Lazy decompression of surface light fields for precomputed global illumination

Dr. Gavin Miller, Dr. Steven Rubin, Interval Research Corp. and Dr. Dulce Ponceleon, IBM Almaden Research Center

miller@interval.com

Abstract. This paper describes a series of algorithms that allow the unconstrained walkthrough of static scenes shaded with the results of precomputed global illumination. The global illumination includes specular as well as diffuse terms, and intermediate results are cached as surface light fields. The compression of such light fields is examined, and a lazy decompression scheme is presented which allows for high-quality compression by making use of block-coding techniques. This scheme takes advantage of spatial coherence within the light field to aid compression, and also makes use of temporal coherence to accelerate decompression. Finally the techniques are extended to a certain type of dynamic scene.

1 Introduction

1996 saw the introduction to the graphics community of the idea of light fields and lumigraphs. The appearance of a scene or isolated object is captured in terms of a 4-dimensional table, which approximates the plenoptic function sampled on an image plane for light fields [11], and on an envelope surface in the case of lumigraphs [5]. Reconstruction of views for locations not on the light field image plane is then possible using interpolation of table entries, typically using quadralinear interpolation.

Light fields have a number of advantages - the rendering time is independent of scene or object complexity; they may represent arbitrary illumination functions and they may be captured for isolated real objects and scenes. However, light fields also have a number of limitations. The camera is restricted to certain regions of space and the finite angular resolution leads to depth of field effects. Objects become blurry in proportion to their distance from the image plane. To overcome these restrictions the lumigraph paper [5] resampled the light field of an object onto a crude approximation of the surface geometry. This helped to reduce the depth of field effect and allowed the camera to be placed in any region outside the approximating surface. In related work, the idea of view-dependent texture mapping was introduced in [4], where a surface texture is created from a blend of images depending on the direction from which it is viewed. A small number of views were combined using blending, or a stereo algorithm was used to reconstruct intermediate views of surface detail. The method was unable to capture detailed surface reflectance effects because of the sparse sampling of orientation space for each surface point. A fine sampling of orientation space leads to an explosion in the memory requirements. It is the use of compression algorithms to best handle such data that forms the basis of this paper.

1.1 Goals of this paper

In this paper, we explore the use of surface light fields to capture the appearance of global illumination for synthetic scenes. The particular examples were generated with ray-tracing, since the specular terms present the most difficulty in terms of sampling and compression, but effects due to interreflection [2] could be incorporated with little difficulty. This approach enables us to explore compression issues for a scene that contains specular objects.

We have a number of objectives for the resultant algorithms. The camera should be free to move anywhere in the scene. Coherence in the light fields should allow for efficient data compression where applicable. Efficient decompression should be accomplished by making use of temporal and cache coherence. (In this case, temporal means from frame-to-frame as the camera moves. However there is also cache coherence within a single frame.) We wish to match the angular resolution of the light fields to the material reflectance properties of the surface and the resultant radiance distribution. Perfect specular reflectors will have higher angular frequencies than nearly diffuse surfaces. Lack of angular resolution in the light fields should appear as changes in surface properties rather than lack of depth of field in the camera. In a progressive refinement scheme, for instance, the surfaces would start diffuse and then become more and more crisply specular.

Unfortunately to make our scheme efficient, the requirement that rendering cost be independent of scene complexity has to be abandoned. By placing the light field directly on the surface we are no longer using light fields to represent detailed surface geometry. They now only store the variations with angle of surface radiance.

The rendering scheme for this paper works in the following way:

- As a pre-processing step, a surface light field is generated for each parametric surface in the scene. This is a four dimensional table for each surface indexed by two surface parameters and two orientation parameters. The light fields are generated using stochastic ray tracing. Twenty-five rays per sample were used for the examples in this paper. The indices were jittered to prevent aliasing in both surface location and ray orientation.
- During real-time interaction, a texture map is generated for each (non-diffuse) surface every frame. Each texture pixel is computed using the position of the camera relative to the surface. The textures are downloaded to the graphics hardware and the geometry is scan-converted.

