Rendering Layered Depth Images

Steven J. Gortler
Harvard University

Li-wei He
Stanford University

Michael F. Cohen
Microsoft Research

March 19, 1997

Technical Report
MSTR-TR-97-09

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

In this paper we present an efficient image based rendering system that renders multiple frames per second on a PC. Our method 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. When n input images are preprocessed to create a single LDI, the size of the representation grows linearly only with the observed depth complexity in the n images, instead of linearly with n. 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 one efficient way of synthesizing views of real and synthetic objects. With traditional rendering techniques, the time required to render an image becomes unbounded as the geometric complexity of the scene grows. The rendering time also grows as the requested shading computations (such as those requiring global illumination solutions) become more ambitious.

In the simplest image based rendering technique, one synthesizes a new image from a single input *depth image*, that is, an image with z-buffer information stored with each pixel. Hidden surfaces are not included in the input image, and thus the image has an effective depth complexity of one. The complexity of the image has constant size, determined by the image resolution, and thus new images can be rendered in constant time. Shading computations, already computed for the input image, can be reused by subsequent images. Finally, if a depth image is obtained from a real world scene using real images, new views can be created with IBR methods without first creating a traditional geometric representation.

Because the pixels of an image form a regular grid, image based rendering computations are largely incremental and inexpensive. Moreover McMillan has developed an ordering algorithm [10, 12] that ensures that pixels in the synthesized image are drawn back to front, and thus no depth comparisons are required. This also permits proper alpha compositing of pixels without depth sorting.

Despite these advantage, there are still many problems with current image based rendering methods. For example, since a single depth image has no information about hidden surfaces, if the viewer moves slightly and thereby uncovers a surface, no relevant information is available for this newly unoccluded surface (see Color Image (a) and (b)). A simple solution to this problem is the use of more than one input image, but this approach is not without its drawbacks. If one uses n input images, one effectively multiplies the size of the scene description by n, and the rendering cost increases accordingly. Moreover, if one uses more than one input image, McMillan's ordering algorithm no longer applies, and one must perform depth computations (z-buffer) to resolve hidden surfaces in the output image.

Another difficulty arises from the fact that the input image has a different sampling pattern and density than the output image. When mapping the discrete pixels forward from the input image, many input pixels might squeeze together in an output pixel and should be properly blended for anti-aliasing. A more serious problem occurs when the forward mapping spreads the pixels apart, creating gaps in the output picture (see Color Image (a)). Proposed solutions to this problem include performing a backwards mapping from the output sample location to the input image [5]. This is an expensive operation that require 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 [8].

The simplest solution to fill gaps in the output image is to predict the projected size of an input pixel in the new projected view, and to "splat" the input pixel into the output image using a precomputed footprint [12]. For the splats to combine smoothly in the output image, the outer regions of the splat should have fractional alpha values and be composed into the new image using the "over" operation. This requires the output pixels to be drawn in depth order. But, as stated earlier, McMillan's ordering algorithm cannot be applied when more than one input image is used, and so a depth sort would seem to be required.

In this paper we present a method that solves many of these difficulties. The result is a very efficient image based rendering system that renders multiple frames per second on a PC. Our solution is as follows: instead of performing IBR directly from a set of n input depth images, we first create a single intermediate representation called a **layered depth image** (LDI). 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. When n input images are preprocessed to create a single LDI, pixels from different images that sample the same surface location are collapsed into a single depth pixel. The size of the representation grows linearly only with the observed depth complexity in the n images, instead of linearly with n. Moreover, because the LDI data are represented in a single image coordinate system, McMillan's 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.

2 Previous Work

Over the past few years, there have been many papers on image based rendering. In [7], Levoy and Whitted discuss rendering point data. Chen and Williams presented the idea of rendering from images [1]. Laveau and Faugeras discuss IBR using a backwards map [5]. McMillan and Bishop discuss IBR using cylindrical views [12]. Seitz and Dyer describe a system that allows a user to correctly model view transforms in a user controlled image morphing system [15]. In a slightly different direction, Levoy and Hanrahan [6] and Gortler et al. [3] 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 [9], 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.

The system presented here relies heavily on McMillan's ordering algorithm [10, 11, 12]. Using input and output camera information, a warping order is deduced such that any pixels that map to the same location in the output image are guaranteed to arrive in back to front order.

