processors execute these instructions at once, the lower execution times of some of the integer operations make them very attractive.

Operation	2-byte short	4-byte long	4-byte float
Addition	0.07 µs	0.13 µs	3.94 µs
Multiplication	0.50 µs	2.00 µs	2.53 µs
Division	1.60 µs	6.40 µs	7.04 µs
Square Root	1.22 µs	3.33 µs	6.98 µs

Figure 6: Execution time of integer versus floating-point instructions.

Conversion from floating point to 4-byte integer format takes 1.35µs, and from 4-byte integer to floating point takes 1.57µs. This makes it feasible to convert representations to use whichever is more advantageous. Whenever possible, we use fixed-point or integer representations.

Memory. Each processor has 256 bytes of local memory and 128 bytes of communication register that may also be used as local memory. Each node can store 16MB of texture information in table lookup memory. This memory may be read or written from each of the pixel processors, thus serving as global storage.

3.2 Achieving interactive shading

Each PixelFlow node possesses an enormous amount of computational power—over 40 billion integer operations or 2 billion floating-point operations per second. In addition, the processors are programmable in a very general way, and we believe that the 256 (+128) bytes of local memory at each processor is sufficient to implement many interesting shading algorithms. However, even this amount of computational power is not enough to achieve our goal of real-time shading. We must harness multiple PixelFlow nodes in an efficient manner to multiply the power available for shading.

PixelFlow rasterizes images using a screen-subdivision approach, sometimes called a *virtual buffer* [11]. The screen is divided into 128x64-pixel regions, and the regions are processed one at a time. When the rasterizers have finished with a particular region, they send appearance parameters and depth values for each pixel onto the image-composition network, where they are merged and loaded into a shader.

If there are s shaders, each shader receives one of every s regions. While it shades the region, it has full use of the local memory at each pixel processor. With this method of rendering, even a small machine can support an arbitrary sized screen. Of course, the more complex the problem, the more nodes that are needed to achieve interactive performance.

Deferred shading. As stated in Section 2.3, deferred shading is a powerful optimization for scenes of high depth complexity. It has an even bigger payoff for a SIMD architecture such as PixelFlow. We implement deferred shading on a machine-wide basis by giving each node a designated function: rasterization or shading. The rasterization nodes implement the first loop in Figure 3b, while the shading nodes implement the second.

As specified in Figure 3b, the rasterization nodes scan convert the geometric primitives in order to generate the necessary appearance parameters. Multiple rasterization nodes can work on a single region of the screen as described by Molnar, et. al. [12]. The composition network collects the rasterized pixels for a given region (including all necessary appearance parameters), and delivers it to the shading node that has been assigned to process that region.

Deferred shading provides an additional computational advantage on PixelFlow because of the SIMD nature of the pixel processors. Consider how a SIMD machine might behave if shading is performed during rasterization (immediate shading—Figure 3a). For each primitive, the processors representing the pixels within the primitive are *enabled*, while all of the others are *disabled*. The subsequent shading computations are performed only for the enabled pixels. The processors representing pixels outside of the primitive are disabled, so no useful work is performed.

Since most primitives cover only a small area of the screen, we would make very poor use of the processor array. The key to making effective use of the SIMD array is to have every processor do useful work as much of the time as possible.

With deferred shading, all of the pixels in a region that require the same shader can be shaded at one time, even if they came from different primitives. This is especially useful when tessellated surfaces are used as modeling primitives. These can be rasterized as numerous small polygons but shaded as a single unit. In fact, disjoint surfaces can be shaded at once if they use the same shading function.

Factoring out common calculations. We can go even further than executing shading functions only once per region. Shading functions tend to be fairly similar. Even at a coarse level, most shading functions at least execute the same code for the lights in the scene even if their other computations differ. All of this common code need only be done once for all of the pixels that require it. As illustrated in Figure 7, if each shading function is executed to the point where it is ready to do lighting computations, the lighting computations for all of them can be performed at once. The remainder of each shading function can then be executed in turn.

```
// Shader-specific code
for each surface shader
   pre-light shading;

// Common code
for each light source
   accumulate illumination;

// Shader-specific code
for each surface shader
   post-light shading;
```

Figure 7: Factoring out common operations for multiple shading functions.

Currently, we code this manually, but this is yet another reason to have a high-level compiler. A suitably intelligent compiler can identify expensive operations (such as lighting and texture lookups) among several shading functions and automatically schedule them for co-execution.

Table lookup memory. Each shader node has its own table-lookup memory for textures but, since it is not possible to know which textures may be needed in the regions assigned to a particular node, the table memory of each must contain every texture. For interactive use this not only limits the size of the textures to the maximum that can be stored at one node, but it also presents a problem for shadow map and environment map algorithms that may generate new textures every frame. After a new map is computed, it must be loaded into the table-lookup memories of every shader node. This aspect of system performance does not scale with the number of nodes: a maximum of 100 512x512

texture maps can be loaded into table-lookup memory per second (2-3 in a 33 ms frame time).

Uniform and varying expressions. For efficiency, expressions containing only uniform shader variables (those that are constant over all of the pixels being shaded) are computed only once on the RISC GP. Varying expressions (those that vary across the pixels), or those containing a mix of uniform and varying variables, are executed on the pixel-processor array.

Shader parameters. There are two ways to communicate parameters to a shader node. One is to send the parameters over the composition network. The other is to send the parameters over the front-end geometry network. Obviously, a varying parameter that must be interpolated over the pixels, such as color or surface normal, is produced on a rasterization node, and should be sent over the composition network.

A uniform parameter that is used at the GP and does not vary from primitive to primitive should be sent over the geometry network because composition network bandwidth is a valuable resource. An example is something like the roughness of a surface which is a fixed parameter for a particular material. If the parameter is needed in the local memory of the pixel processors, it can be broadcast locally at a shading node. We allow the programmer to choose the best way to transmit each parameter.

3.3 Shader programming model

Low-level model. Since instructions for the pixel processors are generated by the GP on a PixelFlow node, the code that a user writes is actually C or C++ code that executes on the GP. The low-level programming model for the pixel processors (called *IGCStream*) consists of inline functions in C++ that generate code for the SIMD array. Some of these functions generate the basic integer operations; others, however, generate sequences of instructions to perform higher-level commands, such as floating-point arithmetic.

We have written a library of auxiliary shading functions to use with this programming model. It provides basic vector operations, functions to support procedural texturing [5, 13], basic lighting functions, image-based texture mapping [14], bump mapping [15], and higher-level procedures for generating and using reflection maps [16] and shadow maps [17, 18]. It is perfectly feasible to program at this level. In fact, we currently use this programming model to write code for testing, and to produce images such as those in the example video. We would prefer, however, to work at a higher, more abstract level.

High-level model. We are implementing a version of the RenderMan shading language that is modified to suit our needs. Our goal in using a higher-level language is not solely to provide architecture independence. That may be useful to us in the future, of course, but since PixelFlow is an architectural prototype it is not necessary. We are more interested in the shading language as a way to demonstrate feasibility and to provide our users with a higher-level interface that they've had [19] in order to encourage wide use of the shading capabilities of our system. Also, as mentioned earlier in this section, a high-level shading language provides opportunities for compiler optimization, such as co-executing portions of several shader functions.

The RenderMan specification has only float, point, and color arithmetic data types. Since we need to be frugal in our use of floating-point arithmetic, we have added integers and fixed-radix-point numbers to the data types of our language. A compiler for the shading language will accept shader code as input, and emit C++ with SIMD processor commands as output. This code will be linked with the auxiliary shading function library and finally with the application program.

API support. We also need some way for graphics applications to access our shading capability. Since one of our main goals for PixelFlow is interactive visualization of computations as they are executing on a supercomputer, we have chosen an immediate-mode application programmer's interface (API) similar to OpenGL [20]. An advantage of choosing OpenGL, and extending it to meet our needs is that students and collaborators are likely to be familiar with the its basic concepts Also, this will make it easier to port software between PixelFlow and other machines.

The current specification of OpenGL only incorporates the limited set of shading models commonly found on current graphics workstations: flat and linearly interpolated shading with image-based textures. We have extended the specification to allow users to select arbitrary shaders.

We do not plan to implement an official, complete OpenGL for two reasons. One is that some of the specifications of OpenGL conflict with our parallel model of generating graphics. The second is that we lack the resources to implement features that we do not use. Consequently, though our functions are similar to OpenGL, we use a *pxgl* prefix instead of OpenGL's *gl* prefix. Within these constraints, we have attempted to stick as closely as possible to the OpenGL philosophy. We intend to describe this API, and the problems involved in implementing it on PixelFlow, in a future publication.

Limitations. Although the PixelFlow shading architecture supports most of the techniques common in "photorealistic rendering," (at least in RenderMan's use of the term), it has a few limitations. Because PixelFlow uses deferred shading, shaders normally do not affect visibility. Special shaders can be defined that run at rasterization time to compute opacity values. However, these shaders poorly utilize the SIMD array and slow rasterization.

A second limitation is that shaders cannot affect geometry. RenderMan, for example, defines a type of shader called a *displacement shader*, which displaces the actual surface of a primitive, rather than simply manipulating its surface-normal vector, as is done in bump mapping. This is incompatible with the rendering pipeline in PixelFlow, as well as that of virtually all other high-performance graphics systems.

4 EXAMPLE

In this section, we present a detailed example of real-time high-quality shading on PixelFlow. The example—bowling pins being scattered by a bowling ball—was inspired by the well-known "Textbook Strike" cover image of the *RenderMan Companion* [6]. We cannot guarantee that the dynamics of motion are computable in real-time, but we are confident that a modest-sized PixelFlow system (less than one card cage) can render the images at 30 frames per second.

