Pad Structures for the Rainbow Workstation

B. A. STYNE, T. R. KING AND N. E. WISEMAN*
University of Cambridge, Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG

The Rainbow Workstation is an experimental device built to evaluate a method of supporting windows by dynamically mapping memory to video. The main features of its architecture have been published elsewhere. In this paper a software interface between the workstation and its driving programs is described. The interface provides simple facilities that assist in setting up a virtual terminal system in the workstation and in addition some novel and powerful graphics effects for sophisticated host programs to call upon. The use of lookup table indirections in conjunction with the mapping of memory to video enables some picture manipulations to be achieved very quickly. For example, object silhouettes of any shape can be clipped to one another as quickly as they can be repositioned on the screen. The blending of an anti-aliased image against its background can also be done, as the image moves about, with no time penalty. The basic system design is given in outline, as the background to these techniques, but the method of recomputing the screen is described in more detail.

1. INTRODUCTION

A previous paper[1] describes the architecture and some of the basic systems ideas of the Rainbow Workstation. Only a brief overview of these aspects is presented here to introduce the central theme of this paper, which is the functional interface between program and terminal. The workstation is regarded as a unit of hardware with some microprogrammed functions and a program that works the display, keyboard, pointing and picking devices and the communications network. The program accessing these facilities may be the workstation manager, or some special applications code that is loaded by the workstation to serve a particular need. We have tried to make the interface easy to drive without compromising either the performance or the flexibility of the workstation itself.

The Rainbow Display configuration is shown in outline in Fig. 1. Keyboard, mouse, tablet and display comprise the actual devices which the workstation may present as an assortment of virtual devices to one or a number of host computers on the network. The output streams are directed at logical units of potentially viewable images called 'Pads', and the hierarchy of pads with its associated data comprises the pad structure abstraction described here. All output functions are controlled through the pad structure. We describe first a simplified structure which illustrates the basic idea but which falls short of the thing we have actually implemented. Suppose that each output stream from a host computer is allocated one rectangular pad and that these are arranged on the screen in some way so that some part of each pad is visible (Fig. 2). A single level of hierarchy in the pad structure controls what we see (Fig. 3), where the nodes represent pads and the arcs show their position and relative priority. The screen is itself a pad composed from the cluster of pads which it owns in the pad structure, and the priority and position data on the arcs specify the screen arrangement and how pads obscure one another. By writing different values for these quantities into the pad structure the program can cause the pads to move about on the screen and alter their ordering. A library of procedures is provided for doing this sort of thing, and for building and dismantling pad structures themselves. Thus:

pad: = CN.create (type, xextent, yextent, bits-per-pixel, starting-plane)

will create a pad of specified size in video memory to hold an image.

arc: = CN.insert (pad, owning-pad, xposition, yposition, priority)

will build an arc joining two pads expressing the owning relation and carrying position and priority values.

CN.move (arc, xposition, yposition)

will reposition a pad by changing the appropriate arc values, and so on. (More details are given in section 3.) Each call to such a procedure will modify the pad structure in some respect but the screen is only brought up to date to match it when the call

CN.display ()

is made. Thus any number of structural changes can be amalgamated into a single screen change. The way it works is explained later.

It is a particular feature of the hardware design that reorganizing the screen image should be rapid, whatever the size, disposition and content of its components. Numerous pads, with several (up to 8) bits per pixel in each are allowed, the pads can overlap one another and the screen edges (the screen is itself a pad), and assorted cursors may be moved about by the input tools or by program commands. Pads which overlap can, under certain conditions, interact with (rather than just obscure) one another, giving rise to a variety of new effects.

