A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm

Michael Kelley Stephanie Winner Kirk Gould

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

1. ABSTRACT

A hardware accelerator for 3D rendering, based on a modified scanline algorithm, is presented. The accelerator renders multiple scanlines in parallel with high efficiency, and is optimized for integration into systems that support high speed data streams (such as video). The architecture has a very high performance/cost ratio, but maintains a low entry cost and a high degree of scalability — key issues for incorporation in personal computers. The performance of both the general algorithm and the prototype implementation is analyzed.

CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles - parallel; **I.3.1** [Computer Graphics]: Hardware Architecture - raster display devices; **I.3.3** [Computer Graphics]: Picture/Image Generation - display algorithms; **I.3.7** [Computer Graphics]: Three-Dimensional Graphics and Realism - visible surface algorithms

General Terms: algorithms, architecture, parallel

Additional Key Words and Phrases: scanline, data sharing, low bandwidth, low cost

2. INTRODUCTION

Advances in CPU technology have given personal computers the compute power necessary to run 3D applications. However, interactive applications require faster rendering than can be achieved by the CPU alone — today's 3D workstations also include sophisticated rendering hardware [1, 2, 11, 14]. The goal of our project is to provide similar functionality in a personal computer.

We began by determining what differences, if any, there were between the requirements of 3D acceleration in a workstation and a personal computer. We identified some key features required for a personal computer:

- *Low cost.* The accelerator must have a very low entry cost.
- *Modularity.* It should be possible to add the accelerator to a system as an option.
- *High performance.* An intuitive, direct manipulation user interface requires a high frame rate and high speed hit testing.
- *Visual realism.* Although schematic representations such as wireframe are useful, naive users are more comfortable with "realistic" shaded images with shadows.

Existing workstations meet some of these requirements, but not all. High end workstations deliver high performance shaded rendering, but prices are stratospheric by personal computer standards. Recent entry level workstations have reduced this cost, but at the expense of lower speed, particularly for high quality rendering modes. And even these systems are many times the entry price of a typical personal computer. Finally, because the graphics accelerator in these systems is tightly coupled to the system frame buffer, providing the accelerator as an option, e.g. a plug-in card, is difficult — typically a local frame buffer must be added to the accelerator, increasing cost.

The architecture described in this paper has been optimized for a personal computer graphics system. It uses a modified scanline rendering algorithm to greatly reduce cost by combining the rasterization hardware and scanline RAM on chip. The architecture emphasizes shaded rendering, and new rendering features (e.g. CSG, antialiasing) can be added with little cost increase. The design is highly scalable because performance and functionality are limited by ASIC complexity (which is rapidly increasing), and because the architecture efficiently supports parallelism — a parallel implementation is described which can rasterize 880K triangles per second.

This paper primarily discusses rasterization, as standard techniques are used for transformation, clipping and shading [e.g. 4, 6]. We plan to use general purpose RISC or DSP devices to accelerate these tasks [2, 11].

3. BACKGROUND AND PREVIOUS WORK

3.1 Screen Z-Buffer Algorithm

The screen Z-buffer algorithm was one of the first shaded hidden surface removal algorithms, and was used by both software and hardware implementations [3, 8]. Its advantage is simplicity — each object to be rendered can be transformed and rasterized independently, allowing an arbitrarily large number of objects to be rendered (given enough patience). Hardware implementations of this algorithm have been very successful; in fact, it is the method used in virtually all 3D workstations [1, 2, 11].

However, the screen Z-buffer algorithm has disadvantages which make it less suitable for personal computers. The most obvious of these is memory use — the algorithm requires storing a Z value for every pixel on the screen. Storing a 24 bit Z for a 1Kx1K screen uses 3M bytes of memory, an appreciable amount in an entry level computer. More importantly, this memory must be very high performance — pixel shading speed is directly proportional to the sustained bandwidth to the RAM. For example, rendering one million 100 pixel triangles/second requires[1]:

1M tri/s x 100 pixel/tri = 100M pixel/s
100M pixel/s x 3 Z bytes/pixel x 1.5 = 450 MB/s

In practice, these very high bandwidths are achieved with wide, fast RAM [1], often coupled with sophisticated caching and prefetching. Although it's reasonable to add these costs to a dedicated workstation, they make it impractical to add high performance graphics to a low cost personal computer.

More abstractly, these disadvantages of the screen Z-buffer algorithm are caused because it stores the state of the rendering calculation for each pixel individually; effectively, the state information necessary to render one pixel is replicated for every pixel on the screen. As a result, rendering algorithms which require additional information per pixel (e.g. CSG, shadows, antialiasing) are expensive to implement; for example, the system described in [10] has over 150 bits per pixel.

3.2 Scanline Z-buffer algorithm

To find a more cost effective implementation of these rendering algorithms, we adopted the same solution used by many software renderers, the scanline Z-buffer algorithm [12]. In comparison to the screen Z-buffer algorithm, where the state information necessary for rendering a pixel is stored for every pixel on the screen, the scanline algorithm presorts the object database in screen space [15], and renders each scanline individually — only one scanline of pixel state information is kept. The difference between the two methods is substantial: for a 1Kx1K screen, the screen Z-buffer algorithm uses 3M bytes for Z, whereas the scanline algorithm uses only 3K bytes [9].