The accompanying video was rendered on the PixelFlow functional simulator. The execution times are estimates based on the times of rasterization and shading of regions, using worst-case assumptions about overlap. We simulated a PixelFlow machine containing three rasterizer nodes, twelve shading nodes, and a frame-buffer node. There are 10,700 triangles in the model. The images were rendered at a resolution of 640x512 pixels with five-sample-per-pixel antialiasing.

4.1 Shading functions

Three shading functions are used to render these images, one for the bowling pins, one for the alley, and one for the bowling ball. Two light sources illuminate the scene, an ambient light and the main point-light source which casts shadows in the environment.

Parameter	Number of bytes
Depth	4
Shader ID	1
Intrinsic Color	1 x 3
Normals	2 x 3
Texture coordinates, *u, v*	2 x 2
Texture gradients	2 x 2
d*P*/d*u*, d*P*/d*v*	2 x 6

Figure 8: Appearance parameters used in bowling example.

Figure 8 shows the data for each pixel that is sent from a PixelFlow rasterizer node to a shader node, a total of 34 bytes. We actually plan to use 10 bits of color per channel on most PixelFlow applications, but 8 bits were used for this simulation. In addition to the appearance parameters used by the shaders, two other parameters are necessary, the depth and a shader identification number for each pixel. The shader ID is used by the shading control program to select the shader code for each pixel.

The bowling ball has a shadow-mapped light source with a Phong shader. The alley has a shadow-mapped light source, reflection map, mip-mapped wood texture, and a Phong shader. The pins have a shadow-mapped light source, procedural crown texture, mip-mapped label, bump-mapped scuffs, mip-mapped dirt, and finally a simple Phong shader. We factor out common lighting computations as described in Section 3.2. Each shader is divided into three parts, the part before the lighting computation, the common lighting computation, and the part after the lighting computation.

4.2 Multiple-pass rendering

The shadow and reflection maps are obtained during separate rendering passes. When each of these 512x512 images has been computed and stored, rendering of the final image begins. In this section we describe, in detail, the steps necessary to render and store the shadow map and to render the final camera-view image. Since computation of the reflection map is similar, we do not describe it in detail.

Shadow map. A shadow map is a set of depth values rendered from the point of view of the light source. We use three rasterizer nodes to rasterize all the primitives and compute the depth at each sample point. Since we do not need to calculate colors or other parameters, this is a simple computation. The worst-case time for this step is approximately 100 µs, although many map regions have very few polygons and take less time to rasterize.

The depth values are then z-composited over the composition network, and the resulting depth is sent to all of the shaders. Composition time is only 5 µs per region. Notice that data transfer and computation can proceed simultaneously.

As mentioned in Section 3.2, storing tables for shadow or reflection mapping is a point of serialization on our system. The combined time to store both the shadow and reflection map takes almost half the time for each frame. Since the hardware can store four values into table memory at one time, we take advantage of this intra-node parallelism by storing the depth map in units of four regions each. Thus, the shader nodes accept four regions of data before storing them.

The total time to complete the shadow map pass is the time consumed by eight table writes, 6.08 ms, plus the time to rasterize the first four regions, for a total time of less than 7 ms.

Reflection Map. Rasterization for the reflection map can begin as soon as enough buffer space is available at the rasterization nodes. Shading for the reflection map can begin as soon as the last table write for the shadow map has begun. The reflection map can be generated and stored in less than 7 ms.

Final Image. The rendering time for the final image is a function of both the rasterization time and the shading time. If the time to rasterize a region is longer than the time to shade it, the shading nodes will be idle waiting for appearance parameters from the rasterizer nodes. The worst-case time will then be the total rasterization time plus the time to shade the final region. If the time to rasterize a region is less than the time to shade it, the shading nodes will always have regions waiting to be shaded. We will see that for this scene shading is the bottleneck, so the rendering time will be the total shading time plus the time to rasterize the first few regions (to get the shading nodes started).

First, consider the rasterization time. With all of the appearance parameters, each of the front-facing triangles in the model takes approximately 0.85 µs to rasterize. One of the busiest frames, with all of the pins visible, contains just under 6400 front-facing triangles (this includes the additional triangles that have to be rendered when triangles cross region boundaries). This total takes 5.4 ms to complete on one rasterizer node. If we also do five sample antialiasing, this becomes 27 ms. To achieve our performance goal we divide the polygons over 3 rasterizers to decrease the time to a little over 9 ms. Details on the use of multiple rasterizers in PixelFlow can be found in [12].

Section of code		Shading function		
		Pin	Alley	Ball
pre-light	procedural crown	2 µs		
	mip-map label	15 µs		
	bump-map scuffs	24 µs		
	mip-map dirt	15 µs		
	mip-map wood		15 µs	
light	shadowing	← 28 µs →		
post-light	Phong shader	12 µs		
	reflection		27 µs	
	Phong shader			12 µs

Figure 9: Shading times (1 node, 1 sample, 1 region) excluding table lookup.

Now, consider the shading time. In PixelFlow, the table lookup time is proportional to the number of pixels that need data, so it is not constant for a region but depends on how many total values are actually needed. The worst case for table lookup will occur if all of the pixels in a region use the bowling pin shading function since it needs to look up four different values: two mip-mapped image textures, one bump map, and one shadow map. To do one table lookup for all 8K pixels on a node takes 190 µs, so looking up four values for a full region requires 760 µs.

The worst-case time for the rest of the shader processing occurs for regions that require all three shading functions, bowling pin, alley, and ball. For regions without all of these elements, only

some of the shading functions need to be run. Figure 9 shows the processing time for the shading functions excluding the table lookup times. Note, however, that the time setting up for a lookup and using the results *is* included. The slowest time for a region is the sum of all the times in the figure or 150 μs.

This time is for only one sample of one region. Since we are doing five samples and a 640x512 video image has 40 regions, there are really 200 regions to shade. The total time comes to 182 ms. By distributing the shading among twelve shading nodes, we can cut the worst-case shading time to about 15.2 ms.

The 9 ms spent rasterizing is less than the shading time. Therefore, the shading time dominates. The total time to compute the final camera view is the shading time plus the time to rasterize the first regions, or about 15.7 ms.

Total frame time. A complete image can be rendered in under 29.7 ms. This includes 7 ms to generate a shadow map, 7 ms to generate a reflection map, and 15.7 ms for the final camera image. These times were computed with pessimistic assumptions and without considering the pipelining that occurs between the rendering phases. This results in a frame rate faster than 30 Hz. With more hardware it will be possible to run even faster.

Additional hardware will not significantly speed the shadow or reflection map computations since they are dominated by the serial time spent writing the lookup tables. But rendering time of the camera image is inversely proportional to the number of rasterization and shading nodes. For more complex geometry, we add rasterization nodes. For more complex shading, we add shading nodes. Note that the hardware for both of these tasks is identical. The balance between them can be decided at run time.

5 CONCLUSION

In this paper, we described the resources required to achieve real-time programmable shading—programmability, memory, and computational power—requirements that many graphics hardware systems are close to meeting. We explained how this shading power can be realized in our experimental graphics system, PixelFlow. And we showed with an example, simulations, and timing analysis that a modest size PixelFlow system will be able to run programmable shaders at video rates. We have demonstrated that it is now possible to perform, in real time, complex programmable shading that was previously only possible in software renderers. We hope that programmable shading will become a common feature in future commercial systems.

ACKNOWLEDGMENTS

We would like to acknowledge the help of Lawrence Kesteloot and Fredrik Fatemi for the bowling simulation dynamics, Krish Ponamgi for the PixelFlow simulator, Jon Leech for his work on the PixelFlow API design, Nick England for his comments on the paper, and Tony Apodaca of Pixar for RenderMan help and advice. Thanks to Hewlett-Packard for their generous donations of equipment.

This research is supported in part by the Advanced Research Projects Agency, ARPA ISTO Order No. A410 and the National Science Foundation, Grant No. MIP-9306208.

REFERENCES

[1] Cook, R. L., L. Carpenter and E. Catmull, "The Reyes Image Rendering Architecture", *SIGGRAPH 87*, pp. 95-102.

[2] Whitted T., and D. M. Weimer, " A Software Testbed for the Development of 3D Raster Graphics Systems", *ACM Transactions on Graphics*, Vol. 1, No. 1, Jan. 1982, pp. 43-58.

[3] Cook, R. L., "Shade Trees", *SIGGRAPH 84*, pp. 223-231.

[4] Hanrahan, P. and J. Lawson, "A Language for Shading and Lighting Calculations", *SIGGRAPH 90*, pp. 289-298.

[5] Perlin, K., "An Image Synthesizer", *SIGGRAPH 85*, pp. 287-296.

[6] Upstill, S., *The RenderMan Companion*, Addison-Wesley, 1990.

[7] Akeley, K., "Reality Engine Graphics", *SIGGRAPH 93*, pp. 109-116.

[8] Fuchs H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *SIGGRAPH 89*, pp. 79-88.

[9] Deering, M., S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *SIGGRAPH 88*, pp. 21-30.

[10] Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading", UNC CS Technical Report TR92-034.

[11] Gharachorloo N., S. Gupta, R. F. Sproull and I. E. Sutherland, "A Characterization of Ten Rasterization Techniques", *SIGGRAPH 89*, pp. 355-368.

[12] Molnar S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *SIGGRAPH 92*, pp. 231-240.

[13] Gardner G. Y., "Visual Simulation of Clouds", *SIGGRAPH 85*, pp. 297-303.

[14] Williams L., "Pyramidal Parametrics", *SIGGRAPH 83*, pp. 1-11.

[15] Blinn, J. F., "Simulation of Wrinkled Surfaces", *SIGGRAPH 78*, pp. 286-292.

[16] Greene N., "Environment Mapping and Other Applications of World Projections", *IEEE CG&A*, Vol. 6, No. 11, Nov, 1986, pp. 21 - 29.

[17] Williams L, "Casting Curved Shadows on Curved Surfaces", *SIGGRAPH 78*, pp. 270-274.

