# Layered Depth Images

Jonathan Shade Steven Gortler* Li-wei He† Richard Szeliski‡

*University of Washington* **Harvard University* *†Stanford University* *‡Microsoft Research*

## Abstract

In this paper we present a set of efficient image based rendering methods capable of rendering multiple frames per second on a PC. The first method warps Sprites with Depth representing smooth surfaces without the gaps found in other techniques. A second method for more general scenes performs warping from an intermediate representation called a Layered Depth Image (LDI). An LDI is a view of the scene from a single input camera view, but with multiple pixels along each line of sight. The size of the representation grows only linearly with the observed depth complexity in the scene. Moreover, because the LDI data are represented in a single image coordinate system, McMillan's warp ordering algorithm can be successfully adapted. As a result, pixels are drawn in the output image in back-to-front order. No z-buffer is required, so alpha-compositing can be done efficiently without depth sorting. This makes splatting an efficient solution to the resampling problem.

## 1 Introduction

Image based rendering (IBR) techniques have been proposed as an efficient way of generating novel views of real and synthetic objects. With traditional rendering techniques, the time required to render an image increases with the geometric complexity of the scene. The rendering time also grows as the requested shading computations (such as those requiring global illumination solutions) become more ambitious.

The most familiar IBR method is texture mapping. An image is remapped onto a surface residing in a three-dimensional scene. Traditional texture mapping exhibits two serious limitations. First, the pixelization of the texture map and that of the final image may be vastly different. The aliasing of the classic infinite checkerboard floor is a clear illustration of the problems this mismatch can create. Secondly, texture mapping speed is still limited by the surface the texture is applied to. Thus it would be very difficult to create a texture mapped tree containing thousands of leaves that exhibits appropriate parallax as the viewpoint changes.

Two extensions of the texture mapping model have recently been presented in the computer graphics literature that address these two difficulties. The first is a generalization of *sprites*. Once a complex scene is rendered from a particular point of view, the image that would be created from a nearby point of view will likely be similar. In this case, the original 2D image, or *sprite*, can be slightly altered by a 2D affine or projective transformation to approximate the view from the new camera position [31, 27, 15].

The sprite approximation's fidelity to the correct new view is highly dependent on the geometry being represented. In particular, the errors increase with the amount of depth variation in the real part of the scene captured by the sprite. The amount of virtual camera motion away from the point of view of sprite creation also increases the error. Errors decrease with the distance of the geometry from the virtual camera.

The second recent extension is to add depth information to an image to produce a *depth image* and to then use the optical flow that would be induced by a camera shift to warp the scene into an approximation of the new view [2, 22].

Each of these methods has its limitations. Simple sprite warping cannot produce the *parallax* induced when parts of the scenes have sizable differences in distance from the camera. Flowing a depth image pixel by pixel, on the other hand, can provide proper parallax but will result in gaps in the image either due to visibility changes when some portion of the scene become unoccluded, or when a surface is magnified in the new view.

Some solutions have been proposed to the latter problem. Laveau and Faugeras suggest performing a backwards mapping from the output sample location to the input image [14]. This is an expensive operation that requires some amount of searching in the input image. Another possible solution is to think of the input image as a mesh of micro-polygons, and to scan-convert these polygons in the output image. This is an expensive operation, as it requires a polygon scan-convert setup for each input pixel [18], an operation we would prefer to avoid especially in the absence of specialized rendering hardware. Alternatively one could use multiple input images from different viewpoints. However, if one uses $n$ input images, one effectively multiplies the size of the scene description by $n$, and the rendering cost increases accordingly.

This paper introduces two new extensions to overcome both of these limitations. The first extension is primarily applicable to smoothly varying surfaces, while the second is useful primarily for very complex geometries. Each method provides efficient image based rendering capable of producing multiple frames per second on a PC.

In the case of sprites representing smoothly varying surfaces, we introduce an algorithm for rendering *Sprites with Depth*. The algorithm first forward maps (i.e., warps) the depth values themselves and then uses this information to add parallax corrections to a standard sprite renderer.

For more complex geometries, we introduce the *Layered Depth Image*, or LDI, that contains potentially multiple depth pixels at each discrete location in the image. Instead of a 2D array of depth pixels (a pixel with associated depth information), we store a 2D array of layered depth pixels. A layered depth pixel stores a set of depth pixels along one line of sight sorted in front to back order. The front element in the layered depth pixel samples the first surface seen along that line of sight; the next pixel in the layered depth pixel samples the next surface seen along that line of sight, etc. When rendering from an LDI, the requested view can move away from the original LDI view and expose surfaces that were not visible in the first layer. The previously occluded regions may still be rendered from data stored in some later layer of a layered depth pixel.

There are many advantages to this representation. The size of the

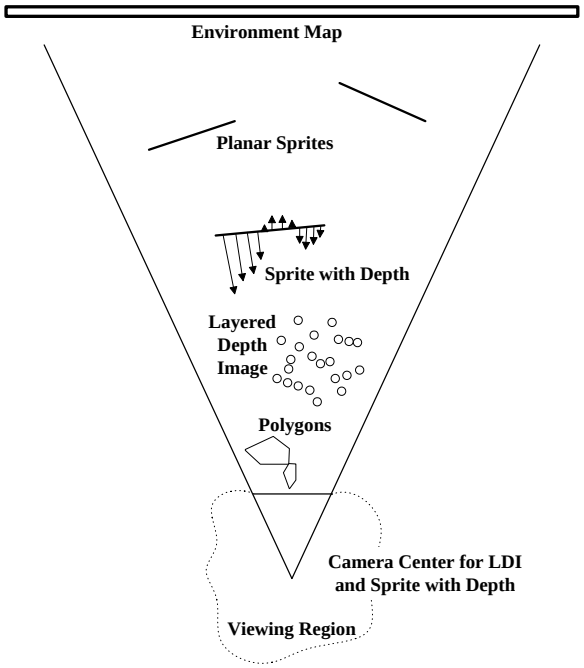


**Figure 1** Different image based primitives can serve well depending on distance from the camera

representation grows linearly only with the depth complexity of the image. Moreover, because the LDI data are represented in a single image coordinate system, McMillan's ordering algorithm [21] can be successfully applied. As a result, pixels are drawn in the output image in back to front order allowing proper alpha blending without depth sorting. No z-buffer is required, so alpha-compositing can be done efficiently without explicit depth sorting. This makes splatting an efficient solution to the reconstruction problem.

Sprites with Depth and Layered Depth Images provide us with two new image based primitives that can be used in combination with traditional ones. Figure 1 depicts five types of primitives we may wish to use. The camera at the center of the frustum indicates where the image based primitives were generated from. The viewing volume indicates the range one wishes to allow the camera to move while still re-using these image based primitives.

The choice of which type of image-based or geometric primitive to use for each scene element is a function of its distance, its internal depth variation relative to the camera, as well as its internal geometric complexity. For scene elements at a great distance from the camera one might simply generate an environment map. The environment map is invariant to translation and simply translates as a whole on the screen based on the rotation of the camera. At a somewhat closer range, and for geometrically planar elements, traditional planar sprites (or *image caches*) may be used [31, 27]. The assumption here is that although the part of the scene depicted in the sprite may display some parallax relative to the background environment map and other sprites, it will not need to depict any parallax within the sprite itself. Yet closer to the camera, for elements with smoothly varying depth, Sprites with Depth are capable of displaying internal parallax but cannot deal with disocclusions due to image flow that may arise in more complex geometric scene elements. Layered Depth Images deal with both parallax and disocclusions and are thus useful for objects near the camera that also contain complex geometries that will exhibit considerable parallax. Finally, traditional polygon rendering may need to be used for immediate foreground objects.

In the sections that follow, we will concentrate on describing the data structures and algorithms for representing and rapidly rendering Sprites with Depth and Layered Depth Images.

