A Dynamic Interactive Real-Time Light Field Renderer

Evan Parker and Katherine Chou

Stanford University Graphics Lab

Abstract

This paper describes the design and implementation of a dynamic light field renderer for interactively viewing real-time light fields with minimal latency. Our light field data consists of synchronized footage captured from 100 commodity video cameras. The system we constructed employs a hybrid of rendering algorithms in order to amortize latency artifacts, which can arise from bandwidth, memory, and processing limitations. Our method demonstrates the advantages of region-based image loading from individually compressed frames using the JPEG2000 encoding and decoding scheme as compared to region-based seeking in raw images straight off disk. The techniques we developed divorce the rendering time from the number of cameras in the system, which allows us to scale the system without performance degradation. Our system specifications were deliberately engineered to minimize perceptible latency while relaxing as many assumptions about the scene geometry and camera configuration as possible. In this paper, we also provide an examination of the relatively unexplored facet of fast light field rendering with respect to *dynamic* scenes. We present early experimental results from our prototype of the first undistributed dynamic light field (DLF) renderer.

1. Introduction

With the advent of multi-camera arrays that can not only capture but also store entire light fields on disk, the question is how do we leverage such a massive amount of visual data to its fullest? Image-based rendering (IBR) has lead to a relatively easy re-creation of photorealistic static scenes often produced by a single camera on a gantry or snapshots from a sequence of cameras in one instance of time. Lately, systems and algorithms have been developed to address the issue of constructing an IBR that incorporates a temporal dimension.

None have yet considered a real-time IBR of a dynamic scene that allows complete navigational freedom in all 5 dimensions of a light field, 4 spatial and 1 temporal. In particular, we believe we are the first to implement a scalable architecture that allows this without reconstruction of a geometric model of the scene or hardware enhancements. This is not surprising since storing, transferring and processing such a large volume of data from a 100 densely packed cameras is a nontrivial problem.

To achieve our goal, we chose to design our IBR based on fast decompression and region-based image selection. Region-based pixel selection bounds the total amount of pixels that is ever needs to be loaded into memory for recreating a novel view. Fast decompression minimizes bandwidth requirements and decoding time. Our proposed rendering algorithm is scalable in the number of cameras and shifts the work of data retrieval to the processor, removing bandwidth between disk and memory as a bottleneck. At the same time, we tried to maintain generality with regards to data specifications for our light field renderer, although we chose a tightly packed camera grid configuration to minimize the amount of calibration and scene geometry calculations necessary.

In our work, we have made the following contributions:

1. An interactive light field renderer that can handle both static and dynamic scenes.
2. Optimized rendering algorithm to suppress latency using fast decompression and region-based data selection to perform just-in-time rendering.
3. A flexible system that can hot-swap data specifications, i.e. using compressed vs. uncompressed data or using cache-based vs. region-based data selection.

2. Previous Work

2.1 Light field and Lumigraph Rendering

Levoy and Hanrahan [1] and Gortler et al. [2] first introduced image-based rendering techniques for viewing a static light field. Since the publication of these papers, there has been research investigating time-critical lumigraph renderings by Sloan et al.[3] and dynamic reparameterization of light fields.

The discovery of these very fast algorithms dismissed parameterization calculations as a bottleneck to real-time light field rendering, reducing its feasibility to a data retrieval problem.

As mentioned earlier, there has been increasing research in the area of enhancing IBR to real-time rendering of dynamic light fields. One such system is the real-time distributed light field

camera implemented at MIT.[4] Their system however has the drawback of not being able to record the light field for future playback or generation of a stereoscopic display. Naemura et al.,[5] on the other hand, handled the bandwidth issues of an interactive application by using only 16 cameras and real-time depth estimation hardware. Similarly, Goldlucke et al.[6] developed an interactive dynamic light field renderer that uses 4 cameras and a depth map constructed using computer vision techniques in a preprocessing step to allow the viewer limited motion within the plane defined by the 4 cameras.

2.2 Compression

Dynamic light field compression has not been widely investigated before now, likely due to the difficulty in the acquisition of dynamic light fields.

