Opacity Light Fields: Interactive Rendering of Surface Light Fields with View-Dependent Opacity

Daniel Vlasic [*] Hanspeter Pfister [†] Sergey Molinov [‡] Radek Grzeszczuk[‡] Wojciech Matusik[*]

Abstract

We present new hardware-accelerated techniques for rendering surface light fields with opacity hulls that allow for interactive visualization of objects that have complex reflectance properties and elaborate geometrical details. The opacity hull is a shape enclosing the object with view-dependent opacity parameterized onto that shape. We call the combination of opacity hulls and surface light fields the *opacity light field*. Opacity light fields are ideally suited for rendering of the visually complex objects and scenes obtained with 3D photography. We show how to implement opacity light fields in the framework of three surface light field rendering methods: view-dependent texture mapping, unstructured lumigraph rendering, and light field mapping. The modified algorithms can be effectively supported on modern graphics hardware. Our results show that all three implementations are able to achieve interactive or real-time frame rates.

CR Categories: I.3.2 [Computer Graphics]: Picture/Image Generation—Digitizing and Scanning, Viewing Algorithms;

Keywords: Image-based rendering, 3D photography.

1 Introduction

We are interested in interactive rendering of 3D objects that have complex reflectance properties and elaborate geometrical details. Good examples are trees, furry Teddy bears, feathers, or other models typically acquired using 3D photography [Matusik et al. 2002]. Our approach employs images to represent both view-dependent appearance (using surface light fields) and view-dependent shape (using opacity hulls). We call the resulting representation *opacity light fields*: images of the object with texture and opacity are mapped onto a coarse polygon mesh and rendered from arbitrary viewpoints. Here we propose and describe three new algorithms for rendering of opacity light fields that are able to faithfully reproduce visually complex objects and that can be efficiently implemented on modern graphics hardware.

[*]MIT, Cambridge, MA.
Email: [drdaniel,wojciech]@mit.edu
[†]MERL, Cambridge, MA.
Email: pfister@merl.com
[‡]Intel Labs, Santa Clara, CA.
Email: [sergey.molinov,radek.grzeszczuk]@intel.com

After a review of previous work we present a general overview of our approach in Section 3. Then we describe how opacity light fields can be applied to three popular light field rendering methods: view-dependent texture mapping (Section 4), unstructured lumigraph rendering (Section 5), and light field mapping (Section 6). Section 7 presents results from our implementations using various models acquired with 3D photography. In Section 8 we compare the three approaches and give an outlook on future work.

2 Previous Work

We are using an image-based rendering approach to visualize the wide range of models than can be acquired with 3D photography. Early image-based rendering methods [McMillan and Bishop 1995; Chen and Williams 1993; Levoy and Hanrahan 1996; Gortler et al. 1996] require a large set of images to achieve high rendering quality. A more scalable approach is to parameterize the textures directly on the surface of the object. This approach enables more accurate interpolation between the images and it has been employed by view-dependent texture mapping (VDTM) [Debevec et al. 1996; Pulli et al. 1997; Debevec et al. 1998], surface light field rendering [Miller et al. 1998; Wood et al. 2000; Nishino et al. 1999], unstructured lumigraph rendering (ULR) [Buehler et al. 2001], and light field mapping (LFM) [Chen et al. 2002]. We will describe VDTM, ULR, and LFM in more detail in the remainder of this paper.

Current surface light field approaches are limited by the complexity of object's geometry. A dense mesh requires many vertices and leads to a decrease in rendering performance. The artifacts of using a coarse mesh are most noticeable at the silhouette of the object. Sander et al. [2000] use silhouette clipping to improve the visual appearance of coarse polygonal models. However, their method might be impractical for complex silhouette geometry like fur, trees, or feathers. Lengyel et al. [2001] use concentric, semi-transparent textured shells to render hair and furry objects. To improve the appearance of object silhouettes they use extra geometry – called textured fins – on all edges of the object. Neither method uses view-dependent textures and opacity from images.

To accurately render silhouettes of high complexity with only simple geometry we are using opacity hulls [Matusik et al. 2002]. Opacity hulls use view-dependent alphas for every surface point on the visual hull [Laurentini 1994] or any other geometry that is bigger or equal to the geometry of the real object. Matusik et al. [2002] use opacity hulls with surface light fields (for objects under fixed illumination) and surface reflectance fields (for objects under varying illumination). They demonstrate that opacity light fields are very effective to render objects with arbitrarily complex shape and materials from arbitrary viewpoints. However, their models are impractically large, on the order of 2-4 Gigabytes, and the point-based rendering algorithm is not hardware accelerated. Their unoptimized implementation takes about 30 seconds per frame. In this paper, we use polygon rendering algorithms to efficiently render opacity light fields on graphics hardware.

Polygon-based rendering of opacity light fields requires visibility ordering. For VDTM and ULR we use depth-sorting of primitives using a BSP-partitioning of 3D space [Fuchs et al. 1980]. Unfortunately, depth-sorting of triangles is particularly hard in the case of LFM, since it is a multi-pass algorithm and the triangles have to be rendered in the order in which they are stored in texture images. Recently several algorithms have been proposed for computing visibility ordering with the aid of graphics hardware [Krishnan et al. 2001; Westermann and Ertl 1998; Everitt 2001]. These algorithms avoid the overhead of building complicated data structures and are often simple to implement. We combine hardware-accelerated visibility ordering with LFM into a novel, fully hardware-accelerated algorithm for rendering of opacity light fields.

In the remainder of this paper we show how to implement opacity light fields on modern graphics hardware using modified versions of VDTM, ULR, and LFM. The resulting new methods have some limitations for highly specular surfaces due to the relatively small number of textures. Furthermore, because surface light fields capture object appearance for fixed illumination, we are not able to render the objects under varying lighting conditions. Neither of these limitations restrict the usefulness of opacity light fields. They are ideally suited for rendering of the visually complex objects and scenes obtained with 3D photography, and they can efficiently be implemented on modern graphics hardware.

3 Overview

The input to our algorithms consists of a triangular mesh and a set of rectified RGBA images captured from known camera locations. The fundamental algorithm for all three rendering methods (LDTM, ULR, and LFM) consists of the following steps

Visible Views: For each mesh vertex v, find the set of visible views. A view is considered visible if the triangle ring around v is completely un-occluded in that view.

Closest Views: For each mesh vertex v, find k visible views closest to a given desired viewing direction d. In our implementation $k \leq 3$, as described later.

Blending Weights: Compute the blending weights for the corresponding closest views, where higher weights are assigned to views closer to d.

Rendering: Render each triangle by blending images from the closest views of its three vertices.

Our methods differ in realization of each stage, but share a number of common techniques. Subsequent sections describe each of the real-time opacity light field rendering methods in detail.

4 View-Dependent Texture Mapping

View-dependent texture mapping for image-based rendering was introduced by Debevec et al. [1996], Pulli et al. [1997] and Debevec et al. [1998]. Our approach is closest to [Debevec et al. 1998], whose method achieves photorealism by blending pictures of a real-life object as projective textures on the surface of a 3D model. To find suitable textures to blend, they construct for each polygon of the model a *view map* – a planar triangulation whose vertices correspond to viewpoints of original pictures. To construct it, one has to first project unit vectors towards visible views onto the plane of the polygon, and then resample them to fit a predefined regular triangulation. Specifically, each vertex of the triangulation corresponds to the closest projected original view.