1.2 Structure of this paper

Section 2 discusses the parameterization of the light fields, and efficient resampling on playback. Section 3 talks about the internal representation of the light fields. Section 3.1 describes the block compression scheme and its internal representation. Section 3.2 explores the use of cache coherence as part of a block decompression scheme. Section 3.3 extends the block-coding algorithm to make use of angular coherence within the light field. Section 4 explores the use of light fields to render a dynamic scene in which a single surface is bump mapped on the fly using a water ripple simulation.

2 Geometry of surface light fields

The examples for this paper consist of smooth parametric surfaces. In a spirit similar to [5] we sample the light field on each parametric surface directly. (We assume that the surfaces are closed, otherwise a light field would be required for each side.) The location on the parametric surface is defined by *P(U, V)* whereas the orientation relative to the tangent coordinate frame is defined by S and T. Figure 1 illustrates the surface parametrization and the tangent coordinate frame.

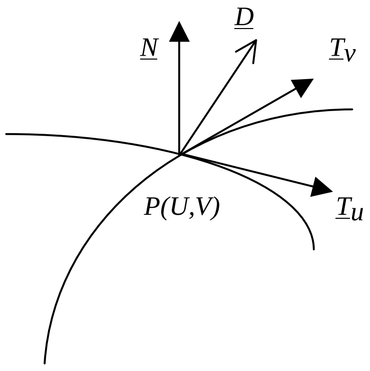

Fig. 1. Parameterization of the surface light field.

Unlike [11], we decided to use the non-linear mapping between (*S*, *T*) and orientation given in Equation 1. This is because we wished to capture a hemisphere of orientation in a single 2-D parameterization, where (S, T) span the range from zero to one.

$$s = 2S - 1$$
$$t = 2T - 1$$
$$n = \sqrt{1 - s^2 - t^2}$$

$$\underline{D} = s\underline{T}_U + t\underline{T}_V + n\underline{N} \quad (1)$$

Where $\underline{D}$ is the view direction vector, $\underline{T}_u$ and $\underline{T}_V$ are the surface tangents and $\underline{N}$ is the surface normal. When n is not a real number (because of a negative square root) n is set to zero. This corresponds to values for S and T that are outside the surface's orientation hemisphere. Note that the light field captures the light leaving the surface in the $\underline{D}$ direction. For an idealized purely diffuse surface, the light leaving the surface will be independent of (S, T). A degenerate light field results, which may be expressed as a single texture map. This case will occur frequently in practice.

2.1 Efficient resampling

To reconstruct the surface texture for a given camera location, it is necessary to compute the (S, T) values for each texture pixel at (U, V) and then index into the light field using (U, V, S, T). This could be done at each texture pixel using the surface equation. In practice, we approximate the surface using a finite resolution mesh. At the vertices of this mesh we compute (S, T). We then scan-convert the polygon into the texture map interpolating S and T values in the interior of the polygon.

We can compute (S, T) at polygon vertices using:

$$S = \frac{\underline{D}.\underline{T}_u + 1}{2}$$

$$T = \frac{\underline{D}.\underline{T}_v + 1}{2}$$

Unfortunately, the use of linear interpolation of S and T over a surface region in U, V parameter space is inexact. To minimize errors, large polygons are diced into smaller ones for the purpose of creating the texture (rasterizing in U, V space), but the larger polygons are used directly for textured polygon rendering in screen-space. For the examples in this paper, the polygons were diced if they were larger than 8 texture pixels along a side.

An advantage of parametric light fields is that they may be reconstructed with bilinear interpolation rather than quadralinear interpolation. This is because the surface texture is already aligned with the surface (U, V) space. A second resampling occurs when the surface is rendered using texture mapping, which is done efficiently by standard hardware.

2.2 Visibility culling and back-face culling

The cost of resampling a light field to create a surface texture is significant, both because of the interpolation cost and also, where used, the decompression cost. For this reason, such resampling is to be avoided whenever possible. When a surface is not visible in the current view, its texture does not need to be computed from the light field. Such surfaces may be culled using a visibility precomputation such as [6].

