PixelVision Architecture

Revision 4.0

23 October 1992

Prepared by:

Brian Kelleher

Digital Equipment Corporation

Workstation Systems Engineering ☐ Palo Alto, California

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the PixelVision architecture, a parallel graphics architecture appropriate for multimedia workstations. The architecture integrates video technologies including 2D and 3D geometric graphics, photorealistic rendering, stereoscopic viewing, windowing operations, live video, and image processing operations. The PixelVision architecture focuses on defining the minimum amount of special-purpose hardware necessary to dramatically accelerate the video technologies. Furthermore, to optimize workstation price and performance, the architecture tightly integrates the graphics hardware into the base workstation.

In the past, graphics applications have been partitioned into two fundamental classes: geometric applications and imaging applications. The recent emergence of multimedia applications requiring live video services adds a third class. The boundaries between these classes helped to make graphics hardware tradeoffs, allowing the designers to make tradeoffs for the needs of the target application class. However, many applications today blur the boundaries between these categories by integrating the video technologies. As applications mature, they will blur the boundaries further. Graphics hardware must integrate the video technologies to match the needs of graphics applications. Eroding the boundaries between the system components through system integration will result in a lower-cost better-balanced system than otherwise possible.

The tasks of traditional geometric graphics applications can be broken into three functional categories: structure traversal, geometry processing, and rendering. Structure traversal refers to the traversal of an application's graphics data structure, either by the application or by a graphics library such as PHIGS. Geometry processing refers to floating point intensive operations, such as vertex transformations and shading, that convert data from an application's format to a format appropriate for rendering. Finally, rendering refers to the transformation of a geometric description of an image, constructed of vertices and vertex properties such as color, into the more verbose description stored in graphics memory. These tasks dominate most 2D and 3D geometric graphics applications.

Multimedia applications, and other applications requiring video services such as live video, image compression and decompression, and computationally intensive imaging operations, have very different functional requirements than geometric applications. However, these applications require many of the same raw materials as the geometric applications: frame buffer access, floating point processing, CPU processing, and a fast connection between the CPU and video services. The PixelVision architecture integrates the video technologies through proper configuration of raw materials to create a balanced graphics workstation.

The PixelVision architecture is the design of an interface and a set of parallel algorithms and hardware that offer a range of high-performance, low-cost, special-purpose graphics hardware. The terms *PixelVision geometry coprocessor* and *PixelVision rendering subsystem* are used throughout this document to refer to the physical components implementing the PixelVision architecture. The geometry coprocessor is a memory mapped floating point coprocessor with four identical floating point units. The rendering subsystem implements a set of parallel rendering algorithms including smooth shading, depth-buffering, antialiasing, transparency, and constructive solid geometry.

This document deliberately leaves implementation-specific areas vague due to its architectural nature. Refer to implementation specifications for implementation details.

2 Overview

2.1 Geometry Coprocessor Overview

The PixelVision geometry coprocessor integrates parallel floating point functional units with the CPU. Intended applications include the basic 3D geometry pipeline, photorealistic rendering such as ray tracing, NURBS, imaging algorithms such as convolution and FFT, and general application acceleration. The PixelVision geometry coprocessor contains an interface to the CPU's system bus, global instruction RAM and a global register file for data, and four floating point units each with 64 local registers. It is a coprocessor to the CPU in the sense that the CPU controls its instruction stream. The geometry coprocessor logically resides in a prominent position alongside the CPU, not on the other end of an IO interconnect, as shown in figure 2-1.

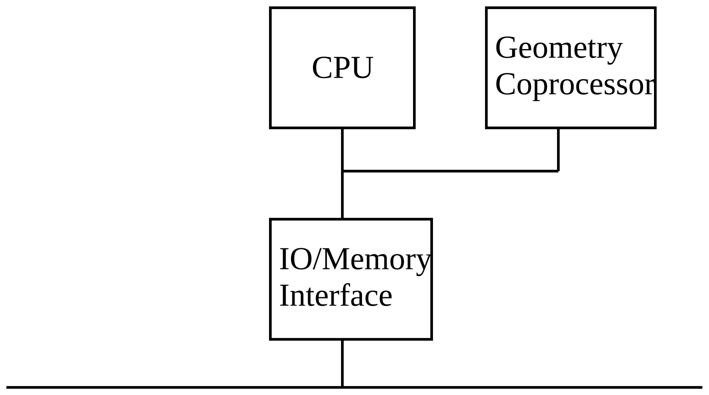


Figure 2-1 Geometry coprocessor in a system

The geometry coprocessor is a shared resource available to all processes. Each process views the geometry coprocessor in its own address space as a dedicated resource. The operating system saves and restores the geometry coprocessor's registers at context switches just as it saves and restores the CPU's registers. The geometry coprocessor appears to an application as 384 registers for data, four KBytes of RAM for instructions, and a set of memory mapped registers to control the coprocessor's operation.

2.2 Rendering Subsystem Overview

The PixelVision rendering subsystem integrates multimedia video technologies including 2D and 3D geometric rendering, photorealistic rendering, stereoscopic viewing, windowing operations, and live video. The rendering subsystem includes a rendering processor and graphics memory system, and may include a video DAC if the monitor is refreshed from the rendering subsystem. All access to the graphics memory system is gained through the rendering processor. Just as was the case with the geometry coprocessor, the rendering subsystem is a shared resource available to all processes.

Although the components comprising the PixelVision rendering subsystem are implementation-specific, this document assumes that a single chip implements the rendering processor -- the PixelVision rendering chip. This document also assumes that a separate workstation module implements the rendering subsystem. These assumptions are for purposes of documentation and clarity only. Note that other system configurations are equally appealing, such as a low-cost implementation with the rendering subsystem implemented alongside the CPU on a single-board system.

Rendering is the process of converting an object from a mathematical representation of color and geometry to a discrete pixel representation in graphics memory. The PixelVision rendering chip renders geometric primitives through parallel execution of SIMD (Single Instruction Multiple Data) algorithms. These parallel rendering algorithms are capable of running at full memory bandwidth, independent of memory size.

The PixelVision rendering chip performs parallel updates on eight pixels, arranged as a 4x2 *update array*. Once again, this arrangement is for documentation purposes only. The rendering algorithms scale to any two-dimensional (MxN) update array. The 4x2 rendering chip described in this chapter can be instanced more than once to achieve larger update arrays. Implementations of the architecture will set practical limits on the size of the update array.

A single rendering chip accesses eight pixels in parallel, each 24 bits deep, yielding a data bus between a rendering chip and graphics memory of 192 bits. Forty eight 256Kx4 video RAMs are required to exhaust the 192 data wires, providing two million 24-bit pixels. Systems using two rendering chips require 96 video RAMs to exhaust the 384 wires, providing four million pixels. Possible uses for this graphics memory include the visible frame buffer, double buffer, depth buffer, and image cache. Figure 2-2 shows a PixelVision rendering module with two rendering chips.

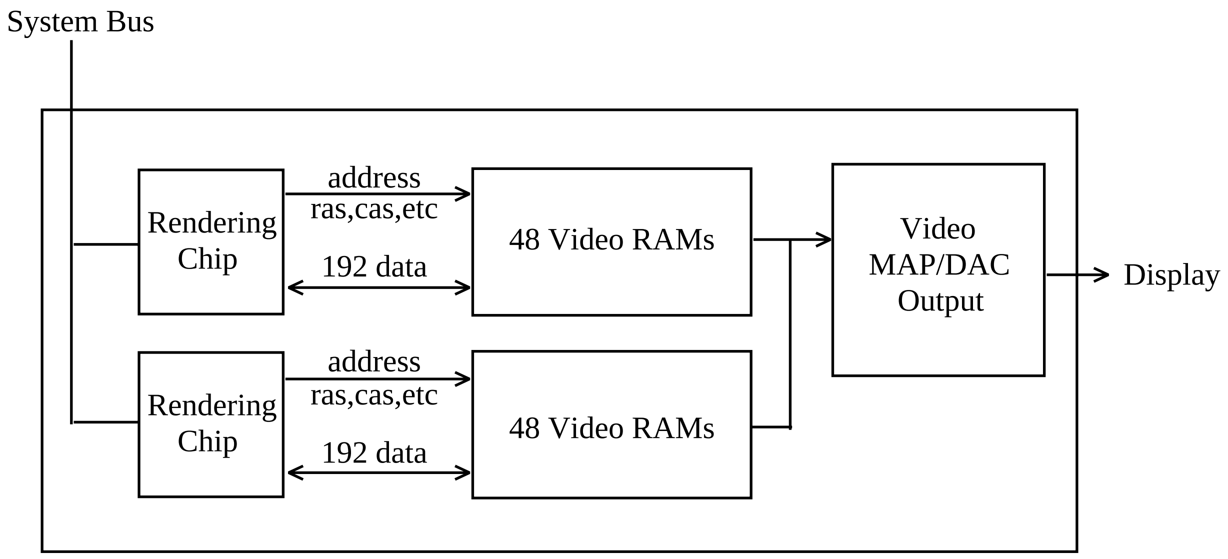


Figure 2-2 PixelVision rendering module with two PixelVision rendering chips

2.3 Rendering Subsystem Architecture Foundations

The PixelVision rendering subsystem architecture draws heavily from several areas of graphics research. Familiarity with some of these areas is helpful in understanding the architecture. The geometric model and parallel algorithms are based on those described in the *PixelStamp Architecture* [1], and derived from the original *Pixel Planes* [2] research at the University of North Carolina. The underlying hardware memory organization derives from the efficient graphics memory access methodology described in *The 8 by 8 Display* [3] research at Carnegie-Mellon University. A brief description of each of these works follows.

2.3.1 Pixel Planes

The Pixel Planes research project constructed a frame buffer of highly specialized VLSI RAM chips. Each word in the RAM represented one pixel of frame buffer data and had a simple associated processor. SIMD parallel algorithms developed for this processor-per-pixel architecture were capable of rendering shaded 3D images from a polygonal database into the color frame buffer.

The Pixel Planes architecture distributed the calculations involved in rendering by assigning a processor per pixel. From the coefficients of linear equations that represent the desired geometry and the (x,y) position of the processor in the frame buffer, each processor was able to compute its relationship to the geometry (inside or outside) and the attributes (color, depth, etc.) of the pixel at that (x,y) position.

The massive parallelism achieved was at the expense of setup and cost. Because of the large task undertaken by the VLSI chips, setting up the internals for each polygon edge was relatively expensive. The chips used serial arithmetic to compute and set up single polygon edges incrementally. Furthermore, the low-density special-purpose nature of the chips made them expensive relative to traditional memory. The result was a prohibitively expensive frame buffer spread across tens of modules. The system was impressive, however, in its ability to render 3D images in real time directly from their mathematical representations.

2.3.2 The 8 by 8 Display

Frame buffer input bandwidth limits graphics performance as main memory bandwidth limits CPU performance. CPU designers solve the memory bandwidth problem by increasing the size of a memory word and by providing high-speed caches. Rendering subsystem designers solve the problem by increasing the size of a frame buffer memory word. A frame buffer memory word is inherently different than a CPU memory word. In particular, linear addressing schemes used for CPUs do not take into account the 2D references common to graphics operations.

The 8 by 8 display described an efficient frame buffer access method symmetric with respect to the coordinate axes. A concept described as the coherence distance is presented, noting that the probability that a given pixel will be affected in a drawing operation is directly proportional to its 2D distance away from any other pixel affected by the operation. Scanline access methods address memory asymmetrically and favor either *x* or *y* aligned drawing operations. As the frame buffer memory word becomes larger, this asymmetry becomes more pronounced. Commonly drawn objects are not 2D asymmetric, implying that the frame buffer memory word should not be either.

2.3.3 PixelStamp

The PixelStamp rendering subsystem is the rendering subsystem of the DECstation 5000 Model 200 PX, PXG, and PXG Turbo graphics subsystems. It combines the rendering model of Pixel Planes with the memory organization of The 8 by 8 Display. The PixelStamp updates in parallel a two-dimensional array of pixels in graphics memory. The size of the pixel array determines the graphics memory organization. PixelStamp's graphics model is similar to that of Pixel Planes, except instead of updating the entire frame buffer in parallel, PixelStamp updates in parallel a single pixel array -- on the order of 4 to 20 pixels.

Thus, to render a triangle into the frame buffer, PixelStamp does it one pixel array at a time. PixelStamp initially references any pixel array within the triangle's boundary, and computes the values of the pixels found there. Using an addressing algorithm based on the convex properties of the geometric figure being rendered, PixelStamp determines another pixel array within the triangle's boundary, and computes the values of the pixels found there. This continues until the PixelStamp has visited every pixel array that is partially inside the boundaries of the triangle. At this point, the figure has been rendered and the resulting image is in graphics memory.

2.4 PixelVision Rendering Model

2.4.1 The Update Array

The PixelVision rendering subsystem can update a single array, called an *update array*, of graphics memory in parallel. An update array is an aligned location (memory word) within graphics memory. The algorithm descriptions in this document use a 4x4 update array as the exemplar, although implementations may differ. An update array consists of an origin located in its upper left corner and sixteen (x,y) offsets from that origin representing pixels. The pixel locations within the update array are called *sites*. Each site contains information relevant to its corresponding pixel.

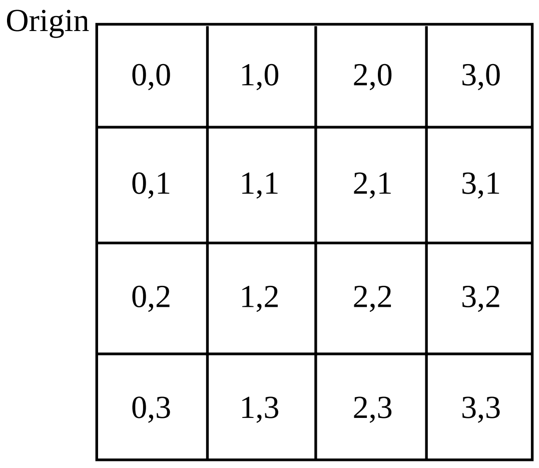


Figure 2-3   Update Array

The two-dimensional representation of the update array, shown in figure 2-3, is useful as a guide to understanding the geometry of the algorithms because there is an intuitive mapping between sites in the update array and pixels in graphics memory. Alternatively, a linear representation of the update array, as shown in figure 2-4, is more useful for describing the hardware implementations of the algorithms.

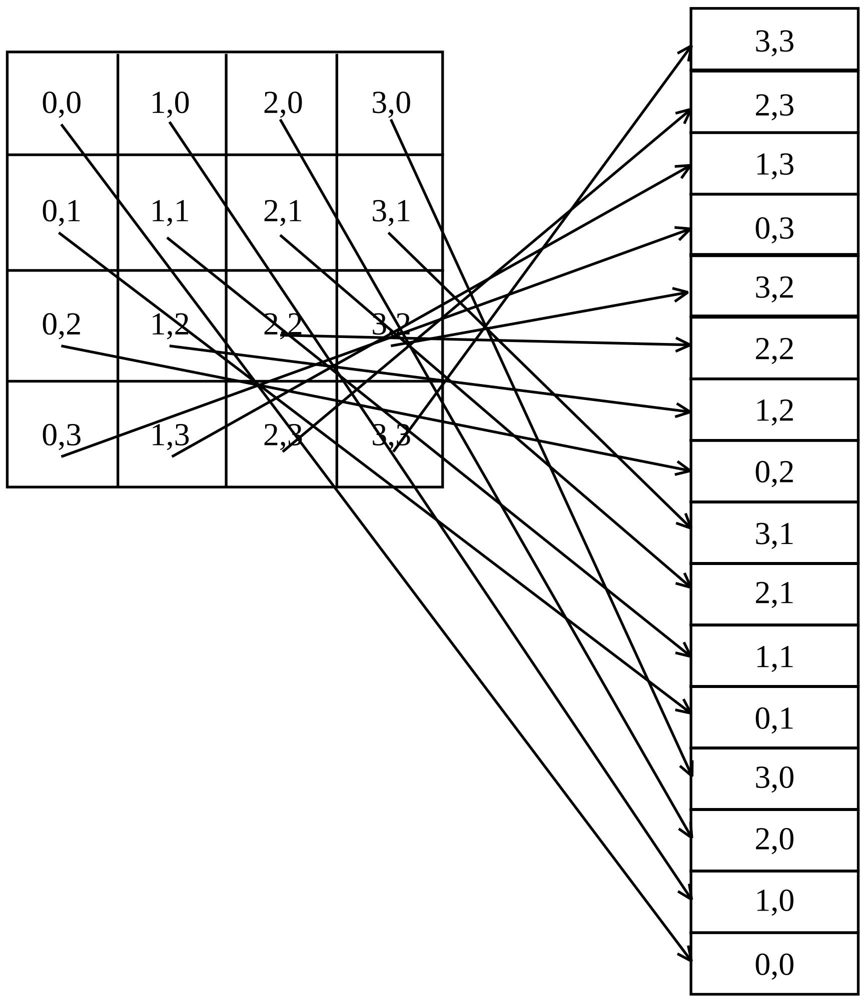


Figure 2-4 Linear representation of the update array

2.4.2 Geometric Half-Spaces

The fundamental geometric primitive in the PixelVision rendering subsystem is the evaluation of a geometric half-space. Any directed line in a plane divides the plane into two half-spaces. The *half-space evaluation* determines on which side of the directed line any point lies. All points to the *right* of the line are in one half-space, while all points to the *left* are in the other.

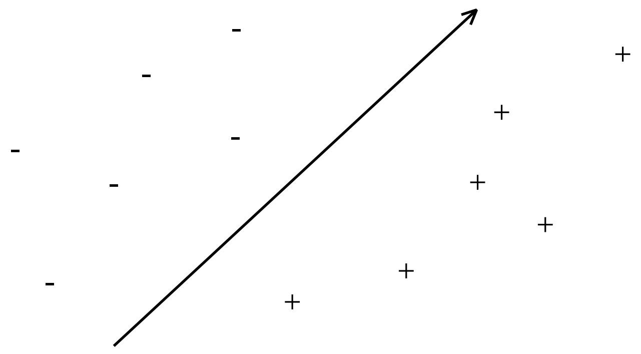


Figure 2-5   A directed line divides a plane into two half-spaces.

2.4.3 Application of Half-Spaces to Graphics

The PixelVision rendering subsystem supports three geometric primitives: points, line segments, and quadrilaterals. These primitives were chosen because of their geometric simplicity, and because more complex primitives can be constructed with them. More complicated primitives such as convex polygons provide more generality, but at the expense of a huge increase in the complexity of the rendering subsystem. Since arbitrary convex polygons are unbounded in the amount of data needed to describe them, the size of an atomic operation to a hardware subsystem implementing them is also unbounded. General convex polygons can easily and efficiently be tessellated into quadrilaterals by the CPU.

2.4.3.1 Quadrilaterals and Triangles

The basic math for half-space geometry is simple, and its VLSI implementation can be made regular and efficient, but by itself the half-space provides no useful primitive for 2D or 3D graphics. However, consider the interior of a quadrilateral. It is nothing more than the intersection of four half-spaces. The PixelVision rendering subsystem can determine on which side of an edge any pixel lies. It can mark a pixel as ‘+’ or ‘-’ with respect to an edge. A ‘+’ means that a pixel is inside a quadrilateral with respect to an edge, while a ‘-’ means that it is outside. A pixel is inside a quadrilateral if it is inside with respect to all four edges, implemented as the logical and of the component half-space evaluations. A triangle is just a quadrilateral with the first two vertices coincident.

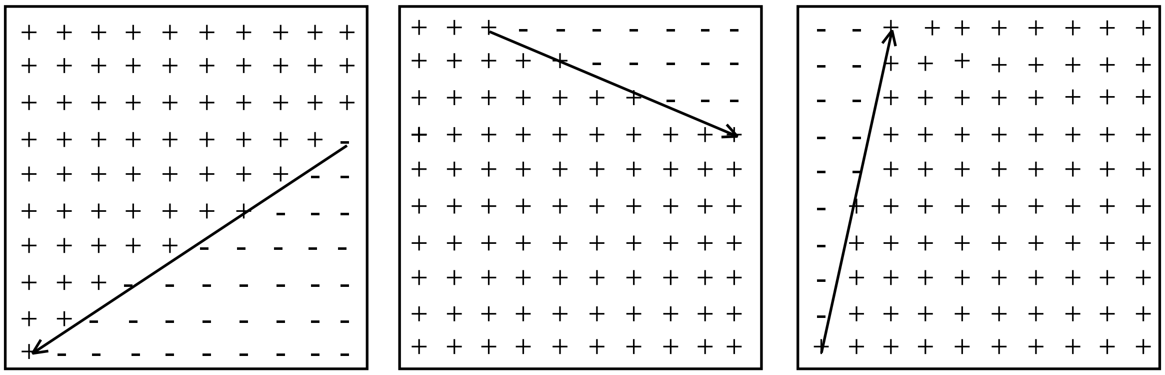


Figure 2-6   Three half-spaces of a triangle sampled at pixel coordinates

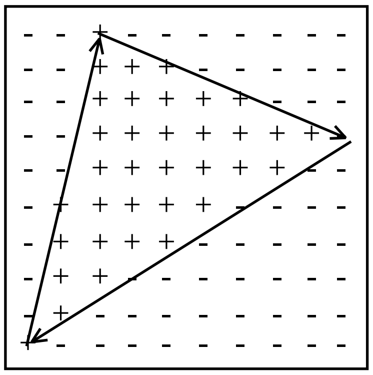


Figure 2-7   The intersection of the half-spaces of figure 2-6

2.4.3.2 Line Segments and Points

Line segments can be modeled as the intersection of four half-spaces, which form a parallelogram as shown in figure 2-8. Two edges are exactly one pixel apart along the minor axis of the line, while two more edges detect the endpoints. A point is just a quadrilateral covering the area of a single pixel.

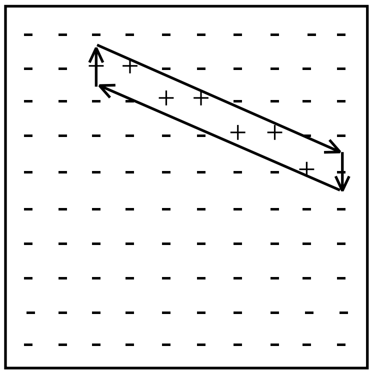


Figure 2-8   A line segment represented as the intersection of four half-spaces

2.4.4 Plane Equations

The PixelVision rendering algorithms include parallel evaluation of plane equations. Plane equations have been used extensively in computer graphics to model object properties across surfaces. They have modeled such properties as distance from the eye for z-buffering, color for smooth shading, and transparency and texture for photorealistic rendering. Although the perspective transformation actually distorts the planarity of object properties, mathematically invalidating the technique, plane equation evaluation is a commonly used efficient technique that produces visually pleasing results. Given that the planar representation of object properties is an approximation anyway, the distortion should not disturb the purists.

2.4.5 Application of Planes to Graphics

2.4.5.1 Smooth Shading

The process of smooth shading implies the evaluation of color planes across the surface of a triangle. There is one plane equation per color channel (3 for an RGB device). Each plane equation is in a three-dimensional space with *x* and *y* as two of the dimensions and a color channel as the third dimension (the red plane has dimensions *x*, y, and *red*). Three points parametrize a triangle's red plane equation: (*x*1, *y*1, *red*1), (*x*2, *y*2, *red*2), and (*x*3, *y*3, *red*3). Evaluating the plane equation at pixel coordinates will generate values to be placed in the red portion of graphics memory. The same is done for green and blue.

2.4.5.2 *z*-buffering

The most common hardware technique to remove hidden surfaces is called *z*-buffering or depth-buffering. It requires not only a color buffer, but a *z* buffer in which *z* values for each pixel reside. Before rendering a scene, the rendering subsystem initializes the *z* buffer to the largest representable *z* value. Three points parametrize a triangle's z plane equation: (*x*1, *y*1, *z*1), (*x*2, *y*2, *z*2), and (*x*3, *y*3, *z*3). The PixelVision rendering subsystem performs the following steps for each pixel in a triangle:

1. Calculate the *z* value at (*x*,*y*) by evaluating the plane equation.
2. If *z* is less than the *z*-buffer value at (*x*,*y*) then place z into the z buffer at (*x*,*y*) and place the color into the color buffer at (*x*,*y*); otherwise, alter neither the color buffer nor the z buffer at (*x*,*y*).

Thus, when a pixel in the triangle being rendered is closer to the viewer than the corresponding pixel in any of the previous triangles processed, then it replaces the pixel in graphics memory.

2.4.5.3 Transparency

Just as the depth and color of an object can be modeled mathematically with plane equations, so can object properties such as transparency and thickness. Three points parametrize a triangle's transparency plane equation: (*x*1, *y*1, *t*1), (*x*2, *y*2, *t*2), and (*x*3, *y*3, *t*3). A later section will describe how the transparency value is used to render transparent objects.

2.4.6 Tiling or Scan Conversion

The basic building blocks have been established: the update array as a unit of graphics memory reference, the half-space evaluations to determine those pixels included in a geometric figure, and the plane equation evaluations to determine object properties. Higher-level geometric models such as lines and quadrilaterals have been discussed. But there has not yet been a discussion of the realization of these geometric models in terms of the update array.

To render a geometric primitive, a *tiling* or *scan conversion* algorithm generates a list of update arrays that completely covers the primitive. The algorithm begins by referencing an update array within the primitive's boundary. Using the half-space representations to determine in which direction the primitive extends, the algorithm determines another update array within the primitive's boundary, and references it. This continues until all update arrays containing a portion of the primitive have been referenced, as depicted in figure 2-9.

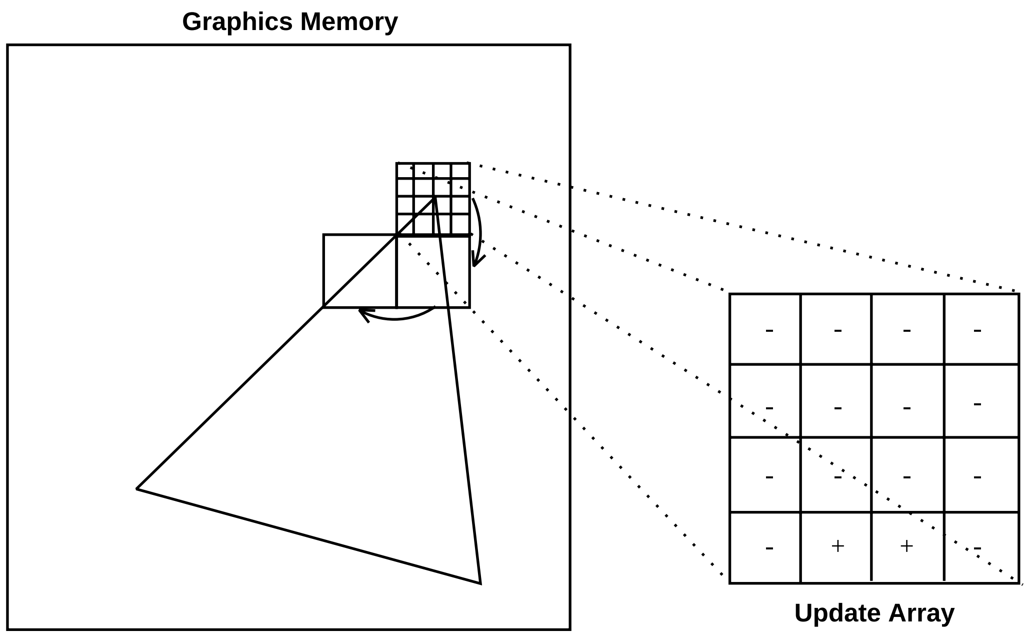


Figure 2-9 Tiling a triangle

2.5 Parallelism in the Rendering Subsystem

The size of a graphics memory reference is the size of an update array. Achieving full memory bandwidth (maximum performance) requires algorithms that can use the available parallelism. A single MxN graphics memory operation must access all MxN pixels in parallel to run at full bandwidth. An increase in performance implies an increase in parallelism.

The key to parallel update is the display memory organization. To update *n* pixels in parallel requires a proportional number of memory chips. Figure 2-10 shows the relationship between a PixelVision rendering chip and graphics memory. The configuration shown has eight memory components accessed in 24-bit quantities.[1] The PixelVision rendering subsystem accepts commands and data over a system bus and turns the data into array accesses of eight 24-bit pixels.