View maps are queried for a triangle that contains the projection of a new desired view. Original pictures corresponding to vertices of this triangle are blended with weights equal to barycentric coordinates of the projected new view within the triangle. While this approach assures smooth transitions between views, it introduces resampling errors. Furthermore, since blending is done per-polygon, continuity across triangle edges is not preserved.

Chen et al. [2002] present an approach that preserves continuity across triangles by blending the views on a per-vertex basis. At each vertex, they construct a Delaunay triangulation of the visible views, which they resample into their own view map – a dense (32×32) uniform grid of views computed by blending the original set of views. This approach efficiently renders complex objects with controllable precision. However, light field approximation using matrix factorization makes the LFM approach sensitive to the accuracy of the data and the model. Instead, our modified VDTM method uses the latest graphics hardware to enable fast rendering directly from the original views, keeping their sharpness and detail.

4.1 General Algorithm

Similarly to Debevec et al. [1998], we blend original views using barycentric weights within a triangulation using local coordinate frames for each vertex. However, we blend on a per-vertex basis, as in Chen et al. [2002]. Finally, in contrast to both methods, we do not resample original views: to find closest views and compute corresponding blending weights, we query the original Delaunay triangulations of visible views at each vertex v. In this process, normalized viewing vectors (from vertex to camera) of visible views are first projected onto the tangential plane of v. They are subsequently triangulated with an efficient and robust 2D Delaunay triangulation algorithm [Shewchuk 1996] (see Figure 1).

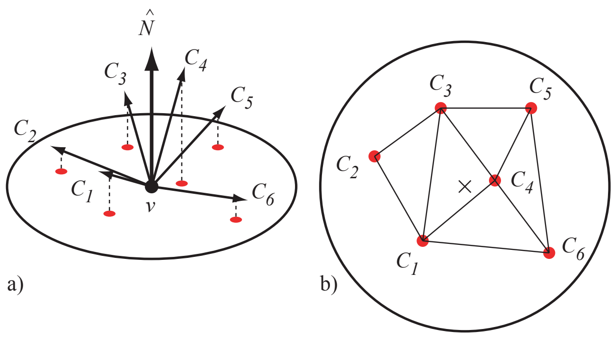


Figure 1: *Delaunay triangulation of visible views. (a) Viewing directions of visible camera views (C_1-C_6) at vertex v are first projected onto the tangential plane of v. (b) Then a 2D Delaunay triangulation of the projected views is constructed.*

In this setting, closest original views for a novel viewing direction, d, are vertices of a Delaunay triangle that d projects into. The blending weights are, consequently, equivalent to barycentric weights of d within the triangle, as depicted in Figure 2a.

Sometimes, however, a desired view direction d does not fall within any of the triangles. In such cases we extrapolate by blending between nearby views on the convex hull of the triangulation (Figure 2b). Those views are found by intersecting convex hull edges with the line connecting d and x, the centroid of the convex hull. The views belonging to the intersecting edge are weighed relative to their distances to the intersection. Specifically, weights are larger for views closer to the intersection. With this technique we ensure continuity of the blending weights.

Finally, if a triangulation is impossible altogether (i.e., there are less than three visible views), all available views are used and weighted according to their distance to d. In our experience, provided that the mesh is smooth and original views densely sampled, this situation rarely occurs.

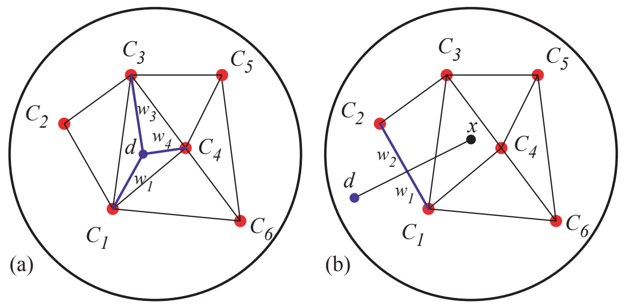


Figure 2: *Closest views and blending weights. (a) For a new viewing direction d, closest cameras are Delaunay vertices of the triangle that d projects into ($C_1C_3C_4$ in the figure). Corresponding blending weights are the barycentric coordinates of d within the triangle (w_1, w_3, w_4). (b) If d falls outside the triangulation, blending occurs between two nearby views on the convex hull of the triangulation (C_1 and C_2 in the figure). x denotes the centroid of the convex hull.*

When computing planar triangulations, we are distorting the distances between views via projection, as well as ignoring views below the tangential plane. Ideally, one should construct spherical Delaunay triangulations to avoid these issues. That approach would also eliminate the need for extrapolation.

Once we have computed per-vertex closest views and corresponding blending weights, we can proceed to the final step of the algorithm - rendering on a per-triangle basis. During rendering, each triangle is textured with the set of closest views of its three vertices. This set can contain up to nine textures, but usually has less since nearby vertices often share closest views. Texture blending weights are combined in the following manner: a vertex keeps the computed weights of its own closest views, and sets weights corresponding to other vertices' closest views to zero (see example in Figure 3). This is identical to combining vertex light fields with hat functions as described in [Chen et al. 2002]. Combining weights in this manner guarantees continuity across triangle edges.

Transparency information in our data imposes an ordering constraint for rendering opacity light fields on commodity graphics hardware. To ensure correct alpha blending, triangles must be rendered in back-to-front order. To accomplish this, we create a BSP tree [Fuchs et al. 1980] of the mesh during preprocessing. We perform visibility, closest views, and weight computation on BSP tree triangles, and traverse the tree while rendering. The actual rendering amounts to blending up to nine textures per triangle, and is carried out completely by the graphics hardware.

4.2 Optimizations

A naive implementation of the algorithm described above does not achieve real-time performance. We are able to achieve much higher frame rates through texture compression, reducing per-frame CPU computation through caching and fast data structures, as well as by exploiting modern graphics hardware capabilities.

Opacity light fields inherently encompass large datasets – on the order of several Gigabytes, mostly comprised of textures. To maximize hardware rendering performance, it is imperative to fit all textures in the on-card graphics memory. We are able to do this by compression; namely, we crop each view to the exact bounding box of the actual object, and resize the resulting image to approximately a quarter its size. As we will show in Section 7, we can fit several Gigabytes of original camera images into the 128MB video memory commonly available on current graphics hardware.

The computation of closest views and weights runs on the CPU, and it is advantageous to speed it up as much as possible. Therefore, Delaunay triangulations are pre-computed and stored along with their connectivity information, enabling fast breadth-first search during runtime. Under the assumption that a viewer will move the virtual camera continuously around the model, the viewing direction d will mostly stay in the same Delaunay triangle or move to a neighboring one, resulting in a short search path.

Rendering essentially blends camera views, which are, by definition, projective textures from the point of the corresponding cameras. Although projective texture mapping runs efficiently on hardware, and is utilized in [Debevec et al. 1998], caching the projective texture coordinates can save a number of GPU cycles. Therefore, we precompute per-vertex texture coordinates for all visible views and store them within the mesh data.