2. BASIC FACILITIES OF THE DISPLAY

The monitor adopts the UK broadcast TV standard of 625 lines 50 Hz field interlaced. The aspect ratio is 4:3 landscape, and since the standard specifies 576 unblanked lines per frame we therefore need 768 pixels along each line to make them square. The RGB signals are derived from D–A converters fed with 8 bits for each colour from the lookup table (Fig. 4). There are 4096 entries in the look-up table and the 12-bit index is derived from the pixel values in memory by a unit called the Context Unit which augments each pixel by forcing and mixing in up

* Primary author for correspondence.

to 8 context bits with an offset. The context bits are associated with each pad and can change only at pad boundaries. A slice unit aligns the words read from memory with the pad boundaries and pixel number space, thus allowing any rearrangement of planes and bit offsets to be achieved. Two buses are shown in Fig. 4. One of these (rbus) conveys control signals between the different units. The other (obus) carries the time-multiplexed memory words which are en route to the slice unit. This bus cycles every 32 nanoseconds.

As the scan proceeds over the displayed raster, the appropriate areas in graphics memory are selected to position the pads where they are wanted on the screen. This is done by reference to a series of horizontal strips previously computed to cover the screen in such a manner that each strip contains only visible vertical pad edges. This data, known as the 'Band Structure', controls the sequencing of memory reads from graphics memory. The graphics processor plants the control information into the relevant memory registers and then the memory units cycle and step until the next visible pad boundary. Up to around 10 visible boundaries per scan line can be accommodated before the graphics processor fails to keep up with its job of reloading memory registers. Changing the band structure is also a time-critical job which influences the speed of picture update. The amount of work involved varies with the number of bands which change and the number of pads encountered in each. It is done in the M 68000. The construction of images in graphics memory and the maintenance of the pad structure itself are also handled by this processor, although some functions are speeded up by calling on assist procedures in the graphics processor – colouring rectangles for example (there are many opportunities for assist procedures to speed up image generation which we have not yet tried). The M 68000 may also run application-specific programs and offers remote procedure calls to host machines for all essential operations to do with pad structure maintenance and tool operation.

3. THE PAD STRUCTURE ABSTRACTION

The pad structure is a rooted acyclic directed graph. The root represents the screen pad itself (although the structure is such that it would be possible to associate the screen node with any node in the system). Nodes are pads which are potentially viewable rectangles of picture data. A leafnode (terminal pad) is a raster of data in graphics memory with a specified number of bits per pixel (from 1 to 8). A clusternode (non-terminal pad) represents a collection of transformed nodes clipped to the pad boundary. Associated with each leafnode is an area of look-up table with sufficient entries to define the rendering of each pixel value. For example a 2-bit deep raster would be entered in a leafnode carrying a 4-entry look-up table. It is also possible to create a leafnode with 0 bit per pixel. Such a node cannot be written to or read explicitly, but can serve as an area of uniform colour for backgrounds. When a pad is constructed, by the procedure CN.create quoted in section 2, its type is specified to be leaf or non-terminal. For a leafnode the number of bits per pixel and a memory allocation strategy must also be given so that an appropriate area in memory can be reserved for the raster attached to this leaf. For a non-terminal, there is no memory allocation to be done, so these parameters are not needed.

A leafnode requires an associated area in look-up table to be allocated and filled with the colour definitions for each pixel value in the raster on that leaf. This is achieved with the procedure

CN.setLUregion (leafnode, region),

where region is a bit of lookup table previously allocated and filled, or ready to be filled, with the colour values needed. A fairly generous amount of lookup table (4096×24) is available for allocation, so it is usually possible to recolour pads independently of one another by giving each pad its own region.

Arcs are used to collect nodes into clusters, the parent node being a non-terminal which clips all its progeny to its own boundary, and each offspring is either a terminal or non-terminal node whose data is mapped by the attributes of the joining arc. These attributes are an X & Y offset, a relative priority and an on/off condition. The offset repositions the offspring with respect to the parent, the priority determines how the different offspring obscure one another in the parent pad and the on/off switch enables or disables the arc. The CN.insert procedure builds an arc of this sort and sets its switch on.