A special form of visibility culling, worth particular mention, is back-face culling. For a light field on a closed surface, back-face culling may remove up to half of the triangles. These regions will not have a texture computed, and the savings are useful. However, having undefined regions of the texture image can lead to problems when generating a Mip map [15]. The examples for this paper were scan-converted using bilinear interpolation of the full resolution texture pixels without pre-filtering. The texture maps were of sufficiently low resolution that they did not alias. If, on the other hand, the surfaces were being rendered using Mip-mapping, we would need to compute a Mip map for each surface texture, each time that the texture changed. Unfortunately, portions of the texture would be undefined in back-facing regions.

Those regions may contribute to visible portions of the final Mip map because of blurring. This can happen near an outline where the effective filter footprint of a pre-filtered region straddles the border of the visible region in texture space. Resolving this problem is an area for future work.

3 Compression

Given the large data size of high-resolution light fields, there is an obvious need for compression. A 256-squared by 64-squared light field takes 803 Megabytes of uncompressed data. In many ways this is comparable to the compression problem with video sequences. However, there are some key differences. As stated in [11], the most salient difference is the need to only access a sparse sampling of the decompressed light field at any frame.

In [11], vector quantization was used to make use of this property. VQ has the advantage that it allows for local decompression, so a random sample may be decompressed in an efficient manner. Unfortunately, the use of VQ in isolation does not allow for the most memory efficient compression since coherence in the image is not fully exploited. Also, transform-based methods, such as Discrete Cosine Transform (DCT) [3], may allow for compression with less artifacts at comparable compression ratios, and have a more naturally scalable quality control mechanism. In [5], it was suggested that transform-based block coding techniques could be used instead, but few details were given. This section explores issues related to block coding strategies including cache coherence and angular coherence.

3.1 Light field as an array of images

If we decide to store a light field as a 2-dimensional array of 2-D images, then it is important to decide which representation to use. It was pointed out in [11], that we could have images that were adjacent in U, V space or S, T space. We chose to store images that span (U, V) space as contiguous pixels for a given value of (S, T). This choice was based on the characteristics of the decoding scheme rather than the degree of coherence in the data. (This will be addressed further in Section 3.2.)

It is worth noting that for dull-specular surfaces, their appearance may be characterized adequately using surface light fields with angular resolutions of (S, T) which are far less than the spatial resolution of the texture (U, V). This arises from the surface microstructure acting as a low-pass filter on the angular dependence of the surface-bidirectional-reflectance-distribution function [1]. Since we are sampling discrete orientations, we will refer to the discrete directions as S_i and T_i. If the angular resolution is low, i.e. adequate for a dull reflector, then S_i and T_i vary only slowly over the surface in a particular view, being the same for many adjacent pixels in U, V space. (Of course, for proper antialiasing, the S_i and T_i each really refer to intervals of orientation.)

A resolution of 16 by 16 (S, T) values, as compared to 128 by 128 (U, V) values, leads to a plausible looking dull-reflective surface, as is illustrated in Figure 2a and Figure 2b (see Appendix). Figure 2a shows a typical U, V slice of the surface light field for the reflective floor in Figure 2b. Since the floor is planar, this corresponds to an orthographic projection of the scene. The depth of field effect is achieved using distributed ray tracing to jitter the S and T values (and hence the direction) of the rays. The visible shadow is seen because of the Phong highlight component of the surface shader [14]. Computation of this highlight involves casting a ray to the light to see if the region is illuminated. For this particular surface there is no diffuse component of the shader, so the shadow only becomes apparent when the highlight is near the shadowed region.

In contrast to the planar floor example, Figure 3a shows a surface light field for the reflective bottle in Figure 3b (see Appendix). Ripples in the orientation of the bottle surface create the striped appearance of the light field U, V slice. The angular S, T resolution was 32 by 32. Correct reflections are seen, which are more accurate than those from reflection mapping, and have the expected depth of field effect. Regions close to the bottle surface are seen reflected in crisper focus, those further away are seen in softer focus. However, the bottle itself and the surrounding scene remain crisp.

