View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data

Kari Pulli* Michael Cohen† Tom Duchamp*
Hugues Hoppe† Linda Shapiro* Werner Stuetzle*

*University of Washington, Seattle, WA
†Microsoft Research, Redmond, WA

Abstract

Modeling arbitrary real objects is difficult and rendering textured models typically does not result in realistic images. We describe a new method for displaying scanned real objects, called *view-based rendering*. The method takes as input a collection of colored range images covering the object and creates a collection of partial object models. These partial models are rendered separately using traditional graphics hardware and blended together using various weights and soft z-buffering. We demonstrate interactive viewing of real, non-trivial objects that would be difficult to model using traditional methods.

1 Introduction

In traditional *model-based rendering* a geometric model of a scene, together with surface reflectance properties and lighting parameters, is used to generate an image of the scene from a desired viewpoint. In contrast, in *image-based rendering* a set of images of a scene are taken from (possibly) known viewpoints and used to create new images. Image-based rendering has been an area of active research in the past few years because it can be used to address two problems:

Efficient rendering of complicated scenes. Some applications of rendering, such as walk-throughs of complex environments, require generation of images at interactive rates. One way to achieve this is to render the scene from a suitably chosen set of viewpoints. Images required during walk-through are then synthesized from the images computed during the pre-processing step. This idea is based on the premise that interpolation between images is faster than rendering the scene.

Three-dimensional display of real-world objects. Suppose we wish to capture the appearance of a 3D object in a way that allows the viewer to see it from any chosen viewpoint. An obvious solution is to create a model of the object capturing its shape and surface reflectance properties. However, generating realistic models of complex 3D objects is a nontrivial problem that we will further discuss below. Alternatively, we can capture images of the object from a collection of viewpoints, and then use those to synthesize new images.

The motivation for our work is realistic display of real objects. We present a method, *view-based rendering*, that lies in between purely model-based and purely image-based methods.

The construction of a full 3D model needed for model-based rendering requires a number of steps: 1) acquisition of range and color data from a number of viewpoints chosen to get complete coverage of the object, 2) registration of these data into a single coordinate system, 3) representation of all the data by a surface model that agrees with all the images, 4) computation of a surface reflection model at each point of this surface using the colors observed in the various images. Despite recent advances [4, 16], automatically creating accurate surface models of complex objects (step 3) is still a difficult task, while the computation of accurate reflection models (step 4) has hardly been addressed. In addition, the rendered images of such models do not look quite as realistic as photographs that can capture intricate geometric texture and global illumination effects with ease.

Our idea is to forgo construction of a full 3D object model. Rather, we create independent models for the depth maps observed from each viewpoint, a much simpler task. Instead of having to gather and manipulate a set of images dense enough for purely image-based rendering, we make do with a much sparser set of images, but use geometric information to more accurately interpolate between them. A request for an image of the object from a specified viewpoint is satisfied using the color and geometry in the stored views. This paper describes our new view-based rendering algorithm and shows results on non-trivial real objects.

The paper is organized as follows. Section 2 casts image-based rendering as an interpolation problem, where samples of the light field function are interpolated to create new images. Section 3 describes our view-based rendering approach. Section 4 presents details of our implementation, including data acquisition, view-based model generation, and use of graphics hardware for efficient implementation, and some results. Section 5 covers related work. Section 6 discusses hardware acceleration and concludes the paper.

2 Image-based rendering as an interpolation problem

The basic problem in image-based rendering is to compute an image of a scene as seen from some target viewpoint, using a set of input images, their corresponding camera poses, and possibly additional associated information. A useful abstraction in this context is the *light field function* (also known as the *plenoptic function*). Levoy and Hanrahan [12] define the light field as the radiance at a point *in* a given direction. For our purposes, it is more convenient to define the light field as the radiance at a point *from* a given direction (see Figure 1).

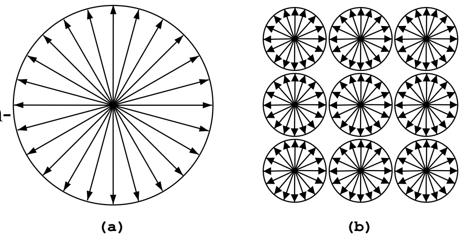

Figure 1: (a) A pencil of rays describes the colors of visible points from a given point. (b) The light field function describes the colors of all rays starting from any point.

More precisely, we define a *ray* to be a directed half-line originating from a 3D *basepoint*. We may therefore represent a ray as an ordered pair $(\mathbf{x}, \hat{\mathbf{n}}) \in \mathbb{R}^3 \times S^2$, where $\mathbf{x}$ is its basepoint, $\hat{\mathbf{n}}$ is its direction, and S^2 denotes the unit sphere. The light field is then a function $f : \mathbb{R}^3 \times S^2 \to \mathbb{R}^3$ which assigns to each ray $(\mathbf{x}, \hat{\mathbf{n}})$ an RGB-color $f(\mathbf{x}, \hat{\mathbf{n}})$. Thus, $f(\mathbf{x}, \hat{\mathbf{n}})$ measures the radiance at $\mathbf{x}$ in the direction $-\hat{\mathbf{n}}$. The collection of rays starting from a point is called a *pencil*. If we had complete knowledge of the light field function, we could render any view from any location $\mathbf{x}$ by associating a ray (or an average of rays) in the pencil based at $\mathbf{x}$ to each pixel of a virtual camera.