In terms of static light field compression, Levoy and Hanrahan[1] discussed a compression method that uses vector quantitization and a codebook to compress the lightfield across all four dimensions. Another scheme for efficient light field compression is multiple reference frame (MRF) encoding, which uses an MPEG algorithm to predict some camera images from neighboring images.[7] Recent research by Chang et al.[8] discusses using four dimensional discrete wavelet transform (DWT) combined with disparity-compensated lifting (similar to MRF) to achieve superior compression efficiency and scalability.

2.3 Camera Array

We are using the Stanford multi-camera array built by Wilburn et al.,[9] which we configured into a 10x10 densely packed grid arrangement. Our light field renderer compliments the array well because both are targeted at compression, storage, and scalability.

3. Acquisition

Each camera captures progressive VGA video at 30fps in real-time. The cameras are synchronized by special-purpose hardware and stores the video streams to disk in PNG format for our purpose. For our first dynamic data set we acquired 20 frames of video for a DLF that is 2/3 of a second long.

4. Design Considerations

4.1 Storage and Compression of the DLF

Dynamic light fields are big. 100 video streams at 30fps, 640x480 resolution, and 24 bits/pixel is approximately 22 Gigabits/sec. Disk space is cheap, but not that cheap. Thus compression of the DLF is a consideration.

In choosing how to store a DLF we had two considerations: storage space and random access speed. On the one hand we would like to compress the DLF as much as possible. On the other hand, in order to provide viewpoint-, camera-, and resolution-scalability, we need essentially random access to the image representing a particular frame within the video stream of a given camera, and within that image random access to a particular region of the image at a particular resolution. Unfortunately these two needs generally conflict: better compression results in slower random access times.

With these two considerations in mind we explored various forms of DLF storage and compression. A DLF is a 5-dimensional space: 2 dimensions in each image, 2 dimensions across the array of cameras, and 1 temporal dimension. Ideally we could take advantage of coherence across all five of these dimensions to achieve high compression ratios. However, given the scope and time-frame of our project as well as our background, we decided to stick with already-existing forms of compression rather than try to invent our own. As no one has researched DLF compression this left us to consider forms of compression that take advantage of only some of the 5 dimensions of coherence.

4.1.1 Raw Storage

One possibility is to leave the DLF in raw, uncompressed form, with the obvious disadvantage of large storage requirements. We implemented this by storing every frame from every camera in its own PPM file. This has the advantage of making random access quite fast since no decompression is involved and the location of regions of the image within the file is easily determined, so one can just seek to the proper location and read only the necessary pixels. However, random access to various resolutions of each image is not nearly as fast because this would involve reading, for example, every 4th pixel out of the file, which would take just as long as reading every pixel. To get around this problem one could imagine storing separate copies of each image for each resolution. We did not explore this option.

4.1.2 Temporal and Intra-Image Compression

The second option we briefly explored was using MPEG video compression on each of the individual video streams. This method of compression would take advantage of compression in 3 of the 5 dimensions (the two dimensions within each image and across the one temporal dimension). The main disadvantage of MPEG is that random access to a particular region out of one frame in a video stream is hard. There are three types of frames in MPEG: I-frames, which are encoded using only intra-frame DCT compression; P-frames, which are encoded with reference to the previous P-frame or I-frame; and B-frames, which are encoded with reference to both the previous and next I- or P-frames. Thus decoding a region from a P- or B-frame would require decoding regions from nearby frames until an I-frame was decoded. If I-frames are

regularly spaced in the video stream, this time may be bounded, but it is unclear how closely spaced the I-frames would need to be to achieve acceptable random access speeds. Also, MPEG does not support decoding at multiple resolutions, so once again we would need to create multiple, separate MPEG streams for each resolution. For these reasons we decided not to pursue this route.

4.1.3 Inter- and Intra-Image Compression

A third option we looked into involves compressing across the two dimensions of the camera array and across the two image dimensions within each image, but not across time. Two examples of this type of compression are the vector quantization (VQ) method described in Levoy and Hanrahan[1] and the multiple frame reference (MRF) method described in Chang et al.[8] Both of these methods provide good compression ratios while maintaining quality, and allow for fast random access to image regions within a static light field (i.e. one frame of a DLF). However, it is unclear whether these methods would allow for fast random access across frames in a DLF. This is because both methods are meant for static light field rendering and hence store large tables that must be loaded and decoded before any of the light field image information can be accessed. Once again, decoding at multiple resolutions may require storing a separate light field for each resolution. Still, these methods and others like them that take advantage of the coherence between camera views look quite promising; given the time, we would like to explore them.

