Optimal Static 2-Dimensional Screen Subdivision for Parallel Rasterization Architectures

Donald McManus and Carl Beckmann
Thayer School of Engineering, Dartmouth College, Hanover, NH 03755
Sanders, a Lockheed Martin Co., Nashua, NH 03061
e-mail [donald.mcmanus,carl.beckmann]@dartmouth.edu

Abstract

Designers of computer graphics hardware have used increasing device counts available from IC manufacturers to increase parallelism using techniques such as putting a longer pipeline of data path elements on integrated circuits, or developing designs which use an array of processors. Pixel-Planes 1-5 and PixelFlow[1] are examples of architectures which use an array of pixel processors for rasterization. Early generations of Pixel-Planes attempted to make these arrays as large as the display providing one processor for each display pixel. Later generations improved performance by grouping processors into multiple smaller arrays, subdividing the screen into sections of a corresponding size and having the arrays independently process the screen subdivisions. This paper describes simulations which were performed to determine the optimum size subdivision for a graphics computer which uses Pixel-Planes type parallelism, i.e. static two dimensional screen subdivision parallel polygon rasterization. We then develop a mathematical approach to determining the optimal subdivision size and show that it agrees well with the experimental data. For special purpose architectures we show that the optimal size depends not only on the polygon size but also on the silicon area consumed by the rasterizer overhead. The mathematical approach can be directly applied to special purpose architectures, and we show how it can be modified for use in analyzing algorithms developed for general purpose architectures such as the Intel Touchstone or Paragon, or the Thinking Machines CM-5.

1 Introduction

Architectures, such as Pixel-Planes 1-5 and PixelFlow implement the rasterizer with a two dimensional array of pixel processors, where each pixel processor is dedicated to a pixel on the screen. In the early generations of Pixel-Planes, the designers attempted to create an array of pixel processors equal to the number of pixels on the screen. For example, for a 512x512 display size, an array of 512x512 pixel processors was provided. This would ensure that all polygons would be processed only once. However, the designers discovered that most of the pixel processors remained idle during the processing of any particular polygon, since the average size of a polygon is much smaller than the screen size and only the pixel processors actually covered by a polygon are utilized in processing that polygon. Later generations of Pixel-Planes sub-divided the screen into regions, and provided multiple arrays where each array was the size of one region. This approach resulted in higher pixel processor utilization, but resulted in some polygons having to be processed more than once since they overlapped more than one region of the screen. (The average number of regions covered by a polygon is referred to as the *overlap factor*.) However, the higher pixel processor utilization more than compensated for this penalty.

The designers of Pixel-Planes discovered that it is more efficient to provide pixel processor arrays smaller than the size of the display. Other researchers have also investigated methods to improve performance by intelligently performing screen subdivision. In [ROBL88] and [WHIT94] methods are described which dynamically subdivide the screen, where subdivisions are created to improve processor load balancing. Pixel-Planes 5 [FUCH89] uses a static screen subdivision and load balancing is improved by using dynamic assignment of pixel processor arrays to the subdivisions. In [ELLS94] a static screen subdivision method is described, and it is shown that load balancing in a static approach depends on a high "granularity ratio", which is the ratio of regions to rasterizers. A high granularity ratio implies subdividing the screen a large number of times, but more subdivisions imply more polygons will be processed multiple times. A balance between these two factors must be achieved.

In this paper we describe a method which analytically determines the optimal size of static screen subdivisions. It seems intuitive that the optimum size is dependent on the average size of the polygons. As it turns out, polygon size is only one factor. Our research has discovered a more complex relationship between the polygon size and the *overhead* inherent in the rasterizer hardware. Overhead is any logic which is not part of a pixel processor but needs to be provided for each array, an example is logic which controls the array. In the extreme case where there is no overhead, the optimum subdivision size is independent of the polygon size! The optimum size is dependent on rasterizer performance and cost, and the number of polygons which overlap multiple regions. In [MOLN94] an equation for determining the overlap factor was presented. We use a more accurate equation in our analysis, presented in Section 6.0 below.

2 MIMD Array of SIMD Processors

The interesting problem posed by PixelFlow is that it uses two levels of parallelism. The rasterizer is a 2-dimensional array of SIMD pixel processors (the first level of parallelism). A rasterizer is paired with a geometry engine to form a renderer, and the renderer is replicated in a MIMD array (the second level of parallelism). Increasing the size of the SIMD array of pixel processors will reduce the overlap factor, and speed-up polygon processing. However, increasing the SIMD array makes it more expensive to replicate and will lower the number of renderers in the MIMD array, given a fixed system cost. Therefore a trade-off

[1]Pixel-Planes 1-5 and PixelFlow were developed at the University of North Carolina.

must be performed to determine the optimal size of the SIMD array so that it can be efficiently replicated.

