EUROGRAPHICS '90 / C.E. Vandoni and D.A. Duce (Editors)
Elsevier Science Publishers B.V. (North-Holland)

A PIPELINED-PARALLEL ARCHITECTURE FOR 2.5-D BATCH RASTERIZERS

Yitzhak BIRK and James M. MCCROSSIN

IBM Research Division
Almaden Research Center
650 Harry Rd.
San Jose, CA 95120-6099, U.S.A.
(birk@ibm.com, mccrosn@ibm.com)

The emergence of application programs that take advantage of highly expressive page description languages has sharply increased the amount of computing required for rasterizing an average page, and single-microprocessor rasterizers presently limit the performance of most printers.

The pipelined-parallel architecture employs intrapage parallelism to permit the construction of cost-effective multiprocessor rasterizers for computer-driven high-function printers. Initially, blocks of datastream that are independent in terms of datastream environment are identified by a sequential "scanner". They are then processed in parallel, and each is converted into a multitude of simple, regular objects, which are sorted by "geographical" target on the page into "bins" that correspond to a predetermined partition of the page. Sequencing information is retained. The bins are then processed in parallel (sequentially within each bin) to build the full-page bitmap. The phases are pipelined for increased performance. By breaking rasterization into two main stages and parallelizing along a different dimension in each of them, we are able to attain intrapage parallelism while maintaining correctness, even with non commutative merging modes, such as "overpaint".

1. INTRODUCTION

1.1. 2.5-D Rasterization for Printers

We use the term *rasterization* to refer to the process of converting a high-level description of a scene into an array of numbers which represent the color and intensity of every picture element (pel) in the projection of this scene onto a plane. Our focus in this paper is on rasterization for printers. We refer to the input form of the description of page contents as the *datastream* and to the output as a *pagemap*. The datastream is expressed in a page-description language, or *PDL*.

Rasterization for printing differs from that for 3-D graphics, such as in graphics workstations. Following are some of the salient differences.

- The printer datastream describes 2-D objects which lie in planes parallel to that of the projection. Moreover, there is a "distance"-ordering of those objects, which is simply the order of their appearance in the datastream (farthest first). Visibility is determined implicitly by projecting the objects onto the plane in datastream order, and using specified *merging modes* to determine the new value of a pel as a function of its old value and the new object. (In "overpaint" mode, for example, a "later" object always hides an "earlier" one.) In 3-D graphics, the objects are three dimensional. Visibility is determined from distance information, which is an integral part of an object's description. For example, a sphere of radius 2 with its center at (x=5, y=6, z=8) would not be visible if there were also a sphere of radius 2.5 with its center at (5,3,2). Furthermore, two objects may hide parts of each other, so the determination must be made at a fine (intra-object) granularity. Since the objects contain the distance information, the order in which they are rasterized and applied to the pagemap is not important.

- Rasterization for printing is invariably a batch job, in the sense that the scene is constructed from scratch, whereas 3-D rasterizers are frequently engaged in incremental rasterization.
- Printer pagemaps are usually dense and shallow, almost always consisting of a single bit per pel. Displays for 3-D graphics have far fewer pels but multiple bits per pel.

In the remainder of the paper, we limit the discussion to rasterization for printing. We refer to this as 2.5-D rasterization, to reflect the fact that, unlike "pure" 2-D rasterization, the order of the objects represents part of a third dimension, and must therefore be correctly reflected in the final appearance of the page.

1.2. The "High-Function" Rasterization Problem

The migration to all-points-addressable printing in conjunction with "high-function" PDLs has brought about a dramatic increase in the amount of processing required for the rasterization of a page. A rasterizer may now be required to process fonts whose characters are described by curves representing their outlines; the characters must be scaled to arbitrary sizes and rotated at arbitrary angles. It may also be required to scale graphical objects, rotate them, and convert them to pelmaps, as well as to scale and rotate images. Lastly, masks may be used to limit the scope of an object (clipping), and various "merging modes" may be used to specify the result of placing overlapping objects. The introduction of color printers is increasing the processing requirements even further. To keep the cost of the rasterizer manageable, as well as for other reasons such as incremental growth capability, it is highly desirable to use several inexpensive processors rather than a single, very expensive one.