4.1.4 Pure Intra-Image Compression

The final option we considered was pure intra-image compression, i.e. only within each image - no compression across time or across the camera array. Intra-frame compression makes random access to a particular image fast, but at the expense of not compressing the DLF as much as would be possible using other methods. To this end we chose to work with the JPEG2000 compression standard. JPEG2000 is the successor to JPEG and uses discrete wavelet transform (DWT) based encoding to achieve better compression than JPEG for the same quality. It is generally considered the state of the art in image compression, but it also has a number of features that make it useful to our project. First of all JPEG2000 encodes multiple resolutions of an image within the same file without extra overhead and allows decoding of lower resolution versions of the image in proportionally less time. Second, the compressed bit stream is split up into blocks that represent blocks of pixels in the image, thus allowing selective decoding of only part of an image. Finally, well written JPEG2000 source code and documentation is available for free online, making it quite accessible to us. Unfortunately, as we discovered, even today's fastest processors struggle to decode JPEG2000 images in real-time.

5. Rendering System Overview

Our light field renderer uses OpenGL as opposed to ray tracing for speed considerations. Here we describe the various objects that make up the renderer.

First, the Renderer requests a set of SamplePoints and a triangulation of those sample points of a BlendingFieldSampler. Each SamplePoint represents a point on the surface of arbitrary scene geometry. The Renderer then passes this set of sample points to a WeightCalculator, which returns a set of <CameraIndex, Weight> pairs for each sample point. Each pair in the set of a given sample point represents the weight of the camera specified by CameraIndex for that sample point, and the sum of all Weights in a given sample's set is 1. The Renderer then uses the reorders the information it obtained from the BlendingFieldSampler and the WeightCalculator into a set of CameraTriangles for each CameraIndex. Each CameraTriangle represents a particular triangle in the triangulation given by the BlendingFieldSampler, and a Weight for each vertex (SamplePoint) of the triangle. A particular triangle will be in a camera's set of CameraTriangles if any of the sample points at the vertices of that triangle have positive weights with respect to that camera. (The reason to order things by camera is so as to avoid as many state changes as possible during rendering.) To begin rendering, the Renderer uses the ViewCamera (which stores info about the current viewpoint) to set up OpenGL's ModelView and Projection matrices. Then, for each camera, the Renderer

1. Loads this camera's projection matrix (which is stored by ImageCamera) into OpenGL's texture matrix,
2. Requests from the DLFImageSet (which stores all the images representing a dynamic light field) the portion of the image for this camera needed to cover all triangles in this camera's set, and loads this image as the current OpenGL texture,
3. Draws the triangles in this camera's set one by one, using the location of each vertex (SamplePoint) as the texture coordinates (thus they get mapped into the correct location of the current image by the texture matrix), and the weight as the alpha color component (this lets OpenGL interpolate the blending field across a the triangle).

5.1 Sampling & Triangulating the Blending Field

Sample points on the view plane are typically chosen in an even manner, triangulated, then projected back out into the scene onto the geometric proxy. In the Buehler et al.[2] paper, sample points are chosen from 3 sources:

1) a uniform sampling of the view plane,
2) the projection of camera locations onto the view plane,
3) the projection of the vertices of the geometric proxy onto the view plane.

Triangulation is then accomplished using constrained Delauney triangulation. In our system, the geometric proxy is a focal plane that can be dynamically positioned by the user. If we had scene geometry for a particular dataset it would not be difficult to incorporate that into our system. Since our geometric proxy is just a plane, it contributes no vertices to the sample points, so that leaves a uniform sampling of the view plane and the projected camera locations. Initially, we used both these sets of points and triangulated them using Delauney triangulation, but this turned out to be prohibitively slow. Hence, at the expense of a slight loss in image quality, we decided to not use the projected camera locations. Therefore, we only use a regular sampling of the image plane (see Figure 1). This makes triangulation trivial and linear in the number of sample points.

Projecting the sample points onto the geometric model is just a ray-plane intersection between the sample-point-view-point ray and the focal plane. There is a tradeoff in choosing the number of sample points between the quality of the constructed image and the speed of rendering. We found a 16x16 grid of sample points produces a good balance between speed and quality.

