A High Performance Renderer for Advanced 3D Graphics Algorithms

Avi Bleiweiss
Stardent Computer
Sunnyvale, California

Tom Diede
Stardent Computer
Sunnyvale, California

ABSTRACT

This paper introduces a rendering engine that raises the quality level of real-time rendering. The renderer combines fast polygon rendering capability with robust performance for many of the popular advanced 3D rendering algorithms. Performance of many of the advanced rendering functions is less than a basic polygon rendering rate, but only by a factor of two to four. The advanced rendering functions supported include anti-aliased lines, texture mapping, transparency, compositing, spheres with specular highlights, general anti-aliasing of polygonal images, shadowing, CSG display, and radiosity acceleration. A rather unique microcode approach at the pixel processing level, combined with a cost effective pixel memory structure are the real driving force behind the renderer. This rendering engine has been incorporated into the overall architecture of the Stardent 3000 Graphics Supercomputer, where it is tightly coupled to system memory and to a scalable scalar and vector floating point compute processors.

KEYWORDS

hardware architecture, parallel processors, pipeline processors, shading and texture, visible surface algorithms, microcode pixel programmability, raster display devices

ACM Reference Format:
Avi Bleiweiss and Tom Diede. 1990. A High Performance Renderer for, Advanced 3D Graphics Algorithms. In *Proceedings of ACM SIGGRAPH conference (SIGGRAPH'90).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

During the past few years the thrust of high performance rendering architectures have been mostly directed toward achieving faster polygon rendering rates (100KQuadrilaterals/sec) [1, 2, 6]. We have also observed a growing interest in the, application of advanced rendering techniques with some ported to high end workstations. Some of the systems though have the tendency to exhibit significant performance degradation, which often may prohibit the use of the complex rendering function. The scope of this paper is to demonstrate a hardware solution to the advanced rendering problem at the pixel level.

1.1 Background

We will examine current approaches to the design of high performance renderers in light of parallelism, pixel memory structure, and programmability, focusing on the requirements to support advanced rendering algorithms.

SIGGRAPH'90, August 1990, Dallas, Texas USA
1990. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

The most effective approach to the distribution of pixel processing power has been a SIMD array of processors operating on a virtual pixel space [2, 8]. In this approach, processors share common control store and a small common scratch pad. Pixel memory bandwidth requirements are addressed either by having the processor driving a small memory locally [8], or by sharing system memory resources via a very wide system bus [2]. Executing the same instruction out of a common control store is efficient enough as long as image coherency ts maintained. However, processor utilization begins to suffer when a function requires non–coherent memory mapping operations. Some examples are: pixel distances to line centers, perspective texture mapping, specular shading for quadratic primitives, and non-aligned bitmaps.

Advanced rendering capability translates very quickly to a high bits per pixel figure. As such, memory structure and allocation has a predominant influence on the renderer cost. In a global sense, pixel memory is partitioned into a rendertng buffer and a viewing buffer. The use of DRAM based system memory as a virtual rendering buffer combined with a single buffer, VRAM based, video buffer [8] provides favorable reduction in memory cost. However. some of this cost advantage is diluted because of the need for a very wide bus to system memory.

Pixel level programmability, generally takes the form of downloadable microcode capability [8]. Its major contribution to a system is to provide a graceful path for newly developed algorithms, as they become available. It is well known that pixel level functions tend to have a very wide option spread. Therefore, the realization of pixel based microcode in a SIMD architecture, where code is straightforward with minimal branching, results in many versions of nearly identical code.

The renderer architecture we will describe below, solves the advanced rendering problem by providing high processor utilization over a broad range of rendering operations, while capturing the advantage of DRAM based rendering memory, without the need for a very wide bus to system memory, and by using a unique compact ROM based microcode control that still provides a rich set of graphics functionality.

1.2 Design Goals

Our main goal for the renderer was to achieve basic rendering rate of 200KQuadrilaterals/sec and 500KVectors/sec, and at the same time support a variety of rendering algorithms operating as fast as possible, consistent with technological constraints and within reasonable economic margins (a single board solution). This goal was further subdivided into the following five implementation aspects:

- Simplicity: all rendering functions must be built on top of a simple, robust, and cost effective hardware algorithm foundation. This foundation must provide a high level of fidelity in terms of drawing accuracy and consistency.
- Balance: renderer performance has to match graphics pipeline geometry computation performed by its compute engine peer, in both anti-aliased lines and polygon rendering.
- Quality: anti-aliasing of lines and polygon outlines must be of high quality and retain performance of greater than the basic rendering rate divided by two.
- Functionality: advanced rendering functions (mentioned above) must be performed at a rate greater than the basic rendering rate divided by four.
- Growth: it must provide a growth path for the future with regard to improved functionality and new algorithm support.

The paper proceeds with an overview of the algorithm foundation used to implement the rendering functions, followed by a detailed description of the overall architecture implementation. We conclude with a presentation of performance analysis details.

2 ALGORITHM FOUNDATION

Our hardware implementation was driven by algorithms. The algorithms are reviewed next in a very broad sense and their key baselines are discussed. The image produced as a results of a photo-realistic rendering on our hardware is depicted in Figure 1.

2.1 Primitive Representation

The backbone of the supported algorithms is the linear function representation: $f(x, y) = Ax + By + C$. This representation applies to both primitives such as line, polygon, and sphere, and also to edges and attributes, for example, color and depth. The simplicity and robustness of this representation, reduces pipeline setup code and provides a highly desirable rotational invariant shading model in image space. The linear representation was extensively enhanced by the implementation of elaborate boolean functions that support the anti-aliasing of lines or polygon outlines and the drawing of wide lines (further discussed in Section 4).