During rendering, each triangle is textured with the set of closest views of its three vertices. We have to define up to nine textures, nine blending weights, and nine pairs of texture coordinates for each triangle vertex. Textures and coordinates are retrieved from the set of closest views and the cached coordinate data, respectively. Texture weights, on the other hand, are combined as shown in Figure 3. Since the same vertex corresponds to different sets of textures in different triangles, data has to be presented to hardware as triangle lists (not strips or fans). After preparation, textures, weights, and coordinates are sent to the graphics card in the form of input streams. Rendering proceeds using programmable vertex and pixel shaders found on modern graphics processing units (GPUs).

Opacity information, as mentioned earlier, enforces back-to-front ordering of rendered triangles, requiring construction of a BSP tree. Although the tree can be created in a preprocess and the computational overhead of its traversal is negligible, additional vertices are created by splitting triangles during BSP tree construction. They significantly decrease the rendering performance. To alleviate the problem, many different BSP trees of the same mesh are constructed by random insertion of triangles, and only the smallest tree is kept.

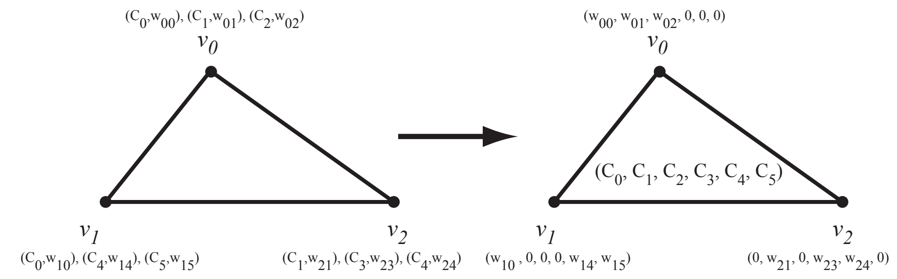
Alternatively, we could use the depth peeling algorithm described in Section 6 for depth sorting of primitives. The computation of the blending weights could then only be done once, making the algorithm much faster. However, current graphics hardware is not capable of blending nine textures and computing depth layers without significant degradation in performance. Consequently, we use BSP trees for VDTM and ULR.

Opacity affects the rendering performance in another way – each triangle must be rendered in a single pass. If there were more than one pass per triangle, alpha blending could not be performed correctly. Since our algorithm blends up to nine different textures per triangle, only the latest graphics hardware can run it in one pass. We currently use ATI's Radeon 9700 card under DirectX 9.0. To run this algorithm on older graphics hardware requires several rendering passes. Each pass renders triangles into a separate image buffer, and proceeds by merging them with the front buffer. Obviously, this would be detrimental to performance.

5 Unstructured Lumigraph Rendering

Unstructured lumigraph rendering [Buehler et al. 2001] can be used in place of VDTM as alternative blending algorithm for opacity light fields. The approach differs from the previously described VDTM only in finding closest views and blending weights. Instead of computing Delaunay triangulations, ULR uses a ranking function based on the angles between original views and desired view. For each vertex v and some viewing direction d, the algorithm finds the k nearest visible views by comparing the angles they subtend with d (Figure 4).

Blending weights are computed according to the following for-



$(C_0,w_{00}), (C_1,w_{01}), (C_2,w_{02})$
v_0
v_1
$(C_0,w_{10}), (C_4,w_{14}), (C_5,w_{15})$
v_2
$(C_1,w_{21}), (C_3,w_{23}), (C_4,w_{24})$
$(w_{00}, w_{01}, w_{02}, 0, 0, 0)$
v_0
$(C_0, C_1, C_2, C_3, C_4, C_5)$
v_1
$(w_{10}, 0, 0, 0, w_{14}, w_{15})$
v_2
$(0, w_{21}, 0, w_{23}, w_{24}, 0)$

Vertex	Closest Camera Views	Blending Weights	Triangle Closest Views	Triangle Blending Weights
V_0	C_0, C_1, C_2	w_{00}, w_{01}, w_{02}	$C_0, C_1, C_2, C_3, C_4, C_5$	$w_{00}, w_{01}, w_{02}, 0, 0, 0$
V_1	C_0, C_4, C_5	w_{10}, w_{14}, w_{15}		$w_{10}, 0, 0, 0, w_{14}, w_{15}$
V_2	C_1, C_3, C_4	w_{21}, w_{23}, w_{24}		$0, w_{21}, 0, w_{23}, w_{25}, 0$

Figure 3: *Vertex weight combining example. After computing per-vertex closest views and blending weights (shown on the left side of the figure), we have to combine them for each triangle. The union of closest views of the three vertices defines the set of views for the triangle. Before rendering, each vertex needs to have a weight associated with each of the views of the set. If that weight is not already computed, it is set to zero (show on the right side).*

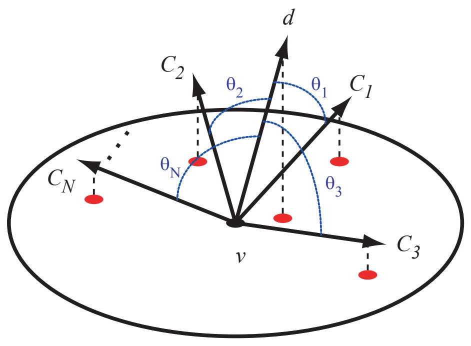


Figure 4: *Unstructured lumigraph rendering. For vertex v and desired view d, blend the closest visible views (C_i) according to the angles θ_i.*

mula:

$$r_i = \frac{1 - \frac{p_i}{t}}{p_i} = \frac{1 - \frac{p_i}{p_k}}{p_i}, \qquad w_i = \frac{r_i}{\sum_{j=1}^{k-1} r_j} \tag{1}$$
$$0 < i < k, \qquad \theta_i \le \theta_j, \qquad \forall i < j$$

Here, r_i represent relative weights of the closest $k-1$ views. Those are normalized to sum up to one, resulting in actual weights w_i. The penalty term p_i quantifies the angular difference between C_i and d: It falls off as the angle between them decreases. Its presence in the denominator of the whole expression enforces epipole consistency. The penalty p_k of the k-th closest view, which is not used for blending, defines an adaptive threshold t. This technique enforces smooth view transitions: As a view exits the closest view set, its weight falls off to zero.

The ULR approach is advantageous due to its simplicity and flexibility. Unlike VDTM, weight computation does not depend on the mesh, making ULR more general. Furthermore, although it requires more per-frame computation than VDTM, it runs interactively.

5.1 Implementation Details

The penalty function as described in [Buehler et al. 2001] can be resolution sensitive. In our implementation, however, we ignore resolution penalties since our original views have the same resolution and similar distance to the object. As penalty we use the following simple expression:

$$p_i = 1 - \cos\theta_i \tag{2}$$

The relative weights then follow directly from Equation 1:

$$r_i = \frac{1 - \frac{1-\cos\theta_i}{1-\cos\theta_k}}{1-\cos\theta_i} = \frac{\cos\theta_i - \cos\theta_k}{(1-\cos\theta_i)(1-\cos\theta_k)} \tag{3}$$
$$0 < i < k, \qquad \cos\theta_i \ge \cos\theta_j, \qquad \forall i < j$$

Because of normalization, we can drop the constant $(1 - \cos\theta_k)$ from the denominator of the expression, resulting in:

$$r_i = \frac{\cos\theta_i - \cos\theta_k}{1 - \cos\theta_i} \tag{4}$$

The ULR algorithm begins by computing cosines (dot products) of the angles between the desired view and each original visible view at each vertex v. It finds $k = 4$ closest views and blends three of them using Equations 1 and 4. If only three views are available, the same equations can be used with $k = 3$. For vertices with less than three visible views, we use all available views and the same equations with a threshold of one ($\cos\theta_k$ is set to zero). A disadvantage of the ULR approach compared to VDTM is that all cosines have to be computed for each frame, slowing down the weight computation.

