Parallel Lumigraph Reconstruction

Peter-Pike Sloan
Microsoft Research
One Microsoft Way 31/1056
Redmond WA 98052
ppsloan@microsoft.com

Charles Hansen
Dept of Computer Science
University of Utah
Salt Lake City, UT 84112
hansen@cs.utah.edu

Abstract

This paper presents three techniques for reconstructing Lumigraphs/Lightfields on commercial ccNUMA parallel distributed shared memory computers. The first method is a parallel extension of the software-based method proposed in the Lightfield paper. This expands the ray/two-plane intersection test along the film plane, which effectively becomes scan conversion. The second method extends this idea by using a shear/warp factorization that accelerates rendering. The third technique runs on an SGI Reality Monster using up to eight graphics pipes and texture mapping hardware to reconstruct images. We characterize the memory access patterns exhibited using the hardware-based method and use this information to reconstruct images from a tiled UV plane. We describe a method to use quad-cubic reconstruction kernels. We analyze the memory access patterns that occur when viewing Lumigraphs. This allows us to ascertain the cost/benefit ratio of various tilings of the *texture* plane.

1 Introduction

Lumigraphs [3] and Lightfields [8] are ways of representing the plenoptic function [9] using four degree-of-freedom (DOF) under the following two conditions: that the viewer is outside the convex hull of the object being viewed[1] and that both the geometry and illumination of the scene are static. The fundamental concept behind these representations is that given a point in space and knowledge about what light leaves that point from any incident viewing angle, it is possible to reconstruct an image of that point from any viewpoint. By extending this to all points on the convex hull of a surface, a box for example, it is possible to reconstruct an image of that surface from any viewpoint.

There are many ways to parameterize this 4D function. The two-plane parameterization, ST and UV, is currently the most common parameterization and is used by both the Lumigraph and the Lightfield. Unfortunately, the Lumigraph and Lightfield papers used similar symbols in different contexts, we will use the convention in the Lumigraph paper and consider the first plane to be ST and the second plane to be UV. We will also just refer to this 4D function as a Lumigraph. In all of the examples discussed in this paper, we choose planes that are parallel to each other which is a reasonable assumption in terms of sampling distribution [8].

It is difficult to interactively reconstruct reasonable sized[2] images using Lumigraphs and Lightfields due to the enormous size of the 4D function. An obvious method for handling the large amounts of data is through compression [8]. Another method for addressing this is to sparsely sample the 4D function thereby trading off image fidelity for interactivity [15]. To render at full fidelity, one needs parallel techniques for two fundamental reasons: reconstruction of the 4D plenoptic function is computationally intensive and the storage requirements for a densely sampled, uncompressed Lumigraph are enormous.

This paper will present parallel methods to accelerate purely software-based reconstruction and demonstrate a parallel implementation using a parallel ray tracer [12]. We also describe an architecture for distributing hardware-based reconstruction using texture mapping by leveraging multiple graphics accelerators in this case on an SGI Reality Monster with eight InfiniteReality pipes. We extend the reconstruction to use a tiled UV plane and show how to reconstruct with higher order basis functions.

In the next section, we review Lumigraphs and the sampling issues for reconstruction. Related work is then briefly discussed. Following that, we present the software-based and hardware-based reconstruction methods. Results are presented of the parallel implementations on a 32 CPU/8 IR SGI Origin 2000. We then conclude with possible future directions for this research.

2 Lumigraph

Lumigraphs/Lightfields are parameterizations of light leaving a convex bounding volume. A "slab" of light consists of two planes, a front and back plane and represents all rays that intersect both of these planes. An object can be represented by surrounding it with slabs that cover all rays in line space that intersect the object, where the ST planes are all outside the convex hull of the object.

The ST plane is sampled into some number of nodes. Each of these nodes can be thought of as an image, with a sheared frustum, of the UV plane. This is shown in figure 1. The image of a point on the UV plane captures the surface radiance [1] at that point only if the point being examined happens to lie on the surface of the object. If the UV plane is not on the surface of the object, the rays from the discretization diverge which can lead to ghosting artifacts. This case is shown in figure 2. Notice that the samples from point **B** capture the surface of the object but the samples which intersect the UV plane at **A** have diverged and actually sample different points on the object's surface. A finer sampling of the ST plane can counteract this effect but leads to much larger Lumigraphs. The original Lumigraph paper presented the notion of geometric correction [3] which addresses this issue, but that will not be dealt with further in this paper.

There are two ways to reconstruct images from a Lumigraph: image order (typically ray tracing or scan conversion) and object order. When reconstructing an image from an arbitrary viewpoint using ray tracing, determining pixel color depends on intersecting the ray from the eye point through a pixel with both the ST and UV planes. However, this intersection point rarely falls directly on the discretized samples of the ST or UV planes. The pixel color is computed by interpolating the neighboring samples as described below.

