FAST SPHERES, SHADOWS, TEXTURES, TRANSPARENCIES, and IMAGE ENHANCEMENTS IN PIXEL-PLANES *

Henry Fuchs, Jack Goldfeather† , Jeff P. Hultquist, Susan Spach‡
John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

ABSTRACT: Pixel-planes is a logic-enhanced memory system for raster graphics and imaging. Although each pixel-memory is enhanced with a one-bit ALU, the system's real power comes from a tree of one-bit adders that can evaluate linear expressions $Ax + By + C$ for every pixel (x, y) simultaneously, as fast as the ALUs and the memory circuits can accept the results. We and others have begun to develop a variety of algorithms that exploit this fast linear expression evaluation capability. In this paper we report some of those results. Illustrated in this paper is a sample image from a small working prototype of the Pixel-planes hardware and a variety of images from simulations of a full-scale system. Timing estimates indicate that 30,000 smooth shaded triangles can be generated per second, or 21,000 smooth-shaded and shadowed triangles can be generated per second, or over 25,000 shaded spheres can be generated per second. Image-enhancement by adaptive histogram equalization can be performed within 4 seconds on a 512×512 image.

* This research supported in part by the Defense Advance Research Project Agency, monitored by the U.S. Army Research Office, Research Triangle Park, NC, under contract number DAAG29-83-K-0148 and the National Science Foundation Grant number ECS-8300970.

† Department of Mathematics, Carleton College, Northfield, MN, on sabbatical at Department of Mathematics at University of North Carolina at Chapel Hill

‡ Now at Hewlett-Packard Labs, Palo Alto, CA

1. INTRODUCTION

The Pixel-planes development grew out of earlier designs for speeding up raster image generation [Fuchs 1977; Johnson 1979]. An enhanced design is described in [Clark 1980]. In these designs, the task of generating pixels is distributed between several dozen processors. Even when we were designing these systems, we realized that the bottleneck in raster image generation was "pushing pixels," since bottlenecks earlier in the image generation pipeline could be eliminated by fast arithmetic hardware. Two present examples are the Weitek multiplier chips [Weitek] and a custom "geometry engine" chip [Clark 1982]. The limitation of these earlier systems that we sought to overcome with Pixel-planes was that once the number of processors increases to one per memory chip, the bottleneck becomes data movement into the chip. Even if the processor were much faster than the memory chip, in any one memory cycle, only one address-data pair can be put into the chip. Pixel-planes attempts to overcome this limitation by putting computation logic right onto the memory chip, with an entire tree of processing circuits generating many pixels's worth of data in each memory cycle.

Central to the design is an array of logic-enhanced memory chips that form the frame buffer. These chips not only store the scanned-out image but also perform the pixel-level calculations of area-definition, visibility calculation and pixel painting. Recently, various individuals have devised other algorithms for the Pixel-planes engine—for computing shadows, sphere displays, and even image processing tasks. It is increasingly evident that the structure of the machine has greater generality and applicability than first imagined.

Although to many first-time observers Pixel-planes appears to be a variant of the parallel processor with a processor at every pixel, its power and speed come more from the binary tree of one-bit adders that efficiently compute a linear expression in x and y for every pixel in the entire system. Given coefficients A, B, and C, the two multiplier trees and a one-bit adder at each pixel compute $F(x, y) = Ax + By + C$ in bit-sequential order for each (x, y) on the screen (see figure 1). If this expression had to be calculated at each pixel with only the one-bit pixel processor alone, the system would take 20 times as long to complete the calculation!

For efficiency in the actual chip layout, the two multiplier trees have been merged into a single tree and that tree

compressed into a single column. Thus, the system contains a unified multiplier tree, a one-bit ALU at each pixel, a one-bit Enable register (controlling write operations of that pixel), and 32 bits of memory (72 bits in the Pxpl4 implementation now being built). Figure 2 illustrates the organization that is used on the actual memory chips.

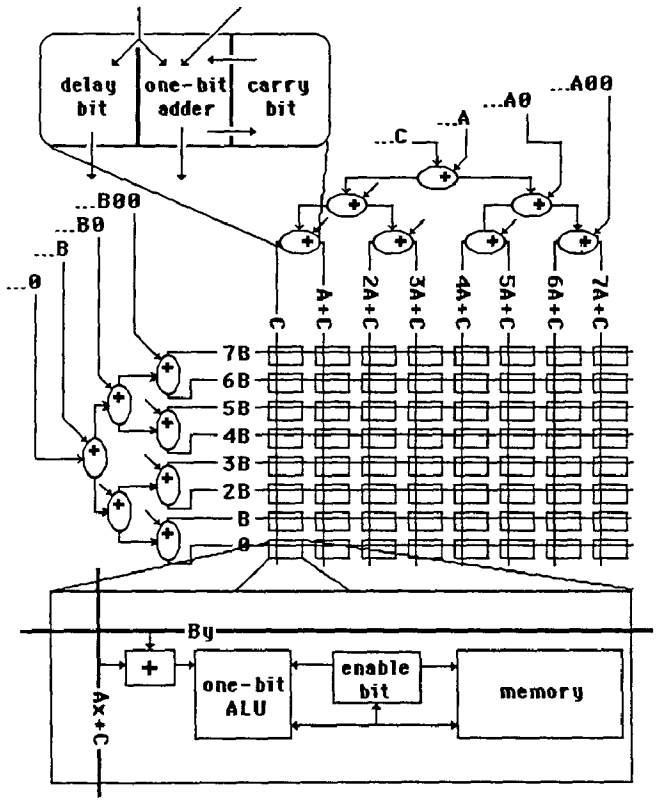


Fig. 1: Conceptual design of an 8×8 Pixel-Planes chip.

The system is driven by a transformation engine, which sends vertices of the database to the *translator*. This board converts this data to a series on linear equations which describe the location of each polygon in screen space. Each linear equation, together with an opcode, is passed to the *image generation controller*, which activates the control lines on the frame buffer chips (see figure 3). Figure 4 shows our latest small working prototype with the color image being generated by six Pxpl3 chips.

Details of the hardware design and the implementation are in [Paeth 1982] and in [Poulton 1985]. The latter of these papers outlines architectural enhancements that may increase the speed of the system by a factor of 5. In the future, we hope to integrate the Pixel-planes architecture with a silicon-based flat-screen display, so that the display itself will handle the display computations [Shiffman 1984; Vuillemier 1984].

