Interactive Rendering of Surface Light Fields

Daniel Azuma
University of Washington
20 May 1999

Abstract

Many interactive 3D applications involve capturing, representing and reproducing the appearance of real objects. The surface light field represents the appearance of a solid object under static lighting by sampling the radiance along rays parametrized by surface position and direction. An implementation can represent this data as high-resolution surface geometry, coupled with multiple texture maps, each associated with a direction. The representation can be rendered by a multipass algorithm utilizing common graphics hardware. Through careful optimization, it is possible to achieve interactive frame rates using existing implementations of OpenGL. Surface light fields tend to produce higher resolution renderings than similar methods such as the lumigraph, and do not exhibit occlusion ghosts and other artifacts associated with the lumigraph at low sampling rates. Construction and optimization of surface light field data present a number of challenges.

1. Introduction

Many interactive 3D applications involve capturing and reproducing the appearance of real objects. Such applications might include virtual reality, games or visualization. Real objects, however, exhibit complex surface properties such as subsurface scattering and phosphorescence. A representation must be general enough to model such properties. In addition, it should provide a rendering algorithm capable of reproducing those properties from the representation at high resolutions.

Surface light field rendering is a novel image-based algorithm for rendering acquired solid objects under static lighting. The algorithm resolves high-resolution surface detail and reproduces surface radiance resulting from complex specular properties. This work presents and analyzes the surface light field rendering algorithm, investigating performance issues on typical graphics hardware and comparing with similar methods.

1.1 Related work

Ubiquitous interactive 3D rendering techniques generally involve rasterizing a model as a set of polygons, applying a simple shading model to simulate surface reflectance properties, and using texture maps to add detail. Most graphics-oriented workstations include hardware capable of accelerating these computations. Unfortunately, such techniques are often not able to reproduce the complex characteristics of real objects, often resulting in the "plastic" appearance associated with Phong shading. Sato, *et al*[10] describe a method by which the specular properties of a surface are estimated from a set of photographs, according to a specified reflection model. Walter, *et al*[13] describe a method for using a simple lighting model to approximate complex specular lobes by fabricating a set of virtual lights. Both these methods are limited by the reflection model implemented, particularly in a hardware-assisted rendering system.

Some more recent work has focused on techniques known collectively as *image-based rendering*. Image-based rendering algorithms involve manipulating acquired or precomputed images of a scene to reconstruct different views. Such techniques often do not require the considerable computational resources needed by traditional 3D rendering; however, they tend to incur high storage costs for the image data.

Many image-based rendering techniques are based on the five-dimensional plenoptic function[1]— radiance as a function of 3D position and direction— which completely describes the appearance of a scene from any viewpoint. Levoy and Hanrahan[7] note that this representation can be reduced to four dimensions in empty space by observing that radiance remains constant along a nonoccluded ray. They parametrize the 4D space of rays according to grid points on two parallel planes, and describe an interactive algorithm for constructing images for arbitrary viewpoints. This light field rendering algorithm does not require any geometric information for the scene. Gortler, *et al*[4] describe an enhancement which makes use of coarse geometric information to increase the depth of field of rendered images. Surface light fields are similar to those methods, but involve a different parametrization of rays, attached to a surface manifold. Miller, *et al*[8] explore this parametrization, focusing on its implications for compression. They present an interactive rendering algorithm, but it is able to model surface detail only at the resolution of the surface tessellation.

The surface light field rendering algorithm is also similar to view-dependent texture mapping, in which multiple textures are weighted and composited to render novel views. Debevec, *et al*[3] and Pulli[9] use this technique to generate new images using previously acquired images. Previous work on view-dependent texture mapping has not systematically modeled surface properties such as specularity. Furthermore, the texture weighting based on direction is done at object resolution, which may not accurately reproduce small-scale behavior. The surface light field fully models arbitrary surface radiance and

supports hardware-assistedper-pixel computation of direction-based texture weights.

1.2 A road map

The remainder of this paper is organized as follows. First, section 2 reviews the representation and the basic rendering algorithm, which were developed jointly with Daniel Wood. Section 3 describes an interactive OpenGL-based rendering implementation of the algorithm, presents a series of performance analyses, and describes optimizations done to improve the performance. Section 4 compares surface light field rendering with a closely related method, the lumigraph[4], and discusses the quality and accuracy of reproductions as a function of data set size. Section 5 briefly outlines the context of the algorithm, including data collection and construction of a surface light field. Section 6 closes with conclusions and discussion.

2. The algorithm

The surface light field is a very general representation capable of capturing arbitrary surface radiance behavior. The representation also suggests a rendering algorithm that may be run at interactive speeds using existing graphics hardware. This section details the representation and the basic rendering algorithm. A useful implementation of this algorithm requires a number of optimizations such as those described in section 3.

2.1 The surface light field representation

The five-dimensional plenoptic function completely describes the appearance of a scene from any viewpoint. This function in its most basic form parameterizes by point in space and direction. In the following function, x, y and z represent a parametrization of 3-space, and θ and ϕ represent a direction parametrization.

$$L(x, y, z, \theta, \phi) \rightarrow rgb$$

Given a surface surrounded by empty space, the plenoptic function can be reduced to four dimensions parametrized by surface position and direction (figure 2.1). Here, s and t define a parameterization of a surface.

$$L(s, t, \theta, \phi) \rightarrow rgb$$

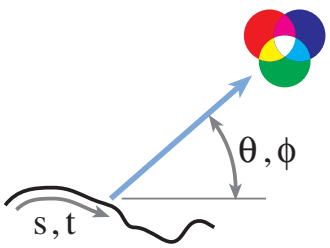

figure 2.1: The surface light field is parametrized by surface position and direction.

Equivalently, this can be represented as a function mapping direction, to functions mapping surface position to radiance (i.e. texture maps).

$$F(\theta, \phi) \rightarrow G(s, t) \rightarrow rgb$$

Consider a rendering algorithm for this representation involving raytracing. Given a pixel x in the image plane, cast a ray through x into the scene and find the intersection $\mathbf{p}$ with the surface. Let $-\mathbf{r} = -(\theta, \phi)$ denote the direction of the ray. Then the color of the pixel is determined by the radiance leaving $\mathbf{p}$ in direction $\mathbf{r}$. Thus, choose the texture map given by evaluating $F(\mathbf{r})$, find the texture coordinates $\mathbf{s} = -(s, t)$ of $\mathbf{p}$, and read the color.

Formally, consider the camera projection function $-\mathbf{r} = C(x)$, mapping pixels in the image plane to ray cast directions. Furthermore, consider the surface parametrization function $\mathbf{s} = T(\mathbf{c}, -\mathbf{r})$, mapping the camera center of projection $\mathbf{c}$ and ray cast direction $-\mathbf{r}$ to texture coordinates on the surface geometry. We can then represent the rendering algorithm as follows:

$$rgb(x) = \left[F(-C(x))\right]\left(T(\mathbf{c}, C(x))\right)$$

In this paper, the notation $[A](b)$ means to evaluate function A and apply the result, also a function, to b.

2.2 The discrete representation

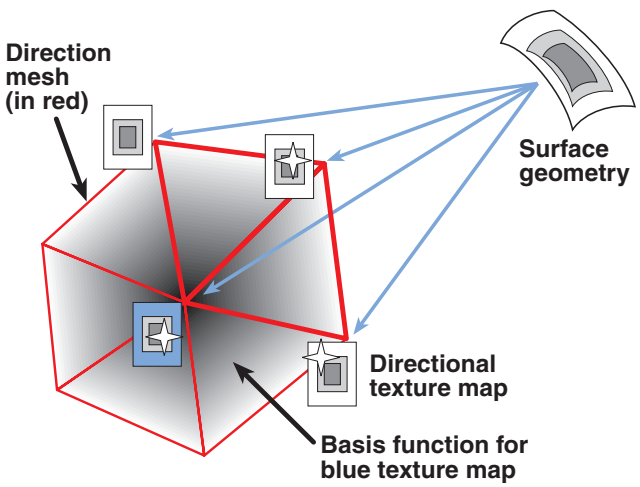

figure 2.2: Texture maps are associated with directions. Directions are arranged in a direction mesh, and each has a corresponding basis function.

In a real data set, F and G must be represented as a discrete set of samples, each associated with a basis function. Discretization of texture coordinates in G is nearly always done by choosing grid points and using bilinear interpolation as a reconstruction kernel. This reconstruction is common in texture implementations, and is therefore not discussed further. One natural and very general way to discretize direction space involves choosing an arbitrary set of direction samples, then triangulating them on the unit

sphere using a method such as Delaunay triangulation. We call this tessellation of the sphere the *direction mesh*. This scheme also suggests a natural set of piecewise linear basis functions: center a hat function at each sample vertex, covering its neighborhood in the mesh. This representation is illustrated in figure 2.2.