To aid in our discussion we need to briefly review the PixelFlow architecture. (For a complete description of the PixelFlow architecture see [MOLN92].) In PixelFlow (see figure 1), polygons are randomly distributed to the geometry engines. The geometry engines transform the polygons then sort them into bins dependent on which screen subdivision the polygon is in with some polygons put in more than one bin. After all polygons are transformed the local rasterizer processes polygons in the first bin then moves on to the other bins. The rasterizer array operates by having each pixel processor in the array determine whether it is inside or outside of the current polygon [FUCH82]. If it is outside it turns itself off, otherwise it participates in the calculations for the current polygon. The time required for a region to process a polygon is independent of the polygon's size, since the polygon's pixel values are simultaneously calculated. Therefore, the amount of time required by PixelFlow to rasterize an image can be calculated by determining how many polygons are processed in each region. This demonstrates that PixelFlow's rasterizer array processing power increases proportionally with the screen subdivision size (which may not be the case in other architectures). The pixels generated by the rasterizers are then combined by the composition nodes to create the final image.

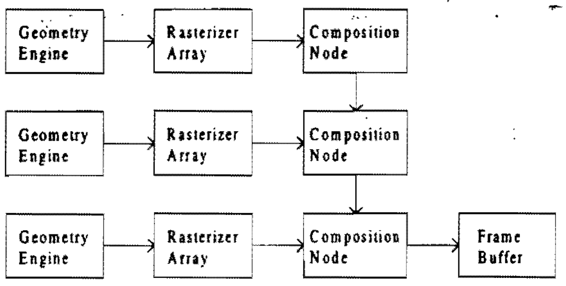


Figure 1) PixelFlow Block Diagram

Throughout the remainder of the paper we will be discussing PixelFlow, however most of the discussion is applicable to any architecture or algorithm which uses 2-dimensional screen subdivision for rasterization. Also, throughout the rest of the paper we will use the term *pixel processor* when referring to one of the processors in the rasterizer array. We will use the term *array* or *rasterizer* when refering to an array of pixel processors. We will use the term *renderer* to refer to a geometry engine / rasterizer pair, and use the term *system* to refer to all renderers.

3 Modeling PixelFlow

Our research began by developing a VHDL algorithmic model similar to OpenGL, the graphics pipeline we plan to implement. The VHDL model includes all operations required to transform polygon vertices, calculate lighting values, clip to the view volume, rasterize the transformed polygons and perform all per pixel operations. The model was used to generate all the images in figure 8 (last page). The figures (a) through (d) in figure 8 were selected because they represent real application data. Figures (a) and (b) are generated from digital terrain and elevation data which is widely used in military applications. Figures (c) and (d) are from the National Computer Graphics Association, Graphics Performance Characterization Committee's Picture Level Benchmark and represent some typical scientific and animation data. Figures (e) and (f) do not necessarily represent real application data, but were included to test our model against extreme examples, figure (e) uses unusually large triangles, and figure (f) has a high depth complexity.

The VHDL model gives direct access to all intermediate data in the pipeline. For the array size experiments we were particularly interested in the intermediate polygon data after geometric transformations, but before rasterization. In the PixelFlow architecture these transformed polygons are sorted into bins, one bin for each screen subdivision. Statistics gathering routines in our model measure the properties of the transformed polygons. One measure is how many rasterization cycles are required for different screen subdivision sizes. We first determine in which bins each polygon is put if the screen is subdivided into regions of size *A* by *A* pixels. To speed up sorting, the rectangular bounding box is used instead of the actual polygon to determine the bins. This approach significantly simplifies the sorting but also creates some inefficiency by having some polygons put in bins in which they don't need to be. The values of *A* we used were all powers of 2 up to the dimensions of the screen size. That is, for a screen size of 512x512 pixels, we used subdivision region sizes of 1x1, 2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128, 256x256 and 512x512. The statistics from the image shown in figure 8a, containing 50,041 polygons with an average polygon bounding box size of 23.91 pixels, is shown in table 1.

Subdivision Size (*A*x*A*)	Number of Rasterization Cycles (R_A)	Overlap Factor
1x1	1,196,367	23.91
2x2	418,132	8.36
4x4	182,325	3.64
8x8	103,069	2.06
16x16	73,077	1.46
32x32	60,400	1.21
64x64	54,714	1.09
128x128	52,020	1.04
256x256	50,754	1.01
512x512	50,041	1.00

Table 1) Rasterization Time and Overlap Factor versus Subdivision Size

As seen in table 1, screen regions of 1x1 required the most rasterization cycles to complete, and regions of 512x512 required the fewest to complete. In fact, the cycles required for the 512x512 array is equal to the number of polygons, since each polygon fits into the array.