## 2 Previous Work

Over the past few years, there have been many papers on image based rendering. In [17], Levoy and Whitted discuss rendering point data. Chen and Williams presented the idea of rendering from images [2]. Laveau and Faugeras discuss IBR using a backwards map [14]. McMillan and Bishop discuss IBR using cylindrical views [22]. Seitz and Dyer describe a system that allows a user to correctly model view transforms in a user controlled image morphing system [29]. In a slightly different direction, Levoy and Hanrahan [16] and Gortler *et al.* [7] describe IBR methods using a large number of input images to sample the high dimensional radiance function.

Max uses a representation similar to an LDI [19], but for a purpose quite different than ours; his purpose is high quality anti-aliasing, while our goal is efficiency. Max reports his rendering time as 5 minutes per frame while our goal is multiple frames per second. Max warps from *n* input LDIs with different camera information; the multiple depth layers serve to represent the high depth complexity of trees. We warp from a single LDI, so that the warping can be done most efficiently. For output, Max warps to an LDI. This is done so that, in conjunction with an A-buffer, high quality, but somewhat expensive, anti-aliasing of the output picture can be performed.

Mark *et al.*[18] and Darsa *et al.*[4] create triangulated depth maps from input images with per-pixel depth. Darsa concentrates on limiting the number of triangles by looking for depth coherence across regions of pixels. This triangle mesh is then rendered traditionally taking advantage of graphics hardware pipelines. Mark *et al.*describe the use of multiple input images as well. In this aspect of their work, specific triangles are given lowered priority if there is a large discontinuity in depth across neighboring pixels. In this case, if another image fills in the same area with a triangle of higher priority, it is used instead. This helps deal with disocclusions.

Shade *et al.*[31] and Shaufler *et al.*[27] render complex portions of a scene such as a tree onto alpha matted billboard-like sprites and then reuse them as textures in subsequent frames. Lengyel and Snyder [15] extend this work by warping sprites by a best fit affine transformation based on a set of sample points in the underlying

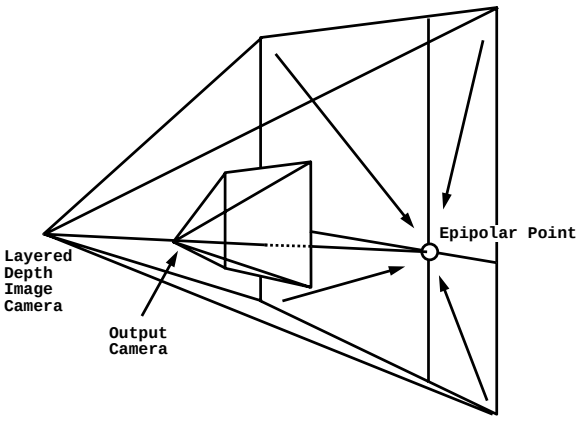


**Figure 2** Back to front output ordering

3D model. These affine transforms are allowed to vary in time as the position and/or color of the sample points change. Hardware considerations for such system are discussed in [32].

Horry *et al.* [10] describe a very simple sprite-like system in which a user interactively indicates planes in an image that represent areas in a given image. Thus, from a single input image and some user supplied information, they can warp an image and provide approximate three dimensional cues about the scene.

The system presented here relies heavily on McMillan's ordering algorithm [21, 20, 22]. Using input and output camera information, a warping order is computed such that pixels that map to the same location in the output image are guaranteed to arrive in back to front order.

In McMillan's work, the depth order is computed by first finding the projection of the output camera's location in the input camera's image plane, that is, the intersection of the line joining the two camera locations with the input camera's image plane. The line joining the two camera locations is called the epipolar line, and the intersection with the image plane is called an epipolar point [6] (see Figure 1). The input image is then split horizontally and vertically at the epipolar point, generally creating 4 image quadrants. (If the epipolar point lies off the image plane, we may have only 2 or 1 regions.) The pixels in each of the quadrants are processed in a different order. Depending on whether the output camera is in front of or behind the input camera, the pixels in each quadrant are processed either inward towards the epipolar point or outwards away from it. In other words, one of the quadrants is processed left to right, top to bottom, another is processed left to right, bottom to top, etc. McMillan discusses in detail the various special cases that arise and proves that this ordering is guaranteed to produce depth ordered output [20].

When warping from an LDI, there is effectively only one input camera view. Therefore one can use the ordering algorithm to order the layered depth pixels visited. Within each layered depth pixel, the layers are processed in back to front order. The formal proof of [20] applies, and the ordering algorithm is guaranteed to work.

## 3 Rendering Sprites

Sprites are texture maps or images with alphas (transparent pixels) rendered onto planar surfaces. They can be used either for locally caching the results of slower rendering and then generating new views by warping [31, 27, 32, 15], or they can be used directly as drawing primitives (as in video games).

The texture map associated with a sprite can be computed by simply choosing a 3D viewing matrix and projecting some portion of the scene onto the image plane. In practice, a view associated with the current or expected viewpoint is a good choice. A 3D plane equation can also be computed for the sprite, e.g., by fitting a 3D plane to the z-buffer values associated with the sprite pixels. Below, we derive the equations for the 2D perspective mapping between a sprite and its novel view. This is useful both for implementing a backward mapping algorithm, and lays the foundation for our Sprites with Depth rendering algorithm.

A sprite consists of an alpha-matted image $I_1(x_1, y_1)$, a $4 \times 4$ camera matrix $C_1$ which maps from 3D world coordinates $(X, Y, Z, 1)$ into the sprite's coordinates $(x_1, y_1, z_1, 1)$,

$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 z_1 \\ w_1 \end{bmatrix} = C_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \tag{1}$$

($z_1$ is the z-buffer value), and a plane equation. This plane equation can either be specified in world coordinates, $AX + BY + CZ + D = 0$, or it can be specified in the sprite's coordinate system, $ax_1 + by_1 + cz_1 + d = 0$. In the former case, we can form a new camera matrix $\hat{C}_1$ by replacing the third row of $C_1$ with the row $[A\ B\ C\ D]$, while in the latter, we can compute $\hat{C}_1 = PC_1$, where

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & c & d \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(note that $[A\ B\ C\ D] = [a\ b\ c\ d]C_1$).

In either case, we can write the modified projection equation as

$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 d_1 \\ w_1 \end{bmatrix} = \hat{C}_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \tag{2}$$

where $d_1 = 0$ for pixels on the plane. For pixels off the plane, $d_1$ is the scaled perpendicular distance to the plane (the scale factor is 1 if $A^2 + B^2 + C^2 = 1$) divided by the pixel to camera distance $w_1$.

Given such a sprite, how do we compute the 2D transformation associated with a novel view $\hat{C}_2$? The mapping between pixels $(x_1, y_1, d_1, 1)$ in the sprite and pixels $(w_2 x_2, w_2 y_2, w_2 d_2, w_2)$ in the output camera's image is given by the transfer matrix $T_{1,2} = \hat{C}_2 \cdot \hat{C}_1^{-1}$.

For a flat sprite ($d_1 = 0$), the transfer equation can be written as

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{3}$$

where $H_{1,2}$ is the 2D planar perspective transformation (*homography*) obtained by dropping the third row and column of $T_{1,2}$. The coordinates $(x_2, y_2)$ obtained after dividing out $w_2$ index a pixel address in the output camera's image. Efficient backward mapping techniques exist for performing the 2D perspective warp [8, 35], or texture mapping hardware can be used.

### 3.1 Sprites with Depth

The descriptive power (realism) of sprites can be greatly enhanced by adding an out-of-plane displacement component $d_1$ at each pixel in the sprite.[1] Unfortunately, such a representation can no longer be rendered directly using a backward mapping algorithm.