Let $\mathbf{r}_i$ denote the *i*th sample direction, let F_i denote the texture map associated with that direction, and let B_i denote the corresponding basis function. Then F may be reconstructed using the following formula:

$$F(\mathbf{r}) = \sum_i F_i B_i(\mathbf{r})$$

We may now modify our raytracing algorithm to work on a real discretized data set.

$$rgb(x) = \left[\sum_i F_i B_i(-C(x))\right]\left(T(\mathbf{c}, C(x))\right)$$
$$= \sum_i B_i(-C(x))\left[F_i\right]\left(T(\mathbf{c}, C(x))\right)$$

In words, given our pixel *x*, we cast a ray and find the texture coordinates of the intersection **p**. We then iterate over all textures, look up the color, scale by $B_i(\mathbf{r})$, and accumulate.

2.3 A hardware-accelerated algorithm

Rendering algorithms based on ray tracing usually cannot be implemented at interactive speeds because of the necessary per-pixel computations. However, modern workstations typically include hardware designed to accelerate certain rendering processes such as rasterizing polygons, texture mapping and alpha compositing. Using OpenGL, a common graphics API, the surface light field rendering algorithm can be modified to take advantage of acceleration provided by existing graphics hardware.

To achieve high performance, we would like to utilize the hardware to perform the per-pixel texture weighting. This can be done by constructing B_i', where $B_i'(\mathbf{r}) = B_i(-\mathbf{r})$. Intuitively, since B_i is a hat function supported by a triangle fan B_i' is the same hat function pointing in the opposite direction. Like B_i, its support consists of triangles from a tessellation of the unit sphere, which we call the *reversed direction mesh*. Our algorithm now looks like this:

$$rgb(x) = \sum_i B_i'(C(x))\left[F_i\right]\left(T(\mathbf{c}, C(x))\right)$$

Consider a single texture map sample F_i associated with basis function B_i. Note that the weight $B_i'(C(x))$ is simply a hat function parametrized according to the camera projection $C(x)$. Therefore, we can deposit this weight at each pixel by simply rendering B_i' directly into the alpha channel. Center the reversed direction mesh at the camera, which matches directions $-\mathbf{r}_i$ in the reversed basis function with ray directions $C(x)$, and render the triangles supporting the basis function using Gouraud shading. Similarly, $\left[F_i\right]\left(T(\mathbf{c}, C(x))\right)$ is a straightforward rendering of the geometry with texture F_i. This suggests a two-stage algorithm: first, render the basis into the alpha channel, then render the textured geometry, weighted by alpha, and accumulate. An OpenGL implementation can perform this weighted accumulation using a blend function. Figure 2.3 illustrates this process.

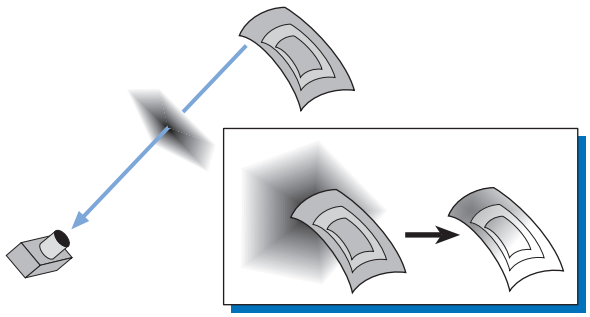

figure 2.3: Textured geometry weighted by hat basis

A parametrization of a complete surface not homeomorphic to a disc typically requires a number of parameter domains, or *texture regions*. Iterating over all texture regions, we have the final algorithm:

Construct B_i' for all *i*.
foreach texture region
  foreach direction *i*
    Render B_i' into alpha channel
    Render surface with texture map F_i
      weighted by alpha, and accumulate
  end foreach
end foreach

Even though this algorithm performs the correct weighting computation per pixel, it is potentially fast because all per-pixel computations may be done in hardware. Note that Gouraud-shaded rendering ("smooth" shading) under OpenGL is not perspective-correct, so this does not give precisely the correct weight at every pixel. The remaining slight error could be corrected by using a 1D texture map instead of Gouraud shading; however, we have found that Gouraud shading is close enough to produce convincing results.

2.4 Summary

A surface light field is represented by a parametrized geometric surface coupled with a direction mesh. Associated with each vertex in the direction mesh is a texture map describing radiance leaving the surface in that direction. This representation can be rendered quickly using a multipass rendering algorithm. Each pass includes two stages: a rendering of the basis function into the alpha channel, followed by a rendering of the textured surface. Both rendering stages may take advantage of existing graphics hardware.

3. Performance optimizations for interactive rendering

This section analyzes the performance of the algorithm described in section 2, and details some of the optimizations required to achieve interactive frame rates.

To perform this analysis, we constructed a surface light field of a small fish statuette with a detailed surface texture and interesting specular properties. Figure 5.2 shows a few photographs of the statuette. The construction process is outlined in section 5. The resulting surface light field includes a surface triangle mesh decomposed into 199 parametrized texture regions. Associated with each texture region is a direction mesh of 66 directions approximately uniformly distributed over the sphere. In this data set, the direction mesh is the same for every texture region.

I implemented the rendering algorithm in C++ using OpenGL, with two user interfaces. The first is an interactive viewer; a screen shot appears in figure 6. The second version accepts a script— specifying viewpoints, camera settings and other commands— and was useful for establishing benchmarks. I tested the implementation on an SGI Indigo2 with a 250MHz R4400 processor and a Maximum Impact graphics board. A few additional tests were done on a 250MHz R10000 with an Infinite Reality engine. Some of the results obtained and optimizations described may be partially dependent on the characteristics of the specific OpenGL implementation.

In this section, I describe the three most important optimizations implemented. I implemented the first— culling rendering passes— in the initial versions of our surface light field viewer. The second and third— accelerating basis rendering and custom texture caching— I implemented later in response to performance data gathered during my profiling analysis.

3.1 Culling rendering passes

The first issue with the algorithm as described in section 2 is the number of rendering passes, which is mn, where m is the number of regions and n is the number of directions per region. For our test data set, this results in more than 13,000 passes, which is clearly too large for an interactive algorithm. Fortunately, it turns out that most of these passes do not contribute to the final image, and it is possible to perform a fast analysis to cull them.

Given a camera pose and a piece of scene geometry, there will nearly always exist a nonempty set of directions S such that for any direction $\mathbf{r} \in S$, no radiance leaving the region geometry in direction $\mathbf{r}$ reaches the camera. Any direction i in the direction mesh that contributes solely to directions in S (that is, the support for its basis function B_i is completely contained in S) does not need to be rendered. In such a case, the projection of B_i' onto the image plane does not intersect the projection of the geometry onto the image plane, resulting in the algorithm weighting all rendered pixels with zero alpha value. Figure 3.1 illustrates this case. To improve performance, it would be useful to detect this condition and skip the rendering pass altogether.

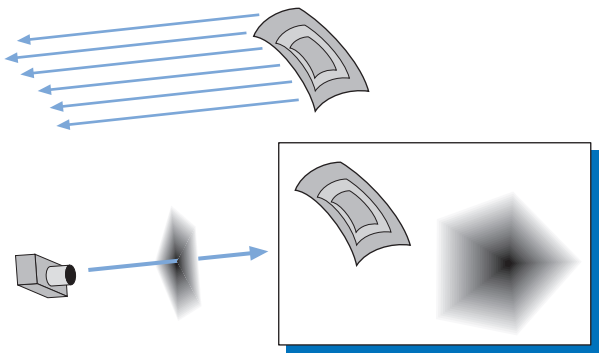

figure 3.1: If no rays in this direction reach the camera, the basis function does not intersect the region.

The rendering pass corresponding to a direction i contributes to the final image if and only if the generalized cone of directions spanned by B_i' intersects the generalized cone of directions spanned by the geometry, as seen from the camera. A plausible algorithm for detecting this condition might test every direction i and perform the polyhedron intersection computations, but this may be expensive for data sets with many direction samples.

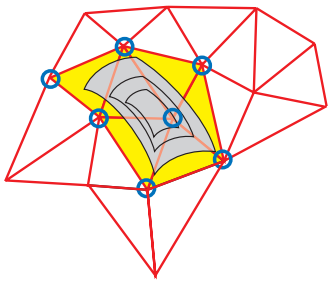

figure 3.2: Reversed direction mesh and region geometry seen from the camera. Only vertices incident on traversed triangles are contributing directions.

A faster algorithm involves traversing the graph defined by the dual of the reversed direction mesh. First, consider any ray which intersects the geometry, and find the triangle in the reversed direction mesh hit by that ray. Call this the *seed triangle*. Next, traverse the graph, crossing an edge into an adjacent triangle if its extrusion into space— a planar fan— intersects the geometry. Given that the piece of geometry in question is contiguous, the contributing directions will be exactly the set of vertices incident on the triangles visited during this traversal (figure 3.2). This process is faster than testing every basis function support independently, because the geometric computations are simpler, and because a given piece of geometry will usually intersect only a few triangles, resulting in a short traversal. My implementation further accelerates this analysis by constructing a conservative bounding volume around the region geometry, thus simplifying the intersection tests.

Finding the seed triangle itself can be implemented in log time using a hierarchical search. However, in an interactive viewer, two subsequent views of the geometry are likely to be from similar camera poses; thus, they will likely have the same or neighboring seed triangles. Therefore, my implementation finds a seed triangle by beginning at the previous seed and walking along the reversed direction mesh. In practice, this algorithm gives better performance than a full hierarchical search.

