To appear in the 1997 Symposium on Interactive 3D Graphics conference proceedings

Time Critical Lumigraph Rendering

Peter-Pike Sloan* University of Utah
Michael F. Cohen† Microsoft Research
Steven J. Gortler‡ Harvard University

Abstract

It was illustrated in 1996 that the light leaving the convex hull of an object (or entering a convex region of empty space) can be fully characterized by a 4D function over the space of rays crossing a surface surrounding the object (or surrounding the empty space) [10, 8]. Methods to represent this function and quickly render individual images from this representation given an arbitrary cameras were also described. This paper extends the work outlined by Gortler et al [8] by demonstrating a taxonomy of methods to accelerate the rendering process by trading off quality for time. Given the specific limitation of a given hardware configuration, we discuss methods to tailor a critical time rendering strategy using these methods.

CR Descriptors: I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism; **Additional Keywords:** image-based rendering, critical time rendering

1 Introduction

The traditional method of creating realistic images on a computer involves first modeling the geometric and material properties of an object or scene as well as any light sources. This is followed by simulating the propagation of light and the actions of a synthetic camera to render a view of the object or scene. The complexity at both the modeling and rendering stages has led to the development of methods to directly capture the appearance of real world objects and then use this information to render new images. These methods have been called image based rendering or view interpolation [5, 4, 11, 6, 13, 10, 8] since they typically start with images as input and then synthesize new views from the input images.

In two papers, *Lightfield Rendering* [10] and *The Lumigraph* [8] it was shown that the light leaving the convex hull of an object (or entering a convex region of empty space) can be characterized by a 4D function over the space of rays crossing a surface surrounding the object (or surrounding the empty space). Given this observation, the process of creating the 4D function, dubbed a *Lumigraph* in Gortler et al, and synthesizing views involves three steps: 1. gathering samples of the Lumigraph, typically from pixel values in static images, 2. approximating and representing the continuous Lumigraph from the samples, and 3. constructing new views (2D slices of the 4D Lumigraph) from arbitrary synthetic cameras. The third step in the process should be fast to allow interactive exploration of the Lumigraph by real-time manipulation of a virtual camera. Both papers [10, 8] discuss different methods to make this possible, however, both are limited in different ways. The method discussed by Levoy and Hanrahan operates pixel by pixel and thus is highly sensitive to image resolution while the method discussed by Gortler et al takes advantage of texture mapping hardware and thus is sensitive to other limitations. Leveraging hardware texture mapping is sensitive to texture memory limitations (in our case 4Mb) and the bandwidth between the host and the accelerator. Architectures that support rendering from compressed textures [15, 1] and ones that read textures directly from host memory could overcome this bottleneck.

This paper extends the work outlined by Gortler et al by demonstrating a taxonomy of methods to limit the amount of texture information required per frame. These methods make different trade-offs between quality and time. Given the specific limitation of a given hardware configuration, one can use these methods to tailor a critical time rendering method. In addition, the methods lead naturally to progressive refinement rendering strategies.

*ppsloan@cs.utah.edu
†mcohen@microsoft.com
‡sjg@deas.harvard.edu

2 The Lumigraph

In both [10] and [8], the 4D Lumigraph function was parameterized by the intersection of a ray with a pair of planes (Figure 2). Fully enclosing a region of space involves using a series of such pairs of planes, however, we will restrict ourselves in this paper to discussing a single pair. In Gortler et al and in the work here, the first plane has axes s and t, with the second plane labeled with axes u and v. The uv plane is situated to roughly pass through the center of the object. Thus any point in the Lumigraph space has parameters (s,t,u,v) and a value (typically an RGB triple) representing the light passing along the ray seen in Figure 1. One can see the duality between getting samples into the Lumigraph from an arbitrary image and constructing an arbitrary image from the Lumigraph. In the first case, given an input image as in Figure 1, the value at (s,t,u,v) is simply the color of the image at the pixel location where the ray intersects the image. Conversely, given a Lumigraph, one can reconstruct the pixel on a desired image by using $L(s,t,u,v)$, the value of Lumigraph at the given parameter location. In this way, any arbitrary image can be constructed pixel by pixel.

