Heresy: A Virtual Image-Space 3D Rasterization Architecture

Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

Abstract

With the advent of virtual reality and other visual applications that require photo and cinema realism, 3D graphics hardware has started to enter into the main stream. This paper describes the design and evaluation of a cost-effective high-performance 3D graphics system called *Heresy* that is based on virtual image-space architecture. *Heresy* features three novel architectural mechanisms. First, the *lazy shading* mechanism renders the shading computation effort to be proportional to the screen area but independent of the scene complexity. Second, the speculative Z-buffer hardware allows one-cycle Z-value comparison, as opposed to four cycles in conventional designs. Third, to avoid the intermediate sorting required by virtual image-space rasterization architecture, we develop an innovative display database traversal algorithm that is tailored to given user projection views. With this technique, the sorting-induced delay and extra memory requirements associated with image-order rasterization are completely eliminated. By replicating the *Heresy* pipeline, it is estimated that the overall performance of the system can reach over 1 million Gouraud-shaded and 2D *mip-mapped* triangles per second at 20 frames/sec with 1K x 1K resolution per frame.

Keywords: image space, object sapce, lazy shading, speculative z-buffer sorting, 3D scan conversion, inverse projection

1997 SIGGRAPH/Eurographics Workshop

1 Introduction

Emerging applications of Virtual Reality technologies require a significant performance improvement in 3D graphics rendering for interactive response. The goals for future 3D graphics rendering are photo realism and cinema realism, which basically means very-high-speed generation of high-quality images. It has been estimated that the required visual reality is at 80 million polygons per picture and at least 24 pictures per second. To provide this level of image quality at 10 frames per second, the total rendering performance is 800 million polygons per second. As a reference point, state-of-the-art high-end graphics workstations such as SGI's RealityEngine offers a performance level of 1 million, anti-aliased, Gouraud shaded, and texture-mapped triangles per second.

Existing graphics architectures typically assume a two-phase pipeline: *geometry manipulation* and *rasterization*. Because geometry manipulation works on a primitive-by-primitive basis, it is rather straightforward to exploit the parallelism among geometric operations performed on the primitives in a 3D model. On the other hand, during the rasterization stage, the pixel values contributed by different primitives eventually have to be projected and combined onto the same screen space. Therefore, synchronization among the rasterization of primitives is required. As a result, it is comparatively more difficult to scale up rasterization performance compared to geometry manipulation. In addition, because rasterization involves low-level pixel handling, the amount of memory access is significantly greater than geometry manipulation. In fact, it is memory access rather than CPU processing that is the performance bottleneck for rasterization.

In this paper, we propose a new 3D graphics architecture called *Heresy* that is based on the *virtual image-space rasterization* architecture [9]. In image-space architecture, given an image region, the system rasterizes the subset of 3D primitives in the model database which contribute to that region. An image region can be a linear scanline or a rectangular block. This process repeats for every image region in the screen space. The notion of virtual image-space architecture means that instead of dedicating a rasterization engine to each and every image region, the same set of rasteriza-

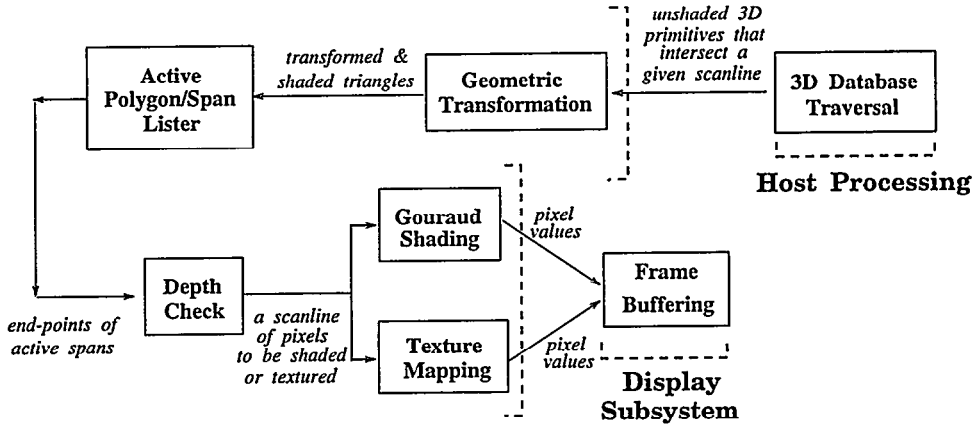

Figure 1: *The data flow in the* Heresy *architecture.*

tion engines are reused, i.e., ***virtualized***, to render the entire screen space. Because an important design goal of *Heresy* is cost-effectiveness, *Heresy* uses one rasterization engine that traverses through the screen space on a scanline by scanline basis. An immediate advantage of this approach is that the amount of Z buffer memory is significantly reduced.

Compared to previously proposed 3D graphics engines, *Heresy* features three innovative architectural mechanisms that collectively reduce the computation overhead of 3D rasterization. First, ***lazy shading*** delays the shading computation for the pixels of the triangles until the responsible triangle for each image plane pixel is determined. In addition, to simplify the data path width through the Z buffer, a ***computation pointer*** mechanism is developed to avoid transfer of the shading arguments. An important advantage of ***lazy shading*** is that the shading computation complexity is now only proportional to the screen size, but is independent of the scene complexity, i.e., the number of triangles. This performance advantage is especially important to *Heresy* because it also supports texture mapping.

Because of ***lazy shading***, the only portion of the rasterization process that is dependent on the number of triangles is depth-value comparison. The second innovation in *Heresy* is a speculative Z buffer implementation, which reduces the Z-value comparison overhead from four cycles in conventional designs to one cycle. Speculative Z buffering requires more expensive hardware support per pixel, but is still a feasible design choice because *Heresy* adopts a virtual image-space model, which only requires a small portion of the full-screen Z buffer memory.