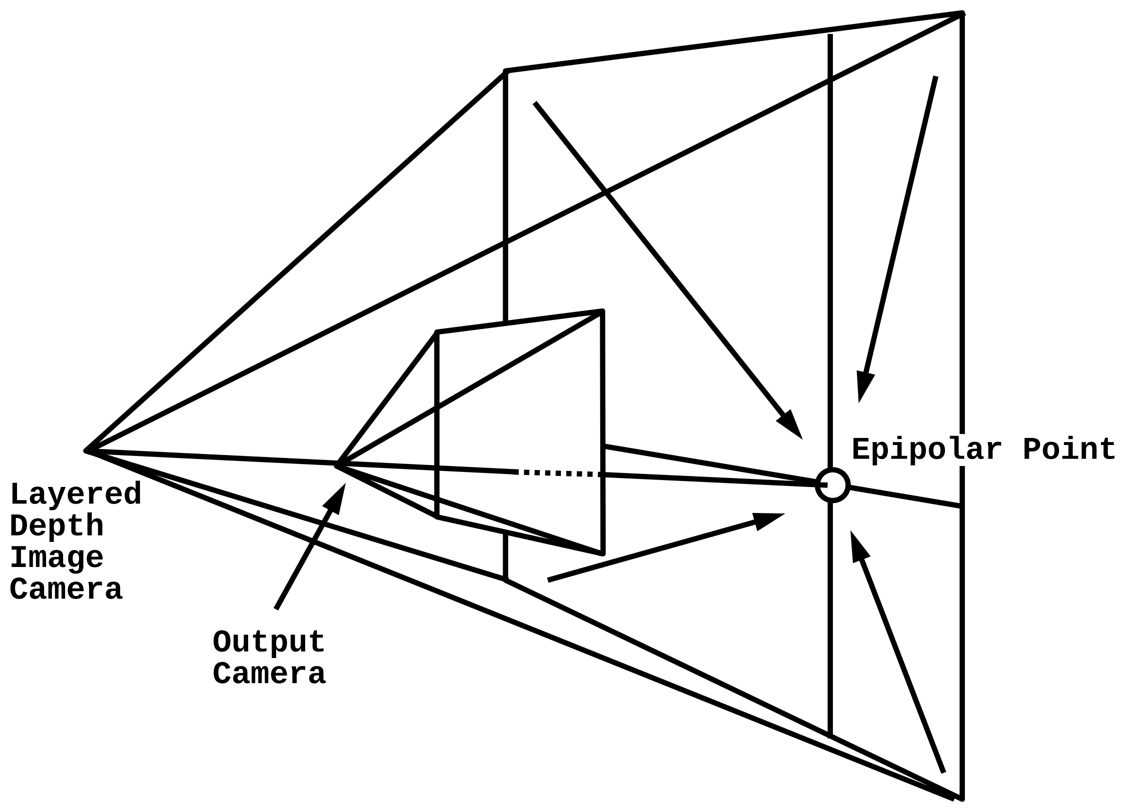

Figure 1: Back to front output ordering

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. This point, known as the epipolar point, is the intersection of the line joining the two camera centers, with the first camera's film plane (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 [11].

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 [11] applies, and the ordering algorithm is guaranteed to work.

3 Rendering System

The layered depth image data structure is defined as follows.

```
LayeredDepthImage{
    Camera;
    LayeredDepthPixel[Xres,Yres];
}

LayeredDepthPixel{
    NumActiveLayers;
    DepthPixel[MaxLayers];
}

DepthPixel{
     RGBcolor;
     Zdepth;
     TableIndex;
}
```

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 layers 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. This index is formed from a combination of the normal of the object seen and the distance from the camera that originally captured the pixel. This index will be used to compute the splat size as outlined in section 3.2.

There are a variety of ways to generate an LDI. For example, one could use a ray tracer that stores all of the computed ray intersections for each pixel. Or one could use multiple images for which depth information is available at each pixel. These can be derived from a multi-baseline stereo camera vision system capable of inferring depth.

In this paper, we construct an LDI by warping n synthetic depth images into a common camera view. [1] If, during the warp from input camera to LDI, two or more pixels map to the same layered depth pixel, then 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., NumActiveLayers is incremented and a new depth pixel is added), otherwise, the values are averaged with the present layer. This preprocess is similar to the rendering described by Max [9]. The 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.

As in any image based rendering system, there needs to be some control mechanism for determining which of the available input images should be used in the rendering. In our case, this amounts to deciding which images should be used to define the current LDI. How these decisions are made is an orthogonal issue and is not discussed in this paper.

The main component of our system is the fast warping based renderer. This component takes as input an LDI along with its associated camera information. Given new desired

[1] 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.

camera information, the warper uses the incremental warping computation 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.

