PS: Polygon Streams
A Distributed Architecture for Incremental Computation Applied to Graphics

Rajiv Gupta

Department of Computer Science
California Institute of Technology
Pasadena, California 91125
USA

Abstract

Polygon Streams is a distributed system with multiple processors and strictly local communication. A unique custom VLSI chip that constitutes an independent processing module forms a stage of the PS pipeline. The number of these modules in PS is a variable that is determined by the application. PS features a modular architecture, multi-ported on-chip memory, bit-serial arithmetic, and a pipeline whose computation can be dynamically configured. The PS design closely subscribes to the system characteristics favored by VLSI.

The task of scan conversion for rendering computer graphics images on raster scan displays is very intensive in computation and pixel information access. It is very coherent and suitable, however, for forward difference algorithms. The discrete and regular layout of the raster display, in conjunction with the largely local effect of a pixel on an image, make rendering amenable to parallel architectures with localized memory and communication. These are precisely the attributes favored by VLSI and typical of PS.

A modification of the Digital Differential Analyzer (DDA) is implemented to Gouraud Shade and depth buffer convex polygons at high speeds. The scan conversion task is distributed over the processors to efficiently subdivide the image space and maximize concurrency of processor operation.

A study of the tradeoffs and architectural choices of the PS reveal the merits and deficits of the PS approach in comparison with Pixel-Planes, SLAMs, Super-Buffers, and SAGE.

1 Introduction

Computer Graphics has traditionally subscribed to two primary goals: realism and speed. The first goal attempts to conceive, create, and display scenes that are true to life. The latter goal is devoted to practicality. We wish to retain as much of the realism as possible as we make the modeling and rendering of images faster. Interactive, realistic computer graphics is our ultimate goal.

In the near future, as the scenes that we want to create get more complex and the phenomena that we want to simulate get more involved, we will continue to lack the ability to manipulate realistic images in real-time. To justify spending the time and computational resources in obtaining an exact image, we should first convince ourselves that the interactions in the scene – of objects, viewing parameters, light sources, to name a few – are as we had envisaged them. In order to make this decision with reasonable accuracy we need a way to quickly preview approximations of these scenes interactively. Real-time simulations and animation also require images containing thousands of polygons to be rendered within a frame time. The work described in this thesis renders shaded and depth-cued polygons on raster scan displays at speeds that match the requirements of these applications.

1.1 Organization of theThesis

In Section 2 we present the context of the PS System application in a graphics pipeline. In Section 3 we discuss the system architecture of PS. This includes the VLSI implementation of the incremental algorithm discussed above and a detailed design of the scan line interpolation subsystem (SLI) of PS. In Section 4 we discuss some of the salient features of the PS design. Section 5 compares the PS architecture with that of the Pixel-Planes, SLAM, Super-Buffers, and SAGE systems. In Section 6 we discuss enhancements to PS and explore the realm of application possibilities of the architecture. Section 7 concludes the document. Appendix A is a report on the fabrication details and expected performance of the PS system.

2 The Graphics Application: The Depth-Buffer Rendering Pipeline

In the graphics pipeline for converting an image from the world or modeling coordinates to the screen or pixel coordinates, a polygonal object is typically described in terms of the position, x, y, and, z, and color, r, g, and b, attributes of its vertices. The viewing, clipping, perspective, and lighting computations give us a new set of vertex and color attributes in screen space. The PS system accepts the screen coordinate and color information of the vertices of a convex polygon and interpolates them to produce (z, r, g, b)-tuples for all pixels that lie within the polygon. This information is suitable for raster-scanned displays. The shading and visible surface determination task is the most computation and communication intensive component of the graphics pipeline. We render convex polygons at very high speeds by implementing two of Computer Graphics' well-known algorithms: depth-buffering [Foley 84] and Gouraud shading [Gouraud 71].

3 The Architecture and Organization of the PS System

3.1 System Architecture

The system performs two distinct but dependent interpolations: along polygon edges and along scan lines. Along polygon edges we interpolate x, r, g, b, and z against y to obtain pixel values at the edge pixels that constrain the polygon on consecutive scan lines. Along a scan line we interpolate r, g, b, and z against x to obtain pixel values at consecutive pixels that lie within the edge pixels of that scan line.

For a given scan line, only after the values at the edge pixels have been calculated may the values for pixels intermediate to those edge pixels be calculated. Note that scan line interpolation is dependent on the polygon edge interpolation only for the values of the edge pixels. Also, interpolations along different scan lines are mutually independent for the same reason. This lends the system to a functional hierarchy – polygon edge interpolation and scan line interpolation. The PS system architecture is shown in Figure 1.

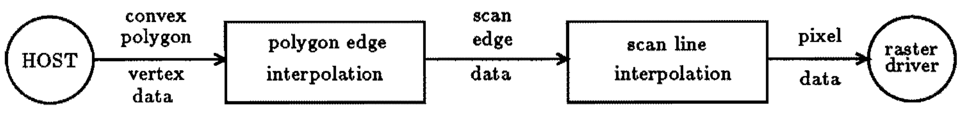

Figure 1: The PS System Architecture

Since we are not constrained by any polygon processing order we can arbitrarily extend the architecture at this level. If the polygon edge interpolation (PEI) is the bottleneck we can balance the system with multiple PEIs. Conversely, if the scan line interpolation (SLI) is the bottleneck we can balance the system with multiple SLIs. A hybrid approach, allows us to optimally balance the number of SLI's with respect to the PEI's, and the numbers of PEI's and SLI's with respect to cost-speed-accuracy indices. It is now trivial to fine tune the architecture to the scene composition. For example, scenes with consistently small polygons, as in the case wherein the accuracy in rendering complex curved surfaces is enhanced by fracturing the surface into larger numbers of smaller polygonal artifacts, require more PEIs.

The potential for parallel hardware is readily apparent. This paper deals with the detailed design and implementation of the scan line interpolator subsystem.

3.2 The Scan Line Interpolation Subsystem

3.2.1 Parallelism

Since the interpolations along different scan lines are independent operations, the scan line interpolator subsystem consists of several scan line interpolators which work in parallel. The best that we can do is to have an independent interpolator for each scan line. The time to interpolate the entire convex polygon is then the time to interpolate its longest scan line extent.

3.2.2 Pipelining

Interpolations along the same scan line for different polygons are mutually independent. Also, the interpolation for a scan line can be broken into discrete and repetitive steps that successively interpolate for the pixels that lie on the scan line. Thus each scan line interpolator can be pipelined so that interpolation for polygon 'a' can be started before that for polygon 'b' is completed.