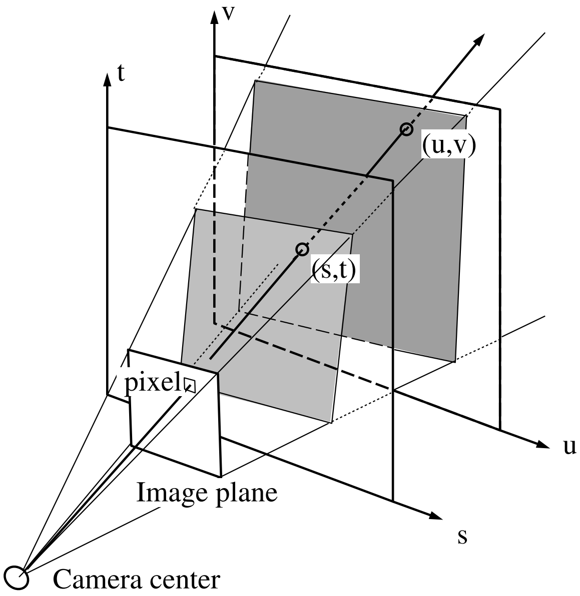

Figure 1: Lumigraph Parameterization

2.1 Discretization

In the finite context of the computer, the Lumigraph is discretized. We have found that a discretization of approximately 32x32 nodes on the *st* plane and 256x256 on the *uv* plane gives good results. One can think of the 256x256 set of RGB values associated with each *st* node (the set of rays passing through the *st* node and intersecting the *uv* plane) as an image as seen from the given *st* point (we will call this a *uv image*). We will use these images as a textures in the reconstruction methods. Given a discretization, one also needs to select a blending function between nodal values both for projection of the continuous function into the discrete one and for reconstruction. We will use a quadralinear basis for these purposes as was done in [8].

2.2 Use of Geometry

So far there has been no notion of the geometry of the object since one of the major features of pure image based rendering is that no geometry is needed. On the other hand, given some approximate geometry as can be constructed from a series of silhouettes [14] one can improve the reconstruction of images as discussed in [8]. We will also use the approximate geometry in the reconstruction methods discussed below.

2.3 Reconstruction as Texture Mapping

Gortler et al [8] describe a texture mapping process to perform the reconstruction of images with hardware acceleration. To fill in the shaded triangle on the image plane as seen in Figure 2, one draws the shaded triangle on the *st* plane three times (once for each vertex of the triangle) as a texture mapped, *alpha* blended polygon. For each of the vertices, the uv image associated with that vertex is used as the texture. An *alpha* value of 1.0 is set for this vertex with an *alpha* of 0.0 at the other vertices. The texture UV coordinates are common to all three versions of the triangle and are found by intersecting rays from the camera center through the *st* nodes with the *uv* plane. Summing the *alpha*s results in full coverage of the triangle and a linear blending of portions of three textures. Repeating this process for each *st* triangle seen in the cameras frustum completes the image. Current texture mapping hardware from SGI can often render more than 30 frames/sec at 512x512 resolution using this method.