3.2 Block coding and temporal coherence

A goal of this paper is to explore the use of block coding schemes to represent light fields. Image compression schemes such as JPEG typically store an image as a set of 8 by 8 blocks [13], each of which is converted into DCT coefficients and then Huffman coded to remove redundancy [8]. This is especially efficient for whole images, since, during decoding, all of the image pixels are required.

Unfortunately, if each light field U, V slice is stored as an array of blocks, then we may need to decode an entire block to access one of its pixels. Vector quantization was used to overcome this problem in [11], by making use of the ability to efficiently decode pixels within a block. In DCT-based methods the inverse DCT may be used to find single pixels. However, Fast Fourier Transform methods are much more efficient when decoding several pixels in a single block. In addition, the Huffman decoding scheme has to reconstruct all of the coefficients, which is a significant cost. To make use of such block coding, we too dice the U, V slice images into blocks of 8 by 8 pixels. The light field is stored as an array of compressed blocks or C-Blocks, each of which may be decompressed into an uncompressed block or U-Block. Figure 4 (see Appendix) shows examples of compressing Figure 2b at different compression ratios. At 36 to 1, the compression artifacts show up clearly in the reflections. At a compression ratio of 24 to 1 the artifacts are less apparent. We found that at 11 to 1, the images were indistinguishable from Figure 2b.

Given a block-based encoding scheme, we need an algorithm to decompress blocks on the fly during real-time rendering. A compute-hungry algorithm is as follows:

> For a given (U, V, S, T) decompress the corresponding C-Block and then sample the U-Block. (In practice you will need to decompress 4 blocks if bilinear interpolation is being used for fractional values of (S, T).)

This is the most memory efficient scheme, but requires a block (or four) to be decompressed for every sample in the U, V texture map.

As noted in Section 3.1, low angular resolutions lead to discrete orientations that vary only slowly across the surface. This implies that a pixel in U, V space will often need to decode information from blocks accessed by its neighbors. A second observation is that, if the camera moves only slowly, the same blocks will probably be accessed in the subsequent time frame. For infinitesimal camera motions, all of the same blocks will be decompressed from frame to frame. This effect will decrease as the angular motion of the camera relative to the surface region approaches the interval between S, T blocks.

A memory-hungry algorithm, that exploits these ideas, is as follows:

> For each (U, V, S, T) interpolated in the texture map, decompress the corresponding C-Block, and keep it in memory. Subsequent accesses merely use the U-Block directly.

In practice, of course, there is a continuum between the two algorithms. By employing a block manager, we can keep the last N decompressed blocks in a block cache, only decompressing additional blocks when required. The block manager size determines the number of blocks that must be decompressed each frame.

A test was made by repeatedly rendering Figure 3b for a static camera. This has an S, T resolution of 32 by 32 images. Table 1 shows the results of this experiment. Rendering times are for a 180 MHz PowerPC microprocessor with an Apple QuickDraw3D accelerator card. The surface U, V map was computed at 256 by 256 pixels. (Uncompressed, this is a 200 Mbytes of data consisting of 1,048,576 blocks.) For reference, it is worth noting that the number of blocks accessed for that frame is 10,194. When the cache size exceeds this value, all of the blocks are kept in cache from the previous frame and the decompression count goes to zero. This is an expression of the frame-to-frame coherence of the cache.

When the cache is just slightly smaller than the number of accessed blocks, all of the blocks need to be decompressed for each frame. With an even smaller cache, each block has to be decompressed many times each frame, since it has been flushed from the block manager before it needs to be reused. It is worth noting that little performance is lost when the cache size is reduced from 10000 to 100. This is because a triangle of the tessellated surface accesses the images in a small region that can be mostly cached in 100 blocks. It is also worth noting that use of no compression block cache, i.e. storing the images as arrays directly, speeds up the rendering by a factor of 7 in this case. In contrast Table 2 shows similar data for the bottle with an S, T resolution of 16 by 16 (and 4 times less data). For small block cache sizes, the timings are similar with the number of block decompressions determining the frame rate. However, when the cache is large enough to store all of the decompressed blocks, the rendering time is only 2.5 times longer than the uncompressed case. We suspect that this performance improvement relative to the 32 by 32 case arises because the recently visited blocks are stored within the processor cache.