The time to generate a frame can be calculated by using the number of rasterization cycles for a particular array size, divided by the number of arrays of that size which are provided in the system multiplied by the time to perform 1 rasterization cycle (this assumes perfect load balancing between rasterizers). For example, if two 512x512 arrays are provided and a rasterization cycle is 1 microsecond, then the time to generate the frame is:

$$\frac{R_{512}}{\#\,\text{Renderers}} * \text{seconds}/\text{raster}-\text{cycle}$$
$$= \frac{50{,}041}{2} * 0.000001$$
$$= 0.025021 \text{ seconds}$$

In the extreme case, if R_{512} 512x512 arrays were provided, the time to generate a frame would be one raster cycle, i.e. 1 microsecond. If 1x1 arrays were used, the rasterization time can also be brought down to 1 microsecond by providing R_1 1x1 processor arrays. Using the numbers from table 1, the total number of *pixel* processors for each array size for systems which require only one raster cycle is:

For 512x512 array:

$$R_{512} * (512 * 512) = 50{,}041 * 262{,}144$$
$$= 13{,}117{,}947{,}900$$

For 1x1 array:

$$R_1 * (1 * 1) = 1{,}196{,}367 * 1$$
$$= 1{,}196{,}367$$

From this example we see that several orders of magnitude fewer pixel processors are required for the 1x1 array size to achieve maximum performance. However, numbers of pixel processors is only one factor to consider. Determining which array size to choose, also requires determining the cost of each array. The cost of the array needs to take into account the other hardware necessary to make the array fit in the system, i.e. control logic, geometry engine(s), etc. These additional costs will be addressed in Section 5.0.

4 Rasterizer Array Cost

In the image from table 1, which was composed mostly of small polygons, given the choice of one 512x512 or four 256x256 arrays where both choices have the same number of pixel processors, the better selection would be the four 256x256 arrays. However, even though the two choices have equal numbers of pixel processors, the cost of four 256x256 arrays is greater than one 512x512. This is because there is overhead associated with each array. In the case of PixelFlow, there is the overhead of an additional controller chip, the integrated circuits to implement the geometry engine, and memory chips to hold the sorted polygons. An accurate cost model for an array must include this overhead.

Several useful measures of cost include silicon area, dollars (manufacturing and parts purchase costs), weight and power. The metric we use throughout this paper is silicon area, but all discussions could be applied to any of the listed metrics. The metric of silicon area was chosen for two reasons: First, because one of our primary concerns was board area due to the target environment of our system, and second because the metric of silicon area measures the efficiency of the architecture. If two systems are designed for identical purposes, and they have equal performance, then the system which uses less silicon is making more efficient use of the silicon.

Taking into account the silicon area including overhead in the above example, a more fair comparison may be between one 512x512 array and two 256x256 arrays since these two systems may require roughly the same amount of silicon. The correct choice is now less intuitive, but using table 1 the better choice is still the 256x256 arrays.

4.1 Additional Cost and Performance Factors

Additional factors affect system design, most of which can be incorporated directly into our analysis:

- **Other Cost Metrics.** Since we mention that one of our primary concerns is with board area, it may seem to make more sense to use package sizes instead of silicon sizes. It would be possible to package each IC separately, or as part of an MCM (Multi-chip-module). Our end goal is to work toward an MCM implementation so silicon area was the best choice for us. It would be possible to do the same analysis using package sizes in which case there will be more discrete steps in cost, i.e. a 1x1 array would probably be put in the same package as a 2x2 array.

- **Anti-aliasing.** Pixel processor arrays can perform super-sampled anti-aliasing by either assigning multiple processors to the same pixel (offset by sub-pixel amounts), or by processing a polygon multiple times (offsetting the entire array for successive processing cycles and accumulating the results). Both approaches can be directly accounted for in our analysis. If multiple processors are assigned to the same pixel then the array size needs to be divided by the number of samples per pixel. If a polygon is processed multiple times, the rasterization cycle time needs to be divided by the number of times the polygon is to be processed.

- **Operations Beyond Rasterization.** It would be possible to use the pixel processor arrays for operations other than rasterization, in which case the performance or utilization may be different. For example, in PixelFlow it is possible to do deferred Phong shading which would typically have a higher processor utilization. Deferred Phong shading will utilize all the processors that were covered by any of the polygons processed in the current screen region whereas rasterization handles polygons one at a time.

- **Array Size and Bandwidth.** Pixels are transferred over the composition bus in blocks related to the processor array size with each transfer having an associated overhead. For small arrays the transfer block size will be small and the percent overhead on the bus will increase. Additionally, if the width of the composition bus is scaled with the size of the array, the overall bandwidth will be reduced. After completing the analysis presented in this paper it is important to check the resulting design to ensure you have met the bandwidth requirement on the composition bus.

5 Raster Cycles vs. Silicon Area

To compare the efficiency of the different array sizes, we can plot system speed, measured in raster cycles, versus the silicon area of the system. Figure 2 shows the number of rasterization cycles vs. silicon area required to compute the image shown in figure 8a. In this plot, only the silicon area used by the pixel processors in the renderer was considered, i.e., no overhead. Each line in the plot represents a system composed of renderers with different array sizes. The upper left end of each line is the performance and area of a system using one array of the given size, the bottom right end of each line is the performance and area of a system using a thousand arrays.