Another analysis can be performed to identify entire regions that are completely back facing. All textures associated with such regions can be ignored completely. A region that is relatively flat will have a large, contiguous cone of directions, called the *blackout cone*, from which the geometry is completely back facing. More formally, given the set of normals associated with the region geometry, the blackout cone is the set of directions whose dot products with every normal are negative. If all directions from the geometry to the camera lie within the blackout cone, the region is completely back facing and can be ignored. My viewer computes a conservative, circular approximation of the blackout cone in a preprocessing step. It is then straightforward to decide whether to cull an entire region, by reversing the blackout cone and testing whether the region geometry falls entirely within the reversed cone. Again, the implementation uses a conservative bounding volume around the region geometry to simplify this test.

Together, these two analyses reduce the number of rendering passes per frame from more than 13,000 to an average of 600 for typical views of the fish data set.

3.2 Accelerating basis rendering

An analysis of the runtime performance of the algorithm at this point revealed that the basis function rendering stage was severely fill-limited. The support of the basis function in a typical rendering pass covered much of the frame buffer. In a 500x500 window, this resulted in more than 100,000 fill pixels per pass, for more than 500 passes, totaling 50-100 million fill pixels per frame.

To reduce the fill requirements, two optimizations were implemented. First, although the support for a basis function is a complete triangle fan, some of the individual triangles may not intersect the geometry. Those triangles need not be rasterized. This optimization reduces the number of basis triangles rasterized by about 3/4 on our data set. It actually does not require any additional analysis because the direction-culling analysis in section 3.1 already computes the set of important direction mesh triangles. Second, I clip triangles by setting the OpenGL view frustum to tightly contain a bounding volume around the region geometry. These optimizations together effectively doubled the overall frame rate (figure 3.3).

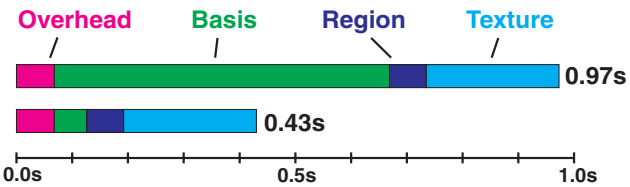

figure 3.3: Rendering profile for a single frame, before and after basis optimizations. “Overhead includes OpenGL state setup and culling of rendering passes. “Basis” includes rendering of basis functions into alpha. “Region” includes rendering region geometry. “Texture” includes loading texture images. Times for the various stages are approximately additive, but not completely due to some pipeline parallelism.

3.3 Custom texture cache management

As shown in figure 3.3, the next bottleneck is texture bandwidth. On average, the algorithm sends in excess of 3 million texels of texture data per frame through the OpenGL pipeline[1], the speed of which is limited by the throughput of the bus. Most graphics hardware includes some amount of high-speed texture memory used to cache texture data. An application typically wraps commonly-used textures in texture objects, which the OpenGL implementation swaps in and out of cache. Unfortunately, this mechanism cannot be used as is by the surface light field rendering algorithm because the number of textures (>13,000) exceeds the number of texture objects that can be efficiently managed by most OpenGL implementations. In addition, the straightforward LRU cache management policy used in many OpenGL implementations may fail to work if the textures for an entire frame do not fit into texture memory. The behavior of the algorithm is such that textures are loaded only once per frame, in a well-defined order. If all the texture data cannot fit into texture memory at once, textures from the beginning of the algorithm will be overwritten before they can be used in the next frame. These textures will be reloaded into the cache, overwriting further textures, and the effect cascades through the entire algorithm, resulting in a zero cache hit ratio.

In order to effectively utilize texture memory, I implemented a custom texture cache manager. First, the manager allocates enough large texture objects to fill available texture memory. It uses this set of objects solely as a means of addressing the texture memory, subdividing them into equal-sized “slots” large enough to hold the largest individual texture, and uploading individual textures to the cache as needed using glTexSubImage2D calls. The custom texture manager uses a modified LRU cache policy in which, in a given frame, textures are cached only until texture memory has nearly overturned. This prevents later textures from overwriting earlier textures, reducing the likelihood of a cascading cache miss during the next frame.

[1] It turns out that this value is rather inflated because the OpenGL 1.0 implementation of glTexSubImage2DEXT required texture sizes to be powers of 2. OpenGL 1.1 implementations do not have this restriction, so their numbers would probably be smaller.

Using the texture memory on our Indigo2, the custom texture cache is large enough to hold approximately 10% of the texture data necessary for one frame. Assuming the best possible cache hit ratio, a speedup of at most 10% could be expected in the texture loading phase; and the observed speedup was correspondingly fairly small (figure 3.4). Preliminary tests on an Infinite Reality engine with a much larger texture memory show more dramatic speedups.

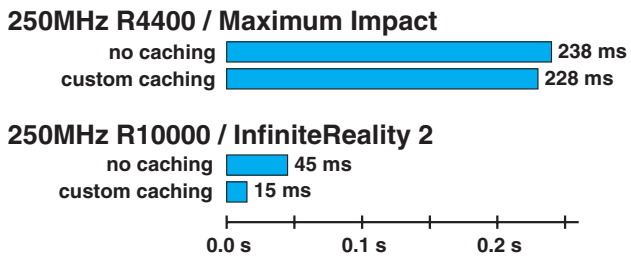

figure 3.4: Effect of custom texture cache management

3.3 Final results

A naive implementation of the algorithm presented in section 2 was never done, but I estimate it may require on the order of several minutes to render each frame. An implementation including an analysis to cull rendering passes demonstrated a frame rate of about 1.0fps for our standard data set on the Indigo2. Informal tests on an Infinite Reality engine suggested performance slightly better than 2.0fps. Through subsequent optimizations, including the second and third ones described above, I was able to improve those frame rates to 2.5fps on the Indigo2, and 5.7fps on the Infinite Reality engine. Additional analysis indicates that it should still be possible to further improve these numbers, particularly on the IR.

4. Surface light fields vs. the lumigraph

The lumigraph[4] is a similar image-based method for representing the appearance of an acquired real scene. Whereas the surface light field parametrizes rays according to surface position and direction, the lumigraph's parametrization is geometry-independent. Instead, two parallel planes are defined near the surface, and rays are parametrized by their points of intersection with the two planes: (s,t) and (u,v). The lumigraph representation also supports an interactive rendering algorithm able to utilize current graphics hardware.

This section compares surface light field rendering with the lumigraph, focusing on tradeoffs between image quality, accuracy and data size. I will argue that the parametrization employed by the surface light field is the natural parametrization for surface appearance, and that the lumigraph's parametrization results in poorer image quality for a given-sized data set. Furthermore, I will demonstrate that the clean division between position and direction in the surface light field allows those dimensions to be simplified independently, resulting in well-understood degradation, whereas the lumigraph exhibits several second-order effects that further limit simplification opportunities. Although it is an important issue, I do not discuss rendering performance because it is not an inherent property of the method, but depends also on details of the implementation and hardware characteristics.

4.1. Comparison overview

The surface light field used for comparisons was the same data set used for performance analysis, including 199 regions, each with 66 uniformly-distributed directions, and texture sizes chosen to provide roughly even coverage of the surface. The complete data set includes 33,235,488 texels (rays). The storage required by the surface geometry is negligible compared to texture data storage.

I implemented a simple lumigraph resampler and constructed several lumigraphs from the same input data used for the surface light field. Ideally, the lumigraph should be the same size, the same resolution "quota," as the surface light field in order to compare results generated by the two methods. However, the surface light field data set covers all faces of an object, whereas a single light field "slab" represented by a lumigraph represents only one of the six cardinal directions. In addition, due to eccentricities of the current surface light field implementation (specifically, texture regions currently overlap), it turns out that the surface light field includes a 3x redundancy. Because a smaller redundancy may end up as part of the surface light field representation in the future, I chose a conservative estimate of 2x to compensate for this factor. Therefore, the final lumigraph quota was determined by reducing by a factor of 12 to 2,769,624 texels.

I generated test lumigraphs using a ray tracing algorithm as follows. Given *st* and *uv* planes, and (s,t,u,v) coordinates, cast a ray and find the intersection of the ray with the object geometry. Given this point of intersection, locate the camera that sees the point of intersection from a direction nearest to the direction of the ray. Find the color from the corresponding pixel in the camera image. Note that this resampling does not attempt any interpolation. Better methods can certainly be implemented, and indeed, Gortler, in the original lumigraph paper, describes a novel algorithm for reconstructing multidimensional functions from scattered data[4]. I chose my resampling method because the cameras densely populate the sphere of directions, and because it matches the currently implemented method for resampling surface light fields (see section 5).

