Hardware Accelerated Rendering of CSG and Transparency

Michael Kelley, Kirk Gould, Brent Pease, Stephanie Winner, Alex Yen
Apple Computer, Inc.

ABSTRACT

This paper describes algorithms for implementing accurate rendering of CSG and transparency in a hardware 3D accelerator. The algorithms are based on a hardware architecture which performs front-to-back Z-sorted shading; a multiple-pass algorithm which allows an unlimited number of Z-sorted object layers is also described. The multiple-pass algorithm has been combined with an image partitioning algorithm to improve efficiency, and to improve performance of the resulting hardware implementation.

CR Categories and Subject Descriptors: 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

Additional Key Words and Phrases: scanline, CSG, transparency, deferred shading, image partitioning

INTRODUCTION AND BACKGROUND

This paper describes a general purpose hardware accelerator for rendering 3D graphics. In addition to basic operations such as Gouraud and texture-mapped triangles, the system includes support for high-performance rendering of Constructive Solid Geometry (CSG) and transparency. These algorithms, and the system architecture that supports them, are the main topics of the paper.

A variety of hardware algorithms have been proposed and implemented for rendering transparency or CSG, but few have been simple enough to be added to a general purpose accelerator. Ray-casting systems for CSG are slow and too complex to be implemented in hardware as low cost accelerators [13][18]. Enhanced Z-buffer algorithms for CSG often require many rendering passes, reducing performance unless extremely fast (and expensive) rasterizing systems are used [10][17]. Z-buffer transparency algorithms based on sub-pixel screen door algorithms aren't accurate for multiple layers of transparency [2]. Binary space partitioning algorithms allow ordered drawing of transparent objects, but construction overhead makes them slow for dynamic scenes, and intersecting objects aren't rendered correctly [8].

SYSTEM OVERVIEW

The system described here is a low cost, single-ASIC accelerator designed to be added to a Power Macintosh™ computer. The rasterizer is based on a scanline rendering algorithm. Several scanline based rasterizers have been proposed or built [6][14][16]; the key features of this design are:

- In addition to vertical bucket sorting (which all scanline algorithms perform), each scanline is horizontally partitioned into 16 pixel segments. The small size of this partition allows all sixteen pixels (at 480 bits/pixel) to be stored on-chip, improving performance and substantially reducing cost.
- Rasterization speed is 20M pixels/second, providing throughput of 120K texture-mapped triangles/second[1]. Images of up to 8Kx8K can be rendered in a single pass.
- An unlimited number of visible layers can be sorted by Z prior to compositing. Sorting is performed per pixel, so intersecting objects are rendered correctly.
- Transparency and CSG are performed by the compositing hardware.
- Texture map look-up is deferred until after visible surface determination, improving performance for scenes with layered texture-mapped objects.

These features will be described in more detail throughout the paper.

Z-ORDERED SHADING

Once it was decided to support rendering of CSG and transparency, it soon became apparent that algorithms based on shading in Z-sorted order (in this system, from front to back) provided the most efficient and accurate solution. The CSG and transparency algorithms themselves are described in the following sections; however, as Z-ordered shading is unusual in hardware accelerators, some background on this is provided first (a detailed discussion is in **IMPLEMENTING Z-ORDERED SHADING**, later in the paper).

Although Z-ordered shading is a common choice for high-quality software renderers [4][20], there are several reasons why it isn't usually implemented in hardware. The first is historical — most hardware acceleration architectures were developed for rendering opaque objects with a single layer Z buffer, a task for which Z-ordered shading offers no advantage. Later implementations of features such as transparency, which would benefit from Z-ordered shading, have been achieved by other algorithms which are a better fit in existing acceleration architectures. An example of this is the Silicon Graphics™ RealityEngine™, which uses sub-pixel screen door coverage masks to implement transparency [2].

In addition to the historical factors, implementing Z-ordered shading in hardware tends to require either large amounts of memory, multiple-pass rendering, or both. The simplest solution is to store multiple Z-ARGB layers per pixel, which allows accurate support of a finite number of layers. However, this requires large amounts of memory, and shows very unpleasant degradation when the number of layers is exceeded. A more elegant solution is a multiple pass algorithm [15]. This provides support of an unlimited number of layers, but it still

[1] 100 pixel triangles, with tri-linearly interpolated mip-mapped textures.

requires increased pixel memory, and becomes inefficient for scenes where large numbers of rendering passes are required.

The Z-ordered shading implementation described in the paper uses both multiple Z-ARGB layers and a multiple-pass rendering algorithm. As described later, this hybrid algorithm, combined with the image partitioning algorithm, provides solutions for both of the problems described above.

Because Z-ordered shading was used, the transparency and CSG rendering algorithms described in the next two sections are similar to those used with ray-casting algorithms. In practice, the most challenging part of the design was implementing Z-ordered shading; once that was in place, a wide range of algorithms for CSG and transparency, and potentially for other effects that require multiple visible layers, became applicable.