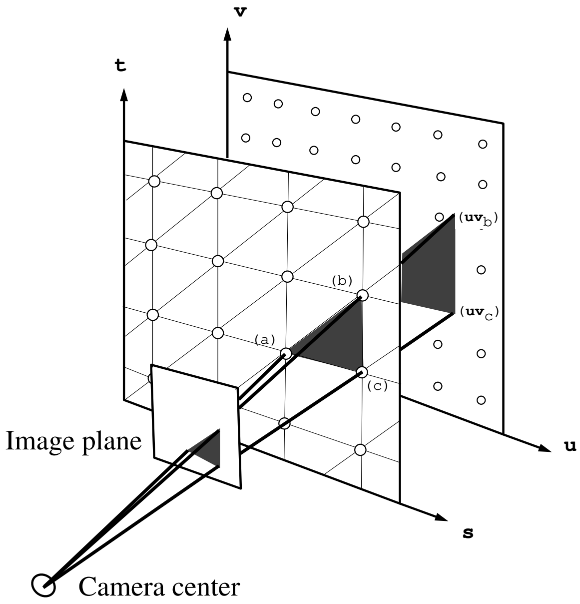

Figure 2: Lumigraph Reconstruction as Alpha Blended Textured Triangles

There are, however, some severe limitation of this approach. The most crucial one is the limited amount of texture memory available on most machines (e.g., 4Mb on our machine). In addition, main memory becomes a scarce resource as well since a full 32x32x256x256 x 6 planes x 3 byte Lumigraph fills more than a gigabyte of information. Moving data to and from disk and to and from texture memory is the major bottleneck, particularly if the individual *uv* images must be decompressed at the same time. Future hardware designs that support rendering directly from compressed textures will make this approach more attractive by ameliorating many of these limitations.

3 Fast Approximate Rendering Given Limited Resources

This section discusses a taxonomy of methods to reuse data from frame to frame. Taking advantage of coherence will allow the construction of critical time Lumigraph rendering algorithms. The methods fall into two categories, ones that use a smaller set of textures than a full rendering, and a method that uses the current reconstructed image as a new texture itself for subsequent nearby frames.

We will discuss each method briefly below.

3.1 Using a Limited Set of st Nodes

Given a limited budget of texture memory, we can devise a number of strategies to use less than the full set of information in the Lumigraph for a given view. In general, by dynamically adjusting the tesselation of the *st* plane and the *alpha* values at each vertex of the tesselation we will use less memory and need to draw fewer polygons (albeit at reduced quality). Each method must (a) cover the *st* plane (or at a minimum the visible portion), and (b) produce a set of *alpha* values that sum to unity.

- Sub-sample: the simplest idea to use a lower resolution Lumigraph. Cutting the resolution of the *st* plane in half results in one fourth of the *uv* images needed for any image. The triangles in Figure 2 would simply be twice as large in each direction. This results in more depth of field blurring which is ameliorated somewhat by the depth correction afforded by the use of the approximate geometry [8]. Cutting the resolution in *uv* also results in the need of less texture memory at a cost of overall blurring. A 2D or 4D Mipmap [16] of multiple levels can be constructed, however, just two levels of a 4D Mipmap already results in a size 1/16th that of the original Lumigraph [see Color Plate 1] .