4.2. A natural parametrization

The surface light field parametrizes the space of rays by surface position and direction. This could be considered the natural parametrization of the light field leaving a solid surface because it directly corresponds to the physical dimensions of the function: surface texture and radiance function. The parametrizations along these dimensions can

be tuned according to the behavior of the surface. In particular, for most surfaces, the interesting directional behavior as defined by the BRDF tends to be distributed more or less evenly across the space of directions, suggesting that a uniform distribution of rays in direction space may be most preferable. If, however, a piece of surface has the property that specular lobes tend to point in the same direction (perhaps because the surface is flat), it may make more sense to adjust the parametrization to provide better resolution in that direction. The arbitrary direction mesh provides support for both cases.

The Lumigraph parametrizes the space of rays using pairs of grid points on two parallel planes. This parametrization serves three purposes. First, it leads to a reasonably efficient interactive rendering algorithm that can be implemented with the assistance of existing graphics hardware. Second, it does not require knowledge of the geometry, although the system can use such information, if available, to improve the quality of renderings. Surface light fields inherently require geometry information. Third, the lumigraph parametrization provides for some simplification and compression opportunities due to the lower frequency often exhibited in the *st* dimensions. Surface light fields provide arguably better opportunities, as described in section 4.3.

4.2.1. Uniformity of parametrization

The fundamental weakness of the parametrization scheme used by the lumigraph, as well as the similar light field rendering work by Levoy, *et al*, is that it is neither physically-based nor well-suited to be adjusted according to surface properties. One property that exposes this weakness is direction nonuniformity. If the natural parametrization of the light field describing the appearance of a solid object is by surface position and direction, the two-plane parametrization is non-ideal because it does not uniformly map to direction space.

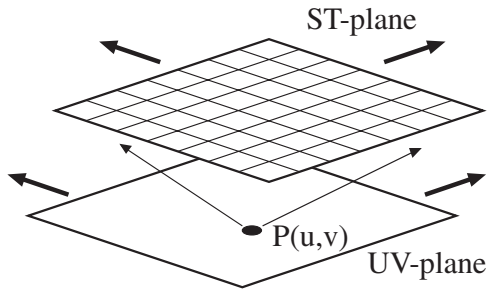

figure 4.1: If P is on the surface, the st plane provides a parametrization of direction.

For example, consider a surface point lying at a point (u, v) on the *uv* plane. Then consider a regularly-sampled *st* plane as shown in figure 4.1. This sampling corresponds to the direction sampling that is denser at grazing angles than at perpendicular angles (figure 4.2). Thus, if resolving the directional appearance of an object requires a certain minimum angular resolution in direction space, the *st* and *uv* parametrizations must be chosen such that they provide at least this resolution in the sparsest region of directions, near the perpendicular. Because of the nonuniformity in the parametrization, this leads to an excessive directional resolution at grazing angles. Attempting to sample the *st* grid non-uniformly to compensate may improve matters, but will not eliminate the problem because each individual (*u,v*) will want a different optimal *st* sampling.

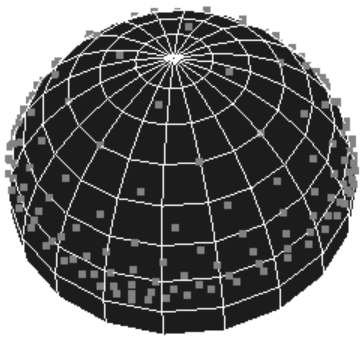

figure 4.2: A regular grid sampling of the st plane maps to a nonuniform sampling of direction space.

Related is the problem of extending the two-plane parametrization to cover all directions and all faces of an object. Both Levoy and Gortler propose providing a pair of planes to cover each cardinal direction, six in all. This causes additional nonuniformities, especially near and across the boundaries. In addition, a large number of rays may not intersect the geometry, thus being wasted. Again, surface light fields do not suffer from these issues.

Because of these parametrization nonuniformities, lumigraphs require a greater texel quota to achieve the same effective resolution as surface light fields. Alternately, a lumigraph utilizing the same number of texels as a surface light field will not achieve the same reproduction quality. Figure 4.6 demonstrates this result. Image A shows a closeup of a surface light field rendering of the fish model. Image B shows a lumigraph of matching texel quota, with the *st* resolution chosen to match the surface light field's direction resolution. *uv* resolution is 238x238; *st* is 7x7. (Note that this results in 49 direction samples for somewhat less than a hemisphere, compared with the surface light field's 66 direction samples for the entire sphere.) This does not leave sufficient *uv* resolution to resolve the detail shown by the surface light field. Image C shows a 416x416 *uv,* 4x4 *st* lumigraph. *uv* detail is closer to that demonstrated by the surface light field, but *st* is now so sparse that highlights are visibly distorted.

4.2.2. Viewing inside the convex hull

Indeed, even though the lumigraph requires a higher texel quota to achieve the same effective resolution, it still does not represent as much data as a surface light field. Specifically, it places a constraint on camera positions for which the function gives well-defined results. This constraint is illustrated in figure 4.3. If the surface is not convex, then there will exist rays that exit the surface from

two different points. The lumigraph must choose one surface from which to represent the radiance; typically, this is the closer surface (A). Unfortunately, this implies that a camera positioned between the two surfaces will not get correct data from the occluded surface (B).

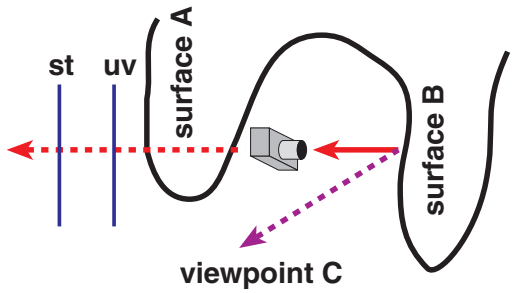

figure 4.3: Attempting to view inside convex hull.

A surface light field, on the other hand, can correctly represent the light leaving surface region B and produce a correct reconstruction. Of course, due to A's occlusion, it may not have been possible to actually capture in a photograph the appropriate radiance leaving surface region B in the desired direction. However, because the representation is tightly integrated with the surface geometry, it is possible to detect this condition and use a hole filling algorithm to fill the missing data using data gathered from other views of the surface patch, such as C. Our system for resampling surface light fields from photographs includes such an algorithm. Because we can perform this step offline during the construction process, it does not affect the interactive rendering speed.

4.3. Distributing the resolution quota

A key issue in both systems is deciding how to split the resolution "quota" between the different dimensions of the parametrization. Surface light fields, by parametrizing the physically based and orthogonal dimensions of texture coordinate and world-space direction, allow an application to decide this distribution according to physical properties of the object, such as surface detail and shininess. Objects with detailed surface texture would require high resolution across the surface manifold dimensions. Insufficient resolution results in blurring of surface detail (figure 4.7). Objects with complex specular properties and sharp highlights would require high resolution in direction space. Along these dimensions, insufficient resolution results in poor reproduction of the radiance function, which results in effects such as dulling and "sloshing" of sharp specular highlights (figure 4.8). These properties are independent; thus the two resolutions can be scaled independently.

The lumigraph does not provide as clear a separation. Sloan, *et al*[11] examine the issue of division of resolution and propose a distribution of 256x256 on the *uv* plane and 32x32 on the *st* plane as giving "good results." Gortler, *et al* note that because the *uv* plane is expected to lie near to the surface of the object, *uv* can be thought of as an approximation to surface texture coordinates, and *st* can be thought of as an approximation to a direction parametrization. This suggests that the *uv* and *st* discretization in the lumigraph might also be scaled independently according to the surface texture and radiance function properties of the surface. Based on this idea, Sloan proposes subsampling *st* as a method of compressing lumigraphs and accelerating their rendering. Unfortunately, the correspondence between *uv* and *st*, and surface and direction, is only approximate. I now demonstrate several second-order effects, which further constrain the relative *uv* and *st* resolutions that produce convincing results. Surface light fields do not exhibit these effects.

4.3.1. Occlusion artifacts

Sparse resolution in *st* leads to artifacts in rendered images due to occlusions. This occurs when, given a pixel being rendered, some of the rays being interpolated are passing through occluding geometry and are therefore giving false data. Figure 4.4 illustrates this case. Consider a lumigraph rendering algorithm based on raytracing. After depth correction, the actual view ray, ray A, is approximated by interpolating rays B and C. However, note that ray C is occluded by geometry not hit by the original ray, and therefore gives incorrect data. More precisely, occlusions introduce discontinuities in the light field along the *st* dimensions. These discontinuities are not modeled well by the piecewise linear reconstruction of the lumigraph function if *st* is not densely sampled. Figure 4.9 illustrates the artifacts that can be generated by this phenomenon. It may be possible to correct this condition by storing depth with each ray and testing against the depth correction factor for the current ray. However, an accurate implementation of this correction may require sacrificing interactive speed. Surface light fields are not subject to this effect because rays are explicitly associated with points on the geometry.

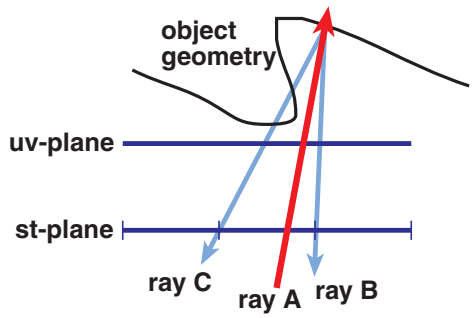

figure 4.4: Occlusion artifacts in a lumigraph.