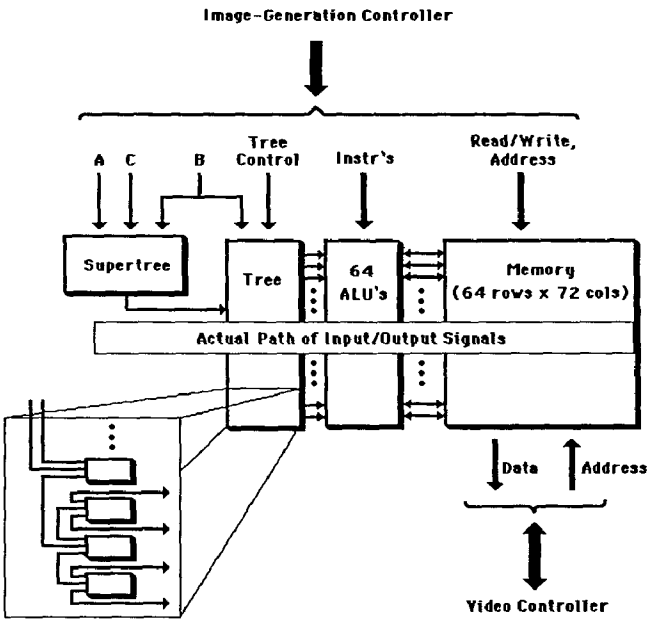


Fig. 2: Floor plan of Pixel-Planes 4 chip.

2. ALGORITHMS IN PIXEL-PLANES

As explained above, the major feature of Pixel-planes is its ability to evaluate, in parallel, expressions of the form $Ax + By + C$, where (x, y) is the address of a pixel. The controller broadcasts A, B, and C, and the expression $Ax + By + C$ is evaluated and then compared and/or combined with information already stored in the memory in each pixel cell. The memory at each pixel can be allocated in any convenient way. A typical allocation might be:

1) buffers for storage of certain key values (e.g., a ZMIN buffer for depth storage, and RED, GREEN, and BLUE buffers for color intensity values)
2) several one-bit flags which are used to enable or disable pixels (via the Enable register) during various stages of processing.

The timing analyses apply to the Pxpl memories and scanout. They assume image generating pipeline modules before them—the geometric transformation unit, the translator, and the controller—operate fast enough to keep up with the Pxpl memories.

2.1 CONVEX POLYGONS

The display of objects made up of polygons is accomplished in three steps: scan conversion of the polygons, visibility relative to previously processed polygons, and shading.

2.1.1 Scan Conversion. The object of this step is to determine those pixels which lie inside a convex polygon. Initially, all Enable registers are set to 1. Each edge of the polygon is defined by two vertices, $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$, which are ordered so that the polygon lies on the left of the directed edge v_1v_2. Then the equation of the edge is $Ax + By + C = 0$, where $A = y_1 - y_2$, $B = x_2 - x_1$, and $C = x_1y_2 - x_2y_1$. Furthermore, $f(x, y) = Ax + By + C$ is positive if and only if (x, y) lies on the same side of the edge as the polygon. The translator computes A, B, and C, and these coefficients are then broadcast to Pixel-planes. A negative $f(x, y)$ causes the Enable register for (x, y) to be set to 0; otherwise the Enable register is unchanged. A pixel is inside the polygon if and only if its Enable register remains 1 after all edges have been broadcast.

2.1.2 Visibility. Once scan conversion has been performed, final visibility of each polygon is determined by a comparison of z values at each pixel. The translator first computes the plane equation, $z = Ax + By + C$, as follows:

Step 1: The plane equation in eye space has the form:

$$A'x_e + B'y_e + C'z_e + D' = 0. \quad (1)$$

The first 3 coefficients, which form the normal to the plane, are found by computing the cross product of the two vectors determined by the first three vertices of the polygon. Alternatively, an object space normal can be part of the polygon data structure and transformed appropriately to produce an eye space normal. Assuming that (x_0, y_0, z_0) is a vertex, the last coefficient is given by:

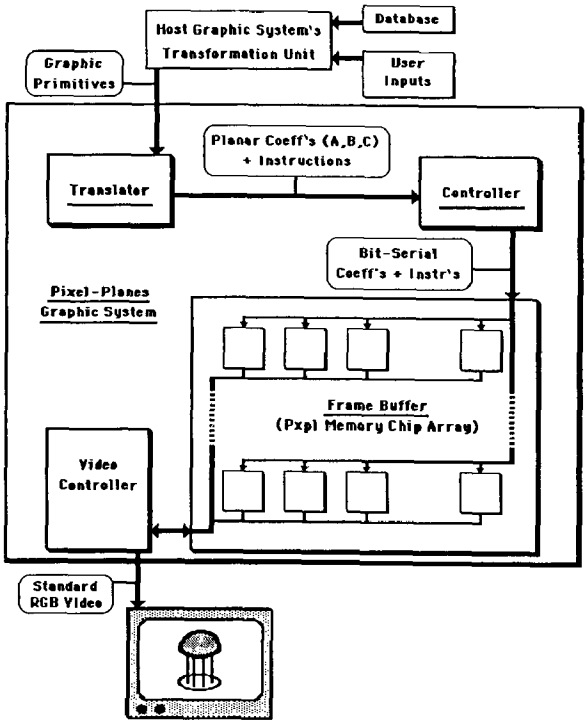


Fig. 3: Logical overview of a 3D graphics system using Pixel-Planes image buffer memory chips.