[18] Reeves W. T., D. H. Salesin, and R. L. Cook, "Rendering Antialiased Shadows With Depth Maps", *SIGGRAPH 87*, pp. 283-291.

[19] Rhoades, J., G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney, "Real-Time Procedural Texture", *Proc. 1992 Symp. on 3D Interactive Graphics*, pp. 95-100.

[20] Akeley K., Smith K. P., Neider J., *OpenGL Reference Manual*, Addison-Wesley, 1992.

Interactive Full Spectral Rendering

Mark S. Peercy
Benjamin M. Zhu
Daniel R. Baum
Silicon Graphics Computer Systems

The scattering of light within a scene is a complicated process that one seeks to simulate when performing photorealistic image synthesis. Much research on this problem has been devoted to the geometric interaction between light and surfaces, but considerably less effort has been focused on methods for accurately representing and computing the corresponding color information. Yet the effectiveness of computer image synthesis for many applications also depends on how accurately spectral information can be simulated. Consider applications such as architectural and interior design, product styling, and visual simulation where the role of the computer is to show how objects would appear in the real world. If the color information is rendered inaccurately, the computer simulation may serve as a starting point; but in the long run, its usefulness will be limited.

Correctly handling color during image synthesis requires preserving the wavelength dependence of the light that ultimately reaches the eye, a task we refer to as full spectral rendering. Full spectral rendering has been primarily in the purview of global illumination algorithms as they strive for the highest degree of photorealism. In contrast, commercially available interactive computer graphics systems exclusively use the RGB model, which describes the lights and surfaces in a scene with their respective RGB values on a given monitor. The light scattered from a surface is given by the products of the red, green, and blue values of the light and surface, and these values are directly displayed. Unfortunately, the RGB model does a poor job of representing the wide spectral variation of spectral power distributions and surface scattering properties that is present in the real world [4], and it is strongly dependent on the choice of RGB monitor. As a result, the colors in an RGB image can be severely shifted from the correct colors.

These drawbacks have frequently been overlooked in interactive graphics applications because the demand for interactivity has traditionally overwhelmed the demand for photorealism. However, graphics workstations with real-time texture mapping and antialiasing are overcoming this dichotomy [1]. Many applications that had previously bypassed photorealism for interactivity now are capable of having some measure of both. The best current example is the blending of visual simulation technology and mechanical computer aided design for the visualization of complex objects such as automobiles. And as workstation technology continues to advance, interactive rendering quality and speed will both increase. Consequently, utilizing interactive full spectral rendering will have significant benefits.

We present an approach to and implementation of hardware-assisted full spectral rendering that yields interactive performance. We demonstrate its use in an interactive walkthrough of an architectural model while changing time of day and interior lighting. Other examples include the accurate simulation of Fresnel effects at smooth interfaces, thin film colors, and fluorescence.

Generalized Linear Color Representations

The architecture uses generalized linear color representations based upon those presented in [7] and [8]. The representations are obtained by considering scattering events as consisting of three distinct elements: a light source, a surface, and a viewer. Light from the source reflects from the surface to the viewer, where it is detected. We use the term *viewer* to apply to a set of n implicit or explicit linear sensors that extract color information from the scattered light. This information might be directly displayed, or it might be used again as input to another scattering event.

To derive the representations we expand the light source spectral power distribution in a weighted sum over a set of m basis functions. The light is represented by a *light vector*, $\vec{\epsilon}$, that contains the corresponding weights. The surface is then described by a set of m sensor vectors, where the i^{th} vector gives the viewer response to the i^{th} basis function scattered from the surface. If we collect the sensor vectors in the columns of a *surface matrix*, S, the viewer response to the total scattered light reduces to matrix multiplication; $\vec{s} = S\vec{\epsilon}$. The effect of geometry on light scattering is incorporated in the chosen illumination model.

The principal advantage of these representations comes when every light source in the scene is described by the same set of basis functions. The light vectors and surface matrices can then be precomputed, and the rendering computation reduces to inexpensive and straightforwardly implemented matrix multiplication. Additionally, the freedom to select appropriate basis functions and sensor responsivities opens wide the applications of this approach.

Address: Silicon Graphics, Inc., 2011 N. Shoreline Blvd, Mountain View, CA 94040
peercy@sgi.com; zhu@sgi.com; drb@sgi.com

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

Selection of Basis Functions: The basis functions are chosen to capture the spectral power distributions of all light sources in the scene. For a small number of independent lights, one could simply choose as basis functions the spectral curves of the lights. However, if the number of spectral power distributions for the lights is large, as, for example, when the sun rises and sets, the dimensionality can be reduced through various techniques, including point sampling and characteristic vector analysis [6] [7] [5].

Selection of Sensor Responsivities: If the scattered light is to be viewed directly, as is typically the case in interactive graphics, the sensor responsivities should be the human color matching functions based on the monitor RGB primaries. For an application such as merging computer graphics and live action film, the sensors can be chosen as the response curves of the camera used. The final image would consist of color values of the synthetic objects as if they were actually filmed on the set, so the image could be blended more easily into the live action. Similarly, the sensor values might be chosen to simulate the shift of non-visible radiation into visible light in, for example, radio astronomy or night vision goggles. If, alternatively, the scattering is only an intermediate step in a multiple reflection path, as when computing environment maps, the sensor responsivities can be chosen as the basis functions of the next event.

Hardware Implementation

Current Capabilities: Current workstations can employ the generalized linear color representations in special circumstances. When a scene contains a set of lights with identical spectral power distributions, only a single basis function is required. Light vectors then have only one component that modulates single-column surface matrices. If the viewer has three or fewer sensors, RGB hardware can perform this modulation. For scenes illuminated by multiple sources, a natural implementation is via the accumulation buffer [3]. Pixels from the framebuffer can be added to the accumulation buffer with an arbitrary weight, so the matrix multiplication can be computed with multiple passes through the accumulation buffer, one for each basis function, as if it were a single illuminant.

Required Modifications: Needless to say, the accumulation buffer involves substantial added computational effort as all of the geometry is recomputed in each pass. A superior solution is obtained by folding the linear color representations into the hardware, a goal we have achieved in a prototype system by altering a Silicon Graphics RealityEngine™ [1]. RGB products must be replaced by matrix multiplication at every point in the rendering path that performs illumination computations – polygon lighting, texture mapping, and environment mapping. Vertex lighting calculations are implemented through microcode modification. Currently, our system allows decal textures or intensity modulated textures; full spectral textures, on the other hand, require ASIC hardware modifications. We have devised solutions to full spectral texturing and environment mapping and the hardware required to implement them.

Full Spectral Examples: We implemented an interactive walkthrough of the Barcelona Pavilion, originally designed by architect Ludwig Mies van der Rohe. The light sources in the scene include ambient skylight, directional sunlight, and multiple interior fluorescent lights. The user can interactively change the time of day while traversing the database. As the time of day changes, the spectral power distributions both from the disk of the sun and from the ambient skylight change – however, the change in spectral power can be captured with a small number of basis functions [5], an excellent demonstration of the flexibility of the generalized linear color representations. Images from a walkthrough are shown in Figure 1 in the color plates.

Environment mapping based upon sphere maps [2] can be used to preserve both surface and illumination information. Therefore, we can reproduce, for example, accurate Fresnel reflection and thin film colors, both of which depend on full spectral data (Figure 2). Additionally, the generalized linear color representations need not be restricted to the visible wavelengths. For instance, fluorescent objects convert ultraviolet energy to visible light. With no additional rendering cost after precomputing the surface matrices, our system can correctly display fluorescent objects (Figure 3). Similarly, these representations may be applied to the simulation of infrared camera response in, for instance, night vision goggles.

Ongoing Research

Improved color calculations during image synthesis will likely become more important as hardware and software improvements continue to be made. One area that increased accuracy in color reproduction can have a significant impact is the seamless merging of computer graphics and live action film or video. By simulating the sensors of the camera that captured the original footage and the lighting information from the set, it is possible to simulate the appearance of computer graphics objects under the same lighting conditions as the actual set. A complete solution to this problem must pay particular attention to the calibration of the color values from the camera, a non-trivial task. We are currently studying this problem and are applying our rendering system to its solution.

References

[1] Akeley, Kurt. RealityEngine Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics (August 1993), 109-116.

[2] Haeberli, Paul and Kurt Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In Computer Graphics 25, 4 (August 1990), 309-318.

[3] Haeberli, Paul and Mark Segal. Texture Mapping as a Fundamental Drawing Primitive. *Proceedings of the Fourth Eurographics Workshop on Rendering* (1993), 259-266.

[4] Hall, Roy. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.

[5] Judd, D. B., D. L. MacAdam, and G. W. Wyszecki. Spectral Distribution of Typical Daylight as a Function of Correlated Color Temperature. *J. Opt. Soc. Am. 54*, 8, (1964), 1031-1040.

[6] Meyer, Gary. Wavelength Selection for Synthetic Image Generation. *Computer Vision, Graphics, and Image Processing 41* (1988), 57-79.

[7] Peercy, Mark S. Linear Color Representations for Full Spectral Rendering. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics (August 1993), 191-198.

[8] Wandell, Brian. The Synthesis and Analysis of Color Images. *IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-9*,1 (1987), 2-13.

Interactive Volume Visualization on a Heterogeneous Message-Passing Multicomputer

Andrei State*, Jonathan McAllister*, Ulrich Neumann‡,
Hong Chen*, Tim J. Cullip*, David T. Chen* and Henry Fuchs*

*University of North Carolina at Chapel Hill
‡University of Southern California

ABSTRACT

This paper describes *VOL2*, an interactive general-purpose volume renderer based on ray casting and implemented on Pixel-Planes 5, a distributed-memory, message-passing multicomputer. VOL2 is a pipelined renderer using image-space task parallelism and object-space data partitioning. We describe the parallelization and load balancing techniques used in order to achieve interactive response and near-real-time frame rates. We also present a number of applications for our system and derive some general conclusions about operation of image-order rendering algorithms on message-passing multicomputers.