The full light field function is only needed to render entire environments from an arbitrary viewpoint. If we are content with rendering individual objects from some standoff distance, it suffices to know the light field function for the subset of $\mathbb{R}^3 \times S^2$ of "inward" rays originating from points on a convex surface M that encloses the object. Following Gortler *et al.* [9], we call this simpler function a *lumigraph*. We call the surface M that encloses the object the *lumigraph surface*. Figure 2 shows a schematic of the lumigraph domain for the case where the lumigraph surface is a sphere.

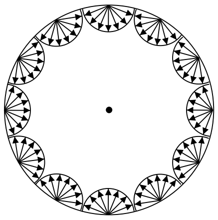

Figure 2: A spherical lumigraph surface.

The lumigraph contains all rays needed to synthesize an image from any viewpoint exterior to the convex hull of the object being modeled. Each pixel in the image defines a ray that intersects the lumigraph surface M at a point, say $\mathbf{x}$. If $\hat{\mathbf{n}}$ is the direction of that ray, then the RGB-color value assigned to the pixel is $f(\mathbf{x}, \hat{\mathbf{n}})$.

2.1 Distance measures for rays

In practice we will never be able to acquire the full 5D light field function or even a complete 4D lumigraph. Instead we will have a discrete set of images of the scene, taken at some finite resolution. In other words, we have the values of the function for a sample of rays (really for local averages of the light field function). To render the scene from a new viewpoint, we need to estimate the values of the function for a set of query rays from its values for the sample rays. Thus, *image-based rendering is an interpolation problem*.

In a generic interpolation problem, one is given the values of a function at a discrete set of sample points. The function value at a new query point is estimated by a weighted average of function values at the sample points, with weights concentrating on samples that are close to the query point. The performance of any interpolation method is critically dependent on the definition of "closeness".

In image-based rendering, the aim is to paint pixels on the image plane of a virtual camera, and therefore the renderer looks for rays close to the one associated with some particular pixel. In the next two sections we examine two closeness measures.

2.1.1 Ray-surface intersection

Figure 3 shows a piece of a lumigraph with several pencils of rays. In Fig. 3(a) there is no information about the object surface geometry. In that case we have to concentrate on pencils whose origins are close to the query ray and interpolate between rays that are parallel to the query ray. The denser the pencils are on the the lumigraph surface M, and the more rays in each pencil, the better the match we can expect to find.

Assuming that the object is a Lambertian reflector, the lumigraph representation has a high degree of redundancy: there are many rays that intersect the object surface at the same point. Figure 3(b) shows a case where the precise object geometry is not known, but there is an estimate of the average distance between the object surface and the lumigraph surface. We can estimate where the query ray intersects the object surface and choose rays from nearby pencils that point to the intersection point. The expected error in our estimate of $f(\mathbf{x}, \hat{\mathbf{n}})$ should now be less than in case (a). Or, to obtain the same error, we need far fewer sample rays (i.e. images).

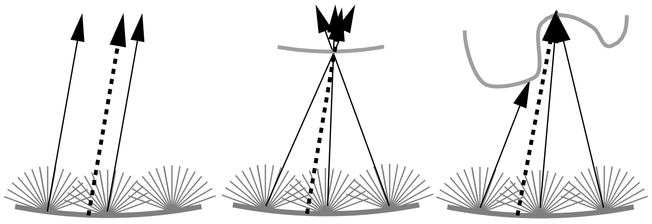

Figure 3 The query ray is dotted; sample rays are solid. (a) Choose similar rays. (b) Choose rays pointing to where the query ray meets surface. (c) Choose rays intersecting the surface where the query ray does.

Figure 3(c) illustrates the case where there is accurate information about the object geometry. To estimate $f(\mathbf{x}, \hat{\mathbf{n}})$, we can locate the sample rays that intersect the object surface at the same location as the query ray. With an accurate surface description it is possible to find all the rays directed towards that location and even remove rays that really intersect some other part of the surface first. Naturally, the expected error with a given collection of rays is minimized.

2.1.2 Ray direction

To improve the estimate of the lighting function we can take into account the direction and more heavily weight sample rays whose direction is near that of the query ray. There are three justifications for this. First, few surfaces reflect the incoming light uniformly in every direction. A typical example of this is specular reflections on shiny surfaces, but the appearance of many materials such as velvet or hair varies significantly with viewing direction. In image-based rendering this suggests favoring rays with similar directions.

Second, undetected self-occlusions may cause us to incorrectly conclude that two sample rays intersect the object surface at the same point and lead us to incorrectly estimate the light field function. If the occlusion is due to a large-scale object feature, and we have enough information about the surface geometry, we may be able to notice the self-occlusion and cull away occluded rays (see Fig. 3(c)). However, if the occlusion is due to small scale surface geometry, and we have only approximate information of the surface geometry, the occlusion is much harder to detect, as shown in Fig. 4(a). Moreover, if the object has thin features, as illustrated in Fig. 4(b), then rays may approach the object surface from opposite directions and intersect it at points that are spatially near, yet far apart with respect to distance as measured along the surface. The likelihood of such errors decreases by more heavily weighting sample rays whose directions are near the direction of the query ray.