Additional information on scanline algorithms can be found in [6].

4. OVERVIEW OF THE PROTOTYPE

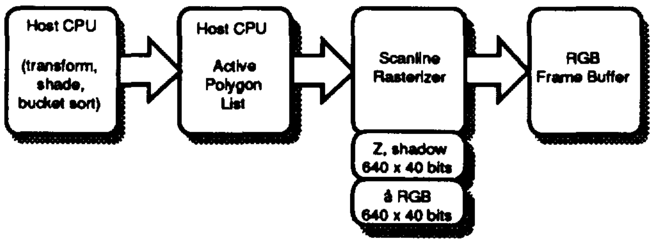

Figure 1

The prototype (shown above) is based on a Macintosh® Quadra™ personal computer; the scanline rasterizer is implemented as a Processor Direct expansion card. The host 68040 is used for transformation, shading, and active list maintenance; faster prototypes, using more powerful floating point engines, are also being developed.

The scanline rasterizer performs Gouraud shading, hidden surface removal via a 32 bit Z-buffer, shadow volumes [7] and alpha blending (with 10 bits of accuracy). The rasterizer is implemented as two 0.8μm ICs (Figure 10, at the end of the paper), designed with silicon compilation tools; the chips operate at 40MHz. The first chip intersects polygons transferred from the active polygon list with the current scanline, generating a series of horizontal spans. The second chip rasterizes the resulting spans, doing hidden surface removal, shadow plane tests and alpha blending. Multiple chip sets can be connected in parallel with virtually no glue logic, providing very high performance with low chip count (sections 7 and 8 discuss parallel rasterization).

The following pseudo-code shows the basic rendering algorithm of the prototype:

```
RenderFrame()
{
  /* First pass: transform, shade, sort */
  foreach (Poly)
  {
    Transform (Poly);
    if ( ! Cull (Poly))
    {
      Shade (Poly);
      BucketSort (Poly);
    }
  }

  /* Second pass: rasterize */
  foreach (Scanline)
  {
    AddNewPolys (BucketedPolys[Scanline]);
    Rasterize (ActiveList, Scanline);
    RemoveFinishedPolys (ActiveList, Scanline);
  }
}
```

Rendering begins when the host CPU traverses the 3D database, generating transformed, projected, clipped and shaded polygons. The polygons are then bucket sorted by the number of the first scanline on which they first become active. Once the main database traversal is complete, the host traverses the bucket sorted list in screen Y order, maintaining an active polygon list which is transferred into the rasterizer to drive rendering.

5. DISADVANTAGES OF THE SCANLINE ALGORITHM

5.1 Database sorting

Computationally, the penalty for the two pass algorithm is relatively minor [5]. Because the second pass is driven by the bucket sorted list, only a single traversal of the database hierarchy is required. The computational overhead of the bucket sort itself is low compared with transformation, clipping and shading, and is principally a problem of efficient memory management.

A more substantial penalty is the memory used to store the bucket sorted polygons; in the prototype, 40 bytes are required for each transformed, shaded triangle. However, this is offset by the other memory savings of the scanline algorithm; for

[1] This assumes 50% of pixels are visible; section 8.4 discusses this in more detail.

example, the 3 megabytes saved by not using a Z-buffer would be enough to store 75K triangles (a large interactive database for a personal computer). In practice, culling and the use of more efficient primitives (e.g. quadrilaterals) further reduce memory use.

5.2 Loss of concurrent scan conversion

Because bucket sorting must complete before scan conversion begins, a simple implementation of the scanline algorithm can cause poor utilization of the graphics subsystem. For example, if the first pass (transform, clip, shade) and the second pass (rasterize) require the same length of time, the hardware will never exceed 50% utilization. This compares poorly to a classical graphics pipeline model, where a well-balanced pipeline can achieve 100% utilization.

One solution to this problem is to double buffer the bucket sorted polygon list; this allows the two passes of the algorithm to run simultaneously, permitting 100% efficiency. Unfortunately this does *not* reduce latency; a direct manipulation interface which requires user feedback every frame will have to flush both buffers, negating the advantage. However, for tasks where latency is less critical (e.g. rotating an object for viewing), double buffering is effective.

A more general solution to this problem is to shorten the time required for first pass processing, which provides more time for rasterization and increases utilization. For polygons, postponing clipping, normal normalization and lighting to the second pass reduces first pass computation by approximately 50% [1][2]; the savings increase as the lighting model becomes more complex. Curved surfaces can be transformed without tesselation, postponing virtually all computation until the object becomes active during the second pass, at which point the object can be decomposed into renderable primitives. This technique also reduces memory use for the bucket sort.

5.3 Poor wireframe performance

Although scanline algorithms are efficient for shaded primitives, there is a substantial performance loss for wireframe rendering — in the prototype, wireframe performance will be roughly equal to shaded performance, as opposed to the 5X - 10X ratio commonly encountered in workstations. Because of our emphasis on shaded rendering, this was considered acceptable.