FRONT-TO-BACK TRANSPARENCY

The ASIC implements an interpolated transparency model, which includes the simplifying assumption that all component colours are filtered by the same coefficient [7]. This model was chosen because it is accurate, simple, and provides high visual quality. The blending function is expressed as:

$$I_r = I_{rFront} + k_{tFront}\, I_{rBack}$$
$$I_g = I_{gFront} + k_{tFront}\, I_{gBack}$$
$$I_b = I_{bFront} + k_{tFront}\, I_{bBack}$$
$$k_t = k_{tFront}\, k_{tBack}$$

Where I_{rBack} is the red intensity of the back object, I_{rFront} is the red intensity of the front object with diffuse and ambient contributions pre-multiplied by $(1\text{-}k_{tFront})$, and k_{tFront} is the transmission coefficient of the front object. Note that pre-multiplication of $I_{rgbFront}$ is necessary so that the specular component of the front object doesn't diminish as transparency increases [12]. Premultiplication creates the potential for the computed rgb values to exceed 1.0; this implementation avoids overflow of rgb by saturating each component at 255 (the internal equivalent of 1.0).

Clearly, the transparency blending function itself is quite simple — the challenge is that it requires processing the contributing objects from front to back, instead of in object submission order as is more common in hardware accelerators.

CONSTRUCTIVE SOLID GEOMETRY

Constructive Solid Geometry (CSG) is a modeling method which constructs new geometry from the union, intersection or difference of other geometry. For example, Figure 2a (end of paper) shows a complex shape created by subtracting a torus from a cube. Referring to the cube as **A**, the torus as **B**, and the result as $\mathbf{Result_a}$, it can be expressed:

$$\boldsymbol{Result_a = A - B}$$

Figure 2b shows the intersection of A and B:

$$\boldsymbol{Result_b = A \cap B}$$

In actual use, many CSG modeling operations are performed on a collection of operand geometries to construct a final object. This resulting object can be represented as an ordered sequence of Boolean set operations on geometry operands, or equivalently as an expression tree with geometry as the leaf nodes [7]. For this paper we've chosen to represent CSG expressions algebraically, like those above, referring to the Boolean set operations as *operators*, and to the leaf node geometries as *operand geometries*.

In practice, the CSG expressions of objects are usually more complex than the examples above, as the expression often represents the entire construction history of the object.

Ray Casting CSG

The most natural method for rendering CSG objects is based on ray casting [9][18]. Briefly, a ray is cast through the operand geometries; each intersection corresponds to the entry or exit of a solid operand geometry's space. These intersections allow the projection of the range of each operand's space as a span on the 1D ray. These spans are then combined using the CSG operators. In Goldstein's original paper the operators are applied in the order of the expression tree [9].

Because the ray casting algorithm uses boundary representations of the geometries, the boundary of the resulting CSG object is actually a composite of patches of the boundaries of the operand geometries. When generating images, the natural result of this is illustrated in Figure 2a and 2b — the portions of the constructed geometry that came from the different operand geometries (in this example, a cube and a torus) retain their original appearances.

The extension of this algorithm to a scanline algorithm by Atherton improved performance [3], and was the starting point for the algorithm implemented in the hardware.

Z-Ordered CSG Evaluation

The algorithm implemented in the hardware evaluates the operand intersections in Z order, rather than in expression order (as is more common in ray-casting implementations). This has two advantages: It reuses the hardware which performs Z-sorting for the transparency implementation, and it dramatically reduces the amount of state that must be stored to evaluate the CSG expression (this is discussed later).

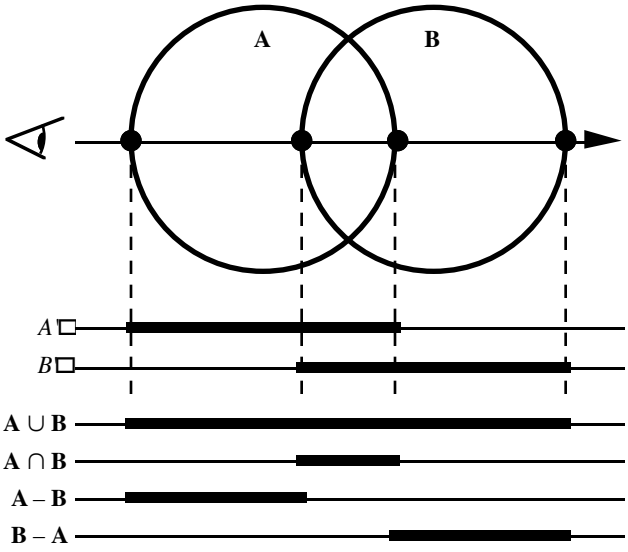

Figure 3

Figure 3, above, shows a single ray cast through two circular objects **A** and **B**. The line *A'* represents a 1 bit state variable that indicates whether the ray is currently inside object **A** (and similarly for *B'*). These state variables are updated whenever an object is intersected — four times in the above example. At any

point on the ray, the Boolean CSG function can be applied to these state variables to determine if the point is inside the constructed object. At the bottom of Figure 3 are the results of applying four different Boolean operations.

In practice, because the system renders only the boundary of objects, the Boolean function (which we will call F_{csg}) is only evaluated at the intersection points. For example, consider the **B – A** operation, which can be represented as:

$$Fcsg\ (A',B') = B' \bullet \neg\ A'$$

As the hardware composites the layers from front-to-back, it maintains the A' and B' state variables. At each intersection, it evaluates F_{csg} . When F_{csg} changes state, it represents a boundary on the constructed geometry, and the current layer is rendered. If F_{csg} doesn't change, the layer doesn't represent a boundary and is discarded.