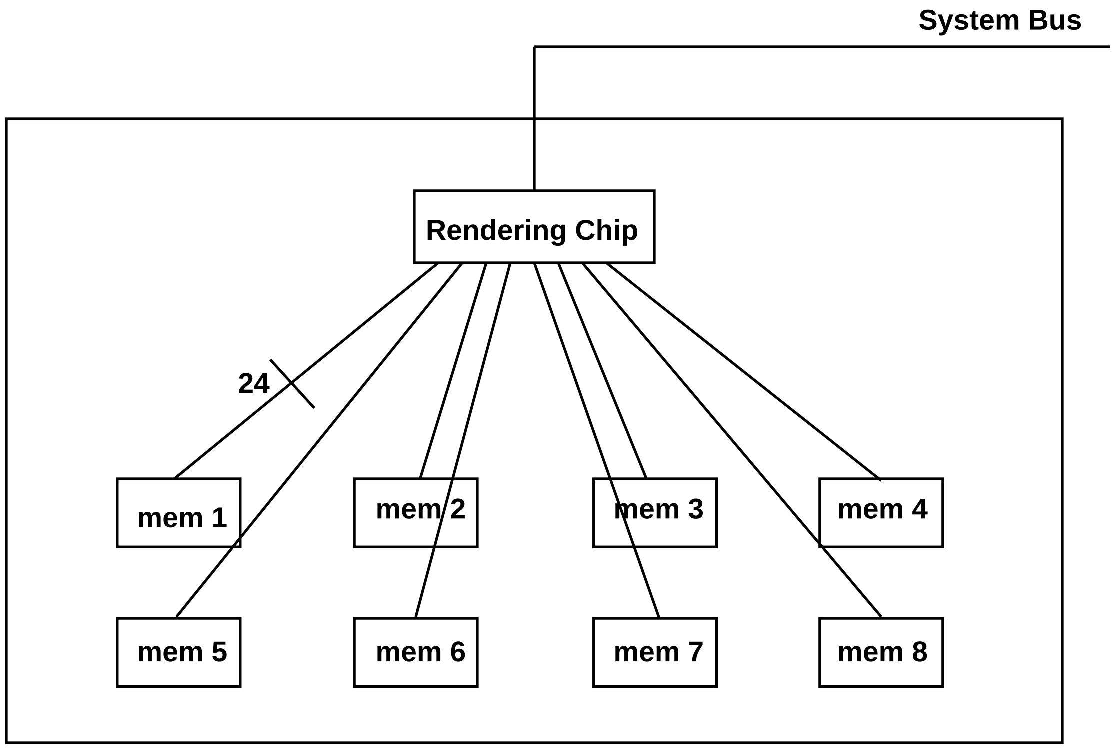


Figure 2-10 PixelVision rendering chip / memory interconnect

To achieve higher performance, this paradigm can be extended to support larger update arrays as price requirements dictate. Increasing the update array size will increase system performance until the system bus or setup time becomes the bottleneck. One 4x2 PixelVision rendering chip provides the front end to 48 256Kx4 memory chips. As more memory is added, more rendering chips are added to maintain the ratio. The raw drawing performance of this type of system is limited only by the amount of memory the designer can afford to incorporate into the PixelVision memory array.

[1] Using 256Kx4 video RAMs, eight 24-bit pixels actually requires 48 (8 x 6) video RAMs.

The performance of any raster graphics algorithm is bounded not only by the number of pixels that can be accessed in parallel, but also by the efficient use of that memory access. Conventional raster graphics algorithms for line and polygon rendering use incremental techniques to determine which pixels belong to a geometric figure. For example, Bresenham's line drawing algorithm generates pixel coordinates by starting at the first point on the line and incrementally calculating successive pixels. Although incremental techniques are efficient for serial implementation, they make no use of parallel update to graphics memory. Hence, performance bounds arise from their incremental nature.

Some parallel graphics algorithms, such as those used by Evans and Sutherland and some Silicon Graphics workstations, run multiple incremental algorithms in parallel. For example, line drawing may use *n* Bresenham evaluators, where each one calculates every *n*th pixel. The first one starts at the first pixel, the second at the second pixel, and so on. The evaluators run in parallel until they reach the end of the line segment. Although this technique provides high-bandwidth access to memory, it requires heavy setup and is specific to line generation. Entirely different algorithms are required to scan convert rectangles or polygons.

The PixelVision rendering architecture uses truly parallel algorithms instead of incremental algorithms in parallel. Mathematical equations of lines and planes provide the foundation for its models and algorithms, making the architecture conceptually appealing. Furthermore, the PixelVision architecture models and renders all supported geometric figures using the same underlying mechanism.

2.6 Rendering Subsystem Interface

The command packet interface to the PixelVision rendering subsystem transfers geometry and graphics context information from the main memory system to the PixelVision rendering module. An application writes a list of commands and data to a physically contiguous locked-down memory buffer in its own address space. This buffer may reside in either the main memory system or locally on the rendering module.

When the application has completed writing the packet, it tells the rendering subsystem, via a read from an IO register on the rendering subsystem, that it should begin a transfer of the command packet. Status of the operation is returned as the result of the IO read transaction. After initiating the command packet transfer, the rendering subsystem parses the packet and executes the appropriate operations, finding the details of the transaction syntax encoded in the first 32-bit word of the command packet. Execution of the commands in the packet is atomic, implying that all commands in one packet are executed before any commands in the next packet. Furthermore, the execution order is guaranteed to be the order in which the commands appear in the packet.

2.7 Graphics Memory System

The PixelVision graphics memory system is part of the rendering subsystem and resides between the rendering processor and the video DAC. It may store the visible image as well as other graphics data including double buffers and depth buffers. The PixelVision graphics memory system provides virtual memory capabilities to graphics applications requiring the services of the rendering subsystem. The terminology and mechanisms are analogous to those of the processor virtual memory system.

In the past, an application (such as the X server) viewed the rendering hardware as being able to create and maintain the relatively small number of images that could fit into its physical memory system. These applications had to shuffle images between main memory and graphics memory in much the same way as applications written for physical memory computers had to shuffle data between disk and main memory. This data movement was at best tedious, and at worst a performance problem.

The virtual memory abstraction for graphics has several important advantages over a strictly physical memory design. First, it allows an application to view the rendering hardware as being able to create and maintain a relatively large number of images. Second, it allows multiple graphics applications to concurrently communicate with the rendering hardware without going through a synchronizing process such as the X server. The burden of graphics memory management is a sufficient reason to force all graphics requests to filter through a common process. Finally, virtual graphics memory allows efficient implementation of memory-intensive rendering algorithms including CSG, transparency, and antialiasing.

2.8 Imaging and Live Video

Perhaps the primary hardware requirement of imaging applications is the fast transference of image data between graphics memory and main memory. Similarly, live video in a workstation environment entails nothing more than the transference of image data to the graphics memory at video frame rates, assuming the data has already been captured. The video data may reside in main memory, on a peripheral digital storage device such as a video disk, or even on another node on the network. PixelVision has defined a high-speed interface between its graphics memory system and the rest of the workstation.

3 PixelVision Memory System

In almost any general-purpose computer system, programs tend to become too large to fit entirely within main memory. Execution of these programs depends on a technique for moving program information between main memory and the auxiliary memory system. It is often difficult for an application to predict its memory requirements, and even more difficult for the application to optimally move its information between main memory and auxiliary memory. Concurrent execution of multiple processes makes the task even more difficult, as now the application cannot independently determine the state of the memory system.

Early computers used a technique called *overlay*, whereby application software managed the memory system by bringing portions of a program into main memory from auxiliary memory as needed. Overlay proved to be a difficult management task, especially with the indeterminacy introduced by multiple-process operating systems. The introduction of virtual memory systems removed this burden from the application by allowing dynamic reconfiguration of the memory system at execution time, as a natural side-effect of the memory reference patterns of the CPU. Virtual memory provides the illusion of a very large main memory system, even though the physical main memory system may be relatively small.

A typical high-performance workstation today consists of a virtual memory computer with a graphics processor that can access only its dedicated physical memory frame buffer. The frame buffer is shared among all graphics applications, each with its own memory requirements. Graphics memory management is necessary. Not surprisingly, a technique similar to overlays has been implemented to move data between the frame buffer and main memory.

The system’s need for virtual memory strengthened as the system hardware and software technology matured. The need for virtual memory in the graphics subsystem strengthens today, as the requirements and algorithms of the graphics subsystem mature. The most obvious benefit to virtual memory in the graphics subsystem is the removal of the memory management burden from the application. A more subtle, yet equally important, benefit is that it allows the implementation of memory-intensive rendering algorithms such as texture mapping, CSG, transparency, and antialiasing.

3.1 Historical Perspective

The most straight-forward graphics subsystem with virtual memory capability is a memory mapped frame buffer such as that of the DECstation 3100 or the DECstation 5000 Model 200. The CPU performs all rendering operations into virtual memory, either to non-paged virtual memory in the frame buffer or to paged virtual memory in main memory. A window server arbitrates the memory in the visible frame buffer among multiple applications.

Ophir is an advanced development project at Digital. Ophir uses the workstation CPU (an Intel I860) to perform rendering operations to system virtual memory. A special-purpose copy engine copies data from system virtual memory to the frame buffer as appropriate. The frame buffer serves only to collect copies of applications' images for display in a window environment. The copy engine performs window clipping in its data path so that occluded portions of windows are not visible.

LCG (Low Cost Graphics) is also a Digital project under development. LCG is a memory controller with integral graphics capabilities for operations such as line drawing and bitblt. LCG can render to both system memory and dedicated frame buffer memory. The graphics processor transacts in system virtual addresses and has its own three entry translation buffer to translate to physical addresses.

The GS-1000 is a high-end workstation designed at Stellar Computer Inc. It boasts both high CPU performance and high graphics performance, as well as high-quality rendering techniques including antialiasing, transparency, and texture mapping. The main data path is the core of the machine, connecting function units including a CPU, vector unit, rendering processor, and memory subsystem. Memory consists of two parts, main memory and video memory, which share a common 512-bit connection to the main data path. Both the CPU and rendering processor can access both main memory and video memory through the main data path. Graphics image data is stored in a data structure called a virtual pixel map (VPM). A VPM is a rectangular array of 32-bit pixels in system virtual memory, addressable through the main data path by both the CPU and the rendering processor.

3.2 Processor Virtual Memory System

The memory hierarchy in a standard computer system consists of caches built from static RAMs backed up to a main memory system built from dynamic RAMs backed up to auxiliary storage devices such as disks and tapes. Access to the caches and main memory is controlled entirely by the computer hardware, requiring no operating system or controller intervention. However, since an IO controller interfaces to the disks and tapes, an operating system device driver manages their access.

Modern computer systems typically layer a virtual memory system on top of the memory hierarchy to give the programmer the illusion of a large main memory system even though the physical main memory system is relatively small. An examination of the benefits of a virtual memory system is best done through the eyes of an application programmer without a virtual memory machine. If a program is fortunate enough to fit entirely within main memory, then the programmer is fortunate enough to be free from dealing with access to the auxiliary memory devices. However, the interaction between larger programs that do not fit entirely within main memory and the memory hierarchy must be managed by the application. The programmer must manage the data movement (referred to as *paging*) between a reserved portion of the auxiliary memory and main memory. The programmer must handle this paging in software, a task that is both tedious and difficult to optimize.

Virtual memory moves the burden of paging from the application programmer to hardware and operating system exception handlers. Virtual memory allows the programmer to view the memory hierarchy as having the aggregate size of the main memory system plus the portion of the auxiliary memory reserved for paging. The programmer can divorce his software from the paging process, as paging occurs as a side effect of the CPU’s instruction sequence.

The paging process views the physical memory system as an array of *page frames*. A page frame is a portion of main memory that is treated atomically by the virtual memory paging process. That is, when the paging process moves data from auxiliary memory to main memory, it moves enough data to completely fill a page frame. A page frame refers only to the memory itself, not to the data stored in it. The data stored in the memory is a *page*. One page frame of memory stores one page of data. A page frame is identified by a *page frame number (PFN)*. A page is identified by a *virtual page number (VPN)*, typically the high-order bits of the virtual address.

The translation from virtual addresses to physical addresses is typically done through a hardware data structure called a *translation lookaside buffer (TLB)*. A TLB is a mapping of the most recently referenced page frames to the pages stored in them. A *TLB entry* is a single entry in the TLB, mapping a single page to a single page frame.

When the processor references memory, if the TLB contains a mapping for the referenced page, then the TLB returns the corresponding page frame number. If the TLB does not contain the mapping, then the processor traps and system software loads the mapping into the TLB. This is called a *TLB miss.* If the page is not resident in main memory, then the system software also pages the data in from auxiliary memory. This is called a *page fault.*

3.3 Graphics Virtual Memory System

Physical memory restrictions have burdened low-level graphics software and firmware developers since the introduction of the first special-purpose graphics hardware. Some graphics software interfaces such as Xlib and PEXlib already provide a virtual memory abstraction to their applications. The primary benefit of this abstraction is to allow the application a view of an unlimited number of coexistent images it can create and maintain through the software interface. Hardware support for virtual memory rendering offers the benefit of a cleaner and superior implementation of this abstraction. Furthermore, when coupled with a robust set of operations in the rendering processor, hardware support allows efficient implementations of memory intensive rendering algorithms, including CSG, transparency, and antialiasing.

Graphics hardware, as outlined previously, has recently begun to address the problems caused by physical memory restrictions. Ophir and simple memory mapped frame buffers achieve virtual memory rendering by allowing the CPU to implement the rendering algorithms. LCG uses a special-purpose memory controller to implement the rendering algorithms. The Stellar GS-1000 shows yet another approach, interfacing a special-purpose rendering processor to the main memory subsystem through a wide bus. While these systems have very different implementations, they are similar in their ability to render directly into processor virtual memory.

Each of these works tightly integrates the graphics with the main system. Unfortunately, systems that tightly integrate graphics with the system generally do not satisfy the full range of paying customers. For example, a workstation that uses the CPU to perform rendering operations may be a good low-cost solution; however, customers will undoubtedly demand higher performance. Similarly, a workstation that places a special-purpose graphics component in a prominent place in the system, such as the GS-1000 or a system built with LCG, is not scalable to please a range of customers.

A robust workstation product line includes a low-cost entry configuration with limited graphics hardware, and allows higher graphics performance through upgrades to higher-cost graphics options. This approach provides a cohesive workstation product line that pleases both demanding customers and frugal customers. The previous works on virtual graphics memory preclude this design approach because of their tight integration of graphics memory and main memory.

The PixelVision memory system remains relatively loosely integrated with the main memory system. Its physical and virtual memory systems are distinct from those of the CPU. In addition to providing an upgrade path, this approach allows the graphics subsystem and base system to evolve separately, and allows the graphics subsystem to be portable to different base system architectures.

3.3.1 Graphics Virtual Memory: An Application's View

Graphics applications view the graphics memory system as a virtual resource just as they view main memory as a virtual resource. Whereas the CPU's instruction set presents an abstraction of main memory in terms of simple data structures such as bytes and words, the PixelVision's instruction set presents a higher-level abstraction that matches the needs of a graphics application. The basic data structures are the *pixel* and the *pixelMap*.

3.3.1.1 Pixels and Channels

A *pixel* is a 24-bit unit of virtual memory. The data stored at a pixel in general has no implied semantics -- it can represent color, depth, transparency, or other graphics data.

A pixel may be further divided into *channels*. Channels represent logically homogeneous data within a pixel. Channels may be one, two, or three bytes per pixel. For example, 24-bit pixels representing color data have three one-byte channels, one for each of red, green, and blue. The red channel affects the monitor's red phosphors, the green channel affects the green phosphors, and the blue channel affects the blue phosphors.

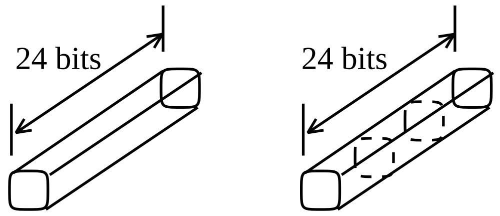


Figure 3-1 Pixels with one and three channels respectively

3.3.1.2 PixelMap Data Structure

A pixelMap is a two-dimensional array of pixels, and is identified by a 16-bit value known as the *pixelMap ID*. An (x,y) coordinate pair references a pixel within the two-dimensional array. A pixelMap's coordinate space ranges from -4096 to 4095 in both x and y.

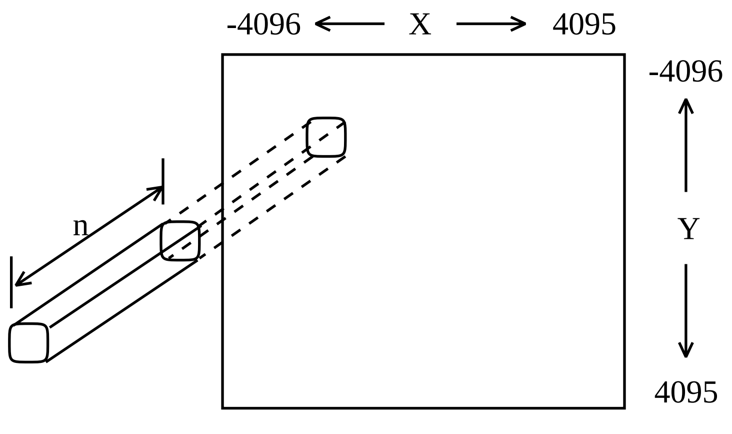


Figure 3-2 A Pixel within a PixelMap

3.3.1.3 Visible PixelMap

PixelVision *may* reserve a portion of graphics physical memory to store the pixels that are visible on the display. These pixels form the *visible pixelMap*, and are referenced through pixelMap ID 0. Alternatively, the visible pixels may reside elsewhere in the system -- as a portion of the main memory system for example. The pixels within the visible pixelMap have implied semantics as they are hardwired to the display circuitry. The video DAC interprets pixels within the visible pixelMap as either true-color pixels or pseudo-color pixels,[1] as specified by the PT field of the display control channel (described below).

An application may access the visible pixelMap just as any other pixelMap. The only distinctions are that the size of the visible pixelMap is limited by the display hardware, and its upper-left corner is at (0,0) instead of (-4096,-4096). The visible pixelMap's size is determined by the rendering subsystem's implementation and revision register. Valid sizes are 1024x1024, 1280x1024, 2048x1280, 2048x1536, 2048x2048, and unsupported (implying that the visible pixels reside elsewhere in the system). If the actual display resolution differs from these supported sizes, then the implementation and revision register contains the next largest size.

PixelVision has specific support for stereo viewing that requires dedication of two screen-size pixelMaps to the display subsystem. These two pixelMaps store a left image and a right image, which the graphics memory system and the monitor work in conjunction to alternate to the display surface. Special glasses prevent the left image from projecting to the right eye and the right image from projecting to the left eye. PixelVision supports stereoscopic viewing by dedicating an additional pixelMap, identified by pixelMap ID 1, to the problem. PixelMap ID 0 stores the *left visible pixelMap* and pixelMap ID 1 stores the *right visible pixelMap*.

[1] Refer to X Windows documentation for a definition of true-color and pseudo-color.

3.3.1.4 Display Control Channel

In addition to the one to three user-definable channels, each pixel contains a predefined 8-bit channel called the *display control channel.* The display control channel resides in a separate bank of graphics memory from the pixel's other channels, and must therefore be addressed separately. The high-order bit of the pixelMap ID selects between the display control channel (when the bit is set) and the rest of the pixel (when the bit is clear). For example, a pixelMap ID of hex 8000 refers to the display control channel of the visible pixelMap, while a pixelMap ID of 0 refers to the color portion of the visible pixelMap. The display hardware divides the display control channel into three fields as shown below.

7 4	3 2	0
OVERLAY	WT	U

where

OVERLAY controls the overlay planes of the video DAC. The precise semantics of OVERLAY are implementation-specific. Refer to implementation specifications for more detail.

WT determines how the video DAC interprets the pixel (e.g. true color, pseudo color). The precise semantics of WT are implementation-specific. Refer to implementation specifications for more detail.

U (USER) is uninterpreted by the video DAC. USER may contain window clipping information, temporary bit masks, or any other appropriate graphics data.

3.3.1.5 DMA PixelMap Virtual Addressing

DMA pixelMap transactions copy data between system memory and the graphics memory system. The CPU initiates DMA pixelMap transactions by writing to eight registers on the PixelVision rendering subsystem: the *VMpixelMapID* register, the *VMxyMin* register, the *VMxyMax* register, the *VMaddrLow* register, the *VMaddrHigh* register, the *VMoffset* register, the *VMformat* register, and the *VMrdwr* register. Additionally, there is a *VMstatus register* that describes the status of the DMA hardware.

The pixel data occupies a block of pixels within a pixelMap in graphics memory. The VMpixelMapID register specifies the pixelMap, and the VMxyMin and VMxyMax registers specify the block's upper-left-most pixel and its lower-right-most pixel respectively. VMxyMin and VMxyMax are (*x, y*) coordinate pairs packed into 32-bit words, where the *y* values are in bits 31:16 and the *x* values are in bits 15:0. The *x* and *y* values are two's complement fixed point numbers in the subpixel coordinate system (described in chapter 4) with 16 significant bits; however, the low three bits are ignored and assumed to be zero. The top row and the left-most column of pixels are included in the transfer, while the bottom row and the right-most column of pixels are excluded.

Each pixel in the block occupies a 32-bit word in system memory. Pixel (xMin+i, yMin+j) in graphics memory corresponds to physical address (VMaddr + (xMax-xMin+VMoffset)*j + i), where i and j are less than (xMax-xMin) and (yMax-yMin) respectively from VMxyMin and VMxyMax, and VMaddr is a 64-bit address formed as the concatenation of VMaddrHigh and VMaddrLow. System memory is partitioned, as described in implementation specifications, between the main memory system and the PixelVision scratch memory system described in section 8.5.

The *VMformat* register indicates the pixel data's format. If VMformat is 0, bits 23:16, 15:8, and 7:0 of pixel data in main memory correspond to the red, green, and blue channels respectively in graphics memory, and bits 31:24 are undefined. If VMformat is 1, then the red and green channel, and bits 23:0 are undefined, and bits 31:24 correspond to the blue channel. Undefined channels in graphics memory are not updated during DMA read transactions. Undefined bytes in main memory may not be updated during DMA transactions (refer to implementation specifications for detail). The control offered by VMformat allows a pixel's display control channel to be stored in main memory in the same 32-bit word as the rest of the pixel's channels.

	31 24	23 16	15 8	7 0
VMformat = 0	Undefined	Red	Green	Blue

	31 24	23 0
VMformat = 1	Blue	Undefined

Writing to the VMrdwr register initiates the DMA pixelMap transfer. If the CPU writes a 0 to the VMrdwr register, then the rendering subsystem transfers data from graphics memory to main memory. If the CPU writes a 1 to the VMrdwr register, then the rendering subsystem transfers data from main memory to graphics memory. Writing to the VMrdwr register may be destructive to the other VM registers.

The CPU should poll the VMstatus register before writing to the VM registers. If VMstatus is non-zero, the DMA hardware is busy and the CPU should not write to the VM registers.

Software must ensure that the transfer of data does not violate any DMA restrictions of the system. DMA pixelMap transactions will transfer a maximum of 32 pixels in one DMA transaction across the system bus, and a DMA transaction will not cross a 32-pixel boundary in graphics memory.

Any pixelMap memory referenced through DMA pixelMap accesses must be resident in graphics physical memory. DMA pixelMap accesses should be issued only when the rendering subsystem is idle (as specified by the rendering subsystem's status register) or stalled as a result of a graphics memory page fault.

The VM registers reside at an offset from an implementation-specific base address as follows:

31 21	20 18	17 2	1 0
RENDERBASE	1	REG	0

where

RENDERBASE offsets to the base address of the rendering subsystem; and

REG is 0 for the VMpixelMapID register, 2 for the VMxyMin register, 3 for the VMxyMax register, 4 for the VMaddrLow register, 5 for the VMaddrHigh register, 7 for the VMoffset register, 8 for the VMformat register, 9 for the VMrdwr register, and 10 for the VMstatus register.

3.3.2 Graphics Virtual Memory: System Software's View

This section describes the system software's view of the virtual memory system for the first PixelVision implementation. The terminology of PixelVision's virtual memory system is similar to that of a system's virtual memory system. PixelVision has pages and page frames, a TLB with TLB entries, and a page faulting mechanism. However, although the terminology is similar, the implementation has one significant difference: PixelVision cannot independently handle page faults. Instead, a page fault in the rendering subsystem interrupts the CPU, which responds by moving the requested data from its virtual memory system to graphics memory. Thus, the graphics virtual memory system is layered on top of the system's virtual memory system, allowing the graphics system to take advantage of existing mechanisms for backing up data to auxiliary memory.

3.3.2.1 Graphics Pages and Page Frames

The first implementation allows a maximum of 256 page frames. The total amount of physical memory determines the size of a page of graphics data. Specifically, a page is a minimum of 16K pixels, arranged in a 128x128 array of pixels. The page size grows to 32K pixels, arranged in a 256x128 array of pixels, as the amount of physical memory grows to 32MBytes. As with system virtual memory, a page frame of memory stores a page of data. Table 3-1 shows the correspondence between the physical memory size and the page size.

Table 3-1 Graphics Memory for PixelVision Systems with 24-bit pixels plus an 8-bit display control channel.

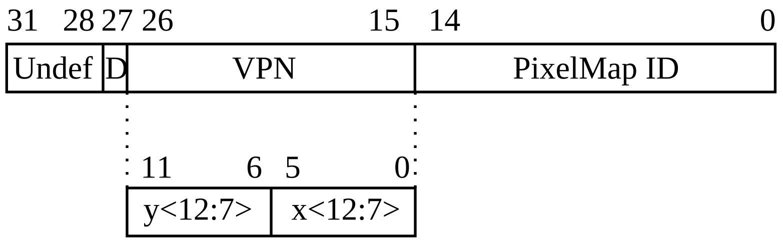
Physical Memory (in pixels)	Physical memory (in MBytes)	Physical memory (in page frames)	Page size (in pixels)
2M	8	128	16K (128x128)
4M	16	256	16K (128x128)
8M	32	256	32K (256x128)

The visible pixelMap resides at the beginning of the graphics physical address space, and should never be paged out. System software must ensure that the visible pixelMap's page frames are not used for pageable pixelMaps (see TLB section).

3.3.2.2 Graphics TLB

The purpose of a TLB is to determine the page frame number (PFN) corresponding to a virtual address. PixelVision's TLB provides a mapping of all page frames in the graphics physical memory system. This differs from most TLB implementations which map a relatively small number of page frames and trap to system software when the TLB does not contain the desired mapping. PixelVision will trap to system software only when a referenced page is not resident in the graphics memory system.

A TLB entry consists of a pixelMap ID, a virtual page number (VPN), and a dirty bit (D). The TLB entry excludes bit 15 of the pixelMap ID because it does not contribute to distinguishing a pixelMap, but instead selects between the display control channel and the pixelMap's other channels. The VPN is a 12-bit value consisting of the high 6 bits of the *x* and *y* coordinates of the virtual address. PixelVision uses the low bits of the *x* and *y* coordinates to index into the page frame. A TLB entry's dirty bit is set by the rendering subsystem whenever the entry matches a referenced graphics memory address. A TLB entry is a 32-bit value constructed as follows:



When bit 14 of the pixelMap ID is set, *x* and *y* are unused in the translation and must be zero in the TLB entry. This feature allows geometric primitives, such as lines or quadrilaterals, to be tiled with a pattern the size of a page of graphics memory.

PixelVision's TLB has 256 entries. This is sufficient to provide a mapping for all page frames on 2, 4, and 8MPixel systems as outlined in table 3-1. On 8MPixel systems, the VPNs stored in the TLB entries should be even. The PFN returned by the TLB is the index of the TLB entry that matches the desired virtual address. Results are undefined in the case where more than one TLB entry matches.

Note that there is no valid bit in the TLB entry. All entries are valid. System software should reserve a pixelMap ID to flag an invalid TLB entry. Entries containing the *invalid pixelMap ID* will never match a referenced pixelMap ID.

As mentioned previously, PixelVision may reserve a portion of the physical address space, and hence a portion of the TLB, for the visible pixelMap. The table below enumerates the supported visible pixelMap sizes and shows the corresponding reserved TLB entries.

Table 3-2 Visible PixelMap Reserved TLB Entries

Visible		Monoscopic Viewing	
PixelMap	2MPixels	4MPixels	8MPixels
1024x1024	0 to 63	0 to 63	0 to 31
1280x1024	0 to 79	0 to 79	0 to 39
2048x1280	unsupported	0 to 159	0 to 79
2048x1536	unsupported	0 to 191	0 to 95
2048x2048	unsupported	0 to 255	0 to 127
unsupported	none	none	none
		Stereoscopic Viewing	
1024x1024	0 to 127	0 to 127	0 to 31, 64 to 95
1280x1024	unsupported	0 to 79, 128 to 207	0 to 39, 128 to 167
2048x1280	unsupported	unsupported	0 to 79, 128 to 207
2048x1536	unsupported	unsupported	0 to 95, 128 to 223
2048x2048	unsupported	unsupported	0 to 255
unsupported	none	none	none

The system may access the TLB only when the rendering subsystem is waiting for command data or waiting for the system to respond to a page fault interrupt. System software should initialize the TLB at boot time by clearing all TLB entries. The TLB resides at an offset from an implementation-specific base address as follows:

31 21	20 18	17 2	1 0
RENDERBASE	3	TLB_ENTRY	0

where

RENDERBASE offsets to the base address of the rendering subsystem; and

TLB_ENTRY is the TLB entry -- 0 to 255.