4.3.2. Effect of depth errors

A large difference in the *st* and *uv* resolutions of a lumigraph leads to an increased sensitivity to errors in the depth correction step. To review, Levoy, *et al*[7] prefilter their light fields prior to rendering in order to eliminate aliasing caused by undersampling in the *st* and *uv* planes. This has the effect of reducing the apparent depth of field in rendered images. To sidestep this issue, Gortler, *et al* introduce a depth correction step into the lumigraph rendering algorithm. This allows them to choose rays that

originate from the same point on the geometry, effectively eliminating blurring across the surface.

An interactive rendering algorithm for a lumigraph, however, typically requires a simplified version of the geometry. This introduces small errors in depth computation, which may be further exacerbated by the approximations necessary for fast depth computation used by Gortler, *et al*. Such imperfections in depth correction may reintroduce small aliasing artifacts, such as double images on a small scale. Figure 4.5 illustrates the geometry of the situation. A simple analysis by similar triangles reveals that the amount of possible offset of rendered pixels, in *uv* pixels, is directly proportional to the ratio between the *st* and *uv* grid spacings, as well as proportional to the ratio between the depth error and the actual depth.

$$error = \frac{pes}{d(d+e)u}$$

where d is the actual depth, e is the error in depth, p is the distance between the two planes, s is the distance between adjacent st grid points, and u is the distance between adjacent uv grid points.

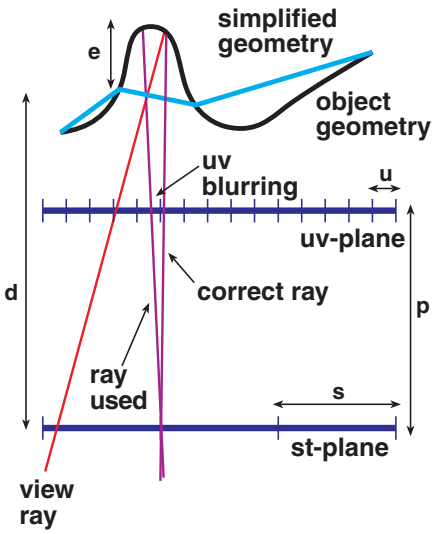

figure 4.5: The effect of simplified geometry.

In practice, because the ratio e/d is generally small, this effect is not very noticeable until there is a very large difference in the *uv* and *st* resolutions. Figure 4.10 demonstrates such a case. The geometry has been simplified to 199 triangles (from the original model of about 130,000 triangles) for the purposes of depth correction, and the *uv* and *st* resolutions are 416x416 and 4x4, respectively. Note the presence of double images of some of the scales on the fish's side. Although this is an extreme distribution of resolution between *uv* and *st*, it does represent a further constraint imposed on how sparse the sampling of *st* can be made. In particular, to construct a lumigraph with high-resolution texture (*uv*) data, it may be necessary to increase *st* resolution also.

Surface light fields are not subject to this type of constraint. Many surface simplification schemes include facilities for preserving some approximation of the surface parametrization. Textures mapped onto simplified geometry may be distorted, but will not exhibit these types of ghosting artifacts, regardless of the simplification level or the relative density of the *st* and $\theta\phi$ resolutions.

4.5. Conclusions

Although similar in their functions, lumigraphs and surface light fields exhibit some key differences related to their respective parametrizations. The parametrization of the 4D function employed by the surface light field can be made more uniform, according to the natural dimensions of position and direction, than can the parametrization employed by the lumigraph, leading to more efficient utilization of a potentially limited data set size. Furthermore, the surface light field provides a cleaner separation between the different dimensions of the parametrization. This allows those resolutions to be scaled independently according to the properties of the object to be reproduced and desired quality of the reproduction. By contrast, manipulating the parametrization employed by the lumigraph may expose additional artifacts.

The primary benefit of the lumigraph's parametrization is that it is basically geometry-independent. This may be useful for representing and rendering data sets with indefinite, unacquirable or exceedingly complex geometry. However, when geometry is available, surface light fields offer significant advantages.

5. Some context

This work is part of a larger project investigating methods of capturing, representing and reproducing the appearance of real objects. This section gives a brief overview of the larger project, including the process of constructing a surface light field, and lists some of the research areas involved.

5.1. Data acquisition

We currently gather raw data in the form of high-resolution geometry and a dense set of photographs. We compute geometry from range images obtained using a Cyberware Model 15 laser range scanner. In-house software is used to register separate range images, and a vrip[2] is used to reconstruct the surface as a triangle mesh (figure 5.1). Photographs can be obtained through a variety of means. Our current datasets include photographs taken using the Stanford spherical gantry, a camera mounted on a programmable gantry arm. Using this system, we capture a large number of video-resolution images with known relative pose (figure 5.2).

To generate a surface light field, we must register the image set to the geometry. Currently, we use in-house

software relying on user-supplied correspondences; however we are also investigating automatic methods. In future work, we also hope to explore the harder problem of dealing with a set of non-interregistered images from a hand-held camera. (Gortler uses a registration pattern for this purpose.) We also plan to investigate methods of reconstructing plausible data from a smaller set of photographs.

We generate the surface light field data set by projecting photo images onto the surface. Currently, it is assumed that the set of photos is large; therefore, for a particular direction, we select the "nearest" photo rather than interpolating. During this step, we fill holes in the data due to occlusions and other effects using an algorithm based on Gaussian smoothing[12].

5.2. Regionification and direction simplification

We decompose the geometry into texture regions and parametrize using a modified version of MAPS[6]. Instead of a hierarchy of simplification levels, our algorithm proceeds greedily from fine to coarse, similar to the progressive mesh simplification[5]. This is because we would like texture regions to exhibit certain properties: specifically, regions should be small and "flat." Flatness improves performance by increasing the effectiveness of the region culling optimization described in section 3.1. It also may improve the direction simplification process described next.

A current active area of research is construction of an optimal set of direction samples for the direction mesh. In the data set used in current tests, every texture region uses the same direction mesh, and furthermore, the direction mesh is roughly uniform over direction space. However, it is conceivable that a different tessellation may provide higher quality results for a given number of directions. In particular, if a piece of surface has a largely diffuse texture with a single sharp specular highlight pointing in a single direction, it would be useful to concentrate direction samples in the direction of the specular highlight. We are investigating methods of optimizing the direction mesh through mesh simplification algorithms.

6. Conclusions and discussion

Surface light fields hold significant promise for reproducing the appearance of real objects. We have demonstrated its effectiveness with a real object with complex surface characteristics.

This project offers two main contributions to the study of surface light fields. First, I have demonstrated a practical implementation of a surface light field rendering algorithm capable of frame rates reasonable for interactive applications. My current implementation achieves 2.5 frames per second on a three-year-old workstation, and close to 6 frames per second on a high-end server. Second, I have elucidated the behavior of the representation in the face of simplification, and shown that a similar representation, the lumigraph, does not fare as well.

I have optimized the current implementation and studied its behavior on two SGI systems. However, with the recent rise of PCs, it would also be useful to understand the algorithm's behavior on PC graphics hardware and optimize accordingly. The current implementation is designed to be cross-platform, so this process should not be difficult.

Considerable work remains to be done in developing robust processes for constructing, compressing and analyzing surface light fields. Some of these directions are mentioned in the section 5. In addition, although surface light fields are designed for generality, able to represent arbitrary surface radiance, an interesting area of future work might be to explicitly model information about surface BRDF and positions of light sources to improve efficiency of the representation. This might also lead to representations capable of handling nonstatic lighting and additional applications.

7. Acknowledgments

I would like to thank my adviser, Brian Curless, for his support and good humor, Daniel Wood for many suggestions concerning the rendering algorithm and for putting up with my constant OpenGL-related complaints, Jonathan Shade for sharing his considerable knowledge of graphics hardware and for stealing time on Stanford's Infinite Reality, and to Stanford University for use of their computing and camera gantry facilities. Further thanks go to Tom Duchamp and Werner Stuetzle for their theoretical insight, Steve Gortler for his lumigraph code (although most of it wasn't used), and Fred Pighin for commenting on an early version of this paper, and for putting up with my constant "Star Wars" music. Finally, I would like to thank Jesus Christ, my Lord and Savior, without whom I'd have far larger problems than our out-of-date compilers. This work was funded by an Osberg Fellowship.

References

[1] E.H. Adelson and J.R. Bergen. “The Plenoptic Function and the Elements of Early Vision” In *Computation Models of Visual Processing*. MIT Press, 1991.

[2] Brian Curless and Marc Levoy. “A Volumetric Method for Building Complex Models from Range Images” In *SIGGRAPH 96 Conference Proceedings*. pages 303-312. ACM SIGGRAPH, Addison-Wesley, August, 1996.

[3] Paul E. Debevec, Camillo J. Taylor and Jitendra Malik. “Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach” In *SIGGRAPH 96 Conference Proceedings*. pages 11-20. ACM SIGGRAPH, Addison-Wesley, August, 1996.