A bounding box is an integral part of each primitive (Figure 2). It forms a geometric domain that bounds the area for which pixels are scanned in order to be touched. Furthermore, the bounding box notion has a global effect on overall system performance, particularly when advanced functionality is required. Intersection tests against rectangular areas are simpler and faster, and improve the efficiency of pipeline clipping code. They allow for primitive pre-pruning when a composite operation of disjoint objects is performed and finally, a pixel intensive compute operation can be limited to an active rectangular area (a result of primitive bounding box unionization) as opposed to computing over the entire view port.

2.2 Scanning

Two major considerations drove our scanning implementation strategy: parallelism at the geometric domain level (in addition to pixel interleaving) and primitive specific scan efficiency.

Our design has the notion of synchronized scanning in three geometric domains: primitive, bounding box, and attribute. Pixels to be touched are determined in the primitive domain traversal. Here, edge functions are evaluated to classify each pixel as being 'in' (inside the primitive), as being 'on' (on a primitive edge), or as being 'out' (outside the primitive). Bounding box scan computes pixel memory addresses at each scan step. This scan algorithm views all four bounding box edges as being traversal clip boundaries. Hitting a clip boundary causes either a traversal turn back or a traversal completion event. Attribute domain scan, applies to primitive patterning and bitmap operations, where a table index is computed. This scan algorithm is identical to the bounding box scan, where pattern offset and pattern repeat count are used as clip boundaries and form a flexible pattern mapping onto a primitive.

Scan efficiency is addressed by implementing three scan modes (illustrated in Figure 2): staircase (for line drawing), bi-direct (for polygon, sphere, and rectangle fill), and scan-line (for copy and bitmaps). In the staircase mode 'on' and 'out' information combined with the knowledge of being to the 'left' or to the 'right' of the line [10] sets the scan direction.

2.3 Generalized Alpha

Advanced rendering algorithms require generalized context with regard to alpha channel information. Current restrictive use of the alpha channel solely for the presentation of pixel coverage (anti-aliasing and compositing) has been extended in our design. In addition, we provide application specific content such as transmission factor (transparency), depth-count (shadowing), and boolean flags (texture and CSG).

2.4 Multipass Rendering

Refined progression of image generation when using advanced algorithms is very fundamental to the renderer design. It operates in a succession of rendering stages, each composed of one or several draw phases followed by one or several compute phases. The nature of the draw and compute phases are application specific. In general, the full scene is always traversed in each of the draw phases, and in the compute phase, a graphic operation applies to all pixels of the active scene bounding-box.

3 ARCHITECTURE

The support of advanced rendering, raises a twofold memory bandwidth issue. The first one is at the renderer to system memory link where copy speeds are critical for some of the multi-pass functions. The second is the classical pixel level where we opted to select an interleaved scheme that allows extensive use of page mode references. The rendering engine consists of a matrix of five by four pixel processing nodes, each handling every fifth pixel on every fourth scan line, that are driven by four high performance graphics IO processors (IOPs) (Figure 3). The renderer is implemented as a DMA module in the Stardent 3000 system and performs 64-bit word read and write transactions over the S_bus [6]. The IOPs interface to the S_bus on one end, and drive four pixel buses - one per same scan line interleave - on the other. The S_bus bandwidth is 128 MBytes/sec and the aggregate pixel bus bandwidth is 256 MBytes/sec. Each pixel processing node is comprised of an integrated pixel evaluator (IPE) tightly coupled to a rendering buffer - draw buffer, and a viewing buffer - display buffer. The draw buffer

Figure 1: Photorealistic rendering on graphics hardware using the Dynamic Object Rendering Environment (DORE): showcases global illumination, procedural textured objects, multi-pass full scene anti-aliasing, true transparency, and shadows (umbra and penumbra)

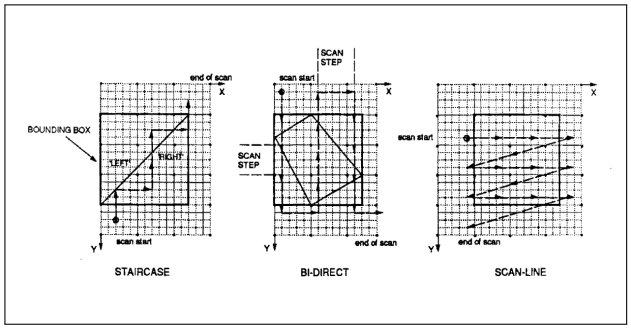

Figure 2: Scan modes

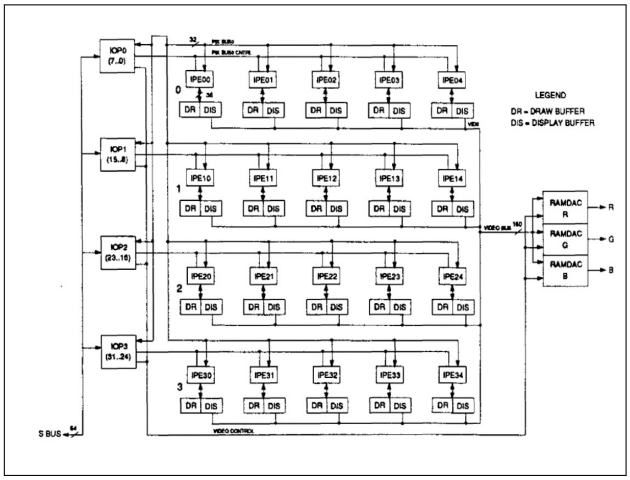

Figure 3: Renderer architecture overview

is DRAM based and the display buffer is implemented with VRAMs. Once a given instruction is received by the pixel processing nodes, each IPE executes its own microinstructions out of its local control store. As such. the renderer effectively becomes a MIMD array of processing elements. Each IPE is associated with a programmable Id that defines its physical interleave position.