Figure 2 shows that the most efficient array size is 1x1. This is explained by considering pixel processor utilization, which can be alternately viewed as how well different array sizes fit to an arbitrary polygon shape. Since the screen has been subdivided into regions corresponding to the size of the pixel processor array, a polygon must be processed once for each subdivision it even partly covers. In the case where the subdivisions are 1x1, a polygon processed in a subdivision will utilize all processors in that subdivision since there is only one processor and if the polygon covers that subdivision it will cover that processor. With a 2x2 array, it is possible that a polygon will cover 1, 2, 3 or 4 pixels in that subdivision, leaving 3, 2, 1 or 0 processors, respectively, unutilized. This lower pixel processor utilization causes the curve for 2x2 arrays to be shifted to the right of 1x1 arrays. As the arrays size increases further, the percent of non-utilized pixel processors increases and causes the curves to continue shifting to the right.

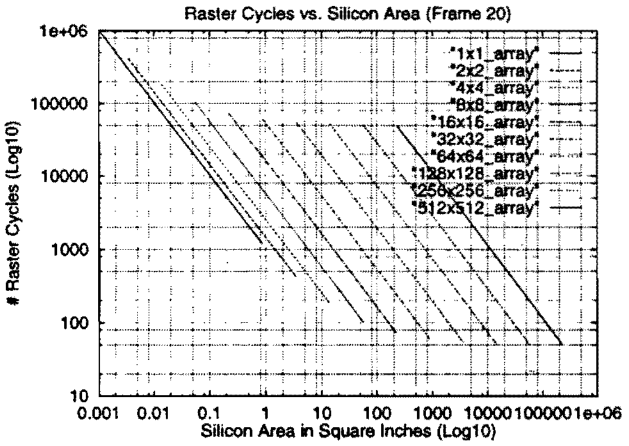


Figure 2) Raster Cycles vs. Silicon Area, No Overhead

5.1 Geometry Engine Sizing

The design of PixelFlow requires that each array be paired with a geometry engine. Therefore, with each additional pixel processor array, an additional geometry engine needs to be replicated. Since larger arrays can process more polygons per second (due to the smaller overlap factor) they require larger geometry engines capable of transforming more polygons. To empirically estimate the silicon area required for a given level of geometry engine performance, we conducted a brief survey of present-day microprocessors and related their performance to an i860, for which geometry processing performance is well known.

Our analysis is based on using a microprocessor as the geometry engine. Microprocessors include overhead which is not necessary for geometry processing. This has driven some designers to develop custom geometry processors which make more efficient use of silicon. It would be possible to make an estimate of the silicon area needed for a custom geometry processor versus performance and use that estimate in place of equation (1) shown below.

If the geometry engine were designed after selecting a microprocessor, only discreet performance levels could be achieved corresponding to multiples of the microprocessor's performance. We chose to develop a more theoretical model of the geometry engine size to avoid making our calculations dependent on a specific microprocessor. In our sizing estimate, we used the starting point that a 50MHz i860XP can transform 150,000 polygons per second [MOLN92]. Comparing the Specfp92 performance of the i860XP, which is several years old, to the newer HP-PA7200 and MIPS R10000 microprocessors, we estimate that the PA7200 can transform 483,000 polygons/sec and the R10000 can transform 1,161,000 polygons/sec. The die area of the PA7200 is 0.3255 in^2 and the R10000 is 0.4603 in^2. Higher geometry engine performance can be achieved using two processors, with twice the silicon area and twice the performance (minus some percentage, to account for multiprocessing overhead, 5% in our analysis). This provides another set of performance-area data points for each MPU.

We can use a least squares fit to obtain a simple linear performance-area model for the geometry engine, based on current microprocessor characteristics. This, in turn, can be used as the cost overhead model for the system, and can be combined with Table 1 (or similar) data to give an analytic performance model of the system. The least squares fit (which includes the microprocessor, memory for each microprocessor and 1 controller chip) generates a line with the formula:

$$G(A) = (7.4645 \times 10^{-7})P(A) + 1.3673$$
$$R(A) = 0.000871A^2 + G(A) \qquad (1)$$

$G(A)$	= Area of geometry engine for array of size AxA
$P(A)$	= Performance of array of size AxA in Polygons/sec
$R(A)$	= Area of renderer for array size of AxA (0.000871 is the area of 1 pixel processor)

Table 2 shows the geometry engine (G.E.) size, the pixel processor array size and the total renderer size for a renderer built for each of the different array sizes.

Subdivision. Size (*AxA*)	Array Perform. (Polys/S)	G.E. Size (Sq. in)	Processor Array Area (Sq. in.)	Total Area (Sq. in.)
1x1	41,828	1.399	0.00087	1.400
2x2	119,678	1.457	0.00348	1.460
4x4	274,460	1.572	0.01394	1.586
8x8	485,510	1.730	0.05574	1.786
16x16	684,770	1.878	0.22298	2.101
32x32	828,493	1.986	0.89190	2.878
64x64	914,592	2.050	3.56762	5.618
128x128	961,957	2.085	14.27047	16.355
256x256	985,952	2.103	57.08186	59.185
512x512	1,000,000	2.114	228.32742	230.441

Table 2) Geometry Engine, Array and Renderer Sizes

When the silicon area of the geometry engine is included with the area of the processor array, there is a change of the ordering of the curves. Figure 3 shows the new plot of raster cycles vs. silicon area, and shows that the optimal array size for the image in figure 8a is 16x16, since it is the lowest curve (lowest time for a given area). In this specific case, selection of either the R10000 or HP-PA7200 would not change the optimal array size. Choosing the HP-PA7200 would cause incremental shifts between the 4x4 and 8x8, and 128x128 and 256x256 curves, but the shifts are not large enough to change the final answer.