For a given scan line y, we may interpolate at $(x1, y)$ for polygon 'a' concurrently with the interpolation at $(x2, y)$, $x1 \neq x2$, for polygon 'b'. Note that $(x1, y)$ and $(x2, y)$ cannot belong to the same convex polygon because the interpolation at $max(x1, x2)$ is dependent on the cumulative error calculated at the interpolation of $min(x1, x2)$ if $x1, x2$ belong to the same polygon. Thus the scan line processor can be split into many sub-scan line processors each responsible for a set of m unique pixels and potentially interpolating for a different convex polygon. In Figure 2 both convex polygons are interpolated in parallel by a non-intersecting set of sub-scan line processors.

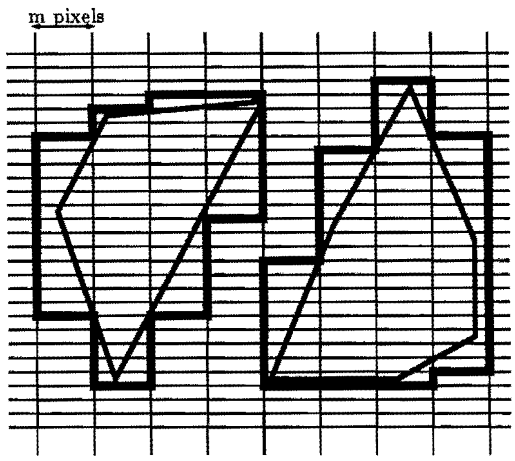

Figure 2: The Parallelism and Pipelining in the system: both polygons are interpolated in parallel by the pipelined scan line processors.

The sub-scan line processor stores its pixel-set information, $(z, r, g, b)_i$ for $1 \leq i \leq m$, in its private memory. This memory is organized as a pixel and depth buffer for the sub-image. Thus over the array of processors we have a distributed frame and depth buffer for the entire image.

The DDA implementation for this processor array and the data format we chose gave us an elegant, efficient, and versatile interconnection structure.

3.3 Data Structure and Algorithm Implementation

Each iteration through the inner loop of the DDA gives us the interpolated value for one pixel. It requires the divisor (dy), the quotient, the remainder, the last interpolated value, and the cumulative error (p_{sum}). Besides the cumulative error and the interpolated value, all the other values need to be calculated once for each scan line. These are calculated by interpolating along the polygon edges and are input from the polygon edge interpolator subsystem for each scan line. Thus the inner loop of the DDA constitutes scan line interpolation.

By pipelining the scan line interpolator into multiple sub-scan line interpolators we are effectively opening the inner loop of the DDA into multiple loops each with a default number of iterations equal to the extent of the sub-scan line processor, m in our case. Thus for a given scan line, y, sub-loop1 interpolates for $0 \leq x \leq m-1, y$; sub-loop2 for $m \leq x \leq 2m-1, y$ and so on. When sub-loop1 finishes, it puts all the required data into the registers for sub-loop2. When sub-loop2 starts working on these values sub-loop1 is now available to interpolate for the sub-scan line $0 \leq x \leq m-1, y$ of another convex polygon. Note that although the sub-loops are working concurrently, they are interpolating for different polygons. *For a given polygon and scan-line there is a definite order of sub-loop operations and a unique data packet in the pipeline.* Parts of this data packet, such as the last interpolated value and the cumulative error get modified as the data packet moves through the sub-loop pipeline. This imposes a simple communication structure: sub-loop$n$ communicates with sub-loop$(n+1)$ to whom it sends the modified data packet, and with sub-loop$(n-1)$ from whom it receives data packets.

If we break the loop as mentioned above, in addition to the values above the data packet needs to carry x_{start} and x_{end}, the start and end values for the loop. ($x_{\text{end}} - x_{\text{start}}$ = the divisor). Note that all the sub-loops perform the same function. For sub-loop$q$ (which interpolates for $((q-1)m \leq x \leq qm - 1, y)$ this would be:

if new data in input registers and $x_{\text{start}} \leq qm - 1$ then interpolate for the scan line section that lies in its domain *i.e.* from $max((q-1)m, x_{\text{start}}), y$ until $min(qm-1, x_{\text{end}}), y$; if we have not reached the last pixel of the polygon on that scan line, fill the registers of sub-loop$(q+1)$ when it is ready to accept the modified data packet.

In spite of their common functionality, the sub-loops need to know their index, q, to make the comparison, $(q-1)m \leq x \leq qm - 1$, and are not identical. This implies that we need $ceil(X/m)$ different sub-loops (processors) for a screen size X pixels wide. This translates to extra logic and initialization overheads to simulate different sub-loops from identical implementations of the processor.

With a slight modification in the data format and functionality this limitation can be overcome. In the format above, the values of x_{start} and x_{end} were fixed with respect to the image axis and unchanged for all of the sub-loops. But the subloops do not need the absolute values of x_{start} and x_{end}; they need these values only with respect to their domain. For example subloop$q$ only needs to know that it should start interpolating from the third pixel in its domain. The connectivity information tells us that this "third" pixel of the subloop corresponds to $x = (q-1)m + 3$. Thus if x_{start} and x_{end} could somehow be "normalized" with respect to the sub-loop indices, then the index information would be redundant and the sub-loops could be identical.

Conceptually, we normalize by associating a y-axis with each data packet. As it moves through the pipeline, the data packet shifts its y-axis. Thus the y-axis for a data packet shifts to the right by the sub-loop count, m, as the packet moves between successive sub-loops. As Figure 3 shows, by this data manipulation we are effectively moving the sub-loops to the left and keeping the modified data packet stationary. Now all the sub-loops believe that they are the starting sub-loop and hence are identical.

We implement the normalization by :

$$\begin{aligned} x_{\text{start}}(\text{to be sent}) &:= x_{\text{start}}(\text{received}) - m(\text{the loop count}) \\ x_{\text{end}}(\text{to be sent}) &:= x_{\text{end}}(\text{received}) - m(\text{the loop count}) \end{aligned}$$

$dx = x_{\text{end}} - x_{\text{start}}$, remains unchanged.

Irrespective of index, all sub-loops now perform the function:
if new data in input registers and $x_{\text{start}} \leq m-1$ then interpolate for the scan line section that lies in its domain *i.e.* from $max(0, x_{\text{start}}), y$ until $min(m-1, x_{\text{end}}), y$; if the last pixel of the polygon on that scan line has been interpolated then do not send any data packet down the scan line. Otherwise, renormalize x_{start} and x_{end} for the next sub-loop; wait until the next sub-loop is ready to accept data; fill the registers of the next sub-loop.