Using the same notation as before, we see that the transfer equation is now

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + d_1 e_{1,2}, \tag{4}$$

where $e_{1,2}$ is called *epipole* [6, 26, 12], and is obtained from the third column of $T_{1,2}$.

Equation (4) can be used to *forward map* pixels from a sprite to a new view. Unfortunately, this entails the usual problems associated with forward mapping, e.g., the necessity to fill gaps or to use larger

[1] The $d_1$ values can be stored as a separate image, say as 8-bit signed depths. The full precision of a traditional z-buffer is not required, since these depths are used only to compute local parallax, and not to perform z-buffer merging of primitives. Furthermore, the $d_1$ image could be stored at a lower resolution than the color image, if desired.

splatting kernels, and the difficulty in achieving proper resampling. Notice, however, that Equation (4) could be used to perform a backward mapping step by interchanging the 1 and 2 indices, if only we knew the displacements $d_2$ in the output camera's coordinate frame.

A solution to this problem is to first *forward map* the displacements $d_1$, and to then use Equation (4) to perform a backward mapping step with the new (view-based) displacements. While this may at first appear to be no faster or more accurate than simply forward warping the color values, it does have some significant advantages.

First, small errors in displacement map warping will not be as evident as errors in the sprite image warping, at least if the displacement map is smoothly varying (in practice, the shape of a simple surface often varies more smoothly than its photometry). If bilinear or higher order filtering is used in the final color (backward) resampling, this two-stage warping will have much lower errors than forward mapping the colors directly with an inaccurate forward map. We can therefore use a quick single-pixel splat algorithm followed by a quick hole filling, or alternatively, use a simple $2 \times 2$ splat.

The second main advantage is that we can design the forward warping step to have a simpler form by factoring out the planar perspective warp. Notice that we can rewrite Equation (4) as

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix}, \tag{5}$$

with

$$\begin{bmatrix} w_3 x_3 \\ w_3 y_3 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + d_1 e^*_{1,2}, \tag{6}$$

where $e^*_{1,2} = H^{-1}_{1,2} e_{1,2}$. This suggests that Sprite with Depth rendering can be implemented by first shifting pixels by their local parallax, filling any resulting gaps, and then applying a global homography (planar perspective warp). This has the advantage that it can handle large changes in view (e.g., large zooms) with only a small amount of gap filling (since gaps arise only in the first step, and are due to variations in displacement).

Our novel two-step rendering algorithm thus proceeds in two stages:

1. forward map the displacement map $d_1(x_1, y_1)$, using only the parallax component given in Equation (6) to obtain $d_3(x_3, y_3)$;

2a. backward map the resulting warped displacements $d_3(x_3, y_3)$ using Equation (5) to obtain $d_2(x_2, y_2)$ (the displacements in the new camera view);

2b. backward map the original sprite colors, using both the homography $H_{2,1}$ and the new parallax $d_2$ as in Equation (4) (with the 1 and 2 indices interchanged), to obtain the image corresponding to camera $C_2$.

The last two operations can be combined into a single raster scan over the output image, avoiding the need to perspective warp $d_3$ into $d_2$. More precisely, for each output pixel $(x_2, y_2)$, we compute $(x_3, y_3)$ such that

$$\begin{bmatrix} w_3 x_3 \\ w_3 y_3 \\ w_3 \end{bmatrix} = H_{2,1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \tag{7}$$

to compute where to look up the displacement $d_3(x_3, y_3)$, and form the final address of the source sprite pixel using

$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 \end{bmatrix} = \begin{bmatrix} w_3 x_3 \\ w_3 y_3 \\ w_3 \end{bmatrix} + d_3(x_3, y_3) e_{2,1}. \tag{8}$$

We can obtain a quicker, but less accurate, algorithm by omitting the first step, i.e., the pure parallax warp from $d_1$ to $d_3$. If we assume the depth at a pixel before and after the warp will not change significantly, we can use $d_1$ instead of $d_3$ in Equation (8). This still gives a useful illusion of 3-D parallax, but is only valid for a much smaller range of viewing motions (see Figure 3).

Another variant on this algorithm, which uses somewhat more storage but fewer computations, is to compute a 2-D displacement field in the first pass, $u_3(x_3, y_3) = x_1 - x_3$, $v_3(x_3, y_3) = y_1 - y_3$, where $(x_3, y_3)$ is computed using the pure parallax transform in Equation (6). In the second pass, the final pixel address in the sprite is computed using

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + \begin{bmatrix} u_3(x_3, y_3) \\ v_3(x_3, y_3) \end{bmatrix}, \tag{9}$$

where this time $(x_3, y_3)$ is computed using the transform given in Equation (7).

We can make the pure parallax transformation (6) faster by avoiding the per-pixel division required after adding homogeneous coordinates. One way to do this is to approximate the parallax transformation by first moving the epipole to infinity (setting its third component to 0). This is equivalent to having an *affine* parallax component (all points move in the same direction, instead of towards a common vanishing point). In practice, we find that this still provides a very compelling illusion of 3D shape.

Figure 3 shows some of the steps in our two-pass warping algorithm. Figures 3a and 3f show the original sprite (color) image and the depth map. Figure 3b shows the sprite warped with no parallax. Figures 3g, 3h, and 3i shows the depth map forward warped with only pure parallax, only the perspective projection, and both. Figure 3c shows the backward warp using the incorrect depth map $d_1$ (note how dark "background" colors are mapped onto the "bump"), whereas Figure 3d shows the backward warp using the correct depth map $d_3$. The white pixels near the right hand edge are a result of using only a single step of gap filling. Using three steps results in the better quality image shown in Figure 3e. Gaps also do not appear for a less quickly slanting $d$ maps, such as the pyramid shown in Figure 3j.

The rendering times for the $256 \times 256$ image shown in Figure 3 on a 300 MHz Pentium II are as follows. Using bilinear pixel sampling, the frame rates are 30 Hz for no z-parallax, 21 Hz for "crude" one-pass warping (no forward warping of $d_1$ values), and 16 Hz for two-pass warping. Using nearest-neighbor resampling, the frame rates go up to 47 Hz, 24 Hz, and 20 Hz, respectively.

### 3.2 Recovering sprites from image sequences

While sprites and sprites with depth can be generated using computer graphics techniques, they can also be extracted from image sequences using computer vision techniques. To do this, we use a layered motion estimation algorithm [33, 1], which simultaneously segments the sequence into coherently moving regions, and computes a parametric motion estimate (planar perspective transformation) for each layer. To convert the recovered layers into sprites, we need to determine the plane equation associated with each region. We do this by tracking features from frame to frame and applying