3.3.2.3 Graphics TLB misses

A TLB miss occurs when the PixelVision rendering subsystem references a virtual address whose mapping is not contained in the TLB, and the referenced address is not completely clipped by the rendering subsystem's clipping rectangle. A TLB miss causes an interrupt to the CPU, indicating that the CPU must move the referenced data into the graphics memory system. System software must determine which virtual page caused the interrupt so that it may transfer the corresponding data. The virtual page is given by the read-only TLBaddr register, which has the same format as a TLB entry, and resides at an offset from an implementation-specific base address as follows:

31 21	20 18	17 2	1 0
RENDERBASE	2	0	0

The paging process comprises the following steps:

1. Page old data out of graphics memory. PixelVision offers no assistance in the determination of which page to remove.
2. Place a mapping of the requested page in the TLB.
3. Page new data into graphics memory.
4. Clear the PAGE_INT bit in the rendering subsystem's status register. The rendering subsystem sets the PAGE_INT bit upon interrupting the CPU in response to a graphics TLB miss.

3.3.3 Graphics Physical Memory Implementation

This section describes the physical memory implementation for the first PixelVision implementation. All graphics memory except that reserved for the visible pixelMap may be used to store pageable graphics data. This section describes the mapping from a graphics virtual address to a graphics physical address.

3.3.3.1 Address Translation in the Visible PixelMap

The mapping of addresses within the visible pixelMap from virtual to physical does not involve the TLB because the visible pixelMap is not pageable. Furthermore, pages within the visible pixelMap are arranged differently than they are for virtual pixelMaps. The visible pixelMap is arranged within video RAM specifically to please the video refresh circuitry. Virtual pixelMaps do not have the same restrictions and therefore are arranged differently.

PixelVision's mapping of a pixel within the visible pixelMap to a location in physical memory depends on the size of the visible pixelMap and the size of the update array. Given an MxN update array and an XxY visible pixelMap, the address of a pixel (p_x,p_y) is:

$$\text{addr} := (p_y/N) \times (X/M) + (p_x/M).$$

Since there are N *times* M pixels in a single word of graphics memory, N *times* M pixels map to the same physical address. An (NxM)-bit mask complements the physical address so that the rendering processor may reference any subset of the pixels in an update array. Since X, Y, M, and N have been carefully chosen, the calculation of addr can be done with just shifts and adds. Using video RAMs with 512 rows and 512 columns, PixelVision determines row and column addresses by:

ra := addr<17:9>;

ca := addr<8:0>.

3.3.3.2 Address Translation in Pageable PixelMaps

The virtual to physical address translation for pageable pixelMaps is different than the translation for the visible pixelMap. Given a page frame number (PFN) from the TLB, the address of a pixel (p_x,p_y) is:[1]

on a 4x2 system with 2MPixels,
addr := PFN<6:2> $\cdot$ p_y<6:5> $\cdot$ p_x<6:5> $\cdot$ PFN<1:0> $\cdot$ p_y<4:1> $\cdot$ p_x<4:2>
on a 4x2 system with 4MPixels,
addr := same as above; PFN<7> acts as a memory bank select
on a 4x4 system with 4MPixels,
addr := PFN<7:2> $\cdot$ p_y<6:5> $\cdot$ p_x<6> $\cdot$ PFN<1:0> $\cdot$ p_y<4:2> $\cdot$ p_x<5:2>
on a 4x4 system with 8MPixels,
addr := PFN<6:2> $\cdot$ p_y<6:5> $\cdot$ p_x<7:6> $\cdot$ PFN<1:0> $\cdot$ p_y<4:2> $\cdot$ p_x<5:2>;
PFN<7> acts as a memory bank select
on a 8x2 system with 4MPixels,
addr := PFN<7:2> $\cdot$ p_y<6:5> $\cdot$ p_x<6> $\cdot$ PFN<1:0> $\cdot$ p_y<4:1> $\cdot$ p_x<5:3>
on a 8x2 system with 8MPixels,
addr := PFN<6:2> $\cdot$ p_y<6:5> $\cdot$ p_x<7:6> $\cdot$ PFN<1:0> $\cdot$ p_y<4:1> $\cdot$ p_x<5:3>;
PFN<7> acts as a memory bank select
on a 8x4 system with 8MPixels,
addr := PFN<7:2> $\cdot$ p_y<6:5> $\cdot$ p_x<7> $\cdot$ PFN<1:0> $\cdot$ p_y<4:2> $\cdot$ p_x<6:3>

Since there are N *times* M pixels in a single word of graphics memory, N *times* M pixels map to the same physical address. An (NxM)-bit mask complements the physical address so that the rendering processor may reference any subset of the pixels in an update array. Using video RAMs with 512 rows and 512 columns, PixelVision determines row and column addresses by:

ra := addr<17:9>;
ca := addr<8:0>.

The low-order two bits of the PFN are the high-order two bits of the column address, placing data from four adjacent PFNs in the same row of the dynamic memory array. This allows an implementation to use page mode memory cycles when transitioning between adjacent PFNs, yielding a significant performance advantage. System software should consider this when paging data in response to TLB misses. For example, corresponding pages of a depth pixelMap and its associated color pixelMap should map to adjacent PFNs so that PixelVision is able to more quickly alternate references between the two.

In addition to the low-order two bits of the PFN, the column address uses low-order bits of both p_x and p_y. A traditional graphics memory organization uses only bits of p_x to construct the column address, or perhaps one of the low-order bits of p_y. The traditional approach allows the rendering processor to use page mode memory cycles only when changing the p_x coordinate, and scan conversion algorithms have been tuned around this restriction. PixelVision's memory organization allows an implementation to use page mode memory cycles when moving within a two-dimensional array of pixels, and implementations can tune the algorithms accordingly.

[1] The $\cdot$ operator concatenates bit vectors.

4 Half-Space and Plane Equation Mathematics

Rendering geometric shapes given half-space and plane equation evaluations as the basic primitives is appealing because of the sound mathematical foundation. Conceptually, this foundation provides a straightforward, elegant approach to the rendering of common graphics databases. This chapter shows that this conceptual elegance lends itself to a regular VLSI implementation, providing high parallelism at relatively low cost.

4.1 Half-Space Mathematics

The basic task in evaluating half-spaces is to determine on which side of a directed line a point lies. Thus, if we can find an equation that is positive for points on one side of a line, negative for points on the other side of the line, and zero for points on the line, then our task is complete, assuming it can be implemented economically.

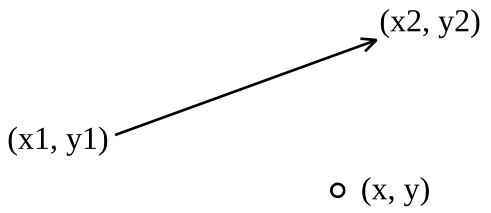


Figure 4-1 A point (x,y) and a directed line from (x1,y1) to (x2,y2)

We first translate the directed line to the origin, represent the point as a vector from the origin, and construct a vector normal to the directed line, onto which we can project the point. In the PixelVision coordinate system, in which x increases to the right and y increases downward, the normal to vector [x y] is [-y x].

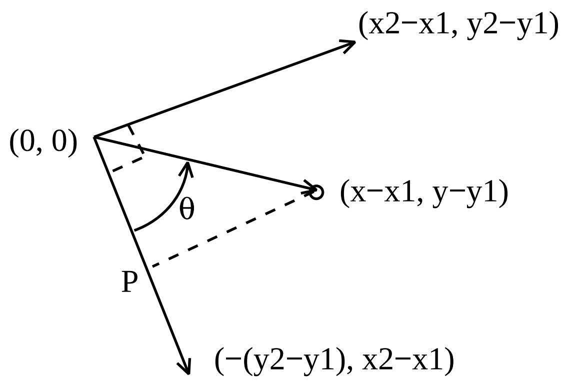


Figure 4-2 The projection P of the point onto the normal to the directed line

The projection of the vector [(x-x1) (y-y1)] onto the normal vector [-(y2-y1) (x2-x1)] will determine the sidedness of the point (x, y) with respect to the directed line. In particular, if the projection is in the same direction as the normal vector, the point is to the right of the directed line; otherwise, it is to the left.

The projection point P is actually of no concern. We are interested only in the component of [(x-x1) (y-y1)] in the direction of [-(y2-y1) (x2-x1)]. This is a value that is positive if the projection has the same direction as the normal vector, and is negative if the projection has the opposite direction. Given two vectors A and B and an angle between them θ, the component of B in the direction of A is given by the equation

$$\text{B-component in A-direction} = \| B \| \cos\theta \tag{4.1}$$

Since the scalar product of two vectors A and B is defined by the equation

$$A \cdot B = \| A \| \, \| B \| \cos\theta, \tag{4.2}$$

we can substitute into equation 4.1 to get

$$\text{B-component in A-direction} = (A \cdot B) \; / \; \| A \| \tag{4.3}$$

Since we are ultimately interested only in the sign of the result, we can multiply both sides by the positive value $\| A \|$ to get the equation for sidedness, the sign of which determines on which side of the directed line the point lies:

$$\text{sidedness} = \| A \| \, (\text{B-component in A-direction}) = (A \cdot B) \tag{4.4}$$

where A is the normal vector to the directed line, and B is the vector from (x1,y1) to (x,y). Substituting the values from our example,

$$\text{sidedness} = (y-y1)\, dx - (x-x1)\, dy \tag{4.5}$$

where $dx = (x2-x1)$ and $dy = (y2-y1)$.

4.2 Discrete Half-Space Geometry

The mathematics of the previous section was presented in a real number system. However, the addressing of raster graphics memory requires the discretization of the real number coordinates. This discretization is known as *sampling*. The task of basic rendering algorithms is to sample geometric shapes at discrete pixel addresses, representing one sample point per pixel. This sampling frequency gives a relatively simple criterion for the decision of whether a pixel is inside or outside a geometric figure.

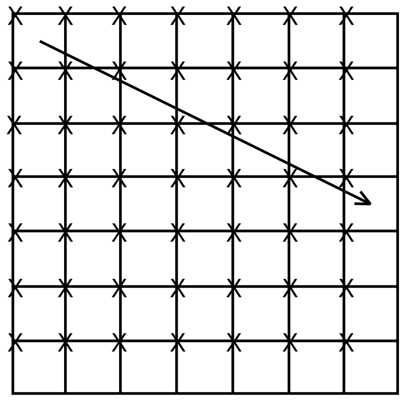


Figure 4-3 Sampling a half-space at pixel locations. Each box represents a pixel and each 'X' represents a sample point.

4.3 Pixel Coordinate System

The PixelVision's *pixel coordinate system* is a typical frame buffer coordinate system in which x increases to the right and y increases downward. Pixel coordinate values have 13 bits, including a sign bit, in both x and y. This gives a pixel coordinate space from -4,096 to 4,095 in both x and y. Applications of the PixelVision rendering subsystem use the pixel coordinate system to describe individual locations within a pixelMap, when transferring data to and from the graphics memory system either through DMA pixelMap accesses or through READ_BLOCKS and WRITE_BLOCKS operations (described in chapter 8).

4.4 Subpixel Coordinate System

When rendering geometric figures such as lines and quadrilaterals, positioning vertices at higher resolution than the pixel coordinate system will produce higher-quality images, especially when antialiasing. For this reason, the PixelVision rendering subsystem supports a higher-resolution coordinate system, called the *subpixel coordinate system*, to describe the vertices of geometric figures.

Subpixel coordinate values have 16 bits, including a sign bit, in both x and y. This gives a subpixel coordinate space from -32,768 to 32,767 in both x and y. The subpixel coordinate system maps directly onto the pixel coordinate system, although it is eight times larger along both axes. The subpixel coordinate system effectively divides each pixel into an 8x8 grid, with (0, 0) in the upper left corner. Specifically, the subpixel coordinate address (*x,y*) is located at pixel coordinate (x div 8, y div 8), and at a subpixel location within that pixel coordinate of (x mod 8, y mod 8).

Recall equation 4.5:

$$\text{sidedness} = (y-y1)\,dx - (x-x1)\,dy \qquad (4.5)$$

Since PixelVision uses the subpixel coordinate system to specify coordinate values, sampling at pixel coordinates implies sampling at {0, 8, 16, 24, ...}. Since the subpixel coordinate system is 8 times larger than the pixel coordinate system (in both x and y), pixels are 8 units apart. Notice that sampling occurs in the upper left corner of a pixel, where the subpixel portion of the address is (0,0).

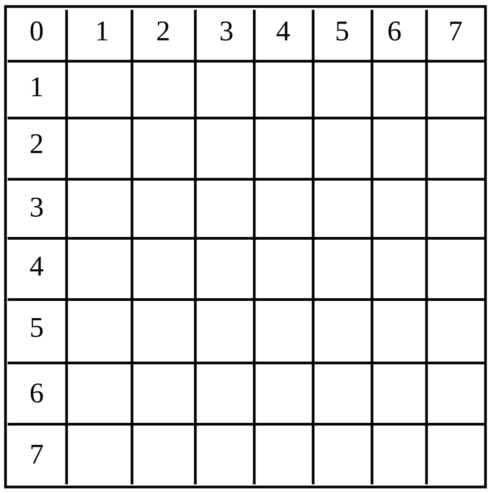


Figure 4-4 A single pixel with 64 subpixel positions

4.5 Discrete Mapping to the Update Array

Given the technology for evaluating a half-space, the next step is to apply this technology to the graphics memory update process. For any given 4x4 update array, the evaluation of the equation,

$$\text{sidedness} = (y-y1)\,dx - (x-x1)\,dy \qquad (4.5)$$

at each of the sixteen pixels in the update array will determine in which half-space each of the pixels lies. Since the PixelVision rendering subsystem evaluates all sixteen pixels in parallel, it makes this determination in a single *evaluation time*. To evaluate a half-space over a region larger than 4x4 requires that multiple update arrays be evaluated, tiling the region completely.

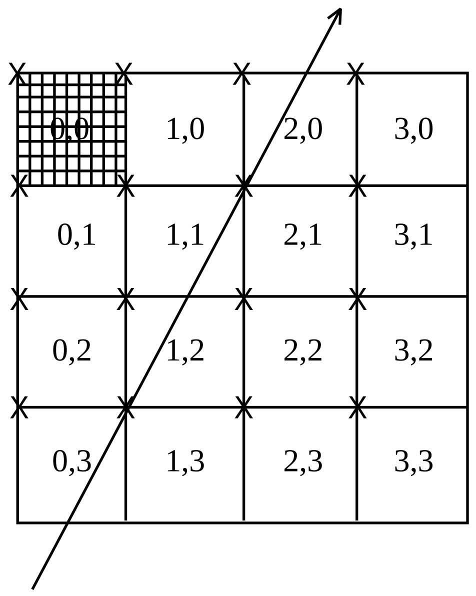


Figure 4-5 Parallel evaluation of a half-space at the sixteen sites in an update array. ‘X’ marks indicate points at which the half-space is sampled.

4.6 Another View of the Half-Space Equation

Since a raster device inherently samples at discrete values, evaluation of a half-space is achieved by evaluating the half-space equation at each pixel in graphics memory.[1] Although conceptually simple, it is not readily apparent that a VLSI implementation of half-space evaluation can be made cheaply and efficiently. This section describes a simple and efficient technique for the parallel evaluation of a half-space equation at discrete values. A logical starting place is with equation 4.5:

$$\text{sidedness} = (y-y1)\,dx - (x-x1)\,dy \qquad (4.5)$$

This can be rewritten:

$$\text{sidedness} = dx\,y - dy\,x - dx\,y1 + dy\,x1 \qquad (4.6)$$

Sidedness is 0 for (*x,y*) on the line, positive on one side of the line, and negative on the other side. By merely evaluating the sign of sidedness at each site in the update array, the rendering subsystem can determine sidedness at each pixel. However, the simplicity sought by a VLSI implementation has not yet been achieved as the calculation of (4.6) is still too difficult, involving four multiplies and three adds at each site. Fortunately, by making a few substitutions, the multiplies can be amortized across the evaluation of multiple update arrays. Let (*originx, originy*) represent the origin of the update array. Also, let (*offsetx, offsety*) be the offset from that origin such that $x = originx + offsetx$, and $y = originy + offsety$.

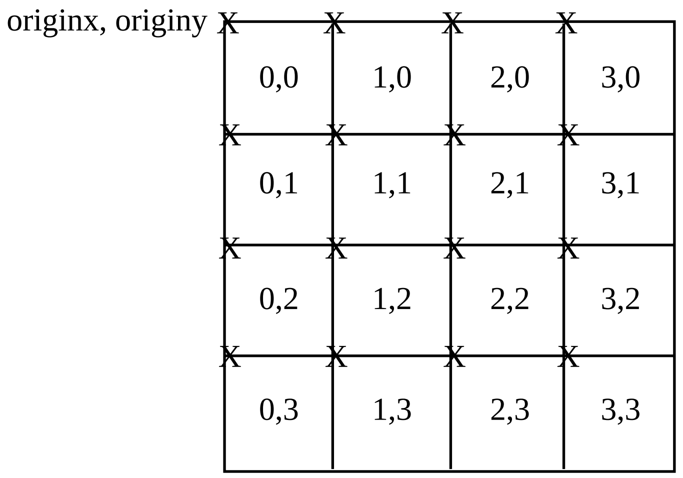


Figure 4-6 The update array is an origin with sixteen offsets from the origin.

[1]In practice, only those pixels in close proximity to the geometric primitive are examined.

Substituting for *x* and *y* in equation 4.6 and rearranging, we get

sidedness = (4.7)

$dx\ offsety - dy\ offsetx +$ (4.7a)

$dx\ originy - dy\ originx +$ (4.7b)

$- dx\ y1 + dy\ x1$ (4.7c)

Notice when the evaluation of the different expressions occurs, as well as the duration of their validity. The rendering processor must calculate some expressions once per half-space, and others once per update array per half-space. No expression is evaluated once per site per update array. Thus, the rendering subsystem can realize a high degree of parallelism.

(4.7a) is calculated just once per half-space. *Offsetx* and *offsety* are positive integers (0, 8, 16, 24)[1] representing *x* and *y* offsets from the origin of the update array, whose values are fabricated into the hardware. Since its values are regularly spaced, the evaluation of (4.7a) can be simplified to successive adds of *dx* and *dy*.

(4.7c) is also evaluated just once per half-space. Like (4.7a), its value is unaffected by the origin of the update array; however, unlike (4.7a), its value is also unaffected by the offset within the update array.

The only expression that is affected by the origin of the update array is (4.7b) -- *dx originy - dy originx*. This leaves only two multiplies and one add per update array. Furthermore, a later section will describe how *forward differencing* techniques can further reduce these compute requirements to a single add per update array.

[1]The values are (0, 8, 16, 24) instead of (0, 1, 2, 3) because coordinate values are in the subpixel coordinate system and the PixelVision rendering subsystem samples once per pixel.

4.7 A VLSI Circuit for Half-Space Evaluation

This section describes a VLSI circuit to solve the half-space equation. The half-space equation,

$$\begin{aligned} &\text{sidedness} = && (4.7)\\ &\quad dx\ offsety - dy\ offsetx\ + && (4.7a)\\ &\qquad dx\ originy - dy\ originx\ + && (4.7b)\\ &\qquad\quad - dx\ y1 + dy\ x1 && (4.7c) \end{aligned}$$

has only one subexpression that is different for each site within an update array -- expression (4.7a). This expression is called the *half-space site value*. As this term must be computed for all sixteen permutations of the allowable values for offsetx and offsety, it is seemingly the most demanding calculation. Fortunately, the regularity of the discrete subpixel coordinate system allows a very efficient implementation. Table 4-1 illustrates the regularity of the values needed for the subexpressions of (4.7a), *dx offsety* and *dy offsetx*.

Table 4-1 dx offsety and dy offsetx

ysite	*offsety*	*dx offsety*
0	0	0
1	8	8*dx*
2	16	8*dx* + 8*dx*
3	24	8*dx* + 8*dx* + 8*dx*

xsite	*offsetx*	*dy offsetx*
0	0	0
1	8	8*dy*
2	16	8*dy* + 8*dy*
3	24	8*dy* + 8*dy* + 8*dy*

Each value in the table is eight times dx or dy, implemented as a left shift by three, added to the previous value in the table. Once these values have been obtained, (4.7a) is calculated by a single subtraction, *dx offsety - dy offsetx*, at each of the sixteen sites. Rendering processor registers store the results of these subtractions as follows:

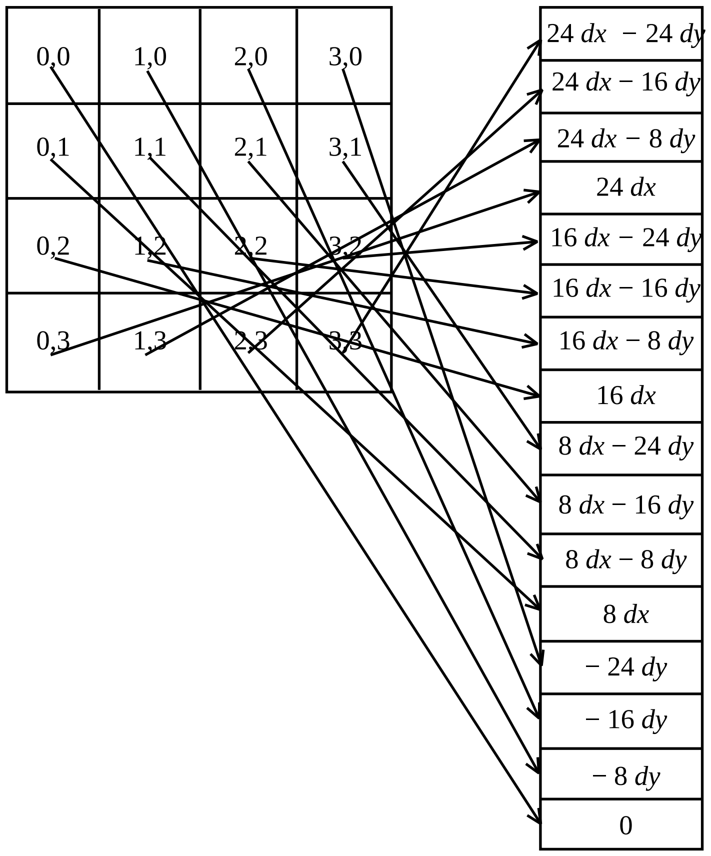


Figure 4-7 Half-space site values from expression (4.7a)

Terms (4.7b) and (4.7c) are independent of the offset within the update array. The sum (4.7b) + (4.7c) is called the *half-space constant*. The sign of the sum of the site value and the half-space constant gives the sidedness of the corresponding pixel with respect to the half-space. Since the half-space constant is independent of the site offset, the same half-space constant value is added to each of the site values.

Thus, the VLSI structure is simple. A single bus carries the half-space constant across each of the site values. Each site has a register to store its site value, which is loaded when initializing the half-space. For each update array, the circuitry at each site adds its site value to the half-space constant on the bus, and outputs the resulting sign bit. Figure 4-8 shows a block diagram of the VLSI cell for each site within the update array.

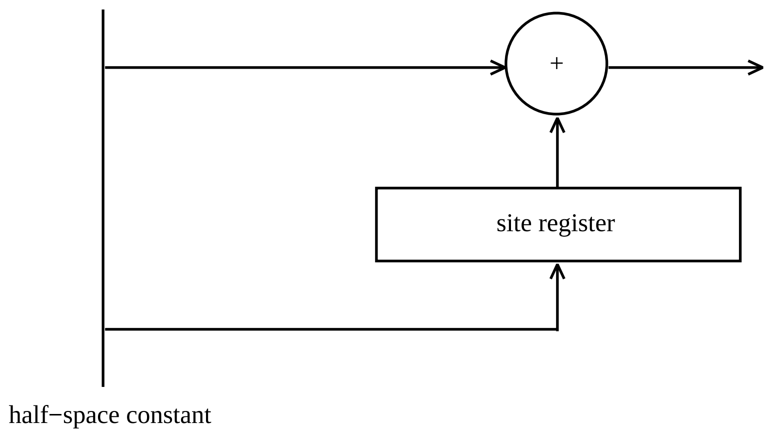


Figure 4-8 Block diagram of VLSI structure for each site within the update array

Figure 4-9 shows the collection of site cells that form the VLSI structure to perform the parallel evaluation of a half-space at each of the sixteen sites within an update array. This structure is called a *half-space evaluator*. The column of wires represents the half-space constant that is added to each of the site values. Each site, represented by a box, contains an adder and a register as shown in figure 4-8. The horizontal lines coming from each site are the resultant sign bits -- boolean values indicating in which half-space the site's pixel lies.

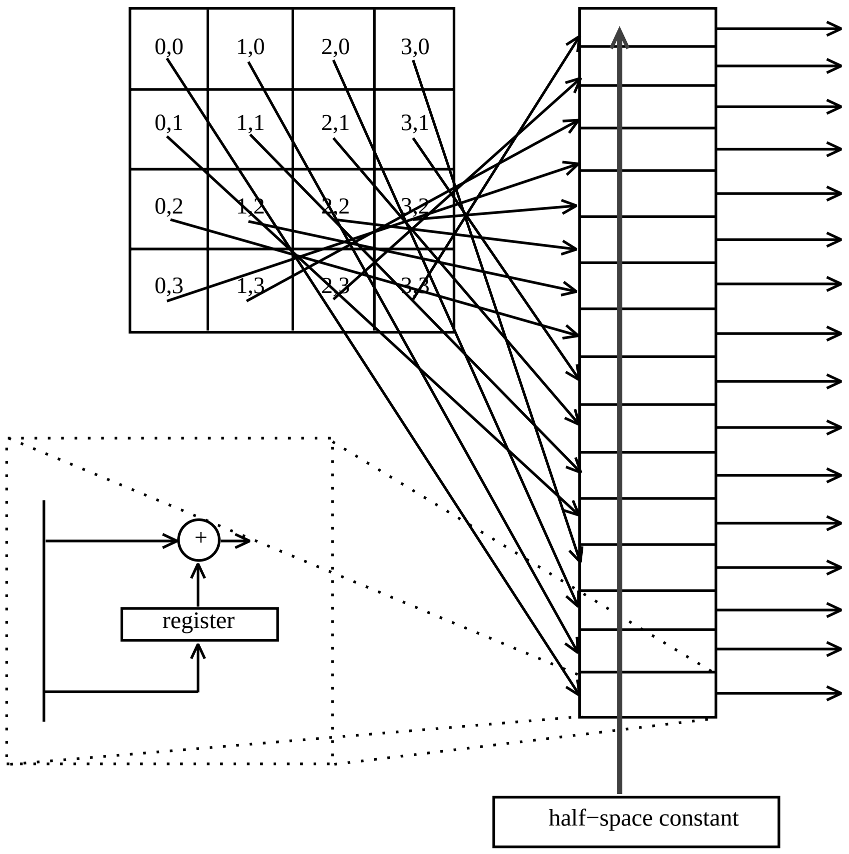


Figure 4-9 A VLSI circuit to evaluate a half-space at sixteen pixels in parallel

4.8 A VLSI Circuit for Line and Quadrilateral Evaluation

Given the half-space evaluator, construction of a quadrilateral evaluator is trivial. A quadrilateral evaluator is just four half-space evaluators, one for each edge of the quadrilateral. The boolean results at corresponding sites in four half-space evaluators are logically anded to determine the insidedness of each of the pixels. If a pixel is inside the quadrilateral with respect to all four edges, then it is inside the quadrilateral; otherwise, it is outside. Figure 4-10 shows the interaction of the individual half-space evaluators. The wires running across (left to right in the diagram) the evaluators are anded to produce the final result. Evaluation of line segments uses the same VLSI structure.