In going about architecting a multiprocessor rasterizer, one faces the following challenges:

- Attain a significant degree of parallelism in order to achieve a high aggregate processing rate.
- Minimize overhead and duplication of computation effort, since these increase the total amount of work, partly offsetting the increase in performance.
- Minimize the sensitivity of processor utilization to page content (e.g., to variability in page complexity within a document).
- Minimize the amount of memory required. Ideally, a k-processor rasterizer should require substantially less memory than k uniprocessor rasterizers.

In order for two *blocks* of datastream (the term "block" is used loosely) to be rasterized concurrently, two types of conditions must be satisfied:

1. **State-independence.** At the beginning of each block, the *datastream processing environment* (e.g., current font, location, coordinate system, etc.) must be known.
2. **Geographic independence.** The target regions of the blocks in the pagemap must not overlap. (With any single commutative merging mode, such as "OR", this is not required. However, it is required for "Overpaint" and even for combinations of merging modes which are individually commutative and associative. For example, suppose that a given pel is initialized to 0 and is then struck with the following two (value, merging mode) pairs: (0,AND),(1,OR). The result clearly depends on the order. We therefore refrain from relying on merging modes.)

While the need to satisfy (1) alone often results in a sufficient number of independent blocks, having to also satisfy (2) can be very restrictive. Furthermore, determining at the outset whether (2) is satisfied is generally very difficult, and the amount of processing required can be comparable to the entire rasterization.

1.3. Existing Multiprocessor Rasterizer Architectures

Several approaches have been taken to date:

- **Pipeline.** The rasterization is broken into stages and those are pipelined. The number of stages is inherently small (2 or 3), and it is difficult to balance the load. As a result, the speedup factor is usually smaller than 2. This approach has no correctness problems, since the datastream is processed sequentially in any given stage.
- **"Functional" Parallelism.** Blocks of different types (e.g., image, text, graphics) are rasterized independently, and the results are merged sequentially into the pagemap. This can only offer moderate parallelism due to the limited number of types; the effective parallelism achieved in a given page is very sensitive to the relative processing loads for the different types and hence to the content of the page.
- **"Geographic" Parallelism.** This represents the intuition that it makes sense to build different regions of a page in parallel. However, the perceived need to satisfy (2) at the outset has prevented the realization of this idea.
- **Page-Parallel** [1]. Each processor works on a different page. With this architecture, requirement (1) is usually easy to satisfy, (2) is always satisfied, and a high degree of parallelism is achievable. However, very large amounts of memory are required, and the rasterization time of a single page is not reduced. Also, processor utilization drops significantly whenever the page-rasterization time is highly variable unless much more memory is used.
- **"Raster Processing Machine" (RPM).** [4] This architecture was developed for electrostatic plotters, which do not have a full pagemap. Incoming graphical objects are converted sequentially into an internal format and are then sorted by geographic location into bins corresponding to the size of the raster buffers of the machine (bands of pagemap). Once this process is completed, the bins are processed, possibly in parallel, and the results are placed into the appropriate partial pagemaps. The processing of the internal format to create raster patterns is carried out by several different chips, depending on the type of object, and the results are ORed into the raster buffer. Here, only part of the process is parallelized, and correct sequencing of overlapping objects is not achieved.
- **3-D Rasterizers with "Z-buffers".** These are exemplified by the "Superdisplays" proposed by Pavicic [2]. Object processors rasterize objects and feed the output to image processors (smart memory). Each sample from location (x,y) is fed to the (x,y) processor. The sample includes an intensity "I" and a "z" value (distance). Whenever an image processor receives a new (x,y,z,I) vector, it compares the z value with the one it is currently holding. If the new z is larger than the old one (farther away), it discards the new entry. Otherwise, it replaces the old one with the new one. It is possible to retain several values to accommodate a more general case wherein an entry only partly covers a pel. This architecture is aimed at displays, where a typical scenario is a list of independent 3-D objects which undergo incremental changes (addition or deletion of objects or changes to objects). It relies heavily on the notion of overpainting of a distant object by a near one, and would not lend itself to more complicated operations in which the resulting intensity and color are arbitrary functions of the past one and new ones. To do so, one would need a possibly infinite list of past (z,I) values for every pel (Z-buffer), since there is no notion of completion of rasterization for a given distance range. The notions of finding the state-independent objects and of resource preparation are entirely absent.