3.1. Pad visibility

It will normally be the case that arcs leaving a given parent will carry different values for their priority attributes. The software will than arrange for the hardware to read only the visible parts of the relevant rasters while making video, selecting the data which belongs to the arc of higher priority when overlap occurs. However, provided the rasters are held in disjoint planes of memory, two (or more) arcs may carry the same priority, and in this case the hardware will read from both (all) of the relevant rasters in the overlapping areas. The pixel values from the constituent rasters are combined and used to index an additional look-up table, unique to that particular overlap. If, say, three pads A, B and C have arcs of equal priority in some common parent, then a separate look-up table can be set up for A–B overlap, A–C overlap, B–C overlap, and A–B–C overlap (as well of course as for the three pads alone). This gives a comprehensive facility for defining the effects desired in the areas of overlap. We can make pads seem transparent to one another, one can recolour another, or even behave as if in front of another for some pixel values, and behind it for other pixel values. For want of a better word, we refer to all of these as 'transparent' pads. As pads move over one another, transparency causes the well-known effects[9] obtainable from look-up table indirections to be given a new lease of life. For example, clipping arbitrary polygons against one another can be achieved as quickly as pads can move – much faster than the clipped result could be computed by geometry. It is even possible to do away with the z-buffer for handling visibility in three dimensions, putting instead the obscuring rule in the look-up table. However, the size of this table (4096 entries) on the Rainbow Display precludes this use for anything other than demonstration-sized pictures.

The pad structure as described allows many different effects to be produced. Consider Fig. 5, which illustrates

a form of schizophrenic pad ordering. Two non-terminal pads each show translated and clipped views of the rasters in pads C and D. However, D obscures C in one case, whereas D is obscured by C in the other.

Fig. 6 shows how pads may interfere. Suppose we set up look-up tables for the two pads as follows:

$\sim A \;\&\; \sim B \Rightarrow 0,$
$\sim A \;\&\; B \Rightarrow Pb,$
$A \;\&\; \sim B \Rightarrow Pa,$
$A \;\&\; B \Rightarrow f(Pa, Pb),$

where Pa, Pb refer to pixel values in pads A and B. In this case f(Pa, Pb) is some combinational function of the pixel values in A, B. Since the look-up table gives us an entry for each minterm of the input variables (if there are m, n bits per pixel in A, B respectively then the look-up table will be of 2^{m+n} entries), every possible interaction between A and B can be specified. In particular, suppose that A contains an image anti-aliased against its (null) background, and that A is to be moved over B with its soft edges correctly blended with the differing colours in B. All we have to do is set f(Pa, Pb) to be the required blending function, such as:

$$A \;\&\; B \Rightarrow FPa + (1 - F)Pb$$

(F could, for example, be the pixel value in A normalized to cover the range 0–1). This idea seems rather good because the anti-aliasing computation is done just once and the blending is automatic (and therefore immediate).

Using the same idea we can arrange to support pads of non-rectangular boundary. Consider Fig. 7, where A and B represent the rasters of data to be displayed and I is a mask pad fully overlapping with A and containing a one-bit-deep raster carrying a filled polygon of the shape desired. The lookup tables are organized thus:

$\sim A \;\&\; \sim B \Rightarrow 0$
$\sim A \;\&\; B \Rightarrow Pb$
$A \;\&\; \sim B \Rightarrow Pi = 0 -> 0,\ Pa$
$A \;\&\; B \Rightarrow Pi = 0 -> Pb,\ Zb > Za -> Pb,\ Pa$

The values Za and Zb refer to the desired priorities for the masked pad A and pad B. Note that in this case they are not the values carried by arc attributes in the pad structure, but are used to specify the lookup table contents. Note also that the pad A and its mask pad I will always totally overlap (they will be moved about in unison), so there is no need to specify lookup tables for $A \;\&\; \sim I$ or $\sim A \;\&\; I$. The effect achieved is of a pad with arbitrary shape – that of the mask I. Of course the mask can be several disjoint polygons, or an inverted thing which cuts holes of an arbitrary shape in the underlying raster A.