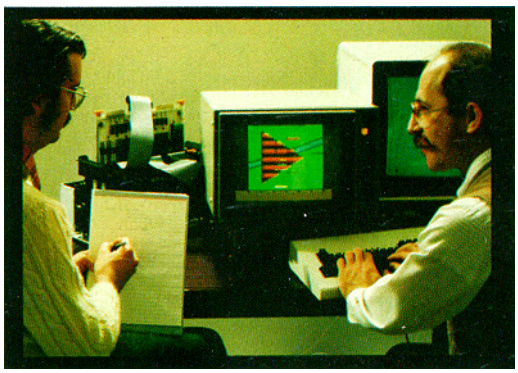


Fig. 4: Pixel-Planes 3 System. John Poulton (left) and Henry Fuchs and the working Pixel-Planes 3 prototype. (photo by Jerry Markatos)

$$D' = -A'x_o - B'y_o - C'z_o \quad (2)$$

Step 2: Using the transformation equations:

$$\begin{aligned} x_e &= (-x - k_1)/(z'r_1), \\ y_e &= (-y - k_2)/(z'r_2), \text{ and} \\ z_e &= -1/z' \end{aligned} \quad (3)$$

where r_1, r_2, k_1, k_2 are constants related to the screen resolution and the location of the screen origin, we can transform (1) to screen space and still retain the form:

$$z' = A'x + B'y + C' \quad (4)$$

Step 3: Given that n bits are reserved for the ZMIN buffer, and that the minimum and maximum z values z_1, z_2, for the object to be displayed are known, we can rescale the equation so that $0 \le z \le 2^n - 1$ by replacing z' by:

$$z = (2^n - 1)(z' - z_1)/(z_2 - z_1) \quad (5)$$

Combining this with (4) we can write z in the form $z = Ax + By + C$. The visible pixels are then determined by using the standard Z-buffer algorithm [Sutherland 1974] at each pixel simultaneously. The controller broadcasts the plane equation of the current polygon, $z = f(x, y) = Ax + By + C$. Each pixel whose Enable bit is still 1 compares its $f(x, y)$ to the value in its ZMIN buffer. The pixel is visible if and only if $f(x, y) < ZMIN$, so pixels with $f(x, y) \ge ZMIN$ set their Enable bits to 0. The controller rebroadcasts A, B, and C so that the still-enabled pixels can store their new ZMIN values.

2.1.3 Shading. To determine the proper color for each pixel, the controller broadcasts 3 sets of coefficients, one for each primary color component. For flat shading, $A = B = 0$ and $C = color$. A smooth shading effect similar to Gouraud shading [Gouraud 1971], created by linearly interpolating the colors at the vertices of the polygon, can also be achieved.

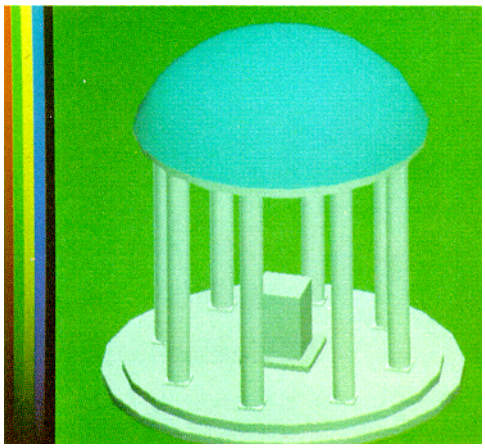


Fig. 5: The Chapel Hill "Old Well" rendered by a Pixel-Planes functional simulator with input of 357 polygons. Estimated image generation time (assuming a 10Mhz clock) is 9 msec.

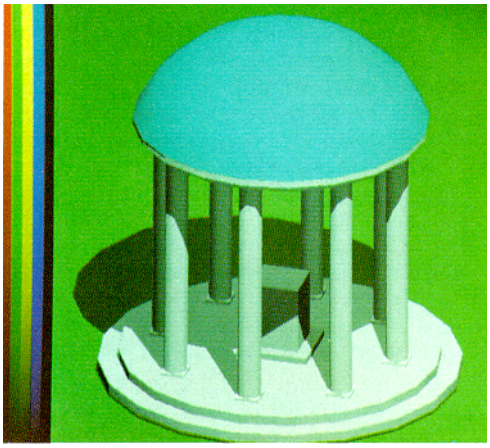


Fig. 6: "Old Well" with shadows (simulation). Estimated time: 13.8 msec.

For example, suppose the polygon has 3 vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3) with red components R_1,R_2,R_3. Geometrically, one can visualize linear interpolation of the red component at (x, y) as selecting the third component of the point (x, y, R) that lies on the plane passing through (x_1, y_1, R_1), (x_2, y_2, R_2), and (x_3, y_3, R_3) in xyR-space. The translator computes the equation of this plane as follows:

Step 1: The vector equation

$$(x,y) = s(x_2 - x_1, y_2 - y_1) + t(x_3 - x_1, y_3 - y_1) + (x_1, y_1) \quad (6)$$

is solved for s and t which are written in the form:

$$\begin{aligned} s &= A_1 x + B_1 y + C_1 \\ t &= A_2 x + B_2 y + C_2 \end{aligned} \quad (7)$$

Step 2: The plane equation is written in the form $R = Ax + By + C$, where

$$\begin{aligned} A &= A_1(R_2 - R_1) + A_2(R_3 - R_1) \\ B &= B_1(R_2 - R_1) + B_2(R_3 - R_1) \\ C &= C_1(R_2 - R_1) + C_2(R_3 - R_1) + R_1 \end{aligned} \quad (8)$$

The controller broadcasts A, B, and C, and $Ax+By+C$ is stored in the RED color buffer for pixels that are still enabled after the scan conversion and visibility computations. If there are more than three vertices, the translator checks the colors $R_4, R_5, \ldots$ at the remaining vertices $v_4 = (x_4, y_4), v_5 = (x_5, y_5), \ldots$. Only in the case that for some i, $R_i \neq Ax_i + By_i + C$ is it necessary to subdivide the polygon by introducing new edges. Note that this subdivision is performed only during the shading stage and is not required during any other phase of processing.

Timing Analysis. The time it takes to process a polygon depends on the number of edges and the number of bits needed for the representation of $Ax+By+C$. Suppose we require an E bit representation for enabling pixels on one side of an edge, a D bit representation for the depth buffer, a C bit representation for each color component, and N bits for the representation of screen coordinates (usually 2 more than the log of the screen resolution), then scan conversion of an edge requires $E+N+3$ clock cycles, and the visibility calculation of a polygon requires $2(D+N+3)$ clock cycles. Once this is determined, shading of the polygon without subdivision requires $3(C+N+3)$ additional clock cycles, while $3(C+N+3)+(E+N+3)$ additional cycles are needed for each subdivision. Hence, the total time to process a "worst case" n-sided polygon is:

$$\begin{aligned} &n(E+N+3) + 2(D+N+3) \\ &\quad + (n-3)(E+N+3) \\ &\quad + 3(n-2)(C+N+3) \end{aligned}$$

clock cycles. If we assume that $E = 12$, $D = 20$, $C = 8$, $N = 11$, $n = 4$, and a clock period is 100 nanoseconds, a 4-sided polygon can be processed in 33 microseconds. Hence, about 30,000 such polygons can be processed per second. This permits real-time display of quite complex objects (see figure 5).