3.1 Incremental Warping Computation

The incremental warping computation is similar to the one used for certain texture mapping operations [4, 14]. The geometry of this computation has been analyzed by McMillan [13], and efficient computation for the special case of orthographic input images is given in [2].

Let C_1 be the four by four matrix for the LDI camera. It is comprised of an affine transformation matrix, a projection matrix, and a viewport matrix, $C_1 = V_1 \cdot P_1 \cdot A_1$, and 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, index a 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, this matrix computes the image coordinates as seen in the output camera.

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \cdot w \\ y_2 \cdot w \\ z_2 \cdot w \\ w \end{bmatrix} = \texttt{resultVec}$$

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

Using the linearity of matrix operations, this matrix multiply can be factored to allow one to reuse much of the computation as one iterates through the layers of a layered depth pixel; `resultVec` can be computed as

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + z_1 \cdot T_{1,2} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \texttt{startVec} + z_1 \cdot \texttt{depthVec}$$

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

$$T_{1,2} \cdot \begin{bmatrix} x_1 + 1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + T_{1,2} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \texttt{startVec} + \texttt{incrVec}$$

The warping algorithm proceeds using McMillan's ordering algorithm as in [10]. 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, `startVec` is computed. The sign of `incrVec` 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 `warpLayeredDepthPixel`. This procedure visits each of the layers in back to front order and computes `resultVec` to determine its location in the output image. As in texture mapping, a divide is required per pixel. Finally, the depth pixel's color is splatted at this location in the output image.

```
warpLayeredDepthPixel(layeredDepthPixel pix,
                      4vec   &startVec,  // reference variable
                      4vec   depthVec,
                      4vec   incrVec,
                      image  outIm)

  for (k=0; k < pix.howManyActiveLayers; k++)
    z1 = pix.layer[k].z
    resultVec = startVec + z1 * depthVec

    //cull if the pixel goes behind the output camera
    //or if the pixel goes out of the output cam's frustum
    if (resultVec.w > 0 &&
        imageRange(resultVec, outIm.width, outIm.height) )

       recip = 1./resultVec.w

       x2  = resultVec.x * recip;
       y2  = resultVec.y * recip;
       z2  = resultVec.z * recip;

       // see next section
       sqrtSize = z2 * lookupTable[pix.layer[k].tableIndex]
       splat(pix.layer[k].rgb, outIm, x2, y2, sqrtSize)

  // increment for next layered pixel on this scan line
  startVec += IncrVec
```

3.2 Splat Size Computation

To splat the LDI into the output image, we approximate the projected area of the warped pixel. The proper size can be computed (differentially) as

$$size = \frac{(d_1)^2\, cos(\theta_2)\, res_2\, tan(.5fov_1)}{(d_2)^2\, cos(\theta_1)\, res_1\, tan(.5fov_2)}$$

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 is the pixel resolution of the LDI camera, and θ_1 is the angle

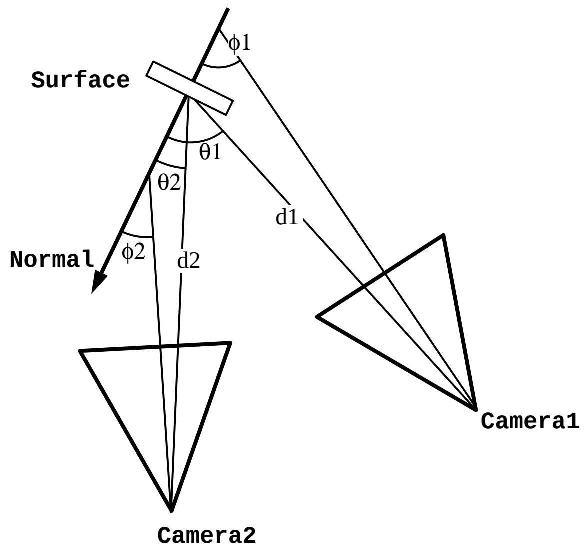

Figure 2: Values for size computation of a projected pixel.

between the surface normal and the line of sight to the LDI camera (see Figure 2). 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,

$$\begin{aligned}\sqrt{size} &= \frac{1}{d_2} \cdot \frac{d_1\sqrt{cos(\theta_2)res_2tan(.5fov_1)}}{\sqrt{cos(\theta_1)res_1tan(.5fov_2)}} \\ &\approx \frac{1}{z_{e2}} \cdot \frac{d_1\sqrt{cos(\phi_2)res_2tan(.5fov_1)}}{\sqrt{cos(\phi_1)res_1tan(.5fov_2)}} \\ &\approx z_2 \cdot \texttt{lookup}[\texttt{nx},\texttt{ny},\texttt{d1}]\end{aligned}$$

We approximate the θs as the angles ϕ between the surface normal vector and the z axes of the camera's coordinate systems. We also approximate d_2 by z_{e2}, the z coordinate of the sampled point in the output camera's unprojected eye coordinate system.