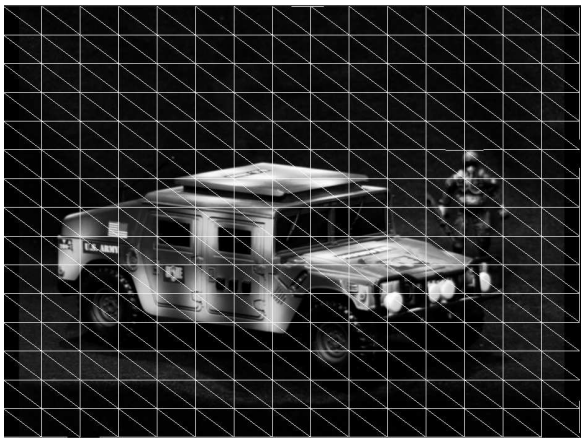

Figure 1: A triangulation of the view plane overlaid on the image constructed from a virtual viewpoint.

5.2 Unstructured vs. Structured Lumigraphs

Our light field renderer is largely based on the triangulation and blending field algorithm we adapted from the *Unstructured Lumigraph Rendering* paper.[2]

The following discussion weighs the tradeoffs between a structured and unstructured lumigraph:

To create a blending field for a lumigraph, we need to determine a set of weights for the k-nearest neighboring cameras (where 'nearness' of cameras is evaluated by angular disparity across camera rays).

In the unstructured lumigraph paper, they assume nothing about the camera positions, which means finding the k "best" cameras (where k is usually ~4) for a given sample and view point requires calculating and sorting the weights for every sample point and every camera. This operation would take

$$O(N * (M + M \log(M))$$

time, where N denotes the number of sample points and M denotes the number of cameras. This becomes prohibitively expensive with 32x32 sample points and one hundred cameras. To reduce this cost we assume the cameras are on a grid, short-circuiting the search for the k "best" cameras and their weight. Now, we simply need to find the intersection of the ray between the sample point and the view point with the plane of the grid and select the 4 cameras surrounding this point of intersection as the "best" cameras. Computing weights thus takes time linear in the number of sample points, which is much more scalable.

5.2.1 Determining the Camera Grid

Our light field renderer accepts an arbitrary grid position for the camera locations, i.e. the x-y plane in the grid is not oriented such that the two sides are parallel to the x-axis and y-axis. We moved to this general strategy to accommodate the data we acquired:

First we need to determine the location and orientation of the grid so that for given a view point we can create a projection matrix that projects the camera grid into the [0,1]^2 square on the x-y plane. This allows us to find the "best" camera locations for any sample point by simply projecting that point using this matrix.

We determine the pose of the grid by calculating the least squares fit plane to the set of camera locations. This allows us to construct a rotation matrix that will place the grid approximately on the x-y plane, with one exception – the sides of the grid may still not be parallel to the x- and y-axes. So we apply another rotation that is obtained by finding the corners of the grid. We achieved this in the following manner:

1. Choose a point in the set and find the point furthest from this point – this will be one corner.
2. Find the point furthest from the first corner - this will be the opposite corner.

3. Find the point such that the sum of the distances from this point to each of corners 1 & 2 is largest - this will be a third corner.
4. Find the point furthest from the third corner - this will be the fourth corner.

Once we have the four corners and the normal of the least squares plane, we have an orthogonal basis that can be used to construct a rotation matrix. Conveniently, the locations of the four corners give us the position of the grid as well.

6 Rendering a DLF in Real-Time

In order to be able to render a DLF in real-time we need to ensure that we can bound the amount of time it will take to render any virtual viewpoint. Specifically, we must ensure that the amount of information needed from the DLF is independent of the viewpoint. Let us quantify this information in terms of the total number of pixels needed from images in the DLF. Assuming we are doing no temporal filtering, we can restrict the information needed to one frame in the DLF. Now consider one pixel in the image being rendered - this pixel is equivalent to a ray between the viewpoint location and a sample point on scene geometry. How many pixels from the current frame in the DLF are required to reconstruct this pixel? If we are doing quadralinear filtering, then 16 pixels are needed: these pixels come from the images taken by the four cameras selected as being closest to the desired ray, and within each image four pixels are needed to do bilinear filtering on the image plane. Thus we can theoretically bound the number of pixels needed from the DLF to be linearly dependent on the size in pixels of the image being rendered. Notice that there is no dependency on either the number of cameras or the resolution of the images taken by the cameras.