In Figure 4, F_{csg} is 0 at the ray origin. The first two intersections I_1 and I_2 don't change the state of F_{csg} so these boundary layers are discarded — in other words, they don't represent a boundary of the resulting constructed object. I_3, however, causes F_{csg} to change from 0 to 1, indicating that the ray has crossed a "real" boundary, so the layer at I_3 is rendered. Similarly, I_4 causes F_{csg} to change from 1 to 0, so it is also rendered.

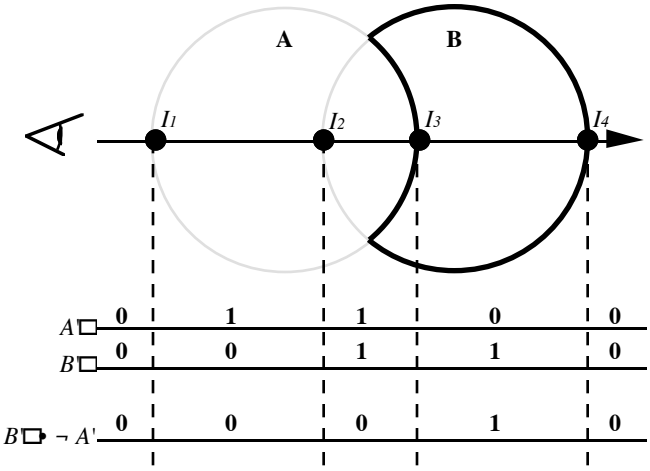

*Figure 4: F_{csg} for **B – A***

Limitations of Evaluating F_{csg} in Hardware

Because F_{csg} can be arbitrarily complex, we simplified the implementation by limiting the complexity of the function to a maximum of 5 operands. This allows F_{csg} to be implemented as a 32 entry look-up table, using the concatenation of A', B',C',D',E' as the 5 bit index (2^5=32). This table is stored in a 32 bit register which the driver software loads before starting the hardware rasterization of the frame.

It is possible, of course, to build hardware that evaluates far more complex expressions than this. We've chosen not to do so for two reasons. This first is cost — this simple evaluator required only ~500 gates, so it was inexpensive and easy to add.

The other reason is that our goal is to accelerate interactive modeling with CSG, not provide a general renderer for arbitrarily complex CSG objects. Even with hardware acceleration, solving the entire CSG expression tree for a complex object can become too slow for interactive use. For real tasks, a hybrid system which uses hardware CSG rendering for the operands which are being manipulated, while converting the stable portion of the expression tree to a boundary representation, appears to be the most versatile solution [7].

State Requirements for Implementing CSG

The principal advantage of Z-ordered evaluation of CSG is that very little state (aka memory) is required. In the ASIC, 5 bits are used to store the operand state variables A' to E', and an additional bit stores the last state of F_{csg} so that its changes can be detected — this is even less than is used to store the composited RGB value during transparency blending. (Actually, the multiple pass algorithm requires that these 6 bits be stored for each of the 16 pixels on-chip, so a total of 96 bits are used.)

By comparison, expression ordered evaluation of the CSG expression requires substantially more state storage. Because the Boolean set evaluation at each operand node can produce a virtually unlimited number of separate segments, hardware implementation is much more difficult than for an algorithm with fixed state requirements.

Z-Ordered CSG vs. Sum of Products

Another popular algorithm for rendering CSG is to decompose the CSG expression into sum-of-products, and then perform multiple-pass rendering of each two-operand expression until the entire CSG expression has been performed. This method is particularly popular with hardware accelerators, because it requires a fixed (and small) amount of per-pixel storage [10][17].

In general, Z-ordered evaluation has an advantage over this method because the entire CSG expression is evaluated in a single rendering pass[2]. Implementations of sum-of-products rendering have varied, but in general they require 2-3 passes for a difference of two objects, with exponential growth as the number of operands increases past two.

An advantage of sum-of-products solutions is that they can render an arbitrarily complex scene (although complex expressions may take a long time).

IMPLEMENTING Z-ORDERED SHADING

Although Z-ordered shading isn't usually performed in hardware accelerators, many high-quality software rendering algorithms operate in this fashion — in particular, ray-tracing and A-buffer algorithms are very popular [4][20].

Ray-tracing intrinsically operates in front-to-back order, and experimental hardware implementations have been built [13]. However, ray-tracing is still too computationally complex for use in a low cost accelerator.

The A-buffer algorithm generates a front-to-back sorted list of all the pixel fragments that affect each pixel, allowing both transparency and anti-aliasing computations to be performed as the list is composited. Front-to-back ordering is performed by a list sorting operation, which is more suitable for hardware implementation. However, because the layer list can

[2] This isn't strictly true when the multiple pass algorithm is added; however, the number of passes is still a fraction of the number of layers, rather than an exponential function of expression complexity.

potentially be quite large, it isn't possible to store it in on-chip RAM, so off-chip memory must be used. Achieving a 40MHz clock speed with the increased latency of off-chip memory proved to be too expensive (and difficult!) to be practical.