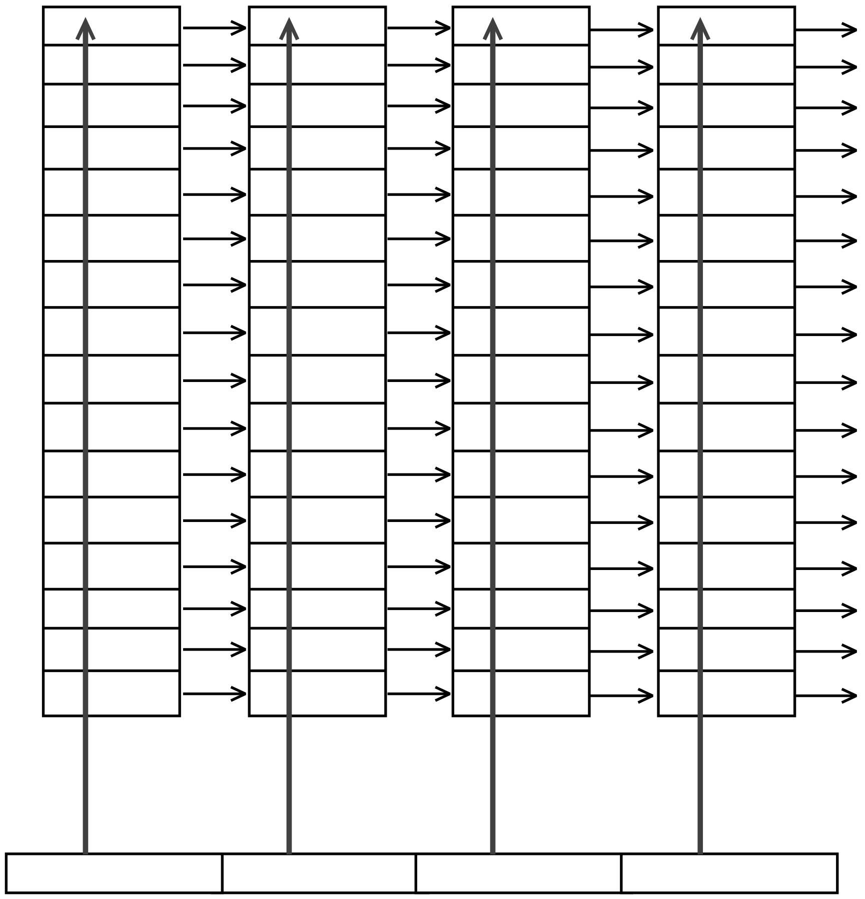


Figure 4-10 Multiple half-space evaluators used to evaluate geometric primitives

4.9 Mathematics of the Plane Equation

The half-space circuitry generalizes well to evaluating plane equations, linear equations in three dimensions. The derivation of the plane equation evaluation circuitry used by the PixelVision rendering subsystem is similar to the derivation of the half-space circuitry. Let's first look at the mathematics of plane equations. Given the general equation for a plane,

$$z = Ax + By + C, \tag{4.8}$$

and three points on the plane, *(x1, y1, z1)*, *(x2, y2, z2)*, and *(x3, y3, z3)*, we find the values of the coefficients *A*, *B*, and *C*. To do this, we solve the following system of simultaneous equations:

$$z1 = Ax1 + By1 + C$$

$$z2 = Ax2 + By2 + C$$

$$z3 = Ax3 + By3 + C$$

Solving this system of linear equations for the unknowns *A*, *B*, and *C* yields the following result:

$$z = -\frac{a}{c}x - \frac{b}{c}y + \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \tag{4.9}$$

where

$$a = (y2 - y1)(z3 - z2) - (y3 - y2)(z2 - z1)$$

$$b = (z2 - z1)(x3 - x2) - (z3 - z2)(x2 - x1)$$

$$c = (x2 - x1)(y3 - y2) - (x3 - x2)(y2 - y1)$$

Now we break the coordinates into an origin plus an offset, just as with half-space evaluation.

$$z = -\frac{a}{c}(originx + offsetx) - \frac{b}{c}(originy + offsety) + \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \tag{4.10}$$

$$z = -\frac{a}{c}originx - \frac{a}{c}offsetx - \frac{b}{c}originy - \frac{b}{c}offsety + \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \tag{4.11}$$

We rearrange the terms for discussion:

$$z = -\frac{a}{c}offsetx - \frac{b}{c}offsety \tag{4.11a}$$

$$- \frac{a}{c}originx - \frac{b}{c}originy \tag{4.11b}$$

$$+ \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \tag{4.11c}$$

Notice the time at which the three subexpressions must be evaluated. (4.11a) is independent of the origin of the update array, but has a different value at each site within an update array. Like (4.11a), (4.11c) is independent of the origin of the update array, but unlike (4.11a), it is also independent of the offset within an update array. (4.11b) is dependent on the origin of the update array. These subexpressions are analogous to those of the half-space equation. (4.11a) is called the *plane site value*; (4.11b) + (4.11c) is called the *plane constant.*

4.10 A VLSI Circuit for Plane Evaluation

The VLSI circuit to evaluate plane equations is similar to the circuit to evaluate half-spaces. Each site stores its permutation of the site value. For each update array, the plane constant is added to each of the site values. Since the plane constant is independent of the site location, the same plane constant value is added to each plane site value.

While each site of the half-space circuits output only one bit, each site of the plane equation circuits outputs a full 24-bit pixel value. The rendering subsystem can calculate in parallel either three 8-bit plane equations (e.g. red, green, and blue) or one 24-bit plane equation (e.g. depth).

The plane equation evaluator has sixteen 24-bit ALUs, one for each site within an update array. Each ALU can be configured to behave as a single 24-bit ALU or as three separate 8-bit ALUs. Figure 4-11 shows the circuitry, consisting of a register file and ALU, available at each site within the update array. The register file, holding the site value for the various plane equations, is one operand to the ALU. The other operand to the ALU is the plane constant.

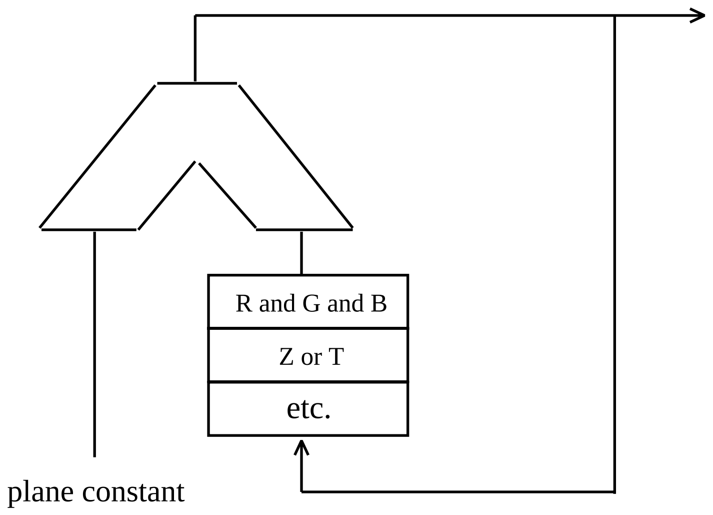


Figure 4-11   Each site has an ALU and a register file.

Sixteen of these circuits running in parallel can perform the parallel evaluation of all pixels in the 4x4 update array. The plane equation circuitry generates not only pixel values to place in graphics memory, but also boolean values which are the results of comparison operations. Figure 4-12 shows sixteen of the cells from figure 4-11. The wires exiting the right side of the structure provide the data path to graphics memory.

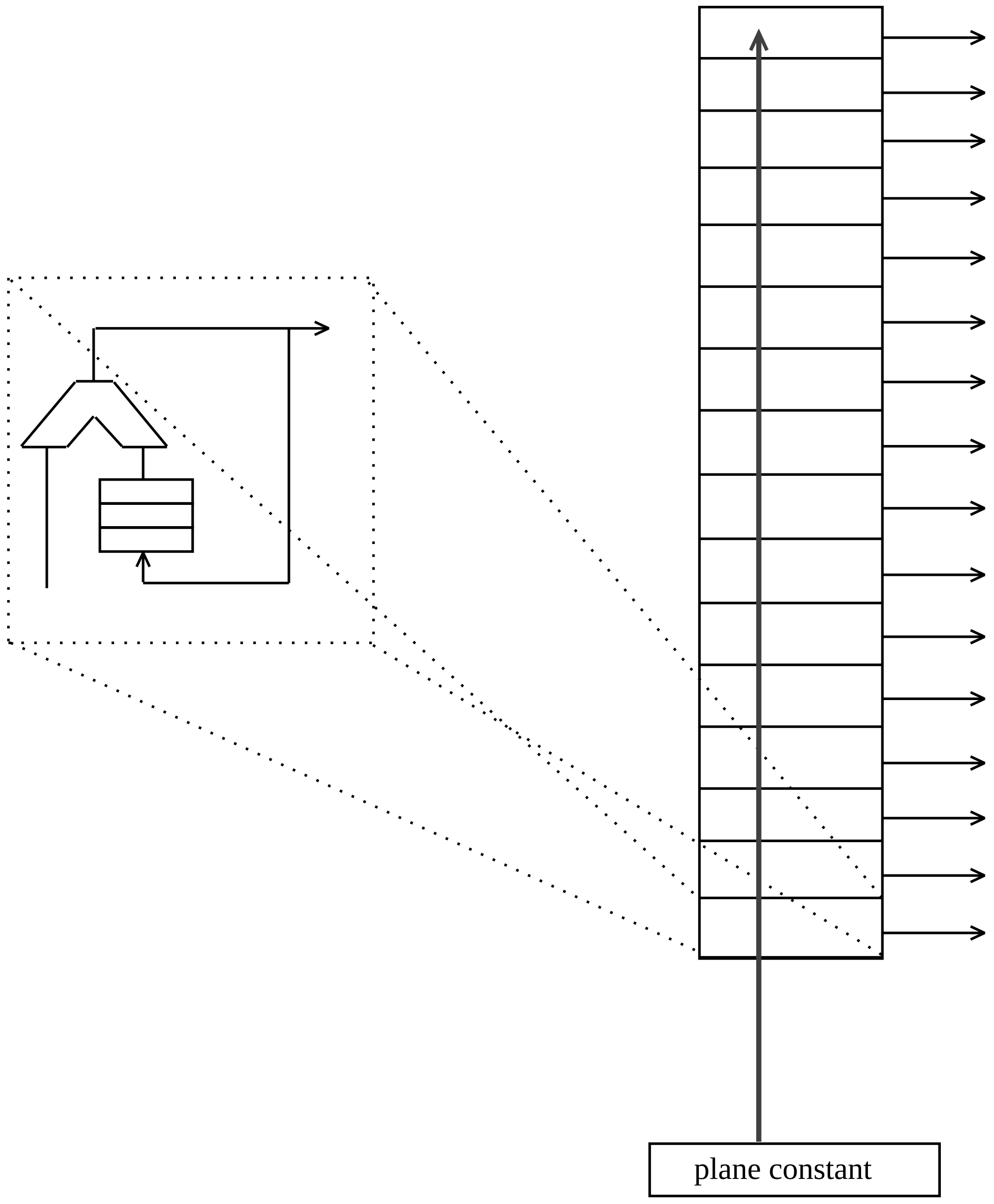


Figure 4-12   A VLSI circuit for the evaluation of plane equations

5 Tiling Geometric Primitives

Rendering geometric figures on a raster graphics workstation implies a transformation of an application's geometric data representation into the discrete pixel representation of graphics memory. This transformation is called *scan conversion* or *tiling*, and has been the focus of algorithms ranging from Bresenham's incremental line generating algorithm [4], found in most special-purpose graphics accelerators, to a smooth shaded z-buffered triangle tiler, until recently reserved for software and high-end hardware implementations. While these tilers generate radically different results, they share the common goal of *visiting* all pixels within a well-specified convex boundary. More sophisticated tilers produce high-quality antialiased images, and reproduce object properties such as texture and transparency. While these tilers are admittedly more complex, they generally share some of the framework of the basic line and triangle tilers.

The discussion of the PixelVision rendering algorithms thus far has focused primarily on what the rendering subsystem can do in a single update array. When positioned on a 4x4 area, the rendering subsystem can evaluate half-space equations to determine which pixels should be updated. Using this underlying primitive to render geometry such as lines and triangles requires a mechanism to reference all update arrays inside the geometric figure being drawn. This mechanism is the PixelVision's tiling algorithm.

5.1 Incremental Techniques for Calculating the Half-Space Constant

Let's first become reacquainted with the half-space equation,

$$\text{sidedness} = \qquad (5.1)$$

$$dx\ offsety - dy\ offsetx + \qquad (5.1a)$$

$$dx\ originy - dy\ originx + \qquad (5.1b)$$

$$- dx\ y1 + dy\ x1 \qquad (5.1c)$$

Notice that moving the origin of the update array requires that only (5.1b) be recalculated. All other terms are constant for the half-space, regardless of update array position. Moving the update array from (*originx*, *originy*) to (*originx + n*, *originy + m*) for unknown values of *n* and *m* requires two multiplies and two adds to recalculate the half-space constant. Alternatively, the PixelVision rendering subsystem can take advantage of the coherence of adjacent pixel addresses and use a technique known as forward differencing to recalculate the half-space constants. In other words, the PixelVision can incrementally update the half-space constant using predefined constant values of *n* and *m.*

Four control signals are introduced, *XInc*, *YInc*, *NegXInc*, and *NegYInc*, to control the order in which the rendering processor references update arrays. The *XInc* signal indicates that *originx* is increasing by 4 pixels, or 32 subpixel units. An *XInc* signal causes -32*dy* to be added to the half-space constant. Similarly, the *YInc* signal indicates that *originy* is increasing by 4 pixels, or 32 subpixel units. A *YInc* signal causes 32*dx* to be added to the half-space constant. *NegXInc* and *NegYInc* indicate that *originx* and *originy* decrease by 32 subpixel units, and that

32*dy* and -32*dx* respectively should be added to the half-space constant.

Scan conversion algorithms can now be layered on top of these signals, making use of local coherence of the geometric figures being rendered. Coherence and forward differencing are powerful tools in low-level computer graphics, used extensively in incremental algorithms.

5.2 Incremental Techniques for Calculating the Plane Constant

The incremental technique for calculating the plane constant is analogous to the technique for calculating the half-space constant. Looking back at the equation for planes,

$$z = -\frac{a}{c}originx - \frac{a}{c}offsetx - \frac{b}{c}originy - \frac{b}{c}offsety + \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \qquad (5.2)$$

$$z = -\frac{a}{c}offsetx - \frac{b}{c}offsety \qquad (5.2a)$$

$$- \frac{a}{c}originx - \frac{b}{c}originy \qquad (5.2b)$$

$$+ \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \qquad (5.2c)$$

shows that (5.2b) is the only term that changes with the position of the update array. Hence, an *XInc* signal adds -32a/c to the plane constant. Similarly, a *YInc* signal adds -32b/c to the plane constant. *NegXInc* and *NegYInc* add 32a/c and 32b/c respectively.

5.3 Bounding Box Tiler

The addition of *XInc* and *YInc* signals turns the PixelVision update array reference into a powerful underlying primitive for rendering complex 3D geometry. The next logical step is to develop tiling algorithms that make use of these signals. Referencing pixels outside a geometric figure isn't harmful as pixels outside the figure will not be drawn, but the tiler must hit at least all of the pixels inside the figure. A trivial tiling algorithm covers the entire bounding box of a primitive. The PixelVision rendering subsystem does not employ this algorithm because it generates unnecessary memory references (outside the geometric figure); it is shown only as a precursory example to provide insight into the supported algorithm and its numerical analysis. Figure 5-1 shows the bounding box scan conversion algorithm.

```
Initialize the half-space equations
Find the bounding box of the geometric primitive (x, y, width, height).
for (h=0; h<=height; h += ARRAY_HEIGHT) {
        oldConstants = Constants;        /* save the half- space constants */
        for (w=0; w<=width; w += ARRAY_WIDTH) {
                Draw();
                XInc();
        }
        Constants = oldConstants;        /* restore the half-space constants */
        YInc();
}
```

Figure 5-1 Bounding Box Tiling Algorithm

This algorithm sweeps out a rectangular region bounding the geometric primitive being rendered. The rendering processor initially references the update array in the upper left corner of the bounding box. XInc signals move the rendering processor to the right edge of the bounding box. After reaching the right edge, the rendering processor restores to the left edge, and a YInc signal moves it down. This continues until the update array in the bottom right corner of the bounding box has been referenced. Draw() initiates a graphics memory reference at the current update array location.

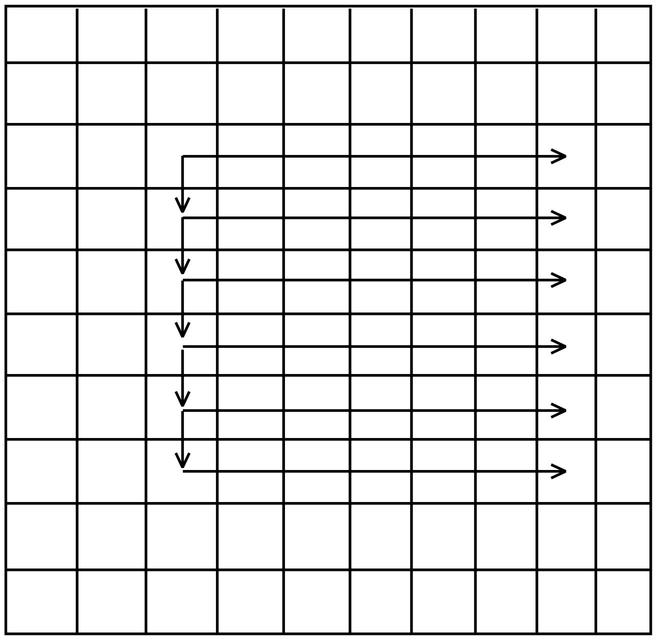


Figure 5-2 Tiling a bounding box using *XInc* and *YInc* signals

The VLSI implementation of a bounding box tiling algorithm is two synchronous loadable down counters. One of the counters runs in the *x* direction and is loaded with the width of the rectangle, while the other counter runs in the *y* direction and is loaded with the height of the rectangle. When the *x* counter reaches 0, it is reloaded and the *y* counter decrements. When both counters are at 0, the operation is complete. The counter circuitry causes *XInc* and *YInc* signals to be issued to the half-space and plane equation circuitry.

5.4 Edge-Seeking Tiler

The bounding box algorithm generates substantially more memory references than necessary for tiling geometric primitives such as lines and quadrilaterals. *Edge-seeking* is a more efficient tiler that generates no unnecessary memory references. It is loosely modeled after a flood fill algorithm, except instead of searching for a color, it searches for a geometric boundary. All geometric primitives supported by the PixelVision rendering subsystem can be rendered efficiently with a single edge-seeking algorithm.

To tile a convex figure into graphics memory, the edge-seeking algorithm sequentially references all update arrays partially within the boundaries of the figure. The rendering processor initially references any update array inside the convex boundary. Using the convex properties of the geometric figure, the edge-seeking algorithm finds and references another update array inside the figure. This continues until the algorithm has referenced every update array that is partially inside the figure, and no others. At each update array, the rendering processor draws to those pixels that are inside the figure's boundary; the half-space evaluators generate a mask that restricts drawing to the interior. When completed, the figure has been scan converted and the resulting image resides in graphics memory. Figure 5-3 shows how the edge-seeking algorithm tiles a triangle.

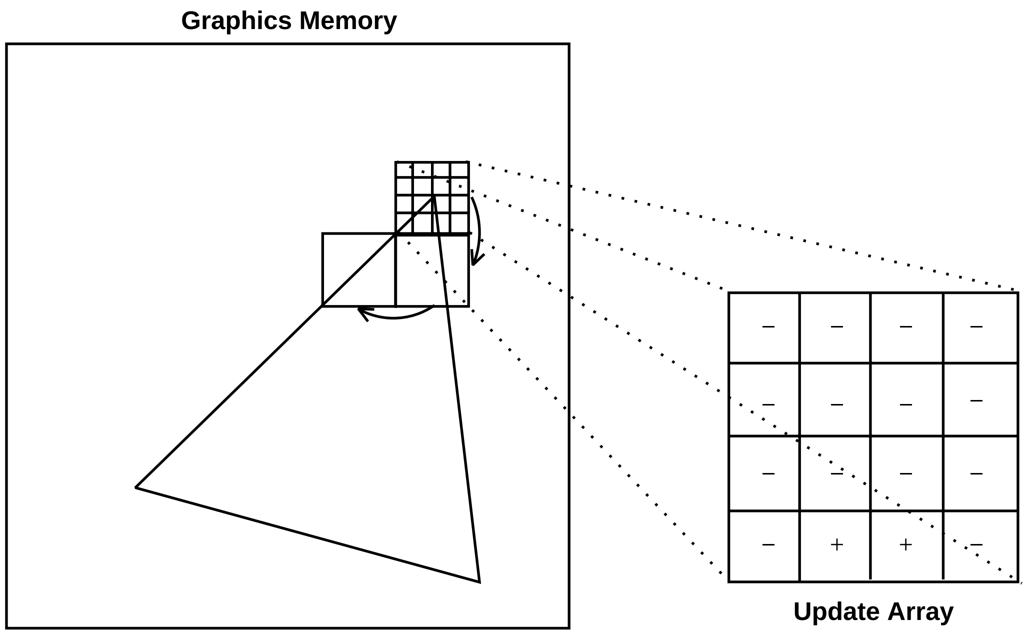


Figure 5-3 Tiling a triangle using the edge-seeking algorithm

5.4.1 The Edge-Seeking Algorithm

The half-space model provides a single low-level data representation for any convex figure, ranging from line segments to complex convex polygons. Traditional scan conversion algorithms require a different representation for different geometric shapes: the representation for line segments is significantly different from the representation of general polygonal figures. The *edge-seeking* algorithm offers a mechanism for tiling any convex figure with no detailed knowledge of the figure's shape. The algorithm works directly from the data provided by the SIMD evaluation of the half-space equations.

The goal of any tiler is to draw those pixels within the specified figure. Parallel rendering processors, where a single address to the memory system identifies more than one pixel, have the further constraint that only a portion of the pixels identified by the tiler should be written. For example, a one-pixel-wide line segment partially obscuring a 4x4 update array will include at most four of the sixteen pixels in the array if rendered using Bresenham's line drawing algorithm. The generation of the pixel mask that constrains drawing to the inside of the figure occurs outside the edge-seeking algorithm through the evaluation of the component half-spaces. The edge-seeking algorithm merely specifies an order for visitation of update arrays.

Figures 5-4 and 5-5 show the update arrays referenced by the edge-seeking algorithm, where boxes represent update arrays and numbers represent relative times. The rendering processor initially references any update array in the figure -- the first vertex is a convenient starting place. The algorithm moves to the left and stops just before exiting the figure. The algorithm then returns to its initial location and moves to the right, stopping once again just before exiting the figure. With each horizontal movement, the edge-seeking algorithm surveys the figure above and below, remembering locations from which the figure extends vertically. If the algorithm finishes with a horizontal row and has not found a location from which it can move vertically, then the scan conversion is complete. Otherwise, the algorithm moves vertically and continues scanning left and right. The algorithm moves down until it has reached the bottom vertex before returning to the initial row from which it moves upward if appropriate.

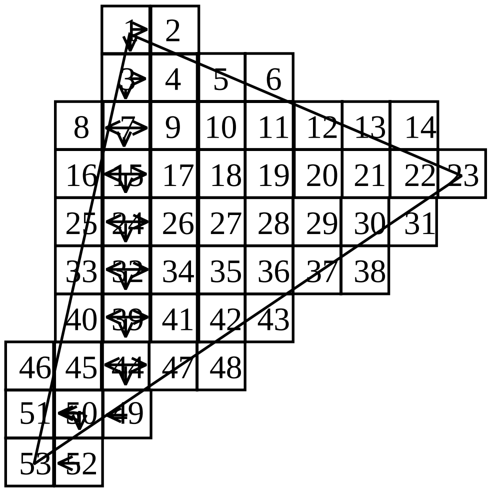


Figure 5-4   A triangle scan converted by the edge-seeking algorithm. Each square represents an update array. Arrows indicate direction of movement.

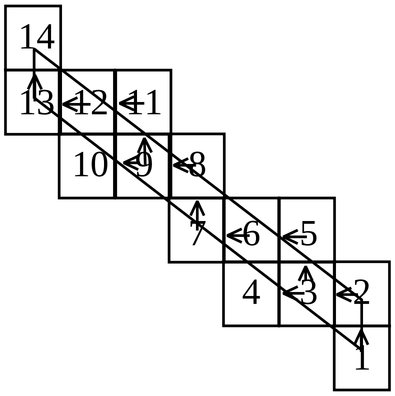


Figure 5-5   A line scan converted by the edge-seeking algorithm. Each square represents an update array. Arrows indicate direction of movement.

The feasibility of this algorithm depends on the stopping conditions. If it is difficult or slow to detect the half-space boundaries, then the algorithm is useless. At first glance, the problem seems trivial. To determine whether to move left, check the sample points along the left border. If one of the points is inside, then the figure extends into the next update array.

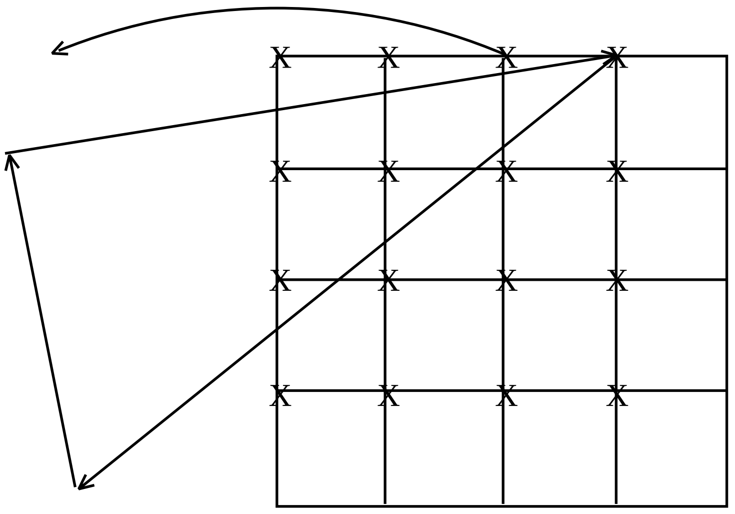


Figure 5-6   If one of the sample points along the left border of the update array is inside, then the figure extends to the left. Sample points are marked by ‘X’.

However, this criterion does not handle all cases. Since discrete pixel locations are the only points sampled, it is possible for a triangle to *slip* between pixels along the border. As shown in figure 5-7, just testing the sample points is insufficient to notice that the figure extends to the left.

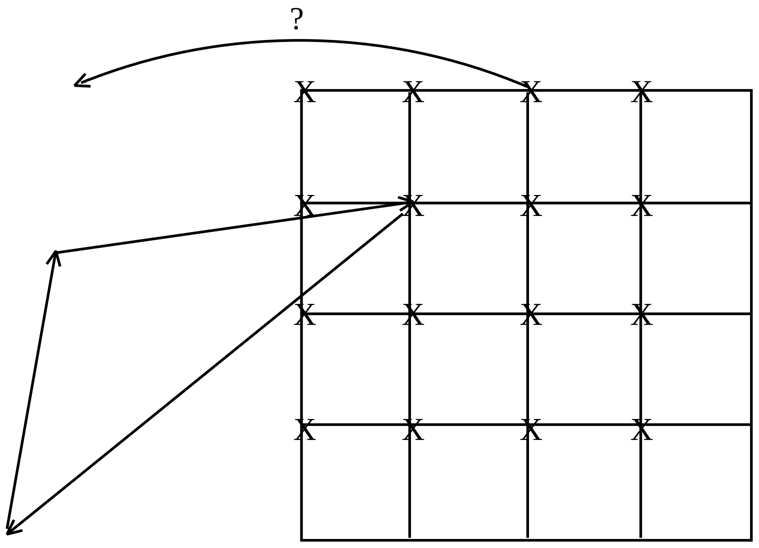


Figure 5-7   A triangle *slips* between pixels along the border indicating that the stopping criterion above is insufficient.

The correct criterion for movement to the left is if any point (not just the sample points) on the left boundary of the update array is inside. This is accomplished by testing the component half-spaces at the upper left and lower left corners of the update array. If each half-space evaluation places either one of those two samples inside the figure, then the algorithm should move left. It is not necessary that the *same* corner be inside with respect to all half-spaces, only that one of the corners is inside for each half-space. Similarly, the criterion for movement in other directions requires sampling at other corners of the update array. Since the half-space evaluators as described earlier sample only in the upper left corner of pixels, three new sample point sites must be added to their workload.

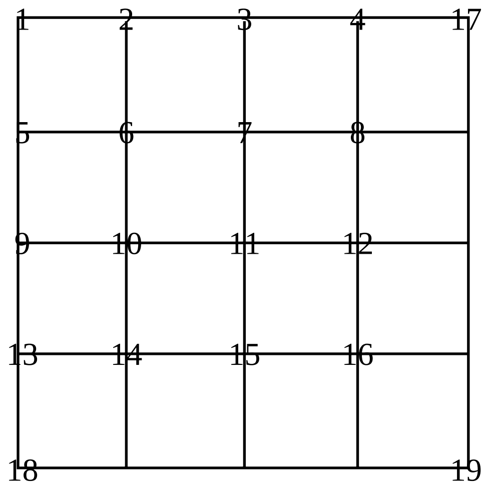


Figure 5-8 Three sample points (17, 18, and 19) are added to allow the edge-seeking algorithm to determine if edges cross update array boundaries.

The conditions for movement reduce to a set of logical operations on the results of the evaluation of the component half-spaces. The *left* condition requires that all half-spaces be inside with respect to either sample point 1 or sample point 18. Figure 5-9 depicts the logic for the condition to move left as it applies to the three half-spaces from the triangle of figure 5-7. The right, up, and down conditions are analogous.