2.2 SHADOWS

After the visible image has been constructed, shadows created by various light sources can be determined (see figure 6). Our approach determines shadow volumes [Crow 1977] defined as logical intersections of half-spaces. This is most similar to [Brotman 1984] except that explicit calculation of the shadow edge polygons is unnecessary in Pixel-planes. Briefly, the algorithm proceeds as follows:

Step 1: Flag initialization. For each pixel, a Shadow flag is allocated from pixel memory, and both the Enable register and Shadow flags are set to 0.

Step 2: Determination of pixels in shadow. For each polygon, the set of visible pixels that lie in the frustum of the polygon's cast shadow are determined and the Enable registers for these pixels is set to 1. The logical OR of Shadow and Enable is then stored in Shadow.

Step 3: Determination of color intensity of shadowed pixels. After all polygons have been processed, those pixels whose Shadow flag is 1 are in the shadow of one or more polygons. The color intensity of these pixels is diminished by an appropriate factor.

The implementation of this algorithm is based on the parallel linear evaluation capability of Pixel-planes, together with *ZMIN* value that is stored for each pixel. The idea is to disable those pixels which are on the "wrong" side of each face of the shadow frustum. We begin by choosing an edge of the current polygon, and finding the plane P determined by this edge and the light source. We want to disable those pixels which are not in the same half-space relative to P as the current polygon (see figure 7). The algorithm must handle two cases.

Case 1: P does not pass through the origin in eye space. In this case we observe that if the eye and the current polygon are in the same half-space relative to P, then it suffices to disable pixels that are farther away than P, and if the eye and the current polygon are in different half-spaces relative to P, then it suffices to disable pixels that are closer than P. In order to accomplish this we do the following:

a) The translator determines the equation of the plane P in the form $z = f(x,y) = Ax + By + C$, chooses a vertex (x_i, y_i) of the polygon not on P, and finds the sign of $f(x_i, y_i)$.

b) The coefficients A, B, and C are broadcast so that f can be evaluated simultaneously at all pixels.

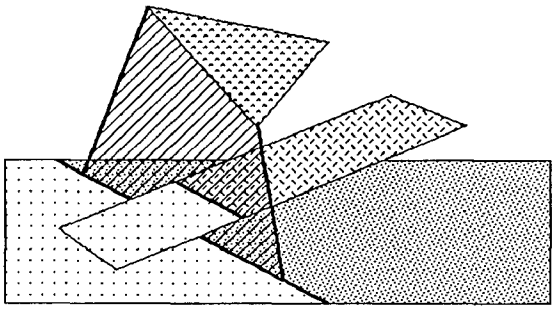
c) If $f(x_i, y_i)$ is positive, all pixels whose *ZMIN* is less than $f(x,y)$ are disabled, and if $f(x_i, y_i)$ is negative, all pixels whose *ZMIN* is greater than $f(x,y)$ are disabled.

Case 2: P passes through the origin in eye space. This relatively rare case is easier to process than Case 1. We observe that P projected on the screen is an edge so it suffices to disable pixels which are not on the same side of this edge as the projected current polygon. We proceed as follows:

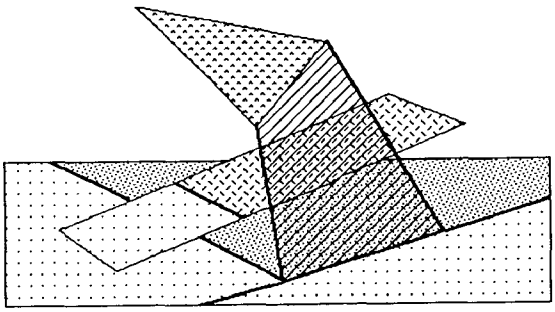
a) The translator determines the edge equation of the intersection of P with the plane of the screen in the form $Ax + By + C = 0$. In addition, the translator determines the sign of $f(x,y) = Ax + By + C$ at a vertex (x_i, y_i) not on P.

b) The coefficients A, B, and C are broadcast and those pixels whose $f(x,y)$ is not the same sign as $f(x_i, y_i)$ are disabled.

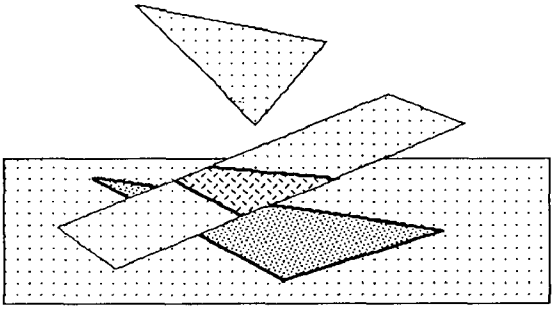
After each edge of the polygon has been processed in this manner, the pixels that are on the same side of the plane of the polygon as the light source must still be disabled. We let P be the plane of the polygon itself, and use



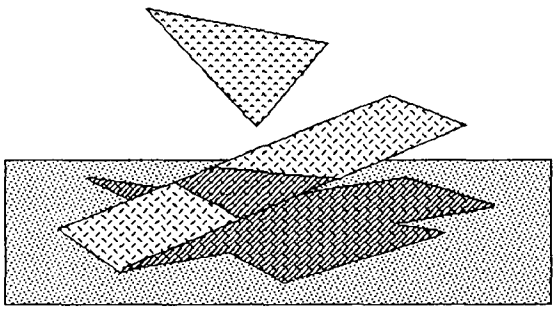
After shadow post-processing of first edge of triangle.



After shadow post-processing of second edge of triangle.



After completing shadow post-processing of triangle.



Result of all shadow processing.

Fig. 7: Shadowing Algorithm

either Case 1 or Case 2 above, with the one exception that we check the sign of f at the light source. Note that in the same half-space relative to P, we disable pixels for which $ZMIN = f(x,y)$, and if they are in different half-spaces we do not disable pixels for which $ZMIN = f(x,y)$. In this way, we can display either the lit or the unlit side of a polygon.