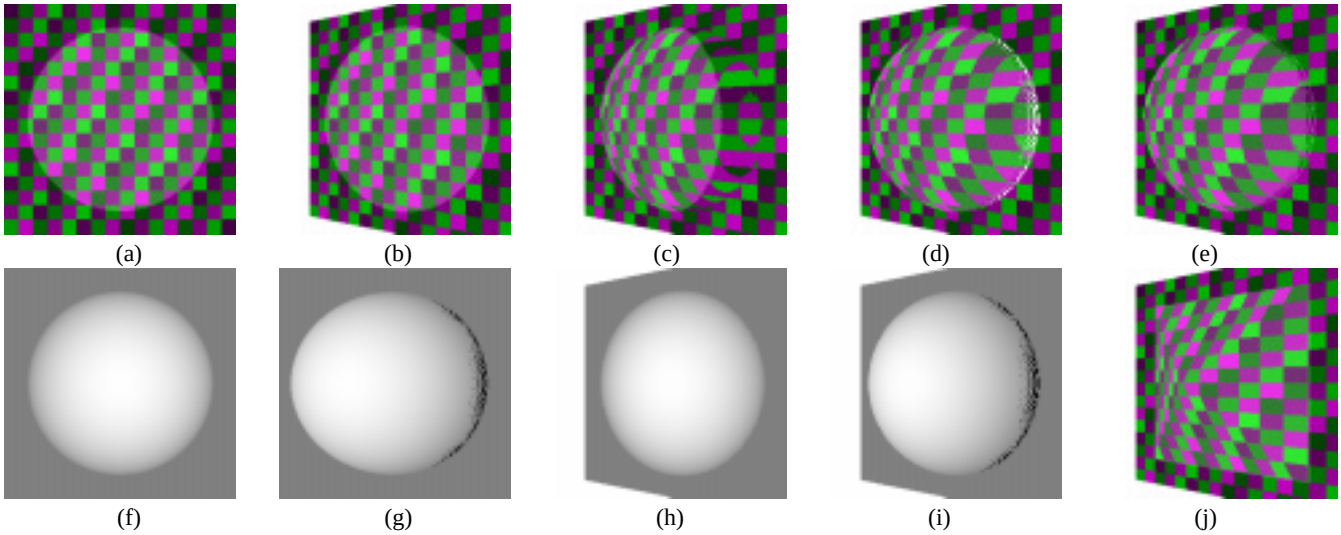


**Figure 3** Plane with bump rendering example: (a) input color (sprite) image $I_1(x_1, y_1)$; (b) sprite warped by homography only (no parallax); (c) sprite warped by homography and crude parallax ($d_1$); (d) sprite warped by homography and true parallax ($d_2$); (e) with gap fill width set to 3; (f) input depth map $d_1(x_1, y_1)$; (g) pure parallax warped depth map $d_3(x_3, y_3)$; (h) forward warped depth map $d_2(x_2, y_2)$; (i) forward warped depth map without parallax correction; (j) sprite with "pyramid" depth map.


**Figure 4** Results of sprite extraction from image sequence: (a) third of five images; (b) initial segmentation into six layers; (c) recovered depth map; (d) the five layer sprites; (e) residual depth image for fifth layer; (f) re-synthesized third image (note extended field of view); (g) novel view without residual depth; (h) novel view with residual depth (note the "rounding" of the people).

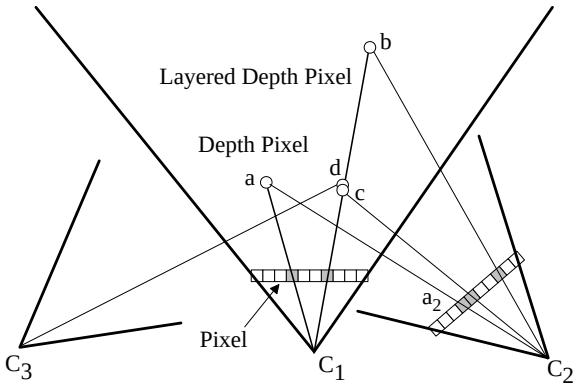


**Figure 5** Layered Depth Image

a standard structure from motion algorithm to recover the camera parameters (viewing matrices) for each frame [6]. Tracking several points on each sprite enables us to reconstruct their 3D positions, and hence to estimate their 3D plane equations [1]. Once the sprite pixel assignment have been recovered, we run a traditional stereo algorithm to recover the out-of-plane displacements.

The results of applying the layered motion estimation algorithm to the first five images from a 40-image stereo dataset[2] are shown in Figure 4. Figure 4(a) shows the middle input image, Figure 4(b) shows the initial pixel assignment to layers, Figure 4(c) shows the recovered depth map, and Figure 4(e) shows the residual depth map for layer 5. Figure 4(d) shows the recovered sprites. Figure 4(f) shows the middle image re-synthesized from these sprites, while Figures 4(g–h) show the same sprite collection seen from a novel viewpoint (well outside the range of the original views), both with and without residual depth-based correction (parallax). The gaps visible in Figures 4(c) and 4(f) lie *outside* the area corresponding to the middle image, where the appropriate parts of the background sprites could not be seen.

## 4 Layered Depth Images

While the use of sprites and Sprites with Depth provides a fast means to warp planar or smoothly varying surfaces, more general scenes require the ability to handle more general disocclusions and large amounts of parallax as the viewpoint moves. These needs have led to the development of Layered Depth Images (LDI).

Like a sprite with depth, pixels contain depth values along with their colors (i.e., a *depth pixel*). In addition, a Layered Depth Image (Figure 5) contains potentially multiple depth pixels per pixel location. The farther depth pixels, which are occluded from the LDI center, will act to fill in the disocclusions that occur as the viewpoint moves away from the center.

The structure of an LDI is summarized by the following conceptual representation:

```
DepthPixel =
    ColorRGBA: 32 bit integer
    Z: 20 bit integer
    SplatIndex: 11 bit integer

LayeredDepthPixel =
    NumLayers: integer
    Layers[0..numlayers-1]: array of DepthPixel

LayeredDepthImage =
    Camera: camera
    Pixels[0..xres-1,0..yres-1]: array of LayeredDepthPixel
```

The layered depth image contains camera information plus an array of size *xres* by *yres* layered depth pixels. In addition to image data, each layered depth pixel has an integer indicating how many valid depth pixels are contained in that pixel. The data contained in the depth pixel includes the color, the depth of the object seen at that pixel, plus an index into a table that will be used to calculate a splat size for reconstruction. This index is composed from a combination of the normal of the object seen and the distance from the LDI camera.

In practice, we implement Layered Depth Images in two ways. When creating layered depth images, it is important to be able to efficiently insert and delete layered depth pixels, so the *Layers* array in the *LayeredDepthPixel* structure is implemented as a linked list. When rendering, it is important to maintain spatial locality of depth pixels in order to most effectively take advantage of the cache in the CPU [13]. In Section 5.1 we discuss the compact render-time version of layered depth images.

There are a variety of ways to generate an LDI. Given a synthetic scene, we could use multiple images from nearby points of view for which depth information is available at each pixel. This information can be gathered from a standard ray tracer that returns depth per pixel or from a scan conversion and z-buffer algorithm where the z-buffer is also returned. Alternatively, we could use a ray tracer to sample an environment in a less regular way and then store computed ray intersections in the LDI structure. Given multiple real images, we can turn to computer vision techniques that can infer pixel correspondence and thus deduce depth values per pixel. We will demonstrate results from each of these three methods.

### 4.1 LDIs from Multiple Depth Images

We can construct an LDI by warping *n* depth images into a common camera view. For example the depth images $C_2$ and $C_3$ in Figure 5 can be warped to the camera frame defined by the LDI ($C_1$ in figure 5). [3] If, during the warp from the input camera to the LDI camera, two or more pixels map to the same layered depth pixel, their Z values are compared. If the Z values differ by more than a preset epsilon, a new layer is added to that layered depth pixel for each distinct Z value (i.e., *NumLayers* is incremented and a new depth pixel is added), otherwise (e.g., depth pixels *c* and *d* in figure 5), the values are averaged resulting in a single depth pixel. This preprocessing is similar to the rendering described by Max [19]. This construction of the layered depth image is effectively decoupled from the final rendering of images from desired viewpoints. Thus, the LDI construction does not need to run at multiple frames per second to allow interactive camera motion.

### 4.2 LDIs from a Modified Ray Tracer

By construction, a Layered Depth Image reconstructs images of a scene well from the center of projection of the LDI (we simply display the nearest depth pixels). The quality of the reconstruction from another viewpoint will depend on how closely the distribution of depth pixels in the LDI, when warped to the new viewpoint, corresponds to the pixel density in the new image. Two common events that occur are: (1) disocclusions as the viewpoint changes,

[2] Courtesy of Dayton Taylor.

[3] Any arbitrary single coordinate system can be specified here. However, we have found it best to use one of the original camera coordinate systems. This results in fewer pixels needing to be resampled twice; once in the LDI construction, and once in the rendering process.