Figure 3 shows this in exaggerated detail. Each circle on the UV plane represents a sample point and each square on the ST plane

[1] or inside if the viewer is inside looking out

[2] We consider reasonable size images to be 512x512 or 1024x1024.

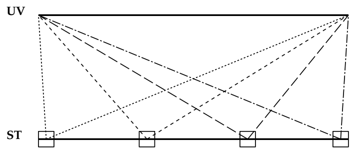


Figure 1: Each node on the ST plane can be thought of as an image, with a sheared frustum, of the UV plane.

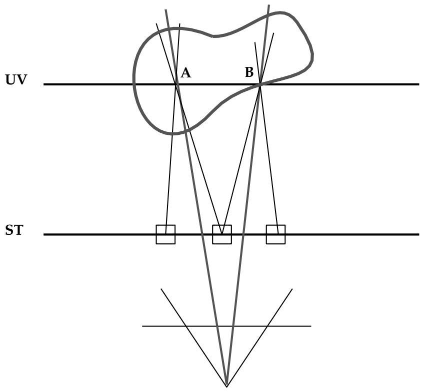


Figure 2: If the UV plane does not lie on the surface of an object, rays from the discretization of the UV plane diverge.

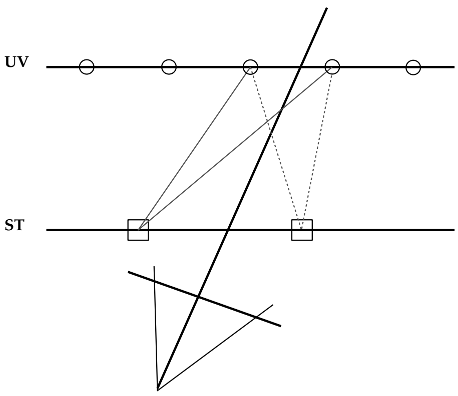


Figure 3: For image order reconstruction of a pixel, multiple samples from the ST plane are required. Each of these contain multiple samples of the UV plane. The resultant pixel color is determined through interpolation of all these values.

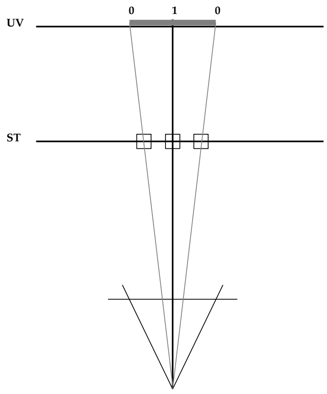


Figure 4: To reconstruct an image using an object order method, portions of the UV plane are projected onto the reconstruction plane. The portions of the UV plane are determined by neighboring ST nodes.

represents a node. For nodes on the ST plane which neighbor the ray/ST intersection the corresponding samples on the UV which are nearest the ray/UV intersection are determined. In 2D, this results in two samples per neighboring ST nodes. In 3D, this would result in four ST nodes each of which have four UV samples spanning the ray intersection. These values are interpolated, typically quadralinearly, to determine the pixel color in the reconstructed image.

Object order reconstruction projects the relevant portion of each ST node onto the reconstruction plane. This is shown in figure 4. Using neighboring ST nodes to provide the bounds, a portion of the UV plane for the central ST node, shown in grey, is projected onto the reconstruction plane. Using piecewise linear reconstruction, the contribution of the segment would falloff from one to zero as indicated. This is done for all neighboring ST nodes resulting in the reconstructed image.

3 Related Work

There have been several relevant papers dealing with lumigraphs. In [10] a parameterization is used where a parametric surface has a parameterization for orientation directly (2 degrees of freedom for surface parameterization, 2 for orientation.) This paper also has an interesting analysis of temporal and angular coherence for Lightfields/Lumigraphs parameterized in this fashion. It also uses block transform coding for decompression. In [4], geometry is decoupled from illumination. The 4D function stores the outgoing reflection direction for the incoming ray. This can be used to lookup color with an environment map or another Lightfield. They also show a method to directly render vector quantized Lumigraphs/Lightfields using graphics hardware. In [5] a very intuitive description of what amounts to geometric correction from the Lumigraph paper is described. They present a way to represent multiple focal surfaces and change them and the apertures dynamically along with a interesting way to find focal surfaces from a give dataset.