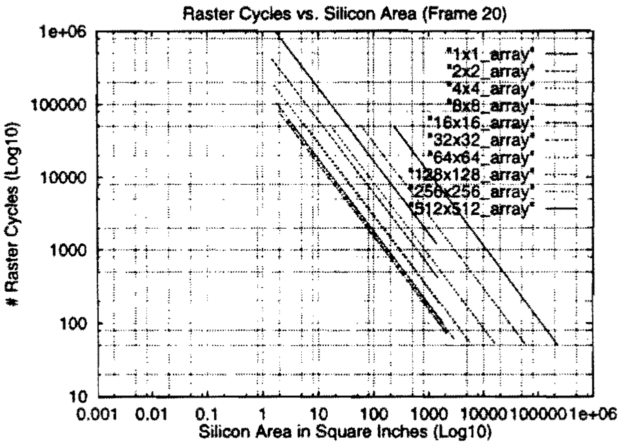


Figure 3) Raster Cycles vs. Silicon Area, Overhead included

5.2 Performance vs. Silicon Area

To determine why the 16x16 array is optimal we need to look at how performance is related to silicon area. Performance is inversely proportional to the number of raster cycles to generate the image. Figure 4 shows a plot of performance vs. silicon area for different size arrays, including geometry engine area, based on the Table 2 data.

In figure 4, the first part of the curve exhibits super-linear speed-up for area increases. The cause for this effect is that the silicon area for the processor array increases at a different rate than the silicon area for the geometry engine. Initially the area of the geometry engine dominates, while the size of the array has a large effect on performance. Eventually the combination of diminishing returns for increasing the processor array size, and the silicon area of the array becoming dominant causes the curve to flatten out below linear speed up.

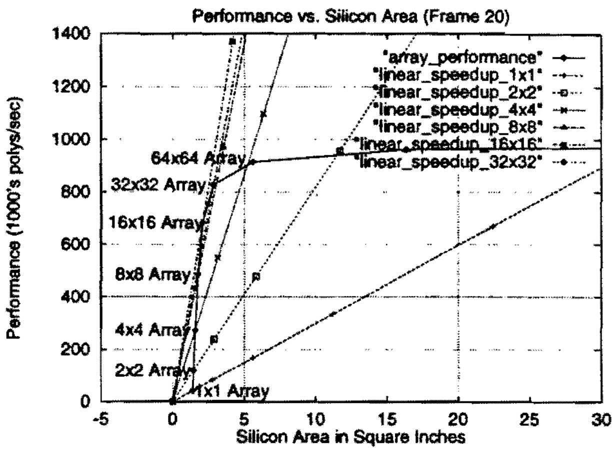


Figure 4) Performance vs. Silicon Area

In the PixelFlow architecture, linear speed-up can be achieved by adding more renderers. Linear speed-up is achieved because all the area of the renderer is replicated to create another renderer with exactly the same performance. Therefore, we can make a trade-off between when it is better to increase the size of the processor array versus when it is better to replicate the renderer. Since we can achieve linear speed-up from replicating renderers, we should pick the point on the array size curve where the slope equals the slope of the linear speed-up curve for that point. When the slope is greater than linear speed-up, we get a better increase in performance by moving to a larger array size. When the slope is less than linear speed-up, we get a better increase by replicating renderers. Table 3 shows the optimal array sizes for the images shown in figure 8.

Image	Ave. Polygon Size	Ave Bounding Box Size	Optimal Array Size
Frame20	10.43	23.91	16x16
Frame70	13.05	59.50	16x16
Headlow	9.99	21.81	16x16
ShuttleB	121.29	367.47	32x32
Room	16,620.88	23,235.70	64x64
Cube20	44.12	60.31	16x16

Table 3) Test Image Statistics and optimal array sizes

6 Derivation of Performance Area Curves and Optimal Array

To facilitate determining the optimal array size for an image without running the simulations, we developed an analytical method to predict the performance of different size arrays given only the average bounding box size of polygons to be rendered. We then developed a method to use this prediction to accurately calculate the optimal array size required.