Although Delaunay triangulations are no longer needed, visibility information is still useful – per-vertex visible views are computed and stored during a preprocess. Note, however, that visibility need not be used: for objects with relatively small occlusions (such as our angel model), all camera views can be considered for blending at each vertex, resulting in a higher-quality rendering.

Other aspects of the algorithm – BSP tree construction and traversal, resizing textures, caching texture coordinates, combining weights, hardware shaders, and rendering – are the same as for VDTM.

6 Light Field Mapping

Light field mapping assumes the light field function approximated by a sum of a small number of products of lower-dimensional functions

$$f(r,s,\theta,\phi) \approx \sum_{k=1}^{K} g_k(r,s)\, h_k(\theta,\phi). \tag{5}$$

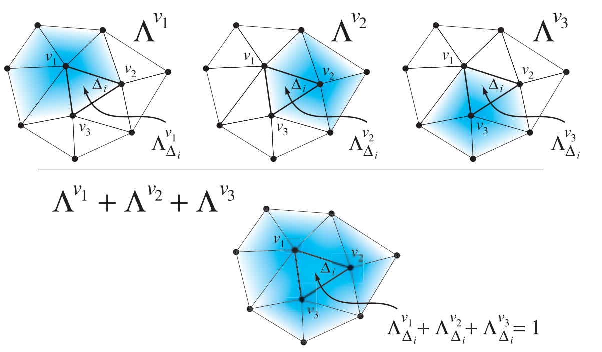

Λ^{v_1}
Λ^{v_2}
Λ^{v_3}
$\Lambda^{v_1}_{\Delta_i}$
$\Lambda^{v_2}_{\Delta_i}$
$\Lambda^{v_3}_{\Delta_i}$
$\Lambda^{v_1} + \Lambda^{v_2} + \Lambda^{v_3}$
$\Lambda^{v_1}_{\Delta_i} + \Lambda^{v_2}_{\Delta_i} + \Lambda^{v_3}_{\Delta_i} = 1$

Figure 5: *The finite support of the hat functions Λ^{v_j} around vertex v_j, $j = 1,2,3$. $\Lambda^{v_j}_{\triangle_i}$ denotes the portion of Λ^{v_j} that corresponds to triangle $\triangle_i$. Functions Λ^{v_1}, Λ^{v_2} and Λ^{v_3} add up to one inside Δ_i.*

Rendering is done directly from this compact representation using commodity graphics hardware. Since this approach allows each surface point to have a unique surface reflectance property, it is ideally suited for modeling and rendering objects scanned through 3D photography. Chen et al. [2002] show how surface light field approximation combined with image compression algorithms can lead to extremely high compression ratios, up to four orders of magnitude.

One limitation of LFM is that it cannot effectively handle objects with complex geometry. Since its representation uses textures both at the triangles and at the vertices of the mesh, this method works best for medium size surface primitives – using a dense mesh unnecessarily increases the size of the textures, while using a coarse mesh leads to artifacts at the silhouettes. We extend the original LFM framework to modeling and rendering of opacity light fields and refer to the new rendering routine as *opacity light field mapping (OLFM)*. OLFM allows us to model objects with complex geometric details while retaining interactive rendering performance and high compression ratios.

Extending the LFM framework to include opacity hulls requires minor changes to the approximation algorithm, since opacity data are processed the same way as color data, but fairly significant modifications to the rendering algorithm, since the transparency layer of the opacity light field necessitates an ordering constraint. To retain the interactive performance of LFM, we choose to use a hardware-accelerated visibility ordering algorithm [Everitt 2001; Krishnan et al. 2001]. The resulting algorithm is completely hardware-accelerated, it can render complex opacity light fields at interactive frame rates, and is easy to implement.

The next section explains the approximation of opacity light fields that leads to compact representation suitable for our rendering algorithm. The following section describes the rendering algorithm itself.

6.1 Approximation of Opacity Light Fields

Approximation of opacity light fields achieves compact and accurate representations by partitioning the light field data around every vertex and building the approximations for each part independently. We refer to the opacity light field unit corresponding to each vertex as the *vertex light field*. Partitioning is computed by weighting the opacity light field function

$$f^{v_j}(r,s,\theta,\phi) = \Lambda^{v_j}(r,s)\, f(r,s,\theta,\phi) \qquad (6)$$

where Λ^{v_j} is the barycentric weight of each point in the ring of triangles relative to vertex v_j. The top row of Figure 5 shows hat functions Λ^{v_1}, Λ^{v_2}, Λ^{v_3} for three vertices v_1, v_2, v_3 of triangle $\triangle_i$. As shown at the bottom of Figure 5, these 3 functions add up to unity inside triangle $\triangle_i$. In the final step of partitioning we reparameterize each vertex light field to the local reference frame of that vertex. We use matrix factorization to construct the approximations. To this end, the vertex light field function is discretized and rearranged into the matrix

$$\mathbf{F}^{v_j} = \begin{bmatrix} f^{v_j}[r_1,s_1,\theta_1,\phi_1] & \cdots & f^{v_j}[r_1,s_1,\theta_N,\phi_N] \\ \vdots & \ddots & \vdots \\ f^{v_j}[r_M,s_M,\theta_1,\phi_1] & \cdots & f^{v_j}[r_M,s_M,\theta_N,\phi_N] \end{bmatrix}, \qquad (7)$$

where M is the total number of surface samples inside the triangle ring and N is the total number of views for each surface sample. We refer to matrix $\mathbf{F}^{v_j}$ as the *vertex light field matrix*. Each column of this matrix represents the appearance of the vertex ring under a different viewing direction. Rearranging the raw light field data into a matrix involves two-step resampling. The first step normalizes the size of each triangle ring view. The size of the normalized patch is chosen to be equal to the size of the largest view. Resampling is done using bilinear interpolation of the pixels in the original views. The second step resamples the viewing directions as shown in Figure 6. Let vectors $\mathbf{d}^{v_j}_{\triangle_i}$ be the viewing directions for the visible views of a given triangle expressed in the reference frame of vertex v_j. We start by orthographically projecting these directions onto the xy plane of the reference frame of vertex v_j. The result of this operation is a set of texture coordinates (Figure 6, left). Next we perform the Delaunay triangulation of these coordinates (Figure 6, middle) and compute the regular grid of views by blending the original triangle views using the weighting factors obtained from the triangulation (Figure 6, right).

Matrix factorization constructs approximate factorization of the form

$$\widetilde{\mathbf{F}}^{v_j} = \sum_{k=1}^{K} \mathbf{u}_k \mathbf{v}_k^T \qquad (8)$$

where $\mathbf{u}_k$ is a vectorized representation of discrete surface map $g_k^{v_j}(r,s)$ for the triangle ring of vertex v_j and $\mathbf{v}_k$ is a vectorized representation of discrete view map $h_k^{v_j}(\theta,\phi)$ corresponding this vertex.