The software interface is simple and encompasses twenty compact high level graphics instructions. Their format provides minimal setup at the pipeline level. Graphic instructions have a generic form consisting of a command packet followed by an optional pixel data structure. In system memory, a scene is constructed by generating a sequence of instructions stored in a command buffer. Pixel data blocks referenced by command packets are stored in a separate data buffer. This decoupling of command and data buffers establishes a very efficient way of performing area copies between system memory and pixel memory. Essentially, pixel data is in virtual space and the associated command packet maps it to physical screen space. Mapping of pixel data to the interleaved pixel memory structure is entirely transparent to the application programmer.

The video-out section is composed of three RAMDACs - one per color channel, each having five colormap tables - that are controlled by the IOPs. Various video and stereo display modes are controlled by an integrated video timing generator.

Pixel memory organization and the two ASICs which are used to implement the rendering engine are further discussed.

3.1 Graphics IO Processor

Architecture overview of the IOP is depicted in Figure 4. High performance area copy between system memory and pixel memory is accomplished via the IOPs' four channel DMA controller, along with local bidirectional fifo memory in its data paths. This allows the renderer to operate at the S_bus bandwidth limit for most area copy operations. The four channel DMA controller has the ability to transfer four raster scan lines in parallel by updating four pixel data buffer pointers simultaneously.

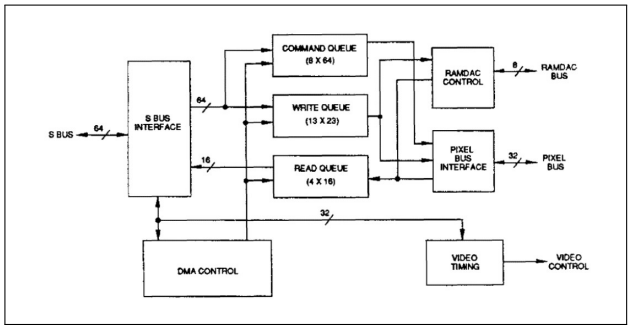

Figure 4: Graphics IO processor

A well established, high bandwidth link between system memory and the renderer allows for an efficient incorporation of objects rendered by alien platforms or images generated by new techniques. Typical pixel data sources include raster devices, surface rendering, and ray-tracing accelerators, or any other random point modeling algorithms. The DMA controller provides a flexible backdoor interface by supporting the following pixel data formats:

- 2D, unpacked: 32 bit word per pixel ('rgba')
- 2D, packed : 32 bit word per multiple pixels (bitmap, 8 bit)
- 3D, virtual : 64 bits word per pixel ('z', 'rgba'J
- 3D, physical: 96 bits word per pixel ('xy', 'z', 'rgba')

The pixel bus is a 32 bit bi-directional bus that operates in two distinct modes: broadcast and data dependent. In broadcast mode, each IPE receives the same instruction and parses command packet parameters (such as bounding box boundaries and linear function coefficients) to their location internally. Upon completion of parsing, the instruction gets executed without further external synchronization requirements. Each IPE has a two deep instruction queue that provides full overlap between current instruction execution and next instruction broadcast. In data dependent mode, subsequent to the command packet reception, the IPE synchronizes with the IOP on a per pixel data transaction.

The graphics IO Processor ASIC is implemented in 1.5 micron Compacted Array CMOS gate array. It contains 38000 gates in a 299 pin package.

3.2 Pixel Memory

Pixel memory organization is scalable and evolves from a basic two dimensional array, defined as a quadrant. Quadrant array size is 1280 by 1024 by 36 bits. The draw buffer consists of four quadrants (2560 x 2048 x 36) and the display buffer is configurable: one quadrant for a 256K VRAM system, or four quadrants for a 1M VRAM system. The total number of bits per pixel available on our design is either 180 or 288, depending on system configuration. IPE pixel memory quadrants and pixel format are shown in Figure 5.

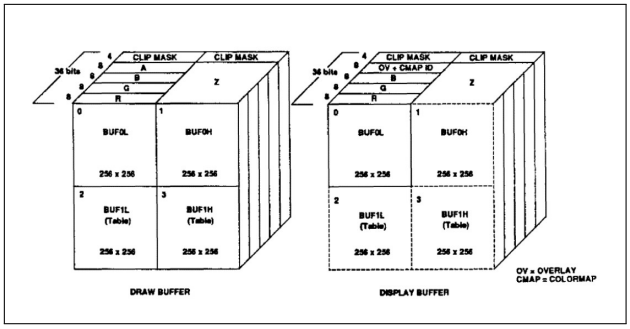

Figure 5: Pixel memory organization

The accommodation of advanced functionality requires the extension of the basic pixel memory space utilization (double buffered color, depth and alpha). In our design the extensions ere:

- Complex math at the pixel level: expanded functionality implies broader pixel arithmetic scoping, beyond the typical multiply_accumulate capability. Because pixel memory bandwidth is inherent to the interlaced nature of the architecture, required math functions such as inverse (texture), square root (sphere), and power (specular shading) are efficiently implemented by means of a lookup table.
- Table locality: the performance of mapping applications (patterning, filtering, texture, and picking) are greatly enhanced when supported at the pixel level. Also, sufficient local table storage can significantly reduce context switching overhead.
- Image buildup buffer: the notion of refined progression in image generation is implemented via a temporary image buffer (for color, alpha, and depth). A partial composite or the partial integration of an image is incrementally accumulated in this buffer, as rendering progresses.