A fundamental problem associated with the virtual image-space rendering model is that an explicit sorting phase must be inserted between geometry manipulation and rasterization. This sorting phase lengthens the rendering latency as well as forces the intermediate buffer to be large enough to hold the entire 3D model database. In *Heresy*, we propose a rasterization-oriented order to traverse the display database in the object coordinate system. Consequently, there is no need to wait for all the primitives to be geometrically transformed before rasterization could start. An immediate benefit of this approach is the elimination of processing delay and buffer memory associated with intermediate sorting.

In addition to the above three features, *Heresy* also incorporates a highly efficient tri-linear interpolation hardware for 2D and 3D texture mapping. The rest of this paper is organized as follows. In the second section, we discuss the design goals and overall architecture of *Heresy*. In Section 3, 4, and 5, *Heresy*'s three unique architectural features are described in more detail. Section 4 includes the discussion of the detailed design of tri-linear interpolation hardware as well. In Section 6, a preliminary performance evaluation of *Heresy* is presented. In Section 7, previous works that influence the design decisions of *Heresy* are reviewed, to set the research contributions of this work in perspective. Section 8 concludes this paper with a summary of main ideas and an outline of the on-going work.

2 Heresy: The Architecture

The three design goals of *Heresy* are ***balance***, ***cost-effectiveness***, and ***modularity***. To attain very high 3D graphics rendering performance, it is essential to maintain the balance among the stages in the rendering pipeline. More concretely, the processing delay of each pipeline stage must be approximately the same to avoid any bottleneck. *Heresy* targets at a single-board implementation that could be added to mainstream PC or workstation machines. As a result, *Heresy* chooses the virtual image-space rendering architecture to minimize the cost of rasterization hardware. Finally, *Heresy* also aims to be scalable from personal interactive graphics to large-scale rendering tasks such as distributed simulation/training. So the architecture is designed in such a modular fashion that higher performance could be achieved by replicating the hardware.

Figure 1 shows how graphics primitives are transformed as they flowed through various pipeline stages of *Heresy*. Because *Heresy* renders one scanline after another, the set of 3D graphics primitives in the database that contribute to one scanline must be traversed before those associated with the next scanline. In *Heresy*, it is the host CPU that performs the traversal of the 3D primitive database according to the above constraint. After relevant primitives are identified, they are passed to *Heresy* and subject to geometric transformation according to view angles and projection parameters. The transformations translate polygons into triangles with each vertex labeled with its normalized projection coordinate, the RGB color components and an α value for compositing. ***Active polygon/span lister*** maintains the set of active triangles for a given scanline and decomposes each active triangle into spans that run in parallel with the X axis by walking through the triangle's edges along the Y axis. Each span is now characterized by its two end-points and passed to the depth check module. The depth (Z value)

of each pixel on a span is first calculated by interpolating the end points' Z values, and compared with the depth value of that pixel currently in the Z buffer. Only after all the triangles that contribute to the scan line have been processed will the depth check module pass the color/coordinate information associated with the scanline pixels to the Gouraud Shading or Texture Mapping modules, which produce the final pixel color values and pass them to the frame buffer for display.

Although the pipeline structure of *Heresy* may seem similar to other graphics machines, the implementation techniques actually are quite different. In the next three sections, we will present how each of *Heresy*'s three architectural mechanisms work in detail.

3 Rasterization-oriented Display Database Traversal

An important assumption of image-space rasterization architectures is that the clipped and shaded 3D primitives must be pre-sorted before rasterization. Pre-sorting is necessary because image-space architecture must deal with all the primitives contributing to a given image region (e.g., a scanline) simultaneously. Previous approaches [6] [5] invariably assume a sorting phase between geometry manipulation and rasterization. This scheme exhibits a fundamental problem: All 3D primitives in the scene must be traversed before rasterization could start. In other words, the processing in geometry manipulation and rasterization cannot be effectively pipelined. In addition, an intermediate buffer memory at least as large as the display database must be present between the two subsystems. High-end systems such as SGI's RealityEngine [1] avoids sorting through a parallel approach in which one rasterization engine is dedicated to every image region and the transformed 3D primitives are broadcast to these engines via a broadcast bus. Although this approach does away with the sorting delay, the utilization of the rasterization subsystem in general is low. This is because for each primitive only those rasterization engines whose associated image regions overlap with the primitive will be active.

In this section, we present a novel technique used in *Heresy* that attempts to eliminate the sorting delay between geometry and rasterization subsystems in a virtual image-space rasterization architecture. Intuitively, this technique attempts to traverse the 3D display database in an order that is demanded by the rasterization process. In other words, suppose the outputs from the geometry subsystem follows an order consistent with the rasterization order, then there is no need to have an explicit sorting phase in between! To the author's knowledge, existing 3D graphics rendering machines never attempt to exploit the extra degree of freedom in display database traversal to avoid the sorting overhead in virtual image-space rasterization architectures.

To determine the set of 3D primitives that contribute to a given scanline, *Heresy* first inverse-projects that scanline back to the original world coordinate system, and then identifies the set of 3D primitives in the display database that interact with the inverse-projected image of the scanline. To put this technique in perspective, previous approaches require the sorting of 3D primitives, after they are transformed to the normalized projection coordinate system, to fit the user-specified 2D view. In contrast, our technique projects the image regions of the given user view back to the original object coordinate system, and identifies related 3D primitives. An important advantage of our approach is

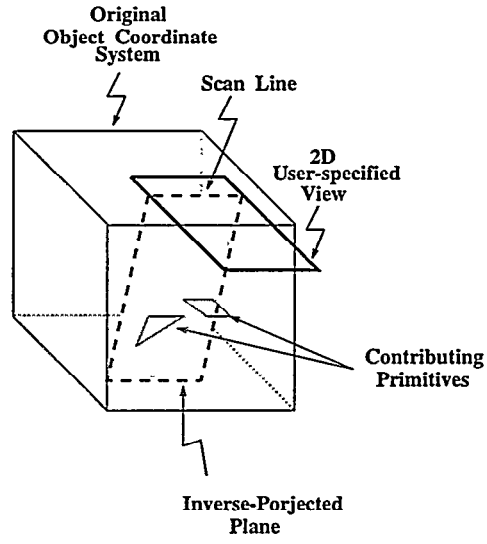

Figure 2: *Each scanline of the screen space is back projected into the original object coordinate system. All the 3D primitives that intersect with the inverse-projected plane associated with the scanline are the primitives that eventually contribute to the scanline.*

that 3D primitives can be pre-sorted according to the object coordinate system, and at run time no additional sorting is necessary. Because the sorting is performed with respect to a coordinate system that is independent of projection parameters, sorting of 3D primitives only needs to be done once!