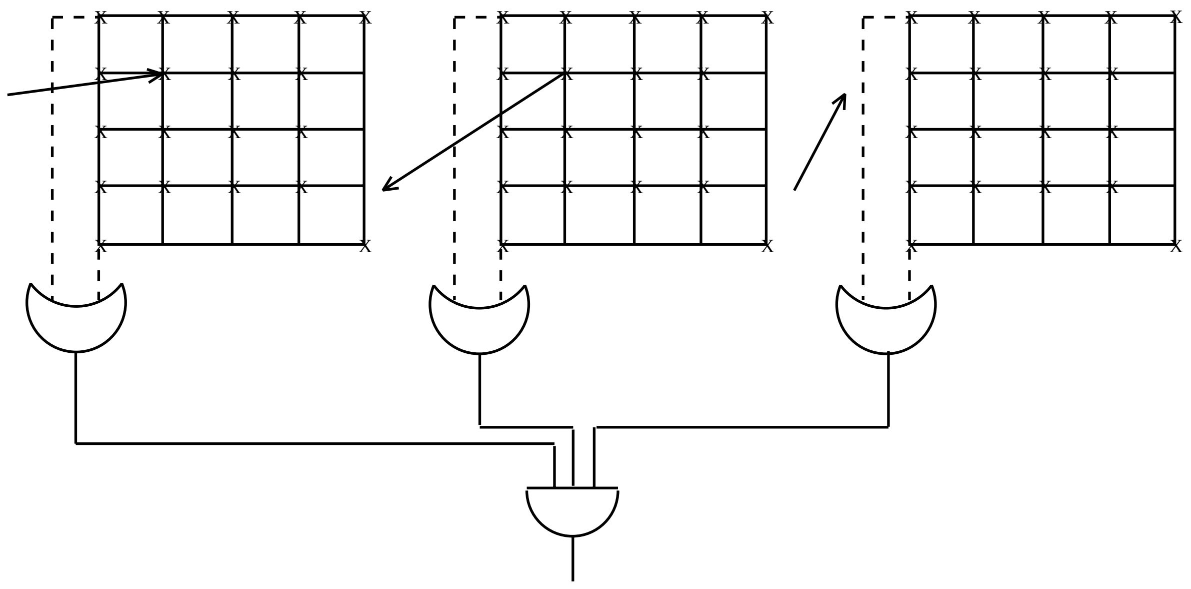


Figure 5-9 Logic for the condition to move left for the triangle of figure 5-7

The stopping criterion described does not differentiate half-space edges crossing update array boundaries inside the figure from half-space edges crossing update array boundaries outside the figure. Since individual half-space evaluators are mathematically infinite and are not bound by the geometric primitive, the constraints presented are insufficient to limit movement to the figure's interior. The primitive's bounding box should be used as a final constraint to remove this problem. Figure 5-10 shows a triangle represented as the intersection of three half-spaces, in a situation where the bounding box condition prevents unnecessarily moving to the left.

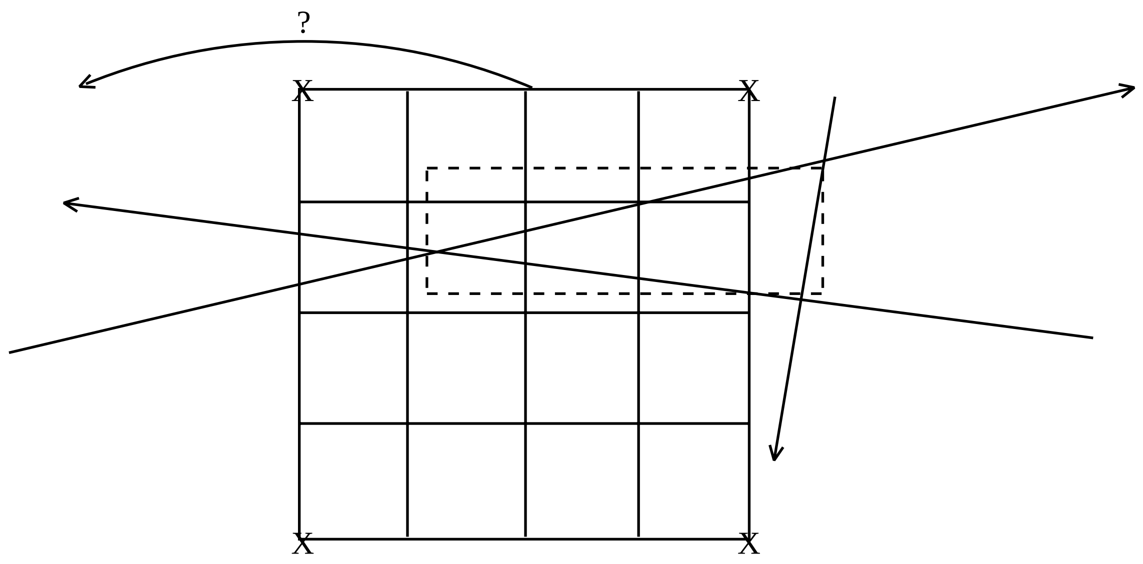


Figure 5-10 Edges going through the update array's left boundary after exiting the geometric primitive. The bounding box is used as an extra constraint to restrain movement to the left.

5.5 Geometric Models

The edge-seeking algorithm is a method of tiling convex figures. The algorithm itself does not define the convex figures. It is important for any rendering system to provide precise definitions of its primitives so its applications can understand the ramifications of the sampling inherent in the scan conversion process. This section defines the geometric primitives not through algorithmic descriptions, but through higher-level models of the sampling process.

5.5.1 Quadrilateral Model

An important characteristic of a quadrilateral scan conversion algorithm is the description of exactly which pixels are considered ‘inside’ the quadrilateral -- this is the *quadrilateral model*. The quadrilateral scan conversion process represents a transformation from a mathematical representation to a discrete pixel representation. In other words, scan conversion is the execution of an algorithm for determining the set of pixels that are inside a quadrilateral given its geometric description. Antialiasing is ignored: insidedness of a pixel in a quadrilateral is a boolean decision. That is not to say that PixelVision quadrilaterals are black and white. The premise is simply that the resolution is fixed (no antialiasing) and that the quadrilateral model is distinct from the pixel values placed in graphics memory.

An important property in scan converting quadrilaterals is that adjacent quadrilaterals *touch cleanly*. The pixels along an edge shared by adjacent quadrilaterals should be included in one and only one of the quadrilaterals. Relating this to the transformation implicit in scan conversion, objects that are mutually exclusive in geometric space should remain mutually exclusive in pixel space.

The PixelVision’s quadrilateral model states that a quadrilateral is composed of the set of all pixels whose coordinate values are inside the quadrilateral’s boundary. The only pixels whose insidedness is in question are those whose coordinate values lie directly on one of the quadrilateral’s edges. Keeping in mind our goal of cleanly touching quadrilaterals, it seems reasonable to simply include left edges but not right. Applying the math from the half-space chapter, if the half-space equation evaluates to zero (indicating that a pixel is exactly on the edge), then it is marked as a ‘+’ for a left edge, and a ‘-’ for a right edge. The rendering subsystem examines edges in a clockwise direction to guarantee that the positive side of a half-space is always towards the inside of the quadrilateral; therefore, a right edge has the property that its second vertex has a larger *y* value than its first vertex, while the opposite is true of a left edge.

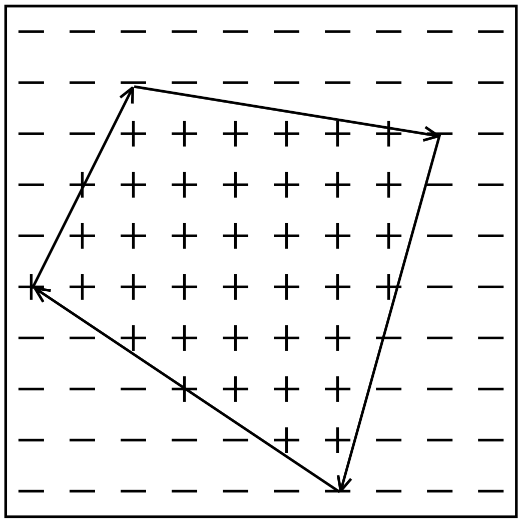


Figure 5-11 Quadrilateral Model

The convention of including pixels on the left edge but not the right edge does not define insidedness of pixels on horizontal edges, since the definitions of left and right break down. For horizontal edges, the PixelVision quadrilateral model includes top edges, and excludes bottom edges. A bottom edge is a horizontal edge whose second vertex is to the left of its first vertex; a top edge is any other horizontal edge. Minimum and maximum vertices belonging to both an edge that is included and an edge that is excluded are excluded.

5.5.2 Line Model

To better understand the design of the PixelVision's line algorithm first requires a description of what is generally considered the best looking set of pixels describing a line segment on a raster device. For a one-pixel-wide line segment, Bresenham's integer algorithm defines these pixels. The PixelVision algorithm generates the same pixel set as Bresenham's algorithm. A line segment from ($x1$,$y1$) to ($x2$,$y2$) includes pixels as described below.

If the absolute value of ($x2$-$x1$) is greater than the absolute value of ($y2$-$y1$), then x is the major axis of the line and y is the minor axis. If x is the major axis , the pixel representation for the line segment is defined as the set of pixels, one for each discrete value of x between $x1$ and $x2$, that are closest to the line. One and only one pixel is chosen for each value of x between $x1$ and $x2$. In the case where two pixels are equidistant from the line, a *screen relative* decision is made so that a line drawn from ($x1$,$y1$) to ($x2$,$y2$) is the same as a line drawn from ($x2$,y2) to ($x1$,y1). An analogous definition holds for lines where y is the major axis and x is the minor axis.

Bresenham's algorithm is an efficient method for choosing these pixels; however, it makes no use of the parallelism and high bandwidth to graphics memory available in the PixelVision rendering subsystem. PixelVision generates the same pixels as Bresenham's algorithm by modeling a line as a thin polygon. An infinite one-pixel-wide line can be modeled as the intersection of two half-spaces with parallel boundaries separated by one pixel. Two more half-spaces recognize the endpoints of the line segment. If the x axis is the major axis of a line segment from (*x*1, *y*1) to (*x*2, *y*2), then the two parallel half-space boundaries are defined by (*x*1, *y*1-.5) to (*x*2, *y*2-.5) and (*x*2, *y*2+.5) to (*x*1, *y*1+.5).[1] If the y axis is the major axis, then the two half-spaces are (*x*1-.5, *y*1) to (*x*2-.5, *y*2) and (*x*2+.5, *y*2) to (*x*1+.5, *y*1). Depending on the direction of the line segment (which endpoint is the first endpoint), the half-space orientation may need to be reversed. The endpoints are detected by two more half-spaces as shown in figure 5-12.

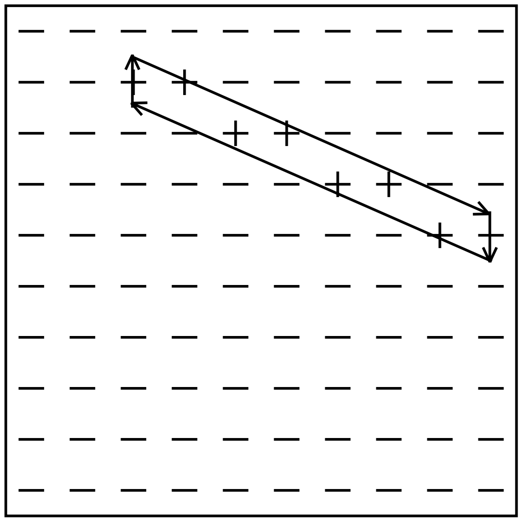


Figure 5-12 Model of a line segment. The top and bottom edges are exactly one pixel apart along the minor axis of the line.

The line model includes all pixels that are inside the quadrilateral established by the four half-spaces. Just as the boundary pixels are special cases in the quadrilateral model, so are they in the line model. The first endpoint of a line segment is always included as part of the line. Inclusion of the second endpoint is selectable by the *line cap style* in the rendering subsystem. To guarantee that one and only one pixel is chosen per step along the major axis, one of the other two half-space evaluators should include boundary pixels and the other should not. In summary, the half-space evaluators act as follows:

Endpoint 1 half-space:	include boundary pixels
Endpoint 2 half-space:	inclusion of boundary pixels determined by line cap style
top half-space for x-major lines, left for y-major:	include boundary pixels
bottom half-space for x-major lines, right for y-major:	exclude boundary pixels

In addition to rendering one-pixel-wide lines, the PixelVision can render wide lines. The wide line model is identical to the one-pixel-wide line model, except instead of offsetting vertices by .5 from the true line, they are offset by a user-specified width. Wide lines are particularly useful for implementing anti-aliased line algorithms.

[1]See the section on subpixel positioning to understand fractional pixels.

5.5.3 Point Model

A single vertex *(x, y)* parametrizes a point to the PixelVision rendering subsystem. The model of a point is based on the model of a quadrilateral. Specifically, a point is a quadrilateral with vertices *(x, y)*, *(x+1.0, y)*, *(x+1.0, y+1.0)*, and *(x, y+1.0)*. Here, "1.0" represents one pixel, or eight subpixel units.

6 High-Quality Rendering Techniques

As the field of computer graphics grows, the quest for visual realism strengthens. Techniques such as ray tracing, texture mapping, antialiasing, transparency, and many others have been used effectively to produce realistic images. These techniques have been predominantly software oriented, or have made use of very expensive special-purpose hardware. The computational simplicity of the PixelVision geometry coprocessor and the geometric simplicity and rich graphics memory update technology in the PixelVision rendering subsystem provide a good opportunity to move some of the advanced rendering techniques into hardware.

Because of their complexity, high-quality photorealistic renderers including ray tracing, zz-buffer, and procedural shading languages have typically been implemented in software running on a host CPU. Simpler (and lower-quality) photorealistic techniques such as antialiasing and transparency have recently gained special-purpose hardware support. These are really two different classes of renderer: one for photorealistic computer graphics and one for high-quality interactive computer graphics.

6.1 Photorealistic Rendering

Photorealistic rendering techniques include algorithms such as ray tracing, zz-buffer, and complex shaders embedded in rendering systems such as the RenderMan shading language. Photorealism requires, more than anything else, general-purpose floating point computes. The techniques are so varied that special-purpose hardware in the form of a dedicated pipeline would be too costly and too specific to justify the cost. Only those few applications that need exactly what the silicon offers would reap the benefits; other applications would just pay the costs.

The PixelVision architecture assists with photorealism through the geometry coprocessor by providing programmable floating point computes for application use. The PixelVision geometry coprocessor is well-suited to assist with photorealistic rendering operations for several reasons. First, it has the raw floating point computational capacity to dramatically accelerate floating point applications such as photorealism. Second, it is a highly-integrated solution so the cost is low relative to other special-purpose approaches. Finally, and perhaps most importantly, it is easily controlled by programs running on the CPU so that the special-purpose programming effort is manageable.

This document does not discuss specific photorealistic algorithms because their implementations vary so greatly and they mature so quickly that it is impossible to predict the needs in the future. This section acts merely to recognize the importance of high floating point performance for photorealistic rendering. The remainder of this chapter concentrates on specific algorithms for high-quality interactive graphics.

6.2 High-Quality Interactive Graphics

While photorealistic techniques can provide visually dramatic results, the results are often too expensive for interactive applications. On the other hand, as technology improves, higher-quality rendering techniques than those traditionally thought of as interactive are becoming interactive. This section discusses some of those techniques as they apply to the PixelVision rendering subsystem.

6.2.1 Antialiasing Polygons

The sampling involved in transforming a geometric shape from a continuous geometric representation to a discrete pixel representation causes defects such as jagged edges and lost detail. The problem is referred to as aliasing, and is caused by an undersampling of the geometry. Basic raster graphics algorithms sample polygons only at discrete pixel coordinates. Aliasing is particularly apparent on low-resolution devices where the sampling frequency is lower. As the resolution increases, the ability of the human eye to detect the aliasing decreases. This section describes the antialiasing technique for polygons used by the PixelVision rendering subsystem.

The PixelVision rendering subsystem uses an antialiasing technique described in *Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique* [5], by Abraham Mammen of Stellar Computer. The algorithm has the same effect as a supersampling algorithm. Supersampling implies sampling geometry at a higher resolution than the display hardware supports, and filtering down to display resolution using a convolution filter kernel. Supersampling at *n* times the display resolution requires *n* times the graphics memory. Since graphics memory is a costly commodity, supersampling is a costly approach.

The antialiasing technique described in [5] is an incremental approach that is equivalent to supersampling, but does not suffer from the same memory burdens as supersampling. Just as with the supersampling algorithm, this technique uses a convolution filter kernel. The algorithm makes one pass over the application data structure for each location in the convolution filter kernel. Each pass jitters the scene in *x* and *y* by a predefined amount less than one pixel. In particular, given an MxN filter kernel, the pass corresponding to location (m,n) in the kernel jitters the scene by m/M pixels in *x* and n/N pixels in *y*. Each pass is rendered into a *scratch* pixelMap and then merged into the *refined* pixelMap using the corresponding filter coefficient.

This approach has several benefits over other antialiasing algorithms. First, since the algorithm requires only one pixelMap more than a standard z-buffer algorithm, the graphics memory requirements are significantly lower than with supersampling techniques. Secondly, the memory requirements are fixed at one additional pixelMap regardless of the filter kernel size. Third, as is the case with some antialiasing algorithms, polygons in the database do not need to be spatially sorted because the z buffer is still used. And finally, by viewing the results as the image is integrated, the technique offers graceful successive refinement in image quality.

6.2.2 Antialiasing Lines

The constraints in rendering line segments differ from those in rendering polygons in that line segments do not represent filled areas, although they are modeled as such. Polygons have well defined borders, and those borders must be respected by rendering algorithms, especially where polygons abut. Line segments do not have well defined borders, and the concept of abutting line segments does not exist. This extra degree of freedom motivates a higher-performance antialiasing algorithm specific to line segments.

To draw an aliased line segment of color C, each of the pixels identified as *on* the line should be given the value C. If the line segment is smooth shaded (depth cued) from color C1 at vertex P1 to color C2 at vertex P2, then the color values should be linearly interpolated between the two endpoints. Whether smooth shaded or not, aliasing artifacts will be apparent.

The following procedure will lessen these artifacts. The first step is to construct two polygons that share the line segment P1P2 as a common edge. The polygons are described by the vertex lists (P1,P2,P4,P3) and (P1,P2,P6,P5) as shown in figure 6-1. P3, P4, P5, and P6 are offset parallel to the y axis for x-major line segments and parallel to the x axis for y-major line segments. The distances from P1 to P3, from P1 to P5, from P2 to P4, and from P2 to P6 are the same. The distance is arbitrarily assigned a value of two pixels in the example; however, experimentation will show what values give pleasing results.

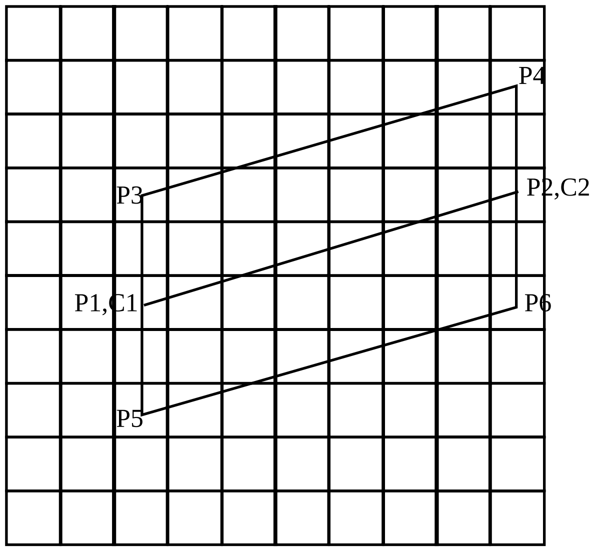


Figure 6-1 An antialiased line from P1 to P2 is constructed of two polygons with vertex lists (P1,P2,P4,P3) and (P1,P2,P6,P5). P1 has color C1. P2 has color C2.

The two polygons can be scan-converted independently. Let’s look at the scan-conversion of polygon (P1,P2,P4,P3). Polygon (P1,P2,P6,P5) is its mirror image. As the polygon is scan-converted, the selection of color values for the interior pixels is done in three steps. The first step is to linearly interpolate color values from C1 at P1 and P3 to C2 at P2 and P4. The second step is to linearly interpolate a value, referred to as α, from 1.0 at P1 and P2 to 0.0 at P3 and P4. The final step is to multiply the interpolated color value by the interpolated α value at each pixel to arrive at the new color value for the pixel. This new color value should be merged into graphics memory as described later. Figure 6-2 shows some α values of the polygons of figure 6-1.

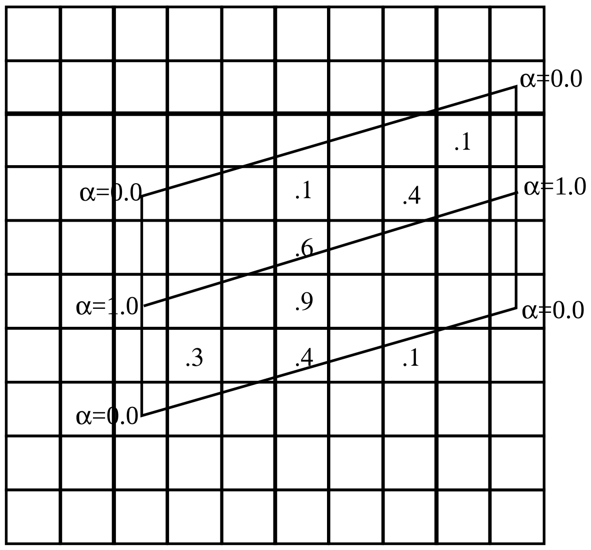


Figure 6-2 α values of figure 6-1. Vertex values are shown as well as select internal values, calculated in the upper left corners of pixels.

The algorithm’s performance can be increased by scan converting the two polygons as a single polygon described by the vertex list (P3,P4,P6,P5). In fact, a wide line from P1 to P2 of width four pixels can conveniently model this polygon. Scan converting polygon (P3,P4,P6,P5) requires the same three steps as before: interpolating color values, interpolating α values, and multiplying the color values by the α values. Of the three steps, interpolation of α values is the most difficult, as α must vary from 0.0 along the edge P3P4 to 1.0 in the center along P1P2 and back to 0.0 along the edge P5P6. This is not a linear interpolation.

Fortunately, there is a clever trick for interpolating the α values. Assuming a fixed point internal data representation for α, this can be accomplished by using an extra bit in the interpolator’s data path, and linearly interpolating from 0.0 along P3P4 to 2.0 along P5P6. Whenever a pixel’s α value is between 1.0 and 2.0, it is replaced by (2.0 - α). This effect is approximated by XORing each bit in the fixed point representation for α with the high-order (extra) bit of that representation, and occurs automatically in the PixelVision rendering subsystem when an 8-bit plane equation value (α in this case) exceeds its 8-bit range.

The description above presents a method for generating the color values for pixels included on an antialiased line segment. Once those color values have been computed, they must be combined with the values in graphics memory in such a way that overlapping line segments do not cause objectionable patterns. In general, the brightest of two overlapping line segments should persist, regardless of the order in which they are drawn. Straightforwardly copying the new color values into graphics memory will not suffice, as darker areas at the edges of a new line segment will persist over brighter areas of a previous line segment.

There are many possible methods for combining pixel values. We will outline one here which yields good quality and reasonable performance. The task of choosing the brightest of two overlapping line segments can be viewed as essentially a hidden surface problem, except instead of qualifying the graphics memory update with a depth comparison, it is qualified with a luminance comparison. In addition to interpolating color values and α values, the algorithm interpolates luminance values, which are computed from the color values at the endpoints. Just as a pixel's color value is the interpolated color value times the interpolated α value, its luminance value is the interpolated luminance value times the interpolated α value.

6.2.3 Transparency and Thickness

The PixelVision rendering subsystem achieves transparent polygons by evaluating a transparency plane equation, and adding an additional stage to the graphics memory update process. System software, such as PEX or G, calculates transparency values for the polygon vertices, and the PixelVision rendering subsystem interpolates those values across the polygon's surface. The transparency of an object at a vertex depends on the thickness of the object and the angle between the eye and the surface normal at that vertex. Refer to a software specification (PEX or G) to understand the vertex transparency calculation.

The PixelVision rendering subsystem allows the implementation of the transparency algorithm described in the aforementioned paper, *Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique* [5]. The algorithm boasts orthogonal use of z-buffering, antialiasing, and transparency. Furthermore, transparent polygons need not be sorted prior to rendering, as is the case with other transparency algorithms.

The algorithm sorts both opaque and transparent polygons during rendering using an additional z buffer (called the zsort buffer) to aid in the depth sort for transparent pixels. Thus, the algorithm requires three pixelMaps: one for color, one for the z buffer, and one for the zsort buffer. First, the algorithm renders all opaque polygons in the scene using the color pixelMap and the z-buffer pixelMap. Then the algorithm makes multiple passes over the application data structure, one pass for each *layer* of transparency.

Each pass over the application data structure consists of two stages. The first stage concerns finding, for each pixel on the screen, the farthest transparent pixel from the viewer that is closer to the viewer than the corresponding opaque pixel. The algorithm initializes the zsort buffer at the beginning of each pass and updates it during the first stage to store the depths of these transparent pixels.

The second stage involves updating the color pixelMap and z-buffer pixelMap. The depth values of those pixels updated in the first stage are copied from the zsort buffer to the z buffer, and the color values are computed with a second traversal of the application data structure. The second traversal searches for the triangles that match those stored in the zsort buffer, and uses the color values of those triangles to compute the new color values as follows:

$$c_{new} = \text{transparency} \times c_{tri} + (1 - \textit{transparency}) \times c_{old} \qquad (6.1)$$

where

c_{new} = color value to be put into the color pixelMap;

transparency = transparency value from the triangle in the zsort buffer;

c_{tri} = color value from the triangle in the zsort buffer; and

c_{old} = color value previously in the color pixelMap.

After each pass, the transparency layer rendered in that pass becomes the new opaque boundary. The algorithm makes passes over the application data structure until all transparency layers have been rendered, at which time the resultant image resides in the color pixelMap.

6.2.4 Constructive Solid Geometry

Constructive solid geometry (CSG) refers to algorithms for the representation and display of set operations, such as union, intersection, and difference, on geometric volumes. For example, a bowling ball can be represented as three cylinders (for the finger holes) differenced from a sphere. Special scanline algorithms are typically used to render CSG databases, performing set operations and resolving hidden surfaces in the process. Unfortunately, scanline algorithms are not amenable to implementation in inexpensive special-purpose hardware.

Jack Goldfeather, Jeff Hultquist, and Henry Fuchs have shown a more interactive approach in *Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System* [6]. The described algorithm uses a hidden surface technique similar to a standard z-buffer algorithm. The first step of the algorithm is to transform an arbitrary CSG tree into an equivalent CSG tree that is the union of simpler subtrees. The transformed tree has the properties that all union operator nodes have parents that are also union operator nodes, and all difference and intersection operator nodes have right children that are leaf nodes (primitives).

Once the tree has been transformed, the image can be composed in steps using the following rules:

1. (Prim1 u Prim2) is rendered using a standard z-buffer algorithm.
2. (Prim1 n Prim2) = (Front(Prim1) - Prim2) u (Back(Prim2) n Prim1).
3. (Prim1 - Prim2) = (Front(Prim1) n Prim2) u (Front(Prim2) n Prim1).

Prim1 and Prim2 refer to leaf nodes of the CSG tree. "u", "n", and "-" refer to the CSG operators union, intersection, and difference. Front(Primitive) and Back(Primitive) refer to the front-facing and back-facing surfaces of "Primitive". The paper shows how all three operators can be resolved with hardware similar to standard z-buffer hardware, given sufficient graphics memory. The PixelVision rendering subsystem has all of the characteristics required to implement this algorithm, including sufficient memory through its virtual memory paradigm and sufficient computational instructions through its rendering interface. Refer to the paper for more detail.

6.2.5 Texture Mapping

Texture Mapping refers to the rendering process of mapping a property from a multi-dimensional array (a two-dimensional array for our purposes) to a surface. The technique has been applied to properties such as color, surface normal, specularity, transparency, illumination, and surface displacement.

All rendering algorithms described thus far (z-buffering, smooth shading, transparency, CSG, antialiasing, and so on) have a consistent relationship between the pixelMaps referenced by the algorithms. The rendering of a pixel (x,y) may depend on data from several pixelMaps, but it will depend only on pixel (x,y) from each of those pixelMaps. For example, a z-buffering operation requires two pixelMaps -- one for depth information and one for color information. The rendering of pixel (x,y) depends only on the initial and intermediate values of pixel (x,y) in the color and depth pixelMaps, and not on any other pixel's initial or intermediate values.