From an application view point, pixel memory allocation (for 2D or 3D images, tables, pixmaps, and temporary image buildup) is at the quadrant granularity level. The only exception is that the 'z' and 'rgba' quadrants have to be horizontally aligned to allow for a fast 3D page mode based memory reference.

3.3 Integrated Pixel Evaluator

IPE hardware (Figure 6) is optimized for drawing quadrilaterals. Lines, triangles and spheres are natural subsets, because they all use a single or multiple edge function evaluators. Our implementation has eight parallel linear function interpolators: one for each edge, one for each color (r, g, b) and one for depth. Each interpolator incorporates an adder, accumulator, and an operand muxing scheme that allows for an interleaved (5A or 4B) or non-interleaved (A or B) scan step in either the x or y directions. Interpolator width is

optimized to its required accuracy (for example, the depth interpolator is 46-bit wide and yields a final 32-bit 'z'). A dedicated boolean logic module provides the interpolator control function which performs primitive interleaved scan, non-interleaved adjustment to IPE id, line width control, endpoint detection for lines and polygons, perpendicular distance computation, and color and depth clipping.

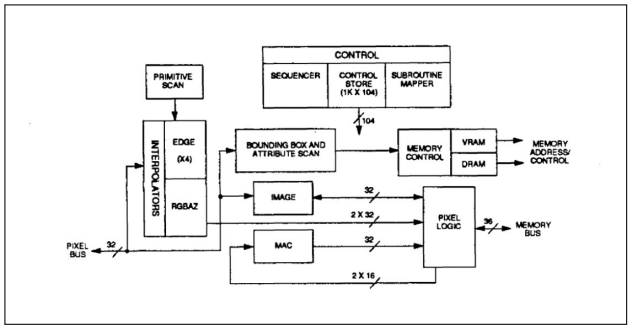

Figure 6: Integrated pixel evaluator

Bounding box scan logic is comprised of symmetric pixel row and column address generation. Bounding box and attribute clip circuitry provide end of scan line, scan direction reversal, and scan termination control. Address generation is synchronized with a variety of pixel memory control sequences. These control sequences include scalar and vector (page mode) write, read, and read-modify-write, as well as flash write and data transfer transactions.

A centralized multiplier_accumulator (MAC) unit - 16 x 16 bit multiplier followed by a 16 bit adder - operates on multiple, on-chip generated source data and on pixel memory destination data. Eight, twelve and sixteen bit operand modes are supported.

Pixel logic is the final processing stage before pixel data is presented at the memory data bus (36 bits). This stage performs clip mask, visibility test, plane mask, and boolean and arithmetic operations on source and destination pixel data.

The control implementation of the IPE was a main technological challenge of the renderer. This was evident for both the incorporation of wide scale functionality and meeting performance bounds that were set forth as design goals. The IPE control scheme is microcode based with a unique flow construction. At the IPE level, all graphics instructions are broken into pixel control executables (PCEs). PCEs form a six-stage deep, pipeline control flow. The IPE control scheme is very flexible with the default flow direction shown in Figure 7. Each pipe stage can be either looped or bypassed.

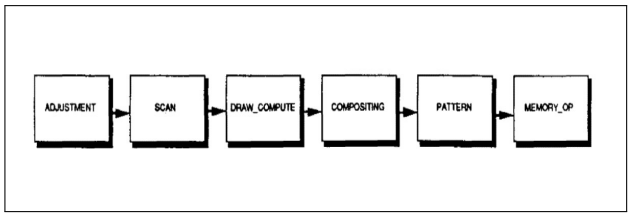

Figure 7: PCE control

IPE Id alignment to the first pixel in the bounding box is performed in the adjustment stage. Since we incorporate free direction line drawing and overlapped area copy, Id alignment is supported at each corner of the bounding box. Subsequently, a parallel scan for both the primitive and the bounding box domains take place. A post attribute scan, when required, always follows the bounding box scan. To iterate, there are PCEs associated with each scan mode, in each domain. The draw_compute stage executes either a featured draw kernel or a compute kernel in one of the multi-pass functions (this is further discussed in section 4.). Compositing covers all binary and unary blending operations as described in [11], with the additional enhancement of pixel multiply operation (c_src × c_dst). Tiles, stipples, and fonts are the patterning executables, and memory_op performs pixel logic functions embedded in pixel memory references.

Video (and memory) refresh events interrupt the PCE control flow. Each PCE has a single or multiple interrupt window openings where an interrupt may be sensed and further processed by specific video refresh PCEs. Video refresh events may occur at any pipe stage and our interrupt service latencies meet the requirements of very high display refresh rates.

Each control pipe stage is represented by a selection of PCEs. All PCEs (initialization, functional, video refresh, and diagnostics) are stored in an on-chip 1K word ROM, where each word is 104-bit wide. The very wide microinstruction word enables fine grain parallelism throughout all IPE hardware functions. Pipe stage overlap is incorporated when feasible. Intra-stage control is performed by a microsequencer, which provides ROM addresses, based on either microinstruction content or an external branch condition. Inter-stage jumps are controlled by a special subroutine mapper which decodes the pixel function to be executed combined with the status of the current pipe stage, and derives the destination jump stage. All jumps are performed in a single clock cycle.