The remainder of the paper is organized as follows. In section 2, we describe the pipelined-parallel architecture, which is the main contribution of this paper. In section 3, we compare it with the foregoing approaches. Section 4 summarizes the contributions of the paper.

2. THE PIPELINED-PARALLEL ARCHITECTURE (PIPAR)

2.1. Overview

While *state-independence* (1) and *geographic-independence* (2) must both be satisfied at the outset in order to permit two blocks of datastream to be rasterized in parallel, we observe that this is strictly necessary only when rasterization is a single-stage process. The true underlying requirements are:

- State-independence (1) suffices as long as rasterization results are not merged, i.e., as long as potentially-overlapping objects are kept in separate memory locations along with sequencing information.
- Objects covering any given location in the pagemap must be merged into that location in the order of their appearance in the original datastream.

To take advantage of these less stringent requirements, we split rasterization into several stages. Initially, state-independent datastream blocks are identified by a sequential *scanner*. These are converted in parallel into intermediate form without merging the results; each block is converted to a multitude of objects. The intermediate-form objects, along with sequencing information, are placed into "geographical" bins, which are then processed in parallel and merged into the pagemap. The need to satisfy the geographic-independence requirement (2) is thus deferred until easily-sortable intermediate form is available, and is applied to small objects rather than to entire datastream blocks. The combination of pipelined stages with intrastage parallelism is expected to result in a substantial speedup relative to a single processor.

2.2. Detailed Outline of the Rasterization

- **Scanner.** Receives the datastream, and outputs it partitioned into state-independent blocks along with state information on block boundaries and a block sequence number for each block. The scanner is sequential and PDL-dependent.
- **Conversion into intermediate-form objects.** In this stage, the state-independent datastream blocks are converted into an intermediate form representation. This entails parsing and interpretation of the datastream as well as the execution of the bulk of the complicated graphics operations, such as rotation, scaling and most of the rendering. However, the pagemap is not touched. Conversion is carried out in parallel for multiple blocks. The intermediate-form objects generated from any given datastream block are generated sequentially by a single processor and placed sequentially in memory. Therefore, it is sufficient to provide the block sequence number once per block. Fig. 1 depicts this stage, as well as the scanner. Letters denote datastream blocks, and numbers denote objects within a block. The input is clearly PDL-specific, as is the required processing. However, the intermediate form can be PDL-independent.

  The processors in this stage also identify references to reusable resources, such as text characters. In such cases, they check to see whether the resources have been prepared (i.e., whether they are available as intermediate-form objects). If not, a preparation request is generated. In either case, a reference to the desired resource is included as an intermediate-form object. Also, a reference count is updated so that the prepared version of the resource is not deleted prematurely.
- **Sorting.** In this stage, depicted in Fig. 2, the intermediate-form objects are sorted into bins that correspond to a geographic partition of the page being built. This stage does not depend on the input PDL. The partition of the page is determined before the sorting begins. The sorting is carried out in parallel for different blocks (potentially on different processors) but sequentially within a block.

  Before commencing the sorting, the processor acquires an allocation of R chunks of memory, one per region. It then places the block sequence number at the beginning of every chunk. As the objects are being sorted, they are appended to one another in the appropriate chunks. When the sorting of the entire block is completed, the processor places an "end-of-block" (#) token at the end of every list. The inclusion of the end-of-block tokens is critical, as will be explained later.

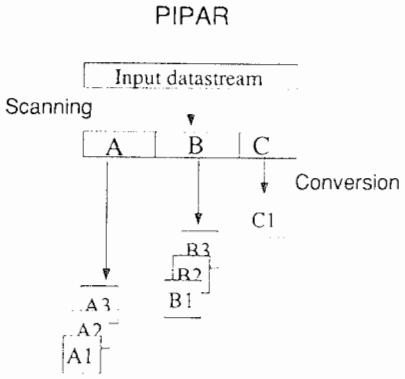

Figure 1: The sequential Scanning stage and the block-parallel Conversion stage in PIPAR. In the example, a single page is partitioned into 3 state-independent blocks, which are then converted in parallel into simple objects.