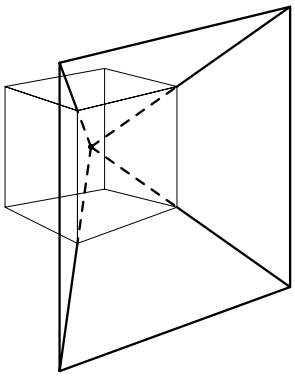

**Figure 6** An LDI consists of the 90 degree frustum exiting one side of a cube. The cube represents the region of interest in which the viewer will be able to move.

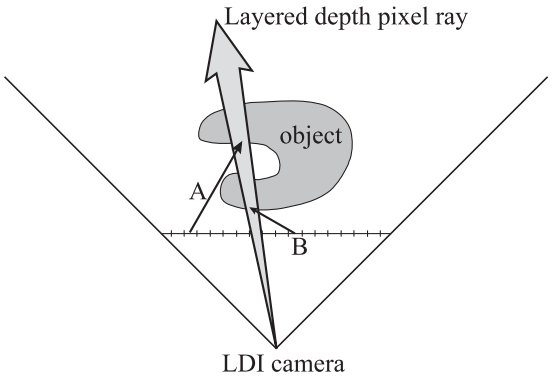


**Figure 7** Intersections from sampling rays A and B are added to the same layered depth pixel.

and (2) surfaces that grow in terms of screen space. For example, when a surface is edge on to the LDI, it covers no area. Later, it may face the new viewpoint and thus cover some screen space.

When using a ray tracer, we have the freedom to sample the scene with any distribution of rays we desire. We could simply allow the rays emanating from the center of the LDI to pierce surfaces, recording each hit along the way (up to some maximum). This would solve the disocclusion problem but would not effectively sample surfaces edge on to the LDI.

What set of rays should we trace to sample the scene, to best approximate the distribution of rays from all possible viewpoints we are interested in? For simplicity, we have chosen to use a cubical region of empty space surrounding the LDI center to represent the region that the viewer is able to move in. Each face of the viewing cube defines a 90 degree frustum which we will use to define a single LDI (Figure 6). The six faces of the viewing cube thus cover all of space. For the following discussion we will refer to a single LDI.

Each ray in free space has four coordinates, two for position and two for direction. Since all rays of interest intersect the cube faces, we will choose the outward intersection to parameterize the position of the ray. Direction is parameterized by two angles.

Given no *a priori* knowledge of the geometry in the scene, we assume that every ray intersection the cube is equally important. To achieve a uniform density of rays we sample the positional coordinates uniformly. A uniform distribution over the hemisphere of directions requires that the probability of choosing a direction is proportional to the *projected* area in that direction. Thus, the direction is weighted by the cosine of the angle off the normal to the cube face.

Choosing a cosine weighted direction over a hemisphere can be accomplished by uniformly sampling the unit disk formed by the base of the hemisphere to get two coordinates of the ray direction, say $x$ and $y$ if the z-axis is normal to the disk. The third coordinate is chosen to give a unit length ($z = \sqrt{1 - x^2 - y^2}$). We make the selection within the disk by first selecting a point in the unit square, then applying a measure preserving mapping [24] that maps the unit square to the unit disk.

Given this desired distribution of rays, there are a variety of ways to perform the sampling:

**Uniform**. A straightforward stochastic method would take as input the number of rays to cast. Then, for each ray it would choose an origin on the cube face and a direction from the cosine distribution and cast the ray into the scene. There are two problems with this simple scheme. First, such *white noise* distributions tend to form unwanted clumps. Second, since there is no coherence between rays, complex scenes require considerable memory thrashing since rays will access the database in a random way [25]. The model of the chestnut tree seen in the color images was too complex to sample with a pure stochastic method on a machine with 320MB of memory.

**Stratified Stochastic**. To improve the coherence and distribution of rays, we employ a stratified scheme. In this method, we divide the 4D space of rays uniformly into a grid of $N \times N \times N \times N$ strata. For each stratum, we cast $M$ rays. Enough coherence exists within a stratum that swapping of the data set is alleviated. Typical values for $N$ and $M$ are 32 and 16, generating approximately 16 million rays per cube face.

Once a ray is chosen, we cast it into the scene. If it hits an object, and that object lies in the LDI's frustum, we reproject the intersection into the LDI, as depicted in Figure 7, to determine which layered depth pixel should receive the sample. If the new sample is within an epsilon tolerance in depth of an existing depth pixel, the color of the new sample is averaged with the existing depth pixel. Otherwise, the color, normal, and distance to the sample create a new depth pixel that is inserted into the Layered Depth Pixel.

### 4.3 LDIs from Real Images

The dinosaur model in Figure 13 is constructed from 21 photographs of the object undergoing a 360 degree rotation on a computer-controlled calibrated turntable. An adaptation of Seitz and Dyer's voxel coloring algorithm [30] is used to obtain the LDI representation directly from the input images. The regular voxelization of Seitz and Dyer is replaced by a view-centered voxelization similar to the LDI structure. The procedure entails moving outward on rays from the LDI camera center and projecting candidate voxels back into the input images. If all input images agree on a color, this voxel is filled as a depth pixel in the LDI structure. This approach enables straightforward construction of LDI's from images that do not contain depth per pixel.

## 5 Rendering Layered Depth Images

Our fast warping-based renderer takes as input an LDI along with its associated camera information. Given a new desired camera position, the warper uses an incremental warping algorithm to efficiently create an output image. Pixels from the LDI are splatted

into the output image using the *over* compositing operation. The size and footprint of the splat is based on an estimated size of the reprojected pixel.

### 5.1 Space Efficient Representation

When rendering, it is important to maintain the spatial locality of depth pixels to exploit the second level cache in the CPU [13]. To this end, we reorganize the depth pixels into a linear array ordered from bottom to top and left to right in screen space, and back to front along a ray. We also separate out the number of layers in each layered depth pixel from the depth pixels themselves. The layered depth pixel structure does not exist explicitly in this implementation. Instead, a double array of offsets is used to locate each depth pixel. The number of depth pixels in each scanline is accumulated into a vector of offsets to the beginning of each scanline. Within each scanline, for each pixel location, a total count of the depth pixels from the beginning of the scanline to that location is maintained. Thus to find any layered depth pixel, one simply offsets to the beginning of the scanline and then further to the first depth pixel at that location. This supports scanning in right-to-left order as well as the clipping operation discussed later.

### 5.2 Incremental Warping Computation

The incremental warping computation is similar to the one used for certain texture mapping operations [9, 28]. The geometry of this computation has been analyzed by McMillan [23], and efficient computation for the special case of orthographic input images is given in [3].

Let $C_1$ be the $4 \times 4$ matrix for the LDI camera. It is composed of an affine transformation matrix, a projection matrix, and a viewport matrix, $C_1 = V_1 \cdot P_1 \cdot A_1$. This camera matrix transforms a point from the global coordinate system into the camera's projected image coordinate system. The projected image coordinates $(x_1, y_1)$, obtained after multiplying the point's global coordinates by $C_1$ and dividing out $w_1$, index a screen pixel address. The $z_1$ coordinate can be used for depth comparisons in a z buffer.

Let $C_2$ be the output camera's matrix. Define the transfer matrix as $T_{1,2} = C_2 \cdot C_1^{-1}$. Given the projected image coordinates of some point seen in the LDI camera (e.g., the coordinates of $a$ in Figure 5), this matrix computes the image coordinates as seen in the output camera (e.g., the image coordinates of $a_2$ in camera $C_2$ in Figure 5).

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \cdot w_2 \\ y_2 \cdot w_2 \\ z_2 \cdot w_2 \\ w_2 \end{bmatrix} = \textbf{result}$$

The coordinates $(x_2, y_2)$ obtained after dividing by $w_2$, index a pixel address in the output camera's image.