Table 1. Block decompressions versus cache size for 32x32 Silver Bottle.

Cache Size in Blocks	Time in Seconds	Blocks Decompressed	Block Decompressions
Uncompressed	0.24	Uncompressed	Uncompressed
10194	1.75	0	0
10193	2.44	10194	10194
1000	2.44	10194	10194
100	2.99	10194	1,042
10	4.84	10194	47646
4	8.15	10194	97646
1	16.18	10194	226088
0	17.59	10194	226332

Table 2: Block decompressions versus cache size for 16x16 Silver Bottle.

Cache Size in Blocks	Time in Seconds	Blocks Decompressed	Block Decompressions
Uncompressed	0.23	Uncompressed	Uncompressed
6136	0.59	0	0
6135	1.01	6136	6136
1000	1.01	6136	6136
100	1.19	6136	9587
10	2.1	6136	25288
4	4.6	6136	65876
1	12.7	6136	197274
0	13.8	6136	197320

To measure frame-to-frame coherence, a camera was moved relative to an object by a given linear distance. The test object was the glass cube in Figure 5 (see Appendix). Initial camera distance from the cube was 5 units of length. The camera was moved perpendicular to the view vector to the cube. When rendering the resultant new view, some additional blocks needed to be decompressed because the S, T values for the visible surface region depend on the position of the camera relative to the local surface frame.

It is instructive to measure the number of block decompressions as the linear distance is increased. Table 3 shows the number of blocks to decompress (assuming a very large cache size) as a function of the distance moved. This is recorded for the same scene but with different surface light field angular resolutions. In the 4 by 4 case, the frame-to-frame coherence meant that no new blocks needed to be decompressed up to a motion of 0.5 units. As the angular resolution is increased, the number of decompressed blocks increases more rapidly with camera motion. So, as expected, the frame-to-frame coherence is most applicable to fairly low angular resolution light fields. Cache coherence within the same frame, however, is still useful as was shown in Table 2.

3.3 Block coding and angular coherence

An important area for improvement is the use of coherence within the light field images to improve the compression rates. In the future, a general motion compensation scheme may be applicable in the style of MPEG [12] and low bit-rate coding schemes such as [9]. Unfortunately, such a scheme is harder to implement in a block caching framework, since motion difference blocks may depend on four or so blocks from an S, T adjacent (in MPEG previous) image. The number of decompressed blocks will grow alarmingly with the number of motion-compensated (in this case angle-compensated) frames.

Table 3. Block decompressions versus camera motion.

Offset Distance	64x64	32x32	16x16	4x4
0.01	29	8	0	0
0.1	148	56	0	0
0.2	337	90	6	0
0.5	583	312	237	0
1.0	828	800	539	256
2.0	847	800	796	256
5.0	861	798	770	512

Fortunately, given the unique structure of surface light fields, there are examples where a very simple frame-differencing scheme is helpful. Consider the box in Figure 6a (see Appendix). Some regions of each face are purely diffuse, with a marble texture. Other regions are reflective in a decorative pattern. This is typical of a composite shader [7].

For purely diffuse regions, the image blocks will be identical for changing values of S, T. By comparing each block to the corresponding block with S and T values of zero, we can detect blocks which do not change. These can be coded with a single bit rather than storing the image coefficients. By detecting such blocks it is possible to double the compression ratio in the example of Figure 6a. Figure 6b was rendered with 17 to 1 block compression, but with the additional use of identical-block elimination, the compression ratio increased to 30 to 1.