There has been much work on parallel techniques for ray tracing. Rather than review the body of knowledge here, we refer the reader to the Jansen and Chalmers overview [6] or the Reinhard, Chalmers and Jansen overview [14]. Additional background to the parallel ray tracing infrastructure used in this paper can be found in [11, 12, 13]. There has been much less work on exploiting multiple graphics pipes in parallel. A recent paper described a system which approximated global illumination by exploiting multiple graphics

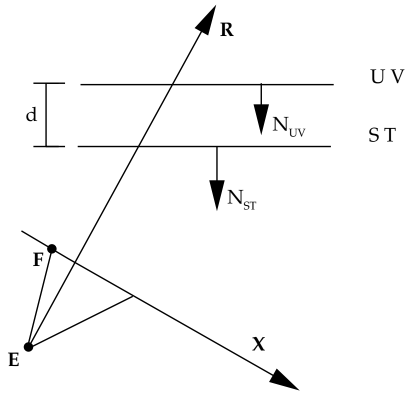


Figure 5: Ray tracing, R, through the film plane, X, into the Lumigraph $ST\ UV$

F is a point on the lower left of the film plane
E is the center of projection of the camera
X is the "x" axis of the film plane
Y is the "y" axis of the film plane
N is the surface normal

adaptors for indirect lighting [16]. They used a similar compositing method to ours.

4 Overview of Parallel Reconstruction Methods

Parallel techniques are necessary for reconstruction of the full Lumigraph due to the enormous size of the uncompressed 4D representation and the reconstruction costs involving ray tracing or hardware based reconstruction leveraging texture mapping hardware. We have chosen to present both software and hardware implementations since multipipe graphics systems are not common whereas multiple CPU systems are much more prevalent.

4.1 Software Reconstruction

The software reconstruction method is similar to the reconstruction used for the Lightfield [8]. In the Lightfield technique, the $STUV$ planes are scan converted into the reconstruction image plane and then texture filtering and lookups are performed. As pointed out in [8], one can extend ray tracing to reconstruct from the two-plane parameterization of the Lumigraph.

Let us consider the generalized ray tracing solution (see figure 5). To reconstruct the image corresponding to the new viewpoint, E, it suffices to intersect the ray, R, passing through the film plane with the ST and UV planes. The intersection point in the ST plane defines 4 views[3]. The UV intersection point is bilinearly interpolated for each of these using the same weights. Finally, these four samples, one for each node on the ST plane are bilinearly interpolated with weights based on the ST coordinates.

However, this can be optimized by employing a canonical *slab space* coordinate system. This system places the origin in the UV plane while aligning the X axis with S and U and the Y axis with T and V. The Z axis is normal to ST and UV. In this configuration, it suffices to employ only the individual coordinates. For example, $(N \cdot R)$ becomes $N.z * R.z$ since the X and Y coordinates are aligned appropriately with ST and UV.

By starting at F, a point on the lower left of the film plane, the ray equation becomes:

$$R = F + xX + yY - E$$

The ray-plane intersection can be written as:

$$(E + R\lambda) \cdot N = -d$$

Which for ST becomes:

$$\lambda = (-d - (N \cdot E))/(N \cdot R)$$

Since $d = 0$ on the UV plane, for UV it becomes:

$$\lambda = (-(N \cdot E))/(N \cdot R)$$

Due to the canonical *slab space*, it suffices to map the $STUV$ points into indices. For example, to map for S:

$$S = E.x + R.x * \lambda$$

$$S_i = S * scale + offset$$

Where scale and offset map from ST space to indices. This combines to:

$$S_i = E.x * scale + offset + R.x * \lambda * scale$$

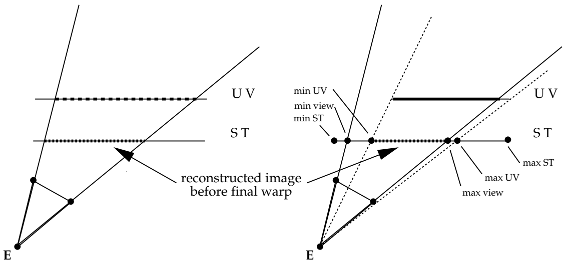
This can be further optimized by noting that $(N \cdot E)$ is constant during a frame and $(N \cdot R)$ is the same for both the ST and UV intersection. Mapping to the canonical *slab space* allows computation with a single coordinate for the different variables, z for ray intersection, x for S and U, and y for T and V. This effectively is scan conversion.

To efficiently compute the reconstructed image, we partition the film plane into blocks that effectively utilize data cache. Since each partition is independent, we can parallelize rendering them. For this, we employ the parallel ray tracing framework described in [11] which provides dynamic load balancing of frames by assigning image subblocks to a work queue for each frame to be rendered. Other processes obtain groups of these blocks from the queue in a monotonically decreasing amount. This heuristic provides quite reasonable dynamic load balancing without the overhead of determining work loads needed for task stealing.