1 INTRODUCTION AND PREVIOUS WORK

Volume rendering is a widely used visualization method. Due to the large number of graphics primitives (voxels) which must be visited during the image generation process, real-time (or even interactive) frame rates are difficult to achieve, even on highest-performance graphics engines. Previous work that addressed this computational expense problem includes [9], in which a number of parallelization and load balancing techniques for the special case of a shared-memory architecture were presented; the rendering algorithm used was ray casting with parallel projection.

We describe an equivalent system, *VOL2*, for a distributed-memory architecture. It uses ray casting with perspective projection, a general volume rendering method suitable for a variety of visualization tasks. Ray casting is an image-order algorithm in which volume data is traversed and sampled by rays emanating from the viewpoint; the rays intersect the image plane; they accumulate (integrate) information about the volume data during traversal. The algorithms and principles used as the basis for VOL2 are outlined in [2,4,5,8,10,13,19]. An early experimental precursor of VOL2 was mentioned in [12,19]. An early version of this paper was published as [11].

The remainder of this work is organized as follows: brief overview sections on the hardware platform used and the type of display presented to the user are followed by a detailed description of the internal pipelined-parallel system layout. We then describe the types of visualization modes and graphics primitives supported by VOL2. The largest section is devoted to methods used to obtain interactive and real-time performance levels; these include a technique derived from "frameless rendering" [1]. We conclude with an overview of applications for our system.

2 HARDWARE PLATFORM

VOL2 is implemented on Pixel-Planes 5, a high-performance graphics engine with general-purpose computing nodes (called Graphics Processors or GPs) based on the Intel i860 microprocessor, and special-purpose rendering nodes based on massively parallel SIMD processor-enhanced memories [3]. Each GP has 8 Megabytes of local memory. Each rendering node can execute pixel operations in parallel on a 128x128 pixel raster, which corresponds to 1/20 of the final 512x640 pixel image. All nodes are interconnected via the system's internal 5 Gigabit/sec token ring network. Also connected to the token ring are frame buffers and the Sun-4 host computer.

3 PRESENTED DISPLAY

VOL2 produces successively refined displays by rendering a coarse image while the view parameters are changing, and by gradually increasing the image quality during interaction pauses (Plate 1). Kinetic depth effect is provided by appending to such a successive refinement sequence a series of seven highest-resolution frames; these cyclically displayed cineloop frames present the visualized structures in animated oscillatory rotation (rocking). The user will observe gradually increasing image resolution, followed by increasingly smooth left-right rocking of the displayed high-resolution structures (as each successive cineloop frame is being computed, it is immediately included in the rocking sequence). Additional depth cues are provided by directional lighting with diffuse and specular reflections.

4 RENDERING PIPELINE

The rendering pipeline has six components (Fig. 1): the host, the master GP, ray casters, compositors, splat processors (for screen interpolation), and the frame buffer. The host provides UNIX services and allows users real-time control through an X-window interface and other input devices (joysticks, trackers). The master

*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175.
(state|mcallist|chenh|cullip|chen|fuchs)@cs.unc.edu

‡Computer Science Department, University of Southern California, Henry Salvatori 338, Los Angeles, CA 90089-0781.
uneumann@usc.edu

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

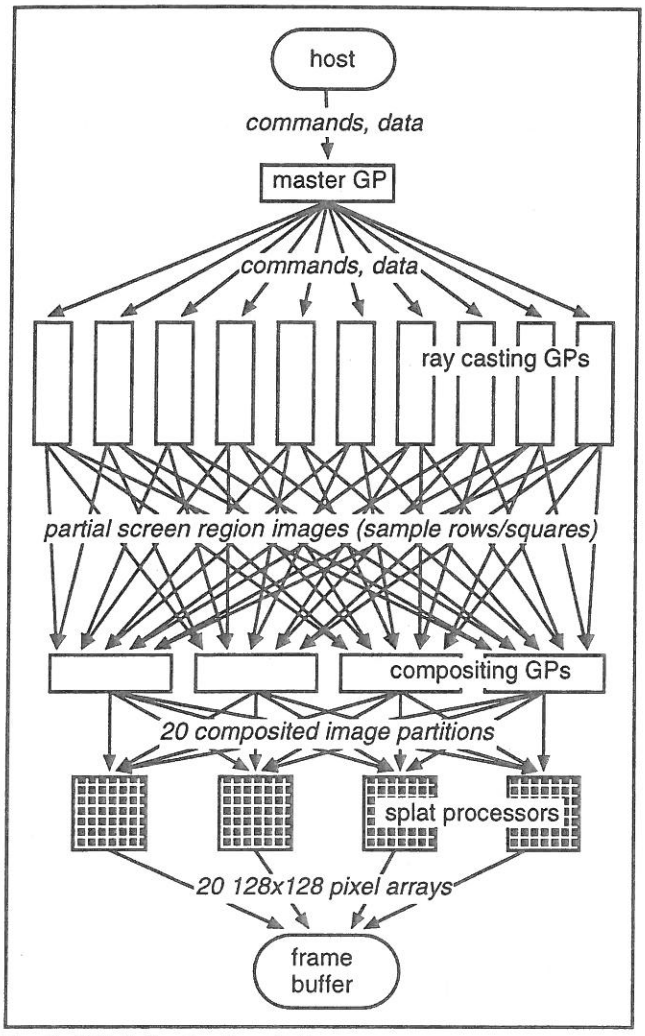

Fig. 1. VOL2 visualization pipeline.

GP is responsible for system synchronization and for load balancing the ray casters. Most of the i860 nodes are allocated as ray casters which compute image samples. Eight i860 nodes are used as compositors which combine the image samples into a final image. This image is sent to rendering nodes operating as splat processors which interpolate the image over the full display resolution and write the result to the frame buffer.

Local memory on each GP can hold only a limited number of voxels (about 6M in 8-bit voxel mode and 1.5M in 32-bit voxel mode). If the data set is too large to be replicated on all ray casting nodes, it is partitioned into slabs at system startup time; the ray casting GPs are partitioned into groups [8]. Each group of ray casting GPs is assigned to a slab of the data set (object-space partitioning, Fig. 2). During rendering, ray casters sample their assigned slabs on an image-space grid, compute partial screen region images (i. e., arrays of partially composited ray segments) and send these to the compositors. The latter combine the partial image samples into final image samples. This is accomplished by front-to-back compositing of the ray segments. Typically 8 nodes are allocated to the compositing task, each responsible for a 640x64 pixel horizontal band of the final 640x512 pixel image. The actual resolution of the computed image varies due to the use of successive refinement; the array of composited image samples sent to the splat processors to generate the fixed resolution (640x512) final image is thus of variable size.

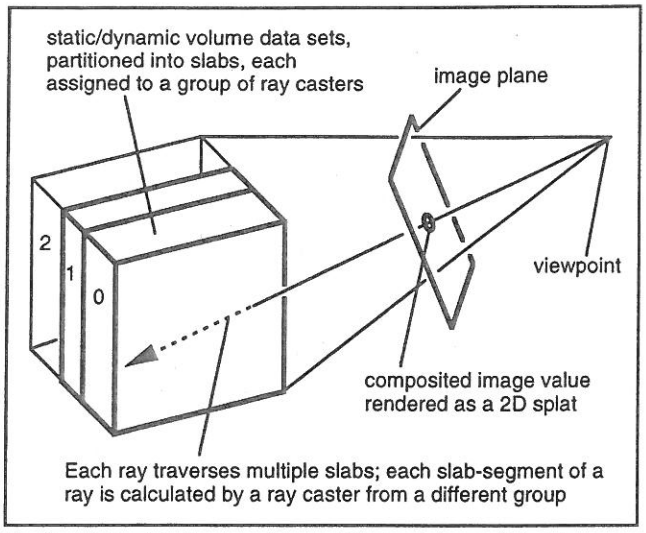

Fig. 2. Static object space data partitioning into parallel slabs.

The SIMD rendering nodes are used as splat processors due to their availability and efficiency at this task [10]. Composited image samples are convolved with a 2D filter kernel to resample the image at frame buffer resolution. Several user-selectable filter kernels are implemented, among them box, bilinear, biquadratic, piecewise quadratic and bicubic filters (Plate 2). The resampled values are sent to the frame buffer for display.

5 RENDERING OPTIONS

The ray caster code implements a number of rendering modes, such as isosurface rendering, direct rendering with and without shading, and maximum intensity projection (MIP). Plate 3 illustrates the visualizations obtained by these modes from the same data. Adding a new rendering mode to VOL2 amounts to writing a new ray caster core function; ray caster core functions are used in the innermost ray casting loop to sample the data set at a specific position along a ray and interpret the sample in a specific way (isosurface search, opacity accumulation, etc.). This modular design allows for easy prototyping and experimentation with new rendering modes without overburdening the programmer with the intricacies of Pixel-Planes 5 multiprogramming.

VOL2 supports wireframe line segments and flat-shaded triangles as graphics primitives. The (antialiased) lines are Z-buffered against isosurfaces and against each other. They are added by the splat processors (Fig. 3) after the image is resampled to frame buffer resolution (lines are only visible within fully transparent areas of the data set). Triangles can be used to add reference geometry to the scene and may penetrate into the volume data set. They are rendered by the ray casting GPs since they must be composited properly with the volume data. Since there are typically few triangles in our applications, their rendering cost is minimized by testing their individual bounding boxes against each screen region to ascertain if rays cast on a particular ray casting GP (i. e., through a specific screen region) will hit any triangles; ray setup involves computing the intersection distance to the polygons to eliminate intersection tests at every ray step.