The concrete steps in the new display database traversal algorithm are as follows.

```
At Pre-processing Time:
   Bucket-sort the 3D primitives in the object
      coordinate system;
   Compute the back-projection matrix that
      inverses model and view transformations;
At Run Time:
   Current_Scanline = 0;
   Projection_Plane = Inverse-project(Current_Scanline);
   While (Current_Scanline < Screen_Length)
      Bucket_Set = Intersect(Projection_Plane);
      Send the 3D primitives associated with the buckets
         in Bucket_Set to the Geometry subsystem;
      Current_Scanline ++;
      Projection_Plane = Inverse-project(Current_Scanline);
   endWhile
```

The bucket sort attaches each primitive to the buckets with which it intersects and is performed only once statically, i.e., not at run time. The *Screen_Length* represents the number of scanlines in the screen, and the *Intersect* function finds all the buckets in the 3D display database that intersects with the projection plane of a given scanline. In practice, the *Intersect* function is implemented as a 3D scan conversion procedure [2] of the back-projection plane with respect to a 3D space whose basic unit is the bucket. The set of primitives contributing to a scanline are those in the buckets that constitute the scan-coverted version of the scanline's back-projected plane, as shown in Figure 2. Of course, primitives that contribute to multiple scanlines are sent to the geometric transformation module only once.

The additional run-time processing overhead associated with the new display database traversal algorithm is due to the scan conversion of the back-projected planes of the scanlines. For parallel projection, all scanlines are parallel to one another; the inverse projection of a scanline, except the first one, is just a simple translation of the previous scanline and thus can be easily calculated. Therefore the additional run-time performance cost of our approach seems rather minimal. On the other hand, the benefit of eliminating the sorting between geometry and rasterization subsystems greatly improves the overall system pipeline efficiency, as well as significantly reduces the intermediate buffer memory requirements between geometry and rasterization subsystems.

4 Shading Computation

4.1 Lazy Shading

The goal of *lazy shading* is to ensure that the total amount of shading computation is proportional to the 2D screen space but is independent of the scene complexity, i.e., number of 3D triangles. The basic idea is to delay the computation of a pixel's color until the 3D primitive that is the dominant contributor to that pixel is determined. In other words, the classical Z-buffer algorithm says

```
Compute the Z-value and color of a 3D primitive
   P at (i, j): Z(P, i, j) and Color(P, i, j);
If Z(P, i, j) < Z(i, j) {
   Color(i, j) = Color(P, i, j);
   Z(i, j) = Z(P, i, j);
}
```

where $Z(i,j)$ and $Color(i,j)$ are the current Z-value and color of the pixel (i, j). The lazy shading version of the Z-buffer algorithm does the following instead.

```
Compute the Z-value of a 3D primitive P
   at (i, j), Z(P, i, j);
If Z(P, i, j) < Z(i, j) {
   Z(i, j) = Z(P, i, j);
   Color_Input(i, j) = Color_Input(P, i, j);
}

Traverse all the primitives that cover (i, j)
   have been traversed;
Color(i, j) = Compute(Color_Input(i, j));
```

where the $Color\_Input(P,i,j)$ is the set of arguments needed to compute the color value contributed by P to (i, j), and the *Compute*(.) function takes these arguments and calculates the color value. The difference between the above two schemes is that lazy shading delays the computation of a pixel's color until the responsible 3D primitive is determined. As a result, the color-value computation associated with the hidden parts of the 3D primitives is completely eliminated. Suppose each 2D pixel has, say four primitives covering it, this approach represents a factor of four reduction in shading computation. Because shading calculation accounts for a major portion of the computational efforts of the rasterization process, lazy shading provides a significant performance improvement at the system level. This effect is particularly drastic when more complex shading computation models are used, such as Phong shading or complicated texture mapping.

Although lazy shading decreases the amount of shading computation required, the volume of color-related data that needs to go through the Z buffer actually increases, because $Color\_Input(P,i,j)$ contains the color information of the span's two end points and therefore is at least twice the size of $Color(P,i,j)$. Larger data volume implies more pins (and thus expensive packaging) as well as wider and more complicated data paths inside the Z-buffering chip. Therefore, an important performance optimization to the lazy shading scheme is the adoption of *computation pointers*, place-holders that contain the pointers to the color computation's arguments rather than the arguments themselves. In *Heresy*, the computation pointer contains a span ID and the pixel's X coordinate. For Gouraud shading, the computation pointer is used to retrieve the color values at the end points of the specified span, and a linear interpolation is performed to derive the final color value on the designated pixel. Because computation pointers take smaller storage space than the color attributes, the amount of color-related data that have to pass into and out of the Z-buffer chip is significantly reduced, even compared to the non-lazy Z-buffering algorithm.