6. ADVANTAGES OF THE SCANLINE ALGORITHM

6.1 On-chip rasterization

A key implementation advantage of the scanline algorithm is that the Z and αRGB memory used for pixel rendering can be placed on the same chip as the rasterization hardware. For example, the prototype stores 32 bits for Z, 8 for shadows, and 10 for each of αRGB, a total of 80 bits per pixel — a 640 pixel scanline requires 51K bits of RAM. In the prototype, this scanline RAM uses only 25% of the rasterizer chip's area.

By placing the rasterization hardware on-chip with the RAM, very high rasterization speeds can be achieved without high off-chip bandwidth — the prototype rasterizer runs at 40M alpha-blended Z-buffered pixels per second, using 720M bytes/s of on-chip RAM bandwidth. Because chip I/O is not in the critical path, speed is limited only by core logic and RAM performance, allowing rapid performance increase as chip technology advances.

6.2 Raster output and modularity

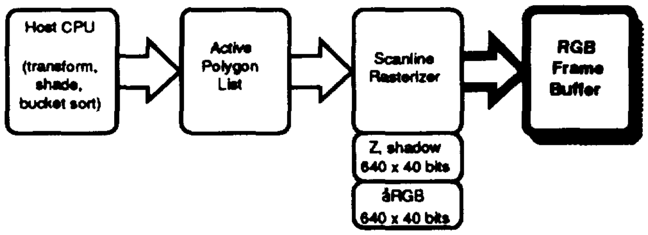

Figure 2

As scanlines complete, they are transferred from the rasterizer into the frame buffer (Figure 2). To the system, these transfers appear as a video-like high speed raster — this similarity allows easy integration into future video-capable personal computers.

Because the dataflow from rasterizer to frame buffer is unidirectional, the rasterizer does not require a tightly coupled low latency frame buffer interface. This makes it possible to add a scanline rendering accelerator to a system as an option (i.e. a card) without the cost of a local frame buffer.

6.3 High resolution rendering and antialiasing

For an NxN image, the memory requirements of the scanline Z-buffer algorithm are proportional to N, whereas the memory requirements of the screen algorithm are proportional to N^2.[3] This allows scanline algorithms to render large (e.g. 8Kx8K) images with high efficiency.

Unfortunately, when the scanline buffer is stored on-chip, scanline width is limited by RAM size — for example, the prototype has a maximum width of 640 pixels. This limitation can be removed by rendering wider scanlines in segments. For each scanline, the active object list is traversed multiple times, the first time rasterizing pixels 0 to 639, the second time 640 to 1279, etc. The resulting segments abut left to right, and are indistinguishable from the result had the entire scanline been rendered in a single pass. Only a single traversal of the main database is required, avoiding the redundant transformation and clipping which are caused by a tiling algorithm.

Because the scanline algorithm generates the image in scanline order, traditional super-sampled antialiasing techniques can be implemented with an accumulation buffer of only a few scanlines; in particular, hosts with video support may already include such hardware for window re-sizing and filtering. More advanced antialiasing algorithms (e.g. A-buffer) could be

[2] From Akeley's paper, 23 of 46.5 Mflops = 50%. These figures are for a single light source at infinity.

[3] Screen algorithms can render high resolution images by tiling, but with a substantial performance penalty for performing multiple database traversals.

implemented; because only one scanline is rendered at a time, the limiting factor is complexity, rather than RAM.

7. MODIFYING THE SCANLINE ALGORITHM FOR PARALLELISM

Thus far has we have discussed the pros and cons of the traditional scanline algorithm. However, a goal of the project was to develop an algorithm with a highly scalable implementation; in particular, our goal was to devise an algorithm where performance could be increased by rendering multiple scanlines in parallel.

7.1 Previous solutions: Screen partitioning

One solution to this problem [13] is to partition the screen into a number of relatively large contiguous regions, and render each region with an unmodified scanline algorithm. When complete, the regions are tiled together to produce a complete frame. While simple to implement, this divide-and-conquer solution suffers from poor load balancing — an image whose complexity is unevenly distributed across the screen will not be efficiently parallelized. Adaptive partitioning of the screen can improve load balancing, but the setup overhead for a region limits efficiency if fine subdivision of the screen is attempted. Also, this algorithm is expensive: active list hardware, memory, and rasterization hardware are all replicated.

Instead, we wanted a solution that renders multiple *adjacent* scanlines simultaneously. This largely solves the load balancing problem, as adjacent scanlines usually have similar complexity. Also, it has the potential to exploit scanline coherence by maintaining a single active polygon list which is shared by multiple rasterizers, each working on a separate scanline.