The parallel algorithm is based on a master/slave configuration. The master process is responsible for populating the work queue, managing the display and performing updates, such as view transformation, from the user. While the implementation is straight forward, careful attention to performance details has allowed this parallel ray tracer to achieve interactive rates. This is particularly interesting for Lumigraph reconstruction where one has multiple CPUs but limited graphics hardware. As will be seen in the results section, we obtain both interactive frame rates for the reconstruction as well as excellent scaling.

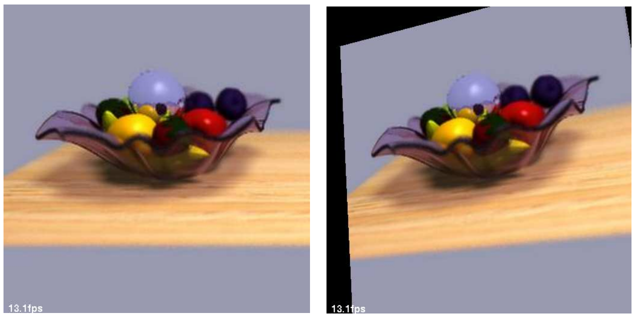
This method can be further improved with the observation that if the reconstructed image is scan converted directly on the ST plane, there is no need to perform any ray plane intersection tests. This can be accomplished through the use of a sheared frustum as shown in figure 6a. We again utilize the canonical *slab space* which ensures that X and Y line up with SU and TV. If we reconstruct onto the portion of the ST plane which is covered by the projection of the film plane, we can warp this resultant image back to the view plane using standard 2D texture mapping hardware with a single texture mapped polygon. This is similar to the idea used in shear/warp

[3] One can think of each node in the ST plane as a separate view.



(a) (b)

Figure 6: Reconstructing on the ST plane.



(a) (b)

Figure 7: Images of the (a) sheared image and the (b) final image. Also see the colorplate.

volume rendering [7]. We can incrementalize the scan conversion on both the ST and the UV planes indicated by the dashed lines in figure 6a. Note, in 3D the camera may be rotated and we can scan convert the bounding box of the projected viewing frustum. This can be further accelerated if the projection of the UV plane onto the ST plane is not fully contained in the frustum as shown in figure 6b. In this case, we need only reconstruct the portion of the projected frustum which contains samples for both the ST and UV planes. We can project the end-points of the UV plane onto the ST plane, project the bounding box of the view frustum onto the ST plane, and we know the extents of the ST plane itself. We can limit the scan conversion to the region defined by the greatest minimum and least maximum of all these points.

Figure 7a shows an image of the sheared frustum reconstructed on the ST plane. Notice how the bowl is warped. Figure 7b shows the final image which warps the sheared image onto the film plane.

By parallelizing the scan conversion process, we can speed up the rendering. We use the same parallel infrastructure as before except the film plane is not partitioned, the portion of the ST plane covered by the view frustum is partitioned. The same dynamic load balancing technique can be employed.

4.2 Hardware Based Reconstruction

As mentioned earlier, the hardware-based techniques are object order traversals. The basic idea is to directly render the contribution of each ST node using graphics hardware [3, 15]. First, basis functions must be defined over the ST plane. A triangulation of the nodes on the ST plane is commonly used because this is the most straightforward way to leverage hardware. Then for each node on the ST plane all of the triangles connected to it are drawn, using an alpha value of 1 at that particular node and 0 at all of the other nodes. The texture coordinates are determined by intersecting rays from the eye through the corresponding ST node with the UV plane. The basis functions for each image are summed, so that each triangle is rendered three times and the results added, summing up to one everywhere in the triangle. For more general basis functions this property must be preserved - the sum of the basis functions at any point on the ST plane is one. Otherwise leveraging the hardware to perform the blending would be difficult.

Figure 8: Two 1D basis functions and the tensor product (bi-linear)

We have implemented bi-cubic reconstruction on the ST plane using multi-pass rendering. The current implementation uses a 1D texture map and three passes: two passes for defining the support of the tensor product basis function, one for drawing the final image. If the application became fill rate limited a 2D image of the basis function could be used instead. The 1D texture map is initialized with the values for the basis function (Cubic Bspline in this case.) The pseudo code is as follows:

```
// only draw into alpha planes
glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_TRUE);
glEnable(GL_TEXTURE_1D); // turn on texturing
// load 1D texture
DrawQuad(0,0,1,1); // horizontal
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO,GL_SRC_ALPHA);
DrawQuad(0,1,1,0); // vertical
glDisable(GL_TEXTURE_1D);
glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
glBlendFunc(GL_DEST_ALPHA,GL_ONE);
// draw normal textured basis function
// except use 2D texture coordinates
```

Figure 8 shows the steps in drawing the support of the basis function.