The bounding box of a polygon will be a rectangle with an arbitrary aspect ratio. Bounding boxes with the same area, but different aspect ratios will overlap differing numbers of screen regions, however the variance is fairly small. Therefore we simplified the problem to consider only square bounding boxes. To determine how many regions are overlapped we must examine which regions are overlapped for each possible position of a bounding box. The position of the bounding box is defined by the placement of the lower left corner in a reference region. For example, if we consider a region size of 2x2 pixels, and a bounding box which is 7 pixels on a side, then there are 4 positions for the bounding box in the reference region. For each position, the number of regions covered by the bounding box is 16 (a square of 4 regions up and 4 regions to the right). If we consider a bounding box which is 8 pixels on a side and a region size of 2x2, then there is one position where 16 regions are overlapped, two positions where 20 regions are overlapped, and 1 position where 25 regions are overlapped. This case is shown in figure 5. (The 2x2 reference region is dark shaded, the polygon is the bold square, regions *not* covered are lightly shaded).

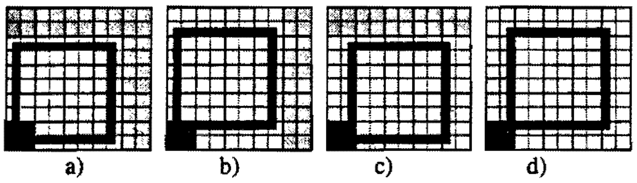


Figure 5) a) polygon placement covering 16 regions;
b & c) polygon placement covering 20 regions;
d) polygon placement covering 25 regions.

A generalization is that for each region size, there are 3 subregions. In subregion I, there are i^2 overlapping regions, in subregion II, $i^2 + i$ regions overlap, and in subregion III, $(i+1)^2$ regions overlap. The size of subregions I, II and III depends on the relationship between the size of the screen regions and the bounding box. There will always be a subregion I, but not always a subregion II and III. The value of i is also dependent on the difference in size between the bounding box and the region size. Let A represent the width and height of the region; for example when A=2, the region size, and therefore the pixel processor array size, is 2x2 and contains 4 pixel processors. Let N represent the number of pixels in the bounding box, i.e. if N=49 then the bounding box is 7x7 pixels, then i can be found by using:

$$i = \left\lceil \frac{\sqrt{N}}{A} \right\rceil \qquad (2)$$

where the $\lceil\ \rceil$ brackets represent the ceiling function, i.e. round up to the nearest integer. The size of the different regions are

Size of subregion I = $\alpha = ((i \times A) - \sqrt{N} + 1)^2$ (3)

Size of subregion III = $\gamma = (\sqrt{N} - ((i-1) \times A - 1)^2$ (4)

Size of subregion II = $\beta = A^2 - \alpha - \gamma$ (5)

Using these relations the formula for determining the average number of regions covered by a bounding box of size N, is:

$$C(A) = \frac{\left[(i^2 \times \alpha) + ((i^2 + i) \times \beta) + ((i+1)^2 \times \gamma)\right]}{A^2} \qquad (6)$$

If we set N equal to the average bounding box size for an image, then equation (6) gives the average number of arrays covered by the bounding boxes in that image. Equation (7) uses equation (6) to calculate the performance of an array, where performance is polygons/second (since there is one polygon for each bounding box).

$$P(A) = \frac{1}{C(A)} \times T \qquad (7)$$

Where T is the speed of the array in number of rasterizer cycles per second. Our simulations showed that equation (6) and (7) agree closely with the experimental results for the image in figure 8a, with the experimental and analytical results varying no more than 7.8%. In order to see how performance varies with silicon area, we need to find the relationship between array size and silicon area since equation (6) is in terms of array size. In section 5.1 we developed equation (1), which is a formula for determining renderer size given the array size and array performance. Using equations (1) and (7) we can plot the calculated performance versus silicon area. Figure 6 shows the plot of this function with the plot of the experimental results from section 5.

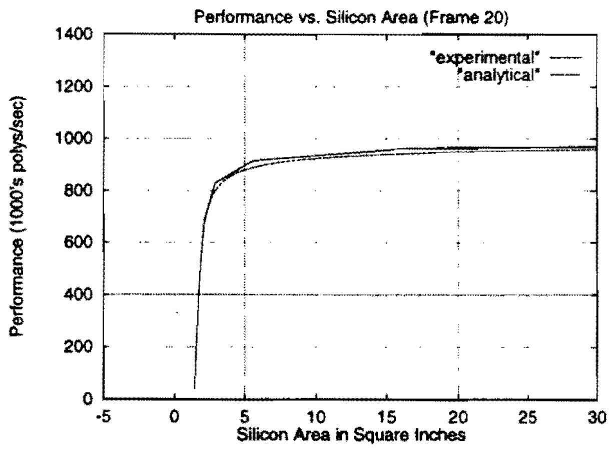


Figure 6) Array Performance

In section 5.2 we saw that the optimal array size is where the slope of the performance curve equals the slope of the linear speed-up. With equations (1) and (7) we can get performance P in terms of array size A, and we can get renderer size R in terms of array size A. To find the optimal array size we need to find the equality between $P'(R)$ and the slope of the linear speed-up curve, $L(R)$. The slope of the linear speed-up curve is simply:

$$L(R) = \frac{P(R)}{R} = \frac{P(A)}{R(A)} \qquad (8)$$

and since

$$P(R) = P(A(R)) \tag{9}$$

we can determine $P'(R)$ using:

$$P'(R) = P'(A)A'(R) = \frac{P'(A)}{R'(A)} \tag{10}$$

The equations for $P(A)$ and $R(A)$ were determined previously, i.e. equations (7) and (1) respectively. So all that remains is to find their derivatives. To calculate the derivatives it is necessary to approximate equation (2) to remove the ceiling function. The approximation used is:

$$i = \frac{\sqrt{N}}{A} + 0.5$$

The derivatives are:

$$P'(A) = \frac{A^2(hA^2 + jA + k)}{aA^6 + bA^5 + cA^4 + dA^3 + eA^2 + fA + g} \tag{11}$$