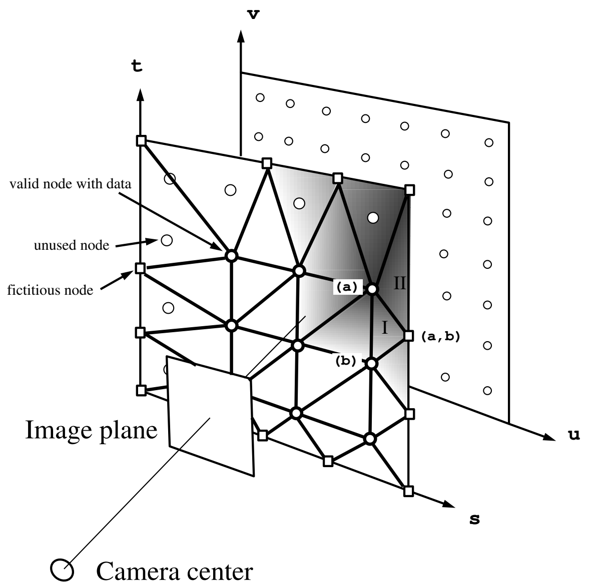

Figure 3: Triangulation of *st* Plane from Subset of Nodes

- Fixed pattern of *st* nodes: if our budget allows a fixed number of textures, for example 9, we can ask which nodes are the best to use and how should we cover the *st* plane with the data from those nodes. We can pick the 9 *st* nodes surrounding the center of the image and generate a triangulation of the *st* plane. The triangulation is created from the 9 selected nodes and other fictitious nodes used to cover the *st* plane (see Figure 3 and Color Plate 2). The triangles within the square of 9 nodes are drawn three times each as before. Those triangles outside are drawn twice. For example, triangle I is drawn first using the *uv* plane associated with node (a) and with *alpha*s of 1.0 at node (a), 0.0 at node (b), and 0.5 at the fictitious node labeled (b,c). A similar pattern is used from the point of view of node (b). Similarly triangle II is drawn with texture (a) with *alpha*s (1.0, 1.0, 0.5) and with texture (b) with *alpha*s (0.0, 0.0, 0.5). Reuse of this fixed pattern of nodes provides good parallax cues near the original image, but flattens out as the center of the image moves outside the fixed square. Thus, the nine nodes can be used for a small amount of motion, after which new nodes need to be brought in and old ones removed from the active set.

- Arbitrary triangulation: any subset of nodes can in fact be used to render an image by generalizing the algorithm above. Once a triangulation of the plane is constructed from the nodes with texture data and fictitious nodes, each triangle can be drawn by the following pseudo-code algorithm:

```
for each vertex a with valid data
  set texture to {\em uv} plane for a
  draw_node_with_data(a)

draw_node_with_data( Vertex a)
{
  set alpha_a = 1.0
  for each vertex b adjacent to a
    set alpha_b = 0.0
    for each triangle adjacent to a
      draw t-mapped alpha summed triangle
    for each vertex b adj. to a wo valid data
      alph = 1.0 / num_valid_verts_adj_to_b
      draw_node_without_data( b, alph)
}

draw_node_without_data( Vertex b, double alph)
{
  set alpha_b = alph
  for each vertex c adjacent to b
    set alpha_c = 0.0
    for each triangle adjacent to b
      draw t-mapped alpha summed triangle
}
```

One constraint in the triangulation is that every vertex must have at least one adjacent vertex that contains valid data, or there will be a hole in the reconstruction. In Figure 3, the shaded area shows the support of influence of vertex (a). A critical time rendering scheme can be designed to bring in new textures when it has time and reuse those already loaded when needed. Later we will show how this scheme can be slightly modified to produce smooth transitions.

- Nodes along a line: if the user's motion can be predicted or restricted to lie along a line (or curve with low curvature) then st nodes can be chosen to lie along this line (or tangent). In this case, the *st* plane is not divided into triangles, but rather strips perpendicular to the line (see Figure 4 and Color Plate 4). Each strip spans the space between adjacent nodes along the line. Each strip is drawn twice with the texture associated with each adjacent node. The *alpha*s are set to 1.0 along the line through the current node, and 0.0 along the opposite edge. Moving back and forth along the line of motion provides parallax cues in the direction of motion. Moving perpendicular to the line appears to just rotate a 2D image. This same trick is used in the construction of horizontal parallax only holographic stereograms [2].

3.2 Using Projective Textures

A second type of method uses the current image as a newly defined texture map for subsequent nearby images. The ap-