Now the question is whether this bound is practical: can it be achieved within our OpenGL rendering framework? The short answer is yes, it can. Consider a particular triangle from the triangulation of the sample points. This triangle will be drawn once for each camera that has positive weight at any of the vertices of the triangle. Call this set of cameras Ct for triangle t. Assuming at most 4 cameras have positive weight at any one sample point, |Ct| will be at most 12 and, assuming sufficiently fine sampling, will average close to 4. This means that the sum over all triangles t of the area of triangle t (in pixels, when projected onto the plane of the image being rendered) multiplied by |Ct| will be approximately 4 times the size of the image being rendered.

How does this translate into pixels requested from the DLF? Well, each triangle in object space maps to a triangle in the image space of a particular camera c. Denote Tc to be the set of triangles such that triangle t is in Tc if and only if camera c is in Ct. Assuming the spacing of cameras in the grid is not too large and the scene geometry is relatively flat (ours is just a focal plane), the area in camera image space covered by all triangles in Tc will be a roughly contiguous, rectangular region. Since no image information is needed outside of this region, we can load only this region as the texture used by OpenGL when rendering the triangles in Tc. This helps reduce the amount of image information needed to render the scene in cases where only a small part of the image from each camera is needed.

However, there are still cases where the entire image is needed from each camera. Imagine moving the focal plane very close to the grid of cameras, so that there is no overlap between each camera's view of the focal plane. Now move the viewpoint way back so it can see the entire field of view of the cameras. The view rendered will contain scaled down versions of each camera's image laid out in a grid. In this case, the region required from each image covers the full image, but needs only be of low resolution. Thus the renderer needs from camera c's image only a particular region at a resolution determined by the ratio of the area covered by the triangles in Tc in view space to the area covered by the triangles in the camera space. There will be some overhead due to restrictions on what regions and resolutions can be requested, but this overhead can be bounded by a constant multiplied by the size of the region being requested relative to the resolution requested. Ignoring this overhead, the area in pixels of the region and resolution requested from a camera c's image will be the same as the area in pixels covered by the triangles in Tc in view space. Thus the total area in pixels requested by the renderer from all camera images will be approximately four times the size in pixels of the image being rendered for the average case, and will be bounded by a constant (greater than 4) times the size in pixels of the image being rendered in the worst case.

In other words, we can achieve the theoretical bound described above in a practical manner, at least from the point of view of our OpenGL renderer.

7 Region-Based Data Selection from Image Files

The implementation of region-based data selection in a raw RGB file format is trivially achieved. Less apparent is how to extract a region from a compressed J2K file. In order to describe how to decode only the desired region in a J2K file, we will first give a brief overview of how data is processed in J2K.

7.1 J2K Compression

There are several encoding operations involved in J2K compression, including transforming between YUV and RGB color spaces, discrete wavelet transformations (DWT), and entropy encoding (T1). The main bottleneck in decompressing a J2K image occurs in DWT, though T1 also takes a significant amount of time. Since the decompression pipeline sends the data first through T1 decoding and then through DWT, we are able to repackage the codeblocks that were decoded in T1 to only contain the region we wanted and pass that down the J2K pipeline. The interesting dilemma was understanding how to filter for the data we wanted and repackage it for further decompression.

7.2 J2K Data Ordering

Repackaging the data requires a careful understanding of how J2K decomposes an image into several structural entities. This decomposition allows for easy and relatively fast regional access and resolution specification as compared to MPEG1-2 or JPEG. In Figure 2, we can see what the image looks like after a J2K module performs entropy decoding on the wavelet coefficients. Each sub-band represents some high-low frequency permutation of the image at a specific resolution. Figure 3 is a visual representation of what an image from one of our data sets looks like when we only decode lowest resolution and leave other resolutions in their discrete wavelet transforms form, still encoded.