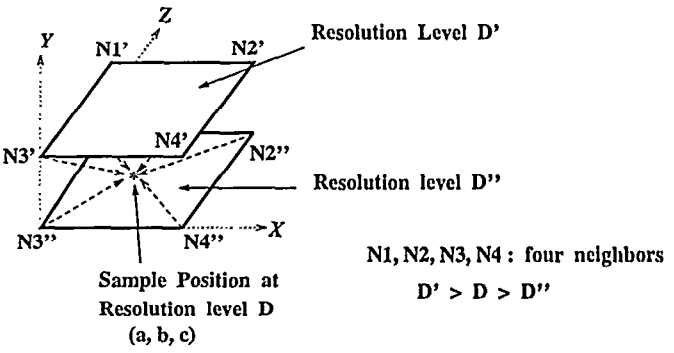

Figure 4: *The multi-resolution prefilter for texture anti-aliasing.*

Figure 3 shows the detailed data flow of the hardware implementation of lazy shading and computation pointers. The span processor maintains the list of active triangles and spans in the active span memory by performing edge walking. In addition, the span processor manages the name space of span ID's so that unused span ID's get recycled. The group linear interpolator performs Z-value interpolation for pixels on a span and is able to emit one Z-value per cycle after initial startup. Note that only the X and Z coordinate and the span ID of a pixel actually passes through the Z buffer logic. To compute the final color value of a pixel, the shader requests from the active span memory the detailed color information, $Color\_Input(P,i,j)$ (i.e., X, R, G, B, and α) with the pixel's associated span ID. Because only Z values are needed for depth check, the span processor can execute the color-related part of edge walking in parallel with Z-value comparison, further reducing the rasterization latency without full-scale parallel implementations. In summary, while the shader renders the pixels of the i-th scanline, the Z-buffer module performs visibility computation for the $(i+1)$-th scanline. The width of span ID in *Heresy* is 16 bits. That is, there are at most 64K triangles that can cover a given scanline. The high-level triangulation procedure will ensure that the applications don't violate these constraints. In the highly unlikely event that there are more than 64K spans covering a scanline, the rasterization for that scanline is split into multiple phases, each of which is responsible for a segment of the scanline so that the number of covering triangles is smaller than 64K.

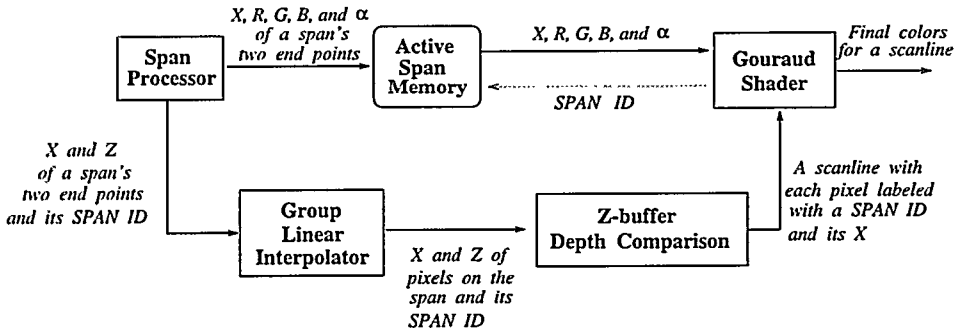

Figure 3: *The detailed data flow in the implementation of lazy shading.*

4.2 Texture Mapping

Heresy supports Williams' *mip-mapping* scheme [11] to minimize the aliasing artifacts due to 2D texture mapping. In this scheme, different resolutions of a 2D texture map are organized into a 3D database. Given a screen space pixel, the corresponding sample point in the texture domain is first identified. Then the four neighbors of the sample position and an additional parameter D, called the resolution level, is computed. As shown in Figure 4, the color value of the pixel in question is determined by performing a 3D interpolation of the eight texels that encompass the sample point in the 3D texture database. Interestingly enough, the same 3D interpolation mechanism can also be used to support 3D texture mapping. Therefore, in this subsection we will focus on a high-performance hardware implementation of tri-linear interpolation used in *Heresy*.

Suppose the relative 3-D coordinate of a sample point within a cube with respect to the corner voxel closest to the origin is $< a, b, c >$, and the data values associated with the corner voxels of the cube, P_{ijk}, where i, j, k = 0 or 1, then the interpolated data value associated with the sample point, P_{abc}, is computed through a tri-linear interpolation as follows:

$$\begin{aligned} P_{abc} = {} & P_{000} * (1-a)(1-b)(1-c) + P_{100} * a(1-b)(1-c) \\ & + P_{010} * (1-a)b(1-c) + P_{001} * (1-a)(1-b)c \\ & + P_{110} * ab(1-c) + P_{101} * a(1-b)c + \\ & P_{101} * a(1-b)c + P_{011} * (1-a)bc \end{aligned} \quad (1)$$

A brute-force implementation of this formula requires 13 multiplications and 20 additions.

The first key idea for the fast 3-D interpolation unit design is to replace time-consuming arithmetic operations with table look-up. From Equation (1), it is clear that the only part that permits pre-computation is the intermediate values involving a, b, and c. Assume that a, b and c are represented with a seven-bit resolution, the number of possible combinations for $< a, b, c >$ triples becomes $2^7 * 2^7 * 2^7 = 2^{21}$. For each triple, they are eight intermediate values, each being 8-bit wide. Thus the total size of the look-up table is 16 MBytes. Simply because of the required memory size, this design is clearly too expensive and potentially slow. We solve the above problem by making the observation that a tri-linear interpolation is actually equivalent to a linear interpolation following two bi-linear interpolations. A bi-linear interpolation assumes the following form:

$$P_{ab} = P_{00} * (1-a)(1-b) + P_{10} * a(1-b) + P_{01} * (1-a)b + P_{11} * ab \quad (2)$$

By substituting two bi-linear interpolations and a linear interpolation for a tri-linear interpolation, the look-up table size is now shrank to 64 KBytes. The price we pay for this design decision is that two more multiplications are needed than the straightforward tri-linear interpolation design. More specifically, each N-linear interpolation needs 2^N multiplications and $2^N - 1$ additions, where linear, bi-linear, and tri-linear corresponds to an N value of 1, 2, and 3, respectively. Therefore, two bi-linear interpolations and one linear interpolation take ten multiplications and seven additions. Fortunately, the performance overhead associated with these additional multiplications can be minimized by exploiting parallelism and pipelining.