[4] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski and Michael Cohen. “The Lumigraph” In *SIGGRAPH 96 Conference Proceedings*. pages 43-54. ACM SIGGRAPH, Addison-Wesley, August, 1996.

[5] Hughes Hoppe. “Progressive Meshes.” In *SIGGRAPH 96 Conference Proceedings*. pages 99-108. ACM SIGGRAPH, Addison-Wesley, August, 1996.

[6] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar and David Dobkin. “MAPS: Multiresolution adaptive parametrization of surfaces” In *SIGGRAPH 98 Conference Proceedings*. pages 95-104. ACM SIGGRAPH, Addison-Wesley, July, 1998.

[7] Marc Levoy and Pat Hanrahan. “Light Field Rendering” In *SIGGRAPH 96 Conference Proceedings*. pages 31-42. ACM SIGGRAPH, Addison-Wesley, August, 1996.

[8] Gavin Miller, Steven Rubin and Dulce Poncelen. “Lazy decompression of surface light fields for precomputed global illumination” In *Eurographics Rendering Workshop*. 1998.

[9] Kari Pulli. *Surface Reconstruction and Display from Range and Color Data*. Ph.D. dissertation. University of Washington. 1997.

[10] Yoichi Sato, Mark D. Wheeler and Katsushi Ikeuchi. “Object Shape and Reflectance Modeling from Observation.” In *SIGGRAPH 97 Conference Proceedings*. pages 379-387 ACM SIGGRAPH, Addison-Wesley, August, 1997.

[11] Peter-Pike Sloan, Michael F. Cohen and Steven J. Gortler. “Time Critical Lumigraph Rendering” In *1997 Symposium on Interactive 3D Graphics*. 1997.

[12] Gabriel Taubin. “A signal processing approach to fair surface design.” In *SIGGRAPH 95 Conference Proceedings*. pages 351-358. ACM SIGGRAPH, Addison-Wesley, August, 1995.

[13] Bruce Walter, Gün Alppay, Eric P. F. Lafortune, Sebastian Fernandez and Donald P. Greenberg. “Fitting virtual lights for non-diffuse walkthroughs” In *SIGGRAPH 97 Conference Proceedings*. pages 49-56. ACM SIGGRAPH, Addison-Wesley, August, 1997.

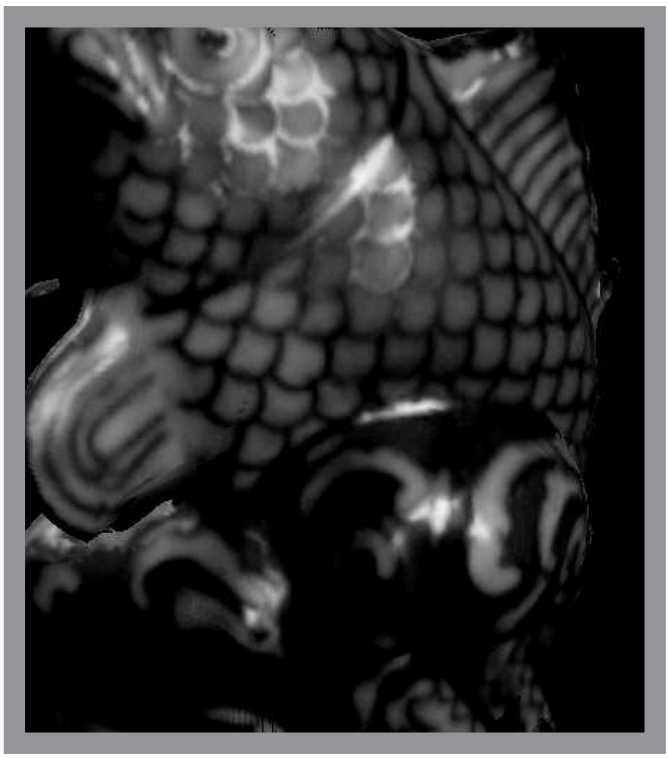

Image A:
Surface light field

Figure 4.6:
With similar-sized data sets, the surface light field gives better-quality images

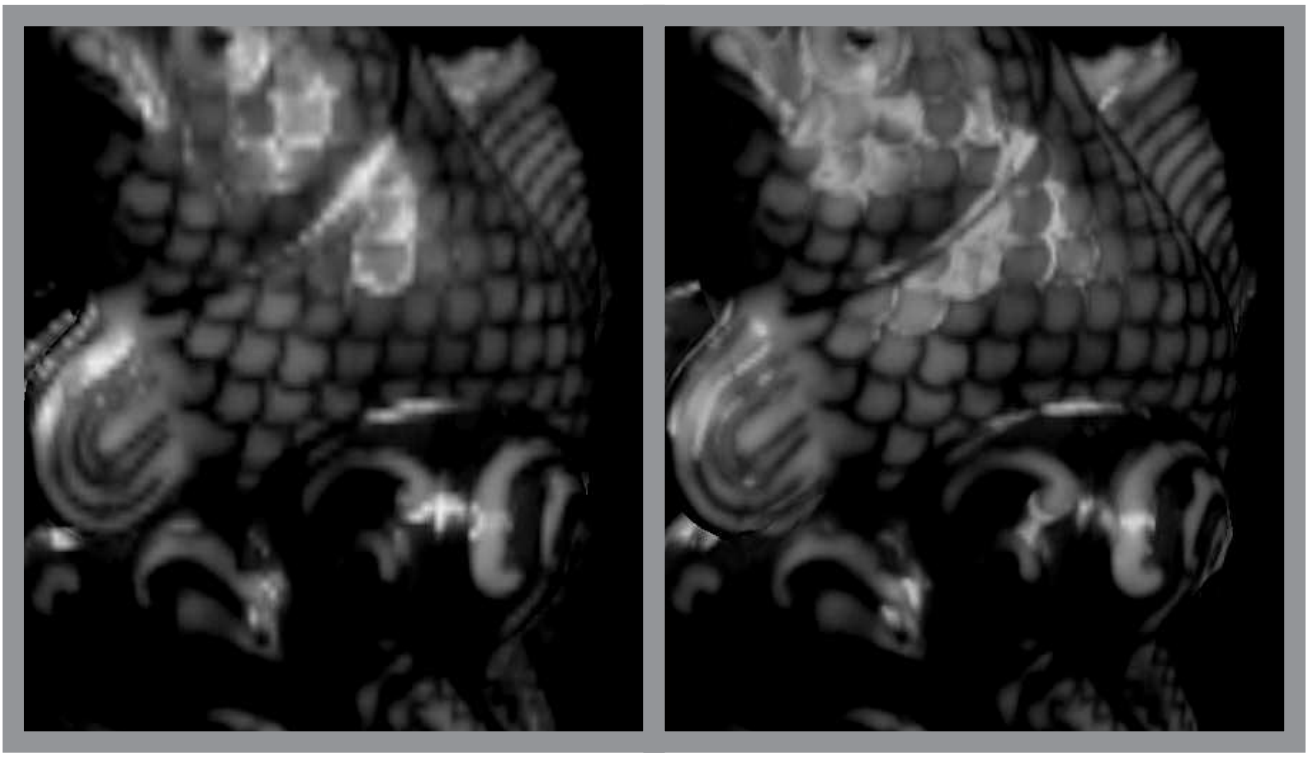

Image B:
Lumigraph, 238x238 UV, 7x7 ST
Insufficient surface detail resolution

Image C:
Lumigraph, 416x416 UV, 4x4 ST
Insufficient directional resolution.

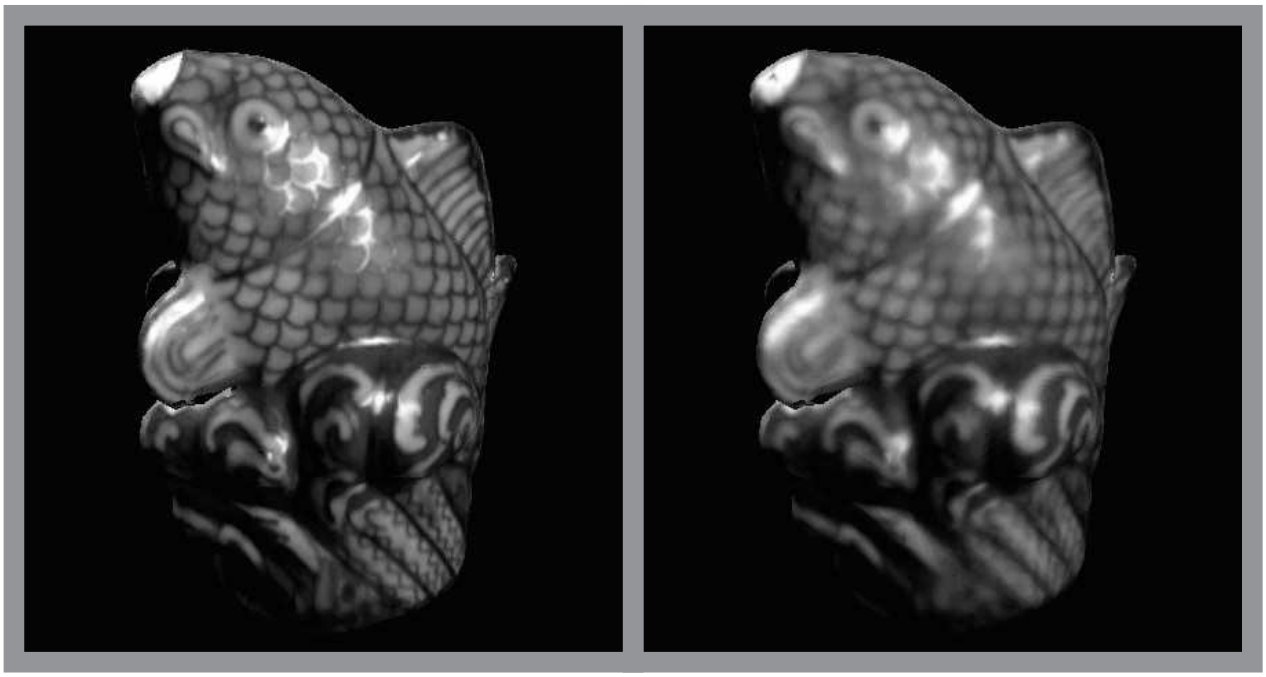

Figure 4.7: Low texture resolution results in blurring. Texels in the second surface light field are subsampled 4:1.