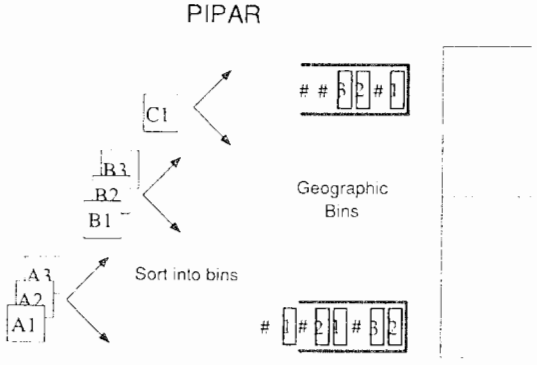

Figure 2: The block-parallel Sorting stage in PIPAR. Note the end-of block (#) markers. The ordering of the blocks within bins is achieved by using a skeletal queue.

An object may span region boundaries. In such a case, it is included in the object lists of all the impacted regions.

The choice of region shapes and intermediate-form object types would be made to facilitate the sorting. For example, the regions may be chosen to be horizontal bands of the page, and the intermediate-form objects may be restricted to be either trapezoids or rectangles whose bases are parallel to the bands, as well as pointers to such objects and to text characters.

The sorting stage is thus a pivoting stage, which permits the transition from the block-parallel conversion stage to the geographically-parallel building of the pagemap. It is important to observe that order information is preserved.

- **Building the pagemap ("Merging").** In this stage, depicted in Fig. 3, the intermediate-form objects are converted to pelmaps (collections of color values of individual pels) and merged into the pagemap, thereby completing the rasterization process. Unlike the previous stages, parallelism here is geographic, as the different bins are processed in parallel by different processors.

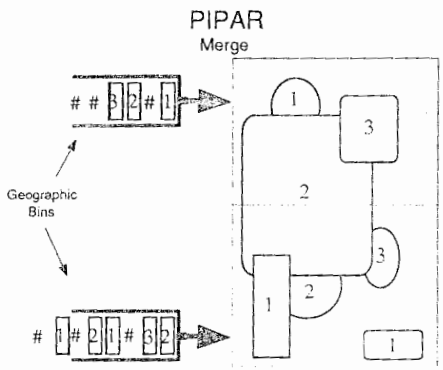

Figure 3: The geographically-parallel Merging stage in PIPAR, illustrated with the overpaint merging mode. The blocks to which the pagemap objects belong can be deduced from the bin contents. Note that correct order is preserved.

In this stage, each processor is assigned a bin and begins processing its content. The processor is the only one that ever updates its region of the pagemap. It processes the objects in its bin in sequential order. In other words, it first processes all objects in the bin that were generated from datastream block No. 1, then those from No. 2, etc. In any given bin, the correct order of objects within a block is maintained by the sequential placement of those objects into memory. The order among objects that were generated from different blocks (in parallel) is maintained by the block sequence numbers. Since a bin processor must finish all block-1 objects in its bin before starting work on block-2 objects, and since the objects are generated in parallel with arbitrary delay and length, the bin processor must be notified explicitly by the sorting processor that all objects for block #1 have been generated. This is done using the aforementioned end-of-block tokens.

Whenever a bin processor encounters a reference to a resource, it checks to see whether the resource is ready. If it is not, the processor must wait for it. After using it, the processor decrements the aforementioned reference count to the resource, so that unreferenced resources can be deleted from memory when it is full.

- **Resource preparation.** This is an auxiliary task, which assists both in balancing the load on processors and in avoiding duplication of work. The resource-preparation requests are entered into a common queue; whenever a processor can contribute to this task, it simply takes the request at the head of the queue. A prepared resource may be represented in intermediate form or as a pelmap (the latter used for small text characters). The locations of prepared (shared) resources must be known to all processors.

The various stages can be pipelined and the data in the pipe can even stretch across page boundaries. Stretching the merging stage across page boundaries would, however, require multiple physical pagemaps.

It can readily be seen that PIPAR rasterizes correctly: state-independence is a condition for beginning to work on a block; the placement of objects in bins preserves the order information; finally, the fact that the bins represent a geographical partition of the pagemap in conjunction with the sequential processing and clipping within each bin in the merging stage guarantees that the true output-independence requirement is satisfied.

The bulk of the rasterization process can thus be parallelized within a page. By substituting the requirement that *intermediate-form objects* be placed into any given region of the page in the correct order for the (unnecessarily) stronger requirement that overlapping *datastream blocks* be placed into the pagemap

sequentially, we both increase the degree of parallelism and facilitate its exploitation. Thus, the proposed architecture does not merely alleviate the problem of determining whether (2) is satisfied; it actually permits parallelism even in cases wherein (1) and (2) are not satisfied by the original datastream blocks.