A cut-plane for the volume data set is also provided. It is textured with the volumetric data (visible in Plate 5). The cut plane can be moved by the user to examine any arbitrarily positioned or oriented cross-section of the volumetric data set.

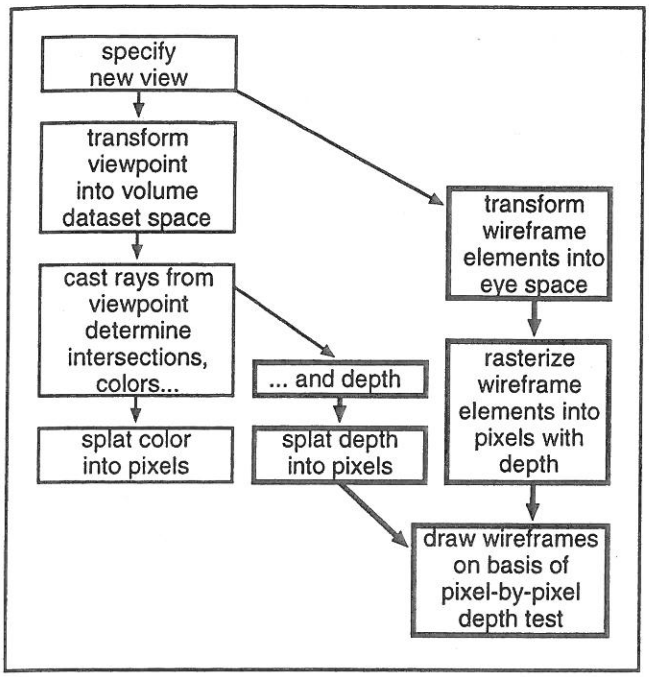

Fig. 3. Algorithm for combining wireframe line segment primitives with volume rendered images.

6 OBTAINING INTERACTIVE PERFORMANCE

The generality of ray-casting (for example, in isosurface rendering the surface thresholds can be changed on-the-fly, since no intermediate geometric primitives have to be generated). has its price. Ray casting is computationally expensive, even for relatively small data sets (1M voxels). We therefore attempted to identify and remove or alleviate VOL2's performance bottlenecks.

6.1 BY-PASS CODE

In order to obtain timing measurements, *by-pass* code was implemented in the master GP, ray casting and compositing nodes. By-pass code is derived from the code normally executing on the computing nodes by removing all compute-intensive operations and retaining only the message-passing instructions, thus preserving a computing node's ability to operate in the system (by essentially "fooling" the nodes it communicates with).

By selectively activating by-pass code for certain nodes, one can determine how fast the rest of the system can be operated. For example, by activating by-pass code for all nodes, we can determine the maximum obtainable system performance for our image generation pipeline layout (Fig. 4); by activating by-pass code for all nodes except the master GP, we can determine at what frame rates the master GP becomes overburdened during system operation (Fig. 5); by activating by-pass code for the ray casters and the master GP, we obtain the maximum speed at which the compositing/splatting/display back-end can operate—compositing performance is fairly independent of image content; it depends mostly on image resolution and the number of object partitioning slabs (Fig. 6).

6.2 LOAD BALANCE AND ADAPTIVE SAMPLING

Unlike the other system components, the ray casting nodes do not exhibit a maximum speed behavior. Their performance depends largely on data set size and image content. While master GP and compositing back-end have to be able to keep up with the ray

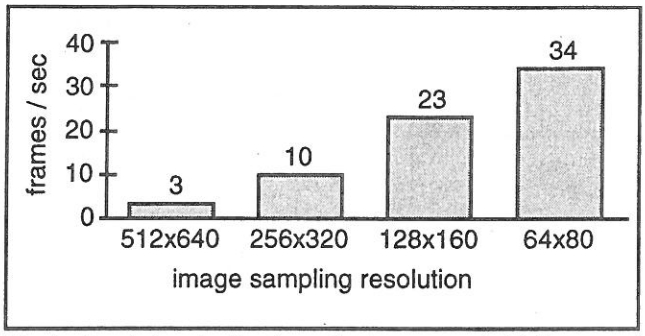

Fig. 4. Rendering pipeline throughput, measured with by-passed image generation code in all pipeline stages; this shows the maximum speed supported by the message-passing framework at different image sampling resolutions. These numbers are independent of the number of computing nodes present in the system.

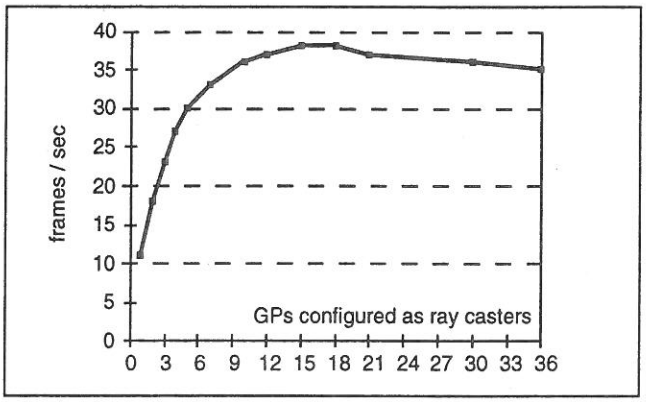

Fig. 5. Master GP maximum performance, measured with by-passed code on all nodes except the master GP. If the system contains few ray casters, the frame rate is low due to screen region processing (however minimal due to by-passing) on a single ray caster. For larger numbers of ray casters, we measure master GP maximum speed.

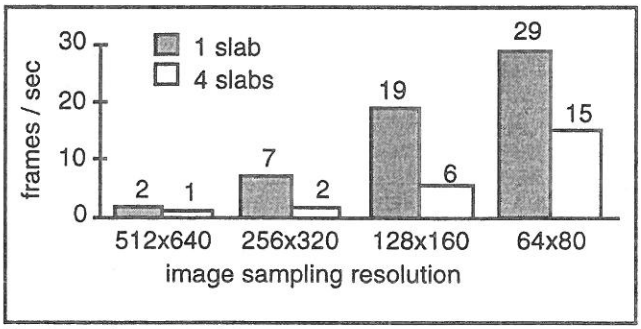

Fig. 6. Performance of the compositing-splatting-display back-end, measured with by-passed image generation code in the ray casting nodes. These numbers are independent of the number of computing nodes present in the system.

casters for the types of data sets and displays VOL2 is normally used for, the ray casters themselves have to be load balanced with respect to each other. To that end, ray casters are dynamically assigned screen regions for image generation processing. Two assignment methods are implemented and are user-selectable. In the *sample rows* approach the master GP assigns sequential rows of samples (Fig. 7, left) to the ray casting nodes on a first-come-first-serve (FCFS) basis. The rows are distributed in order, starting at the top of the image. This provides good load balance, but precludes adaptive sampling since a 2D context is required for it on each ray caster.

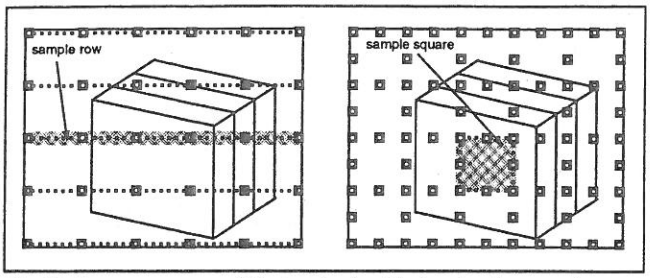

Fig. 7. Image space partitioning for load balancing by sample rows (left) and sample squares (right).

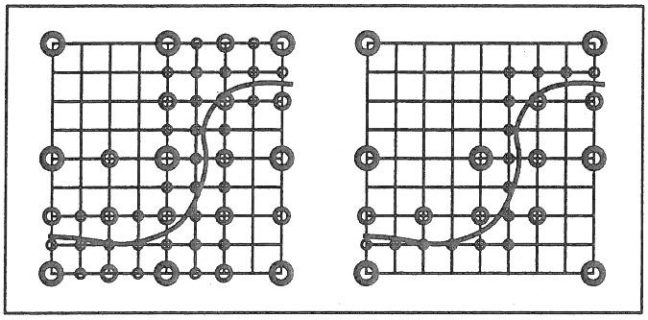

Fig. 8. Conventional adaptive subdivison (left) causes 51 samples to be taken while partial subdivision (right) requires only 29 samples. Samples taken at successive levels of subdivision are represented by progressively finer circles. The curve boundary triggers the subdivision criterion.

In the second load balancing approach (*sample squares*) the master GP distributes a total of eighty 65x65-pixel square screen regions (Fig 7, right) on a FCFS basis to the ray casting nodes. The square region size (1/80th of the final image) includes two edges of replicated rays to support adaptive sampling without seams. Squares are distributed in order of descending cost, where cost is the time taken to render the region in the previous frame (0th order cost prediction). Typically, assignment of squares on the basis of descending cost provides approximately 10-20% increase in frame rate over distribution in screen order.

The sample squares approach is combined with adaptive sampling, implemented as a modified form of recursive square subdivision. It requires fewer rays and provides similar results to that used in [6]. The conventional approach (Fig. 8, left) fully subdivides a square area by computing five new samples when any pair of four corner values exhibit variance above a user-defined threshold. Our partial subdivision approach (Fig. 8, right) computes new samples only between varying sample pairs with the center sample taken if any samples within a square vary. The example shows that the new approach requires fewer samples than the full subdivision method. A triangular subdivision method [15] has similar economy, but is less well suited to square regions.

For the isosurface ray caster, an additional optimization technique is used in combination with adaptive sampling: the ordered sequence of isosurfaces encountered along each ray is encoded in the ray sample. The encoded values are also compared during adaptive sampling; differences between neighboring rays trigger adaptive sampling along contours and isosurface intersection curves even if the threshold criterion is not met, thus enforcing accurate edge and intersection curve display.