Instead, we chose to use a somewhat simpler and less efficient Z-ordered shading algorithm, and then improve the efficiency by combining it with an image space partitioning algorithm. By using a simpler Z sorting algorithm, all the Z and ARGB RAM could be kept on-chip, making a 40MHz clock reasonable. Also, the image space partitioning algorithm reduced system cost substantially by eliminating the system Z-buffer, and by reducing system bandwidth [14][16].

UNLIMITED VISIBLE LAYERS

The algorithm used is a hybrid of a simple list sorting algorithm, and a multiple pass rendering algorithm similar to that described by Mammen [15]. Very briefly, Mammen's algorithm operated by first rendering the opaque objects in the scene. Then the transparent objects were rendered, retaining only the furthest layer of transparency per pixel. Once all transparent objects were rendered, the transparency layer was composited with the opaque layer, Z values updated, and, if necessary, the transparent objects were re-submitted, rendering another layer of transparency. The process was repeated until all transparent layers were composited, requiring one iteration for each layer of transparency at the deepest point in the image.

This algorithm has the advantage of using a fixed (and small) amount of memory per pixel. However, in the original form it is efficient only for scenes without deeply layered transparency, or where only a small percentage of the objects are transparent. In this system Z sorting is used for both transparency *and* CSG, so it's not unusual for all objects in the scene to require Z sorting — if the original form of the algorithm was used, this would cause many rendering passes and correspondingly low performance.

Start with the Four Closest Layers

The implementation process began by determining how many layers of sorted Z could be inexpensively supported in hardware, while still providing enough layers to render typical images efficiently. A normal Z-buffer system has a single layer; we simulated designs with 2, 4 and 8 layers on a variety of tests. The goal was to find a depth which could render the common test cases without overflow, or at least with overflow on only a small percentage of the image. *Figure 5*, at the end of the paper, shows a sample test image containing 17 tori, half of which are transparent. Figure 6a (below) is a gray scale rendering of the number of visible layers at each pixel, ranging from zero (black) to six (white).

As the tests were performed, the effect on ASIC cost and performance was evaluated. In the end, a 4 layer deep Z-ARGB buffer provided the best result. For the test image in Figure 5, only the region shown in Figure 6b requires more than four layers to render (Figure 6b will be discussed again later). With double-buffering and miscellaneous control bits added, this implementation worked out to (24+32+4)*(4*2) = 480 bits/pixel. For the 16 pixels stored on chip, a total of 7.7 KBits of on-chip RAM were used, well within the cost constraints of the system.

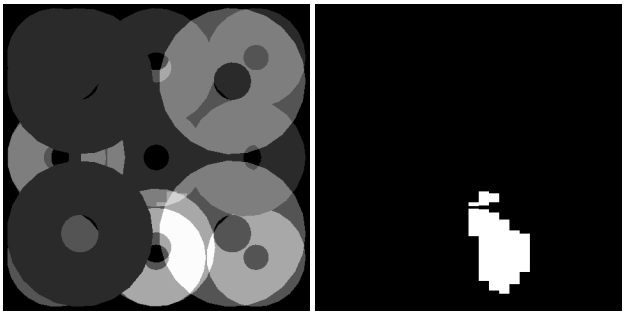

Figure 6a
Number of Visible Layers

Figure 6b
Region Requiring Two Passes

Sort from Front to Back

During rendering, the four layer deep Z-ARGB buffer is used to retain the four closest visible layers per pixel. For simple scenes without transparency or CSG, only the first Z layer is used, and behaviour is identical to a normal Z buffer. However, when transparent or CSG objects are rendered, they are recognized as non-opaque and are inserted into their correct position in each pixel's list. If more than four visible layers are required for any pixel, an overflow flag is set indicating that more than four Z layers will be necessary to complete rendering, and the four closest layers are retained.

If Necessary, Composite and ReRender

When overflow occurs, the system composites the four closest layers into a single layer, and stores the result, with the backmost Z, into the first layer of the Z buffer. The objects are then resubmitted, and, similarly to [15], the remaining three Z layers are used to capture the *next*-closest three layers of each pixel. This process continues until overflow does not occur, effectively compositing an unlimited number of layers from front to back.

Both the CSG and transparency operations are performed as part of the compositing operation. Note that the CSG operand state bits (A', B'...) must be stored per pixel when overflow occurs so processing can be resumed for the next three layers.

Don't Overflow for Hidden Layers

As is usual in software implementations, the system includes an optimization which stops the compositing process once an opaque layer is reached. This avoids unnecessary overflow to process objects hidden behind an opaque object, and improves compositing performance.

This optimization requires that Z-sorting be performed from front to back.

Z Sort Performance

When four or fewer visible layers are necessary to render the image, front-to-back sorting speed is governed by the Z list sort-insert time. For random data, the average number of clocks required to insert a pixel into a list of 0, 1, 2 or 3 layers is:

layers	*clocks*
0	1
1	1
2	1 + 1/2(1) = 1.5
3	1 + 2/3(1 + 1/2(1)) = 2

Assuming that all pixels have 4 visible layers, an average of 1+1+1.5+2 = 5.5 clocks/pixel are required for Z sorting. Note

that this is roughly balanced with compositing speed (1 clock/layer, or 4 clocks/pixel), so Z sort and composite performance stay roughly balanced for up to four visible layers. Also, object processing speed is not too degraded from the opaque object case, with average clocks/pixel layer = 5.5/4 = 1.4.