With this normalization, the sub-loops are all identical and the index information is hardwired in the communication structure. Thus we can replicate this one generic sub-loop for displays of arbitrary size.

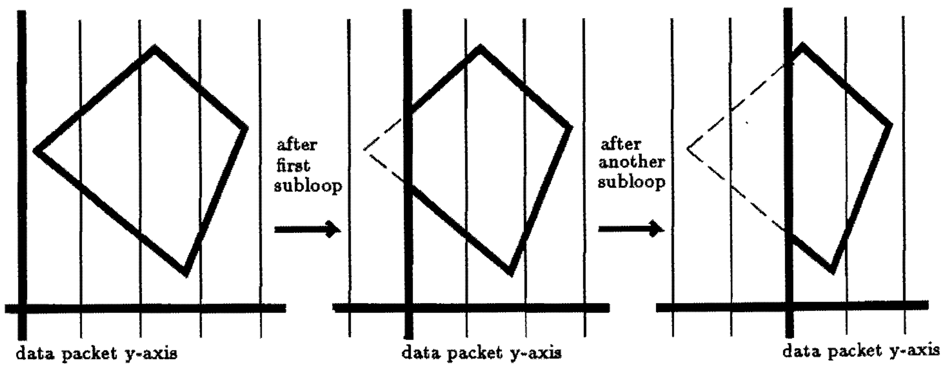

Figure 3: The effect of "Normalizing" a data packet

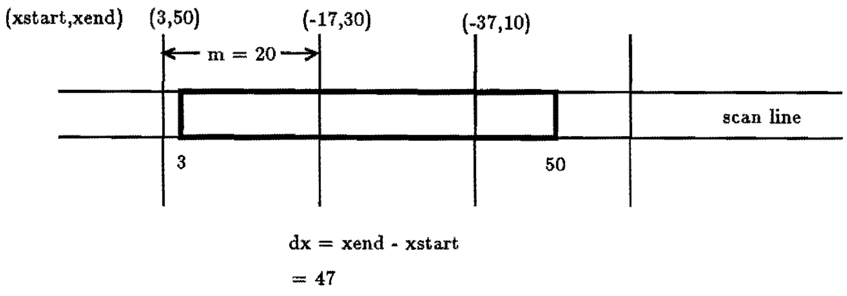

Figure 4: A numerical example to illustrate "Normalization"

3.4 Translation into VLSI

3.4.1 The Two-Dimensional Chip Array

In translating into VLSI, we mapped n of these sub-scan line processors each with its private memory onto one chip. Thus each chip is responsible for a unique m by n pixel subspace. The mapping is illustrated in Figure 5.

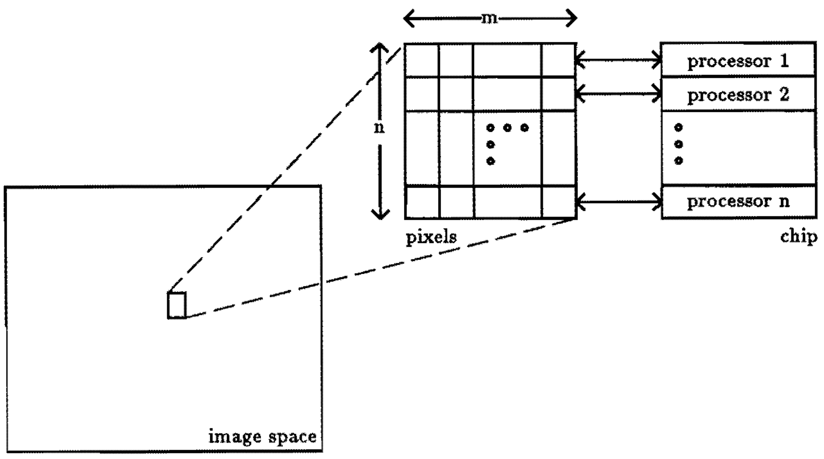

Figure 5: Processor-Pixel Mapping

All processors $i, 0 \leq i \leq n-1$, correspond to the same sub-loop index but for different loops (scan lines). Processor $i, 1 \leq i \leq n$, in chip(j,k) interpolates for pixels that lie in its domain, $((j-1)m \leq x \leq jm-1, (k-1)n+i)$. Processor i in chip$(j+1,k)$ interpolates for $(jm \leq x \leq (j+1)m-1, (k-1)n+i)$ and is the adjacent subloop to processor i in chip(j,k). Thus data packets (unique to a scan line of a polygon) move from left to right in the row of chips and only the data packets for scan lines $(k-1)n \leq y \leq kn-1$ need to be input to row k starting at chip$(1,k)$.

Subpixelling is a way of reducing the aliasing effects inherent in the depth buffer algorithm. Each physical pixel is now fractured into multiple (usually a square grid) virtual sub-pixels. We interpolate in this virtual image space that is larger than the physical image space (screen space) and use an averaging filter to merge the sub-pixels when we paint a physical pixel.

For a display size of Q by R and a subpixelling size of q by r, the array of chips would have indices $1 \leq x \leq ceil(Qq/m) = X, 1 \leq y \leq ceil(Rr/n) = Y$. Each chip communicates with the two chips on either side of it in the row, *i.e.* chip$(a,b), 1 < a < X; 1 \leq b \leq Y$, communicates with chips chip$(a-1,b)$ and chip$(a+1,b)$. Chip$(1,b), 1 \leq b \leq Y$, communicates with chip$(2,b)$ on the right and with the polygon edge interpolator on the left and chip$(X,b), 1 \leq b \leq Y$, communicates only with chip$(X-1,b)$ on the left.

If the incoming data packet is for a busy processor (with all its input buffers full) then the chip passes on a busy signal to the chip on its left so that it may refrain from overwriting this data. This communication ensures that no information is lost when we have a choked pipeline. If the data is for a sub-scan line that intersects the chip subspace, then the data packet is written into the scratch registers of the corresponding processor and the chip is free to accept more data. If the first pixel, x_{start}, lies beyond the chip domain, the x_{start} and x_{end} values are normalized and the data packet is passed onto the next chip (on the right) when it becomes free to accept this data. A normalized data packet is also passed onto the next chip when an on-chip processor completes interpolating and the last pixel on the scan line, x_{end}, does not lie in its domain. If the last pixel has been interpolated then we have finished interpolating that scan line and the data packet is not passed on. Since the polygons are clipped to the image space, it follows that chip(X, y),$1 \leq y \leq Y$, need only communicate with chip$(X-1, y)$ on the left.