This allows reconstruction using any basis functions that are strictly positive and form a partition of unity. That is, at any point on the ST plane the sum of the values of all of the basis functions that it overlaps equals one.

Figure 9a shows an image from one graphics pipe reconstructed with a constant basis function. Figure 9b shows the image from one graphics pipe reconstructed with a piecewise linear basis function. Figure 9c shows the image from one graphics pipe reconstructed with a bi-cubic basis function.

The main bottleneck with these methods is the limited amount of texture memory, and the bandwidth between the host and the accelerator. The images associated with each ST node are distributed by interleaving the pipes along the one-dimensional index of a Hilbert curve through the ST nodes. This distribution is statically created when the program starts and no attempt is made at load balancing. Each pipe simply renders the basis functions for each ST node that it contains, reads back the results and hands it off to the compositing threads. The compositing is done entirely in software, there are two reasons for this, one is that the hardware is the critical resource so it should be used as effectively as it can, the other being that the cost of the compositing can be completely hidden by overlapping the compositing with the rendering of the next frame. This causes a one frame latency, but makes a significant difference in performance.

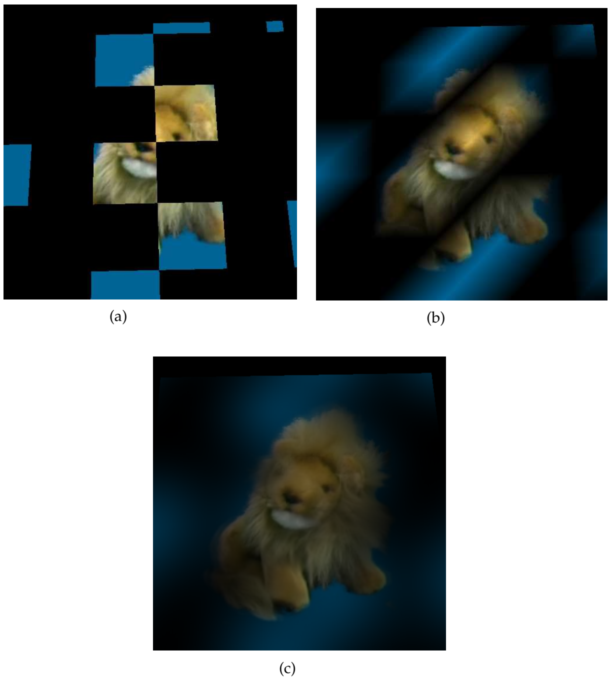


Figure 9: Images of a different basis functions: (a) constant, (b) piecewise linear, and (c) bi-cubic. Also see the colorplate.

4.2.1 Tiled Hardware Reconstruction

Loading full texture maps for each basis function being drawn is more work than is necessary. If every basis function on the ST plane was not clipped by the UV plane, the number of texels that required for a frame would be N times the number of texels on the UV plane. Where N is 1 for constant basis functions, 3 for piecewise linear, 4 for bilinear and 16 for quad cubic. Loading whole textures effectively touches the number of texels on the UV plane times the number of nodes on the ST plane. Intuitively, this can be thought of as projecting the triangulation of the ST plane onto the UV plane.

Our solution to this problem is to tile the UV plane. Instead of representing it as one texture, the UV plane is uniformly tiled into several smaller textures. Recall that figure 4 showed a single basis function viewed from a camera. In figure 10 we show the same basis function, but this time we have tiled the UV plane, only the tiles that overlap the selected ST nodes need to be used during reconstruction. In this case, only tiles 4-8 need to be loaded into texture memory rather than the entire UV plane. Another way to think of this is that you have a lumigraph for each tile on the UV plane, if the tile size was a pixel you would only need to render at most the number of samples on the UV plane times the number of ST images that contribute to each sample (determined by the choice of basis functions.)

When implementing the tiled architecture we found that our current available implementation of OpenGL would crash and/or behave very erratically when the number of texture objects bound was greater then 4096. A workaround could utilize tiles represented as sub regions inside a larger texture, but would complicate the implementation. The solution was to manage the "texture cache" ourselves. With small tile sizes (less then 64x64) it is impossible to have enough texture objects to fully utilize the amount of texture memory in the system (64MB). Currently there are the number of texture objects to fill two times texture memory, for a given tile size,

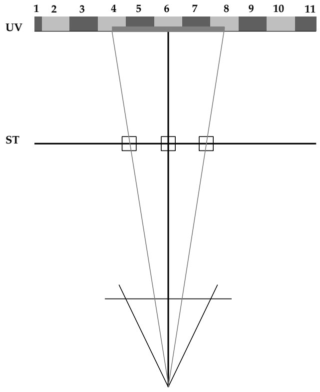


Figure 10: Tiling the UV plane reduces the amount of data needed loaded into texture memory.

but never more then 4096. This technique is more effective at utilizing OpenGL's texture caching scheme and lessens the amount of time spent rebinding texture-IDs to tiles.