Timing Analysis. Step 1 requires 2 clock cycles for each polygon. In order to process each plane of the shadow frustum of a polygon, we need $(E + N + 3)$ cycles for the broadcast of A,B, and C and 2 additional cycles for the resetting of the Shadow flag. After all polygons have been processed, $3C$ cycles are required to modify the color component. Hence, in order to process P polygons, we need $P\big((n + 1)(E + N + 3) + 2\big) + 3C + 2$ clock cycles. For example, if E=12, N=11, n=4, C=8, and a clock period is 100 nanoseconds, 78,000 polygons can be processed per second.

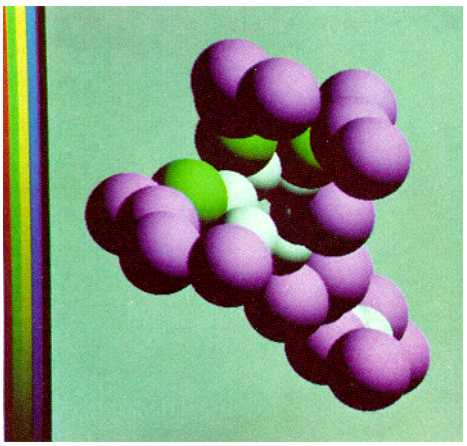


Fig. 9: Trimethoprim (simulation). Presorted data. Estimated time: 1.3 msec.

2.3 CLIPPING

Clipping of polygons by boundary planes, a procedure usually performed in the geometry pipeline, is not necessary when displaying an image in Pixel-planes. Time can be saved by performing only a bounding box type of trivial rejection/acceptance. Edges which lie wholly or partially off the screen will still disable the appropriate pixels during scan conversion. Even hither and yon clipping can be achieved by passing (at most) the two edges of the intersection of the polygon plane with the hither and yon planes, and disabling pixels which are on the appropriate side of these edges. The shadow volumes must be similarly clipped, by the addition of the shadow planes determined by the light source and the line of intersection of the plane of the polygon and each of the clipping planes (see figure 8).

2.4 SPHERES

Fred Brooks suggested to us a method for drawing filled circles in Pixel-planes. We have extended that method to spheres with Z-buffer and an arbitrary light source. Since Pixel-planes is essentially a linear machine, it might seem difficult to display objects rapidly which are defined via quadratic expressions. However, by using an algorithm that, in effect, treats a circle as a polygon with one edge, and by using some appropriate approximations, we can overcome these difficulties (see figures 9,10). Just as in polygon display, we proceed through a scan conversion, a visibility, and a shading phase [Max 1979; Pique 1983].

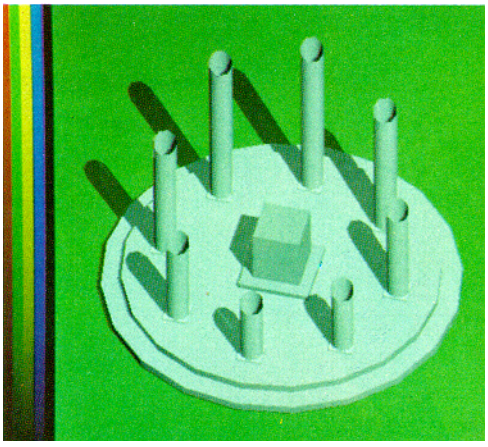


Fig. 8: "Old Well" with shadows cut by hither plane within the Pixel-Planes memories (simulation). 177 polygons after trivial rejection. Estimated time: 8.8 msec.

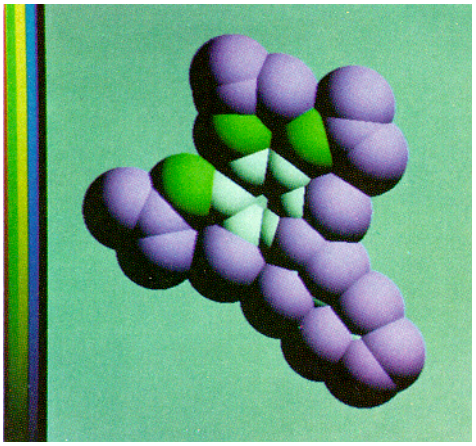


Fig. 10: Trimethoprim with Z-buffer (simulation). Unsorted data. Estimated time: 1.7 msec.

Step 1: Scan Conversion. Note that the equation of a circle with radius r and center (a,b) can be written in the form:

$$g(x,y) = Ax + By + C - Q = 0 \tag{9}$$

where $A = 2a$, $B = 2b$, $C = r^2 - a^2 - b^2$, and $Q = x^2 + y^2$. A section of the memory at each pixel, called the Q-buffer, is allocated for the storage of $x^2 + y^2$, and is loaded with this value at system initialization time. The translator computes A, B, and C and $f(x,y) = Ax + By + C$ is evaluated at each pixel. The value in the Q-buffer is subtracted from $f(x,y)$ and those pixels for which $f(x,y) - Q$ is negative are disabled.

Step 2: Visibility. If the eye coordinate system is chosen so that the $z > 0$ half-space contains the sphere, then the visible hemisphere is the set of points (x,y,z) satisfying

$$z = c - \sqrt{r^2 - (x-a)^2 - (y-b)^2} \tag{10}$$

where r is the radius and (a,b,c) is the center of the sphere. We can approximate this by

$$z = c - \left(r^2 - (x-a)^2 - (y-b)^2\right)/r \tag{11}$$

which in effect approximates the hemisphere with a paraboloid. Using a method similar to that described in Step 1, the expression in (11) can be evaluated, compared with the existing contents of the ZMIN buffer, and then stored if necessary, in the ZMIN buffer. Visibility is then determined in the same way as it is for polygon display.