Communications to the left are to indicate the busy status of the pipeline and those to the right are for transmitting scan data; this data is always normalized. The interprocessor communication is by asynchronous, source-initiated handshaking as data transfer is not lock-step with every clock tick. The communication paths for a chip array are depicted in Figure 6.

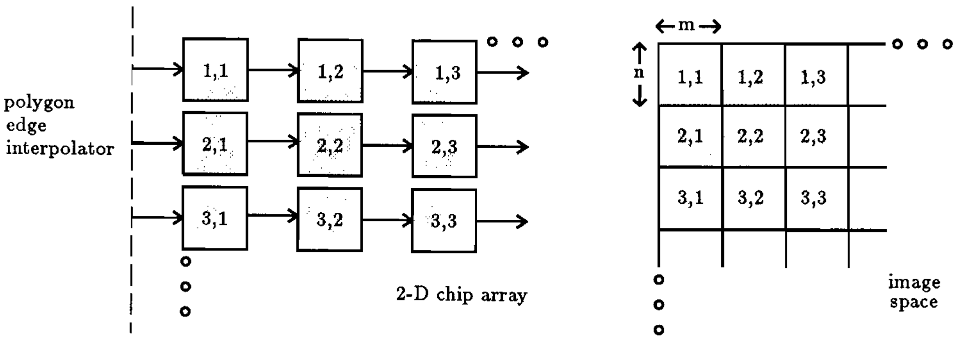

Figure 6: The communication structure of the chip array

We have a wave of data packets moving from left to right through this chip array for each polygon. The wavefront for a polygon initiates interpolation for that polygon in all the incident processors. Polygon waves may be rhythmically pumped into the system in a systolic manner.

3.4.2 Chip Architecture

Each chip has to handle handshaking and data communication for its n processors. The information for communicating with the chip to the left is distinct and independent of that with the right. In the former case we accept data packets and specify our status. In the latter, we send data packets and query the status of the next chip. Still another distinct source of data communication is with the external video circuitry for displaying the pixel values for the m by n pixels of this chip. This requires that we recognize a video-scanning signal and in response output the values of the current pixel as we cycle through all the m by n pixels.

The four distinct functional components are:

- n processors each with its private memory for m unique pixels.

Each processor interpolates for the m pixels in its domain and the memory is part of the distributed pixel and depth buffers.

- Preprocessor
  The preprocessor inputs data packets and outputs its status to the chip at the left or to the polygon edge interpolator in the case of the first chip of the row. If the data packet is not for any of the chip's processors (the starting pixel lies beyond the chip domain) it passes the packet to the postprocessor.
- Postprocessor
  The postprocessor takes data packets from the n processors and the preprocessor, normalizes them, and outputs them to the next chip. It is responsible for all the handshaking signals for this communication with the preprocessor of the next chip.
- Video-processor
  The video-processor cycles through the on-chip buffer and outputs the current pixel information in response to a video-out signal. Its access of the memory banks is transparent to, and independent of, the processors.

Additionally, each chip has a clock and timing signal generator that converts the single phase clock input to an internal two-phase non-overlapping clock. The clock generator is also responsible for producing the control signals for the preprocessor and the postprocessor and the control signals that are common to all the processor and memory banks.

This functional modularity drives the chip architecture shown in Figure 7. In the interests of brevity, only the details of the memory design are included in this document. The interested reader is referred to [Gupta 87].

The Memory Design

The total chip memory, corresponding to the pixel and depth buffers for m by n pixels, is divided into n banks, one for each processor. The memory for each processor is independent of the other processor memories.

The processor access of the pixel memory is pseudo-random; between the first and last pixels to be interpolated the pixel memory is accessed sequentially. Thus only the first pixel access is random. For the subsequent pixels, the address for the pixel just computed can be incremented to access the next pixel to be computed. We implemented this by a token that successively passes through the m pixels of the processor in the spatial order of the pixels in image-space. When the preprocessor routes the incoming data packet to the relevant processor, the "x_{start} control" unit of the preprocessor initiates the start of this token for the processor so that by the time the processor is ready to access pixel memory, the first pixel to be interpolated has the token and can send its data to the processor on a pre-charged bus. The n tokens on the chip, one for each of the n processors and its memory bank, is independent of the video token that crosses memory bank boundaries.

In the other dimension the $r+g+b+z$ bits of each pixel memory are cyclic because the processor is bit-serial and needs only a bit of information on each system clock cycle. The implementation approximates a circular shift register for the $r+g+b+z$ bits of each pixel. Memory read-out is thus destructive.

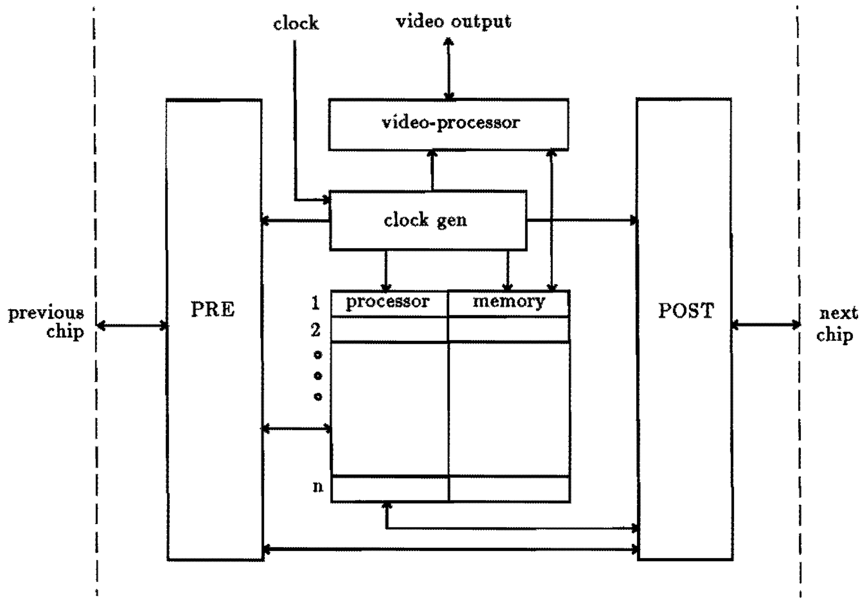

Figure 7: Chip Architecture

Figure 8 illustrates the memory organization.

4 Salient Features of the PS Architecture

4.1 Bit-Serial Arithmetic

Our architecture is communication limited. Bit-serial arithmetic helps reduce the clock period and thus boosts the communication bandwidth by a factor close to the number of bits of alternate parallel hardware. Serializing memory and logic in the design of a module while increasing the number of parallel modules improves hardware utilization and throughput at the cost of an increase in latency.