Using the linearity of matrix operations, this matrix multiply can be factored to reuse much of the computation from each iteration through the layers of a layered depth pixel; **result** can be computed as

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + z_1 \cdot T_{1,2} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \textbf{start} + z_1 \cdot \textbf{depth}$$

To compute the warped position of the next layered depth pixel along a scanline, the new **start** is simply incremented.

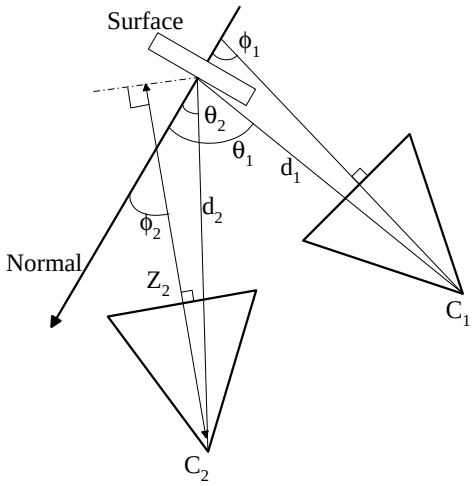


**Figure 8** Values for size computation of a projected pixel.

$$T_{1,2} \cdot \begin{bmatrix} x_1 + 1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + T_{1,2} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \textbf{start} + \textbf{xincr}$$

The warping algorithm proceeds using McMillan's ordering algorithm [21]. The LDI is broken up into four regions above and below and to the left and right of the epipolar point. For each quadrant, the LDI is traversed in (possibly reverse) scan line order. At the beginning of each scan line, **start** is computed. The sign of **xincr** is determined by the direction of processing in this quadrant. Each layered depth pixel in the scan line is then warped to the output image by calling *Warp*. This procedure visits each of the layers in back to front order and computes **result** to determine its location in the output image. As in perspective texture mapping, a divide is required per pixel. Finally, the depth pixel's color is splatted at this location in the output image.

The following pseudo code summarizes the warping algorithm applied to each layered depth pixel.

```
procedure Warp(ldpix, start, depth, xincr)
    for k ←0 to dpix.NumLayers-1
        z1 ←ldpix.Layers[k].Z
        result ←start + z1 ∗ depth
        //cull if the depth pixel goes behind the output camera
        //or if the depth pixel goes out of the output cam's frustum
        if result.w > 0 and IsInViewport(result) then
            result ←result / result.w
            // see next section
            sqrtSize ←z2 ∗ lookupTable[ldpix.Layers[k].SplatIndex]
            splat(ldpix.Layers[k].ColorRGBA, x2, y2, sqrtSize)
        end if
        // increment for next layered pixel on this scan line
        start ←start + xincr
    end for
end procedure
```

### 5.3 Splat Size Computation

To splat the LDI into the output image, we estimate the projected area of the warped pixel. This is a rough approximation to the footprint evaluation [34] optimized for speed. The proper size can be computed (differentially) as

$$size = \frac{(d_1)^2 \cos(\theta_2)\, res_2 \tan(fov_1/2)}{(d_2)^2 \cos(\theta_1)\, res_1 \tan(fov_2/2)}$$

where $d_1$ is the distance from the sampled surface point to the LDI camera, $fov_1$ is the field of view of the LDI camera, $res_1 = (w_1 h_1)^{-1}$ where $w_1$ and $h_1$ are the width and height of the LDI, and $\theta_1$ is the angle between the surface normal and the line of sight to the LDI camera (see Figure 8). The same terms with subscript 2 refer to the output camera.

It will be more efficient to compute an approximation of the square root of size,

$$\begin{aligned}\sqrt{size} &= \frac{1}{d_2} \cdot \frac{d_1 \sqrt{\cos(\theta_2) res_2 \tan(fov_1/2)}}{\sqrt{\cos(\theta_1) res_1 \tan(fov_2/2)}} \\ &\approx \frac{1}{Z_2} \cdot \frac{d_1 \sqrt{\cos(\phi_2) res_2 \tan(fov_1/2)}}{\sqrt{\cos(\phi_1) res_1 \tan(fov_2/2)}} \\ &\approx z_2 \cdot \frac{d_1 \sqrt{\cos(\phi_2) res_2 \tan(fov_1/2)}}{\sqrt{\cos(\phi_1) res_1 \tan(fov_2/2)}}\end{aligned}$$

We approximate the $\theta$s as the angles $\phi$ between the surface normal vector and the $z$ axes of the camera's coordinate systems. We also approximate $d_2$ by $Z_2$, the $z$ coordinate of the sampled point in the output camera's unprojected eye coordinate system. During rendering, we set the projection matrix such that $z_2 = 1/Z_2$.

The current implementation supports 4 different splat sizes, so a very crude approximation of the size computation is implemented using a lookup table. For each pixel in the LDI, we store $d_1$ using 5 bits. We use 6 bits to encode the normal, 3 for $n_x$, and 3 for $n_y$. This gives us an eleven-bit lookup table index. Before rendering each new image, we use the new output camera information to precompute values for the 2048 possible lookup table indexes. At each pixel we obtain $\sqrt{size}$ by multiplying the computed $z_2$ by the value found in the lookup table.

$$\sqrt{size} \approx z_2 \cdot \texttt{lookup[nx, ny, d1]}$$

To maintain the accuracy of the approximation for $d_1$, we discretize $d_1$ nonlinearly using a simple exponential function that allocates more bits to the nearby $d_1$ values, and fewer bits to the distant $d_1$ values.

The four splat sizes we currently use have 1 by 1, 3 by 3, 5 by 5, and 7 by 7 pixel footprints. Each pixel in a footprint has an alpha value to approximate a Gaussian splat kernel. However, the alpha values are rounded to 1, 1/2, or 1/4, so the alpha blending can be done with integer shifts and adds.

### 5.4 Depth Pixel Representation

The size of a cache line on current Intel processors (Pentium Pro and Pentium II) is 32 bytes. To fit four depth pixels into a single cache line we convert the floating point Z value to a 20 bit integer. This is then packed into a single word along with the 11 bit splat table index. These 32 bits along with the R, G, B, and alpha values fill out the 8 bytes. This seemingly small optimization yielded a 25 percent improvement in rendering speed.

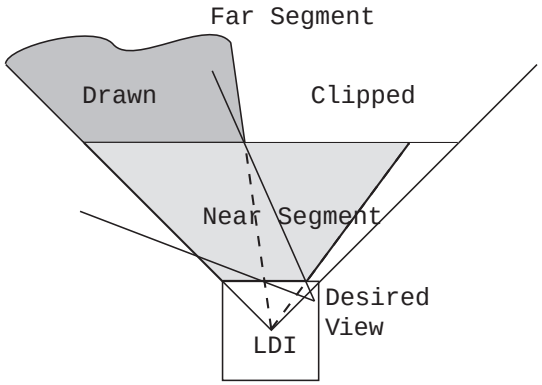


**Figure 9** LDI with two segments

### 5.5 Clipping

The LDI of the chestnut tree scene in Figure 11 is a large data set containing over 1.1 million depth pixels. If we naively render this LDI by reprojecting every depth pixel, we would only be able to render at one or two frames per second. When the viewer is close to the tree, there is no need to flow those pixels that will fall outside of the new view. Unseen pixels can be culled by intersecting the view frustum with the frustum of the LDI. This is implemented by intersecting the view frustum with the near and far plane of the LDI frustum, and taking the bounding box of the intersection. This region defines the rays of depth pixels that could be seen in the new view. This computation is conservative, and gives suboptimal results when the viewer is looking at the LDI from the side (see Figure 9). The view frustum intersects almost the entire cross section of the LDI frustum, but only those depth pixels in the desired view need be warped. Our simple clipping test indicates that most of the LDI needs to be warped. To alleviate this, we split the LDI into two segments, a near and a far segment (see Figure 9). These are simply two frustra stacked one on top of the other. The near frustum is kept smaller than the back segment. We clip each segment individually, and render the back segment first and the front segment second. Clipping can speed rendering times by a factor of 2 to 4.