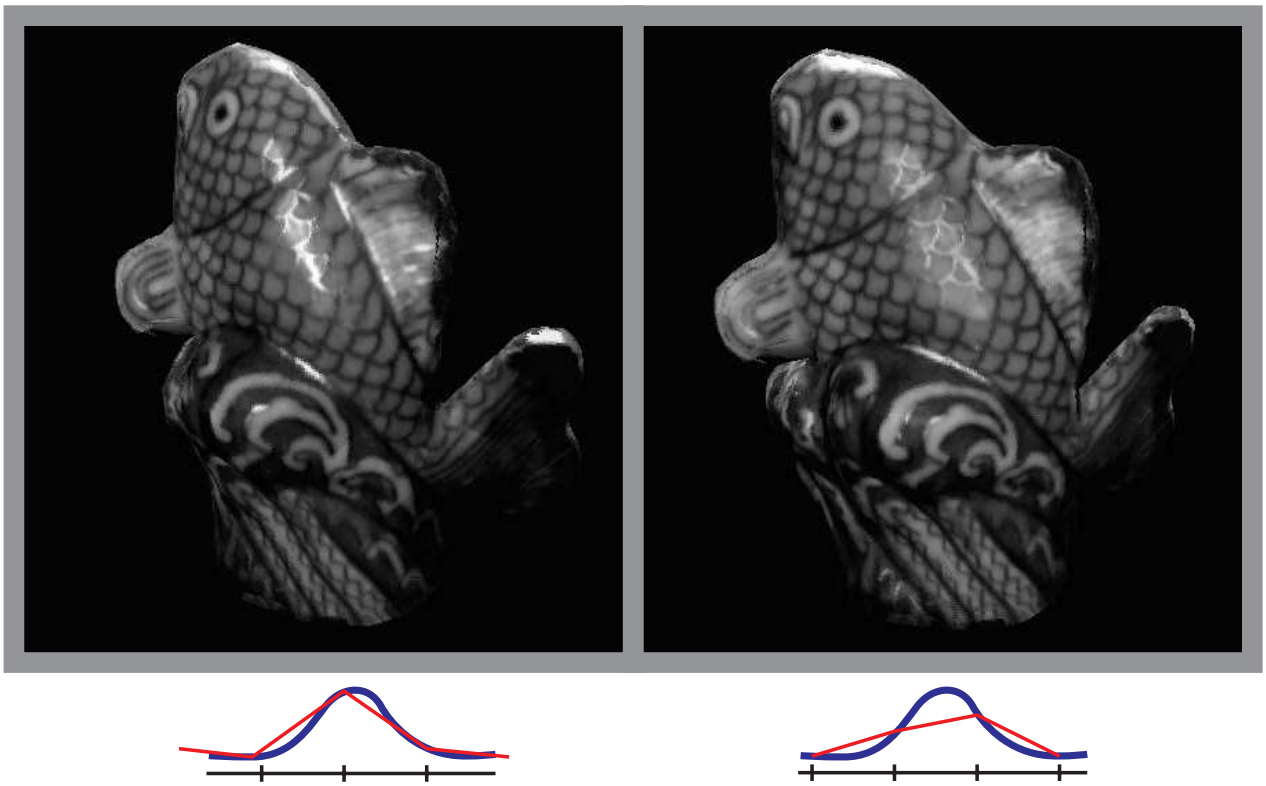

Figure 4.8: Low direction resolution can result in “sloshing” of highlights due to the coarse piecewise-linear approximation. Two renderings of the same surface light field with 66 uniformly-distributed directions.

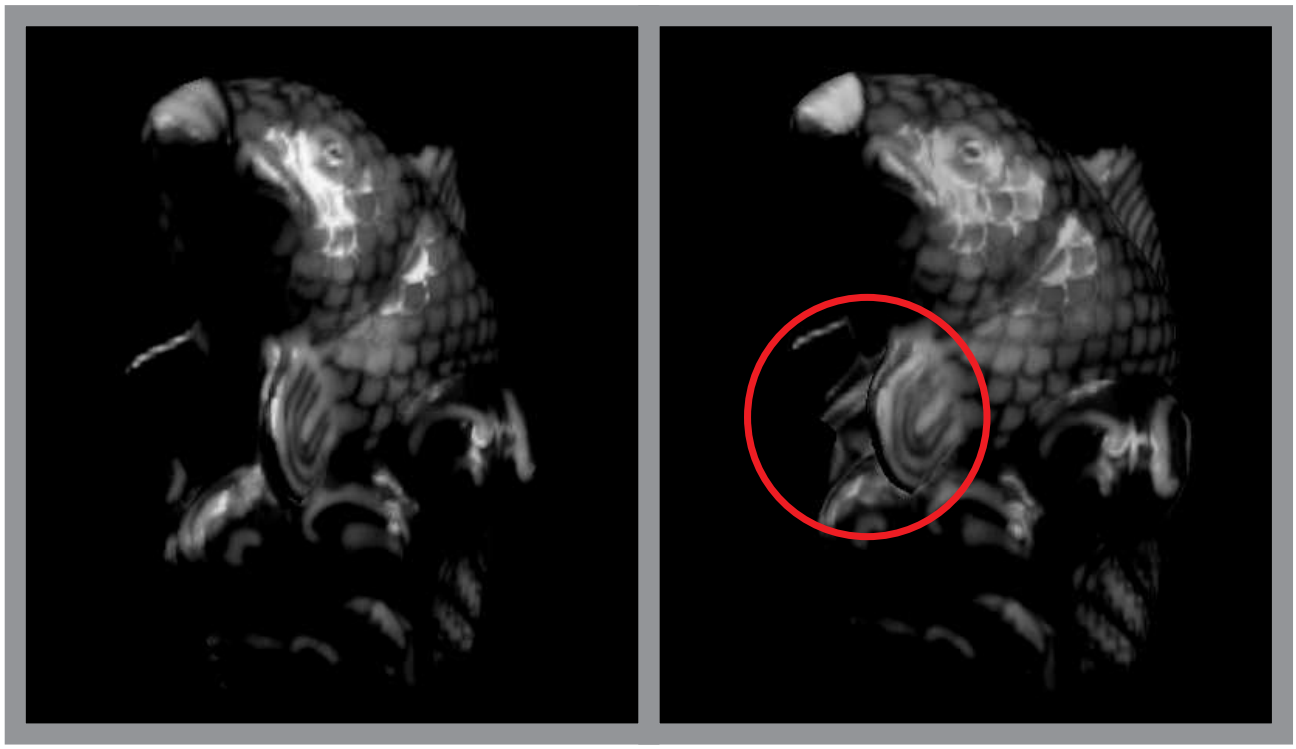

Figure 4.9: Two nearby views of a lumigraph, showing the appearance of an occlusion artifact. Lumigraph 334x334 *uv*, 5x5 *st*