Each tile in each basis function has a simple structure, which contains a "valid" word, that represents the last frame that the tile was used in and an index into the list of texture-IDs which also indicates if the tile isn't currently bound to a texture.

There is also a word associated with each texture-ID, which references the tile to which it is currently pointing. Thus, when a texture-ID is reused, the tile which is using it can be updated to reflect the fact that it is not currently bound to a texture.

The pseudo code for the system is as follows:

```
ComputeBasisFunctions(); // detail below
QueryTextureResidence();
DrawBoundTilesThatAreResident();
DrawBoundTilesThatAreNotResident();
DrawTilesSwapOld();
DrawTheRestOfTheTiles();
```

ComputeBasisFunctions creates the texture coordinates for all of the basis functions, it also computes the tile indices that overlap the bounding box of the basis functions. *QueryTextureResidence* just executes an OpenGL command that given a list of texture-IDs tells which ones are in texture memory. Strictly speaking this should not be necessary if the number of texture-IDs generated can all fit in texture memory. *DrawBoundTilesThatAreResident* loops through all of the tiles that are in texture memory and need to be drawn this frame. DrawBoundTilesThatAreNotResident is only applied to tiles that are bound and not in texture memory. After these tiles have been drawn it is necessary to begin reassigning texture-IDs to the tiles since the only remaining tiles that need to be drawn have no associated texture object. Two lists are built, one for texture-IDs assigned to tiles that were not drawn this frame and one for all of the rest of the IDs. The first list (IDs for tiles that aren't being used this frame) is exhausted followed by the second list. If there still are more tiles to draw, texture-IDs can be processed in order.

It is useful to understand the behavior of this tiling scheme. To achieve this, it is possible to optionally gather the following statistics for every frame: the total number of tiles needed for this frame, the number of tiles that were in texture memory, the number of tiles that are in this frame but were not in the last frame, and the number of tiles that were in last frame but not in this frame. This information will be presented in the results section and can be used in the future when trying to ascertain how much CPU time can be spent decompressing Lumigraphs.

	Number of CPUs					
512^2	1	2	4	8	16	30
RT	1.24	2.46	4.78	9.56	19.11	36.65
SW	2.18	4.31	8.35	16.56	32.77	58.88
1024^2	1	2	4	8	16	30
RT	0.32	0.63	1.25	2.51	4.97	9.58
SW	0.56	1.12	2.20	4.77	9.28	15.97

Table 1: Software-based parallel Lumigraph reconstruction for 512x512 and 1024x1024 images. Units are in frames-per-second. RT are the times for the ray traced algorithm while SW are the times for the shear-warp technique.

4.2.2 Software Architecture

For each graphics pipe there are two threads, one that processes input from the GUI and one that renders using OpenGL. There is also a extra window on the main graphics pipe that is used to asynchronously display the results of the software compositing.

Structuring the code this way made the implementation cleaner - both threads share the same "drawable" (in X11 parlance) and do not have to worry about mutual exclusion because they are using unique Display pointers. The GUI thread creates the window, the OpenGL thread creates the context, using a unique Display pointer but the same drawable.

The rendering process for the display-only window simply waits at a barrier for the compositing threads and then displays an image. The other rendering threads are driven by the non-display rendering thread on the main pipe.

The compositing threads divide the frame buffer into strips, each thread only being responsible for compositing into their strip. Since the basis functions form a partition of unity, the blending can be done by adding 64bit unsigned integers (2 pixels at a time.)

5 Results

Our initial results are encouraging. The machine used was an SGI Origin 2000 with 32 250mhz R10000 CPUs, 8 IR pipes (1RM7 64MB Texture) and 8 gigabytes of memory. Both the hardware and software methods demonstrate good scaling.

5.1 Software-based Reconstruction

We ran the parallel versions of reconstruction using both the ray tracing-based intersection and the shear/warp-based reconstruction. Table 1 shows the results for reconstructing into a 512x512 image and a 1024x1024 image. The tests were run on a single slab with 32x32 ST nodes and 256x256 pixels per UV image. The view chosen had every pixel in the frame buffer intersect the Lumigraph (the film plane is the UV plane, with a 36 degree field of view.) At each resolution the test was run 50 times and the results were averaged. The shear/warp-based algorithm was around 1.8 times as fast as the incremental ray tracing mode.