Partitioning of light field data into vertex light fields ensures that in the resulting approximation each triangle shares its view maps with the neighboring triangles. This results in an approximation that is continuous across triangles regardless of the number of approximation terms K. Note that each triangle light field can be expressed independently as a sum of its three vertex light fields using the following equality

$$\widetilde{f}^{\triangle_i}(r,s,\theta,\phi) = \sum_{j=1}^{3} g^{v_j}_{\triangle_i}(r,s)\, h^{v_j}(\theta,\phi) \qquad (9)$$

where index j runs over the three vertices of triangle $\triangle_i$ and $g^{v_j}_{\triangle_i}(r,s)$ denotes the portion the surface map corresponding to the triangle $\triangle_i$.

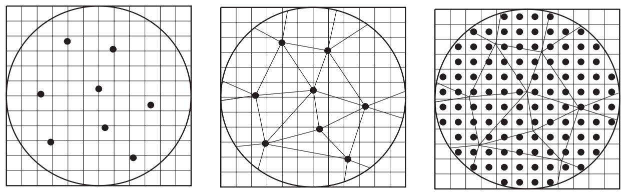


Figure 6: *Resampling of views. Projection of original views (left), Delaunay triangulation of projected views (center), uniform grid of views computed by blending the original set of views (right).*

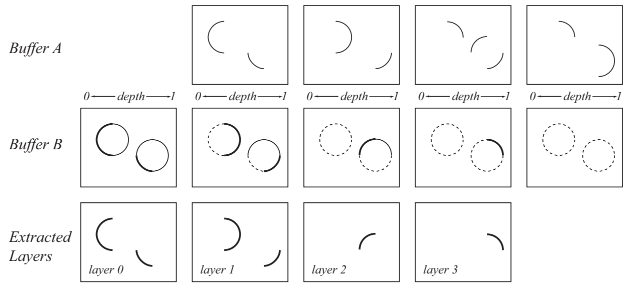


Figure 7: **Depth peeling**. *At each iteration, the algorithm extracts a layer of the scene that has fragments with depth values more than the values in Buffer A and less than the values in Buffer B. The dashed lines indicate the portions of the scene that do not pass the depth test of Buffer A and the bold lines indicate the portions of the scene that pass both depth tests.*

6.2 Rendering

Rendering of opacity light fields requires visibility ordering. The prevalent class of algorithms for visibility ordering today uses depth-sorting of primitives using BSP-partitioning of 3D space [Fuchs et al. 1980]. However, depth-sorting of triangles this way is particularly hard in the case of LFM, since it is a multi-pass algorithm and, to avoid excessive texture swapping and improve rendering efficiency, it employs tiling of the light field maps. This forces rendering of triangles in the order in which they show up in the tiled textures, making depth-sorting of the triangles impossible. (The algorithm would constantly perform costly switching of textures.)

Recently several algorithms have been proposed for computing visibility ordering with the aid of graphics hardware [Krishnan et al. 2001; Westermann and Ertl 1998]. These algorithms avoid the hassle of building complicated data structures, are often simpler to implement and offer more efficient solutions. We describe here a variant of the algorithm called *depth peeling* [Everitt 2001]. A similar fragment-level depth sorting algorithm is introduced in [Diefenbach 1996]. Subsequently, we describe the opacity light field mapping algorithm, a fully hardware-accelerated algorithm for rendering of opacity light fields using a combination of depth peeling and LFM.

Depth peeling works by extracting layers of primitives that do not relate to each other in the visibility order, starting with the layer that is closest to the camera. The algorithm uses two depth buffers, as shown in Figure 7. Buffer A, shown in the top row, indicates how much of the scene has been *peeled away* for each fragment, i.e., how much of the scene has the depth smaller than the depth in Buffer A. The depth-test function for Buffer A is set to ZTEST_GREATER. Buffer B, shown in the middle row, performs depth buffering of the portion of the scene that passes the depth test of Buffer A using the depth-test function set to ZTEST_LESS. This means that at each iteration of the algorithm, a layer of the scene gets extracted that has fragments with the depth more than the depth in Buffer A and less than the depth in Buffer B. These layers of the scene get saved in textures for later alpha-blending. The alpha-blending is done in back-to-front order, opposite to the order of peeling the layers.

Note that no depth comparison with Buffer A is done in the first pass of depth peeling, as indicated in the figure by a missing rectangle in the left most column. The algorithm stops when it reaches a layer that contains no visible primitives or that has the same set of visible primitives as the previous layer. After each iteration of the algorithm, the content of Buffer B is moved to Buffer A. In the figure, the dashed lines indicate those portions of the scene that do not pass the depth test of Buffer A and the bold lines indicate those portions of the scene that pass both depth tests. The bold lines represent the portion of the scene that corresponds to the extracted layer. The extracted layers are shown separately in the bottom row.

There are many possible ways of implementing depth peeling and the choice largely depends on the type of hardware available. In our case, Buffer A is implemented using a combination of shadow mapping and alpha testing. We set the orientation and the resolution of the shadow map to be identical to that of the camera. Buffer B is the regular z-buffer. The results of the shadow map comparison are written to the alpha channel of the fragments and used to perform the alpha test. Only those fragments that pass the alpha test get written to the framebuffer.

Since both LFM and depth peeling have been designed with hardware-acceleration in mind, it is rather simple to combine them into a fully hardware-accelerated algorithm for rendering of opacity light fields that maintains the crucial properties of its components: interactivity and ease-of-use. For a scene with depth complexity L, the algorithm requires L iterations. Each iteration starts with a rendering pass that performs the depth peeling of the current depth layer. Subsequently, the iteration continues with LFM rendering of the current layer of the opacity light field. This is accomplished by setting the depth-test function of the z-buffer to ZTEST_EQUAL. The result of each opacity light field rendering is stored in an RGBA texture T_i, $1 \leq i \leq L$. Once all L layers are processed, the textures T_i are alpha-blended together using the framebuffer in back-to-front order. See Appendix A for pseudo-code for the opacity light field mapping algorithm.

Implementation Issues

The opacity light field mapping algorithm described above requires $K+1$ rendering passes to render each depth layer – one pass to perform depth peeling of each layer and K passes to perform the LFM rendering. In our implementation, the depth peeling step has to be done in a separate rendering pass from the LFM step, since both these steps use the alpha channel of the framebuffer – depth peeling uses it to store the results of the alpha test performed on the shadow map and LFM rendering uses it to accumulate the transparency values computed during multiple rendering passes required to evaluate the opacity light field approximation.

Since ATI's Radeon 9700 has up to 32 texture lookups per rendering pass, it is safe to assume that the LFM rendering of each layer of opacity light field can be implemented in a single rendering pass.[1] Additionally, since the depth peeling algorithm is sensitive to the resolution of the depth buffer, the floating point pixel pipeline supported in the new graphics hardware would greatly improve the accuracy of the algorithm.

7 Results

To compare the quality and performance of the three algorithms we use four models that have been acquired with the image-based 3D photography system by Matusik et al. [2002]. For each model, the images were acquired with six high-resolution digital cameras from 72×6 viewpoints around the object. At each viewpoint, alpha mattes were captured using background monitors and high-quality multi-background matting techniques. The texture images were acquired from the same viewpoints under controlled, fixed illumination.