Increasing the number of layers to 7 both increases the rendering time of the first pass, as the additional 3 layers take an average of 2 clocks/pixel-layer to insert, and requires one additional rendering pass to sort and composite the 3 overflow layers. During the first pass, the average clocks per pixel rises from 5.5 to 11.5 (5.5 + 3*2). During the second pass, the four previously composited layers are discarded in 1 clock/layer, and the three overflow layers are inserted in:

layers	*clocks*
4	1
5	1 + 1/2(1) = 1.5
6	1 + 2/3(1+1/2(1)) = 2

Total processing per pixel therefore averages 11.5+4+1+1.5+2 = 20 clocks/pixel. Because seven layers are being processed, this raises the average clocks/pixel layer to 20/7 = 2.8. Although this represents a substantial performance drop, it's much less than the 7X penalty a single layer multiple-pass algorithm would impose.

Of course, real images do not have the homogenous distribution of layers that these calculations have assumed. The next section discusses system performance at the image level.

IMAGE PARTITIONING

Image partitioning is a well established method used with many software algorithms and hardware architectures [6][14][16]. In general, image partitioning algorithms divide the entire image to be rendered into a number of smaller regions, each of which is rendered separately.

Image partitioning algorithms can have a number of advantages. If the algorithm is designed to render each region independently, it becomes possible to render multiple regions simultaneously, enabling the use of parallel rendering hardware. Alternatively, the algorithm can be designed to share rendering state between the partitions, in which case the main advantage is a reduction of working memory (usually because fewer pixels are stored at once); Watkin's scanline algorithm is a classic example of this [19]. Some systems have exploited both of these advantages [14].

There were two reasons for using an image partitioning algorithm in this system. The first was to reduce working memory to an amount that could be stored in on-chip RAM. Keeping the multiple Z and ARGB layers stored on-chip made it possible to design a Z-sort module that operates at 40M pixel-layer/s, using 400 MB/s of on-chip RAM bandwidth. Although it would have been possible to achieve this performance with off-chip RAM, the system cost would have been substantially higher .

The second reason to use image partitioning was to improve the efficiency of the multiple-pass Z sorting algorithm described earlier. By applying the overflow and re-render tests at a finer level than for the entire image, the efficiency of the algorithm was greatly improved.

In addition to these two performance improvements, image partitioning allowed us to virtually eliminate the limit on image size usually imposed by hardware accelerators. The system can render an image of up to 8Kx8K resolution in a single pass.

Two Dimensional Bucket Sorting

The image space partitioning algorithm is a variation of the classic scanline algorithm. It begins by bucket sorting all triangles in the image by their first active scanline, a step which has been included in several other hardware accelerators [6][14][16]. Rendering traversal then begins by creating a Y active object list, which is maintained as rendering advances, scanline by scanline, down the image.

At the beginning of each scanline, the Y active list is bucket sorted horizontally (i.e. by X), with each bucket representing a 16 pixel segment of the scanline. (This is variation from the classic scanline algorithm, which performs the X sort by pixel.) Once the X bucket sort is complete, rendering proceeds from left to right across the scanline, maintaining an X active list. Rendering of each 16 pixel segment is completed before advancing to the next segment.

In pseudo-code, and with the multiple-pass algorithm added, the partitioning algorithm can be written as:

```
YBuckets [NScanlines];
XBuckets [ScanlineWidth/16];

ForEachTriangle {
  bucket = FindFirstScanline (tri);
  AddToYBucket (bucket, tri);
}

ForEachScanline {
  AddToYActiveList (YBuckets [scanline]);

  ForEachYActiveTri {
    bucket = FindFirstXSegment (tri);
    AddToXBucket (bucket, tri);
  }

  ForEachSegment{
    AddToXActiveList (XBuckets [segment]);
    do {
      ForEachXActiveTri {
        Render (tri);
      }
    } while (Overflow);
    ScanoutBucket;
  }
}
```

Figure 7

Although the code above shows all operations occurring sequentially, whenever possible the hardware overlaps the different phases of the algorithm to avoid idle rasterization time. In particular, for high performance it is necessary to double buffer the Z-ARGB memory so Z sorting can advance to the next segment while the previous segment's compositing is performed.

Improving Efficiency of the Multiple Pass Algorithm

An advantage of partitioning the image is that the multiple pass algorithm can be applied with much finer granularity than re-rendering the entire image. In many cases only a small percentage of the image will require more than four composited layers; by testing each individual region for overflow and resubmitting only the active objects for that region, overall system efficiency is greatly improved.