7.2 Scanline coherence vs parallelism

Unfortunately, the typical implementation of the scanline algorithm makes this difficult. To increase performance, scanline renderers exploit coherence by converting the interpolation calculation of the edge parameters at the current scanline into a forward differencing calculation in Y [6, 15]:

Y_{top}, Y_{bottom} = top and bottom of edge

$P_{top[n]}, P_{bottom[n]}$ = parameter *n* at top, bottom

H_{inv} = inverse of height of edge

$\Delta P_{[n]}$ = change/scanline of parameter [n]

$P_{[n]}$ = interpolated value of parameter [n]

$$H_{inv} = 1 / (Y_{bottom} - Y_{top})$$
$$\Delta P_{[n]} = H_{inv} \times (P_{bottom[n]} - P_{top[n]})$$
$$P_{0[n]} = P_{top[n]}$$
$$P_{i+1[n]} = P_{i[n]} + \Delta P_{[n]}$$

(The initialization function $P_{0[n]}$ has been simplified for clarity; the actual subpixel accurate function is more complex.) Rendering multiple scanlines in parallel requires parallelizing this forward differencing calculation, which is possible but awkward; in particular, the calculation of $P_{0[n]}$ and the end-of-edge test must be replicated for every interpolator.

7.3 Direct evaluation vs forward differencing

A different approach to the problem is to abandon the forward differencing solution and directly evaluate the interception calculation of the object edge and the scanline:

w_y = interpolation weight at Y

$$w_y = (Y - Y_{top}) / (Y_{bottom} - Y_{top})$$
$$P_{y[n]} = (P_{top[n]} \times (1-w_y)) + (P_{bottom[n]} \times w_y)$$

Direct evaluation is computationally more expensive than forward differencing (for each scanline, an additional divide and two multiplies per parameter). However, because the cost of computation on an ASIC is declining so rapidly, we felt comfortable adopting a computationally intensive solution, as it provides advantages in the following, more problematic areas.

7.3.1 Simplified setup

The most obvious advantage of direct evaluation is that it greatly simplifies the setup procedure for an active object — the calculation of H_{inv}, $\Delta P_{[n]}$ and $P_{0[n]}$[4] is completely avoided. With direct evaluation, the object is simply inserted into the active object list; no other processing is necessary.

7.3.2 Reduced active list memory and bandwidth

To forward difference a parameter $P_{[n]}$ of N bits requires storage of $\Delta P_{[n]}$, typically 2N bits[5], and the current parameter value $P_{i[n]}$, also 2N bits, for a total of 4N bits. Using direct evaluation, only $P_{top[n]}$ and $P_{bottom[n]}$ are stored, a total of 2N bits, providing a 50% memory saving.

More important, however, is the reduction of memory bandwidth. To directly evaluate a parameter, $P_{top[n]}$ and $P_{bottom[n]}$ are read for a total of 2N bits of bandwidth. For forward differencing, $\Delta P_{[n]}$ and $P_{i[n]}$ are read, and then $P_{i+1[n]}$ is written back, a total of 6N bits or three times the bandwidth of direct evaluation. As shown later, active list bandwidth is directly proportional to rasterization speed; the reduction from direct evaluation allows much higher performance.

Also, because direct evaluation does not require writing back $P_{i+1[n]}$, the dataflow is unidirectional, substantially simplifying system design. For example, high latency bursting memory such as VRAM can be used for active polygon storage, reducing cost.

7.3.3 Data sharing for parallelism

The most important advantage of direct evaluation is that the data in the active object list no longer reflects the state of any particular scanline — the same data can be used to render multiple scanlines in parallel. In fact, the prototype is designed to transfer data from the active object list to multiple rasterizers simultaneously:

[4] Direct evaluation is intrinsically sub-pixel accurate, so sub-pixel alignment does not have to be specially treated.

[5] The extra N bits are fractional guard bits. The number of guard bits is actually $\log_2$(max number of iterations), i.e. forward differencing an 8 bit value across an 8K screen requires 15 guard bits.

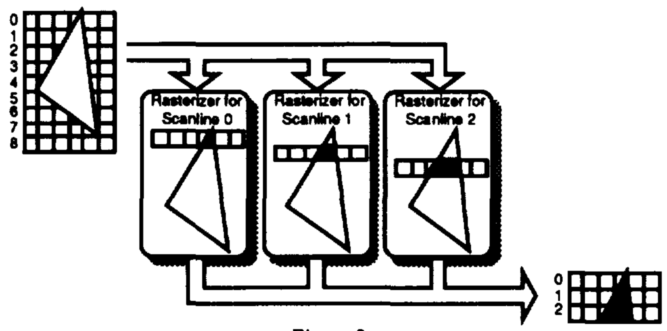
0
1
2
3
4
5
6
7
8
Rasterizer for Scanline 0
Rasterizer for Scanline 1
Rasterizer for Scanline 2
0
1
2

Figure 3

In this example, the triangle description is transferred into all three rasterizers simultaneously. Each rasterizer intercepts the triangle with its target scanline, and renders the resulting horizontal span. All three rasterizers then output their scanlines, generating a three scanline strip of the final image.

Because the data is shared by all rasterizers, efficiency is high: bandwidth is wasted only when an object does not intersect all of the scanlines being rendered. In this example, the triangle description is transferred only twice to render scanlines 0 to 5 (each transfer renders three scanlines), yielding an ideal 6/2 = 3X increase in rasterization speed over a single rasterizer.[6] However, when rendering scanlines 6 to 8, the triangle intersects only two of the three rasterizers, leaving one idle. Therefore, a frame that would have required 8 triangle transfers in a single rasterizer design requires 3 transfers, for a speed increase of 8/3 = 2.7X, and a rasterizer utilization efficiency of (8/3) x (1/3) = 8/9 = 89%.