Where:

$$\begin{aligned}
h &= 1.5\sqrt{N} - 2 \\
j &= 2N = 4\sqrt{N} \\
k &= 2\sqrt{N} \\
a &= 1.5575 \\
b &= 3.75\sqrt{N} \\
c &= 4.75N - 11\sqrt{N} + 4 \\
d &= 3N\sqrt{N} - 10N + 13\sqrt{N} \\
e &= N^2 - 4N\sqrt{N} - 8\sqrt{N} + 10N \\
f &= 4N\sqrt{N} - 8\sqrt{N} + 10N \\
g &= 4N
\end{aligned}$$

and,

$$R'(A) = 0.001742A + (7.4645 \times 10^{-7})P'(A) \tag{12}$$

To find where the functions expressed in equations (8) and (10) intersect we simply set them equal as follows:

$$\begin{aligned}
L(R) &= P'(R) \\
\frac{P(A)}{R(A)} &= \frac{P'(A)}{R'(A)} \\
\frac{P(A)}{R(A)} - \frac{P'(A)}{R'(A)} &= 0
\end{aligned} \tag{13}$$

A plot of $L(R)$ and $P'(R)$ is shown in figure 7 using the value of the average bounding box size from the image in figure 8a of N=23.9. Figure 7 shows that the analytical solution is very close to the experimental solution.

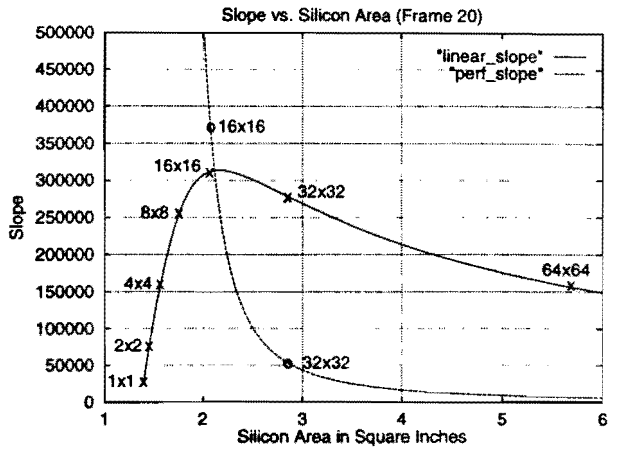


Figure 7) Optimal Array Size

7 General Purpose Massively Parallel Computers

While the above discussion focused on a custom designed massively parallel architecture, the results can also be used to aid in the design of rasterization algorithms which run on general purpose massively parallel computers. In the case of general purpose machines, the number of processors available is fixed, but often times the configuration of those processors is controllable. In equation (13) from section 6.0, the optimum size pixel processor array is determined. However, for general purpose computers this equation is not appropriate. In the above discussion, the goal was to develop the most efficient configuration, and with special purpose designs the designer has control over both performance and silicon area. However, for general purpose computers the number of processors is fixed, and the amount of silicon area is fixed. The only control the designer/programmer has is to effect performance by optimizing code and choosing the optimum processor configuration.

For general purpose massively parallel computers, equation (7) can still be used to predict performance dependent on pixel processor array size. Using this equation we can develop a formula for predicting the performance if the number of pixel processors is fixed. If the total number of pixel processors is X, we simply divide the predicted performance by the number of AxA arrays which can be formed with X processors. The equation is:

$$P_{GP}(A) = \frac{\frac{1}{C(A)} \times T}{\frac{X}{A}} \quad , \text{ For } A \leq X \tag{14}$$

This equation can be used as a component in the analysis of algorithms developed for general purpose machines. A complete analysis will depend on other factors such as interprocessor communication costs.

8 Conclusions

We have shown that the optimal size array for a graphics computer which uses 2-D arrays of pixel processors for rasterization can be analytically determined. In the case of a special purpose computer design, the optimal array size is dependent on the average size of polygons being processed, and on the overhead associated with each array. The optimum size occurs when the slope of the performance curve equals the slope of the linear speed-up curve, since linear speed-up can be attained by replicating processor arrays. We have shown that the performance curve can be accurately predicted with equation (7). Overhead of the rasterization array plays an important role in determining the optimum array size, and in the extreme case where there is no overhead, the optimal array size is 1x1 and is independent of polygon size.