Bit-serial processors improve the flexibility of data formats. In our case, altering the data format or resolution implies minor changes in the clock and timing signal generator. It also helps in lowering the pin count and silicon area of the chip. Enhancing the connectivity of the structure, say to a hexagonal connectivity for dithering, now will not increase the pin count inordinately.

4.2 Systolic Adaptation

We have the strictly local and regular communication structure of systolic architectures. The DDA we implement, closely maps the pipelining inherent in the systolic approach. Our adaptation diverges from a purely systolic computation in that every processor does not compute on every clock tick. We use the systolic paradigm to capture the concepts of parallelism, pipelining, and interconnection structure but we do not subscribe to the strict lock-step computation of systoles. Just as in systolic systems two communicat-

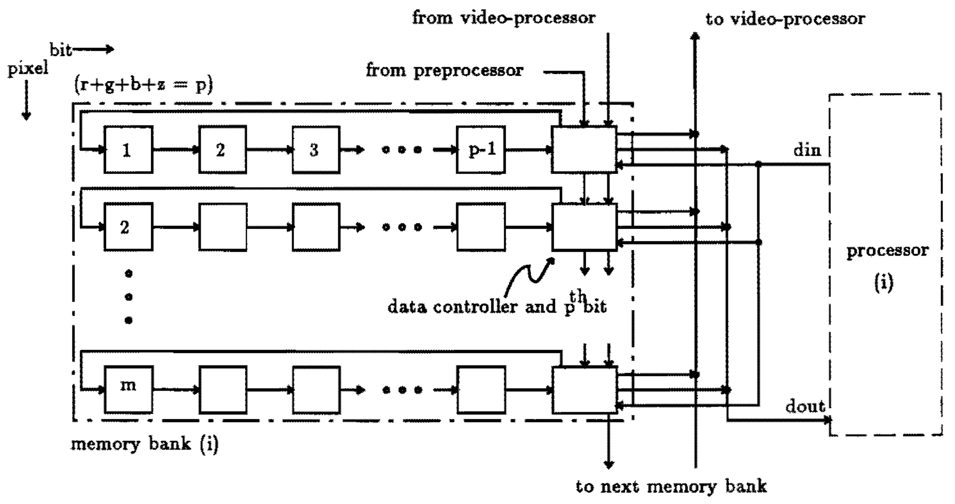

Figure 8: Memory Bank Design

ing processors have a data path between them but in our case the communication is asynchronous.

Another divergence is that the sequencing of operations is neither built into the nodes nor are control signals broadcast into the array in an SIMD fashion; in our adaptation the control for the computational array is distributed into the array. In the current implementation, the data encodes the operation – normalize, interpolate for a variable number of pixels, kill the data packet – mimicking a data-flow architecture. In manipulating the data packet each processor is effectively encoding the command for the next processor. Additionally, the data packet also contains an explicit command field. Currently the only encoded command controls z-compare. PS allows each processor to affect the operation of subsequent processors in the scan line by manipulating the data and also by explicitly manipulating the command field. We hope to exploit this flexibility to address complex applications.

4.3 On-chip Memory

Scan line conversion requires a high memory bandwidth. Keeping in mind the penalties that off-chip accesses entail in terms of power, speed, pin count, and silicon area, we have a compelling reason for on-chip memory. There are other not so obvious advantages of on-chip memory. Distributed memories that are in close proximity to the processor offer less memory contention and global address decoding. The on-chip memory and processor share a common clock reducing concerns of clock skew.

We can exploit the VLSI advantage by specializing the on-chip memory to the processing task. In our case this corresponded to a multiple-ported, dynamic memory that mimics a circular shift register for each pixel. It is optimized for bit-serial arithmetic. The data dependencies and locality that are typical to our processing task give us a simplified, fast, and small memory decoder, and reduced wiring.

4.4 Algorithm Implementation

The algorithm implementation for the distributed system tracks the simple and local connectivity of the systolic structure. The data structure carries all relevant information thus avoiding the need for common busses or global communication. It automatically tesselates polygons onto the underlying chip array structure adding extensibility to the implementation. Tesselation does not incur any space or time overhead as would be the case in a higher level partitioning scheme.

4.5 Hierarchical and Distributed Control

The primary considerations in the design of the control circuit are the silicon area, the speed of the chip, and the design time. For VLSI these are direct manifestations of the cost of the circuit. We employed a hierarchical design that separates the sequencing and the generation of commands [Obrebska].

In our design the clock and timing signal generator forms the top level of the hierarchy of the control. By extracting the timing signals that are common to the different functional units of the chip we avoid repetition and redundancy in the signal generation. These signals form the skeleton of the control structure providing validation and supervision for the flesh of the structure within and between the units. This flesh is distributed at various levels. The distribution gives us flexible and localized control, and minimizes long-distance communication.

The clock and timing signal generator is implemented as a PLA. The PLA construct is easy to generate, test, and debug. Its use at this level makes the implementation flexible. We can change the parameters of the implementation *viz.* the system vs. processor clock ratio, the data format, the buffer sizes, the screen resolution, the number of pixels per processor (m), with minor modifications. The versatility of the PLA structure, the irregular nature of the control logic, the large size of the corresponding FSM, and the fact that there is only one of these per chip justify the area of the PLA.

Token-passing is used to control communications between units. Pre, post and processor communications use one polling token, video communication employs a token that circulates through all the m by n pixels of the chip, and each processor-memory bank communication is controlled by an independent token; thus each chip has $n+2$ independent tokens. All information transfers from the processors to the postprocessor, from the preprocessor to the processors, and between each processor and its pixel memory bank are initiated by the processor clock. A processor clock period being $r+g+b+z$ times the system clock period, we have $r+g+b+z$ system clock cycles for decoding communication requests. At the same time, we want to avoid personalized control lines for every type of communication. Token-passing gives us the best tradeoff between decoding time, size of decoding circuitry, and multiplicity of control lines. The use of a shifting token for processor-memory bank control exploits pixel coherence (interpolation for pixel $i+1$ follows that for pixel i). The video-memory token requires no decoding.

Within each unit, control is implemented by gates and dynamic shift registers mimicking the "delay-element method" [Hayes]. This design style conserves area and reduces delays because of the small incremental cost of latches in MOS technologies. The internal functions of the processors are directly controlled at this level. It is therefore repeated as many times as the number of processors per chip (n); this makes the area considerations significant. The regularity of the algorithm and the "tight" properties of this design style justify its non-uniformity and thus lack of automation.