The arbitrary mapping implicit in texture mapping causes a different relationship between the algorithm's pixelMaps. In particular, the rendering of a pixel (x,y) may depend on any pixel in a texture pixelMap, not just pixel (x,y) in the texture pixelMap. This implies that for each pixel, the rendering algorithm must be able to randomly access the texture pattern. To perform this operation with a parallel interface such as that between the PixelVision rendering processor and its graphics memory system would waste the available parallelism. For this reason, the PixelVision rendering processor gives no assistance with texture mapping.

Instead, the CPU and its memory system paired with the PixelVision geometry coprocessor can implement texture mapping. The main memory system is better-equipped to perform random accesses to pixelMaps than the PixelVision memory system. As it performs these random accesses serially for each pixel rendered, the CPU prepares the geometry coprocessor for parallel computation. The nature of the computation depends on the property being texture mapped. As an example, bump mapping algorithms map the surface normal from a texture map to a surface and perform the shading computation for each pixel. In this case, each floating point unit in the geometry coprocessor operates on a different pixel, and the shading computation is performed in parallel.

6.2.6 Shadows

Like the transparency and antialiasing algorithms described, we also promote a shadowing algorithm that makes use of standard z-buffering hardware. The technique was originally proposed by Lance Williams in *Casting Curved Shadows on Curved Surfaces* [30]. The algorithm requires two passes over the application data structure. The first pass renders a view of the scene from the point of view of the light source, storing depth values but not color values. The second pass renders the scene from the point of view of the observer's eye. At each pixel generated in the second pass, the point is transformed into the computed view in the light source space and tested for visibility in that space. If it is not visible, then it is in shadow and its color value should be appropriately attenuated.

The algorithm described is difficult for a hardware rendering subsystem to implement efficiently because each pixel generated must be transformed into a different space. As with texture mapping, the memory access in that different space is random, and therefore not well-suited to the PixelVision parallel memory organization.

Williams describes a modification to the algorithm that is appropriate for implementation in the PixelVision rendering subsystem. The modified algorithm renders the scene independently from both the observer's eye and the light source. Since the views are rendered independently, the PixelVision rendering subsystem is well-equipped for the task. As a postprocessing step performed by a general-purpose CPU, the pixels rendered from the point of view of the observer's eye are transformed into the light source space. If they are farther from the light source than the corresponding pixels rendered from the light source's point of view, then they are in shadow and their color values appropriately attenuated.

An unfortunate problem with this modification to the original algorithm is its inability to properly render specular highlights. Since the algorithm renders the initial scene independently of the light source's view, specular highlights are created in shadowed area where they should not exist. The algorithm can be further modified to resolve this problem with the addition of an extra pass through the application data structure and an additional pixelMap. The first two passes are as before, rendering the scene from the same two points of view, except only the depths are calculated in both renderings. Next, the CPU maps the depth values from the eye space to the light source space so that visibility tests can determine which pixels are in shadow. During this pass over the depth pixelMaps, the CPU generates another pixelMap, called the *attenuation pixelMap*, to store a percentage of each pixel's color value that should persist in the final image. If the pixel is not in shadow, the percentage will be 1.0; otherwise it will be a value between 0.0 and 1.0. Note that soft shadows or antialiased shadows can be represented by the attenuation pixelMap by varying the attenuation values based on the percentage of the pixel in shadow. As a final step, the color values are calculated with a a final pass through the data base in which color values are attenuated by the attenuation pixelMap. Those pixels in shadow are rendered without the specular component to the shading calculation.

6.2.7 Stereoscopic Viewing

Monoscopic "3D" images present the viewer with depth cues through shading, shadows, hidden surface removal, and perspective. Stereoscopic viewing gives the viewer another depth cue -- parallax. Parallax refers to the delivery of a slightly different image to the viewer's left eye than to the right eye, accounting for the slightly different viewing position and viewing direction.

PixelVision achieves parallax in conjunction with a special stereoscopic monitor and special glasses. The graphics memory system and monitor alternate the two images on the display, allowing the left image to project only to the left eye and the right image only to the right eye. Special glasses block the left image from the right eye and the right image from the left eye.

Since a stereoscopic image is effectively two distinct monoscopic images, a stereoscopic system requires either twice the video bandwidth, half the display resolution, half the refresh frequency, or some combination of the above. PixelVision implementations may differ in their approaches to stereo viewing. Refer to implementation specifications for more detail.

7 Geometry Coprocessor Interface

This chapter describes the PixelVision geometry coprocessor. The geometry coprocessor is a SIMD (Single Instruction Multiple Data) floating point coprocessor with four identical single-precision floating point units. Each floating point unit has its own register file and execution units. Additionally, there is a global register file accessible to all floating point units. This chapter describes the architectural characteristics of the geometry coprocessor, including the instruction set and register definitions. Implementation details are deferred to implementation specifications.

7.1 Geometry Coprocessor Overview

In the past, base workstations did not have sufficient computational power to achieve dynamic shaded graphics. As a result, graphics engineers found it necessary to build acres of special-purpose hardware and reams of device-dependent software to achieve dynamic performance levels. Geometry subsystems for graphics workstations have often been implemented as processors separated from the CPU by an IO bus. They have implemented their own instruction sets, program counters, physical memory systems, and even virtual memory systems.

Today, as general-purpose computing continues to push performance levels higher, it becomes possible to offer high-performance graphics subsystems in price bands previously thought of as the low-end. It also becomes more convenient to engineer graphics hardware and software, translating into faster time-to-market. PixelVision's geometry subsystem moves the computational responsibility closer to the CPU, where general-purpose mechanisms perform roles previously reserved for special-purpose mechanisms.

PixelVision's geometry subsystem is a coprocessor to the CPU, not a full geometry subsystem in the traditional sense. The coprocessor does not require a special-purpose memory system, and does not have its own integer unit. The coprocessor hardware consists of four single-precision floating point units operating concurrently. A single instruction stream, which originates at RAM internal to the coprocessor, controls the FPUs. Each FPU has its own register file, and there is an additional globally-accessible register file. Figure 7-1 shows a high-level block diagram of the geometry coprocessor.

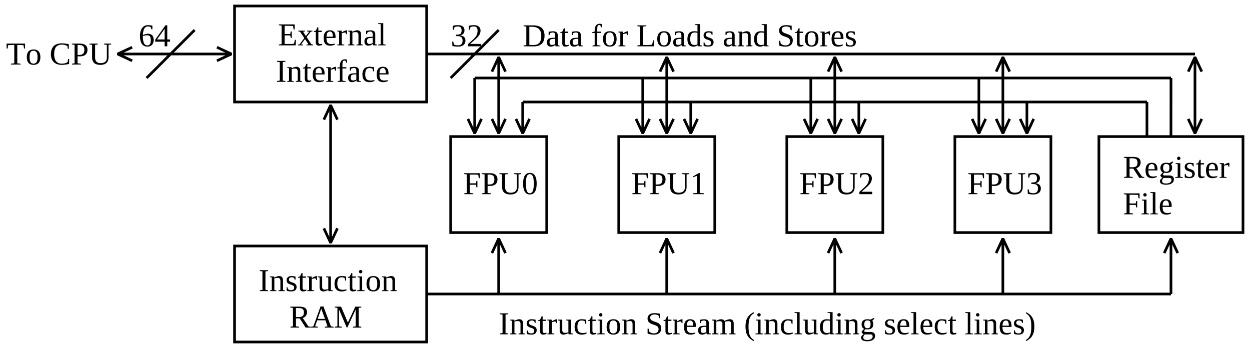


Figure 7-1 Geometry Coprocessor Block Diagram

7.2 Geometry Coprocessor Registers

The PixelVision geometry coprocessor presents an interface to the CPU through a set of 32-bit memory mapped registers. The *computational registers* store all of the geometry coprocessor's data. The coprocessor also implements an *implementation and revision register*, a *compare register,* three *instruction registers,* two *DMA registers,* and 1,024 32-bit words of *instruction RAM*. The implementation and revision register allows software to determine on which implementation and revision it is executing. The compare register stores the status bits from compare instructions so that they may be read subsequently by the CPU. The DMA registers allow data to be transferred directly between the geometry coprocessor and the the main memory system. Finally, the instruction registers and instruction RAM provide an interface to manage the instruction stream.

7.2.1 Computational Registers

The computational registers are data registers used as the floating point units' instruction operands. Each floating point unit has its own *local* register file with 64 32-bit registers. A floating point unit's computational registers are accessible by its corresponding floating point unit and the external interface, but not by the other floating point units. Additionally, there is a *global* register file with 128 32-bit registers that are accessible to all floating point units.

The floating point units access the computational registers through the geometry coprocessor's instructions as described in section 7.3. The local registers may be used as source or destination operands; the global registers may be used only as source operands. The external interface accesses the registers through a memory mapped region of the CPU's address space. The physical address of a computational register is constructed as follows:

31 16	15	14	13	12 10	9 2	1 0
GEOMBASE	0	0	G	FPU	REG	0

where

GEOMBASE offsets to the base address of the geometry coprocessor;

G selects between the global register file (when G is set) and the local register files (when G is clear);

FPU identifies the floating point unit being addressed (0 to 3). FPU is ignored when G is set. Note that FPU contains an extra bit for future expansion; and

REG identifies the particular computational register being addressed (0 to 127). Note that REG contains an extra bit for future expansion.

The external interface to the computational registers supports both word (32-bit) and double-word (64-bit) transactions. Word transactions reference the specified register of the specified local or global register file. Double-word transactions must be aligned on double-word boundaries and reference adjacent registers of the specified local or global register file.

The geometry coprocessor interface supports four data formats within the computational registers: IEEE single-precision floating point, two’s complement integer, packed XY, and packed RGB. The IEEE single-precision floating point format supports IEEE positive and negative normalized numbers, positive and negative zero, and positive and negative infinity; it does not support NANs and denormalized numbers. The two’s complement integer format supports sign-extended numbers within the range -2^{24} to 2^{24}-1. These data formats are defined below.

Single-precision floating point format:

31	30 23	22 0
S	EXP	FRACTION

where

S is the sign bit;

EXP is the exponent portion of the floating point number; and

FRACTION is the fractional portion of the floating point number.

Two’s complement integer format:

31	30 0
S	INTEGER

where

S is the sign bit; and

INTEGER is the two’s complement number.

Packed XY (for coordinate data):

31 16	15 0
Y Coordinate	X Coordinate

Packed RGB (for color data):

31 24	23 16	15 8	7 0
Undefined	Red	Green	Blue

7.2.2 Implementation and Revision Register

The implementation and revision register is read-only and may be used by diagnostic or system software to identify the implementation and revision of the implementation. The physical address of the implementation and revision register is constructed as follows:

31 16	15	14	13 2	1 0
GEOMBASE	0	1	0	0

where

GEOMBASE offsets to the base address of the geometry coprocessor.

The format of the implementation and revision register is as follows:

31 24	23 16	15 8	7 4	3 0
Undefined	COUNT	IMPL	MAJ	MIN

where

COUNT is the maximum allowable value for the COUNT field in the DMA operation register;
IMPL is the implementation;
MAJ is the major revision number; and
MIN is the minor revision number.

7.2.3 Compare Register

The compare register is a 64-bit (two adjacent 32-bit) readable and writable register that acts as an intermediate storage location for results of geometry coprocessor compare instructions. The geometry coprocessor's sequencer or the CPU reads this register sometime after a compare instruction to determine the result of the instruction and branch accordingly. The register is writable so system software can restore its state during a context switch. The physical address of the compare register is constructed as follows:

31 16	15	14	13 2	1 0
GEOMBASE	0	1	2 and 3	0

where

GEOMBASE offsets to the base address of the geometry coprocessor.

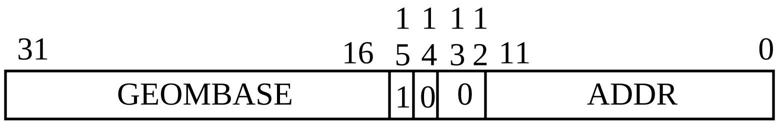
The compare register comprises four fields, one corresponding to each FPU, as shown below. Each of the four fields contains sixteen bits, exposed through the instruction set as the destinations of the compare instructions. Bit 63 has special meaning, allowing the geometry coprocessor to interrupt the CPU through an implementation-specific mechanism. See section 7.3.14 through 7.3.18 for more detail on the use of the compare register.

63 48	47 32	31 16	15 0
FPU3	FPU2	FPU1	FPU0

7.2.4 The Instruction RAM and the Instruction Registers

The *instruction RAM* is a 4,096 byte RAM containing 1,024 32-bit instructions. The instruction registers comprise the *program counter register*, the *status register*, and the *halt register*. The instruction RAM and program counter are readable and writable, the status register is read-only, and the halt register is write-only. The instruction registers provide the control necessary to execute and manage the instructions stored in the instruction RAM.

The instruction RAM stores geometry coprocessor instructions, which are described in detail in section 7.3. As is true of the computational registers, the instruction RAM may be addressed through the external interface as either 32-bit words (one instruction per word) or 64-bit double-words (two instructions per double-word). The instruction RAM should not be accessed through the external interface if the status register indicates that the coprocessor is BUSY. The physical address of the instruction RAM is constructed as follows:



where

GEOMBASE offsets to the base address of the geometry coprocessor; and

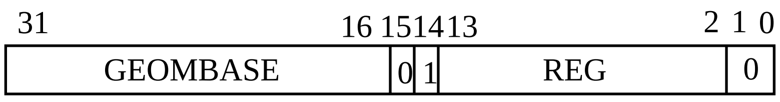
ADDR is 12-bits, addressing the full 4,096 bytes of instruction RAM.

Writing to the program counter register loads the program counter and, as a side effect, immediately begins executing instructions from that location in the instruction RAM. The program counter must be between 0 and 1023; results are undefined otherwise.

The status register is zero if the geometry coprocessor is IDLE and therefore available, and non-zero if the geometry coprocessor is BUSY and therefore unavailable. The status register is set upon writing to the program counter register, and cleared upon execution of a STOP instruction.

Writing to the halt register causes the geometry coprocessor to halt execution. If a 1 is written to the halt register, instruction issuance and execution continues until all hardware state is readable, at which time the status register reflects that the device is IDLE and the program counter register contains the address of the last executed instruction. If a 2 is written, instruction execution halts immediately and the hardware is reset.

The physical addresses of an instruction register is constructed as follows:



where

GEOMBASE offsets to the base address of the geometry coprocessor; and

REG is 4 to identify the program counter, 5 to identify the status register, and 6 to identify the halt register.

7.2.5 DMA registers

The DMA registers, the *DMA address high, DMA address low,* and *DMA operation* registers, allow data to be transferred directly between the geometry coprocessor and main memory. The DMA address registers store the main memory address to or from which the data is transferred. The address is up to 64 bits as defined by the concatenation of the DMA address high and low registers. The DMA operation register stores the remaining information necessary to perform the DMA transaction, including the geometry coprocessor address (GCP_ADDRESS), the number of 32-bit words to be transferred (COUNT), and the direction of the transfer (OP). The format of the DMA operation register is shown below:

31 16	15 8	7 0
GCP_ADDRESS	COUNT	OP

If OP is zero, the geometry coprocessor transfers COUNT words from a contiguous address range in system memory to a contiguous address range in the geometry coprocessor. If OP is one, the geometry coprocessor transfers COUNT words from the geometry coprocessor to system memory. System memory is partitioned, as described in implementation specifications, between main memory and the PixelVision *scratch memory system* described in section 8.5. The main memory address range begins at the address specified by the DMA address register; the geometry coprocessor address range begins at an offset from the geometry coprocessor's base address as follows:

31 16	15 0
GEOMBASE	GCP_ADDRESS

where

GEOMBASE offsets to the base address of the geometry coprocessor; and

GCP_ADDRESS comes from the DMA operation register.

The geometry coprocessor initiates the DMA transaction immediately after the CPU writes to the DMA operation register. The DMA transaction will complete before any subsequent IO transactions to the geometry coprocessor succeed. An implementation may restrict the maximum value for COUNT, and must indicate this restriction in the geometry coprocessor's implementation and revision register. After a DMA transaction, the DMA address registers will point to the 32-bit word in system memory immediately following the last word of the transaction.

The physical address of a DMA register is constructed as follows:

31 16	15	14	13 2	1 0
GEOMBASE	1	1	REG	0

where

GEOMBASE offsets to the base address of the geometry coprocessor; and

REG is 0 to identify the DMA address low register, 1 to identify the DMA address high register, and 2 to identify the DMA operation register.

7.3 Instruction Set

The PixelVision geometry coprocessor implements a SIMD single-precision floating point instruction set, enumerated in table 7-1. The instruction set contains the traditional floating point computational instructions: *add*, *subtract*, *multiply*, and *reciprocal*. Additionally, *multiply / accumulate and reciprocal square root* are included as they are particularly useful for graphics applications. There are two instructions to convert between floating point format and integer format: *truncate* and *convert*. There are two instructions to pack 16-bit integer data and 8-bit integer data into 32-bit words: *pack xy* and *pack rgb*. Similarly, there are two instructions to unpack these data types: *unpack xy* and *unpack rgb*. There are three compare instructions: *compare greater than, compare less than or equal,* and *compare equal*. There are two conditional instructions: *conditional branch* and *move*. The *move to, move from,* and *sync* instructions move data between registers on the chip. The *select* instruction indicates which FPUs should execute subsequent instructions. Finally, there is a *stop* instruction to halt processing.

The instruction formats are defined in the instruction synopses below. Source operands may access either the local register files or the global register file. Source operand values 0 to 63 reference local registers, while values 128 to 255 reference global registers. Destination operands should not reference the global register file. Note that to allow for future expansion, some fields within the instruction formats are larger than required.

Table 7-1 Geometry Coprocessor Instruction Set

OPCODE	Mnemonic	Operation
0	ADD	Add
1	SUB	Subtract
2	MUL	Multiply
3	REC	Reciprocal
4	MACC	Multiply / Accumulate
5	MACCS	Multiply / Accumulate Start
6	RECSQRT	Reciprocal Square root
7	TRUNC	Convert from floating point to integer format
8	CVT	Convert from integer to floating point format
9	PKXY	Pack two 16-bit integer values into a 32-bit word
10	PKRGB	Pack three 8-bit integer values into a 32-bit word
19	UNPKXY	Unpack a 32-bit word into two 16-bit integer values
22	UNPKRGB	Unpack a 32-bit word into three 8-bit integer values
13	CMPGT	Greater Than Compare
14	CMPLE	Less Than Or Equal Compare
15	CMPEQ	Equal Compare
16	CBRANCH	Conditionally replace the program counter
17	CMOVE	Conditionally move the contents of a register
18	MOVE_TO	Move data to the global register file
11	MOVE_FROM	Move data from the global register file
20	SYNC	Wait for data to be moved by MOVE_TO or MOVE_FROM
21	SELECT	Select which FPUs should execute subsequent instructions
12	STOP	Halt instruction sequencing
23	NOOP	No Operation

7.3.1 Add -- ADD

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	ADD

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically added. The result is truncated toward zero and placed in register DST.

7.3.2 Subtract -- SUB

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	SUB

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically subtracted (SRC0 - SRC1). The result truncated toward zero and placed in register DST.

7.3.3 Multiply -- MUL

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	MUL

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically multiplied. The result is truncated toward zero and placed in register DST.

7.3.4 Reciprocal -- REC

Format:

31 24	23 16	15 8	7 0
SRC0	ZERO	DST	REC

Description:

For each selected floating point unit, the reciprocal of the contents of register SRC0 is taken. The result is truncated toward zero and placed in register DST.

7.3.5 Multiply Accumulate -- MACC

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	MACC

Description:

For each floating point unit, the contents of registers SRC0 and SRC1 are arithmetically multiplied. The product is then added to the result of the MACC or MACCS instruction issued four cycles earlier. For each selected floating point unit, the result is truncated toward zero and placed in register DST. A MACC instruction may not be immediately preceded by an instruction that stalls the pipeline (see section 7.4). The instruction four cycles before a MACC instruction must be either a MACC or MACCS. The three instructions following a MACC are restricted to NOOP, STOP, CBRANCH, SELECT, MACCS, or MACC.

7.3.6 Multiply Accumulate Start -- MACCS

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	ZERO	MACCS

Description:

For each floating point unit, the contents of registers SRC0 and SRC1 are arithmetically multiplied. The result is stored within the floating point unit's pipeline for use by a subsequent MACC instruction. A MACCS instruction may not be immediately preceded by an instruction that stalls the pipeline (see section 7.4). The instruction four cycles after a MACCS instruction must be a MACC. The three instructions following a MACCS are restricted to NOOP, STOP, CBRANCH, SELECT, MACCS, or MACC.

7.3.7 Reciprocal Square Root -- RECSQRT

Format:

31 24	23 16	15 8	7 0
SRC0	ZERO	DST	RECSQRT

Description:

For each selected floating point unit, the positive arithmetic reciprocal square root of the contents of register SRC0 is taken. The result is truncated toward zero and placed in register DST.

7.3.8 Truncate -- TRUNC

Format:

31 24	23 16	15 8	7 0
SRC0	ZERO	DST	TRUNC

Description:

For each selected floating point unit, the floating point value in register SRC0 is arithmetically converted to integer format. The result is truncated toward zero and placed in register DST.

.

7.3.9 Convert -- CVT

Format:

31 24	23 16	15 8	7 0
SRC0	ZERO	DST	CVT

Description:

For each selected floating point unit, the integer value in register SRC0 is arithmetically converted to floating point format. The result is truncated toward zero and placed in register DST.

7.3.10 Pack Two 16-bit Integers Into a 32-bit Word -- PKXY

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	PKXY

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are merged into register DST such that DST<31:16> contains SRC0<15:0> and DST<15:0> contains SRC1<15:0>.

7.3.11 Pack Three 8-bit Integers Into a 32-bit Word -- PKRGB

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	PKRGB

Description:

For each selected floating point unit, the contents of registers SRC0, SRC1, and DST are merged into register DST such that DST<23:16> contains SRC0<7:0>, DST<15:8> contains SRC1<7:0>, and DST<7:0> remains unchanged. DST<31:24> will contain zeros.

7.3.12 Unpack a 32-bit Word into Two 16-bit Integers -- UNPKXY

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	UNPKXY

Description:

For each selected floating point unit, the contents of local register DST are un-packed into registers SRC0 and SRC1 such that SRC0<15:0> contains DST<31:16> and SRC1<15:0> contains DST<15:0>. SRC0 and SRC1 will be sign extended.

7.3.13 Unpack a 32-bit Word into Three 8-bit Integers -- UNPKRGB

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	DST	UNPKRGB

Description:

For each selected floating point unit, the contents of local register DST are unpacked into registers SRC0, SRC1, and DST such that SRC0<7:0> contains DST<23:16>, SRC1<7:0> contains DST<15:8>, and DST<7:0> remains unchanged. SRC0<31:8>, SRC1<31:8>, and DST<31:8> will contain zeros.

7.3.14 Greater Than Compare -- CMPGT

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	CMP	CMPGT

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically compared. If SRC0 is greater than SRC1, then the corresponding bit in the compare register is set; otherwise it is cleared. The affected bit in the compare register is bit (FPU x 16 + CMP), where FPU is the corresponding floating point unit ID and CMP is a value between 0 and 15.

7.3.15 Less Than Or Equal Compare -- CMPLE

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	CMP	CMPLE

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically compared. If SRC0 is less than or equal to SRC1, then the corresponding bit in the compare register is set; otherwise it is cleared. The affected bit in the compare register is bit (FPU x 16 + CMP), where FPU is the corresponding floating point unit ID and CMP is a value between 0 and 15.

7.3.16 Equal Compare -- CMPEQ

Format:

31 24	23 16	15 8	7 0
SRC0	SRC1	CMP	CMPEQ

Description:

For each selected floating point unit, the contents of registers SRC0 and SRC1 are arithmetically compared. If SRC0 is equal to SRC1, then the corresponding bit in the compare register is set; otherwise it is cleared. The affected bit in the compare register is bit (FPU x 16 + CMP), where FPU is the corresponding floating point unit ID and CMP is a value between 0 and 15.

7.3.17 CBRANCH

Format:

31 24	23 8	7 0
CMP	TARGET	CBRANCH

Description:

The program counter is conditionally replaced with the value TARGET and execution continues with the new program counter. The branch is taken if CMP[7:4] is zero and any of the compare register bits, compare[FPU x 16 + CMP[3:0]] for the selected FPUs, is set. The branch is also taken if one of the following conditions is satisfied: CMP[7] is set and any of the compare register bits, compare[FPU x 16 + i] for the selected FPUs and for values of i from 0 to 7, is set; CMP[6] is set and the bitwise logical AND of the low 8 compare register bits from the selected FPUs is non-zero; CMP[5] is set and the PACKET_STATUS bit from the rendering subsystem's status register is set; or CMP[4] is set and there was a DMA error on the most recently executed DMA transaction. The instruction following CBRANCH is always executed.

7.3.18 CMOVE

Format:

31 24	23 16	15 8	7 0
CMP	SRC	DST	CMOVE

Description:

For each floating point unit, the contents of register SRC is conditionally moved to register DST. The move occurs if the condition outlined in the description of the CBRANCH instruction is met.

7.3.19 MOVE_TO

Format:

31 24	23 8	7 0
DST	GCP_ADDRESS	MOVE_TO

Description:

The MOVE_TO instruction moves a double-word of data to global registers DST and (DST+1) from the two 32-bit registers at location:

31 16	15 0
GEOMBASE	GCP_ADDRESS

DST and GCP_ADDRESS must be double-word aligned.

7.3.20 MOVE_FROM

Format:

31 24	23 8	7 0
SRC	GCP_ADDRESS	MOVE_FR

Description:

The MOVE_FROM instruction moves a double-word of data from global registers SRC and (SRC+1) to the two 32-bit registers at location:

31 16	15 0
GEOMBASE	GCP_ADDRESS

SRC and GCP_ADDRESS must be double-word aligned. Moving data to the DMA operation register has the side effect of initiating a DMA transfer.

7.3.21 SYNC

Format:

31 8	7 0
ZERO	SYNC

Description:

The SYNC instruction stalls the instruction stream until all the MOVE_TO and MOVE_FROM instructions have completed. MOVE_TO and MOVE_FROM instructions complete when all data, including DMA data when appropriate, has been moved.

7.3.22 SELECT

Format:

31 16	15 8	7 0
ZERO	MASK	SELECT

Description:

The SELECT instruction sets a register, called the *select register*, to indicate which FPUs should execute subsequent instructions. Instructions are executed by those floating point units whose corresponding select bit is set, and ignored by those floating point units whose corresponding select bit is clear. MASK[0] corresponds to FPU0, MASK[1] to FPU1, MASK[2] to FPU2, and MASK[3] to FPU3.

In addition to being exposed through the instruction stream, the select register is exposed as a read-only register through the external interface. The physical address of the select register is constructed as follows:

31 16	15	14	13 2	1 0
GEOMBASE	0	1	7	0

where

GEOMBASE offsets to the base address of the geometry coprocessor.

7.3.23 STOP

Format:

31 8	7 0
ZERO	STOP

Description:

Program execution halts and the geometry coprocessor's status is set to IDLE. The instruction following STOP is always executed.

7.4 Instruction Sequencing and Timing

Since certain instructions require a pipeline resource for multiple cycles, the geometry coprocessor can not always retire an instruction every cycle. For these *complex instructions*, a hardware stall mechanism stalls the instruction RAM from issuing subsequent instructions until the internal pipeline can accept them. Table 7-2 shows how each instruction affects the stall mechanism for the first implementation of the geometry coprocessor.

The latency values in table 7-2 indicate the number of cycles until the instruction's destination has been updated. The instruction stream must wait (*latency* minus *stalled cycles)* before referencing the instruction's result. For example, a latency of 1 means that the instruction stream may reference the result in the next instruction, while a latency of 6 means that the instruction stream must wait six instructions before referencing the result. Software should not reference an outstanding destination register before waiting the full latency.

Table 7-2 Geometry Coprocessor Instruction Set Timings

OPCODE	Stalled Cycles	Latency
ADD	0	6
SUB	0	6
MUL	0	6
REC	10	17
MACC	0	9
MACCS	0	NA
RECSQRT	16	22
TRUNC	0	6
CVT	0	6
PKXY	0	6
PKRGB	1	7
UNPKXY	1	7
UNPKRGB	2	9
CMPGT	0	6
CMPLE	0	6
CMPEQ	0	6
CBRANCH	0	2 (until executing at new program counter)
CMOVE	0	6
MOVE_TO	Use sync instruction to synchronize	
MOVE_FROM	Use sync instruction to synchronize	
SYNC	indeterminate	NA
SELECT	0	1 (until the mask register is modified)
STOP	indeterminate[1]	NA
NOOP	0	NA

[1] The STOP instruction modifies the status register when the internal pipeline flushes. In the worst case, if the instruction stream is in the middle of a string of MACC instructions, the status register is modified after the last MACC instruction completes. Otherwise, the status register is modified after the current instruction completes.