4. PAD STRUCTURE MAINTENANCE

A variety of cluster manipulation procedures are provided by the interface for the purpose of reconfiguring the pad structure so that nodes of the structure may be moved about, resized or reconstructed by a client program. The processing involved in taking a pad structure update into a corresponding screen update is described below and consists of three operations. The first operation is a recompilation of the 'source band structure' which is a data structure maintained in the M 68000 main memory and which essentially is a list of the bands that are present in the picture. The second operation is the generation from this list of a new 'object band structure' which is in a convenient form for the display processor to interpret and which is constructed in the graphics memory of the display processor itself. The third operation is the display processor interpreting the new band structure.

4.1. Recompilation of the source band structure

Once a cluster manipulation procedure has changed the pad structure it must cause the source band structure to be altered accordingly. It does this by considering the picture update to be a sequence of simple operations on single pads of the type insert, remove, move, and recalculate context value. Since the response time of the screen update is to a large extent dependent on the speed with which this can be done, efficiency of code is important.

The basic philosophy which was adopted is that, since bands are represented as linked lists which are computationally quite expensive objects to construct and that, since a typical screen update may well only affect a minority of the bands present, then bands should be allowed to persist until directly affected by a screen update. Moreover, a band should be re-used whenever possible such as, for example, when it expands or shrinks or is split up. A table of pointers is maintained which maps the top and bottom lines of clusters on the screen into the linked list which represents the bands they fall across so that it is a simple matter, given a pad to be inserted or removed, to find the bands which need to be adjusted.

4.2. Regeneration of the object band structure

All the structural changes having been completed, a call to CN.Display is made in order to cause the new picture to be put on the screen. This procedure scans the source band structure and produces as output a list of new band descriptors, each band descriptor itself consisting of a sub-list of rectangle descriptors.

A rectangle descriptor defines a rectangular area of screen upon which is shown an image formed by the parallel composition of up to eight planes of contiguous bit-map. It holds pointers to the areas of graphics memory contributing to this rectangle and a value representing the number of bits by which the data extracted from these areas of memory should be offset, with respect to their word-alignment in memory, in order for the image correctly to be aligned on the screen. Information relating the physical location of a memory plane in which an image resides with the logical bit-position in a pixel for which it is responsible is also stored here, as is a value, known as the context word, which augments the pixel values for this rectangle in a particular way. A field representing the width of the rectangle in pixels completes the rectangle description.

A band descriptor is completed by having two more fields, one saying how many scan lines high the band is and one saying how many rectangles it contains. Two lists of this kind are maintained in the graphics memory to double-buffer the updates. One is the band structure

currently being interpreted by the display processor and the other is the new band structure currently being generated by the CN.Display() procedure. When construction is complete CN.Display() updates a band structure root pointer so that it refers to the new structure.

4.3. Interpretation of the object band structure

During normal operation the display processor is running one of eight possible tasks as described in ref. 1. Two such are the frame flyback and video tasks. The frame flyback task runs once at the end of each frame time and takes a copy of the band structure root pointer. When the video task is awakened at the start of the next frame it uses this copy to find the correct band structure to interpret.

When the first band is found its height is copied and thereafter, at the start of every subsequent line of video shown, the copy is decremented until it reaches zero, when it is time to look for the next band. If the last line of the band is the last of an even field then no more rectangles are interpreted until the next frame time.

At the start of each scan line within a band the first rectangle descriptor for that band is read out, the control information is dispatched to the display hardware and the width information used to determine for how long the video for this rectangle should be generated until the next descriptor is read out. When the last rectangle descriptor for this band has been interpreted in this way the video task suspends itself until the next line time.

5. PERFORMANCE

To attempt a quantitative comparison of the display dynamics with that of other systems is very difficult. Some screen images favour the Rainbow hardware and appear very responsive – these are not the same images which would work best on a different architecture. It seemed to us that a single set of tests should be reported to give an impression (no more) of what the system can do, and to rely on the reader's judgement of how the speed, flexibility and power balance has turned out.