A second approach to exploiting angular coherence is to compare a block with its corresponding neighbors in S, T space assuming some relative motion. We implemented a scheme in which a block was compared to its left neighbor in S space. MPEG style motion compensation was used to find the optimal offset between neighboring blocks. The key difference was that pixel queries outside of the reference block merely replicated the value of the nearest pixel inside the block (rather than decoding other adjacent blocks). For offsets of a few pixels, the overlap was still most of the block area. Differences between the offset adjacent block and the current block were encoded and stored. Every eighth block was made a key-block and encoded without motion compensation. Our experience with this scheme was disappointing. For the example of the silver bottle in Figure 3b (see Appendix), best results were found with an angular (S, T) resolution of 64 by 64. For comparable image quality the compression ratio was doubled. However, for smaller angular resolutions, the compression improvement was less noticeable. This is because the amount of angular change increases between blocks as the angular resolution goes down. Motion vectors need to be stored with the blocks and the motion is poorly approximated by a uniform offset. A better scheme for encoding angular coherence is an important area for future work.

4 Dynamic scenes

So far we have considered static scenes. An area for future work is how far these ideas may be generalized to dynamic scenes. In general this is very difficult, but there is one simple example where we can add some degree of animation and get accurate results.

Consider the scene in Figure 7 (see Appendix). A shiny floor reflects a room and a ball. We can use the light field for the floor to not only render the appearance of a flat floor, but also that of any bump-mapped reflective floor in the same location. Bump mapping changes the orientation of the surface normal, which may be used to compute the reflected ray direction. The reflected ray then gives an associated viewing direction for that surface point viewed in the flat case. This, in turn, is used to find the reflected ray color by indexing into the surface light field. This will give accurate images of the scene, provided that the other objects do not in turn show reflections of the floor.

The algorithm proceeded as follows:

- Detect hit events from the mouse. If the water polygon is hit, pass the U, V location of the hit to the water simulation, adding a disturbance at that location.
- Compute the water depth values based on a finite difference equation [10].
- Compute the normals of the water depth.
- Compute the reflected ray direction for each texture pixel.
- Index into the surface light field to compute the reflected color and place in the corresponding texture pixel.
- Download the completed texture to the rendering hardware.
- Scan-convert the scene.

Figure 7 illustrates the results of this algorithm. The water simulation was computed on a 64 by 64 grid and then the texture was computed using reflected ray S, T values interpolated up to 128 by 128. Figure 7 was rendered at 13 frames per second when no compression was used, and at (a maximum of) 6 frames per second when using 10 to 1 compression. When the surface was newly disturbed, the frame rate dropped to 2.5 frames per second. This was with a block cache that was double the size required to cache the same view of the smooth water. Figure 8a (see Appendix) shows the cache map for the flat floor. Each small box is a U, V image for a given value of S, T. Each pixel in the image represents an 8 by 8 block of U, V pixels in the light field. Blocks decompressed this frame are shown as red pixels. Those previously cached are shown in green. Because of the smooth surface and relatively distant camera, the blocks are clustered in one region of the S, T map. Since the cache is large, all of the blocks are green. In Figure 8b, on the other hand, the cache map shows a much more scattered structure and the red blocks indicate a large number are being decompressed this frame (about a quarter of the number of blocks in the cache).

Note that unlike the use of reflection maps for water ripples, these ripples have the correct appearance, as if they were being ray-traced on the fly. This is apparent from the reflections of the vertical lines on the walls. The reflections near the base of the wall wiggle less than those further from the wall, a characteristic of ray-traced images of rippling water, as opposed to reflection-mapped images which tend to wiggle different regions of objects all by the same amount. As given, the technique would be best applied to architectural scenes with ornamental ponds. To extend the scheme to more general dynamic scenes is an area for future work.