8 Rendering Subsystem Interface

This chapter describes the programmer's interface to the PixelVision rendering subsystem. The interface is designed specifically to be efficient in software environments where multiple processes independently communicate with the rendering subsystem. The overhead introduced by the support for multiprocess graphics software environments is minimal, thereby making single-threaded implementations attractive as well.

The descriptions in this document are architectural. Refer to implementation specifications for implementation details, including a list of implementation-specific registers such as video timing registers and color lookup table registers.

8.1 RISC Graphics

The primary goal of the PixelVision rendering interface is that it remain as simple as possible while exposing the hardware units employed. Engineering productivity is severely hampered by complex interfaces. If the interface model or the interaction between state and commands is difficult to understand, then the interface will be difficult to program. The PixelVision designers have made a conscious decision to make the interface as lightweight as possible, while still meeting the needs of 2D and 3D graphics libraries, as well as offering the appropriate support for multiprocess graphics software.

As the interface is simplified, it becomes apparent to the hardware designers what commands are important to optimize, and higher performance should be realized on those commands deemed important. This philosophy is much like that of RISC designers, limiting the instruction set to a handful of important operations so that engineering effort can be concentrated in those areas. Part of the PixelVision philosophy is that features must be well justified before being moved from the CPU environment to the special-purpose graphics hardware environment. Greatly increased parallelism or pipelining for frequently executed operations are good justifications for special-purpose hardware.

8.2 Multiprocess Graphics

Another goal of the PixelVision rendering interface is to effectively deal with the implications of multiprocess graphics. Traditional graphics systems in which all graphics requests filter through a central server process, such as the X server, are limiting in several respects. First, the server model introduces unwanted overhead into the graphics pipeline, as applications must communicate through interprocess communication protocols. Secondly, as the supported graphics functionality grows, so does the server, until it reaches an unmanageable size. This does not imply that the X server is an unreasonable model, only that it should not be abused by forcing it to do all things for all applications. Interfaces to the graphics components that promote multiple threads of control will help to overcome these problems.

There are three major obstacles for multiprocess graphics software implementations: sharing the graphics memory system between processes, synchronizing and sharing the geometry coprocessor between processes, and synchronizing and sharing the rendering subsystem between processes.

The graphics memory system is shared through the virtual graphics memory capabilities of the rendering processor, as described in chapter 3. The graphics virtual memory system provides each process its own virtual view of graphics memory that is independent of other processes.

The geometry coprocessor's synchronization and sharing mechanisms are essentially extensions to the CPU's mechanisms. That is, the operating system swaps context in and out of the geometry coprocessor whenever a context switch occurs, and transfers *ownership* of the geometry coprocessor to the process executing on the CPU.

Synchronization and sharing of the rendering subsystem are handled by a communication protocol, called the command packet protocol, described in this chapter. The problems of synchronization and sharing get more difficult in a multiprocessor environment, where processors attempt to access the rendering subsystem simultaneously. It should be noted that the PixelVision architecture applies equally well to multiprocessor workstations.

8.3 Synchronization of PixelVision Rendering Resources

Both the rendering subsystem and its command packets are shared resources, and therefore have synchronization issues. While the rendering subsystem is shared among several processes in the system, the command packets are shared between a single process and the rendering subsystem. The process writes data to the packet and the rendering subsystem reads data from it.

The system bus synchronizes access to the rendering subsystem from multiple processes. Since the size of a CPU's communication with the rendering subsystem to initiate a command packet transfer is a single bus transaction (described in detail later), the natural serialization process of the system bus synchronizes requests from different processes. The rendering subsystem's role in synchronization is to guarantee that it processes all data from one packet before processing any data from the next packet. This is trivial and is a side-effect of the interface.

In addition to synchronizing access to the PixelVision rendering subsystem, access to the command packets must be synchronized to control contention between the CPU and the rendering subsystem. The CPU must be able to detect when the rendering subsystem has completed reading a packet so it knows when it may start writing to that packet. This synchronization occurs by using more than a single command packet; while the rendering subsystem reads data from one of the packets, the CPU can write to another. PixelVision has a programmable limit, with an implementation-specific maximum value, on the number of packets to which it may have simultaneous pending access.

8.3.1 *A Stateless Interface*

Traditional rendering processors maintain *graphics state* in internal registers where programmers can store infrequently loaded information. This state is the subset of the application's graphics context information that the rendering processor supports directly. The results of commands on a device with complex state may be highly dependent on that state. This can easily lead to hours of debugging a piece of software in search of a bug that was caused by a completely different piece of code.

The simplest device to program is arguably a memory mapped frame buffer. A memory mapped frame buffer is a *stateless* device; all graphics context resides in the CPU environment and is managed entirely by software. Unfortunately, CPUs and system busses are not yet at the point where a memory mapped 24-plane frame buffer will perform adequately for serious 3D applications.

It is a strict goal of the PixelVision rendering interface that programmers need have little understanding of the internals of the hardware implementation to use the interface. The PixelVision rendering interface is a compromise between the simplicity of the memory mapped frame buffer and the power of a more complex interface. All graphics state information is localized to a command packet, where it should be viewed as parameters to a subroutine instead of state behind an interface.

8.4 Rendering Subsystem Implementation Registers

The PixelVision rendering subsystem contains several registers that pertain to the global state of the rendering subsystem: the implementation and revision register, the configuration register, and a list of implementation-specific registers, such as memory timing and VDAC registers. The PixelVision architecture does not define these implementation-specific registers, but does reserve an address range for them. Refer to implementation specifications for more detail.

8.4.1 Implementation Register Address Range

The PixelVision rendering subsystem reserves an address range for implementation-specific registers such as video timing registers. The registers reside at an offset from an implementation-specific base address as follows:

31 21	20 18	17 2	1 0
RENDERBASE	0,5	REG	0

where

RENDERBASE offsets to the base address of the rendering subsystem; and

REG is the address of an implementation register.

8.4.2 Implementation and Revision Register

The implementation and revision register is read-only and may be used by diagnostic or system software to identify the implementation and revision of the implementation. The physical address of the implementation and revision register is constructed as follows:

31 21	20 18	17 2	1 0
RENDERBASE	2	1	0

where

RENDERBASE offsets to the base address of the rendering subsystem.

The format of the implementation and revision register is as follows:

31 29	28 24	23 20	19 16	15 8	7 4	3 0
RSVD	MSZ	VSZ	USZ	IMPL	MAJ	MIN

where

MSZ is the size of the physical memory system (measured in MPixels);
VSZ is the size of the visible pixelMap -- 0 is 1024x1024; 1 is 1280x1024;
2 is 2048x1280; 3 is 2048x1536; 4 is 2048x2048; and 5 is *unsupported* (the implementation does not support a visible pixelMap);
USZ is the size of the update array -- 0 is 4x2; 1 is 4x4; 2 is 8x2; and 3 is 8x4;
IMPL is the implementation; and
MAJ and MIN are the major and minor revision numbers.

8.4.3 Configuration Register

The configuration register sets the global state of the rendering subsystem. The configuration register contains seven significant fields, one to set the maximum number of packets to which the rendering subsystem can have simultaneous access, one to enable/disable *stereo mode*, and five to enable/disable the five interrupt conditions described in the status register description in section 8.6.1. The physical address of the configuration register is constructed as follows:

31 21	20 18	17 2	1 0
RENDERBASE	2	2	0

where

RENDERBASE offsets to the base address of the rendering subsystem.

The format of the configuration register is as follows:

31 19	18 15	14	13	12	11	10	9	8	7 0
Reserved	MBF	K	R	M	P	V	S	X	MAX_PKTS

where

MBF must be set to 0xF. This field is reserved for future enhancements.

K, R, M, P, and V are set to enable the interrupt conditions corresponding to the PACKET_INT, RENDER_INT, VM_INT, PAGE_INT, and VERT_INT bits in the status register respectively. Section 8.6.1 describes the status register. V is supported only if the visible pixelMap is supported.

S is set to place the display subsystem in *stereo mode*. When in stereo mode, the display subsystem will display left and right stereo views, alternating with each vertical sync event. Stereo mode applies globally to the entire visible pixelMap, and is supported only if the visible pixelMap is supported.

X is reserved.

MAX_PKTS contains the maximum number of command packets to which the PixelVision rendering subsystem can have simultaneous pending access. After resetting the rendering subsystem, MAX_PKTS will contain its maximum allowable value.

8.5 Command Packet Overview

Rendering commands, such as drawing lines and triangles, are transferred to the rendering subsystem through a command packet stream. An application using the command packet interface writes commands and data to a physically contiguous locked-down memory buffer. This buffer resides either in a special-purpose memory system called the PixelVision *scratch memory system*, or within the low 64 MBytes of main memory. The scratch memory system resides at an offset from an implementation-specific base address as follows:[1]

31 21	20 18	17 2	1 0
RENDERBASE	6,7	INDEX	0

where

> RENDERBASE offsets to the base address of the rendering subsystem; and
> INDEX indexes into the buffer.

When the application has completed writing the packet, it tells the PixelVision rendering subsystem, via a read from the read-only *polling register*, to transfer the command packet. The data read from the polling register indicates whether the rendering subsystem is BUSY, in which case the application should retry at a later time, or FREE, in which case the rendering subsystem will begin processing the command packet.

After initiating the command packet transfer, the rendering processor parses the packet and executes the appropriate operations. The first 32-bit word of a command packet contains a packet header describing the packet's syntax. To terminate a command stream to the PixelVision rendering subsystem, the 32-bit word following the command packet should contain *zero*. If it contains a non-zero value, then the PixelVision rendering subsystem interprets it as the header of the next packet, and continues.

Execution of the commands in the packet is atomic, implying that the rendering processor executes all commands in one packet before executing any commands in the next packet. Furthermore, the execution order is guaranteed to be the order in which the commands appear in the packet.

This section describes the general framework of application software using the command packet interface. The following sections define the specifics of the protocol. Figure 8-1 shows the outline of a procedure that issues a command to draw a line. The global variable, pPacket, is the virtual address of a physically-contiguous locked-down memory buffer serving as a PixelVision command packet buffer. To avoid synchronization problems concerning the command packet memory, multiple command packet buffers are used cyclically. The cycling of command packets is handled by the switchPacket() routine.

The Line procedure first fills in the command packet with header information and vertex data. It then reads the polling register and receives status over the system bus. Table 8-1 enumerates the status values that the PixelVision rendering subsystem can return as a result of reading the polling register.

[1] System software must handle the sharing of the scratch memory system between processes.

Table 8-1   Status Values Returned from Polling Register

Value	Status	Description
0	FREE	Data was accepted
0x00800000	BUSY	Rendering Subsystem is unavailable; retry later

The Line procedure checks the status in the polling register. If the status is FREE, then the PixelVision rendering subsystem immediately initiates the transfer of the command packet. A BUSY status causes a software retry, since the procedure just spins on the status word. Neither the code below nor the procedural interface presented is appropriate for high-performance graphics; they serve only as a trivial example of the framework of the command packet interface.

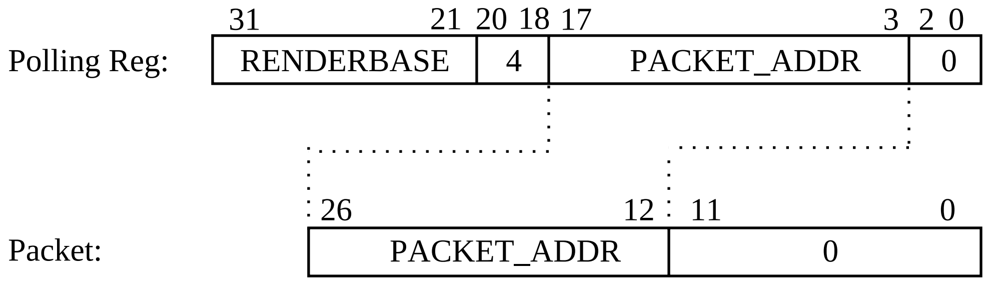
```
/*
 * Write vertex data into the command packet and
 * inform PixelVision that it may transfer the packet.
 */
Line(x1, y1, x2, y2)
        int x1, y1, x2, y2;     /* screen-coordinate endpoints of the line */
{
        fillLinePacket(pPacket, x1, y1, x2, y2);
        while (*polling == BUSY)
                ;
        pPacket = switchPacket();
}
```

Figure 8-1   Line Procedure Using Command Packet Protocol

8.6 Command Packet Protocol

The command packet interface provides atomic access for high-level commands such as drawing lines and quadrilaterals. Access to the PixelVision rendering subsystem is gained through the framework described in the previous section. In order to guarantee an atomic operation containing both the polling register read transaction issued by the application and the command packet transactions issued by the rendering subsystem, the physical address of the command packet is encoded in the address of the polling register read transaction. This encoding provides the packet address to the rendering subsystem so that it can initiate the command packet transfer, while the application receives status as part of the same atomic operation.

The first word of the command packet may reside on any 4096-byte boundary within a 27-bit (128-MByte) physical address space. The address of the command packet is created as follows: bits 26:12 equal bits 17:3 of the polling register read transaction address, and bits 11:0 are zero. If bit 26 of the packet address is set, then the packet resides in the low 64 MBytes of main memory (given by bits 25:0); otherwise, the packet resides in the PixelVision scratch memory system. The following figure depicts the mapping from the address of the polling register read transaction to the base address of the command packet.



where

RENDERBASE offsets to the base address of the rendering subsystem; and

PACKET_ADDR offsets to the base address of the command packet.

Command packet data is partitioned into four categories: the packet header, the rendering program, graphics context, and primitive data. The packet header is the first 32-bit word in the packet, and describes the syntax of the remainder of the packet. The rendering program follows the packet header and describes the semantics of the packet. Graphics context includes object properties such as the XY mask and plane equation data. Primitive data includes coordinate values.

Graphics context is further divided into three categories: per-packet context, per-primitive context, and per-vertex context. As the names imply, per-packet context applies to all primitives in the command packet, per-primitive context changes with each new primitive (e.g. with each new line in the packet), and per-vertex context changes with each vertex in each primitive. When command packets are strung together in the main memory buffer, per-packet graphics context information will persist from one packet to the next, unless it is supplied again.

Figure 8-2 depicts the overall structure of a command packet. Graphics context appears in braces ([]). The figure is not a formal grammar because some of the permutations are illegal. For instance, the figure does not make it clear that the *XYMASK* cannot be included as both per-packet and per-primitive context. The figure acts only as a high-level overview of command packet syntax, although it also provides the correct order for the elements in the command packet. The formal grammar can be extracted from the descriptions in the remainder of this chapter.

```
Packet Header
        HEADER
Rendering Program
        Up to six pixelMap IDs
        Up to fifteen instructions
Per-Packet Data
        [CLIPRECT_MIN, CLIPRECT_MAX per-packet]
        [XYMASK0 per-packet -- 8 32-bit words]
        [ZMASK per-packet]
        [RGB0 per-packet]
        [RGB1 per-packet]
        [Z0 per-packet]
        [Z1 per-packet]
Per-Primitive Data -- repeated for the number of primitives in the packet
        [CLIPRECT_MIN, CLIPRECT_MAX per-primitive]
        [XYMASK per-primitive -- 8 32-bit words]
        [ZMASK per-primitive]
        [RGB0 per-primitive]
        [RGB1 per-primitive]
        [Z0 per-primitive]
        [Z1 per-primitive]
        [RGB0_V0, RGB0_V1, [RGB0_V2] per-vertex]
        [RGB1_V0, RGB1_V1, [RGB1_V2] per-vertex]
        [Z0_V0, Z0_V1, [Z0_V2] per-vertex]
        [Z1_V0, Z1_V1, [Z1_V2] per-vertex]
        Vertex Data
ZERO or Next Packet Header
```

Figure 8-2 Command Packet Syntax Overview

8.6.1 *Status Register*

The rendering subsystem implements a status register, called the *rendering status* register, which describes the status of the rendering subsystem. The physical address of the rendering status register is constructed as follows:

31 – 21	20 – 18	17 – 2	1 – 0
RENDERBASE	2	3	0

where

RENDERBASE offsets to the base address of the rendering subsystem.

The rendering status register contains nine significant bits. The PACKET_STATUS bit is set if the rendering subsystem is unable to transfer the command packet buffer into its internal pipeline. This same bit forms the command packet polling register. The RENDER_STATUS bit is set if the rendering subsystem has command packet data anywhere in its pipeline that has not yet been rendered. The VM_STATUS bit is set if there is a pending DMA pixelMap access, as described in chapter 3. This same bit forms the VMstatus register. The PACKET_INT, RENDER_INT, and VM_INT bits are set in response to a high to low transition on the PACKET_STATUS, RENDER_STATUS, and VM_STATUS bits respectively. The PAGE_INT bit is set if the rendering subsystem is waiting for the CPU to process a graphics memory page fault. The VERT_INT bit is set in response to the video refresh circuitry issuing a vertical sync event. The _INT bits may be used in conjunction with their corresponding bits in the configuration register to determine the cause of an interrupt. The CPU sets the STALL bit to stall the rendering subsystem's packet processing, and clears it to continue processing. DMA pixelMap access may occur while the STALL bit is set. Finally, the FIELD bit is set if the display subsystem is in stereo mode and is either scanning the left view or is in the vertical retrace period before scanning the left view.

The PACKET_STATUS, RENDER_STATUS, VM_STATUS, and FIELD bits are read-only. The STALL, PACKET_INT, RENDER_INT, VM_INT, PAGE_INT, and VERT_INT bits are readable and writable. The rendering subsystem sets the INT bits to initiate an interrupt, and the CPU clears them after servicing the interrupt. Attempts by the CPU to set an INT are ignored. The format of the rendering status register is as follows:

31 – 10	9	8	7	6	5	4	3	2	1	0
Undefined	S	K_i	K_s	R_i	R_s	M_i	M_s	P_i	V_i	F

where

S is the STALL bit;
K_i is the PACKET_INT bit;
K_s is the PACKET_STATUS bit;
R_i is the RENDER_INT bit;
R_s is the RENDER_STATUS bit;
M_i is the VM_INT bit;
M_s is the VM_STATUS bit;
P_i is the PAGE_INT bit;
V_i is the VERT_INT bit; and
F is the FIELD bit.

8.6.2 Packet Header

The packet header describes the syntax of the command packet and is decoded as follows:

HEADER<2:0>	GEOM_FMT	Operation Identification

GEOM_FMT is described in section 8.6.4 and decoded as follows:

Value	Format	Associated Primitive Data
0	POINTS	V1
1	LINES	V1, V2
2	WIDE_LINES	V1, V2, WIDTH
3	QUADS	V1, V2, V3, V4
4	READ_BLOCKS	XYMIN,XYMAX, ADDR,OFFSET, FORMAT
5	WRITE_BLOCKS	XYMIN,XYMAX, ADDR,OFFSET, FORMAT
6	WIDE_LINES_4	V1, V2, WIDTH
7	WIDE_LINES_2	V1, V2, WIDTH

HEADER<3>	CAP_STYLE	Line cap style

CAP_STYLE specifies the cap style for line segments, to specify the inclusion or exclusion of the second endpoint of the segment. CAP_BUTT includes the second endpoint; CAP_NOTLAST excludes the second endpoint. CAP_STYLE is decoded as follows:

Value	Format	Description
0	CAP_BUTT	Include second endpoint on line segment
1	CAP_NOTLAST	Exclude second endpoint on line segment

HEADER<5:4> CLIP_FMT Format of the clipping rectangle

If non-zero, then the command packet includes CLIPRECT_MIN and CLIPRECT_MAX as two 32-bit words of either per-packet or per-primitive context. Section 8.6.5 describes the clipping rectangle in detail.

Value	Format	Description
0	NONE	No clipping rectangle is supplied
1	PERPACKET	Clipping rectangle per packet
2	PERPRIMITIVE	Clipping rectangle per primitive
3		undefined

HEADER<7:6> XYMASK_FMT Format of XY Mask Information

If non-zero, the command packet includes XYMASK as a 32-bit word of either per-packet or per-primitive context. Section 8.6.6 describes the XY mask in detail.

Value	Format	Description
0	NONE	No XY mask is supplied
1	PERPACKET	XY mask per packet
2	PERPRIMITIVE	XY mask per primitive
3		undefined

HEADER<9:8> ZMASK_FMT Format of Z Mask Information

If non-zero, then the command packet includes ZMASK as a 32-bit word of either per-packet or per-primitive context. Section 8.6.7 describes the Z mask in detail.

Value	Format	Description
0	NONE	No Z mask is supplied
1	PERPACKET	Z mask per packet
2	PERPRIMITIVE	Z mask per primitive
3		undefined

HEADER<11:10> RGB0_FMT Format of RGB0 Plane Equations

RGB0_FMT is described in section 8.6.9 and decoded as follows:

Value	Format	Description
0	NONE	No rgb data is supplied
1	PERPACKET	rgb data per packet
2	PERPRIMITIVE	rgb data per primitive
3	PERVERTEX	rgb data per vertex

HEADER<13:12> RGB1_FMT Format of RGB1 Plane Equation

Same as RGB0_FMT.

HEADER<15:14> Z0_FMT Format of Z0 Plane Equation

Same as RGB0_FMT.

HEADER<17:16> Z1_FMT Format of Z1 Plane Equation

Same as RGB0_FMT.

HEADER<21:18> INST_COUNT Number of instructions in the packet

INST_COUNT tells how many instructions are in the rendering program body. The instructions are described in detail in section 8.6.3.

HEADER<23:22> PID_COUNT Number of pixelMap IDs in the packet

PID_COUNT tells how many 32-bit words containing pixelMap IDs are in the rendering program body. Since a pixelMap ID has 16 bits, two of them may be packed into a 32-bit word; therefore, PID_COUNT is actually the number of pixelMap IDs in the packet divided by two. The use of these pixelMap IDs is described in detail in section 8.6.3.

HEADER<31:24> PRIM_COUNT Number of primitives in the packet

PRIM_COUNT tells how many primitives are in the command packet. For example, if the opcode is LINES, then the packet contains PRIM_COUNT lines. PRIM_COUNT must be non-zero.

8.6.3 Rendering Program

The rendering program follows the packet header. The rendering program consists of up to six pixelMap IDs and up to fifteen rendering instructions. The rendering instructions comprise the inner loop of the edge-seeking algorithm, which defines an order for visitation of the update arrays contained in a geometric figure. At each of those update array locations, an action takes place to affect the graphics memory. The program instructions describe this action. Specifically, they describe how to combine source pixel values and plane equations with destination pixel values at each update array location.

The first portion of the rendering program contains up to six pixelMap IDs, as specified by the PID_COUNT field in the packet header. Following the pixelMap IDs is up to fifteen rendering instructions, as specified by the INST_COUNT field of the packet header. Each pixelMap ID has 16 bits; therefore, a single 32-bit word contains two pixelMap IDs. An instruction references a pixelMap ID indirectly through its index in the command packet. Even indices (0, 2, 4) reside in bits 15:0 of the 32-bit words in the packet, while odd indices (1, 3, 5) reside in bits 31:16.

The instructions perform logical and arithmetic operations, as described by their OPCODE fields, on two operands, described by the SRC0 and SRC1 fields. The result of the operation is stored at a destination location described by the DST field. The instruction's source operands may be one of the pixelMaps, one of the primitive's interpolated plane equations, one of the two internal pixel registers (described below), or one of the constant values 0.0 or 1.0. The destination may be one of the pixelMaps, or one of the two pixel registers.

As described in previous chapters, each pixel in the update array has associated processing elements which operate in parallel. The data path in the rendering subsystem that implements the packet instructions contains an ALU and multiplier for each pixel in the update array. It also contains two registers, called *pixel registers*, for each pixel in the update array. Pixel registers are temporary storage elements used to store intermediate results. Whereas other storage elements in the rendering subsystem, such as pixelMaps, store unsigned data, the pixel registers store signed partial results. When data is transferred from a pixel register to a pixelMap, the data is assumed to be positive (i.e. the sign bit is ignored).

The instruction set supports both 8-bit and 24-bit instructions. 8-bit instructions operate on three 8-bit channels in parallel at each pixel in the update array. For example, red, green, and blue values are calculated and operated on in parallel. 24-bit instructions operate on one 24-bit channel at each pixel in the update array. The instruction format is described below.

INST<7:0>   OPCODE   Opcode

The source values (SRC0 and SRC1) are combined as specified by the opcode, and the result is placed in the destination (DST). Operations not in the instruction set, such as (DST := NOT (SRC0 AND SRC1)), (DST := SRC0 * SRC1 + DST), and (DST := MIN(SRC0, SRC1)), can be synthesized using the basic operations provided. OPCODE is described in section 8.6.10 and decoded as follows:

Type	Value	Format	Description
Logical Functions	0	and	DST := SRC0 AND SRC1
	1	or	DST := SRC0 OR SRC1
	2	xor	DST := SRC0 XOR SRC1
	3	not	DST := NOT SRC0
	4	copy	DST := SRC0
Arithmetic Functions	5	Add	DST := SRC0 + SRC1
	6	Sub	DST := SRC0 - SRC1
	7	CmpLT	cc <- SRC0<SRC1; DST = SRC0
	8	CmpLE	cc <- SRC0<=SRC1; DST = SRC0
	9	CmpEQ	cc <- SRC0==SRC1; DST = SRC0
	10	CmpGE	cc <- SRC0>=SRC1; DST = SRC0
	11	CmpGT	cc <- SRC0>SRC1; DST = SRC0
	12	CmpOVF	cc = OVF; DST := SRC0 + SRC1
	13	MulHigh	DST := SRC0 * SRC1 (high 8 bits)
	14	MulLow	DST := SRC0 * SRC1 (low 8 bits)
	15	Add24	DST := SRC0 + SRC1
	16	Sub24	DST := SRC0 - SRC1
	17	CmpLT24	cc <- SRC0<SRC1; DST = SRC0
	18	CmpLE24	cc <- SRC0<=SRC1; DST = SRC0
	19	CmpEQ24	cc <- SRC0==SRC1; DST = SRC0
	20	CmpGE24	cc <- SRC0>=SRC1; DST = SRC0
	21	CmpGT24	cc <- SRC0>SRC1; DST = SRC0
	22-255		Reserved

INST<11:8> DST Destination of Instruction

DST is the destination of the operation. It can be one of the pixel registers or one of the pixelMaps.

Value	Description
0-1	Pixel Register 0-1
2-3	Reserved
4-9	PixelMap IDs supplied in the packet (0-5)
10-15	Reserved

INST<15:12> SRC1 Source 1 of Instruction

SRC1 is one of the source values for the operation. It can be one of the pixel registers, one of the pixelMaps, one of the constant values 0.0 or 1.0, or one of the interpolated plane equation values.

Value	Description
0-1	Pixel Register 0-1
2-3	Reserved
4-9	PixelMap IDs supplied in the packet (0-5)
10	Constant Value 0.0
11	Constant Value 1.0
12	Interpolated RGB0 Plane Equations
13	Interpolated RGB1 Plane Equations
14	Interpolated Z0 Plane Equation
15	Interpolated Z1 Plane Equation

INST<19:16> SRC0 Source 0 of Instruction

SRC0 is one of the source values for the operation. SRC0 is decoded just as SRC1.

INST<21:20> CC_OP Condition code replace method

The condition code register may be replaced directly with the result of the compare instruction, or it may be ANDed or ORed with the result of the compare instruction. CC_OP indicates the operation performed before replacing the condition code. Section 8.6.10 describes CC_OP in detail.

Value	Format	Description
0	REPLACE	Replace directly
1	AND	AND previous condition code with new condition code
2	OR	OR previous condition code with new condition code
3		Reserved

INST<23:22> XYMASK Configuration of XY mask

If XYMASK is nonzero, then the XY mask masks write operations to pixelMaps. The XYMASK does not apply to write operations to internal pixel registers. XYMASK configures the XY mask as either a 16x16 pixel array, a 256x1 pixel array, or a 1x256 pixel array. The XY mask is described in section 8.6.6.

Value	Description
0	Disable XY mask
1	Enable XY mask and configure it as a 16x16 pixel array
2	Enable XY mask and configure it as an 256x1 pixel array
3	Enable XY mask and configure it as an 1x256 pixel array

INST<24> XYMASK_SENSE Sense of the XYMASK

XYMASK_SENSE indicates the polarity of the XYMASK. In other words, it indicates if set bits in the mask allow writing and clear bits prevent writing, or the converse. XYMASK_SENSE is described in section 8.6.6.

Value	Description
0	Set bits are enabled; Clear bits are disabled.
1	Set bits are disabled; Clear bits are enabled.

INST<25>	ZMASK_EN	Enable the Z mask

If ZMASK_EN is set, then the ZMASK masks write operations to pixelMaps. The ZMASK does not apply to write operations to internal pixel registers. The ZMASK is described in section 8.6.7.

INST<26>	CLIP_EN	Enable the clipping rectangle

If CLIP_EN is set, then the clipping rectangles mask write operations to pixelMaps. The clipping rectangle does not apply to write operations to internal pixel registers. Section 8.6.5 described the clipping rectangle in detail.