A triangular mesh of each object is computed using a volumetric technique [Curless and Levoy 1996] from three orthographic layered depth images (LDIs) of the visual hull of the object. The LDIs are computed using the method described in [Matusik et al. 2000]. Figure 10 shows the object meshes in column one, and actual photographs of the objects in column two. The companion animations show each model rendered with the various approaches discussed in this paper.

[1]Light field mapping of a 3-term light field approximation requires 17 texture lookups from 6 distinct texture sources.

Models	Triangle Count	# BSP Tree Triangles	Auxiliary Data
Angel	8,053	8,742	64 MB
Bear	8,068	9,847	73 MB
Bonsai Tree	7,852	8,884	70 MB
Teapot	2,955	3,958	26 MB

Table 1: *Size of the models used in our experiments. Auxiliary data includes cached texture coordinates, Delaunay triangulations, and visibility information. All models use 100 MB texture data after compression.*

7.1 VDTM and ULR

Table 1 shows the number of original triangles, the number of triangles in the BSP tree, and the size of the auxiliary data that is cached to accelerate rendering performance for VDTM and ULR. The original texture size for all images is 2.37 GB for all models. The simple compression scheme described in Section 4 reduces this to 110 MB for a compression ratio of 23:1. Hardware texture compression could further reduce this size by a factor of six with a slight reduction in image quality.

Figure 10 shows opacity light fields rendered with VDTM (column three) and ULR (column four) using the meshes shown in column one. The models were rendered from novel viewpoints that were not part of the original capturing process. Note that both VDTM and ULR follow epipole consistency (renderings from viewpoints that were used to capture the data correspond exactly to the original photographs.)

Both VDTM and ULR were implemented using DirectX 9.0 beta 2 on the ATI Radeon 9700 graphics card. As mentioned in Section 4, this is currently the only card that can render nine textures in one pass. Figure 8 shows the rendering performance of both algorithms for the four models. The frame rates are virtually identical, with a slight speed advantage for VDTM. This is due to the fact that ULR requires to take all visible views into account. The breadth-first search used in VDTM usually does not require considering many views during smooth motion. A fair amount of time is spent on the CPU for the computation of blending weights. For VDTM the breakdown is 25% CPU versus 75% GPU time, and for ULR it is 37% CPU versus 63% GPU time. These ratios are fairly consistent across all models. Not moving the objects avoids the computation of blending weights and increases the rendering performance to more than 30 frames per second for both methods.

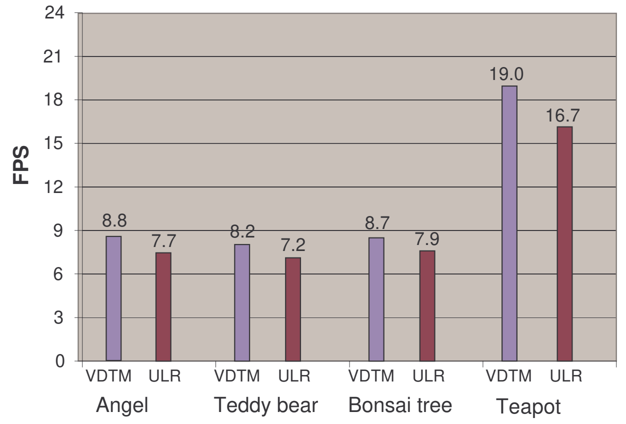


Figure 8: Rendering performance of VDTM and ULR using the ATI Radeon 9700 graphics card on a 2GHz Pentium 4 PC displayed at 1024×768 resolution with objects occupying approximately 1/3 of the window.

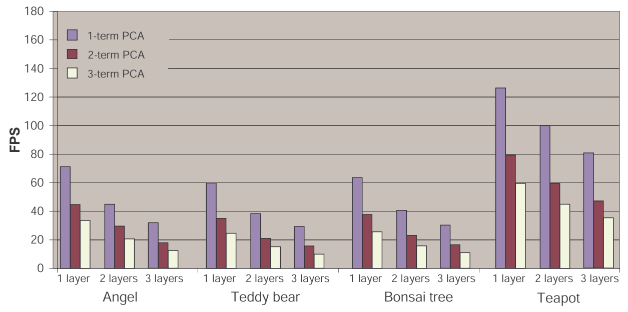


Figure 9: *Rendering performance of opacity light field mapping using Nvidia GeForce4 Ti 4600 graphics card on a 2GHz Pentium 4 PC displayed at* 1024×768 *window with objects occupying approximately 1/3 of the window.*

7.2 Opacity Light Field Mapping

We have computed the 3-term PCA approximation of opacity light fields for the four objects shown in Table 1. We have experimented with different levels of tessellation of the meshes. The teapot, which has a very simple geometry, only requires about 3,000 triangles. For the other models with complex geometry, we computed the approximations using two resolutions of the same mesh: one resolution had approximately 8,000 triangles, the other one had approximately 15,000 triangles. We noticed that the finer tessellation was reproducing the objects better, probably because it leads to smaller surface patches that can be easier approximated using matrix factorization algorithms. Coarser tessellation results in increased blurriness of the models. Column 3 of Figure 10 shows opacity light fields with the 3-term PCA approximation using the higher tessellation meshes. The screen shots were captured from the same camera location that was used to capture the photographs in column 1.

Note that OLFM – in contrast to VDTM and ULR – never uses the input images for rendering, instead it renders directly from the compressed data stored in the form of light field maps. OLFM might not produce as good an approximation to the input images as the other two methods. However, it guarantees consistent quality of the approximation for novel views, and it is faster, and it uses much less texture memory.

Unlike VDTM and ULR, OLFM offers progressive support. Figure 11 illustrates the progressive improvement in the rendering quality of OLFM as we increase the number of terms that approximate the opacity light fields. The biggest difference occurs when we add the first and the second approximation terms.

Since the depth complexity of our models is not very high, when using depth peeling, no more than three layers were ever required. Figure 9 shows the rendering performance of opacity light field mapping for different number of depth layers for the 3-term PCA approximation of the angel model. OLFM retains the interactive rendering frame rate and high compression ratios of the original LFM algorithm.

Table 2 shows the compression results obtained for the 3-term PCA approximation of opacity light fields for the models. We routinely get three orders of magnitude of compression. This is slightly less than for LFM, since opacity light fields are slightly more sensitive to compression and we use larger number of triangles. The resolution of the view maps used here was 16×16 for the first three models and 32×32 for the teapot. Currently, our software does not support higher view map resolution for meshes with that many triangles. There is very little visual degradation when compressing textures using hardware texture compression.

8 Conclusion and Future Work

We showed how to implement opacity light fields in the framework of three surface light field rendering methods: VDTM, ULR, and OLFM. The resulting algorithms are easy to implement and are fully hardware-accelerated. We found that they offer an interesting set of solutions in the quality-performance spectrum.

VDTM and ULR provide very high image quality at interactive speeds. We found that both algorithms compare favorably to the point rendering method described in [Matusik et al. 2002]. Point rendering achieves higher image quality by using many more vertices (0.5 to 1.5 million) compared to the relatively coarse triangle meshes. On the other hand, VDTM and ULR are two orders of magnitude faster and are hardware accelerated.