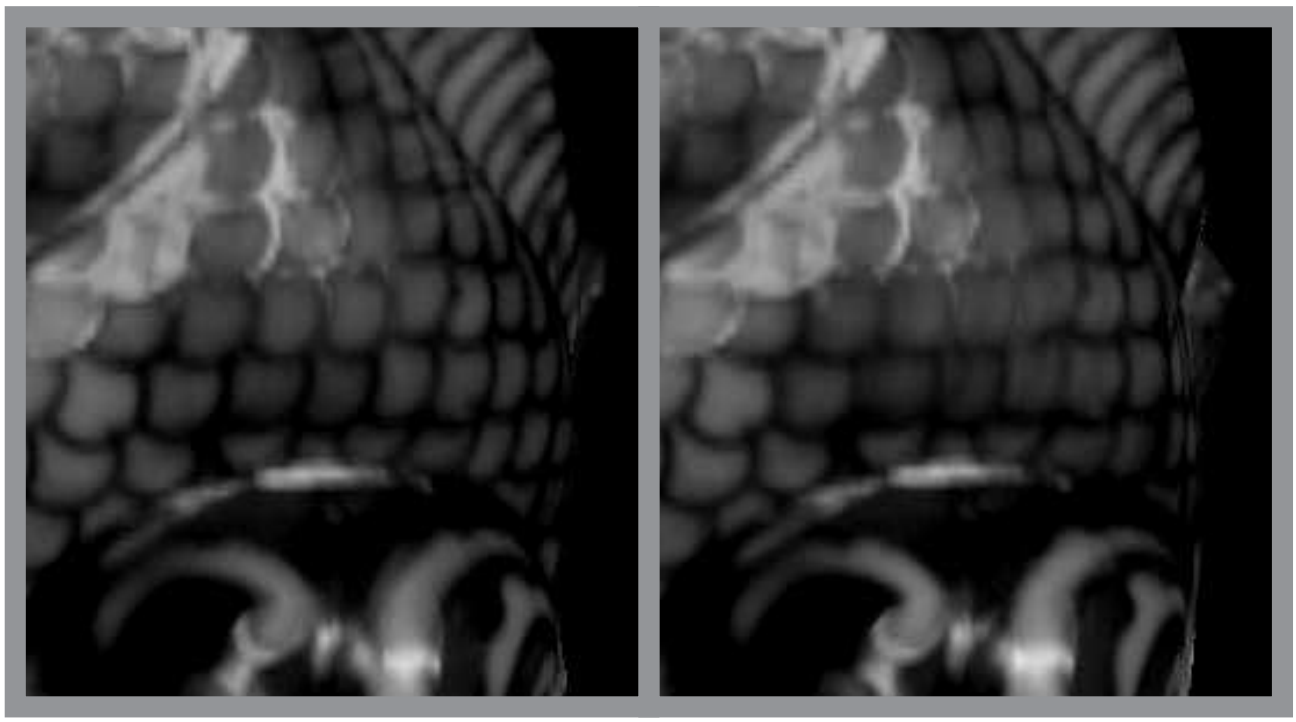

Figure 4.10: Inexact depth artifacts (Lumigraph 416x416 *uv*, 4x4 *st*)

3184 geometry triangles.
Minimal ghosting due to bad geometry.

199 geometry triangles.
Minor but noticeable ghosting due to bad geometry.

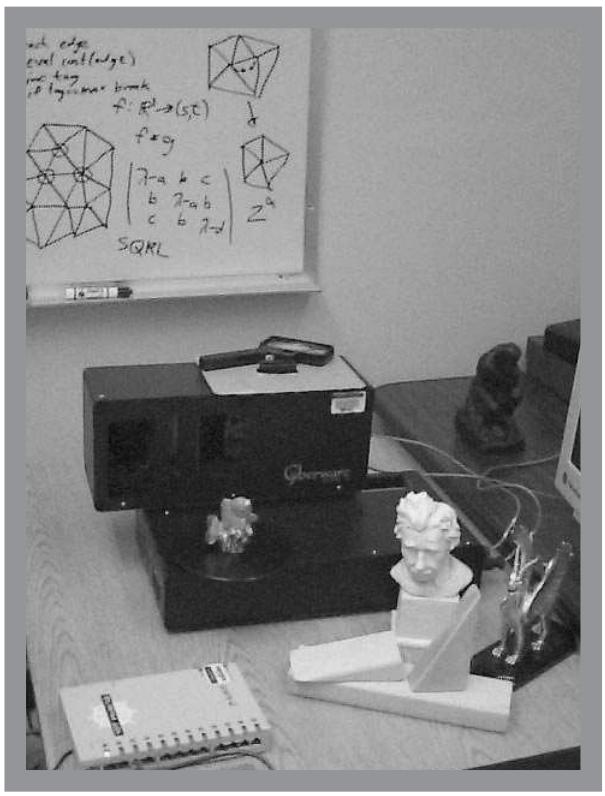

Figure 5.1:

Cyberware Model 15 laser range scanner

Object geometry (130,000 triangles)

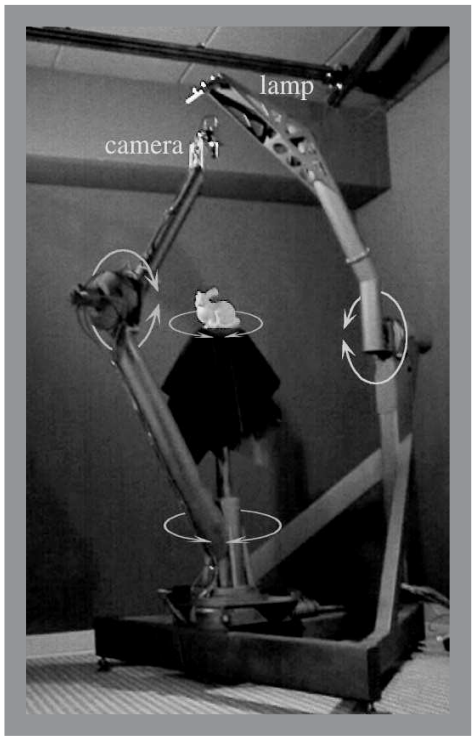

Figure 5.2:

Stanford Spherical Gantry

680 input photographs

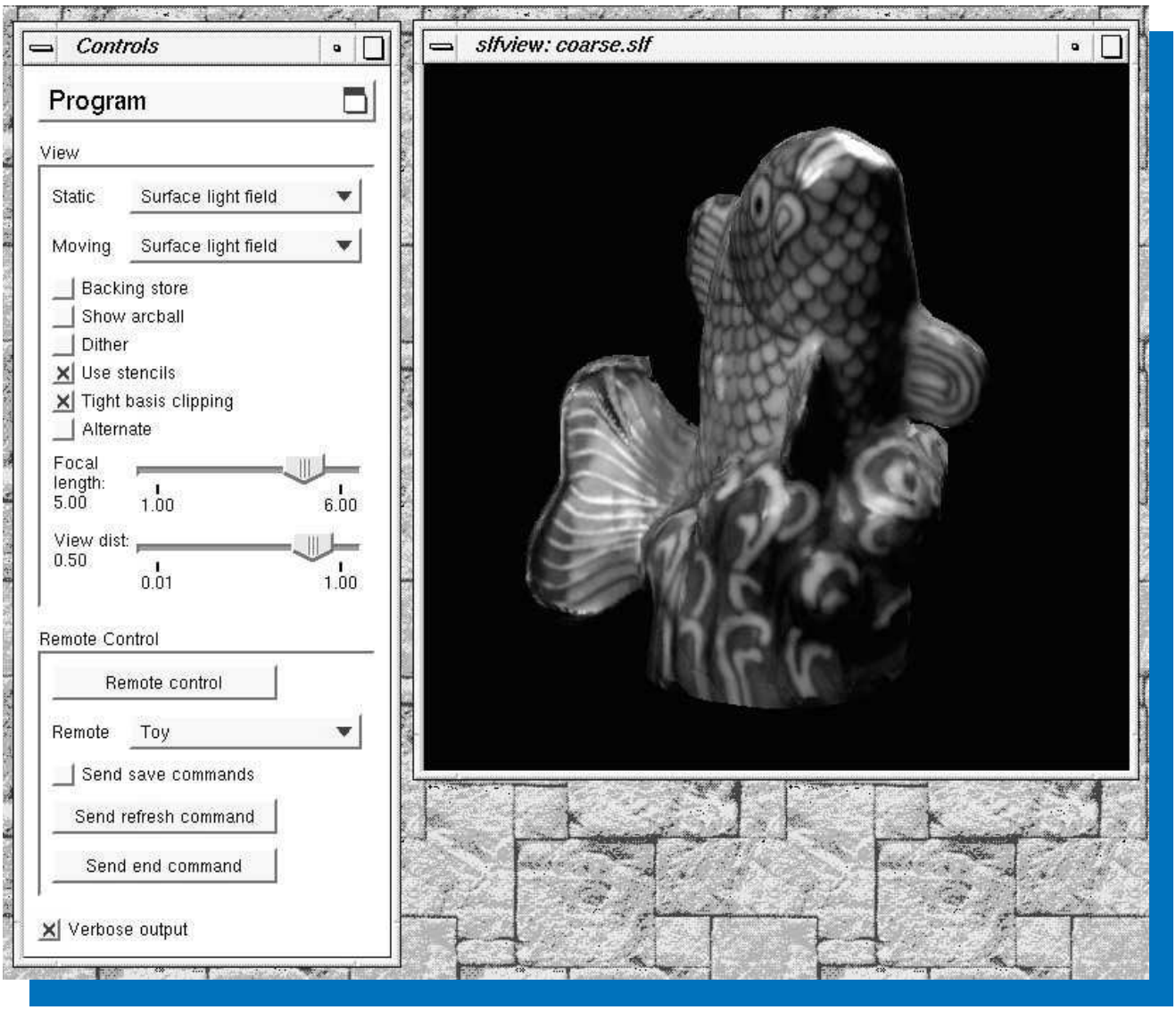

Figure 6: slfview, the interactive surface light field viewer