Now that the intermediate values involving relative coordinates are available by table lookup, the actual computation of 3-D interpolation is reduced to those operations involving sample data values. The second key idea of the fast 3-D interpolation unit design is to exploit the internal structure of a parallel multiplier. To a first approximation, a parallel multiplier is nothing more than a two-dimensional array of single-bit *carry-save adders*. Therefore, it is possible to integrate a multiplication and an addition operation by inserting an extra row of carry-lookahead adders. Moreover, one can pipeline multiple multiply-add operations through such an augmented parallel multiplier to reduce the hardware cost. Consequently, it becomes feasible to implement the entire 3D-interpolation function in one chip without resorting to exotic technologies like multi-chip modules or wafer scale-integration.

The system architecture of the fast 3D-interpolation unit is shown in Figure 5. Two parallel multipliers are used to execute two bi-linear interpolations in parallel. At the end of bi-linear interpolation, the results are fed to the third parallel multiplier to execute the finishing linear interpolation. The third multiplier is included to allow pipelining among multiple tri-linear interpolation operations. In this design multiple multiply operations can occupy different sections of a parallel multiplier, and the interface between the operand register file and the array multiplier is tightly integrated.

Assume that the result from table look-up is 8-bit wide. With a modified 2-bit Booth encoding, a multiplication with a 8-bit multiplier involves four partial products, or three additions to complete. The detailed architecture for the augmented parallel multiplier is shown in Figure 6, where each CSA stands for a carry-save adder for accumulating intermediate partial products, and the R_k's are shift registers that form the partial products. For explanation purposes, we will focus on the bi-linear interpolation of the following form:

$$P_1 * C_1 + P_2 * C_2 + P_3 * C_3 + P_4 * C_4 \quad (3)$$

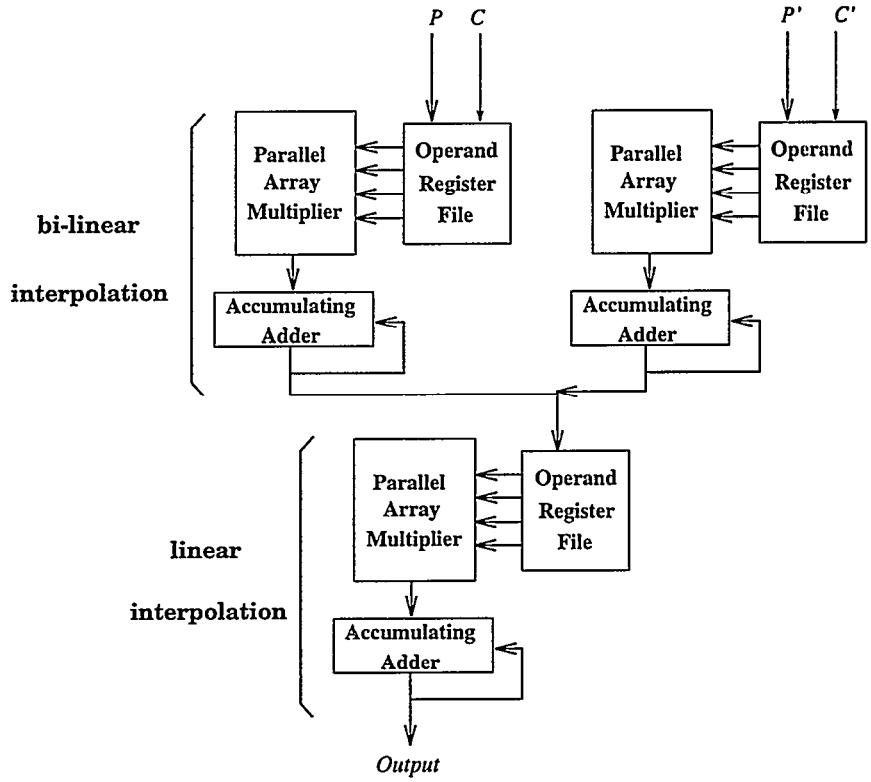

Figure 5: *The system architecture for the 3D-Interpolation unit.*

where P_i's pre-stored sample data values and C_i's the 8-bit interpolation coefficients from table look-up. The partial product shifter control subsystem takes C_i's as multipliers and controls the shift registers R_i's to generate multiples of corresponding P_i's as partial products. In *Heresy* the look-up table actually stores the Booth encoding of C_i's rather than the actual values of C_i's. So there is no run-time overhead for Booth encoding, saving both hardware complexity and time.

To start a bi-linear interpolation computation, only P_1 and C_1 need to be on the chip. The rest can be fetched sequentially as the computation unfolds. In the following, we will use PP_{ij} to denote the j-th partial product from the multiplicand P_i, j = 1, 2, 3, or 4. Also assume each multiplier C_i contains four 2-bit patterns, each of which is denoted by C_{ij}, where j = 1, 2, 3, or 4. Therefore,

$$P_i * C_i = \sum_{k=1}^{4} PP_{ij} \tag{4}$$

$$PP_{ij} = (P_i << 2 * j) * C_{ij} \tag{5}$$

In the beginning, PP_{11} and PP_{12} occupy R1 and R2, respectively. At the end of the first cycle, $PP_{11} + PP_{12}$ is available from the output of the first CSA, and at the same time PP_{21}, PP_{22}, and PP_{13} are stored in R1, R2, and R3, respectively. At the end of the second cycle, $PP_{11} + PP_{12} + PP_{13}$ and $PP_{21} + PP_{22}$ are available from the second and first CSA, respectively, and now PP_{31}, PP_{32}, PP_{23}, and PP_{14} occupy R1, R2, R3, and R4, respectively. At the end of the third cycle, $P_1 * C_1$ is available from the third CSA and is fed into the accumulation carry-lookahead adder. The