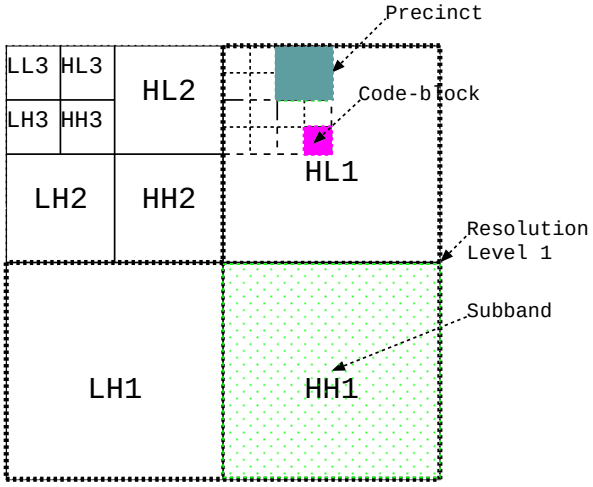

Figure 2
Each block represents 1 of 4 possible sub-bands. The number affixed to each sub-band label identifies that sub-bands resolution level.

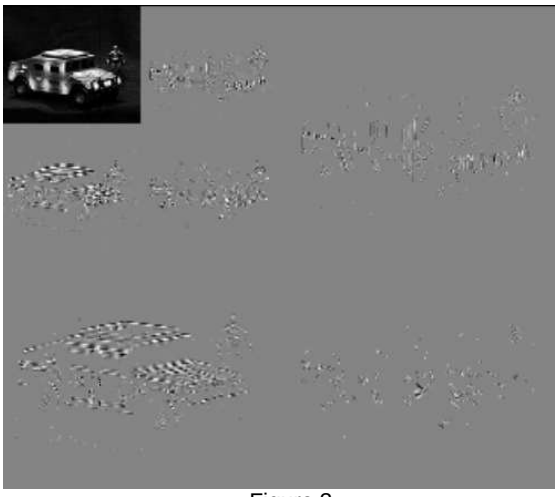

Figure 3
Sub-bands in the DWT decomposition

7.3 J2K Repackaged

For J2K to seamlessly transition from T1 to DWT in the decompression pipeline, DWT needs to receive a data package with all the sub-bands intact and all resolution levels less than and equal to the resolution level we desired. This means that when we want to retrieve a region of an image at full resolution, we would locate that region in each sub-band at each resolution level and repackage it to fit the same format that you see in Figure 2, but scaled proportionally to the size of the region requested to the original image size.

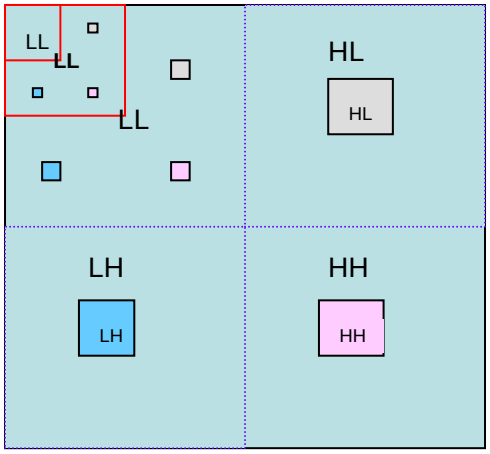

Figure 4
Repackaging a region out of each sub-band

8 Results & Discussion

We acquired two data sets, a static light field, which we used to test our nascent light field renderer with, and the dynamic light field that we described in section 3 of this paper.

Our system can process 640x480 30fps video from over a hundred cameras. It allows for the user to navigate the dynamic light field in real-time. In particular, the user can play, pause, rewind, pan, rotate, zoom, and change the focal

depth of the view being rendered. In addition, the user is given control over the resolution of the view being rendered; dropping the resolution generally improves the performance of the system.

8.1 Frame Rate with respect to Resolution

We tested our light field renderer on a 1.5GHz Pentium M processor with 768MB of RAM, an nVidia GeForce FX Go 5200 graphics card, and a 5400rpm disk running windows XP.

Resolution Rendered	Frames Per Sec (J2K, 100:1 compression)	Frames Per Sec (Uncompressed)
640x480	0.8	4.0
320x240	1.8	1.3
160x120	3.0	1.8
80x60	6.0	4.0
40x30	10.0	6.0

Table 1
Rendering speeds at various resolutions while playing a DLF video sequence

In Table 1 we see the rendering speeds our renderer obtains while playing a DLF video sequence forward in time using both compressed and uncompressed data. Notice that at the highest resolution, loading raw images directly off disk is about 5 times faster than loading and decoding compressed J2K images. This ratio is roughly determined by the ratio of processor speed to the bandwidth between disk and memory: in a system with a faster processor, J2K decoding would improve; in a system with more bandwidth off disk, the uncompressed case would improve.