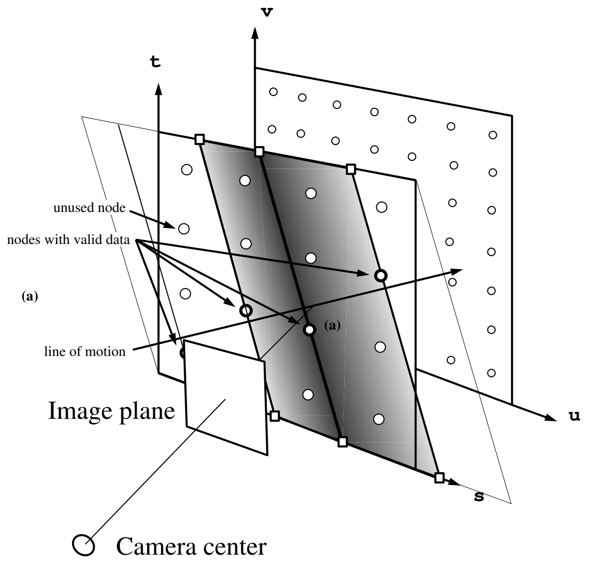

Figure 4: Subdivision of *st* Plane from Line of Nodes

proximate geometry of the object is then used to warp the texture to the new viewpoint. This is similar in flavor to the original view interpolation work by Chen and Williams [5] and the more recent work by Seitz and Dyer [13] and by Debevec et al [6]. After an initial image is created, a matrix M is constructed and used for each subsequent frame. The composite modeling and viewing matrix used for the initial frame transforms points *(x,y,z)* in model space to screen coordinates between -1 and 1 in X and Y. Composing this matrix with a translation of (1,1) and a scale of (0.5,0.5) results in a matrix that maps points in space to a UV texture space between (0,0) and (1,1). This texture matrix, M, is the used to find texture coordinates for vertices of the approximate model for each subsequent frame [12]. The texture coordinates for each vertex are set to their *(x,y,z)* location. These are transformed by M into the 2D UV texture coordinates.

In pseudo-code:

```
render an image I as usual
use image I as texture for subsequent frames
set matrix M to composite model/view matrix
compose M with translation(1,1) and scale (.5,.5)
set M as texture matrix

for each subsequent image
  set new view matrix for desired camera
  for each polygon in approximate model
    for each vertex of polygon
      set texture coordinate to (x,y,z)
      /* to be transformed by M into U,V */
      set position to (x,y,z)
      draw polygon
```

For subsequent images, near the original image that set the texture matrix M, the warp will provide correct parallax cues. As the new camera position moves away from the one used for original image, the inaccuracies in the approximate geometry, self occlusion changes in visibility, and any specular properties will become obvious [see Color Plate 5].

A critical time rendering approach thus can use each image as a texture over multiple frames. In the meantime, a new image is constructed from a new camera position as close as possible to a predicted position at the end of the rendering cycle. This new image and its associated texture matrix are then swapped in to replace the old ones, and the process is repeated.

3.3 Progressive Refinement

The methods outlined above lead naturally to a number of possible progressive refinement and/or progressive transmission methods. One can transmit an initial (small) set of *st* nodes and information about the database (i.e., *st* and *uv* resolutions, crude geometric representation). As more nodes are transferred they are inserted into the triangulation (initially without data) and schedule the texture to be decompressed and added to the new node. The geometry also could, of course, also be represented in a progressive format [9].

With a simple modification of the arbitrary triangulation algorithm the data for vertices can be smoothly blended in over time, or blended out as the vertex is removed. This requires a simple change in the classification of vertices, adding a value per vertex for the validity of its associated data, between 0.0 and 1.0. Changing this value over time smoothly reduces or increases the contribution of this node to the final image [Color Plate 3]. In pseudocode, a slightly modified `draw_node_with_data`, would be called for any vertex, a, with data validity greater than 0.0:

```
draw_node_with_data( Vertex a )
{
  set alpha_a = data validity at a
  for each vertex b adjacent to a
    set alpha_b = 0.0
  for each triangle adjacent to a
    draw texture mapped alpha summed triangle
  for each vertex b adjacent to a
    if (b.validity < 1.0)
      vb = b.validity
      alph =  (1.0 - vb)/num_valid_verts_adj_to_b
      /* same as before */
      draw_node_without_data( b, alph )
}
```

4 Critical Time Rendering Strategies

We will explore a critical time rendering strategy applied to the general case of triangulations of an arbitrary set of *st* nodes. This basic strategy can be applied to all the methods discussed above. The user sets a desired frame rate. Then, at each frame (or every few frames) a decision is made about

- which new *st* nodes to bring in (this may involve decompression, and loading/binding as textures) and which nodes to delete
- triangulating the currently valid nodes and possible fictitious nodes to cover the *st* plane, and
- rendering the frame.

The latter two points are discussed in the previous sections. This leaves us with the decision, given a fixed budget of time and/or texture memory constraints, what is the best set of nodes to use, keeping in mind which ones we already have ready to be used. To do this, like in [7] we define two concepts for each node, the benefit that will be derived from adding that node, and the cost of using it. The nodes are sorted in order of the ratio of their benefit:cost. Given this ordered set of nodes, the strategy is to add the highest ranking nodes until the time budget is used up. As the texture memory is used up the lowest benefit node is deleted and replaced by the next node in line to be added. The time budget is dynamicaly rescaled as in [3] so that the desired frame rate is met. This allows for variable load on the machine and error in the machine dependent cost measurements.