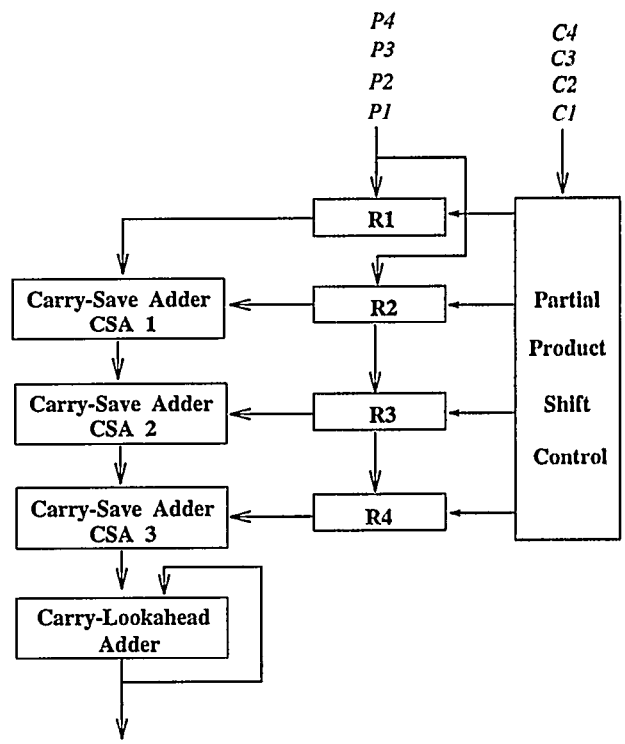

Figure 6: *Detailed hardware structure to perform bi-linear interpolation. The partial products PP_{ij} in R_k's are formed by appropriately shifting/complementing P_i's controlled by C_{ij}'s.*

Cycle	R1	R2	R3	R4	CSA 1	CSA 2	CSA 3	Final CLA
1	PP_{11}	PP_{12}						
2	PP_{21}	PP_{22}	PP_{13}		$PP_{11}+PP_{12}$			
3	PP_{31}	PP_{32}	PP_{23}	PP_{14}	$PP_{21}+PP_{22}$	$\sum_{i=1}^{3} PP_{1i}$		
4	PP_{41}	PP_{42}	PP_{33}	PP_{24}	$PP_{31}+PP_{32}$	$\sum_{i=1}^{3} PP_{2i}$	$P_1 * C_1$	
5			PP_{43}	PP_{34}	$PP_{41}+PP_{42}$	$\sum_{i=1}^{3} PP_{3i}$	$P_2 * C_2$	$P_1 * C_1$
6				PP_{44}		$\sum_{i=1}^{3} PP_{4i}$	$P_3 * C_3$	$\sum_{k=1}^{2} P_k * C_k$
7							$P_4 * C_4$	$\sum_{k=1}^{3} P_k * C_k$
8								$\sum_{k=1}^{4} P_k * C_k$

Table 1: *The contents of each module in the bi-linear interpolation unit at the beginning of each cycle.*

detailed timing for bi-linear interpolation on the augmented parallel multiplier is shown in Table 1.

As can be seen, it takes seven internal cycles to complete a bi-linear interpolation. Similarly, the finishing linear interpolation will take 5 cycles. Therefore totally 12 internal cycles are needed for the 3-D interpolation function. For the total delay to be under 100 ns, the internal cycle time needs to be under 8 ns, which is relatively straightforward to achieve, considering that each carry-save adder only involves less than five gate delays. The multiplicands, i.e., P_i's, are assumed to be 50-bit wide to accommodate R, G, B, and α simultaneously. As for the silicon cost, the parallel multiplier array accounts for the major portions of the 3D-interpolation unit. We estimate that each single-bit full adder takes under 100 transistors (including routing areas), then the parallel multiplier and the accumulating adder takes 20,000 transistors. Assume that the operand register file and the control logic takes the same amount of transistors. Then the total transistor count for the 3D-interpolation unit is roughly $3*2*20{,}000$, or 120K, a rather conservative design point even based on the 1.2 micron MOSIS service.

5 Speculative Z-buffering

In the previous sections, we show how *Heresy* eliminates intermediate sorting and unnecessary rendering computation due to invisible primitives. Because depth comparison needs to be done for every pixel on every primitive, the remaining system bottleneck most likely lies in the Z buffer logic. Since *Heresy* is based on a virtual image-space rasterization architecture, the amount of Z-buffer memory is smaller than its object-space counterpart. Other than the obvious cost benefit, there are two fundamental performance advantages associated with this approach. First, smaller Z-buffer memory requirement encourages direct integration of depth comparison logic with Z-buffers on the same chip, which significantly increases the speed of Z-buffer access. Second, smaller Z-buffer memory requirement also allows a higher-performance though more expensive implementation of each Z-buffer element. In this section, we will present a Z-buffer

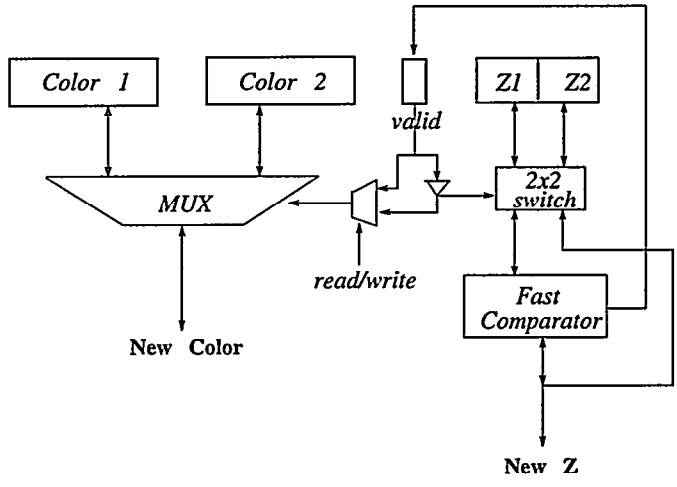

Figure 7: *The hardware design of a speculative Z buffer cell*

cell design that performs each Z-value comparison in one cycle, rather than at least four cycles in traditional designs.