The band structure computation tends to dominate the display update time, and this is simpler for fewer numbers of pads, however big or deep they may be. Using a raster-op (bitblt) approach the bigger deeper pads would take the longer time, so our best case for comparative performance is one large pad of 8-bit-deep pixels moved over a blank screen. Our worst case for comparison is, conversely, moving a small pad through a sea of other small pads, for which a raster-op method would be better. In Table 1 we compare the time in field ticks (fiftieths of a second) to move a pad of the stated size along an oblique line stretching over the whole screen height (576 pixels) in single pixel steps. The screen also carries an array of the stated number of stationary pads of a similar size spaced along a screen diameter. All pads are square and 8 bits deep.

The equivalent raster-op would in the most favourable (for us) comparison require some 100 Mbytes per second of data movement in memory. In the test there was no waiting for a frame tick interrupt before starting the recomputation, so all figures less than 576 would, in practice, be replaced by 576. One does not update the screen twice in a single field!

Table 1

	Pad size		
No. of pads	50	200	500
1	165	164	149
2	181	194	174
3	222	306	313
6	312	577	653

The remaining illustrations (Figs 8–14) show examples of the effects of 'transparency' and give some idea of the image quality. The use of anti-aliasing (not shown) improves the visibility of fine detail and makes for a more restful and steady picture. Many visitors are surprised that the standard TV line rates and resolution can look so good.

6. CONCLUSIONS

The basic facilities needed to establish a comprehensive workstation function are complete. Working these facilities into a production quality suite of programs is presently under way, and will, it is hoped, form the basis for a further publishable paper. The intention is to form links between host and terminal through remote procedure call and remote tasking, and a version of the library for supporting the pad structure remotely in this way is already beginning to work.

The Rainbow Workstation was designed to experiment with the particular method described for mapping pads in memory to their screen positions. It is not well equipped to generate image data in pads very rapidly and for general high performance it perhaps should have both bitblt and mapping hardware. The earlier paper about it[1] is more specific on these issues. Nevertheless, the equipment is able to function as a workstation of unusual power and flexibility and, through collaboration with our colleagues in industry, we expect that improved versions will be built which retain the major features of the design.

Acknowledgements

Besides our colleagues in the department, we would like to record thanks to David Otway and Arthur Foster at GEC Hirst Research Centre for their support and GEC itself for assistance with the cost of colour printing the illustrations for this paper and for financial support for the continued development of the project. One of us (BAS) was supported by the ICL Research Studentship and a grant from the Committee of Vice Chancellors & Principals.

REFERENCES

1. A. J. Wilkes, D. W. Singer, J. . Gibbons, T. R. King, P. Robinson and N. E. Wiseman, The Rainbow Workstation. *The Computer Journal* **27**, 2 (1984).
2. C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull and D. R. Boggs, *Alto: a personal computer. In Computer Structures: Readings and Examples* (edited O. Siewiorek, G. Bell and A. Newell). McGraw-Hill, New York (2nd edition 1981).
3. *PERQ – A Landmark Computer System.* Three Rivers Computer Corporation, Pittsburgh, Pa. (1979).
4. *Apollo Domain Architecture.* Technical Report, Apollo Computer Inc., Chelmsford, Mass. (1981).
5. A. Bechtolsheim and F. Basket, High-performance raster graphics for microcomputer systems, *Proceedings SIGGRAPH 80* . In *Computer Graphics* **14**, 3 (1980).
6. Xerox's Star, *The Seybold Report* **10**, 16 (1981).
7. I. Page and A. Walsby, Highly dynamic text display system. *Microprocessors and Microsystems* **3**, 2, 73–76 (1979).
8. R. Pike, Graphics in overlapping bitmap layers. *ACM Trans. Graphics* **2**, 2 (1983).
9. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, London (1982).