Step 3: Shading due to light sources at infinity. The unit outward normal at the visible point (x,y,z) on the sphere with center (a,b,c) and radius r is

$$\begin{aligned}\overline{N} &= (1/r)(x-a,\ y-b,\ z-c) \\ &= (1/r)\left(x-a,\ y-b,\ -\sqrt{r^2-(x-a)^2-(y-b)^2}\right)\end{aligned} \tag{12}$$

Let $\overline{L} = (l_1, l_2, l_3)$ be the unit direction of an arbitrary light source. Then the point of maximum highlight on the sphere is $(rl_1 + a, rl_2 + b, rl_3 + c)$. Denote by *CMIN* the ambient color value and by *CMAX* the maximum color value for a given color component. Then for diffuse shading of the sphere, the color value at (x,y) is

$$Color(x,y) = \tag{13}$$

$$\begin{cases} CMIN + (CMAX - CMIN)(\overline{L}\cdot\overline{N}), & \text{if } \overline{L}\cdot\overline{N} \geq 0; \\ CMIN, & \text{if } \overline{L}\cdot\overline{N} < 0. \end{cases}$$

Using the parabolic approximation of the hemisphere as we did in Step 2, we can approximate $\overline{L}\cdot\overline{N}$ by:

$$\begin{aligned}\overline{L}\cdot\overline{N} \approx\ & (l_1(x-a) + l_2(y-b))/r \\ & - l_3(r^2 - (x-a)^2 - (y-b)^2)/r^2\end{aligned} \tag{14}$$

Then the color at a given pixel can be written in the form:

$$Color(x,y) = K(Ax + By + C - Q) + CMIN \tag{15}$$

where

$$\begin{aligned} K &= -(CMAX - CMIN)l_3/r^2, \\ A &= -l_1 r/l_3 + 2a, \\ B &= -l_2 r/l_3 + 2b, \\ C &= l_1 ra/l_3 + l_2 rb/l_3 + r^2 - a^2 - b^2 \end{aligned} \tag{16}$$

The translator computes A, B, C, and K. Multiplication by K is accomplished by first approximating K by the first n non-zero bits of its binary representation:

$$K \approx \sum_{i=1}^{n} 2^{j_i} \tag{17}$$

Then for each j in the sum, the controller broadcasts $2^j A$, $2^j B$, $2^j C$. Q is shifted by j bits and subtracted from the linear expression determined by the three broadcast coefficients. The resultant value:

$$2^j(Ax + By + C - Q) \tag{18}$$

is added to the contents of the appropriate color buffer, COLBUF. After all the terms in the sum have been processed, we set *COLBUF* to 0 if *COLBUF* < 0. The constant value *CMIN* is broadcast and added to *COLBUF*.

Timing Analysis. The initial loading of the Q-buffer requires $37(E + N + 3)$ clock cycles. Scan conversion and visibility are the same as in polygon processing and take $(E + N + 3)$ and $2(D + N + 3)$ cycles, respectively. Shading requires $4(C + N + 3)$ cycles for each term in the sum used to approximate K, and the broadcast of *CMIN* requires 20 cycles. Hence, if k is the number of terms in the approximation of K, it takes

$$\begin{aligned} &37(E + N + 3) \\ &\quad + S\big((E + N + 3) + 2(D + N + 3) \\ &\quad + 4k(C + N + 3) + 20\big) \end{aligned}$$

clock cycles to process S spheres. For example, if $k = 3$, $E = 20$, $N = 11$, $D = 20$, $C = 8$, then 34,000 spheres can be processed per second.

2.5 ADAPTIVE HISTOGRAM EQUALIZATION

In computed tomographic (CT) scan displays, CT numbers must be assigned (grey) intensity levels so that the viewer can perceive appropriate degrees of contrast and detail. Because the range of CT numbers is, in general, greater than the range of intensity levels, some compression has to take place. This makes it difficult to control the contrast in both light and dark areas. The standard method, selection of windows in the CT range, results in intensity discontinuities and loss of information. AHE [Pizer 1984] is an assignment scheme that makes use of regional frequency distributions of image intensities. The processed image has high contrast everywhere and the intensities vary smoothly (see figures 11,12). The method proceeds as follows. For each point (x, y) in the image:

Step 1: A "contextual" region centered at (x, y) is chosen, and the frequency histogram of CT numbers in this region is computed. Typically, this region is a circle, or a square with edges parallel to the screen boundaries.

Step 2: In this histogram, the percentile rank, r, of the CT number at (x, y) is determined.

Fig. 11: Original CT scan image.

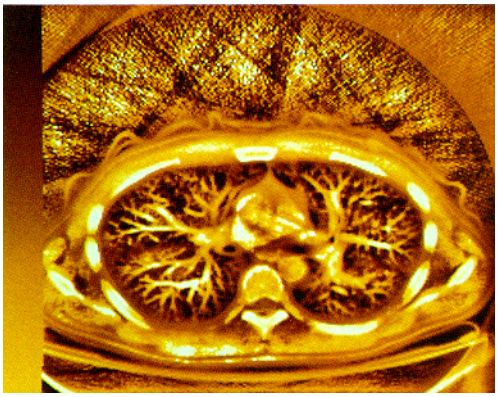


Fig. 12: CT scan image after AHE enhancement (simulation). Estimated time for this 256×256 pixel image: 1 second.

Step 3: This rank is used to compute an intensity level, i, in some grey scale ranging between , say, i_1 and i_2. Specifically, $i = i_1 + r(i_2 - i_1)$.

This method requires the computation of a CT distribution at every pixel in the image, and so it is far too inefficient for most uses, requiring approximately 5 minutes to compute on a 256 × 256 image on a VAX 11/780. A more efficient alternative, requiring about 30 seconds for a 256 × 256 image, is to compute the distribution only at a small set of sample points and use a linear interpolation scheme to approximate the intensity levels at the other points.

An efficient alternative, which finds the exact value at each pixel, can be implemented in Pixel planes. The idea is to make use of the parallel processing capability to construct the rank incrementally at each pixel simultaneously.

Step 1: The CT numbers are loaded into the pixel memories, and a counter at each pixel is initialized.

Step 2: For each pixel (x_0, y_0):

a) The coefficients necessary to disable those pixels that are outside the contextual region centered at (x_0, y_0) are broadcast. For example, if the region is a polygon or a circle, this is equivalent to the scan conversion step discussed earlier.

b) The CT number, $N(x_0, y_0)$, is broadcast and compared, in parallel, to the CT number, $N(x, y)$ which is stored at each enabled pixel (x, y). If $N(x, y) > N(x_0, y_0)$, the counter at (x, y) is incremented.

Step 3: After all pixels have been processed, the counter at each pixel contains the rank of the pixel CT number within its own contextual region. If both the number of pixels in the contextual region and the length of the grey scale are powers of 2, this rank can easily be scaled to an intensity by shifting bits.