2.3. Possible Degree of Parallelism

- The scanning, or at least parts of it, is inherently sequential; while this is the eventual limiting factor, the scanner's job can be kept to a minimum so that other factors limit the parallelism. It is also important to note that the ease of discovering state-independent blocks depends on the page description language; a properly designed language and this architecture can thus leverage each other. Although the scanner is primarily sequential, it can be parallelized by making it hierarchical. For example, a sequential scanner would perform a very "shallow" scan, resulting in a single block per page. Each page would then be handed over to a separate scanning processor for a deeper scan.
- In the conversion stage, the potential for parallelism within a page is limited by the datastream content (a single spiral drawn on a page may be difficult to parallelize), by the number of processors, and by the speed of the scanner relative to that of a conversion processor; the datastream limitation can be alleviated by cross-page pipelining, provided that sufficient intermediate-form memory exists.
- The degree of parallelism in sorting is at least as high as in the conversion stage and is not expected to ever constitute a bottleneck.
- The degree of parallelism in the merging stage cannot exceed the lesser of the number of bins and the number of processors, but is not expected to be a problem.
- As has already been stated, the various stages can be pipelined, thereby increasing the effective parallelism.

The overall degree of parallelism depends on many factors, such as the PDL and the page content, the number of pages over which we are willing to stretch the pipeline, and the hardware flexibility. The latter is maximized if every processor can work on any stage.

2.4. Implementation Notes

In this section we present a more detailed discussion of several issues, along with implementation-related comments.

- **Depth of scanning.** There is a trade-off between the amount of processing devoted to scanning and the resultant block sizes. A minimal scanner, for example, would simply look for page boundaries. When this is done by choice, the implementation can be somewhat simpler due to the fact that all the objects for any given page are generated by a single processor (i.e. in the correct order), and each bin is thus filled by a single source. The end-of-block tokens and the associated synchronization mechanism are thus no longer required. Yet multiple processors can still work on the same page in the pagemap-building stage. At the other extreme, the scanner would generate many blocks per page. The optimal "depth" of the scanning depends on the PDL as well as on the number of processors in the system.
- **Combining conversion with sorting.** If position information is available to the conversion processor, it may be able to do the sorting at very little additional cost. It may thus be desirable to combine the two stages.
- **Sorting of multi-region objects.** Whenever an intermediate-form object crosses bin (geographical region) boundaries, it can be split by the sorter which will send each piece to the appropriate bin; alternatively, the sorter will send the entire object to all bins representing pagemap regions which will contain a piece of the object. In the latter case, the merging processor will then clip the object to the appropriate region. Since only a small fraction of the objects will typically cross region boundaries, the replication of effort in the merging stage is not expected to degrade performance in a meaningful way.

- **Considerations in page partitioning.** A page will most likely be partitioned into bands that are parallel to the scan lines of the marking engine. They can then easily be made to correspond to swathe buffers whenever appropriate. Having more bins (regions) than processors helps in balancing the load on the processors whenever the processing requirements for merging are skewed geographically. While the resulting increase in effective parallelism must be weighed against an increase in overhead, we expect that having more bins than merging processors (roughly twice as many) will be beneficial if the numbers are such that only a small fraction of objects cross region boundaries. Finally, regions need not be of equal size; for example, a region containing the top margin may be bigger than others.
- **Hardware configuration.** Using multiple identical processors with shared memory offers the most flexibility and minimizes the sensitivity of processor utilization to page content. However, this minimizes communication bandwidth and does not take advantage of dedicated hardware for specific tasks. For example, if one uses "run ends" as the intermediate form, SLAM (scan-line access memory) chips [5] are very attractive for implementing the pagemap-building stage. We feel that this should be an implementation-specific decision.
- **Data structures.** The figures presented earlier hide the complexity of the data structure required to maintain the sorted intermediate form. This complexity stems from the fact that any given bin is filled in parallel but is emptied sequentially in a particular order. To obviate the need for fairly expensive updates to the queuing structures, we take advantages of the following:

  - The sequential scanner sees the datastream before any other stage, and it knows the delimiters of the state-independent blocks because it determines them.
  - All the objects generated for any given block are generated by a single processor in datastream order.