In the case of general purpose multiprocessor computers, the optimal array size is also dependent on the average size of polygons being processed, but is independent of array overhead since the overhead is fixed. Equation (7) can be modified to become equation (14), and used to determine the optimal processor configuration given a fixed number of pixel processors.

9 Future Work

We plan on implementing the PixelFlow architecture with the array sizes optimized for our application[2].

In related work, we are investigating caching strategies for rendering systems based on multiprocessor systems. These systems are not configured as arrays of pixel processors.

10 Acknowledgments

We would like to thank the Eurographics Hardware Workshop reviewers for their constructive comments, many of which we included in the final draft of this paper.

11 References

[ELLS94] David Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers", IEEE Computer Graphics and Applications, July 1994, pp. 33-40.

[FUCH82] Henry Fuchs, John Poulton, Alan Paeth, Alan Bell, "Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System", Proceedings of 1992 Conference on Advanced Research in VLSI, M.I.T., pp. 137-146.

[FUCH89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", Proceeding of SIGGRAPH 1989, pp. 79-88.

[MOLN92] Steven Molnar, John Eyles, John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", Proceeding of SIGGRAPH 1992, pp. 231-240.

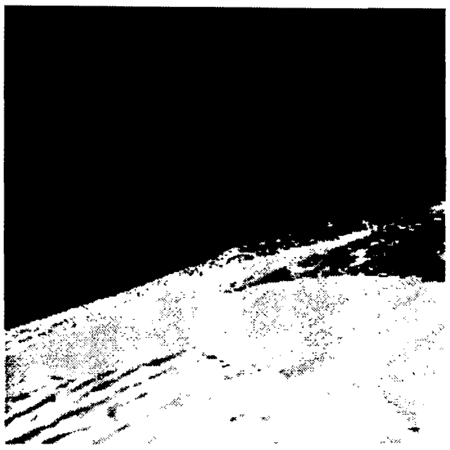
[MOLN94] Steven Molnar, Michael Cox, David Ellsworth and Henry Fuchs, "A Sorting Classification of Parallel Rendering", IEEE Computer Graphics and Applications, July 1994, pp. 23-32.

[ROBL88] D.R. Roble, "A Load Balanced Parallel Scanline Z-buffer Algorithm for the iPSC Hypercube", Proceedings First International Conference Pixim 88, Editions Hermes, Paris, 1988, pp. 177-192.

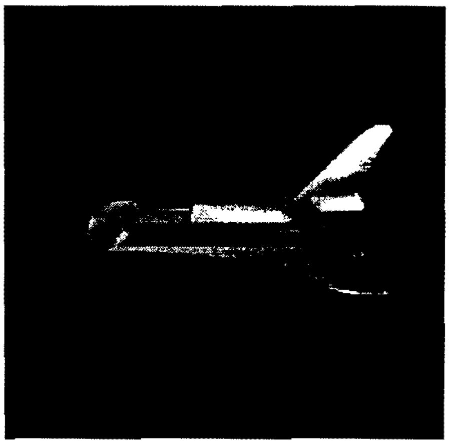
[WHIT94] Scott Whitman, "Dynamic Load Balancing for Parallel Polygon Rendering", IEEE Computer Graphics and Applications, July 1994, pp. 41-48.

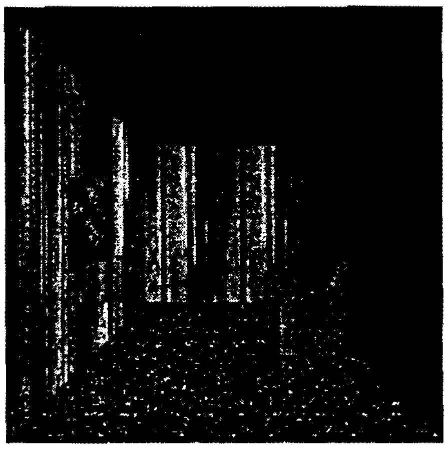
[2] The PixelFlow architecture is covered by patents and requires a license agreement for use.

a) frame 20


b) frame70

c) headlow


d) shuttleB


e) room_w_chair

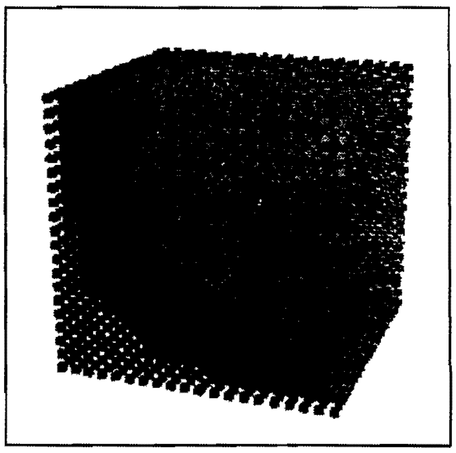

f) cube20

Figure 8) a) & b) were generated from digital terrain and elevation data. c) & d) are from the Picture Level Benchmark developed by the Graphics Performance Characterization Committee of the National Computer Graphics Association. e) & f) were generated by the authors.