The IPE ASIC is implemented in 1.5 micron standard cell CMOS and exploits a high level of macrocell integration. It contains 52000 gates 1n a 155 pin package (Figure 8)

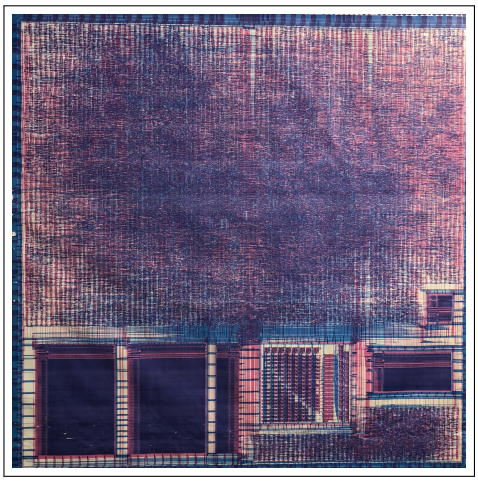

Figure 8: Authentic routing plot of the pixel processor (1.5 micron technology)

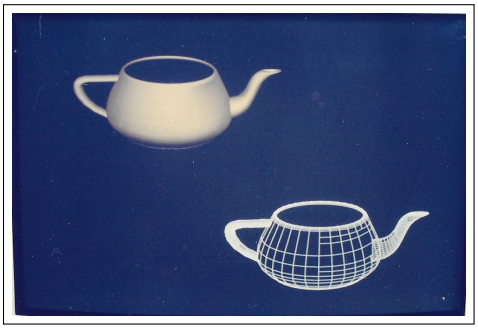

Figure 9: Rendered teapot on hardware in both fill and outline drawing modes

4 RENDERING FUNCTION MAPPING

The algorithm discussions presented in this section focus on pixel level functionality. Application issues are addressed only when they have a direct bearing at the pixel level or at the pixel memory quadrant allocation. Each function, with its full option spread, is mapped onto a concatenation of multiple PCEs. Figure 9 illustrates basic rendering of a teapot in both fill and outline drawing modes.

4.1 Line Drawing

Line drawing utilizes a normalized linear representation for an edge of the form:

$$f(x, y) = \frac{Ax + By + C}{sqrt(A^2 + B^2)}$$

At each pixel, the value of this normalized edge function is the perpendicular distance from the pixel to the edge. This is key to the way lines or polygon outlines are drawn. At each staircase scan step the IPE compares the distance value f to a predefined line width (major axis dependent) parameter and determines 'on' drawable pixels.

Line drawing parameters include width, style, depth-cueing, and anti-aliasing. Wide line drawing and line style are further covered in Section 4.4.

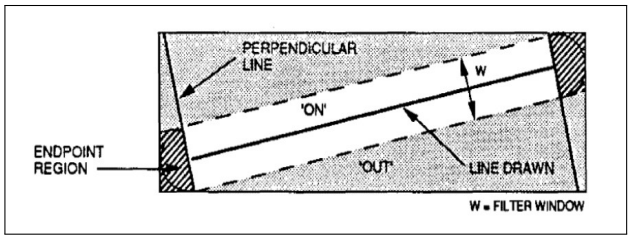

Figure 10: Line anti-aliasing

Depth-cueing and anti-allastng are table driven operations. A superposition of a depth-cue function (linear or non-linear) and a filter function (filter shape and size, line widths, and screen gamma) are contained in a single table. The index to the table is a concatenation of two eight bit values: pixel depth - 'z' - and pixel distance to the centerline. The low eight bits, address the filter function component of the table. The high eight bits, address the depth-cue component of the table. The table occupies 8 bits of depth in one of the pixel memory quadrants. Table entries are used to scale 'on' pixel intensity.

Anti-aliased line drawtng also requires logic for endpoint detection. The renderer has the notion of an endpoint region - the triangle formed by the corner of the line bounding box and a line which is perpendicular to the line drawn at the line endpoint (Figure 10). Here the bounding box is extended in proportion to the filter size in pixels. In anti-aliased line drawing, three edge functions are evaluated in parallel - one for the line drawn and two for the pair of perpendicular lines. Note that pixels at which the perpendicular edge function value is negative are considered to be in an endpoint region. The intensity of endpoint pixels is scaled by a value which is proportional to a radial distance approximation. Line body pixel intensity is scaled in proportion to its perpendicular distance. The table entry used for intensity scaling may optionally be written to the alpha channel for each drawn pixel. These alpha channel values are then available for further image compositing application.

4.2 Polygon Drawing

Planar and convex quadrilaterals (quads) and triangles are the polygonal primitives supported in our implementation. There are three fundamental polygon drawing modes, namely fill only, outline only, and combined fill and outline. Polygon outline and line drawing are identical and share the same level of functionality, such as wide vectors, depth-cueing, and anti-aliasing. Polygon patterning is further covered in section 4.4.

In quad drawing, four (or three for triangles) edge functions are evaluated in parallel. Pixels which are classified as 'on' or 'in' in a bi-direct scan are considered to be drawable. Pixels which are 'on' can be shaded per the preset foreground color or they may be smoothly shaded in the same way as 'in' pixels.

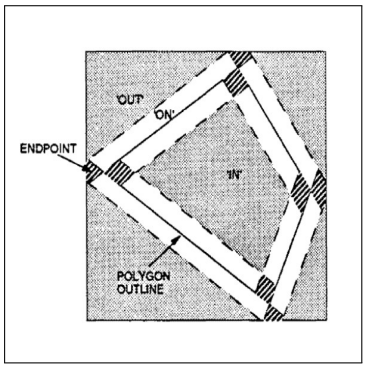

Figure 11: Polygon anti-aliasing

Anti-aliased polygonal outline drawing is viewed as an intermediate step between aliased images and general anti-aliasing (discussed in Section 4.8). This is sustainable both in terms of quality and speed. Polygon endpoint region detection is much more involved when compared to the line endpoint detection task. All possible edge intersection areas are detected and multiple perpendicular distance selection rules are applied. In an endpoint region

(illustrated in Figure 11), the minimum distance value of the two edge functions involved is selected as the filter table index where both functions evaluate to a positive number. On the other hand, maximum distance is picked where both functions have a negative sign. The end result is that undesired endpoint highlighting are avoided.