Figure 8a and 8b are derived from the test image shown in Figure 5. 8a shows the number of layers per pixel without

considering opacity; 8b shows the number of visible layers per pixel. In both cases, the maximum depth is 6 layers, so if the image was rendered using the multiple pass algorithm, two passes would be required (assuming a four layer Z-ARGB buffer). For this test image of 13056 triangles, $N_{TriRend}$, the number of objects * the number of times each object is rendered, can be computed:

$N_{TriRend}$ = 2 passes * 13056 tris/pass = 26112 tris

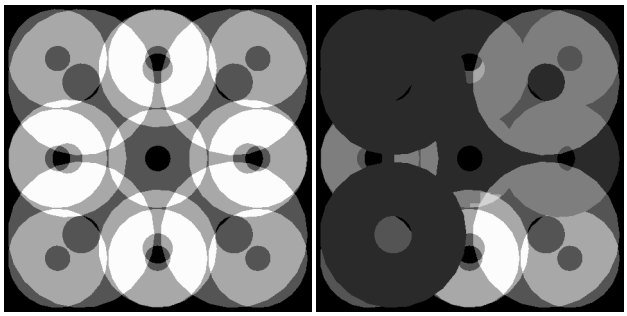

Figure 8a *Figure 8b*

$N_{TriRend}$ can be used to measure the efficiency improvements that result from partitioning the image.

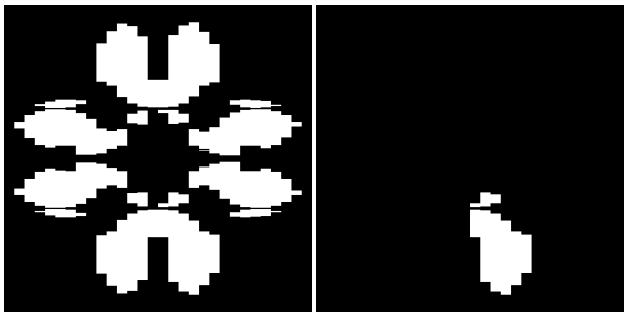

Figure 9a $N_{TriRend}$ *= 21934*

Figure 9b $N_{TriRend}$ *= 14231*

Figure 9a, above, shows the number of rendering passes required for each 16 pixel segment of the scene, assuming the additional optimization of testing for opacity before re-rendering is *not* performed. Even for this simplified algorithm, image partitioning provides an improvement — the simulation measures $N_{TriRend}$ at 21934, a 16% reduction over re-rendering the entire image.

However, the real savings are indicated by Figure 9b, which includes the additional optimization of testing opacity before asserting overflow. In a scene like this test image, where opaque and transparent objects are intermingled, this yields substantial performance improvements — in this case, $N_{TriRend}$ reduces to 14231, a 45% reduction over performing two rendering passes on the entire image.

Other Image Partitioning Solutions

The efficiency improvements described in the previous section could be further improved by switching to an image partitioning system with better 2D image locality — for example, a 16 pixel partition which was 4x4 pixels instead of 1x16. However, we found that the other advantages of rendering in scanline order (mainly simplicity) outweighed any potential gains.

SYSTEM ARCHITECTURE

The system splits the rendering task between software and hardware. Transformation, clipping and shading are performed by the PowerMac™ CPU. Rasterization is performed by the ASIC described in this paper. The algorithms used for transformation, clipping and lighting are typical of hardware accelerated workstations; [1][5][11] describe these algorithms in detail. The CPU also performs the initial Y bucket sort of the triangles.

The ASIC rasterizer performs several different tasks, as shown by the pseudo-code in Figure 7 (earlier in the paper). In hardware, these tasks are implemented by multiple modules (shown in Figure 10) which are linked by high speed datapaths. The Z and ARGB RAM are on-chip, as they require very high bandwidth and low latency; system memory is used for triangle storage.