## 6 Results

Sprites with Depth and Layered Depth Images have been implemented in C++. The color figures show two examples of rendering sprites and three examples of rendering LDIs. Figures 3a through 3j show the results of rendering a sprite with depth. The hemisphere in the middle of the sprite pops out of the plane of the sprite, and the illusion of depth is quite good. Figure 4 shows the process of extracting sprites from multiple images using the vision techniques discussed in Section 3. There is a great deal of parallax between the layers of sprites, resulting in a convincing and inexpensive image-based-rendering method.

Figure 10 shows two views of a barnyard scene modeled in Softimage. A set of 20 images was pre-rendered from cameras that encircle the chicken using the Mental Ray renderer. The renderer returns colors, depths, and normals at each pixel. The images were rendered at 320 by 320 pixel resolution, taking approximately one minute each to generate. In the interactive system, the 3 images out of the 17 that have the closest direction to the current camera are chosen. The preprocessor (running in a low-priority thread) uses these images to create an LDI in about 1 second. While the LDIs are allocated with a maximum of 10 layers per pixel, the average depth complexity for these LDIs is only 1.24. Thus the use of three

**Figure 10** Barnyard scene

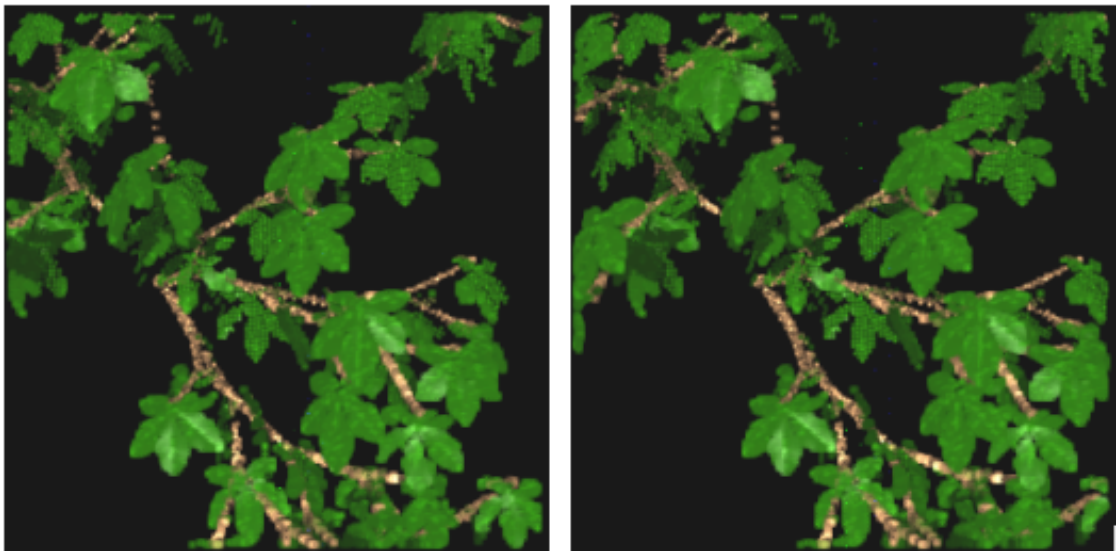

**Figure 11** Near segment of chestnut tree

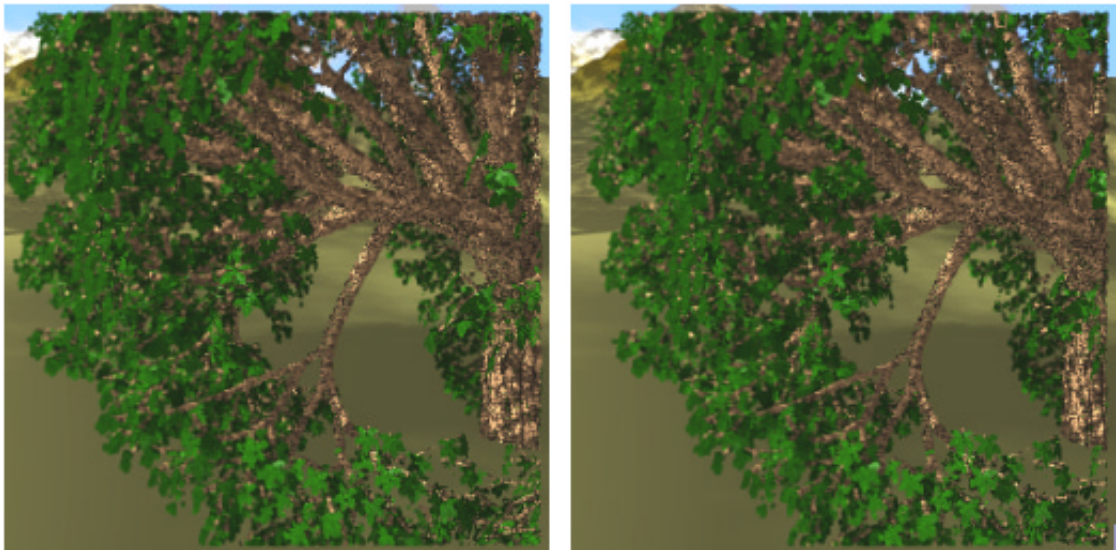

**Figure 12** Chestnut tree in front of environment map

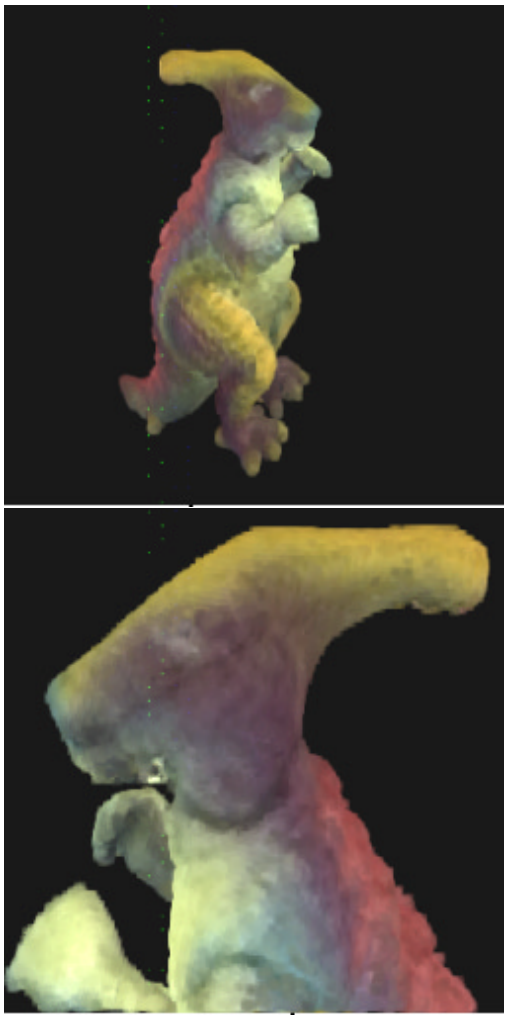

**Figure 13** Dinosaur model reconstructed from 21 photographs

input images only increases the rendering cost by 24 percent. The fast renderer (running concurrently in a high-priority thread) generates images at 300 by 300 resolution. On a Pentium II PC running at 300MHz, we achieved frame rate of 8 to 10 frames per second.