In the default pixel shading mode, the three color channels are evaluated in parallel, each to a final eight bit intensity level. Additionally, the renderer supports single channel, twelve bit pseudo color rendering, where the interpolated shading value is used to index a table holding 4096 full color (24 bits) entries. A single pixel memory quadrant can store up to sixteen unique color tables that are selected on a per window basis.

4.3 Sphere Drawing

Sphere drawing is based on the algorithm presented in the work by Fuchs et al. [7]. One quadrant of pixel memory is used to store ($x^2 + y^2$). The data from this table is made available as an input to one of the IPE interpolators. This linear edge function combined with the quadratic ($x^2 + y^2$) data, evaluates 'in' and 'out' sphere pixels. The depth and 'r' (evaluates diffuse light term $N \cdot L$) interpolators have access to a square root function stored in another pixel memory quadrant. The square root operator serves both for the visibility test and for the evaluation of a shading table index. Multiple shading tables (up to 16, 4K by 16 bits each), for a variety of diffused and specular light models, are stored adjacent to the square root function in pixel memory. We note that filled circle drawing is a subset of sphere drawing.

4.4 Windowing

We assume an X windowing environment for an application to run, and most of the graphic functions defined by Xllb are directly supported in our hardware. Though not directly related to the subject of advanced rendering, hardware solutions at the pixel level are still required to maintain interactive performance goals, given a user interface environment. The following highlights some of the drawing extensions and window clipping functionality we support:

- wide line drawing: line width could be equal to one, two or four pixels. Endpoint join style is restricted to a non-filled triangular notch. Wide line drawing is primarily applicable to straight horizontal and vertical lines.
- primitive patterning: line style, tiles, and stipples are table oriented operations. Up to eight two dimensional pattern arrays of 32 by 32 pixels, 32 bits each, are stored locally in pixel memory. All four Xlib fill styles are supported namely solid, stippled, opaque_stippled, and tiled. The application task for primitive patterning is limited to pattern table selection and to the specification of offset and repeat count in the pattern domain.
- bitmap: the application of bitmaps apply to fonts, icons and images. Bitmaps can be non-aligned with regard to either word or bounding box boundaries. Bitmap color assignment is implemented in a stippling manner where stippled and opaque_stippled mode are supported.
- clip mask: primitive clipping to window boundaries is handled by comparing pixel clip mask to a predefined, registered destination window Id. Area copy masking is per either a source window Id, a destination window Id, or both.

4.5 Compositing

The PCE structure is very flexible in providing a choice for selecting source data for compositing. It can be either an image data residing in system memory or interpolated data, derived at the IPE level. Compositing arithmetic is an eight bit operand mode and can be performed under visibility test. The availability of pixel multiply operation allows for a unique intensity scaling per color channel (shaded texture).

4.6 Texture

We provide a broad range of support for texture mapping. The baseline for the implementation assumes screen order scanning (inverse mapping) and perspective projection. In texture mode, linear representation of texture space coordinates (u, v) replaces the default color representation as part of a polygon instruction. Two out of the three IPE color lnterpolators, linearly evaluate texture coordinates as the first part of the inverse mapping process. Supported texture modes of operation are:

- point sampling only, local texture: perspective inverse mapping requires a division by 'z'. This is accomplished by storing an inverse function in one of the pixel memory quadrants. Furthermore, in this mode texture values are locally stored in another pixel memory quadrant. Each texture pixel consists of 32 bits and may include an alpha value for further compositing. The user picks one of two texture table configurations:
  – single texture, 256 by 256 pJxels
  – multi-texture, up to seven tables, each of 64 by 64 pixels
  Texture space is point sampled only and no filtering is performed .
- filtered texture using summed area lookup [5] in system memory: in this mode the renderer performs inverse mapping and stores texture coordinates (u, v) in one pixel memory quadrant. Subsequently, the Stardent 3000 vector processor scans the image bounding box, retrieves texture coordinates and applies repeated integration filtering in a summed area lookup table form. Filtered texture pixels are then copied from system memory to the pixel memory. A destination shaded image is textured by applying the pixel multiply operation to a source filtered texture image. Maximum texture resolution in this mode is 4K by 4K pixels.
- 2D lookup: in this mode texture coordinates have a broader application sense. They represent generic scene attributes (such as temperature, stress, pressure etc.) which are further superimposed on a rendered image. It is assumed that they have a linear behavior in a polygonal extent. The renderer provides a two virtual axes interpolation mode where interpolated values are combined to serve as an index to a color lookup table (64Kx24), stored locally in pixel memory.

4.7 Transparency

The underlying transparency model assumes that scene surfaces are associated with transmission coefficients to form transparency

layers. Each image pixel is covered by several layers and the final shading value at that pixel is an integration of layer shading contributions. The contribution of each layer is proportional to the sum of transmitted light from a front closest layer combined with the reflected light of the layer examined. The integration process terminates when an opaque surface is hit (alpha = 0).

Multi-stage transparency rendering is performed in a succession of stages, each composed of a closest layer evaluation draw phase, followed by a shading integration compute phase. An IPE special visibility test operator evaluates closest layer by comparing pixel current depth to both the last depth encountered in the current draw phase and to the last depth that was computed in the previous draw phase (stored in the buildup buffer).

At each pixel the shading contribution is computed by the IPE MAC and gets added to the partial shading value stored in a buildup buffer, by applying the pixel add operation.