Timing Analysis. It requires 25 cycles to load each pixel with its CT value and initialize its counter. It requires $2(E+N+3)$ cycles to disable pixels outside each contextual region and 40 cycles to broadcast the CT numbers and increment the counters. On a 512 × 512 display with $N = 11$ and $E = 12$, we have estimated the time required to perform AHE is about 4 seconds.

3. ALGORITHMS UNDER DEVELOPMENT

This section describes algorithms still under development. Only functional simulations (rather than detailed behavioral ones) have been executed and the timing estimates are thus less precise. In particular, we are still exploring speedups for multiplication and division in the pixel processors. The timing estimates given in the figures are conservative (we hope), but still assume a 10MHz clock.

3.1 TEXTURE MAPPING

One way of producing a texture on a polygon is to compute a texture plane address (u, v) associated to each pixel (x, y) and then look up the appropriate color value in a texture table indexed by u and v. The Pixel planes linear evaluator can be used to determine, in parallel, this texture plane address.

To see how this is done, we proceed through some mathematical computations. In order to orient a texture on a polygon in eye space we first choose a point $P0$ on the polygon and 2 orthonormal vectors $\overline{S}$ and $\overline{T}$ in the plane of the polygon. Then the texture address (u, v) associated to the point X on the polygon is given by:

$$\begin{aligned} u &= \overline{S} \cdot (X - P_0), \\ v &= \overline{T} \cdot (X - P_0). \end{aligned} \tag{19}$$

If $\overline{S} = (s_1, s_2, s_3)$, $\overline{T} = (t_1, t_2, t_3)$, $P = (p_1, p_2, p_3)$, and $X = (x_e, y_e, z_e)$, equations (19) can be rewritten in coordinate form as:

$$\begin{aligned} u &= s_1(x_e - p_1) + s_2(y_e - p_2) + s_3(z_e - p_3) \\ v &= t_1(x_e - p_1) + t_2(y_e - p_2) + t_3(z_e - p_3). \end{aligned} \tag{20}$$

Substituting the equations (3), which relate screen space to eye space, into (20) and using the plane equation $Ax_e + By_e + Cz_e + D = 0$, we can write u and v in the form:

$$\begin{aligned} u &= (A_1 x + B_1 y + C_1)/z \\ v &= (A_2 x + B_2 y + C_2)/z \end{aligned} \tag{21}$$

The translator computes A_1, B_1, and C_1, and the controller broadcasts them to Pixel-planes. The division of

$A_1 x + B_1 y + C_1$ by z (which is already stored in ZMIN) is done in parallel at the pixel level, and the result is stored in a U-buffer. The V-buffer value is found in a similiar manner. A texture table is then passed, entry by entry, to Pixel-planes, and each pixel selects a texture value corresponding to its stored (u, v) value. For periodic patterns (checkerboards, bricks, etc.) it is only necessary to transmit a small table defining the unit pattern (see figure 13).

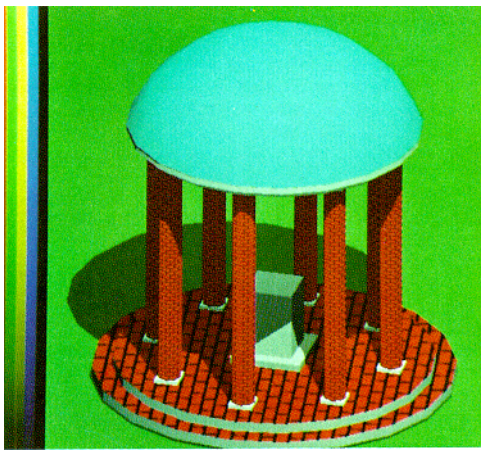


Fig. 13: Bricked "Old Well" (simulation). 66 textured polygons out of a total of 357. Estimated time: 14.3 msec.

3.2 TRANSPARENCY

Transparency effects can be achieved by disabling patterns of pixels prior to polygon processing. For example, one could broadcast the coefficients 1, 1, 0 in order to evaluate $x + y$, and disable those pixels for which $x + y$ is even (see figure 14).

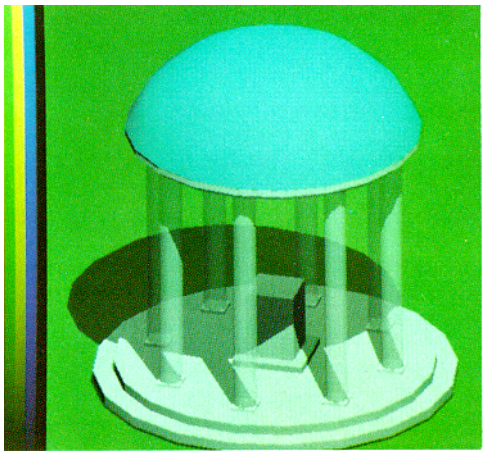


Fig. 14: "Old Well" with transparent columns (simulation). 64 transparent polygons out of a total of 357. Estimated time: 13.8 msec.

Transparency effects can also be produced with subpixel mask successive refinement, where transparent polygons are ignored on particular passes over the database. For example, transparent polygons can be ignored every other pass or every third pass, thereby yielding different degrees of transparency.

3.3 ANTI-ALIASING

We have been developing several anti-aliasing techniques for polygons. We have come to believe that the essential difference between various approaches is whether the visibility at the subpixel level is performed before or after the anti-aliasing computations. Our first approach, which aims at producing an image rapidly and "improving" the image with each screen refresh, makes no assumptions about visibility determination before the Pxpl memories. The second approach, which takes more time, but produces a high quality anti-aliased image initially, assumes visibility ordering has already been done.

Method 1: Successive Refinement. Each pixel (x, y), is subdivided into a grid of subpixels so that each subpixel has an address of the form $(x + xoffset, y + yoffset)$. We generate the image several times (16, perhaps), each time offsetting the image slightly by some $(xoffset, yoffset)$ in such a way that the sample points within a pixel's area form a reasonable distribution. (The shift is easily achieved by adding $A \cdot xoffset + B \cdot yoffset$ to the C coefficient of each broadcast triple.) Two sets of color buffers are maintained, one to store the color generated by the latest image generation offset and the other to store a running average as we move around the subpixel grid.