  The Scanner constructs a skeletal queue structure, which (for a single page) consists of a collection of Region queues. Whenever it detects a new block, it creates an entry for it in every Region queue, and includes a pointer to each of those locations when it passes the block to a processor for Conversion. A processor that converts a block to intermediate form and sorts the resulting objects builds R lists of objects, one per Region, and then inserts into each region queue (at the location handed to it by the Scanner) a pointer to the appropriate list. A merging processor later processes a single skeletal region queue in FIFO order, and follows the pointers to the actual object lists.

3. COMPARISON WITH OTHER ARCHITECTURES

- **Pipeline.** This can be viewed as a degenerate case of PIPAR.
- **Functional.** PIPAR is orthogonal to the functional approach. In other words, one could implement each of the Conversion processors as a collection of function-specific processors.
- **Page-parallel.** PIPAR is more complicated than page-parallel. The added complexity is primarily in two places: (i) the scanner must dig deeper into the datastream to discover block boundaries, and (ii) intermediate form objects must be shipped between processors. (In page-parallel, one could have a single processor rasterize a given page in its entirety.) On the positive side, PIPAR offers some important advantages:

  - Since several processors are working on the same page, substantially less memory is required (per processor) for pagemaps. The benefit will become even more pronounced with the expected increase in the number of bits per pagemap (increased resolution, color, grayscale).
  - With the page-parallel architecture, a processor working on a difficult page cannot be assisted by other processors. In an extreme situation (highly variable page complexity), an N-processor page-parallel rasterizer may have processor utilization of roughly $1/N$ (i.e., effectively only one active processor). PIPAR addresses this problem by allowing all processors to collaborate on the same page, so all processors are always working.

- **RPM.** Both architectures consist of two primary phases, namely a conversion from some external content description to an internal form followed by the conversion of that form into raster. Furthermore, the intermediate form is sorted into geographical bins as it is being generated. (A similar approach has been taken in other rasterizers (e.g., [3]) whose purpose is to permit arbitrary positioning of objects on a page without requiring a full pagemap and without stopping the mechanism in the midst of a page.) However, there are several important differences between PIPAR and RPM.

  - RPM has no parallelism in the parsing phase, which is a combination of scanning and conversion. That phase is consequently its performance bottleneck. The authors in [4] do mention an unsuccessful attempt to parallelize the parsing, apparently by splitting the function between tightly coupled processors.

    PIPAR offers parallel execution of the conversion phase, so only the scanning is sequential. With high-function PDL, we expect the amount of computing required for conversion to far exceed that for scanning. The system will thus be balanced when it matters! The skeletal queue structure constructed by the scanner permits a very simple implementation of a queue that is filled in a different order than the one in which it is emptied. This sharply reduces the penalty overhead of splitting the Scanner from the Conversion and parallelizing the latter. (The different processors are very loosely coupled.)
  - In the RPM with parallel execution of the Merging phase, unlike in PIPAR, different objects are placed into different copies of the same geographical regions. The data from the various copies are then ORed bit by bit on the way to the marking engine. Consequently, the order in which multiple objects affect a given pel does not reflect their relative locations in the datastream. Also, it appears that the outputs of the chips that rasterize different types of elementary objects are merged into the pagemap in arbitrary order.
  - RPM does not address the issue of resource preparation and management, which is key to efficient operation for high-function printing.

  In summary, while the two architectures appear similar, there are important differences. The most pronounced difference is the fact that RPM does the computationally intensive Conversion sequentially and does not guarantee merging into the pagemap in datastream order.
- **3-D Rasterizers with "Z-buffers".** Both approaches employ two main rasterization stages, each carried out by a group of processors, but there are major differences. While the 3-D rasterizers could usually be used for 2.5-D with only minor changes, this would be highly inefficient. For example, printers typically have 1-4 bits per pel in the pagemap. Adding some 10 bits for "distance" information would constitute intolerable overhead. Also, applying the relative distance test to individual pels would constitute an intolerable processing overhead. Finally, the 3-D approach cannot handle combinations of merging modes without requiring unbounded amounts of memory per pel. This is due to the fact that when multiple merging modes are employed, they must be applied in the exact order in which they are specified.

