRIPT language manual[7] . Most colour displays have a colour map which restricts the number of visible colours at any one time depending on the depth of the display. If a POSTSCRIPT program attempts to draw too many colours, the system would run out of colour map entries. This is not a problem on a grayscale

implementation, because the colour space is one-dimensional, so it can be sampled reasonably. It would be possible to sample a colour space in three dimensions, but very sparsely.

An alternative is to half-tone the two closest colours in the colour map to the colour requested when the colour map is full. Unfortunately there may be times when there is nothing even remotely close in the colour map. It might be possible to prime the colour map with anchor points, but this would not work well without multicolour half-tones, for which a solid function is not sufficient. The problem deserves further research.

6. SUMMARY AND CONCLUSIONS

An implementation of POSTSCRIPT has been described, which is complete enough for most usage, and efficient enough to be a practical tool for previewing. The bottle-neck in execution time is in floating point arithmetic; this was amply demonstrated by compiling it on a Sun3 with and without hardware floating point support. We have considered the possibility of using fixed point arithmetic, but haven't tried this yet.

During the development, we encountered aliasing in various guises – thin lines vanishing as they fell between pixels and some fairly unpleasant examples of stair-casing along adjacent curves. This problem appears to be basically unsolvable.

We did a small amount work to make drawn characters look readable at small sizes, but this was not very successful. The only way to get reasonable characters is to buy properly designed raster fonts. We have not done this yet, so we haven't attempted to integrate bitmapped fonts with our caching machinery, though this should be automatic.

We have no clipping support yet, as our area fill is rather simplistic (though correct). Our implementation of the imaging operators is also somewhat simplistic.

Every POSTSCRIPT operator has some implementation, even if it doesn't provide complete functionality. This allows documents to print without failing, even if the output has bits missing when it is completed.

ACKNOWLEDGEMENTS

I would like to thank Tony Williams for his continual support and advice during the development of the interpreter, and also for his helpful comments on this paper.

I would also like to thank Mark Martin for writing the ww graphics package, which got me off the ground and saved me from battles with some bizarre window systems. Also for his comments on the paper.

REFERENCES

1. Adobe Systems Inc., *PostScript Language Reference Manual*, Addison Wesley, Reading, Mass.. ISBN 0-201-10174-2
2. D. G. Bobrow et al., "COMMONLOOPS: merging common lisp and Object-oriented programming.," *Intelligent Systems Laboratory Series*, Xerox, Palo Alto Research Center (1985).
3. Martin E Newell and Carlo H Sequin, "The Inside Story on Self-Intersecting Polygons," *Lambda* 1(2), pp. 20-24 (Second Quarter, 1980).
4. Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass (1984).
5. B. A. Barsky and A. D. DeRose, "The Beta2-spline: A special Case of the Beta-spline Curve and Surface Representation," *IEEE Computer Graphics and Applications* (September 1985).
6. Rob Pike, Leo Guibas, and Dan Ingalls, "Bitmap Graphics," SIGGRAPH 84 Course Notes, AT&T Bell Laboratories (1984).
7. A. R. Smith, "Color Gamut Transform Pairs," *Computer Graphics* 12(3) (August 1978).