As explained in the last section, the vanilla Z-buffer algorithm requires at least four instructions: a memory access to fetch the current Z value, a subtraction between the computed Z value and the current Z value, a conditional branch, and if success another memory access to store the new Z and possibly color values. Because there are conditional branches as well as multiple memory access instructions that pound at the same resource, pipelining techniques can not be easily applied to improve the system throughput. Since Z-value comparison is performed for every pixel on every 3D primitive, the computational efforts are proportional to the scene complexity. Therefore, fast Z-buffer hardware implementation plays a critical role in improving the overall performance of 3D graphics rendering.

The proposed Z buffering hardware implementation in *Heresy* is shown in Figure 7, which represents an individual cell. The key idea of the proposal is to shadow the Z buffers

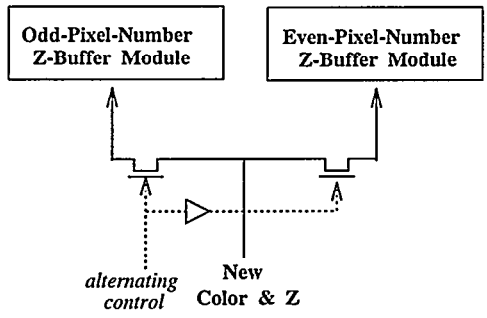

Figure 8: *The alternating partitions of the Z buffer logic module, which contains only one scanline.*

so that newly computed Z and color values[1] can be stored *speculatively*. In other words, each logical Z buffer cell is physically represented in two memory cells. To distinguish the most up-to-date version of a particular pixel's Z and color values, a *valid* bit is used, which is computed as a result of comparing new Z values and current Z values. As a result, the conditional branch in the generic Z buffering algorithm is turned into a manipulation of the *valid* bit.

Given a pixel coordinate, the corresponding *valid* bit is first fetched to determine the shadow version, into which the newly computed Z and color values are stored. At the same time, the current version of the pixel's Z value is fetched and compared with the new Z value. Through a customized fast comparator, the pixel's *valid* bit is modified based on the result of the comparison. The *valid* bit not only distinguishes between the current and shadow versions of the pixel's Z value, but also controls the read/write modes of the two memory cells: The shadow version performs a write while the current performs a read.

The 2x2 bi-directional switch establishes the connection between the Z memory banks and the fast comparator and the new Z register. The fast comparator is a customized circuit that toggles the valid bit if and only if when the new Z value is less than the current Z value.

The critical path of the proposed Z-buffer hardware is the following: fetch the *valid* bit, load the current Z value, perform Z-value comparison, and modify the *valid* bit. Note that the pre-charging and decoding delay of the three memory accesses, Z, color value, and valid bit, are completely overlapped. At the target clock rate of 100 MHz, these four operations can be safely performed within two cycles.

To further reduce the Z-value comparison latency, *Heresy* exploits the fact that a span consists of a contiguous sequence of pixels by decomposing the Z buffer for a scanline into two partitions: one for even-numbered pixels and the other for odd-numbered pixels, as shown in Figure 8. Because span processing tends to touch pixels consecutively, the two partitions service the Z-buffer accesses alternately. At any given cycle the even partition may be at the first stage of the Z-value comparison pipeline, and the odd partition is at the second stage; at the next cycle, the pipeline stages of these two partitions switch. Consequently, the system's overall throughput is at one Z-buffered comparison per clock cycle, a factor of four improvement over conventional designs.

[1]Because of lazy shading, they are actually computation pointers. Without loss of generality, let's assume they are color values.

6 Preliminary Performance Analysis

The performance goal of a full-scale *Heresy* implementation is 1 million Gouraud shaded, texture-mapped, and anti-aliased triangles per second. In this section, we analyze the performance of each pipeline stage of *Heresy*, and investigate how additional hardware parallelism can be introduced to achieve the performance goal.

In *Heresy*, the geometric transformation module is implemented by a high-speed DSP processor such as TI's MVP [7], which has an estimated performance of 500K triangles per second. The active span processor is implemented by a special ASIC that performs edge walking and the maintenance of SPAN ID name space and active span memory. An ASIC implementation is chosen because *forward differencing* is used to compute the interpolated X, Y, Z, R, G, B, and α values along the triangle edges. This requires large on-chip memory to maintain the intermediate state of each active triangle and multiple adders to emit the coordinates and colors of the end points of a span. Assume that the start-up delay for an interpolation operation is 5 cycles and one cycle per interpolation thereafter, and a triangle in average consists of 15 spans. Then the average edge walking delay per triangle is

$$\frac{7*2*5+15*1}{15} = 5.67(cycles)$$

where 7 denotes the seven interpolations and 2 the two end points of each span. With a 50 MHz clock rate, the span processor can complete 588K triangles per second.