The extra cost of the algorithm over standard subpixel "super-sampling" is the color blending between each pass over the graphic database. This is less than 1000 clock cycles (100 microseconds) per pass. This particular super-sampling successive refinement technique, however, supports dynamically interactive applications. The initial images appear similar to common anti-aliased images, and significant refinement is produced within a few additional sampling passes.

Method 2: Subpixel Coverage Mask. The polygons are sorted from front to back, perhaps by first transforming the polygon list into a BSP tree [Fuchs 1983]. Each pixel is subdivided into a number of subpixels and one bit of the pixel memory is reserved for each such subpixel. During the scan conversion step of polygon processing, the coefficients defining each edge are normalized to yield the distance from the center of the pixel to the edge. The coverage mask and area contribution of an edge can be passed from a precomputed table [Carpenter 1984] in the controller indexed by this distance and A, the coefficient of x. (Note that only one row of the table needs to be passed for any edge.) The number of ones in the mask is used to compute a color contribution which is added to the color buffers. When the number of ones in the coverage mask stored at each pixel reaches the total number of subpixels, the pixel is disabled. Since polygons are processed in front to back order, "leakage" of color from hidden polygons is avoided. This approach is somewhat similar to the one used in the Evans and Sutherland CT-5 real-time image generation system often used for flight training [Schumacker 1980].

4. CONCLUSIONS

We have highlighted in this paper the aspects of Pixel-planes that give it computing power and efficiency—the parallel linear expression evaluator embodied in the tree of one-bit adders. We have illustrated this capability by describing a variety of algorithms (shadows, spheres, image enhancement) that appear to run efficiently in this machine. Pictures from the Pixel-planes simulators indicate that high-quality images can be generated rapidly enough for dynamic, often real-time, interaction. The images from the working small prototype (see figure 4) are simpler than the images from the simulators due to the small number of custom chips presently available. We expect Pixel-planes 4, with considerably increased speed and resolution, to start working by June 1985. We expect that a full-scale (500–1000 line) display system can be built with less than 500 Pxpl memory chips in currently available (1.5micron CMOS) technology. We also hope that the algorithm developments, especially those based on simplifying algorithms into linear form, will be useful for those developing graphics algorithms on other parallel machines.

5. ACKNOWLEDGEMENTS

We wish to thank Fred Brooks for the basic circle scan-conversion algorithm, Alan Paeth and Alan Bell of Xerox Palo Alto Research Center for years of assistance with the design and early implementations of Pixel-planes, Scott Hennes for assistance with the implementation of the Pxpl3 memory chip, Hsieh Cheng-Hong and Justin Heinecke for discussions about architecture and algorithm interactions, Turner Whitted for discussions about anti-aliasing and transparency algorithms, Eric Grant for 3D data of the Old Well, Steve Pizer, John Zimmerman, and North Carolina Memorial Hospital for CT chest data, Mike Pique, Doug Schiff, Dr. Michael Corey and Lee Kuyper (Corey and Kuyper from Burroughs Wellcome) for Trimethoprim drug molecule data, Trey Greer for TEX help, and Bobette Eckland for secretarial support. Special thanks go to Andrew Glassner, who supervised the layout and paste-up of this paper.

6. REFERENCES

Brotman, L.S. and N.I. Badler. October 1984. "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications*, 5–12.

Carpenter, L. July 1984. "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics*, **18**(3), 103–109 (Proc. Siggraph '84).

Clark, J.H. July 1982. "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, **16**(3), 127–133 (Proc. Siggraph '82).

Clark, J.H. and M.R. Hannah. 4th Quarter, 1980. "Distributed Processing in a High-Performance Smart Image Memory," *Lambda*, 40–45 (*Lambda* is now VLSI Design).

Crow, F.C. July 1977. "Shadow Algorithms for Computer Graphics," *Computer Graphics*, **11**(2), 242–248 (Proc. Siggraph '77).

Fuchs, H. 1977. "Distributing a Visible Surface Algorithm over Multiple Processors," *Proceedings of the ACM Annual Conference*, 449–451.

Fuchs, H. and B. Johnson. April, 1979. "An Expandable Multiprocessor Architecture for Video Graphics," *Proceedings of the 6th ACM-IEEE Symposium on Computer Architecture*, 58–67.

Fuchs, H., J. Poulton, A. Paeth, and A. Bell. January, 1982. "Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System," *Proceedings of the 1982 MIT Conference on Advanced Research in VLSI*, 137–146.

Fuchs, H., G.D. Abram, and E.D.Grant. July 1983. "Near Real-Time Shaded Display of Rigid Objects," Computer Graphics, **17**(3), 65–72 (Proc. Siggraph '83).

Gouraud, H. 1971. "Computer Display of Curved Surfaces," *IEEE Transcations on Computers*, **20**(6), 623–629.

Max, N.L. July 1979. "ATOMILL: Atoms with Shading and Highlights," *Computer Graphics*, **13**(3), 165–173 (Proc. Siggraph '79).

Pique, M.E. 1983. Fast 3D Display of Space-Filling Molecular Models, Technical Report 83-004, Department of Computer Science, UNC Chapel Hill.

Pizer, S.M., J.B. Zimmerman, and E.V.Staab. April 1984. "Adaptive Grey Level Assignment in CT Scan Display," *Journal of Computer Assisted Tomography*, **8**(2), 300–305, Raven Press, NY.

Poulton, J., J.D. Austin, J.G. Eyles, J. Heinecke, C.H. Hsieh, and H. Fuchs. 1985. Pixel-Planes 4 Graphics Engine, Technical Report 1985, Department of Computer Science, UNC Chapel Hill (to appear).

Schumacker, R.A. November 1980. "A New Visual System Architecture," *Proceedings of the 2nd Annual IITEC*, Salt Lake City.

Shiffman, R.R. and R.H. Parker. 1984. "An Electrophoretic Image Display With Internal NMOS Address Logic and Display Drivers", *Proceedings of the Society for Information Display*, **25**(2), 105–152.

Sutherland, I.E., R.F. Sproull, and R.A. Schumacker. 1974. "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, **6**(1), 1–55.

Vuillemier, R., A. Perret, F. Porret, P. Weiss. July 1984. "Novel Electromechanical Microshutter Display Device," *Proceedings of the 1984 Eurodisplay Conference.*

Weitek. 1983. *Designing with the WTL 1032/1033*, Weitek Corporation, Santa Clara, CA (Weitek publication 83AN112.1M).