4.1 Benefit

The benefit assigned to a node is set as a function of:

- its distance from the intersection on the *st* plane of the center of projection of the camera (the point labeled *(s,t)* in figure 1)
- its position relative to the recent motion of the camera's center of projection (nodes close to a predicted path are assigned a higher benefit), and
- its distance from other currently valid nodes

Specifically,
*benefit(i,j) = Dist(s,t)*Path(s,t)*Neighbor(s,t)*
where *s,t* are the coordinates of node *(i,j)* on the *st* plane, and Dist, Path, and Neighbor each return values in [0,1]. Dist is inversely proportional to the squared distance (plus a constant) between the camera center and proposed node. Path determines the proposed node's distance from the predicted future path of the camera on the *st* plane, and Neighbor is based on the fraction of the neighboring nodes currently not in the triangulation.

4.2 Cost

The cost of adding a node are primarily the fixed costs of fetching a node from disk, decompressing it, and loading it into texture memory. The relative costs of each of these operations is machine dependent. At any time, some nodes may be in memory but not bound to textures, while others are only resident in compressed form on disk.

There is also the incremental cost of triangulation and rendering of the additional triangles introduced by the new node.

5 Performance Results

5.1 Node Usage Methods

The performance of the texture mapping algorithms depend on:

- Texture Fill Rate: How many million bilinear filtered textured pixels the hardware can generate per second. Each pixel has to be textured 3 times, for 30 fps at 512x512 resolution requires a texture fill rate of 23.6 million textured pixels/second. This is well within the range of modern 3D hardware (a Maximum Impact has peak rates of 119MTex/sec) and even quite reasonable for 3d hardware destined for personal computers (with fill rates between 17 and 60 MTex/sec). If the fill rate is close to the required amount you will have to work with almost the entire working set in texture memory.
- Texture Storage: Number of Texels that can be stored. Our Maximum Impact has fairly limited texture memory - 4 megabytes that can only be addressed as 4 channel textures for this application. This is a significant bottleneck.
- Texture Paging Bandwidth: MTex/sec that can be loaded into texture memory. The Impact loads textures from the host extremely fast, 40 MTex/s in our benchmarks. Other platforms (PCs, RE2, IR, DEC POWERSTORM) may have more texture memory (8-64 MB) and better utilization however (paletized textures or 3 channel memory layout).
- Disk Access and Decompression: How fast can a *uv* image be made ready to move to texture memory. Depending on the compression scheme used this may be significant, as is the case in our current implementation using JPEG. Levoy and Hanrahan use a two stage scheme, first a vector quantization step (with fast decompression) and then an entropy encoding step (slower to decompress). By performing the entropy decompression up front they are able to perform the remaining decompression at frame rates for the required pixels. The texture mapping process may also require decompressing some pixels that are never used. Compression issues remain a significant research topic.
- Frame Buffer to Texture Memory Bandwidth (for projective textures): How fast a previously rendered frame can be sent to texture memory. The Impact performs this at roughly 10 MTex/sec.

Below is a table that characterizes the different methods outlined above. Fill Rate refers to the average number of bilinear textured pixels that need to be drawn per pixel. Texture Load refers to how efficiently texture loads can be scheduled. Locality refers to the distribution of detail for the given method where "strong" locality means that the image can be generated with higher fidelity in a smaller neighborhood. For the Fixed method we are assuming that the 9 central nodes cover one quarter of the screen. In the fill rate equation for Arbitrary - partial valid, x represents the percentage of nodes with partially valid data. This has a minimum at 25% and the maximum is at 100%. For projection we are assuming that the model covers about 1/2 of the screen and has a low depth complexity (2).

Method	Fill Rate	Texture Load	Locality
Sub-sample	3	poor	low
Fixed	2.25	good	strong
Arbitrary	3	strong	good
Partial Valid	$6x^2 - 3x + 3$	strong	strong
Line	2	good	strong
Projection	1.5	not a factor	strong

Table 1: Characterization of different reconstruction methods

Based on the target platforms performance in the above metrics you could determine what kind of load it can take to give a desired frame rate, where load is the average necessary texture bandwidth per frame.