4.6 Logic Implementation

The use of steering logic allowed us to exploit the features typical of the implementation technology, namely CMOS. We have pass transistor logic networks with the logic level appropriately restored. These networks control data or signal flow between data latches - another elegant and low cost feature of MOS logic.

Static CMOS logic, however, requires us to implement both the complimented and uncomplimented forms of the function in n-type and p-type transistors respectively. Not only is silicon area wasted in the duplication, but the problem of routing signals is aggravated. We used precharge logic whenever possible. Besides the savings in circuitry and routing, precharge logic gives us lower capacitances and hence higher speeds. With only one p-type transistor in the pull-up path, load capacitances charge faster. We easily derived mutually exclusive compute-precharge phases from the two-phase non-overlapping clock on the chip thus fulfilling the precharge requisite that inputs to precharged busses be glitch-free and that the bus not be used for the time it needs to precharge. We opted not to concern ourself with the lower speed bound in using dynamic logic.

5 Comparison with Existing Polygon Rendering Systems

Our design exploits parallelism by partitioning the image space so that each processor is responsible for only a sub-image and consequently fewer objects. Another possible approach is to partition the object space so that each processor is responsible for only a subset of the objects in the image. We compare the PS architecture with four prominent and recent architectures for scan conversion that apply image space partitioning. These are:

- Pixel-planes by Henry Fuchs et al. at the University of North Carolina
  [Fuchs 81,Fuchs 83,Poulton et al. 85]
  Pixel-planes is a processor-per-pixel architecture for Gouraud shading and depth-buffering polygons. The designers have since discovered other interesting applications that map onto the functional paradigm of the architecture. The most unique aspect of the architecture is the use of multiplier trees to calculate the linear equation, $F(x) = Ax + By + C$, at each pixel. Differently said, the multiplier trees decode linear equations onto the pixel space.

- Scan Line Access Memories by Stefan Demetrescu at Stanford University
  [Demetrescu 85]
  Scan Line Access Memories, or SLAMs, use conventional high density RAMs enhanced with limited processing for high speed rasterizing of large sections of scan lines. The current implementation runlength encodes a specified 16-bit halftone pattern to affect an entire, or a subset of, a scan line. The architecture draws strength from its simplicity and the use of conventional RAM design concepts.

- Super-Buffers by Nader Gharachorloo et al. at Cornell University [Gharachorloo 85]
The Super-Buffer architecture is a systolic approach to the task of rasterizing polygons. Super-Buffers, developed independently, share many features with PS. Notably, we both divide the task of rendering polygons into polygon edge interpolation and scan line interpolation. In polygon edge interpolation both systems employ an incremental algorithm. In addressing the communication problem, our approach is a variation of the systole to which the Super-Buffers adhere strictly.

- Systolic Array Graphics Engine by Nader Gharachorloo et al. at IBM, Thomas J. Watson Research Center [Gharachorloo et al. 88]
Systolic Array Graphics Engine, or SAGE, is an update to the Super-Buffer system. Also developed independently of PS, the SAGE architecture is very similar to Super-Buffers with the added functionality of Gouraud shading. The primary differences between SAGE and PS remain the processor-per-pixel of SAGE versus the processor-per-subscanline of PS, the external frame–buffer requirement of SAGE versus the distributed frame–buffer of PS, and the common video and processing clocks of SAGE versus the independent video and processing clocks of PS.

In our comparison, we characterize the efficiency of an architecture by:

- the extensibility of the architecture to increasing functionality.
- the extent of host involvement.
- the speed potential.
- the processor utilization.
- the communication bandwidth.

Only PS, Pixel-Planes, and SAGE currently implement Gouraud shading and depth-buffering. SLAMs and Super-Buffers have a one-bit plane at each pixel and do not support hidden surface removal.

Pixel-Planes inherently support all functions that can be represented as a linear equation. Functions that do not have the linear equation as their natural representation require extensive preprocessing. For example, polygons are most conveniently described in terms of their vertices. Edge equations are easily obtained from this information; however planar equations for color require more preprocessing. Super-Buffers and SAGE are limited in that no memory is retained past scan line boundaries, requiring all pixel information to be retransmitted through the row of Graphics Engines at the onset of scan lines. This requires pixel information for the entire image space to be stored externally along with data for all the active polygons. The SLAM architecture is optimally designed for a 16-bit function repeatedly applied to all pixels on a scan line. The architecture lòses most of its elegance and performance metrics for more involved processing. The PS architecture is optimized for incremental arithmetic and local communication. With the understanding that all global communication can be implemented as local communication using systolic conversions [Leiserson a,Leiserson b], we foresee few limitations to applications with our architecture.

Global communication in Pixel-Planes and SLAMs limit the performance potential of the systems. In Super-Buffers and SAGE, the image clock is also the video clock, limiting the polygon throughput. In all of these systems the memory is essentially single-ported; thus, video access and image processing are not completely transparent. Our simple decoding structure gives us multi-ported on-chip memory with completely independent and transparent video and image accesses. Also, in all the other architectures that we have considered all the chips are functionally different. This translates to additional logic, delay, or host involvement in an initializing phase. With a simple "normalizing" function, we avoided that requirement.

Processor utilization in Pixel-Planes is low by virtue of the processor-per-pixel approach to a lock-step-with-polygon character enforced by the global communication. In SLAMs, processor utilization is high only for applications with a repeatedly applied pattern covering large sections of a scan line. Super-Buffers, SAGE, and PS enjoy a high processor utilization profile. In Super-Buffers and SAGE, however, the processor-per-pixel architecture witnesses many processors simply performing the x_{left} and x_{right} comparison. This dilutes the effective processor utilization. In the same token, the processor-per-subscanline of PS might witness bottlenecks when consecutive data packets are destined for the same processor. In the worst bottleneck case, the enhanced processor-pixel mapping of Subsection 6.1 loses a factor of m, where each processor is responsible for an m-pixel sub-scanline, in comparison to Super-Buffers or SAGE. This underlines the tradeoff between processor utilization and speed potential, or the average case versus the worst case.

Of all the architectures reviewed in this comparison, the SAGE architecture bears the closest resemblance to PS and merits a closer inspection. In SAGE just as in PS, the boundary between the polygon edge interpolator and the scan line interpolator limits the communication bandwidth of data packets along a scan line. This observation in conjunction with the processor-per-pixel approach justifies the purely systolic communication of SAGE. In the processor-per-subscanline or processor-per-multiple pixels architecture of PS, however, the number of pixels interpolated by a processor may vary from 0, in cases of sub-scanlines that do not intersect the domain of the processor, to m, in cases of sub-scanlines that overlap the entire m-pixel domain of the processor. Thus the processing time per data packet is not constant, necessitating the asynchronous hand-shaking communication in PS.