Plate 4 shows a bar graph of the ray casting GP workloads normalized to the highest load (these and other types of test displays have proven very useful for observing the behavior of our system). Both the sample rows and sample squares approaches

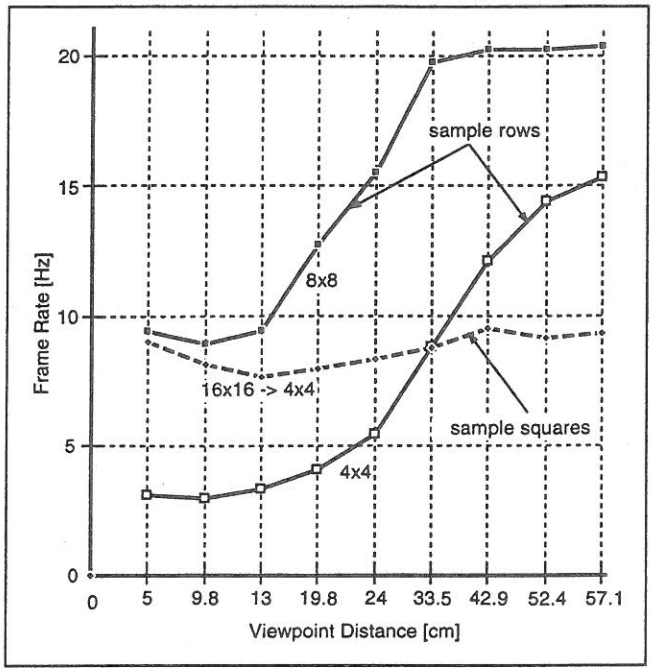

Fig. 9. Frame rate comparison between sample rows and sample squares for varying volume data set size in image space. (Full screen images are produced by 5 cm viewpoint distance, 60 cm distance produces approximately 1/16 screen coverage.) Line partitions cast rays every 8x8 or 4x4 pixels. Square regions are adaptively sampled initially at one ray per 16x16 pixels and refined up to one ray per 4x4 pixels. These measurements were taken on a system containing a total of 21 GPs, of which only 4 were allocated as compositors. Figures for larger systems with 8-compositor allocation are slightly higher.

produce good load balance with the former giving better performances for images with low screen coverage and the latter approach giving better performance for full-screen images, as well as more consistent frame rates over varied image sizes (Fig. 9).

6.3 PARTIAL UPDATING

In addition to the multiple successive refinement levels, the user can select a partial updating mode to increase the frame rate. Partial updating is loosely based on the frameless rendering technique described in [1]. This causes a new frame to be displayed as soon as a user-selected fraction of the image samples have been updated. Update levels of 25%, 50%, and 100% are currently implemented. For example, if the partial updating fraction is 25%, each sample is updated once every 4 frames. When user interaction pauses, an image at the lowest successive refinement level will have filled in after four frames.

Rather than updating a randomly distributed set of samples, we update the samples on a regular grid, which has the benefit that the bookkeeping required to ensure every sample eventually gets replaced if samples are chosen randomly all but disappears; a simple modulus of the sample coordinates with the frame number tells whether to cast a ray for a given sample on a given frame.

Partial updating implies incremental image modification, requiring the array of screen samples to be preserved from one frame to the next. In our implementation, this array is stored on the compositing nodes. Note that due to image partitioning for dynamic load balancing of the ray casters, it would be difficult to preserve the (fragmented) images on the ray casters; the existence of a compositing step in our pipeline proved advantageous for the implementation of partial updating.

6.4 OTHER OPTIMIZATIONS

A number of standard techniques are used to speed up ray casting. The voxels are stored with 13-bit pre-computed normals. A shading table is computed at the start of every frame that encodes the Lambertian coefficient for the given light direction(s) as a function of the surface normal. Voxel shading is efficiently performed by lookup into this table. Pre-computed threshold bits at each voxel accelerate ray processing by flagging whether an 8-voxel cell has "interesting" material within it. The highest value in each cell is also pre-computed and used to speed up ray casting. Rays are terminated when an opacity threshold is reached.

7 APPLICATIONS

VOL2 has been used as a rendering engine (both as a separate stand-alone server and embedded in a more complex system) for a number of research projects:

7.1 INTERACTIVE RADIATION THERAPY PLANNING

VOL2 is used as a visualization tool within VISTAnet, a collaborative project whose principal application is interactive radiation therapy planning (IRTP); the goal is to deliver lethal radiation to cancerous tissue, while keeping the doses received by healthy tissue at non-lethal levels. The treatment strategy is to intersect multiple treatment beams onto a predetermined 3D target region of a patient's anatomy, a complex task requiring comprehension of shape and sensitivity of the anatomy. VISTAnet is an experimental tool enabling 3D IRTP through rapid radiation dose computation (on a Cray Y-MP™ supercomputer) combined with interactive radiation dose visualization. Cray and Pixel-Planes 5 are linked by a near-gigabit communication network.

During an interactive session, a physician user specifies anatomy data sets and defines or modifies treatment beam parameters. These are transmitted to the Cray, which computes the dose distribution produced within the anatomy by the current treatment beam configuration and sends the dose data over the high-speed network to Pixel-Planes 5, where a combined image of anatomy, treatment beams, and resulting dose is generated; the physician examines the rendering and continues to adjust the parameters. The current processing rate is several such adjustments per second for anatomy data sets containing about 1M voxels. The display (Plate 5) must hence be able to quickly convey the treatment plan's characteristics to the user.

A special ray caster core function was added to VOL2 for operation under VISTAnet; it performs isosurface rendering of anatomy and dose data sets. For the anatomy, user-defined thresholds in the CT data and pre-defined organ or tumor segmentation data are both used for on-the-fly isosurface search during ray traversal; simultaneously, the radiation dose data set is traversed in search of up to three radiation dose isosurfaces, also with user-defined thresholds. Proper compositing of the dose, anatomy and organ or tumor surfaces must be ensured, especially when multiple surfaces lie between ray samples (Fig. 10); each surface's distance from the previous sample point along the ray is computed and sorted to establish the correct order for compositing. Wireframe outlines for the radiation treatment beams are rendered using VOL2's line segment primitives.

The (dynamically changing) radiation dose data set is received asynchronously from the Cray (via the Network Interface Unit or NIU, also attached to the Pixel-Planes 5 token ring and providing access to the external VISTAnet Gigabit network). An incoming

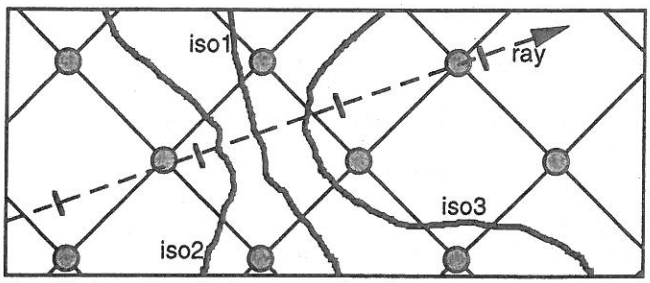

Fig. 10. Isosurfaces between samples along a ray are encountered in algorithmic order but must be sorted for compositing. Surfaces detected in <1,2,3>-order must be composited in <2,1,3>-order.

radiation dose preempts ongoing rendering for the current frame and switches context to a different task which distributes the new radiation dose to all ray casters as it is received. The distribution scheme follows the data set slab partitioning scheme described.

VISTAnet is described in more detail in [12,15].

7.2 INTERACTIVE 3D ULTRASOUND VISUALIZATION

The dynamic data set updating capabilities developed for VISTAnet are also used in an experimental augmented-reality ultrasound visualization system (Plate 6). For this system we have allocated a number of computing nodes to a volume reconstruction task: video images from an ultrasound machine are resampled into a volume data set, which is then transmitted to the ray casters for near-real-time image generation [17]. This system also required the incorporation of virtual-reality-type head and hand tracking support.

7.3 STEREOSCOPIC DISPLAY

Support for stereoscopic visualization using field-sequential stereo display on a large rear-projection screen was added to VOL2 for virtual reality experiments, as was the capability to generate such displays for head tracked viewing; this includes off-center perspective projection and the ability to position the viewpoint inside the volumetric data set.

7.4 OFF-LINE IMAGE GENERATION

Finally, VOL2 has also been used as an off-line rendering tool for simulated augmented-reality ultrasound visualization [17] and as an image precomputation tool for an experimental head-motion parallax visualization system [16].

8 CONCLUSIONS

The methods used to obtain the current performance (pipelined system layout, load balance between pipeline stages as well as between parallel nodes of individual pipeline stages) were successful—VOL2 has even been used as a skeleton for other Pixel-Planes-5-based parallel image-order renderers: polygon-based interactive ray tracing and interactive image-based morphing; both take advantage of the sophisticated, finely tuneable control over the performance/image quality tradeoff provided by the VOL2 framework.

We consider the by-pass code method one of the most useful lessons learned while building this system. This technique is generally applicable to the design of parallel/pipelined image-order renderers and has proven extremely useful as a tool to detect and eliminate performance bottlenecks in a complex multicomputer-based real-time rendering system.

9 FUTURE WORK

It has been extremely difficult to achieve VOL2's current frame rates and interactive response characteristics. The performance/resolution tradeoff is particularly unsatisfactory since it weakens kinetic depth cues. The system does indeed provide both interactive frame rates and strong kinetic depth, but not simultaneously (and hence not interactively), due to insufficient computational power. We expect significant performance improvements from an implementation of a general-purpose volume rendering algorithm on next-generation graphics multicomputers [7].