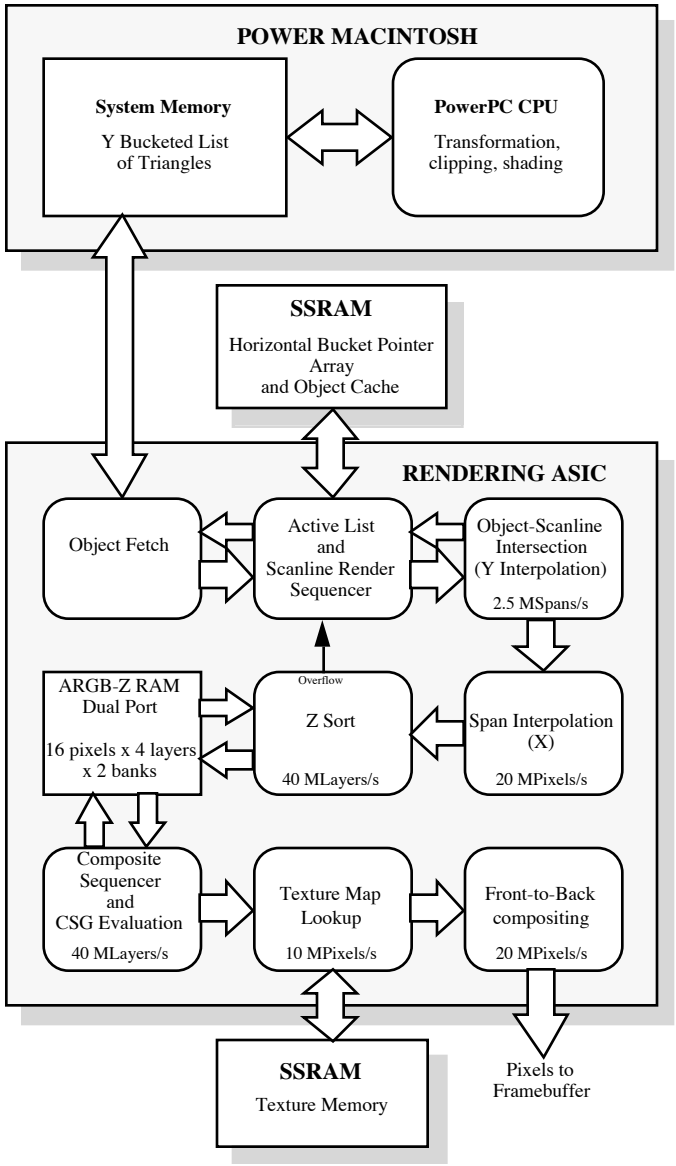

Figure 10

OBJECT FETCH

This module provides the interface to the system bus.

ACTIVE LIST AND SCANLINE RENDER SEQUENCER

The **Scanline Render Sequencer** controls the rendering of the current scanline, and performs horizontal bucket sorting of the active objects before starting the scanline render. Vertical and horizontal active list maintenance is also performed in this module. The array of object pointers used for the horizontal bucket sort is stored in off-chip SSRAM.

OBJECT-SCANLINE INTERSECTION

This module intersects triangles with the current scanline, and computes the span endpoint values for the interpolated parameters (such as ARGB and Z). In addition, the X intersect values are fed back to the **Active List** module to allow horizontal bucket sorting of the triangles.

SPAN INTERPOLATION

This module clips the horizontal span endpoints generated by triangle-scanline intersection to the current X segment, and the parameter values are interpolated for each pixel within the span. As pixels are interpolated, they are output to the **Z Sort** module. Performance is 20 MPixels/s.

Z SORT

This module inserts the interpolated pixel data into the appropriate layer of the four layer **ARGB-Z RAM**. If overflow occurs, an **Overflow** signal is send to the **Scanline Render Sequencer** indicating that an additional rendering pass will be necessary for the segment. Note that the RAM is double buffered, so that compositing of the previous segment can be performed while the next segment is being sorted.

The **Z Sort** module runs faster than **Span Interpolation** (40 MLayers/s vs. 20 MPixels/s) to compensate for the non-linear increase in sorting operations required for scenes with multiple visible layers per pixel.

COMPOSITE SEQUENCER AND CSG

The **Composite Sequencer** reads the pixel layers in front-to-back order from the double-buffered **ARGB-Z RAM**, and performs CSG visibility evaluation at 40 MLayers/s. Non-visible layers are discarded.

TEXTURE MAP LOOKUP

If the incoming pixel layer should be texture mapped, eight texture map values are read from the off-chip RAM and tri-linearly interpolated to generate the diffuse color, after which diffuse and specular lighting are applied. Texture mapping is limited by RAM bandwidth to 10 MPixels/s; however, because texture is applied after hidden surface removal and CSG evaluation, rendering throughput is typically not degraded.

FRONT-TO-BACK COMPOSITING

The final pixel processing is performed by compositing the incoming pixel layers in front-to-back order, after which the resulting ARGB values are output to the frame buffer.

CONCLUSIONS

These are the main design goals met by the system:

CSG and Transparency

An unlimited number of transparent layers can be composited in correct Z order. Z-ordering is performed per-pixel, so intersecting objects are rendered correctly. Arbitrary CSG operations of up to five operands are supported; more complex operations could be implemented if desired.

Single ASIC Implementation

The entire rasterization engine has been implemented as a single ASIC, with all low latency datapaths on-chip. This resulted in a low-cost implementation, and potential for substantial performance growth as ASIC technology advances. Off-chip Z buffer memory isn't required.

High Performance

The ASIC can rasterize 120K texture-mapped triangles/s. (This would only qualify as midrange performance by today's workstation standards, but is respectable for a single ASIC rasterizer.) Interestingly, even this basic benchmark (i.e. no CSG or transparency) benefited from Z-ordered shading because the hidden surfaces were discarded before texture mapping was performed.

FUTURE WORK

Although the implementation described here provides good performance for a single ASIC rasterizer, scaling performance up to higher levels will require additional parallelism. Some form of parallelism at the image partition level would be ideal.

ACKNOWLEDGEMENTS

The authors wish to thank Kai-Fu Lee and Rick LeFaivre for supporting this research in Apple's Interactive Media Lab.

Thanks to Jill Huchital and Dan Venolia for their reviews.