OLFM offers very good compression and performance. The models, when fully compressed, are of the order of several Megabytes. We can render them directly from the compressed representation at higher frame rates than VDTM and ULR. However, OLFM is sensitive to inaccuracies in geometry and calibration of data. When the geometry becomes very imprecise, the rendering of the object looks blurry. In the future, we would like to look at using relief textures as means of getting a better shape approximation.

Opacity light fields offer a trade-off between the amount of detailed geometry and the amount of view-dependent opacity. A very dense mesh that fits the object perfectly does not require any opacity information. However, storing and rendering of huge meshes is inefficient, and detailed geometry of real-life objects is very hard to get. We find that using coarse polygon meshes in combination with accurate opacity information is a very good solution in this tradeoff.

In the future, we would like to investigate interactive rendering of opacity reflectance fields – the combination of opacity hulls and surface reflectance fields [Matusik et al. 2002]. Opacity reflectance fields can render complex objects under varying illumination.

9 Acknowledgments

MERL would like to thank the following people for their help and support: Remo Ziegler, Addy Ngan, Seth Teller, and Leonard McMillan. We especially thank Michael Doggett, Joe Chien, and Jeff Royle from ATI for providing us with the latest graphics hardware and DirectX 9.0 drivers. Intel would like to thank Wei-Chao Chen, Jean-Yves Bouguet, Karl Hillesland, and Clinton Taylor for their help in this project.

References

BUEHLER, C., BOSSE, M., MCMILLAN, L., GORTLER, S., AND COHEN, M. 2001. Unstructured lumigraph rendering. In *Computer Graphics*, SIGGRAPH 2001 Proceedings, 425–432.

CHEN, S. E., AND WILLIAMS, L. 1993. View interpolation for image synthesis. In *Computer Graphics*, SIGGRAPH 93 Proceedings, 279–288.

CHEN, W.-C., BOUGUET, J.-Y., CHU, M. H., AND GRZESZCZUK, R. 2002. Light Field Mapping: Efficient Representation and Hardware Rendering of Surface Light Fields. *ACM Transactions on Graphics 21*, 3 (July), 447–456. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

CURLESS, B., AND LEVOY, M. 1996. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings of SIGGRAPH 96*, ACM SIGGRAPH / Addison Wesley, New Orleans, Louisiana, Computer Graphics Proceedings, Annual Conference Series, 303–312. ISBN 0-201-94800-1.

DEBEVEC, P., TAYLOR, C., AND MALIK, J. 1996. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Computer Graphics*, SIGGRAPH 96 Proceedings, 11–20.

DEBEVEC, P., YU, Y., AND BORSHUKOV, G. 1998. Efficient view-dependent image-based rendering with projective texture-mapping. In *Proceedings of the 9th Eurographics Workshop on Rendering*, 105–116.

DIEFENBACH, P. 1996. *Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering*. PhD thesis, Department of Computer Science, University of Pennsylvania.

EVERITT, C. 2001. Interactive Order-Independent Transparency. Nvidia's Developer Website.

FUCHS, H., KEDEM, Z. M., AND NAYLOR, B. F. 1980. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, vol. 14, 124–133.

GORTLER, S., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. 1996. The lumigraph. In *Computer Graphics*, SIGGRAPH 96 Proceedings, 43–54.

KRISHNAN, S., SILVA, C., AND WEI, B. 2001. A Hardware-Assisted Visibility-Ordering Algorithm With Applications to Volume Rendering. *Data Visualization 2001*, 233–242.

LAURENTINI, A. 1994. The visual hull concept for silhouette-based image understanding. *PAMI 16*, 2 (February), 150–162.

LENGYEL, J., PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2001. Real-time fur over arbitrary surfaces. In *Symposium on Interactive 3D Graphics*, 227–232.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. In *Computer Graphics*, SIGGRAPH 96 Proceedings, 31–42.

MATUSIK, W., BUEHLER, C., RASKAR, R., GORTLER, S., AND MCMILLAN, L. 2000. Image-based visual hulls. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, 369–374.

MATUSIK, W., PFISTER, H., NGAN, A., BEARDSLEY, P., ZIEGLER, R., AND MCMILLAN, L. 2002. Image-based 3d photography using opacity hulls. *ACM Transaction on Graphics 21*, 3 (July), 427–437. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

MCMILLAN, L., AND BISHOP, G. 1995. Plenoptic modeling: An image-based rendering system. In *Computer Graphics*, SIGGRAPH 95 Proceedings, 39–46.

MILLER, G., RUBIN, S., AND PONCELEON, D. 1998. Lazy decompression of surface light fields for precomputed global illumination. In *Proceedings of the 9th Eurographics Workshop on Rendering*, 281–292.

NISHINO, K., SATO, Y., AND IKEUCHI, K. 1999. Eigen-texture method: Appearance compression based on 3d model. In *Proc. of Computer Vision and Pattern Recognition*, 618–624.

PULLI, K., COHEN, M., DUCHAMP, T., HOPPE, H., SHAPIRO, L., AND STUETZLE, W. 1997. View-based rendering: Visualizing real objects from scanned range and color data. In *Eurographics Rendering Workshop 1997*, 23–34.

SANDER, P., GU, X., GORTLER, S., HOPPE, H., AND SNYDER, J. 2000. Silhouette clipping. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, 327–334.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *First Workshop on Applied Computational Geometry*, 124–133.

WESTERMANN, R., AND ERTL, T. 1998. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of SIGGRAPH 98*, ACM SIGGRAPH / Addison Wesley, Orlando, Florida, Computer Graphics Proceedings, Annual Conference Series, 169–178. ISBN 0-89791-999-8.

WOOD, D., AZUMA, D., ALDINGER, K., CURLESS, B., DUCHAMP, T., SALESIN, D., AND STUETZLE, W. 2000. Surface light fields for 3d photography. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, 287–296.

A Appendix: Pseudo-Code

A.1 Modified VDTM Rendering Algorithm

```
VDTM( desired view direction $d$ )
**if** $d \neq$ old_view **then**
  ComputeWeights($d$)
  old_view = $d$
  **for** each mesh triangle $t$ **do**
    combine closest views and weights from $t$'s vertices
  **end for**
**end if**
**for** each mesh triangle $t$ in back-to-front order **do**
  set the textures
  set texture coordinates and weights as shader streams
  render $t$ using programmable graphics hardware
**end for**

ComputeWeights($d$)
**for** each mesh vertex $v$ **do**
  project $d$ onto the Delaunay triangulation of $v$
  render $t$ using programmable graphics hardware
**end for**
```

A.2 Modified ULR Rendering Algorithm

```
ULR( desired view direction $d$ )
  *this method is identical to the VDTM method*
ComputeWeights($d$)
**for** each mesh vertex $v$ **do**
  compute $d$ dot $C_i$, i.e., $\cos\theta_i$ for every visible view $C_i$
  pick views with largest $\cos\theta_i$ as closest views
  compute their blending weights according to equations 1 and 4
**end for**
```