INST<27>	CC_EN	Enable the condition code bits

If CC_EN is set, then the condition code bits (saved previously by one of the compare instructions) mask write operations to pixelMaps. The condition code bits do not apply to write operations to internal pixel registers.

8.6.4 *PixelVision Packet Primitives: Geometry Format*

The GEOM_FMT field of the packet HEADER defines the rendering primitives that are supported by the PixelVision rendering subsystem's packet interface. For each packet, PRIM_COUNT primitives are executed, as given by the value of the PRIM_COUNT field of the packet header. All primitives in a packet use the same geometry format. Each primitive includes geometry data as shown in table 8-2.

Each vertex (V1 through V4 and XYMIN and XYMAX) is an (*x,y*) coordinate pair packed into a 32-bit word, where the *y* coordinate is in bits 31:16, and the *x* coordinate is in bits 15:0. The *x* and *y* values are two's complement fixed point numbers in the subpixel coordinate system with 16 significant bits. See the chapter on tiling geometric primitives to understand PixelVision's geometric models.

Table 8-2 Command Packet Primitives

GEOM_FMT	Primitive	Vertex Data
0	POINTS	V1
1	LINES	V1, V2
2	WIDE_LINES	V1, V2, WIDTH
3	QUADS	V1, V2, V3, V4
4	READ_BLOCKS	XYMIN,XYMAX, ADDR,OFFSET, FORMAT
5	WRITE_BLOCKS	XYMIN,XYMAX, ADDR,OFFSET, FORMAT
6	WIDE_LINES_4	V1, V2, WIDTH
7	WIDE_LINES_2	V1, V2, WIDTH

POINTS **V1**

Purpose:

Draw a list of one-pixel points.

Description:

The POINTS opcode causes PixelVision to draw a list of one-pixel points. Each point is parametrized by its coordinate location, V1. When using per-vertex plane equations, the packet contains only one RGB or Z value per point, thereby giving the same effect as per-primitive plane equations.

LINES **V1, V2**

Purpose:

Draw a list of one-pixel-wide line segments.

Description:

The LINES opcode causes PixelVision to draw a list of one-pixel-wide line segments. Each line segment is parametrized by its two endpoints, V1 and V2. When using per-vertex plane equations, the packet contains two RGB or Z values per line segment. The RGB or Z values vary linearly between V1 and V2.

WIDE_LINES **V1, V2, WIDTH**
WIDE_LINES_2 **V1, V2, WIDTH**
WIDE_LINES_4 **V1, V2, WIDTH**

Purpose:

Draw a list of line segments with user-definable width.

Description:

The WIDE_LINES opcode causes PixelVision to draw a list of line segments with user-definable width. Each line segment is parametrized by its two endpoints V1 and V2, and width WIDTH. WIDTH is an unsigned value with 15 significant bits. Each line segment is WIDTHx2 subpixel units wide. The packet contains two RGB values for each RGB plane equation that is included as per-vertex context. The two RGB values vary linearly between V1 and V2. The packet contains three Z values for each Z plane equation that is included as per-vertex context. The three Z values correspond to V1, V2, and V2 offset by half of the line's width (WIDTH subpixel units) along the line segment's minor axis. The WIDE_LINES_2 and WIDE_LINES_4 opcodes are identical to the WIDE_LINES opcode, with the exception that WIDTH must be 2 and 4 pixels (values of 8 and 16) respectively. WIDE_LINES_2 and WIDE_LINES_4 may be faster than the equivalent WIDE_LINES operations.

QUADS **V1, V2, V3,V4**

Purpose:

Draw a list of convex quadrilaterals.

Description:

The QUADS opcode causes PixelVision to draw a list of filled convex quadrilaterals. The first quadrilateral is parametrized by the four vertices V1, V2, V3, and V4. The second quadrilateral is parametrized by the next four vertices, and so on. When using per-vertex plane equations, the packet contains three RGB or Z values, which vary linearly across the quadrilateral. V1 corresponds to the first RGB or Z value; V2 corresponds to the second RGB or Z value; and V3 corresponds to the third RGB or Z value. The quadrilaterals specified must be convex and the vertices must be given in clockwise order. If the vertices are given in counter-clockwise order, nothing will be drawn.

READ_BLOCKS XYMIN, XYMAX, ADDR, OFFSET, FORMAT

Purpose:

Transfer pixel data from graphics memory to main memory.

Description:

The READ_BLOCKS opcode causes the PixelVision rendering subsystem to transfer blocks of pixel data from graphics memory to main memory. The parameters, XYMIN, XYMAX, OFFSET, and FORMAT, are syntactically and semantically identical to the VMxyMin, VMxyMax, VMoffset, and VMformat registers described in chapter 3. ADDR is a 64-bit value corresponding to the concatenation of the VMaddrHigh and VMaddrLow registers. Instead of operating on the data as specified by the rendering program, the READ_BLOCKS operation just *copies* the data from the pixelMap identified by the first pixelMap ID in the packet into main memory.

Software must ensure that the transfer of data does not violate any DMA restrictions of the system. The READ_BLOCKS operation will transfer a maximum of 32 pixels in one DMA transaction across the system bus, and a DMA transaction will not cross a 32-pixel boundary in graphics memory.

WRITE_BLOCKS XYMIN, XYMAX, ADDR, OFFSET, FORMAT

Purpose:

Transfer pixel data from main memory to graphics memory.

Description:

The WRITE_BLOCKS opcode causes the PixelVision rendering subsystem to transfer blocks of pixel data from main memory to graphics memory. The parameters, XYMIN, XYMAX, OFFSET, and FORMAT, are syntactically and semantically identical to the VMxyMin, VMxyMax, VMoffset, and VMformat registers described in chapter 3. ADDR is a 64-bit value corresponding to the concatenation of the VMaddrHigh and VMaddrLow registers. Instead of operating on the data as specified by the rendering program, the WRITE_BLOCKS operation just *copies* the data from main memory directly into the pixelMap identified by the first pixelMap ID in the packet.

Software must ensure that the transfer of data does not violate any DMA restrictions of the system. The WRITE_BLOCKS operation will transfer a maximum of 32 pixels in one DMA transaction across the system bus, and a DMA transaction will not cross a 32-pixel boundary in graphics memory.

8.6.5 *Clipping Rectangle*

The command packet includes a clipping rectangle as specified by the CLIP_FMT field of the packet header. The clipping rectangle restricts pixelMap write access to the pixels within the extents of the clipping rectangle. A clipping rectangle is parametrized by its upper left corner, CLIPRECT_MIN, and its lower right corner, CLIPRECT_MAX. Each of CLIPRECT_MIN and CLIPRECT_MAX is an (x,y) coordinate pair packed into a 32-bit word, where the y coordinate is in bits 31:16, and the x coordinate is in bits 15:0. The x and y values are two's complement fixed point numbers in the subpixel coordinate system with 16 significant bits. The clipping rectangle applies only to writes to pixelMap memory, not to internal pixel registers, and only when it is enabled by the instruction's CLIP_EN bit.

8.6.6 *XY Mask*

The command packet includes XY mask information as specified by the XYMASK_FMT field of the packet header. The XY mask is a 256-bit mask that masks writing of pixels by gating the write enable lines to graphics memory. Eight 32-bit words in the packet header comprise the XY mask. The first word contains bits 31:0, the second word contains bits 63:32, and so on. The XY mask tiles the affected portion of graphics memory with a repeating 16x16, 256x1, or 1x256 pattern, as specified by the rendering instruction's XYMASK field. The XY mask applies only to writes to pixelMap memory, not to internal pixel registers, and only when it is enabled by the instruction's XYMASK field.

Before determining whether to inhibit or allow writes to graphics memory, the XYMASK is XORed with the XYMASK_SENSE field of the instruction. The resulting XYMASK gates the access to pixelMap memory. Specifically, for a 16x16 pattern, the XY mask inhibits writes to pixel coordinate (x,y) if the (16x(y%[1]16)+(x%16))th bit of the mask is clear; for a 256x1 pattern, the XY mask inhibits writes to pixel coordinate (x,y) if the (x%256)th bit of the mask is clear; and for a 1x256 pattern, the XY mask inhibits writes to pixel coordinate (x,y) if the (y%256)th bit of the mask is clear.

8.6.7 *Z Mask*

The command packet includes Z mask information as specified by the ZMASK_FMT field of the packet header. The Z mask masks individual planes of pixelMap memory during write operations. There is one bit in ZMASK for each plane of graphics memory. A set bit permits writing to the corresponding plane while a clear bit prevents writing. The low-order byte of ZMASK masks the blue channel, with the most significant image plane corresponding to the most significant bit in the byte. The second low-order byte of ZMASK masks the green channel. The third low-order byte masks the red channel. The high-order byte of ZMASK is ignored. The Z mask applies only to writes to pixelMap memory, not to internal pixel registers, and only when it is enabled by the instruction's ZMASK_EN bit.

[1] % denotes the modulus operator.

8.6.8 Plane Equations

The plane equation values in the packet contain the data needed to interpolate object properties across the packet's geometric primitives. There are 8 plane equations supported by the PixelVision rendering subsystem: RED0, GREEN0, BLUE0, RED1, GREEN1, BLUE1, Z0, and Z1. If a rendering instruction references the RGB0 plane equation set, then in the course of rendering a geometric primitive, RED0, GREEN0, and BLUE0 are interpolated simultaneously as three 8-bit plane equations. The RGB1 plane equation set has analogous semantics regarding the RED1, GREEN1, and BLUE1 plane equations. If a rendering instruction references the Z0 plane equation, then in the course of rendering a geometric primitive, Z0 is interpolated as a single 24-bit plane equation. Z1 has the same properties as Z0.

RGB values in the command packet are packed 32-bit words, where red is in bits 23:16, green is in bits 15:8, blue is in bits 7:0, and bits 31:24 are ignored. Z values in the command packet contain 24 significant bits, stored in the low bits of the word.

8-bit plane equations (including 24-bit plane equations that are truncated to the low 8-bits) are guaranteed to stay within the range 0 to 255, even when interpolating across the entire screen. This is accomplished by inverting all bits in the pixel value when an out-of-range condition occurs. In addition to preventing visible out-of-range errors, this mechanism is also used in drawing anti-aliased lines, as described in chapter 6. 24-bit plane equations should be clamped to a restricted range in software (1 to 2^{24}-2) to prevent overflow and underflow errors.

The command packet may include plane equations as per-packet, per-primitive, or per-vertex graphics context. Per-packet and per-primitive plane equations are used to implement flat shading. Per-vertex plane equations assign a different value to each vertex of a primitive and then interpolate the values across the primitive's surface. Per-vertex plane equations are used to implement smooth shading, depth-buffering, transparency, and other higher-quality rendering techniques.

8.6.9 Opcodes and the Condition Code Register

Instructions operate on two data types: 8-bit quantities and 24-bit quantities. All instructions operate on a total of 24 bits per pixel in parallel. The 8-bit instructions perform three 8-bit operations per pixel in parallel, while the 24-bit instructions perform a single 24-bit operation per pixel. The logical operations operate on both 8-bit and 24-bit operands -- the operand type is implicit in the operand.

24-bit instructions should not reference 8-bit plane equations; results are undefined if they do. 8-bit instructions may reference the Z0 and Z1 plane equations providing their initial values are within the range 0 to 255. When an 8-bit arithmetic instruction references Z0 or Z1, internal circuitry replicates its value three times -- once for each 8-bit channel affected by the instruction.

The semantics of the *logical* instructions, the *add* instructions, and the *subtract* instructions are self-explanatory and do not warrant further discussion.

The compare instructions cause SRC0 to be copied to DST, and the condition code register to be altered as specified by the CC_OP field of the instruction. If CC_OP is REPLACE, the condition code register is set if the condition is TRUE and cleared if the condition is FALSE. If CC_OP is AND or OR, the appropriate logical operation is performed between the previous condition code and the new condition code, and the result is placed in the condition code register.

The condition code register has one bit for each of three 8-bit quantities per pixel. These condition code bits are set independently by the 8-bit compare instructions; 24-bit compare instructions set all three condition code bits for a given pixel to the same value. For example, during a CmpEQ instruction, a bit in the condition code register is set for each byte of each pixel in which SRC0 equals SRC1. The CmpOVF instruction causes a condition code bit to be set for each byte of each pixel in which SRC0+SRC1 causes an overflow.

Finally, the *MulHigh* and *MulLow* instructions operates only on 8-bit data types. Multiplying two 8-bit operands generates a 16-bit result. *MulHigh* multiplies two 8-bit operands, SRC0 and SRC1, and places the high 8 bits of the result in DST; *MulLow* multiplies two 8-bit operands, SRC0 and SRC1, and places the low 8 bits of the result in DST.

9 Numerical Analysis of Algorithms

This document thus far has been architectural, at least as much as was practical, focusing on design goals, mathematical models, and interface issues. There have been hints at the practicality of VLSI implementations, but nothing concrete. The first step towards a physical implementation is an understanding of the size of the large special-purpose data paths required to implement the algorithms.

Such analysis is very important in understanding design tradeoffs between 'bits of precision' and accuracy in evaluation. It is fundamental in the design of well-behaved algorithms to understand if and when numerical problems can occur. The numerical analysis of the algorithms drives the decisions for the number of bits needed in internal structures. This chapter discusses the numerical precision required by the algorithms and quantifies the error in evaluating the half-space and plane equations.

9.1 Numerical Analysis of Half-Space Evaluation

We begin with a review of the half-space equations of chapter 4:

$$\text{sidedness} = (y\text{-}y1)\ dx - (x\text{-}x1)\ dy \qquad (9.1)$$

Through appropriate substitutions, equation (9.1) leads to equation (9.2):

$$\text{sidedness} = \qquad (9.2)$$

$$dx\ offsety - dy\ offsetx + \qquad (9.2a)$$

$$dx\ originy - dy\ originx + \qquad (9.2b)$$

$$- dx\ y1 + dy\ x1 \qquad (9.2c)$$

A discrete number system can represent equation (9.2) exactly. Since there are no divides and all variables are integers, evaluation of the equation will have no numerical errors; therefore, for any given value of originx and originy, the PixelVision rendering subsystem can determine exactly the insidedness of each sample point in an update array. Furthermore, changing the referenced update array in response to *XInc* and *YInc* signals causes accumulation of integer values to the expression, indicating that error is not introduced by the edge-seeking algorithm. Fixed point or floating point DDA tiling algorithms can not boast the same accuracy, introducing many potential visual problems.

An exact edge evaluator is very important, but equally important to a VLSI implementation is the size of the edge evaluator's data path. A half-space evaluator is built in a single data path for simplicity and compactness. Therefore, instead of analyzing the different components of equation (9.2), we can analyze the original equation (9.1). Lets first review a few simple numerical analysis rules:

1. Multiplying an *n*-bit number by an *m*-bit number yields an *(n+m)*-bit number.
2. Adding an *n*-bit number to an *n*-bit number yields an (*n+1*)-bit number.

3. If the operands in (1) and (2) are signed numbers, then n and m refer only to the unsigned parts. The result requires an extra sign bit as well.

The data path must be wide enough to accommodate the largest possible value of equation (9.1). The following sequence of equations summarizes the numerical analysis for the half-space evaluator, using the rules from above and substituting bit counts for variables. Coordinate values are 16-bit signed fixed point values with three bits of subpixel precision; therefore, the original vector components, dx, dy, (x-x1), and (y-y1), are 17-bit signed fixed point values.

$$\begin{aligned}\text{sidedness} &= (y-y1)dx - (x-x1)dy \qquad (9.1)\\ &= (16s \times 16s) - (16s \times 16s)\\ &= 33s\end{aligned}$$

The final tally shows that the half-space evaluator data path must have 34 bits, including the sign bit.

9.2 Numerical Analysis of The Plane Equation

We begin with a review of the plane equations of chapter 4:

$$z = -\frac{a}{c}originx - \frac{a}{c}offsetx - \frac{b}{c}originy - \frac{b}{c}offsety + \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \qquad (9.3)$$

where

$$a = (y2 - y1)(z3 - z2) - (y3 - y2)(z2 - z1)$$

$$b = (z2 - z1)(x3 - x2) - (z3 - z2)(x2 - x1)$$

$$c = (x2 - x1)(y3 - y2) - (x3 - x2)(y2 - y1)$$

We rearrange the terms of the plane equation for discussion:

$$z = -\frac{a}{c}offsetx - \frac{b}{c}offsety \qquad (9.3a)$$

$$- \frac{a}{c}originx - \frac{b}{c}originy \qquad (9.3b)$$

$$+ \frac{a}{c}x1 + \frac{b}{c}y1 + z1 \qquad (9.3c)$$

The plane equation analysis is not as forgiving as the half-space equation analysis because the plane equation contains divisions. This section quantifies the error in the plane equation, derives the total number of bits required to accurately represent the plane equation, and explains why the divisions are a necessary part of the setup procedure.

To better understand the analysis, it is necessary to understand the underlying data representation. From there we can analyze the algorithms and attempt to quantify the errors. Representing the components of the plane equation with integer values is insufficient because errors would accumulate much too quickly. An extended precision data representation is necessary. Since the dynamic range of floating point is not required, a fixed point representation is the obvious choice. As a basis for analysis, we assume that the PixelVision rendering subsystem calculates a/c, *b/c*, and the plane equation constant (9.3b + 9.3c) as fixed point numbers that are accurate to within 2^{-nbits}, where *nbits* is the number of bits to the right of the binary point in the fixed point representation.

9.2.1 Bits to the Right of the Binary Point for Plane Equations

The error in the calculation of 9.3a in the plane equation increases linearly with *offsetx* and *offsety*. Remember that the PixelVision rendering subsystem samples at exact pixel coordinates, or at values for *offsetx* and *offsety* of {0, 8, 16, and 24}. When *offsetx* is 0, there is clearly no error in the calculation of *-(a/c)offsetx*. When *offsetx* is 8, the error is eight times that of *(a/c)*: 8×2^{-nbits}. When *offsetx* is 16, the error is sixteen times that of *(a/c)*: 16×2^{-nbits}. The progression is linear with the size of *offsetx*. Similarly, the calculation of *-(b/c)offsety* has the same relationship with the value of *offsety*. Thus, the error in (9.3a) is the sum of the errors in its two components, expressed as (*offsetx* + *offsety*) 2^{-nbits}. From our initial assumptions, the error in (9.3b+9.3c) is just 2^{-nbits}. Thus, the overall error in the plane equation is:

$$E = (offsetx + offsety + 1)\, 2^{-nbits} \tag{9.4}$$

However, this error applies only locally to a single update array. We have not yet looked at the error introduced by changing the referenced update array with a tiling algorithm. The discussion here focuses on the bounding box tiler because the worst case of the edge-seeking algorithm is the same as the worst case of the bounding box tiling algorithm. An *XInc* signal causes a precomputed value, -32*(a/c)* for a 4x4 update array, to be added to an accumulator which stores the current value of the plane equation constant. The additional error caused by this add is 32 times that of (a/c): 32×2^{-nbits}.

Notice that the *XInc* signal has the same effect on the error as increasing the size of the update array so that it covers the same area covered by the *XInc*. The *YInc* signal has a similar effect. So for ease of analysis, we assume an infinitely large update array, starting at (0,0) and extending outwards. The error of equation (9.4) can be put into terms of pixel coordinate offsets (dx,dy) from the origin of the bounding box (offsetx and offsety are the subpixel coordinate offsets):

$$dx = offsetx / 8$$

$$dy = offsety / 8$$

Therefore,

$$E = 8(dx + dy)\, 2^{-nbits} + 2^{-nbits} \tag{9.5}$$

The overall goal of this exercise is to determine exactly how many bits are needed to the right of the binary point in the fixed point representation of a plane equation. In other words, we need to find the maximum value of 2^{-nbits} that will give an acceptable error. Drop the addition of the last 2^{-nbits} from equation (9.5) because it is negligible:

$$E = 8(dx + dy) 2^{-nbits} \tag{9.6}$$

$$2^{-nbits} = E / 8(dx + dy) \tag{9.7}$$

Assuming the maximum error tolerable is .5, and the maximum values for *dx* and *dy* are 2048 and 2048 respectively, 2^{-nbits} = .5 / 32,768 = 1 / 65,536. Since $\log_2 65{,}536 = 16$, the plane equation data representation should have 16 bits to the right of the binary point.

9.2.2 Bits to the Left of the Binary Point for Plane Equations

Now that we know that 16 bits to the right of the binary point is sufficient, we need to determine the number of bits to the left of the binary point. But this analysis is trivial. The only plane equation values of interest are those contained within the boundary of the geometric figure being rendered. Since the plane equation definition guarantees that the internal values are linear combinations of the three vertex values, the size of the vertex values will bound the size of the internal values.

Temporary results may stray from this restricted range; however, since the high-order bits of these temporary results will be ignored anyway, the data path width need only be large enough to contain the final results. 8-bit plane equations require 8 bits to the left of the binary point; 24-bit plane equations require 24 bits to the left of the binary point. In support of the compare operations, the plane equation data path maintains an extra high-order bit for overflow and underflow detection.

9.2.3 Plane Equation Hardware

The plane equation circuitry can be implemented in a single 40-bit data path, using 24 bits to the left and 16 bits to the right of the binary point. However, this would be very expensive in terms of silicon area. Compromises are necessary to reduce the amount of circuitry required for plane equation evaluation. The first implementation uses three data paths of different sizes: placc8, placc24, and planeEval.

Placc8 and placc24 contain the plane constants and the accumulator that operates on those constants in response to *XInc* and *YInc* signals. Placc8 acts on 8-bit plane equations using 9 bits to the left of the binary point and 16 bits to the right of the binary point. Placc24 acts on 24-bit plane equations using 25 bits to the left of the binary point and 16 bits to the right of the binary point. Placc8 interpolates three 8-bit plane equations in parallel; placc24 interpolates one 24-bit plane equation at a time. The outputs of placc8 and placc24 are multiplexed into the planeEval data path.

The planeEval data path contains the site values for the plane equations, data busses connecting to graphics memory, ALUs to combine graphics memory data with the interpolated plane equations, and other miscellaneous circuitry. PlaneEval is logically a 27-bit data path that can be configured to behave either as a single 25-bit data path (using all 25 bits to the left of the binary point), or as three 9-bit data paths (using all 9 bits to the left of the binary point). PlaneEval maintains the plane site values with lower precision than placc8 and placc24 maintain the plane constant values. This introduces an additional error that is not considered in the analysis above. The additional error is incurred during setup of the site values and can be as much as 1.0. This error is justified by a huge reduction in the size of the plane equation circuitry.

The planeEval data path also has circuitry to limit the effect of the numerical error so that it is not visually objectionable. If color values overflow the allowable number range and wrap to the beginning of the range, then pixels that should be white become black. Similarly, if color values underflow the allowable number range and wrap to the end of the range, then pixels that should be black become white. PixelVision avoids these numerical artifacts by clamping the color values at the ends of the range so that values that overflow get set to the maximum value and values that underflow get set to the minimum value. This clamping is done only for 8-bit plane equations.

9.2.4 Ideal Plane Equation

Now that the error has been quantified, let's look at what it would take to remove the error completely. Ideally, the plane equation could be structured without the division by *c* at initialization:

$$c\ z = -a\ \mathrm{offsetx} - b\ \mathrm{offsety} - a\ \mathrm{originx} - b\ \mathrm{originy} + a\ x1 + b\ y1 + c\ z1 \quad (9.8)$$

XInc and *YInc* signals would cause -a and -b respectively to be added to the plane constant, maintaining exact precision while changing the referenced update array. At the final evaluation stage, to find the actual value of *z* at each site within the update array, the rendering subsystem must divide equation (9.8) by *c*. The error at any pixel now depends only on the final roundoff, not on any intermediate calculations.

However, the price we would pay for this precision is a divider at each site and a significantly larger data path. This is not a practical solution. Since the error can be reasonably contained, and a small error in color or *z* values will generally not cause serious visual problems (compared to errors in the edge evaluation), the non-exact algorithm is a good tradeoff for a VLSI implementation.

A Bibliography

[1] Kelleher, B., and T. Furlong, 1989. PixelStamp Architecture, unpublished. Palo Alto, CA: Digital Equipment Corporation.

[2] Fuchs, H., and J. Poulton. 1981. "A VLSI-Oriented Design for a Raster Graphics Engine", *Lambda* **2(3)**:20-28.

[3] Sproull, R. F., I. E. Sutherland, A. Thompson, S. Gupta, and C. Minter. 1981. "The 8 by 8 Display. Comp. Sci. Dept. Tech. Rep." Pittsburgh, Penn.: Carnegie-Mellon University.

[4] Bresenham, J. E. 1965. "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal* **4(1)**:25-30.

[5] Mammen, A. 1989. "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique", *IEEE CG&A* **0272-1716:43-55.**

[6] Goldfeather, J., and J. Hultquist. 1986, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System", *Computer Graphics* **20(4):**107-116**.**

[7] Bechtolsheim, A., and F. Baskett. 1980. "High-Performance Raster Graphics for Microcomputer Systems", *Computer Graphics* **14(3)**.

[8] Cohen, D., and S. Demetrescu. 1979. "A VLSI Approach to Computer-Generated Imagery, Technical Report." Los Angeles, Calif.: University of Southern California.

[9] Crow, F. C. January, 1981. "A Comparison of Antialiasing Techniques", *IEEE CG&A* **0272-1716**:40-48.

[10] Foley, J. D., and A. Van Dam. 1982. *Fundamentals of Interactive Computer Graphics*. Reading, Mass.: Addison-Wesley.

[11] Gupta, S., R. F. Sproull, and I. E. Sutherland. July 1981. "A VLSI Architecture for Updating Raster-Scan Displays", *SIGGRAPH '81 Conference Proceedings*. ACM 71-78.

[12] Hannah, M. R. 1985. Parallel Architectures for Computer Graphics Displays. Ph.D. Dissertation. Palo Alto, Calif.: Stanford University.

[13] Heckbert, P. S. December, 1986. "Survey of Texture Mapping", *IEEE CG&*A **0272-1716**:56-67.

[14] Newman, W. M., and R.F. Sproull. 1979. *Principles of Interactive Computer Graphics*. New York: McGraw-Hill.

[15] Carpenter, L. 1984. "The A-Buffer, an Antialiased Hidden Surface Method," *Computer Graphics* **18(3)**:103-108.

[16] Clark, J. H., and M. R. Hannah. 1980. "A High-Performance Smart Image Memory", *Lambda*, 3rd Quarter.

[17] Porter, T., and T. Duff. 1984. "Compositing Digital Images", *Computer Graphics*. ACM **18(3)**.

[18] Sproull, R. F. 1979. "Raster Graphics for Interactive Programming Environments", *Computer Graphics* **13(2)**:83-93.

[19] Finkbeiner, D. 1978. *Introduction to Matrices and Linear Transformations*. San Francisco, CA.: W.H. Freeman and Company.

[20] Hwang, K., and F.A. Briggs, 1984. *Computer Architecture and Parallel Processing.* New York, NY.: McGraw-Hill Inc.

[21] Akin, A. 1990. G: A Raster Graphics Acceleration Library for Digital Workstations, unpublished. Palo Alto, CA: Digital Equipment Corporation.

[22] Fisher, F. 1989. Ophir Hardware Specification, unpublished. Palo Alto, CA: Digital Equipment Corporation.

[23] Salesin, D., and Stolfi, J.. 1990. "Rendering CSG Models with a ZZ-Buffer", *Computer Graphics* **24(4)**:67-76.

[24] Rost, R.J. 1988. PEX Introduction and Overview, PEX Version 3.20, MIT X Consortium, Cambridge MA.

[25] PEX Protocol Specification, Version 3.20, PEX Architecture Team, R.J. Rost doc. ed., MIT X Consortium, Cambridge MA.

[26] Programmer's Hierarchical Interactive Graphics System (PHIGS), Draft Standard ISO dp9592-1:1987(E), International Standards Organization, Geneva, Oct. 1987.

[27] Meinerth, K., B. Fanning, and K. Srinivasan, 1988. Badger (PVAX1) Low Cost Graphics/Memory/Video Controller, unpublished. Maynard, MA: Digital Equipment Corporation.

[28] Voorhies, D., D. Kirk, and O. Lathrop, 1988. "Virtual Graphics", *Computer Graphics* **22(4)**:247-253.

[29] Apgar, B., B. Bersack, and A. Mammen. 1988. "A Display System for the Stellar Graphics Supercomputer Model GS1000", *Computer Graphics* **22(4)**:255-262.

[30] Williams, L., 1978. "Casting Curved Shadows on Curved Surfaces", *Computer Graphics* **12(3):**270-274.

[31] Reeves, T., D. Salesin, and R. Cook, 1978. "Rendering Antialiased Shadows with Depth Maps", *Computer Graphics* **21(4):**283-291.

[32] Thomas, G., and R.L. Finney, 1980. *Calculus and Analytic Geometry.* Reading, MA.: Addison Wesley.

[33] Dennis Adams and Albert Xthona. 1990. Stereo Image Display on Microcomputers, StereoGraphics Corporation, San Rafael, CA.