  PIPAR takes advantage of the fact that each object is at a constant distance and the order of appearance of objects in the datastream reflects their relative distances. This greatly reduces the amount of processing and memory.
- **Scan Converting Extruded Lines** [6]. This is a sequential scheme, but some similarities are worth mentioning. The starting point here is a sequence of simple objects (stroked straight lines). The order corresponds to relative distance. The objects are sorted into buckets, one per scan line, based on their lowest point, and are stored in those buckets along with the order information. Incremental rasterization then takes place from bottom to top, with the list of active objects updated at each scan line so as to drop those objects whose rasterization has been completed and to include those that begin in that scan line. The inclusion of new objects is done by merging the active list with the contents of the new bucket while maintaining the original order among the objects. The order of rasterizing the objects that impact a given scan line is such that the latest one is rasterized first, and no objects are permitted to modify a pel that has already been impacted. (The order of rasterization is such that the "nearest" object impacts first.)

The similarity between PIPAR and this scheme is the sorting into buckets with preservation of relative distance. However, the rasterization of the last object first precludes pipelining, since all objects would have to be created from the input datastream before the pagemap building could begin). The incremental rasterization precludes "geographic" parallelism. Finally, stroked straight lines are not a natural intermediate form in processing highly expressive PDLs.

It should be noted that the intent of that work was to obviate the need for a full pagemap; instead, only the "objects" for an entire page would have to be stored. This mitigates the increase in memory requirements with increased resolution.

4. SUMMARY

We have described an architecture for a rasterizer which permits a substantial degree of parallelism within a page as well as across pages. By breaking the rasterization into multiple stages with an appropriate intermediate form, we are able to convert datastream blocks into intermediate-form objects in a "block-parallel" fashion, then merge those objects into the pagemap in a "bin-parallel" fashion ("geographical" parallelism). By deferring the test for the non-overlap requirement until there are simple objects that can be sorted easily, and then replacing it by the combination of the sorting and the order-preserving processing of the objects within each bin, preserving the correct merging order in spite of the parallel processing is greatly simplified. In fact, even blocks that would have failed a direct non-overlap test and therefore couldn't have been processed in parallel as blocks, are now effectively processed in parallel through both stages of the rasterization.

With the standard-processor implementation, the proposed architecture scales easily across a broad range of required computational power. If processors are allowed to work on any stage, performance will be determined primarily by the total amount of computation that is required, not by its constituents (graphics, image, etc.) and the available parallelism will be exploited efficiently.

The pipelined-parallel architecture is not based on an assumption of unlimited parallelism potential or on a claim that all possible parallelism can be exploited easily. Nevertheless, we do believe that a substantial degree of parallelism which can be discovered with moderate effort does exist in most cases. Clearly, the construction or identification of state-independent datastream blocks can be greatly facilitated by appropriate constructs in the PDL. This creates an opportunity for a rasterizer architecture, a page description language and applications that generate datastream to leverage one another.

ACKNOWLEDGMENTS

We wish to thank Sig Nin, Jeff Lotspiech, Wil Plouffe, Jim King, Arlen Strietzel and Steve Scott for fruitful discussions concerning the proposed architecture. We also wish to thank one of the referees for suggesting some of the references.

REFERENCES

[1] "The Advanced-Function Printer Control Unit", IBM Internal Document.

[2] M.J. Pavicic, "Superdisplays: Improving the Speed and Quality of Image Synthesis Through Parallelism and Selective Refinement", Ph.D. Dissertation, Columbia University, 1985.

[3] C. Barrera and A.V. Strietzel, "Electrophotographic Printer Control as Embodied in the IBM 3800 Printing Subsystem Models 3 and 8", *IBM J. Res. Develop.*, vol. 28, No. 3, pp. 263–275, May 1984.

[4] A. Ben-Dor and Brian Jones, "New Graphics Controller for Electrostatic Plotting", *IEEE CG&A*, pp. 16-25, 1986.

[5] S. Demetrescu, "High Speed Image Rasterization Using Scan Line Access Memories", in Proc. 1985 Chapel Hill Conference on Very Large Scale Integration, H. Fuchs, Ed., pp. 221–243, Computer Science Press, 1985.

[6] P. Willis and G. Watters, "Scan Converting Extruded Lines at Ultra High Definition", *Computer Graphics Forum*, vol. 6, pp. 133–140, North Holland, 1987.