5.2 Hardware-based Reconstruction

The hardware-based tests were run on two data sets, a 512 MB one (2 32x32x256x256 slabs) and a 3 GB data set (3 32x32x512x512 slabs) both were reconstructed on a 512x512 window. 8 compositing threads were used. The small dataset was the bowl of fruit in figure 7. The large dataset was a Lumigraph constructed from a set of isosurface images from the visible woman dataset as seen in the figure 11.

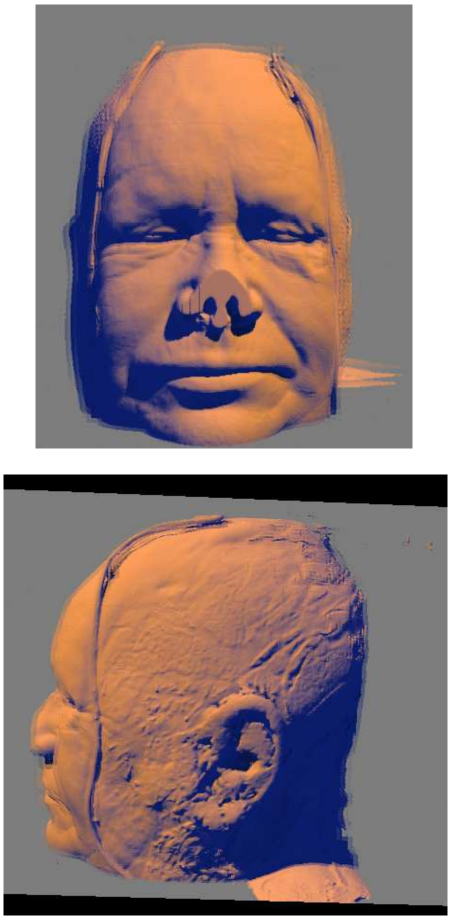


Figure 11: Images of the large dataset used in the hardware test cases. Also see the colorplate.

	Number of Tiles		
	1 tile	2 tiles	4 tiles
1 pipe	0.928	2.327	16.024
2 pipe	2.727	16.252	25.616
4 pipe	18.474	38.164	35.433
8 pipe	47.219	41.89	39.228

Table 2: Average FPS for the 512MB Lumigraph dataset.

	Number of Tiles			
	1 tile	2 tiles	4 tiles	8 tiles
1 pipe				2.006
2 pipe			0.921	8.256
4 pipe		0.823	4.246	14.019
8 pipe	0.752	2.153	22.559	18.505

Table 3: Average FPS for the 3GB Lumigraph dataset.

We captured a interactive sequence, 352 camera views, and replayed this sequence with various pipe/tiling/database selections. The first frame of the sequence always took the longest to render since there were no data from previous frames already in texture memory.

In tables 2 and 3 we present the average frames per second (FPS) over the last 351 frames of this sequence for the 512MB and the 3GB dataset. Slowing down when moving to smaller tile sizes was expected. At some point the inefficiency of transferring very small textures and the overhead from rendering the basis functions multiple times, most likely the former, will start to cause the performance to degrade. Some tiling is always faster if the entire scene doesn't fit into texture memory, often significantly so. When the frame rate dropped significantly below 1 FPS, we left the table entries blank.

The information gathered about the memory access patterns are shown in tables 4 and 5. The numbers reported are in MegaTexels (2^{20}th texture elements). The four data series are the number used per frame, the number that were already bound when the frame started, the number that were new (i.e., had not been rendered in the previous frame) and the number that were old (i.e., were rendered in the previous frame but not in the current frame). These numbers are all averages over the sequence. The number of pipes only changed the average number of bound textures.

6 Conclusions and Future Work

We have shown three parallel methods for rendering Lumigraphs. Two of them are strictly based on software (with minimal hardware support - a single texture mapped quad - for one of them) and one is based on using multiple graphics pipes. For the software method we showed how optimizing ray two-plane intersection tests across a scan line effectively boils down to scan conversion. We also showed a method of extending the principles of shear/warp rendering to reconstruct Lumigraphs that is significantly faster then the standard software technique. For the hardware reconstruction technique we showed how to reconstruct Lumigraphs with quadcubic basis functions, and an initial multi-pipe implementation. We also showed how to reconstruct from a tiled UV plane which significantly speeds things up over using full textures.

	Number of Tiles		
	1 tile	2 tiles	4 tiles
used	52.203	23.120	7.452
bound	15.747	15.558	7.115
new	0.878	0.697	0.399
old	0.872	0.703	0.402

Table 4: Average MegaTexels for the 512MB Lumigraph dataset (1 Pipe).