The current implementation supports 3 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 2 bits. We use 4 bits to encode the normal, 2 for n_x, and 2 for n_y. This gives us a six-bit lookup table index. Before rendering each new image, we use the new output camera information to precompute values for the 64 possible lookup table indexes. During rendering we choose the projection matrix P_2 such that $z_2 = 1/z_{e2}$. At each pixel we obtain $\sqrt{size}$ by multiplying the computed z_2 by the value found in the lookup table.

The three splat sizes we currently use are a 1 pixel footprint, a 3 by 3 pixel footprint, and a 5 by 5 pixel footprint. Each pixel in a footprint has an alpha value of 1, 1/2, or 1/4, so the alpha blending can be done with integer shifts and adds. The following splat masks are used:

$$1 \quad \text{and} \quad .25\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \text{and} \quad .25\begin{bmatrix} 1 & 2 & 2 & 2 & 1 \\ 2 & 2 & 4 & 2 & 2 \\ 2 & 4 & 4 & 4 & 2 \\ 2 & 2 & 4 & 2 & 2 \\ 1 & 2 & 2 & 2 & 1 \end{bmatrix}$$

4 Results

We have implemented our system in C++. In the accompanying video, we used a barnyard scene modeled in Softimage, and a cache of 10 images rendered from the scene with the Mental Ray renderer. The renderer returns colors, depths, and normals at each pixel. The images were rendered at 320 by 320 pixel resolution. Each image took approximately one minute to generate.

In the interactive system, a controller uses the current camera information to pick the "best" 3 images out of the image cache. The preprocessor (running in a low-priority thread) uses these images to create an LDI in about 1 second. The LDIs are allocated with a maximum of 10 layers per pixel. The average depth complexity for these LDIs was approximately 1.24. Thus the use of three input images only increases the rendering cost by 24 percent. The fast renderer (running concurrently in a high-priority thread) generates images at 256 by 256 resolution (see Color Images). On a Pentium-Pro PC running at 200Mhz, we achieved a five fps frame rate.

5 Discussion

In this paper we have described a method for efficient image based rendering using a layered depth image representation. This representation allows us to use the information available from an input set of n images without incurring most of the cost associated with multiple images. The average depth complexity in our LDI's was 1.24. The LDI representation allows us to apply McMillan's ordering algorithm, and allows us to splat pixels with the *over* compositing operation.

Choosing a single camera view to organize the data does have 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 chosen camera view, the depth complexity for that LDI increases, while the spatial sampling resolution over that surface degrades.

The sampling and aliasing issues involved in image based rendering are still not fully understood; a formal analysis of these issues would be invaluable.

References

[1] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Computer Graphics, Annual Conference Series, 1993*, pages 279–288.

[2] William Dally, Leonard McMillan, Gary Bishop, and Henry Fuchs. The delta tree: An object centered approach to image based rendering. Mit ai technical memo, 1996.

[3] Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Computer Graphics, Annual Conference Series, 1996*, pages 43–54.

[4] 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.

[5] 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, Jerusalem, Israel, October 1994. IEEE Computer Society Press.

[6] Mark Levoy and Pat Hanrahan. Light-field rendering. In *Computer Graphics, Annual Conference Series, 1996*, pages 31–42.

[7] Mark Levoy and Turner Whitted. The use of points as a display primitive. UNC Technical Report 85-022, University of North Carolina, 1985.

[8] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In *1997 Symposium on Interactive 3D Graphics (to appear)*. ACM, 1997.

[9] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer z-buffers. In *Seventh Eurographics Workshop on Rendering*, pages 166–175. Eurographics, June 1996.

[10] Leonard McMillan. A list-priority rendering algorithm for redisplaying projected surfaces. UNC Technical Report 95-005, University of North Carolina, 1995.

[11] Leonard McMillan. Computing visibility without depth. unpublished manuscript, 1996.

[12] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics, Annual Conference Series, 1995*, pages 39–46.

[13] Leonard McMillan and Gary Bishop. Shape as a pertebation to projective mapping. unpublished manuscript, 1996.

[14] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics*, 26(2):249–252, 1992.

[15] Steven M. Seitz and Charles R. Dyer. View morphing. In *Computer Graphics, Annual Conference Series, 1996*, pages 21–30.