For J2K compression, as we decrease the resolution of the image being rendered, frame rates steadily improve by a factor of 2 for each drop in resolution when using J2K compression. Ideally the frame rate should go up by a factor of 4 since the amount of image information being decoded goes down by a factor of 4 when the resolution is halved. We attribute the disparity between the ideal and what we see here to overhead in J2K decompression and in disk accesses.

For no compression, there is an initial drop in the frame rate when the resolution is lowered. This is likely due to the fact that we do not store multiple versions of each image at different resolutions, instead selectively reading exactly the pixels we need out of each image (for instance, every other pixel in the 320x240 case). This is highly inefficient, hence the drop in frame rate. If we did store multiple resolutions of each image we could expect frame rates to improve by approximately a factor of 4 for each drop in resolution.

8.2 Effects of Caching

When the user pauses the renderer, consecutive rendered views begin to reuse image information from the previous rendered view. In the uncompressed case this means that the same files are re-read each time. Since operating systems often cache recently used files, these duplicate reads will not necessarily require going out to disk, resulting in a large increase in frame rate. Thus viewing a static light field (or, equivalently, one frame in a DLF) is quite smooth.

8.3 J2K Compression

In the test results shown in Table 1 we compressed our original images by a factor of 100 using J2K compression. While artifacts due to compression at this level are noticeable if one compares the compressed image to the original, they are not objectionable at all, especially when viewing a video sequence.

9 Conclusions & Future Work

Although, we are only seeing 2 fps with region-based J2K decompression, we are optimistic that there is many more optimizations that can be done with J2K and our current system.

There are optimizations that can be done with the compression, specifically switching to a 4D discrete wavelet transformation, although it is still unclear if that allows for fast random access of data.

In terms of system optimizations we would like to explore asynchronous disk I/O to allow image loading to occur in parallel with image decoding. Another area we could explore is pre-fetching images that are likely to be used to render a view in the near future.

We also see areas where we can reduce the time taken for J2K decompression but have not had the chance to test it. Specifically, the region we select from J2K must start at a power of 2 byte entry point in the image.

Finally, we could consider how to leverage dynamic resolution (that is trivially attained with J2K decompression) in a scene so that we can drop the resolution in uninteresting parts of the image being rendered.

[1] [Levoy96] Levoy, M., Hanrahan, P. “Light Field Rendering”, Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 96), pp31-42.

[2] [Buehler01] Buehler, C. Bosse, M. McMillan, L. Gortler, S. Cohen, M. “Unstructured Lumigraph Rendering”, Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 01).

[3] [Sloan97] Sloan, P., Cohen, M., Gortler, S. “Time-Critical Lumigraph Rendering.” 1997 Symposium on Interactive 3D Graphics, pp. 17-24.

[4] [Yang02] Yang, J., Everett, M., Buehler, C., McMillan, L. “A Real-Time Distributed Light Field Renderer”, Thireteen Eurographic Workshop on Rendering.

[5] [Naemura02] Naemura, T., Tago, J., Harashima, H. “Real-time video-based modeling and rendering of 3d scenes.” IEEE Computer Graphics and Applications 2002, pp. 66-73.

[6] [Goldlucke] “Hardware-Accelerated Dynamic Light Field Rendering”

[7] [Zhang00] Zhang C., Li J. “Compression of Lumigraph with Multiple Reference Frame Prediction and Just-in-Time Rendering.” IEEE Proceedings of the Data Compression Conference 2000, pp. 254-263.

[8] [Chang] Chang, C., Zhu, X., Ramanathan, P., Girod, B., “Inter-View Wavelet Compression of Light Fields with Disparity-Compensated Lifting.” *Invited Paper.*

[9] [Wilburn02] Wilburn, B., Smulski, M., Lee, H.K., and Horowitz, M. 2002. “The Light Field Video Camera.” Media Processors 2002, vol. 4674 of SPIE.

[10] [Anon04] “3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes.”