10 ACKNOWLEDGMENTS

We acknowledge the significant algorithm and software development efforts of John Rhoades, Qin Fang, Matt Lavoie, Jim Symon and Suresh Balu. This work was supported by NSF and ARPA under Cooperative Agreement NCR-8919038 with CNRI ("VISTAnet: A Very High Bandwidth Prototype Network for Interactive 3D Imaging"), by BellSouth, and by GTE. Additional funding was provided by ARPA ISTO contract DAEA 18-90-C-0044 ("Advanced Technology for Portable Personal Visualization").

REFERENCES

1. Bishop, Gary, Henry Fuchs, Leonard McMillan and Ellen J. Scher Zagier. "Frameless Rendering: Double Buffering Considered Harmful," Proceedings of SIGGRAPH '94 (Orlando, FL, July 24–29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 175–176.

2. Drebin, Robert A., Loren Carpenter, and Pat Hanrahan. "Volume Rendering," Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, 22, 4, (August 1988), ACM SIGGRAPH, New York, pp. 65–74.

3. Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs and Laura Israel. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," Proceedings of SIGGRAPH '89 (Boston, MA, July 31–August 4, 1989). In *Computer Graphics*, 23, 3 (August 1989), ACM SIGGRAPH, New York, 1989, pp. 79–88.

4. Levoy, Marc. "Volume Rendering by Adaptive Refinement," *The Visual Computer,* 1990, 6, pp 2–7.

5. Levoy, Marc. "Design for a Real-Time High-Quality Volume Rendering Workstation," *Proceedings of the Chapel Hill Volume Visualization Workshop* (Chapel Hill, NC, May 1989), pp. 85–92.

6. Levoy, Marc. "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, May 1988, pp. 29–37.

7. Molnar, Steven, John Eyles and John Poulton. "PixelFlow: High-Speed Rendering Using Image Composition," Proceedings of SIGGRAPH '92 (Chicago, IL, July 26–31, 1992). In *Computer Graphics*, 26, 2 (July 1992), ACM SIGGRAPH, New York, 1992, pp. 231–240.

8. Montani, C., R. Perego and R. Scopigno. "Parallel Volume Visualization on a Hypercube Architecture," Proceedings of the 1992 Workshop on Volume Visualization (Boston, MA, October 19–20, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992, pp. 9–16.

9. Nieh, Jason and Marc Levoy. "Volume Rendering on Scalable Shared-Memory MIMD Architectures," Proceedings of the 1992 Workshop on Volume Visualization (Boston, MA, October 19–20, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992, pp. 17–24.

10. Neumann, Ulrich. "Interactive Volume Rendering on a Multicomputer," Proceedings of the 1992 Symposium on Interactive 3D Graphics (Cambridge, MA, March 29–April 1, 1992), special issue of *Computer Graphics,* ACM SIGGRAPH, 1992, pp. 87–93.

11. Neumann, Ulrich, Andrei State, Hong Chen, Henry Fuchs, Tim J. Cullip, Qin Fang, Matt Lavoie and John Rhoades. "Interactive Multimodal Volume Visualization for a Distributed Radiation-Treatment Planning Simulator," Technical Report TR94-040, University of North Carolina at Chapel Hill, Computer Science Department, June 1994.

12. Rosenman, Julian, Edward L. Chaney, Tim J. Cullip, James R. Symon, Vernon L. Chi, Henry Fuchs and Daniel S. Stevenson. "VISTAnet: Interactive Real-Time Calculation and Display of 3-Dimensional Radiation Dose: An Application of Gigabit Networking," *Int. J. Radiation Oncology Biol. Phys.*, 25, Pergamon Press Ltd., 1992, pp. 123–129.

13. Sabella, Paolo. "A Rendering Algorithm for Visualizing 3D Scalar Fields," Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, 22, 4, (August 1988), ACM SIGGRAPH, New York, pp. 51–58.

14. Shu, Renben and Alan Liu. "A Fast Ray Casting Algorithm Using Adaptive Isotriangular Subdivision," Proceedings of Visualization '91 (San Diego, CA, October 22–25, 1991), Gregory M. Nielson and Larry Rosenblum, Editors, IEEE Computer Society Press, Los Alamitos, CA, October 1991, pp. 232–238 and 426.

15. State, Andrei, Julian Rosenman, Henry Fuchs, Tim J. Cullip and Jim Symon. "VISTAnet: Radiation therapy treatment planning through rapid dose calculation and interactive 3D volume visualization,"*Visualization in Biomedical Computing 1994* (Rochester, MN, October 4–7, 1994), Richard A. Robb, Editor, Proc. SPIE 2359, 1994, pp. 484–492.

16. State, Andrei, Suresh Balu and Henry Fuchs. "Bunker View: Limited-range head-motion-parallax visualization for complex data sets," *Visualization in Biomedical Computing 1994* (Rochester, MN, October 4–7, 1994), Richard A. Robb, Editor, Proc. SPIE 2359, 1994, pp. 301–306.

17. State, Andrei, David T. Chen, Chris Tector, Andrew Brandt, Hong Chen, Ryutarou Ohbuchi, Mike Bajura and Henry Fuchs. "Case Study: Observing a Volume Rendered Fetus within a Pregnant Patient," *Proceedings of Visualization '94* (Washington, DC, October 17–21, 1994), R. Daniel Bergeron and Arie Kaufman, Editors, IEEE Computer Society Press, Los Alamitos, CA, pp. 364–368 and CP-41.

18. Westover, Lee. "Interactive Volume Rendering," *Proceedings of the Chapel Hill Volume Visualization Workshop* (Chapel Hill, NC, May 1989), pp. 9–16.

19. Yoo, Terry S., Ulrich Neumann, Henry Fuchs, Stephen M. Pizer, Tim Cullip, John Rhoades and Ross Whitaker. "Direct Visualization of Volume Data," *IEEE Computer Graphics and Applications*, 12, 4, Los Alamitos, CA, July 1992, pp. 63–71.

The Sort-First Rendering Architecture for High-Performance Graphics

Carl Mueller

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Interactive graphics applications have long been challenging graphics system designers by demanding machines that can provide ever increasing polygon rendering performance. Another trend in interactive graphics is the growing use of display devices with pixel counts well beyond what is usually considered "high resolution." If we examine the architectural space of high-performance rendering systems, we discover only one architectural class that promises to deliver high polygon performance *with* very-high-resolution displays and do so in an efficient manner. It is known as "sort-first."

We investigate the sort-first architecture, starting with a comparison to its architectural class mates (sort-middle and sort-last). We find that sort-first has an inherent ability to take advantage of the frame-to-frame coherence found in interactive applications. We examine this ability through simulation with a set of test applications and show how it reduces sort-first's communication needs and therefore its parallel overhead. We also explore the issue of load-balancing with sort-first and introduce a new adaptive algorithm to solve this problem. Additional simulations demonstrate the effectiveness of this algorithm. Finally, we touch on a variety of issues that must be resolved in order to fulfill sort-first's ultimate promise: millions of polygons for zillions of pixels.

1. Introduction

The demands for better interactivity and realism in applications such as vehicle simulation, architectural walkthrough, computer-aided design, and scientific visualization have continually been driving forces for increasing the graphics performance available from high-end graphics systems.

Interactivity implies that the images are drawn in real-time in rapid response to user input. This immediately brings out two requirements from the graphics system: it must be able to draw images at approximately 30 frames per second (real-time), and it must have low latency (rapid response).

Realism implies that the images are rendered from detailed scene descriptions, meaning that the scenes consist of many thousands of graphics primitives. Realism also requires a display system that can show the scenes with a level of detail matching what the eye can see. Providing such detail for a reasonable field of view requires millions of pixels.

There are a variety of display devices on the path toward offering better realism. The proposed HDTV standard aims at nearly two million pixels. CAVE-type immersive displays [5] cover 4 walls of a room with a total of 5 million pixels, a number much smaller than what is desirable. Even head-mounted displays (HMDs), which would seem to require many fewer pixels, already are reaching one million pixels per eye [12] and are expected to go much further. In fact, Kaiser Electro-Optics is working on an ARPA-sponsored project to create an immersive HMD system with 4.6 million pixels per eye [7].

The number of applications which will want to take advantage of such high-resolution display devices will only increase as such devices become more popular. Yet so far, the only way to generate interactive images for these devices requires massive duplication of graphics hardware. Without an efficient solution, use of such devices will be limited to those parties with large acquisition budgets.

2. Parallel Graphics Systems

The task of a graphics computer can be described fairly simply. Given a mathematical model of all the objects in a particular environment, it must compute the visual contribution of each object for each pixel in a given viewing plane. This is a type of sorting problem, a fact recognized by Sutherland, Sproull, and Schumacher in 1974 [18]. For interactive graphics, the task is performed in two major stages: transformation and rasterization. The former converts the model coordinates of each object primitive into screen coordinates, while the later converts the resulting geometry information for each primitive into a set of shaded pixels.

The graphics performance demanded by the aforementioned applications requires parallel processing at both the transformation and rasterization stages of the graphics pipeline. The former is needed to cope with the large number of primitives, while the latter is needed for the large number of display pixels. The choices for how to partition and recombine the parallelized work at the different pipeline stages lead to a taxonomy of different architectures: sort-first, sort-middle, and sort-last [13, 15].

We now briefly examine each of sort-first, sort-middle, and sort-last. In the following descriptions, we consider a framework of an application host computer working with a

UNC Sitterson Hall CB 3175; Chapel Hill, NC 27599-3175
phone: (919) 962-1878; email: mueller@cs.unc.edu

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

graphics computer subsystem. The latter consists of many parallel processors working to produce the desired images in real time. Initially, the display database is partitioned and distributed among all the processors.

2.1 Sort-first

In sort-first (figure 1), each processor is assigned a portion of the screen to render. First, the processors examine their primitives and classify them according to their positions on the screen. This is an initial transformation step to decide to which processors the primitives actually belong, typically based upon which regions a primitive's bounding box overlaps. During classification, the processors redistribute the primitives such that they all receive all of the primitives that fall in their respective portions of the screen. The results of this redistribution form the initial distribution for the next frame.