In practice, rather than allowing the decrease in utilization to reduce rasterization speed, the input data bandwidth is increased. Because the rasterizers can discard a triangle that doesn't intersect the scanline much faster than rendering it, increasing input bandwidth to compensate for the unnecessary triangle transfers permits all rasterizers to run at full speed (an input FIFO on the rasterizer is used to smooth the dataflow). Therefore, the efficiency of a given number of parallel scanline rasterizers can be characterized by the increase in input bandwidth necessary to keep all rasterizers fully utilized:

N_{par} = number of parallel scanlines
H_{avg} = average height of an active object
B_{par} = bandwidth increase over a single rasterizer

$$B_{par} = (N_{par} - 1) / H_{avg}$$

For the example above, with three parallel rasterizers and an average object height of eight scanlines:

$$B_{par} = (3 - 1) / 8 = 25\% \text{ increase}$$

In other words, a 25% increase in active list bandwidth over a single rasterizer would be enough to keep three rasterizers fully utilized.

[6] For this example, we assume performance is limited by active list bandwidth. More complete performance analysis is in the following section.

8. PERFORMANCE ANALYSIS

8.1 Rasterization performance

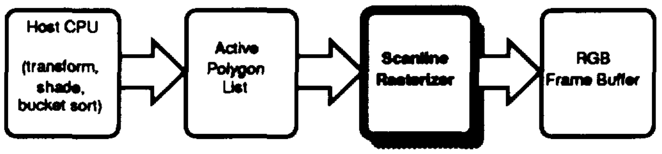

Figure 4

Rasterization performance can be characterized by two values:

P_{int} = primitive-scanline intersections/s
P_{raster} = rasterization speed (pixels/s)

The prototype hardware intersects a triangle with the current scanline in 14 clocks, yielding a P_{int} of:

$$P_{int} = 40 \text{ MHz} / 14 \text{ clocks/intersection} = 2.86\text{M intersections/s}$$

Rasterization speed is one pixel per clock, or:

$$P_{raster} = 40\text{M pixels/s}$$

From this, rasterization performance can be approximated by the formula:

A_{avg} = average pixels per primitive
P_{prim} = primitives/second
= min $((P_{int} / H_{avg}), (P_{raster} / A_{avg}))$
P_{pixel} = rendered pixels/second
= $A_{avg} \times P_{prim}$

For example, a 100 pixel triangle:

H_{avg} = 13 scanlines
A_{avg} = 100 pixels
P_{prim} = min (2.86M / 13, 40M / 100)
= 220K triangles/s
P_{pixel} = 100 x 220K = 22M pixels/s

8.2 Active list bandwidth

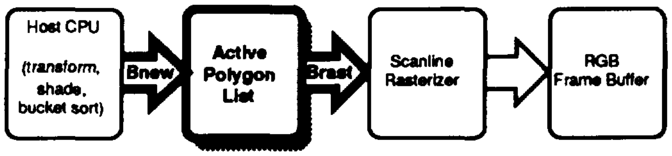

Figure 5

A key factor for system performance is B_{rast} (shown in Figure 5), the bandwidth used to transfer polygons from the active list to the rasterizer(s). For a single rasterizer this can be computed as:

S_{prim} = size of primitive
$B_{rast} = P_{prim} \times S_{prim} \times H_{avg}$

For the example given earlier, a triangle requires 40 bytes; the resulting rasterizer input bandwidth is:

$$B_{rast} = 220\text{K tri/s} \times 40 \text{ bytes} \times 13 \text{ scanlines/tri} = 114 \text{ MB/s}$$

In addition, new primitives must be added to the active object list as they first become active:

$B_{new} = P_{prim} \times S_{prim}$

So total active list bandwidth for a single rasterizer system is:

$B_{total} = B_{new} + B_{rast}$
$B_{total} = P_{prim} \times S_{prim} + P_{prim} \times S_{prim} \times H_{avg}$
$B_{total} = P_{prim} \times S_{prim} \times (1 + H_{avg})$

For the previous example:

$B_{total} = 220K \times 40 \times (1 + 13)$
$= 123M$ bytes/s

8.3 Parallelism

Rewriting the equations for active list bandwidth to include N_{par} and B_{par} from section 7.3.3 yields (Σ is used to indicate the sum of parallel rasterizers):

N_{par} = number of parallel scanlines
B_{par} = bandwidth increase over a single rasterizer

$P_{\Sigma prim} = N_{par} \times P_{prim}$
$P_{\Sigma pixel} = A_{avg} \times P_{\Sigma prim}$
$B_{\Sigma rast} = (B_{par} + 1) \times P_{prim} \times S_{prim} \times H_{avg}$
$= (N_{par} + H_{avg} - 1) \times P_{prim} \times S_{prim}$
$B_{\Sigma new} = P_{\Sigma prim} \times S_{prim}$
$= N_{par} \times P_{prim} \times S_{prim}$
$B_{\Sigma total} = B_{\Sigma new} + B_{\Sigma rast}$
$= P_{prim} \times S_{prim} \times (2N_{par} + H_{avg} - 1)$