4.8 General Anti-Aliasing

A general anti-aliasing rendering stage is comprised of a draw phase followed by a filter phase. For each draw phase, the image is generated with a sub-pixel offset, which provides adequate sampling of a pixel area. The filter phase, adjoined to a draw phase, computes sub-pixel intensity contribution and integrates the result to the partially integrated image. Filter arithmetic is handled at the IPE MAC and assumes 16 bits for both the normalized filter coefficients and the pixel color channel. The number of rendering stages are determined by the filter kernel area in pixels. A final normalization phase truncates the 16 bits to 8 bits color channels.

4.9 Shadowing

The supported shadow algorithm is based on the generation of shadow volumes [4]. Image generation flow is per light source and is broken into three parts: a model draw phase, a shadow draw phase, and an intensity compute phase. In the shadow phase, the application flags the front and back facing polygons and respectively, a pixel depth-count attribute is incremented or decremented, based on a visibility rule. A search for shadow pixels having positive depth-count and a subsequent intensity attenuation operation are incorporated in the compute phase. Both depth_count updates and intensity attenuation are performed by the pixel logic stage of the IPE.

A final ambient thresholding, by applying the pixel 'max' logic rule, and a specular light component integration are shading quality options that may be performed as a post shadow generation pass.

4.10 CSG Display

The display of CSG defined objects, requires the support of set boolean operations at the pixel level. A normalized CSG tree is assumed in which paring convex objects are our supported primitives. The tree is broken into subtrees that are unioned together and set operations at each node are either intersection or difference [9]. The display of a normalized CSG tree is a recursive process of primitive paring where a composite image is built up incrementally.

CSG display is implemented as a multi-stage rendering process where each stage is composed of four phases:

- First object draw phase: front face (if intersection) or back face (if difference) of a pared object is rendered, as a bounding box image is saved.
- Rest of objects draw phase: front and back faces of rest of objects that have a bounding box overlap with the first paring object bounding box, are drawn. Boolean flags are set per a visibility test, per pixel.
- Boolean phase: pixels in the above defined bounding box are retained or removed if the set boolean operation is intersection or difference, respectively.
- Union phase: a partial composite is being stored in the image buildup area.

The number of rendering stages for the entire composite is equal to the number of objects in the scene.

4.11 Radiosity

The renderer is used as an accelerator for the hemi-cube rendering part of the form-factor computation [3]. Delta form-factor summation and the solution of the radiosity equation are both performed by the system vector floating point processor. The hemi-cube resolution in our Implementation is 128 x 128. Hemi-cube rendering assumes that polygons are 'flat shaded' and the foreground register holds object identification (ld) type of data. With the assumption of no more than 64K objects per scene, we can pack the data for two five faced hemi-cube cells in one draw buffer quadrant. Therefore, herni-cube cell data (object Id) is transferred to system memory at a rate of 64 MCells/sec for further processing.

5 PERFORMANCE

Several aspects of performance are illustrated. The first represents real vectortzable pipeline code that was written for the Stardent 3000 and reflects how well the rendering engine matches its floating point compute peer in some basic line and polygon drawing cases. Then, we examine how the engine handles generic system overheads in an animation application and finally, we characterize rendering performance over a broad range of graphics and imaging functions.

Table 1 shows basic graphics performance numbers for two Stardent 3000 system configurations: a two and a four processor cluster. Pipeline code includes transformation, clip test, shading (single directional light, diffused only), perspective divide, scaling, integer conversion, and engine setup tasks. All our rendering figures assume full color, depth buffering on, and window clipping. Numbers in the right hand column relate to stand alone rendering rates. These numbers are by no means peak figures and they rather reflect average performance of ten primitives, each with a different primitive to bounding box pixel ratios.

In an animation type of application, each frame is encountered by a post rendering area copy overhead due to the separation of draw and display buffers. This copy is performed at a rate of 156MPixels/sec (100MPixels/sec under clip mask). Therefore, for a typical 15 frames/sec (frame = 1280×1024 pixels) animated movie, the expected system draw efficiency,

$$((\text{frame time} - (\text{clear} + \text{area copy}))/\text{frame time})) \times 100$$