Following classification, each processor performs the remaining transformation and rasterization steps for all of its resulting primitives. Finished pixels are sent to one or more frame buffers to be displayed.

2.2 Sort-middle

In sort-middle (figure 2), there is a set of transformation processors and a set of rasterization processors. Physically, the two sets may use the same hardware, but they remain logically separate sets. Each rasterization processor is assigned a portion of the screen. To produce an image, each transformation processor completely transforms its portion of the primitives. The resulting primitive information is again classified by screen location and sent to the correct set of rasterization processors. After rasterization, finished pixels go to the frame buffer(s).

In contrast to sort-first, the original distribution of primitives is maintained on the transformation processors. For each frame, all of the transformed primitives must be routed to the correct set of rasterization processors.

2.3 Sort-last

For sort-last (figure 3), each processor has a complete rendering pipeline and produces an incomplete full-area image by transforming and rasterizing its fraction of the primitives. These partial images are composited together, typically by depth sorting each pixel, in order to yield a complete image for the frame buffer. The composition step requires that pixel information (at least color and depth values) from each processor be sent across a network and sorted along the way to the frame buffer.

Naturally, each architecture has a set of advantages and disadvantages. We outline these briefly here; for a more complete comparison, refer to [15].

2.4 Comparison

Sort-last is a very promising architecture and is discussed in detail in [13] and [14]. It offers excellent scalability in terms of the number of primitives it can handle. However, its pixel budget is limited by the bandwidth available at the composition stage. Using a specialized composition network can help to overcome this problem.

Anti-aliasing is a major problem for sort-last: regardless of the solution chosen, the composition task is non-trivial. Using super-sampling multiplies the amount of pixel bandwidth required, since each sample must be composited. A-buffer approaches introduce new complications to the composition process, since the number of fragments per pixel may vary and become arbitrarily large.

Finally, since visibility is not decided until after the composition stage, sort-last places limitations on the kinds of rendering algorithms which may be used. The choice of algorithms available for rendering transparent polygons becomes limited, for example, and visibility-based culling algorithms are less useful on sort-last.

Because of the way it builds upon traditional graphics pipelines, sort middle is a fairly natural architecture which has resulted in many implementations. Some examples are [1], [3], [6], [9], [11], and [21]. However, sort-middle's requirement that any transformation processor be able to talk to any rasterization processor means that its scalability is limited. Increasing the number of processors geometrically increases the demands on the communications network between them.

In addition, sort-middle faces load-balancing problems when the on-screen distribution of primitives is uneven. This will result in rasterization processors becoming unevenly loaded, and this in turn may degrade system performance unless careful attention is given to this problem. A variety of solutions have been used to address this issue (refer to the references above).

Sort-first is a promising architecture that has until now received little attention. It is the only architecture which inherently takes advantage of frame-to-frame coherence. In an interactive application, the viewpoint changes very little from frame to frame, and thus the on-screen distribution of primitives does not change appreciably. Since primitives in a sort-first system are only transferred when they cross from one processor's screen region to another's, only a fraction of them will have to be communicated each frame. Also, any

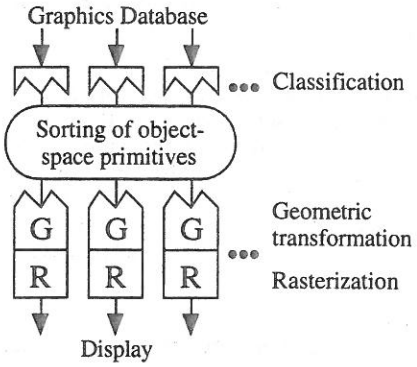

Figure 1. Sort-First Pipeline

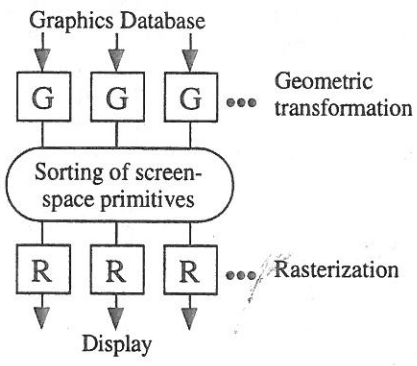

Figure 2. Sort-Middle Pipeline

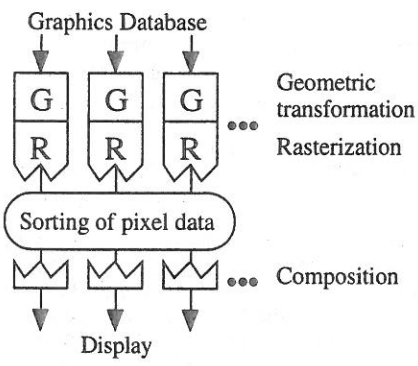

Figure 3. Sort-Last Pipeline

communication that does occur is typically fairly local; usually only "neighboring" processors will need to talk with each other. These facts suggest that it has good scalability in terms of the number of primitives it can handle.

In sort-first, once a processor has the correct set of primitives to render, only that processor is responsible for computing the final image for its portion of the screen. This allows great flexibility in terms of the rendering algorithms which may be used. All the speed-ups which have been developed over time for serial renderers may be applied here.

Since only finished pixels need to be sent to the frame-buffer, sort-first can easily handle very-high-resolution displays. This is the bottleneck for sort-last. Sort-middle also sends only finished pixels to the frame-buffer, but increasing the display resolution requires increasing either the size or number of rasterization processors, either of which causes problems. Thus sort-first is the only architecture of the three that is ready to handle large databases *and* large displays.

However, sort-first is not without its share of problems. Load-balancing is perhaps one of the biggest concerns: because the on-screen distribution of primitives may be highly variable, some thought must go into how the processors are assigned screen regions. Also, managing a set of migrating primitives is a complex task. These and other problems are the focus of this research.

3. Coherence Study

Because sort-first utilizes the coherence of on-screen primitive movement, we performed experiments to analyze this factor and determine what kind of savings might be achieved with actual applications. We wanted to know what fraction of primitives would need to be sent from processor to processor in a sort-first implementation. This testing was done using a simulation with several simplifying assumptions.

The testing involved two phases. The first was to make recordings from actual applications running on UNC's Pixel-Planes 5 graphics system. The resulting recordings contain a series of viewpoint information for each frame rendered while the application was run. The second phase was to take this information and the graphics database archive files and feed them to the simulation program. This program is based upon a framework written by David Ellsworth for his study of sort-middle systems [9]. Code was added to implement a sort-first partitioning and to calculate the resulting primitive traffic.

Various applications were used for the different test cases. "PLB" spins its database on the screen's vertical axis (named after a graphics performance benchmark from [16]). "Vixen" is a HMD-based visualization program that allows one to fly through an arbitrary display database. Finally, "Xfront" is similar except that it is joystick-controlled.

The setup for these tests is as follows:

- The database is simply a list of polygons (no structure).
- The aspect ratio of the screen is square.
- The screen is subdivided into equal-size square regions with one region assigned to each processor.
- The primitives are initially randomly distributed (the first frame's data is ignored for this reason).
- Primitives are redistributed according to the regions their bounding boxes cover.
- If a primitive falls into multiple regions, the processor at the upper-left region is deemed to be "in charge" of it.
- Off-screen primitives remain at the processor where they were last on-screen.

In these tests, the screen resolution is irrelevant; only the number of regions (and thus processors) matter. Several configurations of regions were tested: 4x4, 8x8, and 16x16. The simulation program outputs a series of values per frame representing the percentage of primitives that had to be communicated in that frame. From these figures, we calculate the arithmetic mean, the high value, the standard deviation, and the 95th percentile value.

For PLB, the database is a scanned model of a human head (see plate 1). The model is placed in the center of the screen and spun at 4.5 degrees per iteration around a vertical axis through its center (as in [16]).

PLB head	59,592 polygons, 80 frames		
regions:	4x4	8x8	16x16
mean	4.06 %	8.80 %	18.07 %
high	5.19	10.30	20.80
std-dev	0.54	0.70	1.05
95-%	5.07	9.92	20.06

For Vixen, the test case is a HMD walk-through of a Sitterson Hall's lobby (plate 2). The path starts on the mezzanine, goes down the stairs, and then turns around to look back at the starting point.

Lobby	16,267 polygons, 218 frames		
regions:	4x4	8x8	16x16
mean	2.13 %	4.95 %	11.41 %
high	21.17	45.17	87.44
std-dev	3.38	7.28	15.05
95-%	8.67	20.67	44.60

For Xfront, the model is a terrain database of a section of the Sierra Nevada mountains (plate 3). The model undergoes a series of zooms, rotations, and translations, with an abrupt reset between each sequence.

Sierra	162,690 polygons, 234 frames		
regions:	4x4	8x8	16x16
mean	3.17 %	6.08 %	11.51 %
high	98.07	102.26	107.38
std-dev	7.68	9.53	11.76
95-%	5.04	10.36	20.53

Looking at the results, we can see that increasing the number of regions increases the percentage of primitives that are communicated. This is fairly obvious, since increasing the number of region borders will increase the chance of a primitive crossing them.

The high values are somewhat interesting. For Sierra, the large values resulted from the abrupt transitions in this sequence. These exceeded 100% for two of the cases since primitives which fall into multiple regions may need to be sent more than once.

The percentiles perhaps are of greatest interest. They show that for moderately interactive applications (PLB, Xfront), 95% of the rendered frames require reshuffling of only about 20% of the primitives or less. For more highly interactive applications (Vixen), this figure goes up to about 45% in the worst case. As one may expect, this figure is directly related to the type of motion present in the application and how it affects the scene. A HMD user can create a lot of relative motion simply by rapidly turning his head.

The figures suggest that temporal coherence can provide sort-first with a dramatic savings in the amount of communication it must perform. The amount of savings is related directly to the