Figure 6 graphs $B_{\Sigma total}$, showing the relatively gradual increase in bandwidth necessary to drive parallel rasterizers:

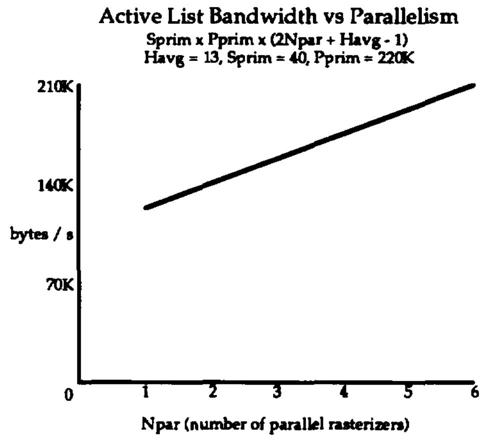

Figure 6

This gradual increase is responsible for a rapid improvement in the performance/bandwidth ratio ($P_{\Sigma prim}/B_{\Sigma total}$) as parallelism is increased:

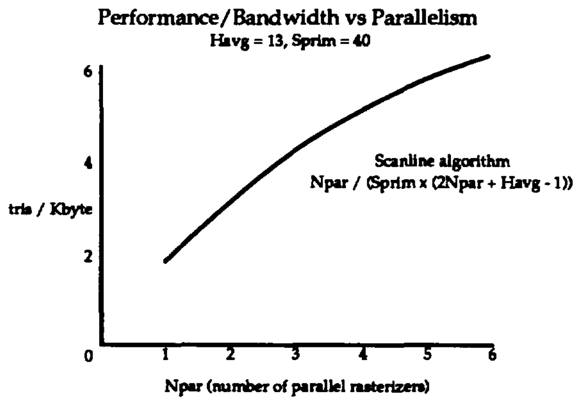

Figure 7

Repeating the previous example from 8.2, but with four parallel rasterizers:

$P_{\Sigma prim} = 4 \times 220K = 880K$ triangles/s
$P_{\Sigma pixel} = 100 \times 880K = 88M$ pixels/s
$B_{\Sigma rast} = (4 + 13 - 1) \times 220K \times 40 = 141M$ bytes/s
$B_{\Sigma new} = 4 \times 220K \times 40 = 35M$ bytes/s
$B_{\Sigma total} = 35M + 141M = 176M$ bytes/s

Figure 8 shows the total system dataflow, adding the assumption of 30 fps update of a 1M pixel frame buffer:

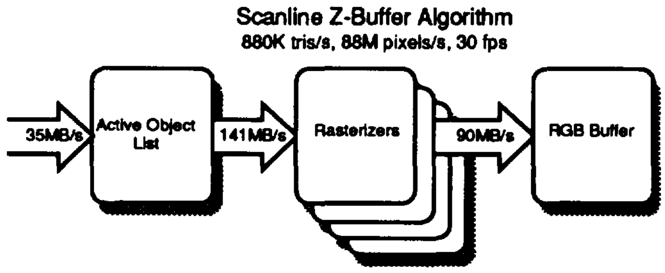
Scanline Z-Buffer Algorithm
880K tris/s, 88M pixels/s, 30 fps
35MB/s
Active Object List
141MB/s
Rasterizers
90MB/s
RGB Buffer

Figure 8

Note that the bandwidth is distributed and unidirectional, allowing an inexpensive high latency dataflow design. For this example, a 4X increase in rasterization performance required a 176M/123M = 43% increase in total active object list bandwidth.

8.4 Comparison to screen Z-buffer

For comparison, we can analyze the bandwidth that would be required to achieve this performance using a screen Z-buffer algorithm. The combined Z-buffer/frame buffer bandwidth necessary can be calculated by:

V = % of pixels visible
α = % of pixels with alpha blending
B_z = Z bytes/pixel
$B_{\alpha rgb}$ = αRGB bytes/pixel
$B_{zbuf} = P_{pixel} \times (B_z + V \times (B_z + B_{\alpha rgb} + \alpha \times B_{\alpha rgb}))$

Assuming 50% of the pixels are visible[7], 50% are blended, and 32 bits for Z and αRGB , this would be:

[7] E.g. at 30 fps, 88M/30 = 2.9M pixels/frame x 50% = 1.5M "visible" pixels. Of these, only 1M are actually visible; the others are subsequently overwritten (or blended) as rendering continues.

$$B_{zbuf} = 88M \times (4 + 50\% \times (4 + 4 + 50\% \times 4))$$
$$= 792M \text{ bytes/s}$$

Which is 4.5X the active list bandwidth of the scanline algorithm. A diagram of system dataflow for the screen Z-buffer algorithm shows this more clearly:

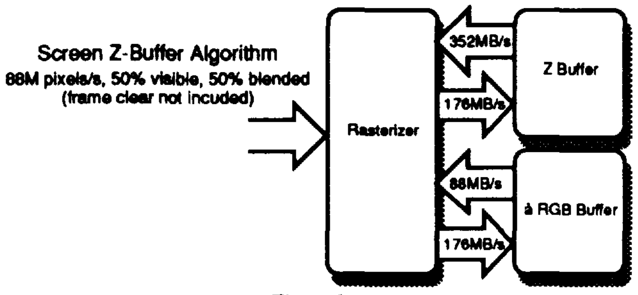
Screen Z-Buffer Algorithm
88M pixels/s, 50% visible, 50% blended
(frame clear not incuded)
Rasterizer
352MB/s
176MB/s
Z Buffer
88MB/s
176MB/s
à RGB Buffer

Figure 9

Here, high performance bidirectional (i.e. low latency) bandwidth is required for both the Z and αRGB buffers, increasing cost.

9. MEASURED PERFORMANCE

Figure 11 (at the end of the paper), shows a test image rendered on the prototype hardware. The image has 18 torusus, each composed of 1020 triangles; 9468 tris are rendered after culling. Rendering the scene requires a total of 2.06 MBytes of input data to the rasterizer. The current prototype (which is primarily a test vehicle) is limited by the 68040 host to between 5K and 10K triangles/s; however, we were able to evaluate the performance of the rasterization silicon itself by reducing the rasterizer clock until rasterization became the system bottleneck.

With the clock reduced to 1MHz, Figure 11 requires 1.5 seconds to render on a single rasterizer. Extrapolating performance to the full 40MHz clock rate:

$$Framerate = 1 \text{ frame} / 1.5\text{s} \times (40\text{MHz}/1\text{MHz})$$
$$= 26.7 \text{ frames/s}$$

$$Pprim = 9468 \text{ tris} \times 26.7 \text{ frames/s}$$
$$= 252\text{K tris/s}$$

The input bandwidth required for this performance is[8]:

$$B_{rast} = 2.06 \text{ MBytes/frame} \times 26.7 \text{ frames/s}$$
$$= 54.9 \text{ MBytes/s}$$

A higher performance, RISC-based prototype is being developed which will provide system performance that better matches rasterization capability.

10. OTHER ISSUES

10.1 Latency

For the examples presented here, the delay from start of database traversal to screen update (i.e. swap buffers) is the same for the screen and scanline algorithms. However, if double buffering is *not* being used (for example, a very large database is being drawn on the screen while the user watches), a screen algorithm would be perceived as having lower latency, as the screen update would begin sooner. This wasn't considered a serious disadvantage, however, as our target is interactive applications running at high frame rates, necessitating a double buffered display.

10.2 Hit testing

The scanline algorithm has advantages for direct manipulation applications with an interaction loop like:

```
while (FOREVER)
{
  DrawDatabase();
  GetMouse (&x, &y);
  SelectedObject = HitTestDatabase (x, y);
  UpdateDatabase (SelectedObject);
}
```

For this interaction model, the Y-sorted object activation list created during DrawDatabase() can be used to dramatically accelerate HitTestDatabase() by providing a list of only those objects visible on scanline y. This is an improvement over the classic screen Z-buffer model, where HitTestDatabase() requires a second traversal of the database and hit testing of every primitive.

11. ACKNOWLEDGEMENTS

Lee Mighdoll was a major contributor to the early stages of this project.

This research was performed in the Systems Technology Research Group of Apple Computer, Inc. Thanks to Frank Crow, Brian Heaney, Jill Huchital, David Jevans, Al Kossow, Lee Mighdoll and Libby Patterson for their reviews.

12. REFERENCES

1. Akeley, Kurt and T. Jermoluk, "High-Performance Polygon Rendering", Computer Graphics, Vol. 22, No. 4, August 1988, 239-246
2. Apgar, Brian, B. Bersack and A. Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000", Computer Graphics, Vol. 22, No. 4, August 1988, 255-262
3. Catmull, E., "A Subdivision Algorithm for Computer Display of Curved Surfaces", UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT, December 1974
4. Clark, James, "The Geometry Engine: A VLSI Geometry System for Graphics", Computer Graphics, Vol. 16, No. 3, July 1982, 127-133
5. Deering, Michael, S. Winner, B. Schediwy, C. Duffy and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", Computer Graphics, Vol. 22, No. 4, August 1988, 21-30
6. Foley, James, A. van Dam, S. Feiner and J. Hughes, "Computer Graphics Principles and Practice, 2nd Edition", Addison-Wesley, 1990, 96-99 (basic scan conversion), 201-283 (transformation), 680-685 (scanline algorithms), 885-887 (scanline rasterization)
7. Fournier, Alain and D. Fussell, "On the Power of the Frame Buffer", Transactions on Graphics, Vol. 7, No. 2, April 1988, 103-128

[8] For this image, input bandwidth is reduced approximately 30% by shared vertex data.