A.3 Opacity Light Field Mapping

This is the pseudo-code for the opacity light field mapping algorithm. Note that L is the number of depth layers of the scene.

```
{Render layers front to back}
**for** $i = 1, \ldots, L$ **do**
  Set depth test of z-buffer to ZTEST_LESS
  **if** $i$==1 **then**
    renderGeometry()
  **else**
    computeLayer()
  **end if**
  shadow map $\leftarrow$ z-buffer
  Clear z-buffer
  Set depth test of z-buffer to ZTEST_EQUAL
  LFMRender() to texture $T_i$
**end for**
Clear frame buffer
{Composite layers back to front}
**for** $i = L, \ldots, 1$ **do**
  Alpha blend $T_i$ with framebuffer
**end for**
```

The renderGeometry() function simply creates a depth buffer by rendering the geometry. Color writes may be turned off. The function computeLayer() is the same as renderGeometry(), but uses the shadow buffer to eliminate fragments from previous layers. A fragment is killed by setting its alpha to zero and setting the alpha test to pass on GREATER_THAN_ZERO. The fragment shader for computeLayer() follows.

```
**for** every pixel **do**
  {Begin shadow map test}
  **if** fragment depth greater than shadow map depth **then**
    $res \leftarrow 1$
  **else**
    $res \leftarrow 0$
  **end if**
  {End shadow map test}
  $alpha \leftarrow res$
**end for**
```

Models	Triangle Count	Raw Light Field Data	Light Field Maps (3-term)	Compression of Light Field Maps		
				VQ	DXT3	VQ+DXT3
Angel	13,651	3.3GB	63.0 MB (52:1)	10.5 MB (314:1)	15.7 MB (210:1)	2.6 MB (1256:1)
Teddy Bear	16,136	6.4GB	108.0 MB (59:1)	18.0 MB (356:1)	27.0 MB (237:1)	4.5 MB (1424:1)
Bonsai Tree	15,496	6.3GB	98.0 MB (64:1)	16.3 MB (387:1)	24.5 MB (257:1)	4.1 MB (1548:1)
Teapot	2,999	5.4GB	34.6 MB (156:1)	4.3 MB (1256:1)	8.6 MB (628:1)	1.1 MB (5024:1)

Table 2: *The size and compression ratio of opacity light field data obtained through the light field map approximation and additional compression of the surface light field maps.*

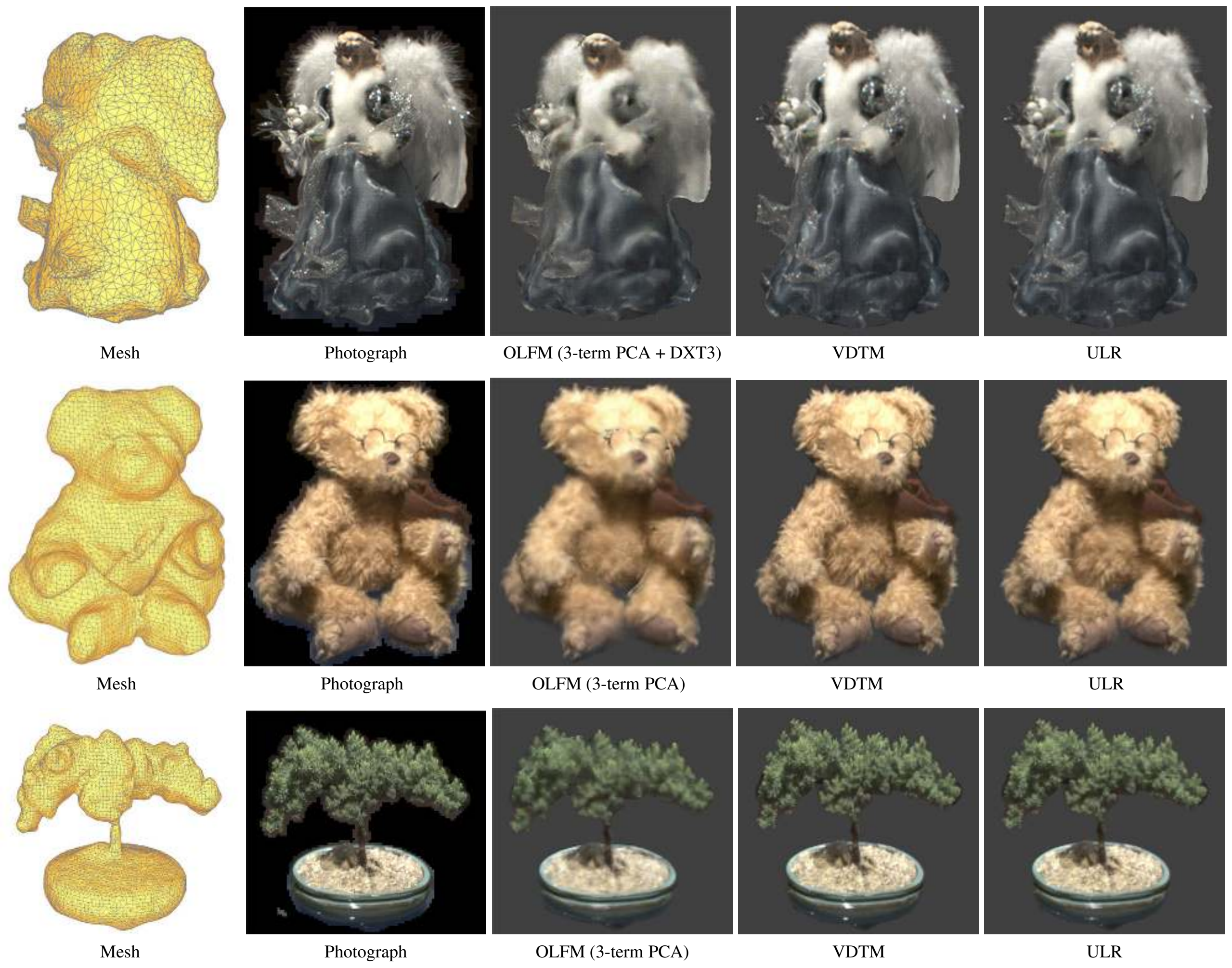


Figure 10: *Comparison between three methods of rendering opacity light fields.*

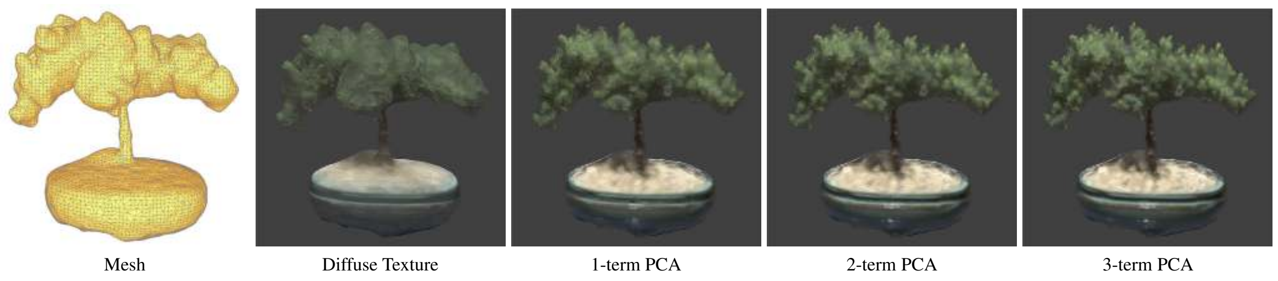


Figure 11: *Improvement in rendering quality for opacity light field mapping with increased number of approximation terms.*