REFERENCES

1. Akeley, Kurt and T. Jermoluk, "High-Performance Polygon Rendering", Computer Graphics, Vol. 22, No. 4, August 1988, 239-246

2. Akeley, Kurt, "RealityEngine Graphics", ACM Computer Graphics Conference Proceedings, August 1993, 109-116

3. Atherton, Peter, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry", Computer Graphics, July 1983, 73-82

4. Carpenter, Loren, "The A-buffer, an Antialiased Hidden Surface Method", Computer Graphics, Vol. 18, No. 3, July 1984, 103-108

5. Deering, Michael, and S. Nelson, "Leo: A System for Cost Effective 3D Shaded Graphics", ACM Computer Graphics Conference Proceedings, August 1993, 101-108

6. 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

7. Foley, James, A. van Dam, S. Feiner and J. Hughes, "Computer Graphics Principles and Practice, 2nd Edition", Addison-Wesley, 1990, transparency 754-755, CSG tree 557-558, CSG b-rep 546-547
8. Fuchs, Henry, G. Abram, and J. Poulton, "Near Real-Time Shaded Display of Rigid Objects", Computer Graphics, Vol. 17, No. 3, July 1983, 65-72
9. Goldstein, R. and R. Nagel, "3-D Visual Simulation", Simulation 16(1), January 1971, 25-31
10. Goldfeather, Jack and J. Hultquist, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System", Computer Graphics, Vol. 20, No. 4, August 1986, 107-116
11. Harrell, Chandlee, and F. Fouladi, "Graphics Rendering Architecture for a High Performance Desktop Workstation", ACM Computer Graphics Conference Proceedings, August 1993, 93-100
12. Kay, D., "Transparency, Refraction and Ray Tracing for Computer Synthesized Images", Thesis, Cornell University, January 1979
13. Kedem, G. and J. Ellis, "The Raycasting Machine", Proceedings of ICCD, October 1984, 533-538
14. Kelley, Michael, S. Winner, and K. Gould, "A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm", Computer Graphics, Vol. 26, No. 2, July 1992, 241-248
15. Mammen, A., "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique", Computer Graphics and Applications, 9(4), July 1989, 43-55
16. 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
17. Rossignac, Jaroslaw, and A. Requicha, "Depth-Buffering Display Techniques for Constructive Solid Geometry", IEEE Computer Graphics and Applications, September 1986, 29-39
18. Roth, Scott, "Ray Casting for Modeling Solids", Computer Graphics and Image Processing, 18, 1982, 109-67
19. Watkins, G. "A Real-Time Visible Surface Algorithm", Computer Science Department, University of Utah, UTECH-CSC-70-101, June 1970
20. Whitted, T. "An Improved Illumination Model for Shaded Display", CACM 23(6), June 1980, 343-349

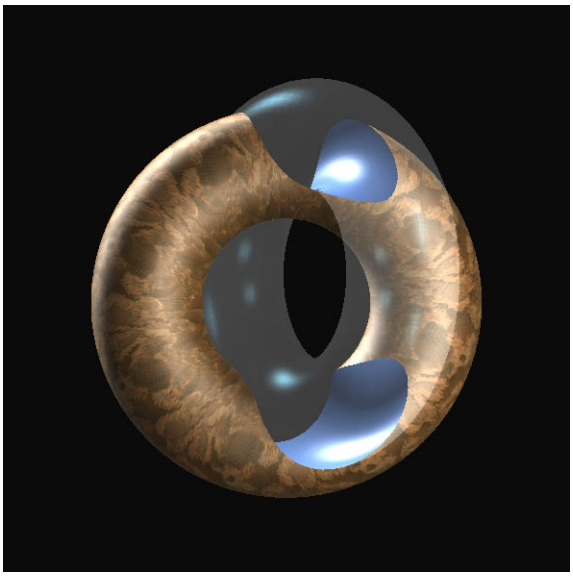

Figure 1: Difference of Two Tori

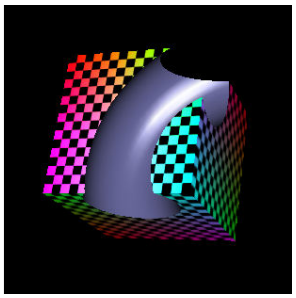

Figure 2a
Cube – Torus

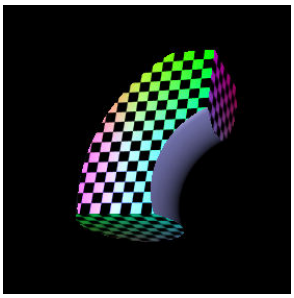

Figure 2b
Cube ∩ Torus

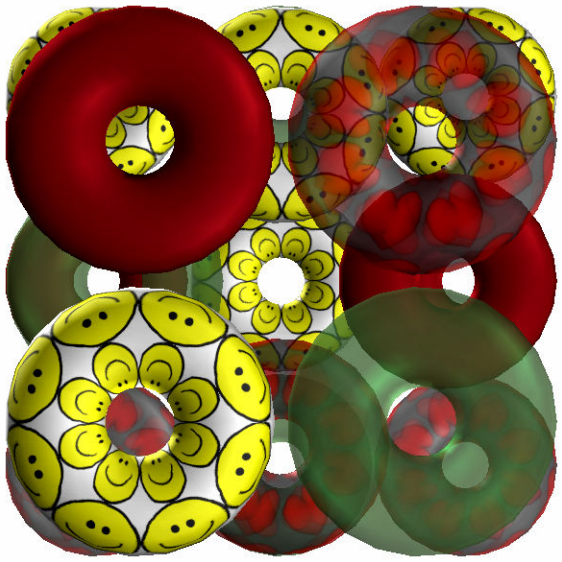

Figure 5: Test Image