Figures 11 and 12 show two cross-eye stereo pairs of a chestnut tree. In Figure 11 only the near segment is displayed. Figure 12 shows both segments in front of an environment map. The LDIs were created using a modified version of the Rayshade [11] raytracer. The tree model is very large; Rayshade allocates over 340 MB of memory to render a single image of the tree. The stochastic method discussed in Section 4.2 took 7 hours to trace 16 million rays through this scene using an SGI Indigo2 with a 250 MHz processor and 320MB of memory. The resulting LDI has over 1.1 million depth pixels, 70,000 of which were placed in the near segment with the rest in the far segment. When rendering this interactively we attain frame rates between 4 and 10 frames per second on a Pentium II PC running at 300MHz.

## 7 Discussion

In this paper, we have described two novel techniques for image based rendering. The first technique renders Sprites with Depth without visible gaps, and with a smoother rendering than traditional forward mapping (splatting) techniques. It is based on the observation that a forward mapped displacement map does not have to be as accurate as a forward mapped color image. If the displacement map is smooth, the inaccuracies in the warped displacement map result in only sub-pixel errors in the final color pixel sample positions.

Our second novel approach to image based rendering is a Layered Depth Image representation. The LDI representation provides the means to display the parallax induced by camera motion as well as reveal disoccluded regions. The average depth complexity in our LDI's is much lower that one would achieve using multiple input images (e.g., only 1.24 in the Chicken LDI). The LDI representation takes advantage of McMillan's ordering algorithm allowing pixels to be splatted back to Front with an *over* compositing operation.

Traditional graphics elements and planar sprites can be combined with Sprites with Depth and LDIs in the same scene if a back-to-front ordering is maintained. In this case they are simply composited onto one another. Without such an ordering a z-buffer approach will still work at the extra cost of maintaining depth information per frame.

Choosing a single camera view to organize the data has the advantage of having sampled the geometry with a preference for views very near the center of the LDI. This also has its disadvantages. First, pixels undergo two resampling steps in their journey from input image to output. This can potentially degrade image quality. Secondly, if some surface is seen at a glancing angle in the LDIs view the depth complexity for that LDI increases, while the spatial sampling resolution over that surface degrades. The sampling and aliasing issues involved in our layered depth image approach are still not fully understood; a formal analysis of these issues would be helpful.

With the introduction of our two new representations and rendering techniques, there now exists a wide range of different image based rendering methods available. At one end of the spectrum are traditional texture-mapped models. When the scene does not have too much geometric detail, and when texture-mapping hardware is available, this may be the method of choice. If the scene can easily be partitioned into non-overlapping sprites (with depth), then triangle-based texture-mapped rendering can be used without requiring a z buffer [18, 4].

All of these representations, however, do not explicitly account for certain variation of scene appearance with viewpoint, e.g., specularities, transparency, etc. View-dependent texture maps [5], and 4D representations such as lightfields or Lumigraphs [16, 7], have been designed to model such effects. These techniques can lead to greater realism than static texture maps, sprites, or Layered Depth Images, but usually require more effort (and time) to render.

In future work, we hope to explore representations and rendering algorithms which combine several image based rendering techniques. Automatic techniques for taking a 3D scene (either synthesized or real) and re-representing it in the most appropriate fashion for image based rendering would be very useful. These would allow us to apply image based rendering to truly complex, visually rich scenes, and thereby extend their range of applicability.

### Acknowledgments

The authors would first of all like to thank Michael F. Cohen. Many of the original ideas contained in this paper as well as much of the discussion in the paper itself can be directly attributable to him. The authors would also like to thank Craig Kolb for his help in obtaining and modifying Rayshade. Radomir Mech and Przemyslaw Prusinkiewicz provided the model of the chestnut tree. Steve Seitz is responsible for creating the LDI of the dinosaur from a modified version of his earlier code. Andrew Glassner was a great help with

some of the illustrations in the paper. Finally, we would like to thank Microsoft Research for helping to bring together the authors to work on this project.

## References

[1] S. Baker, R. Szeliski, and P. Anandan. A Layered Approach to Stereo Reconstruction. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'98)*. Santa Barbara, June 1998.

[2] Shenchang Eric Chen and Lance Williams. View Interpolation for Image Synthesis. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288. August 1993.

[3] William Dally, Leonard McMillan, Gary Bishop, and Henry Fuchs. The Delta Tree: An Object Centered Approach to Image Based Rendering. AI technical Memo 1604, MIT, 1996.

[4] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating Static Environments Using Image-Space Simplification and Morphing. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 25–34. 1997.

[5] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996.

[6] O. Faugeras. *Three-dimensional computer vision: A geometric viewpoint*. MIT Press, Cambridge, Massachusetts, 1993.

[7] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The Lumigraph. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996.

[8] Paul S. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[9] Paul S. Heckbert and Henry P. Moreton. Interpolation for Polygon Texture Mapping and Shading. In David Rogers and Rae Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.

[10] Youichi Horry, Ken ichi Anjyo, and Kiyoshi Arai. Tour Into the Picture: Using a Spidery Mesh Interface to Make Animation from a Single Image. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 225–232. ACM SIGGRAPH, Addison Wesley, August 1997.

[11] Craig E. Kolb. *Rayshade User's Guide and Reference Manual*. http://graphics.stanford.edu/ cek/rayshade, 1992.

[12] R. Kumar, P. Anandan, and K. Hanna. Direct recovery of shape from multiple views: A parallax based approach. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, volume A, pages 685–688. IEEE Computer Society Press, Jerusalem, Israel, October 1994.

[13] Anthony G. LaMarca. Caches and Algorithms. Ph.D. thesis, University of Washington, 1996.

[14] S. Laveau and O. D. Faugeras. 3-D Scene Representation as a Collection of Images. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, volume A, pages 689–691. IEEE Computer Society Press, Jerusalem, Israel, October 1994.

[15] Jed Lengyel and John Snyder. Rendering with Coherent Layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997.

[16] Marc Levoy and Pat Hanrahan. Light Field Rendering. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996.

[17] Mark Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical Report 85-022, University of North Carolina, 1985.

[18] William R. Mark, Leonard McMillan, and Gary Bishop. Post-Rendering 3D Warping. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 7–16. 1997.

[19] Nelson Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 165–174. Eurographics, Springer Wein, New York City, NY, June 1996.

[20] Leonard McMillan. Computing Visibility Without Depth. Technical Report 95-047, University of North Carolina, 1995.

[21] Leonard McMillan. A List-Priority Rendering Algorithm for Redisplaying Projected Surfaces. Technical Report 95-005, University of North Carolina, 1995.

[22] Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995.

[23] Leonard McMillan and Gary Bishop. Shape as a Pertebation to Projective Mapping. Technical Report 95-046, University of North Carolina, 1995.

[24] Don P. Mitchell. *personal communication*. 1997.

[25] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 101–108. ACM SIGGRAPH, Addison Wesley, August 1997.

[26] H. S. Sawhney. 3D Geometry from Planar Parallax. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 929–934. IEEE Computer Society, Seattle, Washington, June 1994.

[27] Gernot Schaufler and Wolfgang Stürzlinger. A Three-Dimensional Image Cache for Virtual Reality. In *Proceedings of Eurographics '96*, pages 227–236. August 1996.

[28] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252. July 1992.

[29] Steven M. Seitz and Charles R. Dyer. View Morphing: Synthesizing 3D Metamorphoses Using Image Transforms. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 21–30. ACM SIGGRAPH, Addison Wesley, August 1996.

[30] Steven M. seitz and Charles R. Dyer. Photorealistic Scene Reconstruction by Voxel Coloring. In *Proc. Computer Vision and Pattern Recognition Conf.*, pages 1067–1073. 1997.

[31] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996.

[32] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 353–364. ACM SIGGRAPH, Addison Wesley, August 1996.

[33] J. Y. A. Wang and E. H. Adelson. Layered Representation for Motion Analysis. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'93)*, pages 361–366. New York, New York, June 1993.

[34] Lee Westover. Footprint Evaluation for Volume Rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376. August 1990.

[35] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, California, 1990.