5 Conclusions

We explored the use of surface light-fields for walk-throughs of pre-ray-traced scenes. A caching scheme for decompressed blocks was introduced to allow the use of transform coding techniques with near real-time playback. Block cache coherence led to order of magnitude improvements in efficiency for on-the-fly decompression. It was found that frame-to-frame coherence was most effective for low-angular resolution light fields (up to about 16 by 16). However, cache coherence within the frame was effective for all angular resolutions, and great performance increases were achieved for relatively small block caches. Finally a simple example was given of a dynamic scene rendered accurately using a precomputed light field.

While the block-based compression schemes proved effective, the compression ratios achieved for good image quality were rarely greater than 20 to 1. To achieve higher compression ratios with comparable quality, a different formulation will probably be required. However, it is worth noting that the choice of angular resolution to match the angular content of the surface radiance was already a form of compression. Other experiments with light field compression tend to concentrate on mostly diffuse surfaces, which yield higher compression ratios. However, with our formulation, no compression of the angular component would be required for such diffuse surfaces. The trade-off is the requirement for more complexity in the geometric model.

6 Acknowledgements

This work was completed at the Graphics Research Group of Apple Research Laboratories at Apple Computer, Inc. in Cupertino, California. Special thanks to Douglass Turner for the use of his ray tracing code.

References

1. Arvo, James, "Application of Irradiance Tensors to the Simulation of Non-Lambertian Phenomena", Computer Graphics Proceedings, Annual Conference Series, 1995, pp. 335-342.

2. Chen, Schenchang Eric, Holly E. Rushmeier, Gavin Miller, Douglass Turner, "A Progressive Multi-pass Method for Global Illumination", Computer Graphics, Vol. 25, No. 4, July 1991.
3. Chen, W. H., C. H. Smith and S. C. Fralick "A fast computational algorithm for the discrete cosine transform," IEEE Trans. Commun., vol. COM-25, pp. 1004-1009, Sept 1977.

4. Debevec, Paul E., Camillo J. Taylor, Jitendra Malik, “Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach”, SIGGRAPH ’96, Computer Graphics Proceedings, Annual Conference Series, 1996. pp. 11-20.

5. Gortler, S. J, Radek Grzeszczuk, Richard Szeliski, Michael F. Cohen, "The Lumigraph", Proc SIGGRAPH '96 (New Orleans, Louisiana, August 4-9, 1996). In Computer Graphics Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 43-54.

6. Greene, Ned, "Hierarchical Polygon Tiling with Coverage Masks", Proc SIGGRAPH '96 (New Orleans, Louisiana, August 4-9, 1996). In Computer Graphics Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 65-74.

7. Hanrahan, Pat, and Jim Lawson, "A Language for Shading and Lighting Calculations", Computer Graphics, Vol. 24, No. 4, (August 1990), 289-298.

8. Huffman, D. A., "A method for the Construction of Minimum-Redundancy Codes," Proc. IRE, 40(9):1098-101, Sept 1952.

9. ITU-T (CCITT) "Recommendation H.261: Video Codec for Audiovisual Services at p x 64 kbits/s". , Mar 93.

10. Kass, Michael, Gavin S. P. Miller, "Rapid Stable Fluid Dynamics for Computer Graphics", Computer Graphics, Vol. 24, No. 4, Aug. 1990. pp. 49-58.

11. Levoy M., and P. Hanrahan, "Light Field Rendering", Proc SIGGRAPH '96 (New Orleans, Louisiana, August 4-9, 1996). In Computer Graphics Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 31-41.

12. Mitchell, J. L. , W. B. Pennebaker, C. E. Fogg, and D. J. LeGall, MPEG Video Compression Standard, Digital Multimedia Standards Series, Chapman & Hall, New York, 1997.

13. Pennebaker, W. and J. L. Mitchell, JPEG Still Image Data Compression Standard. Van Nostrand Reinhold, New York, 1993. ISBN 0-442-01272-1.

14. Phong, B. T., "Illumination for computer generated pictures", Communications of the ACM, Vol. 18, No. 6, (June 1975), pp. 311-317.

15. Williams, Lance, "Pyramidal Parametrics", Computer Graphics (SIGGRAPH '83 Proceedings), Vol. 17, No. 3, July 1983, pp. 1-9.