	Number of Tiles			
	1 tile	2 tiles	4 tiles	8 tiles
used	385.065	186.012	64.582	21.846
bound	125.619	119.279	61.274	19.641
new	6.637	6.235	4.148	2.477
old	6.606	6.254	4.149	2.478

Table 5: Average MegaTexels for the 3GB Lumigraph dataset (8 Pipe).

The most compelling future work is to leverage the data we have collected on the memory access patterns of Lumigraphs for investigating compression algorithms. In particular the CODEC has to be able to decompress at a sufficient rate to feed the rendering engine. For a desired frame rate (for a hardware implementation) the number of tiles needed for a given frame must be decompressed for rendering. We believe that the access patterns are coherent enough to try more aggressive compression then have already been successfully used (standard VQ - which has a very fast decompression time.) Hardware implementations that support compressed textures (like the S3TC technique - based on color cell compression [2]) should be leveraged to explore much more aggressive compression ratios. Also a more general tiling, not strictly based on "whole" UV planes, where locality with respect to ST and UV planes was considered would be beneficial since the tiles are clearly correlated in more then the UV plane.

Addressing the load imbalance in the current hardware method should be examined. Perhaps by using a modified work queue where there is a pool of ST nodes that aren't in any of the pipes texture memories. These would be parceled out via a work queue. Batching texture tiles together into metatiles could help with the performance impact incurred when using smaller tile sizes. The tiles to be uploaded could be batched and sent in a single glTexSubImage command.

The software version currently also runs on parallel Intel machines running Windows NT. We would like to investigate using the SIMD Floating Point instructions and potentially the streaming memory extension that exists in the Pentium III to accelerate the shear/warp and incremental rendering algorithms. The hardware-based methods for multiple pipes could be used for systems that support multi-monitor configurations.

The shear warp method has only be implemented for single slab datasets, dealing with sampling issues (based on the warp) intelligently would make a lot of sense. This method would be attractive for rendering from compressed representations because it is so memory coherent.

7 Acknowledgments

This work was supported in part by the DOE Advanced Visualization Technology Center (AVTC). Thanks to Chris Johnson for providing the open collaborative research environment that allowed this work to happen. Thanks to Yarden Livnat and the SCI group for allowing the first author to spend his weekends there working on the big machine. We wish to acknowledge the reviewers for their excellent comments. Discussions of this work with Harry Shum and Michael Cohen of Microsoft Research and Steven Gortler of Harvard University were helpful. Thanks to Steven Parker for having such a great infrastructure for plugging in the software side.

References

[1] James Arvo, Kenneth Torrance, and Brian Smits. A framework for the analysis of error in global illumination algorithms. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 75–84. ACM SIGGRAPH, ACM Press, July 1994.

[2] Graham Cambell, Tom A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, Lawrence A. Leske, John A. Lindberg, and Daniel J. Sandin. Two bit/pixel full color encoding. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 215–223, August 1986.

[3] Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael Cohen. The lumigraph. In Holly Rushmeier, editor, *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–55, August 1996.

[4] Wolfgans Heidrich, Hendrik Lensch, Michael F. Cohen, and Hans-Peter Seidel. Light field techniques for reflections and refractions. In *Eurographics Rendering Workshop 1999*. Eurographics, June 1999.

[5] Aaron Isaksen, Leonard McMilland, and Steven J Gortler. Dynamically reparametrized light fields. Technical Report MIT-LCS-TR-778, Computer Science Department, Massachusets Institute of Technology, May 1999.

[6] Frederik W. Jansen and Alan Chalmers. Realism in real time? In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 27–46, 1993.

[7] Philippe Lacroute and Marc Levoy. Fast volume rendering using shear-warp factorization of the viewing transformation. In Andrew Glassner, editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 451–458, July 1994.

[8] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[9] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In Robert Cook, editor, *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 39–46, August 1995.

[10] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Eurographics Rendering Workshop 1998*. Eurographics, June 1998.

[11] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, April 1999.

[12] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, To Appear 1999.

[13] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of Visualization '98*, October 1998.

[14] E. Reinhard, A.G. Chalmers, and F.W. Jansen. Overview of parallel photorealistic graphics. In *Eurographics '98*, 1998.

[15] Peter-Pike Sloan, Steven Gortler, and Michael Cohen. Time critical lumigraph rendering. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 17–24, April 1997.

[16] Tushar Udeshi and Charles Hansen. Towards interactive photorealistic rendering of indoor scenes: A hybrid approach. In *Eurographics Rendering Workshop 1999*. Eurographics, June 1999.