The group linear interpolator in Figure 3 computes the Z values of the pixels on each span. This module is again implemented by an ASIC, which can perform four linear interpolations per cycle under a 50 MHz clock. Given the same interpolation delay assumption, then this ASIC can perform $\frac{50*4}{(5+10*1)*15}$ = 889K triangles per second. where we assume there are in average 10 pixels per span. The speculative Z-buffer module is another ASIC that implements depth comparison. As mentioned in Section 5, this chip runs at 100 MHz and is capable of performing one Z-value comparison per cycle with a 2-cycle start-up overhead. So the speculative Z-buffer module perform $\frac{100}{(2+10*1)*15}$ = 556K triangles per second. The Gouraud shader computes the final R, G, B, and α values and can perform four linear interpolations in parallel in five cycles. At 50 MHz, the number of pixels that this shader can compute is 10 million, or 10 frames/sec at 1K x 1K resolution per frame. As mentioned in Section 4.2, the texture mapping module can perform one 3D interpolation per 100 ns, or 10 million 2D mip-mapped or 3D texture mapped pixels per second. In terms of bandwidth requirements, the path that gets stressed most is between the group linear interpolator and the scanline Z-buffer module. Due to the use of computation pointers, the input data rate to a Z-buffer module is reduced by a factor of four to $556Ktriangles/sec * 150pixels/triangles * 5bytes/pixel = 417Mbytes/sec$, because for each pixel only its X and Z coordinate needs to be passed into the Z-buffer. With the advance of semiconductor process technology, we expect that the group linear interpolator and the scanline Z-buffer module will eventually be integrated in one chip, thus eliminating this bandwidth problem altogether.

The architecture of *Heresy* is inherently scalable by replicating the hardware modules. To double the performance, one simply replicates another *Heresy* pipeline, and makes sure that the host distributes the 3D primitives evenly between the two pipelines. The reason for this scalability is

that the rasterization-oriented display database traversal algorithm pushes synchronization to the very beginning of the pipeline, and therefore maximizes the extent of hardware parallelism.

7 Related Works

There are several graphics machines that are based on image-space architecture. The most powerful commercial offering is SGI's RealityEngine [1], which dedicates one rasterization engine to each image region and thus bypasses the intermediate sorting delay problem. UNC's Pixel-Planes 5 [5] is also based on parallel image-space architecture. Therefore bucket sorting of transformed primitives must be completed before parallel rasterization can start. In some sense UNC's Pixel-Flow [10] is the antithesis of *Heresy* in that PixelFlow delays synchronization to the end of the rendering pipeline via a compositing network, whereas *Heresy* forces synchronization at the beginning of the pipeline. PixelFlow also supports the notion of lazy shading. [6] described a virtual image-space architecture called *SAGE* that is based on the systolic computing paradigm. Again, it requires intermediate bucket sorting. [8] described a prototype based on the scanline Z buffer architecture, which is essentially the same as *Heresy*'s architecture. However, this system still needs a sorting phase between rasterization and geometry subsystems. [3] discussed a system built at SUN called Leo, which adopted a parallel image-space architecture. As a result, the Z-buffer cannot be integrated with the rasterization engine one one chip as *Heresy* does. Eventually the author reported that the system performance bottleneck is imposed by VRAM access speed. [4] proposed a new memory architecture called FBRAM that attempts to address this problem.

8 Conclusion

This paper describes *Heresy*, a high-performance 3D graphics rendering system currently under development at our institution. *Heresy* supports lazy shading in hardware, implements a one-cycle speculative Z-buffer comparison logic, incorporates a highly efficient table-lookup-based 2D/3D texture mapping mechanism, and finally features an innovative display database traversal algorithm that completely eliminates the intermediate sorting associated with virtual image-space rendering architecture. We estimate that by replicating the *Heresy* pipeline, the overall performance of the system can reach over 1 million Gouraud-shaded and 2D mip-mapped triangles per second at 20 frames/sec with 1K x 1K resolution per frame.

Currently we are working on a function-level simulator of *Heresy* for architectural performance evaluation. The ongoing efforts of our group will continue with two directions. One is to perform a more comprehensive workload characterization of 3D graphics programs, especially in the domain of interactive games and distributed simulation/training. The other is to complete the VLSI circuit design of each of the five ASIC chips described in this paper.

Acknowledgement

This research is supported by an NSF Career Award MIP9502067 and a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program.

9 References

[1] K. Akeley, "RealityEngine Graphics," *Proceedings of the 20th SIGGRAPH Conference,* pp. 109-116, Anaheim, California, August 1993.

[2] D. Cohen, *3D Scan Conversion of Geometric Objects,* PhD Thesis at the Computer Science Department of SUNY at Stony Brook, December 1991.

[3] M. Deering, S. Nelson,"Leo: A System for Cost Effective 3D Shaded Graphics," *Proceedings of the 20th SIGGRAPH Conference,* pp. 101-108, Anaheim, California, August 1993.

[4] M. Deering, S. Schlapp, M. Lavelle, "FBRAM: A New Form of Memory Optimized for 3D Graphics," *Proceedings of the 21th SIGGRAPH Conference,* pp. 167-174, Orlando, Florida, July, 1994.

[5] H. Fuchs, et al., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Proceedings of the 16th SIGGRAPH Conference,* pp. 79-88.

[6] N. Gharachorloo, et al., "Subnanosecond Pixel Rendering with Million Transistor Chips," *Proceedings of the 15th SIGGRAPH Conference,* pp. 41-49.

[7] K. Guttag, rt al., "A Single-Chip Multiprocessor for Multimedia: the MVP." IEEE COMPUTER GRAPHICS AND APPLICATIONS (Nov. 1992) vol.12, no.6, pp. 53-64.

[8] M. Kelly, S. Winner, K. Gould, "A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm," *Proceedings of the 19th SIGGRAPH Conference,* pp. 241-248, Chicago, Illinois, July 1992.

[9] S. Molnar, H. Fuchs, "Advanced Raster Graphics Architecture," Chapter 18 of *Computer Graphics: Principles and Practice* by J. Foley, A. van Dam, S. Feiner, and J. Hughs, 1990.

[10] S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Proceedings of the 19th SIGGRAPH Conference,* pp. 231-240, Chicago, Illinois, July 1992.

[11] L. Williams, "Pyramidal Parametrics," *Proceedings of the 10th SIGGRAPH Conference,* pp. 1-11.