8. Fuchs, Henry and J. Poulton, "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine", Computer Graphics, Vol. 15, No. 3, August 1981, 80-81

9. Gharachorloo, Nader, S. Gupta, R. Sproull and I. Sutherland, "A Characterization of Ten Rasterization Techniques", Computer Graphics, Vol. 23, No. 3, July 1989, 355-368

10. Haeberli, Paul and K. Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering", Computer Graphics, Vol. 24, No. 4, August 1990, 309-318

11. Kirk, David and D. Voorhies, "The Rendering Architecture of the DN10000VS", Computer Graphics, Vol. 24, No. 4, August 1990, 299-307

12. Myers, A. J., "An Efficient Visible Surface Program", Report to the National Science Foundation, Computer Graphics Research Group, Ohio State University, Columbus, OH, July 1975

13. Niimi, Haruo, Y. Imai, M. Murakami, S. Tomita and H. Hagiwara, "A Parallel Processor System for Three-Dimensional Color Graphics", Computer Graphics, Vol. 18, No. 3, July 1984, 67-76

14. Rhoden, Desi and C. Wilcox, "Hardware Acceleration for Window Systems", Computer Graphics, Vol. 23, No. 3, July 1989, 61-67

15. Watkins, G. "A Real-Time Visible Surface Algorithm", Computer Science Department, University of Utah, UTECH-CSC-70-101, June 1970

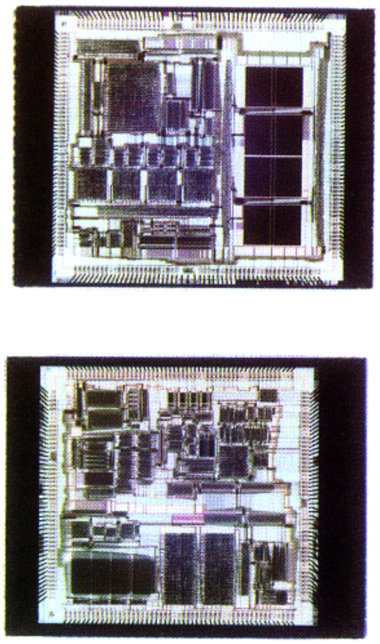

Figure 10

Rasterizer chips

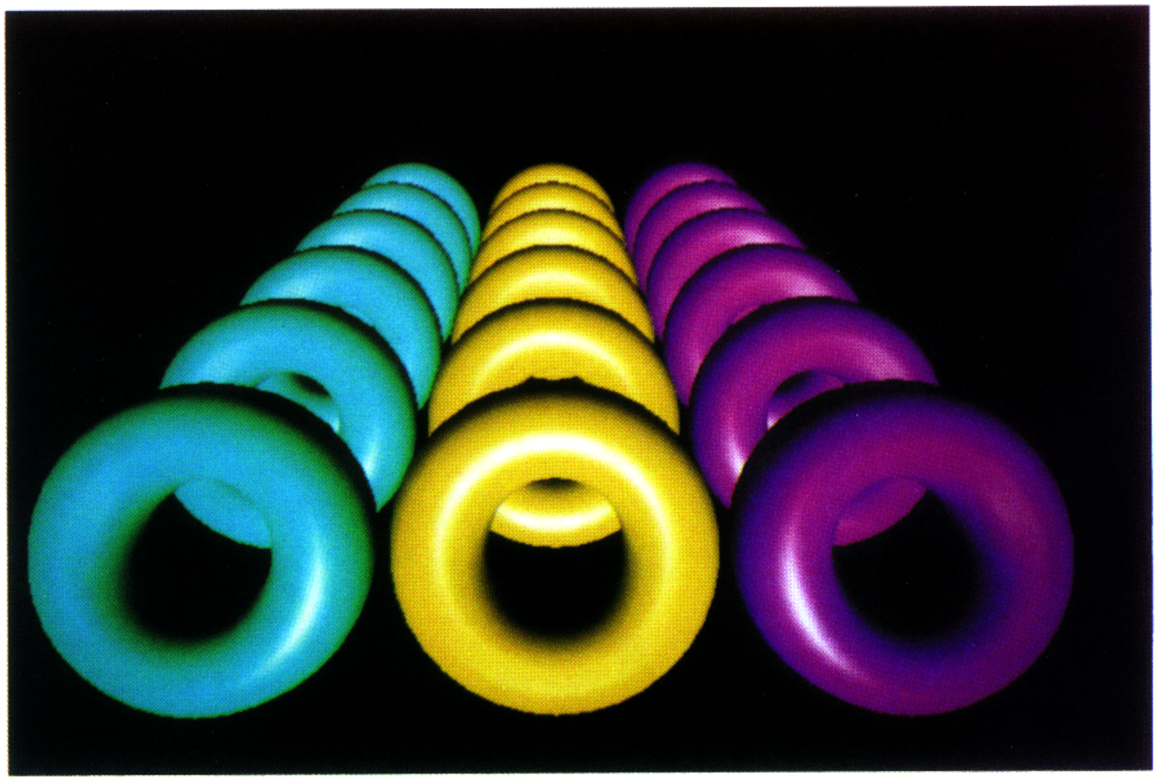

Figure 11

9468 triangle image rendered on prototype