The absence in SAGE of a distributed frame–buffer is closely tied to the reuse of the processing clock as the video clock; this is by far the most significant difference between SAGE and PS. When rendering more complex scenes than those permitted by the video rate, SAGE requires an external frame–buffer for storing the output of the SAGE chips. This implies either that the buffer values for a scan line have to be shifted into the vector of processors in time before the new interpolation for the same scan line, or external circuitry is needed for correlating the new and previous values at each pixel. The distributed frame–buffer and independent video and processing clocks of PS permit graceful video degradation with fewer frames per second. Even in the case of simpler scenes, the processing rate matches the video rate only for four interleaved SAGE chips. SAGE appears to have no way of recovering and retransmitting the error term of the interpolation thus potentially introducing aliasing artifacts at the boundary between the chips.

The extent of host involvement is significantly different between the SAGE/Super-Buffer and PS architectures. Consider the following abstraction: the task of depth-buffering is effectively the sorting of all polygons along x, along y, and then for a given (x, y), *ie.* for each pixel on the display, the sorting of all relevant polygons along z. For each pixel, the z-buffer retains the intensity data only for the closest polygon. The sorting along x and y is simply an optimization to avoid processing all polygons at every pixel. Polygon sorting along x is circumvented by providing a processor at each pixel along the scan line; the z-sorting of all polygons for all values of x is then off-loaded from the host to PS/SAGE/Super-Buffers/etc. Similarly, scan line processors for each scan line in PS circumvent the need for polygon sorting along y; the z-sorting of all polygons for all values of (x, y) is off-loaded from the host to PS. SAGE/Super-Buffers on the other hand, only z-sort polygons along x, the sorting of polygons along y continues to remain the responsibility of the host. In this sense PS is an evolution of the SAGE/Super-Buffer architectures.

6 Application Possibilities and Enhancements of the PS Architecture

This chapter is organized in an extensibility precedence: the extensions to the system are described in a decreasing order of detail and/or an increasing magnitude of modifications. All the enhancements share the same design philosophy – on-chip memory, decoding that exploits data dependencies, local communication, bit-serial processing – as the current system, and most cases propose a simple increase in the functionality of the processor.

6.1 A Better Processor-Pixel Distribution

The connectivity of the chip array in the current implementation introduces an asymmetry in communication. The communication requirements decrease as we move from the first column of chips to the last as the first chip in the row has to transfer/process data packets for all the chips in the row. As we move down the row, in increasing x, data packets terminate and the rate of data flow between chips falls.

Currently, the n processors of a chip are stacked vertically so that each chip is responsible for a square array of pixels. However, if we connect the processors back-to-back so that they span contiguous sections of the same scan line, we continue to have XY/N^2 chips in the system but we change the aspect ratio of the array. The chip array changes to X/N^2 by Y from X/N by Y/N, and the processor array is the same at X/N by Y. However with X/N processors per scan line and only one scan line per row, we now have X/N processors per row of chips and not X processors per row as earlier. Also, data packets span potentially larger sections of a chip's domain. These reduce the data packet requirements per row of chips. Conversely, with a chip now processing larger scan sub-sections the data traffic between chips falls. Thus we help remedy the asymmetry of communication discussed above.

The lower communication requirements at the same communication bandwidth improves throughput and processor utilization. In the context of locality of reference which has data for scan line $i + 1$ following that for scan line i, we increase parallelism by assigning one scan line per row of chips. Note that we have the same number of processors per scan line leaving the performance contribution of pipelining intact.

In this approach consecutive processors span contiguous sections of the pipeline. In the context of pixel coherence, the data dependency that applies to adjacent pixels within a scan line, we can now simplify and enhance the power of the token decoders for higher performances.

6.2 Processor Enhancements

We only consider applications that can take advantage of the system architecture. We expect that many algorithms could be reformulated to exploit the distributed control, communication and processing structure of the system. The list of possible applications is not purported to be complete. The last two in the list below embody the structure which promises interesting solutions to difficult problems.

6.2.1 Anti-aliasing

Aliasing effects arise from the finite size of pixels. Differently said, it is the quantization error of converting real pixel values to the integer precision of the display. In the DDA computation although the data packet carries only integer information, all the precision of the real number interpolation at each pixel can be reconstructed from the values in the data packet. As an example, p_{sum}/dy, the difference in the integer x pixel value we adopt and the true x value at the scan line, is available as the integer values p_{sum} and dy. We can use this information to color edge pixels so that edges appear anti-aliased. Note that this is only an approximation to true anti-aliasing and hence is not free from defects.

6.2.2 BitBlt

By BitBlt [Foley 84] here we mean copying or moving arbitrary sections of the screen to another location on the screen with one command.

The pixel data now originates from the source sub-screen and is routed to the destination sub-screen through the switch network between the polygon edge and scan line interpolator systems. A data packet with the source pixel location, length on the scan line, and destination scan line and pixel location moves through the scan line and affected pixel data is appended to the data stream at the source and stripped from the top of the stream at the destination.

6.2.3 Other Incremental Algorithms

The architecture supports algorithms that can be described incrementally. Bresenham's Circle algorithm [Foley 84] is a typical example. It ideally maps the architecture and can be implemented with only minor enhancements of the processor, making the circle throughputs of the system comparable to the polygon throughputs. Other interesting classes of algorithms and graphical effects can be obtained from the current data structure. A case in point involves successively changing the constants in the data packet. By left-shifting the quotient at each computation, we can trivially compute the function 2^x where x is the pixel attribute. Algorithms that can optimally utilize the parallelism of PS can be found in many disciplines. These algorithms run inhibitively slowly on current mainframes and the performance improvement by running them on PS would make them practical and useful.

6.3 Fault-Tolerance

We can incorporate a measure of fault-tolerance by having an extra processor and memory bank per chip and storing the bank attribute (good/bad) in the token decoder. If the processor or memory of any one of the n banks of a chip, say p, is found to be defective the token decoder delays by one clock cycle the valid signal for banks that follow the defective bank in the token path. In this simple way banks $p \ldots n-1$ are effectively replaced by banks $p+1 \ldots n$, where banks $0 \ldots n-1$ are the chip's n banks and bank$n$ is the extra bank added to tolerate a faulty bank.

The detection of a problem is simplified by the existence of the postprocessor; since all banks are identical the same scan line can be sent to successive processors and the result compared in the postprocessor. We do require some additional logic on the chip to do this and to reflect the result in the token decoder. The loss in area incurred by this fault-tolerant measure is less than $1/m$ times the chip area. Also, the chip forms a generic PS stage and we can replace a faulty chip with a new chip which will reset itself with the background polygon at the start of the next frame. We are not required to interrupt and reinitialize the entire system.