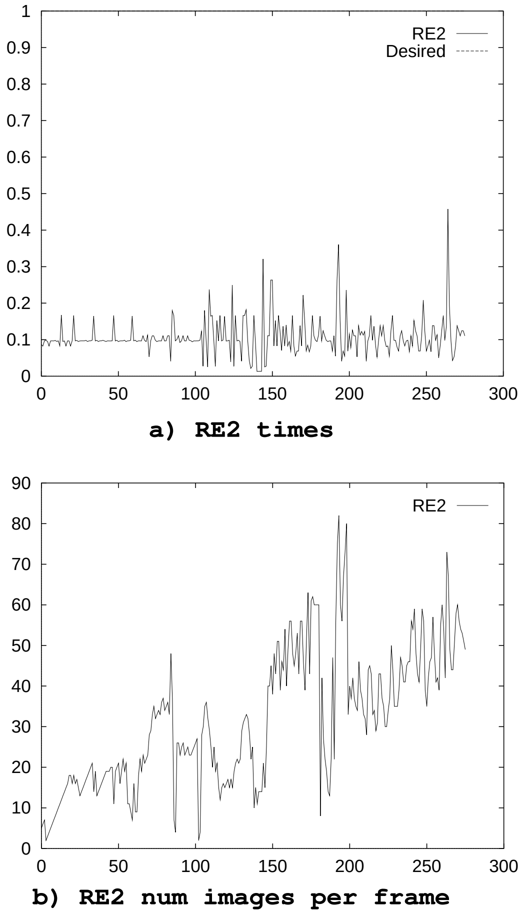

Figure 5: Timing results for RE2

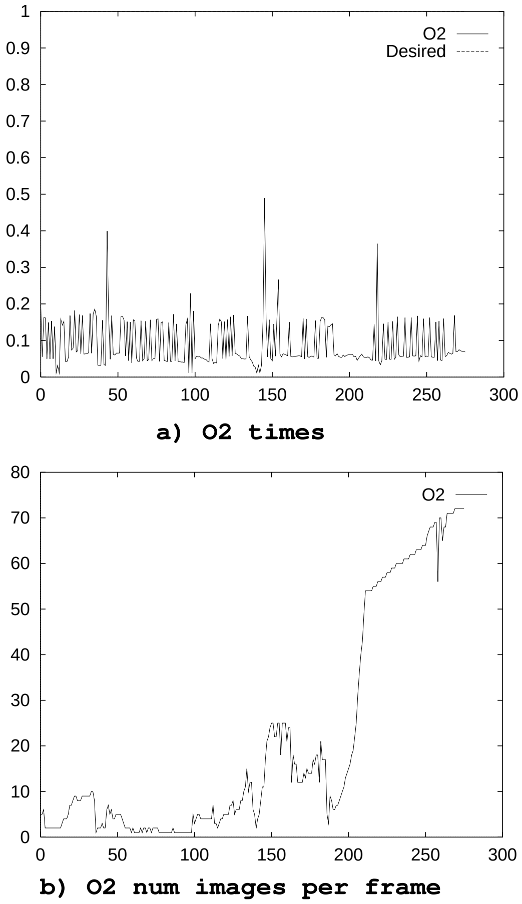

Figure 6: Timing results for O2

5.2 Critical Time Rendering

Figure 5 shows our intial results using the critical time rendering strategy previously defined running on an SGI Reality Engine 2. The graphs represent time per frame (where the user asked for 10 FPS) and the number of textures that were used for each frame. In general, the strategy leads to a reasonably constant frame rate.

The spikes in the graphs appear to be due to a bug in the `gl` implementation of a function that determines which textures are actually in texture memory. In addition, the individual costs (rasterization, texture paging, JPEG decompression) are not dynamicaly scaled, but rather the computations as a whole are scaled by the same factor. Individual scaling factors will also lead to a more constant frame rate.