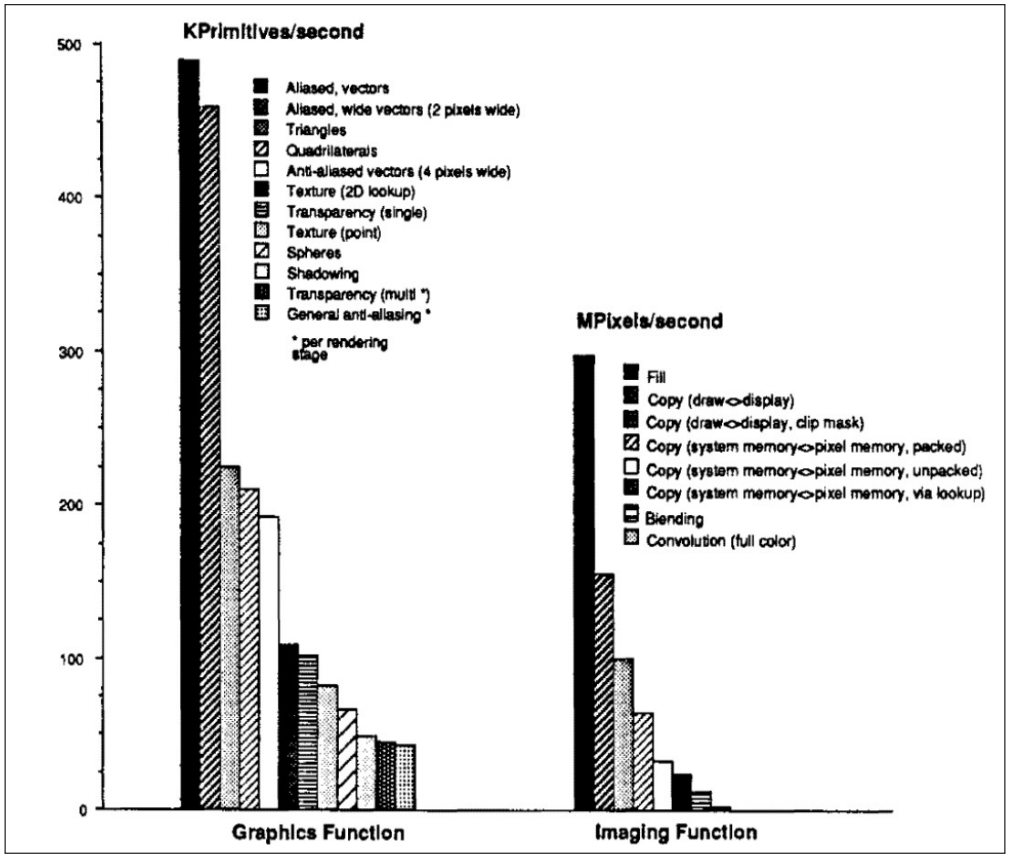

Figure 12: Wide scale rendering performance

Table 1: System basic rendering performance

Benchmark	Size	Metrics	2-Processor System	4-Processor System	Standalone Graphics
depth-cued polyline	10-pixel	KVectors/sec	225	400	490
depth-cued, anti-aliased lines	10 pixel	KVectors/sec	192	192	192
smoothly shaded triangles	100 pixel	KTriangles/sec	145	225	225
smoothly shaded quads	100 pixel	KQuads/sec	92	163	210
smoothly shaded spheres	10 pixel	KSpheres/sec	66	66	66
clear (color and depth)	1M pixels	MPixels/sec	150	150	150

will be 75 percent. This is translated to a rendering capability of 10K polygons per frame.

Figure 12 shows performance behavior for a wide function scale. These functions were simulated on our hardware modeling environment where final PCE code was in place, already loaded. This model was written in Verilog VHDL and did include gate level (ASICs) and functional level (S_bus, system memory, and pixel memory) components. Note that for the pixel intensive functions the renderer is expected to be the performance limiting factor. Therefore, the associated measurements effectively reflect system performance figures.

6 CONCLUSIONS

The renderer architecture was presented in light of its algorithm foundation, architecture, implementation aspects and performance. The renderer forms a twofold well matched architecture. Internally, IPE processing power equates pixel memory bandwidth by widely exploiting page mode references. Furthermore, as a renderer which coexists with a floating point vector processor peer, a balanced system performance over a broad range of graphic functions has been accomplished.

There are certain implementation limitations associated with the renderer design. The limitations and possible improvement extensions are further discussed:

- Multi-feature type of application: an application that requires pixel memory space of more than four quadrants to draw a frame is prone to context switching overhead (moving data to and from system memory) when running on a 256K VRAM configured system. However, a 1M VRAM based system can handle most of the feature concatenation options with minimal paging.
- Programmability: although we provide support for a very rich rendering function base, the possible need for future user algorithm enhancement is plausible. Because the IPE is microcode controlled, we observed that a small additional RAM which stores one PCE per pipe stage could be added to the existed ROM and form a user microcode space. This extension will most likely require a geometry shrink step in VLSI technology.

Interface simplicity at both the graphics software level and the IPE level, provides a smooth scalable path for either high or low end type of applications. A Stardent 3000 system can be configured with up to two rendering engines, one per seat, for the high end target, and a reduced matrix size of IPEs can serve the low end market very well.

ACKNOWLEDGMENTS

We would like to thank: our team colleagues Micheal Sleator for architecting the IOP and Zaigham Ahsan for implementing the IPE microcode; Kevin Weiler, Mike Kaplan, and Bruce Borden for their helpful ideas during the design period. Special thanks to Paul Ausick for his help in the preparation of the final copy for publication.

REFERENCES

[1] Kurt Akeley and Tom Jermoluk. 1988. High-performance Polygon Rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88)*. ACM, New York, NY, USA, 239–246. https://doi.org/10.1145/54852.378516

[2] Brian Apgar, Bret Bersack, and Abraham Mammen. 1988. A Display System for the Stellar Graphics Supercomputer Model GS1000. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88)*. ACM, New York, NY, USA, 255–262. https://doi.org/10.1145/54852.378518

[3] Michael F. Cohen and Donald P. Greenberg. 1985. The Hemi-cube: A Radiosity Solution for Complex Environments. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/325334.325171

[4] Franklin C. Crow. 1977. Shadow Algorithms for Computer Graphics. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '77)*. ACM, New York, NY, USA, 242–248. https://doi.org/10.1145/563858.563901

[5] Franklin C. Crow. 1984. Summed-area Tables for Texture Mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. ACM, New York, NY, USA, 207–212. https://doi.org/10.1145/800031.808600

[6] Tom Diede, Carl F. Hagenmaier, Glen S. Miranker, Jonathan J. Rubinstein, and William S. Worley, Jr. 1988. The Titan Graphics Supercomputer Architecture. *Computer* 21, 9 (Sept. 1988), 13–28, 30. https://doi.org/10.1109/2.14344

[7] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. 1985. Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-planes. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/325334.325205

[8] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. 1989. Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/74333.74341

[9] S. Monar J. Goldfeather, H. Fuchs and G. Turk. 1989. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *Computer Graphics and Applications* 9, 3 (May 1989), 20–28.

[10] Juan Pineda. 1988. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88)*. ACM, New York, NY, USA, 17–20. https://doi.org/10.1145/54852.378457

[11] Thomas Porter and Tom Duff. 1984. Compositing Digital Images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. ACM, New York, NY, USA, 253–259. https://doi.org/10.1145/800031.808606