7 Conclusions

PS exhibits the characteristics of distributed systems: increased performance, extensibility, and modular architecture. An eclectic bag of architectural methodologies directly addresses the limitations on communication bandwidth and processor utilization. The power of PS lies in the extensive pipelining and parallelism it incorporates. Bit-serial arithmetic, systolic connectivity, on-chip memory, and hierarchical control effect the performance metrics and extensibility of PS. In its implementation, PS exploits the fabrication technology, CMOS.

A number of tradeoffs and marriages make the PS architecture unique: the distributed communication of systoles leading to high bandwidth and the independence of asynchronous communication; decoding time *vs* the minimal control wires as in token-passing; the low processor utilization of a processor per pixel *vs* the communication and processing bottlenecks of an m by n approach.

In the context of the current application, the reformulation of the DDA maps the architecture efficiently. A judicious processor-pixel distribution is shown to enhance the symbiotic relationship of the application and the architecture. The proposed architecture exploits the data dependencies peculiar to the task of scan conversion. The exercise underlines the advantages of concurrently developing the algorithm, data structure, and architecture.

In designing the PS system, we witnessed an anachronism in design psychology: "memory" = "RAM". Even for the on-chip memory of specialized hardware, the organization is or closely resembles a RAM. However in most cases, certainly in ours, random access is overkill; And there is a price to pay for it! For most applications, data coherencies can lead to faster, smaller, and more efficient memories. Semi-random access memories are finer tuned to the application and therefore more effective. A case in point are the ***multiple ports*** of the PS on-chip memory.

A Implementation Details

A.1 Architecture Details

m, the number of pixels per processor $= 16$
n, the number of processors per chip $= 16$

Pixel Data Format:

r, g, b	$= 8$ bits each
z	$= 16$ bits

Therefore the depth buffer stores 40 bits per pixel. This corresponds to:

40×16	$= 640$ bits per memory bank, and
$16 \times 40 \times 16$	$= 10,240$ bits of memory per chip.

1 processor cycle $= 48$ system clock cycles.
The chip was designed to run at clock rates of 40 Mhz and over.

A.2 System Performance Numbers

$p = 48$ clock cycles per pixel
$c = 40$ MHz
$X = 2^{10}$
$Y = 2^{10}$
$N = 16$
$v = 6$ clock cycles per pixel

the best pixel rate $= cXY/(pN) \approx 5.4613 \times 10^{10}$ pixels/sec.
the worst pixel rate $= c/p \approx 833,333$ pixels/sec.
the best polygon rate (current architecture) $= cY/(pN) \approx 5.3 \times 10^7$ polygons/sec.
the best polygon rate (modified architecture) $= cY/p \approx 8.533 \times 10^8$ polygons/sec.
the worst polygon rate $= c/(pN) \approx 52,083$ polygons/sec.
the best frame rate $= c/(vN^2) \approx 26,041$ frames/sec.
the row pixel-bus frame rate $= c/(vXN) \approx 407$ frames/sec.
the column pixel-bus frame rate $= c/(vYN) \approx 407$ frames/sec.

the worst case row pixel-bus latency (current) $= (Xp/N + p + XNv)/c$ seconds.
$\approx 2.5356 \times 10^{-3}$ seconds.

the worst case row pixel-bus latency (modified) $= (Xp/N^2 + p + Xv)/c$ seconds.
$\approx 1.596 \times 10^{-4}$ seconds.

A.3 Chip Details

Technology: 3 micron cmos/bulk
Silicon Vendor: MOSIS

Thus the 3-transistor per bit dynamic memory requires $16 \times 40 \times 16 \times 3 = 30,720$ transistors.

Chip size: $9mm \times 4mm$
Pin count: 18
Transistor count: 50,000

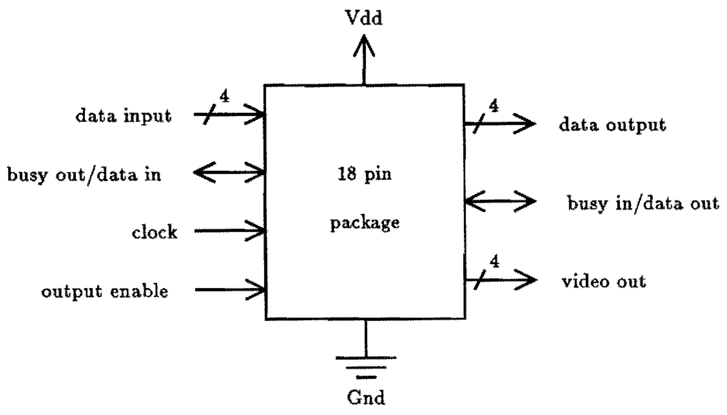

Figure 9: The PS Chip

References

[Demetrescu 85] Stefan Demetrescu. High speed image rasterization using scan line access memories. *1985 Chapel Hill Conference on VLSI*, 1985.

[Foley 84] J.D. Foley and A. vanDaam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1984.

[Fuchs 81] Henry Fuchs and John Poulton. Pixel-planes: a vlsi-oriented design for a raster graphics engine. *VLSI Design*, Third Quarter 1981.

[Fuchs 83] Henry Fuchs, John Poulton, Alan Paeth, and Alan Bell. Developing pixel-planes, a smart memory-based raster graphics system. 1983.

[Gharachorloo 85] Nader Gharachorloo and Christopher Pottle. Super buffer: a systolic vlsi graphics engine for real time raster image generation. *1985 Chapel Hill Conference on VLSI*, 1985.

[Gharachorloo et al. 88] Nader Gharachorloo et al. Subnanosecond pixel rendering with million transistor chips. *SIGGRAPH '88 Conference Proceedings*, August 1988.

[Gouraud 71]	H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, June 1971.
[Gupta 87]	Rajiv Gupta. *PS: Polygon Streams, A Distributed Architecture for Incremental Computation Applied to Graphics.* Master's thesis, California Institute of Technology, 1987.
[Hayes]	J.P. Hayes. *Computer Architecture and Organization.*
[Leiserson a]	Charles E. Leiserson. *Area-Efficient VLSI Computation.*
[Leiserson b]	Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1).
[Obrebska]	Monika Obrebska. Comparative survey of different design methodologies for control part of microprocessors. *VLSI Systems and Computations.*
[Poulton et al. 85]	John Poulton et al. Pixel-planes: building a vlsi-based graphic system. *1985 Chapel Hill Conference on VLSI*, 1985.