5.3 Issues related to the O2 Architecture

Silicon Graphics Inc., has recently begun delivering machines based on a new architecture which they dub the O2. This design uses main memory to hold both the frame buffer (i.e., it has no separate frame buffer) and textures (i.e., no separate texture memory). A performance price is paid for this in terms of raw rendering speed (number of textured polygons per second). However, for the Lumigraph application this is generally a win since we render relatively few textured triangles, but with many textures. The bottleneck of severely limited texture memory is lifted and replaced only with the limitations of main memory. There is also hardware support for asynchronously decompressing JPEG images. Many of the strategies outlined above are still valid while others are of reduced importance. We have made some initial experiments on this machine as well by adjusting the cost function for individual nodes. We have not yet taken advantage of the JPEG decompression hardware, but the initial results are promising. Figure 6 shows the results of the same sequence shown earlier for the RE2, when run on the O2 machine.

6 Conclusion

At SIGGRAPH 1996, Levoy and Hanrahan [10] and Gortler at al [8] showed how a four dimensional data structure, called a Lumigraph in the latter work, could be constructed to capture the full appearance of a bounded object or the light entering a bounded region of empty space. Methods were presented to quickly reconstruct images from the Lumigraph from arbitrary objects. Unfortunately, these methods are limited by the image resolution in the methods in the first paper and by limited texture memory in the latter work.

In this paper we have shown a number of fast approximate reconstruction methods for images from Lumigraphs using the texture mapping process described in Gortler et al. These methods fall into two classes, those that extend the use of a limited or constant number of textures per image, and a method that uses the current image as a texture itself in subsequent images. The second type of reconstruction relies on the approximate geometry of the object being represented by the Lumigraph. The trade-offs of speed versus artifacts varies for each of these methods.

These reconstruction methods form the basis for developing strategies for critical time rendering from Lumigraphs. One such strategy based on a cost/benefit analysis for each node is discussed in the context of limitations of texture memory and disk access and decompression time.

More work needs to be done to discover optimal strategies for rendering and transmission of Lumigraphs given the varying constraints of hardware and networks.

References

[1] Beers, A. C., Agrawala, M., and Chaddha, N. Rendering from compressed textures. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 373–378.

[2] Benton, S. A. Survey of holographic stereograms. *Proc. SPIE Int. Soc. Opt. Eng. (USA) 367* (1983), 15–19.

[3] Bryson, S., and Johan, S. Time management, simultaneity and time-critical computation in interactive unsteady visualization environments. In *Visualization '96* (1996), pp. 255–261.

[4] Chen, S. E. Quicktime VR - an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 29–38. held in Los Angeles, California, 06-11 August 1995.

[5] Chen, S. E., and Williams, L. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 279–288.

[6] Debevec, P. E., Taylor, C. J., and Malik, J. Modeling and rendering architecture from photographs: A hybrid geometry-and-image-based approach. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 11–20.

[7] Funkhouser, T. A., and Séquin, C. H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 247–254.

[8] Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F. The lumigraph. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 43–54.

[9] Hoppe, H. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 99–108. held in New Orleans, Louisiana, 4-9 August 1996.

[10] Levoy, M., and Hanrahan, P. Light field rendering. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 31–42.

[11] McMillan, L., and Bishop, G. Plenoptic modeling: An image-based rendering system. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), R. Cook, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 39–46. held in Los Angeles, California, 06-11 August 1995.

[12] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., and Haeberli, P. E. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (July 1992), E. E. Catmull, Ed., vol. 26, pp. 249–252.

[13] Seitz, S. M., and Dyer, C. R. View morphing. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 21–30.

[14] Szeliski, R. Rapid octree construction from image sequences. *CVGIP: Image Understanding 58*, 1 (July 1993), 23–32.

[15] Torborg, J., and Kajiya, J. T. Talisman: Commodity realtime 3d graphics for the pc. In *Computer Graphics Proceedings, Annual Conference Series, 1996* (1996), pp. 353–363.

[16] Williams, L. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)* (July 1983), vol. 17, pp. 1–11.