Third, as shown in Fig. 4(c), when the angle between the query ray and the sample ray is large, small errors in the surface geometry can lead to large errors in the estimate of distance between the intersection points with the object surface. We get more robust results by favoring rays with similar direction to that of the query ray.

3 View-based rendering

The preprocessing of the input data is described in more detail in Section 4, but for clarity we summarize it here. The input to our view-based rendering system is a set of views,

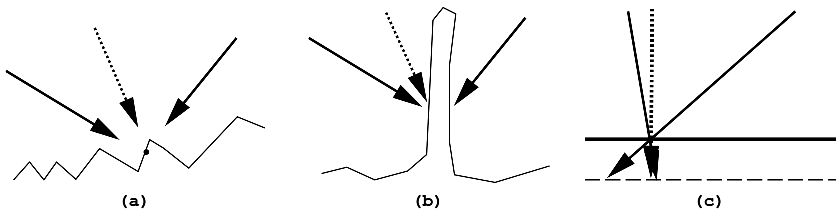

Figure 4 (a) Detailed surface geometry can cause occlusions that make the surface appear different from different directions. (b) Thin features can cause a discrepancy between surface distance and spatial distance of intersection points. (c) The more parallel the rays the less damaging an error in an estimate of surface distance.

i.e., colored range images of an object. Registering the range maps into a common coordinate system gives us the camera locations and orientations of the colored images with respect to the object. We replace each dense range map by a sparse triangle mesh that closely approximates it. We then texture map each triangle mesh using the associated colored image. To synthesize an image of the object from a fixed viewpoint we individually render the meshes constructed from three close viewpoints and blend them together with a pixel-based weighting algorithm that uses soft z-buffering.

3.1 A simple approach

To better understand the virtues of our approach, it is helpful to contrast it with a simpler algorithm. If we want to view the object from any of the stored viewpoints, we can place a virtual camera at one of them and render the associated textured mesh. We can move the virtual camera around by rendering the mesh from the new viewpoint. But as the viewpoint changes, parts of the surface not seen from the original viewpoint become visible, opening holes in the rendered image. If, however, the missing surface parts are seen from one or more other stored viewpoints, we can fill the holes by simultaneously rendering the textured meshes associated with the additional viewpoints. The resulting image is a collage of several individual images.

The results are displayed in Fig. 10(a). In terms of ray interpolation, the graphics hardware interpolates the rays within each view, finding a ray for each pixel that intersects the surface approximately where the query ray of the pixel does. However, there is no interpolation between the views, only the ray from the mesh that happens to be closest to the camera at the pixel is chosen. With imperfect geometrical information and registration, we get a lot of visible artifacts.

We can improve on this by interpolating rays between different views. The next section describes how we use various weights that account for such factors as viewing directions and surface sampling densities and how we blend rays correctly even in presence of partial self-occlusions. The results of the better interpolation are shown in Fig. 10(b).

3.2 Three weights and soft z-buffering

We preprocess the viewing directions of the input views as follows. Each viewing direction corresponds to a point on the unit sphere. The viewing directions thus define a set of points on the sphere and we compute the Delaunay triangulation of this set, as illustrated in Fig. 5(a).

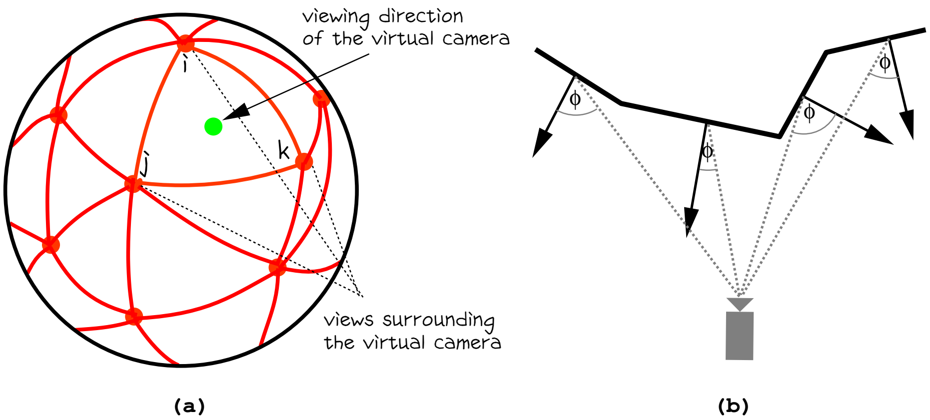

Figure 5 (a) The weights w_β assigned to the views at the vertices i, j, and k of the Delaunay triangles containing the current view are its barycentric coordinates. (b) The weight w_ϕ is the cosine of the angle ϕ between the normal and the ray to the sensor.

To synthesize an image of the object from a fixed viewpoint, we first select the three views corresponding to the vertices of the Delaunay triangle containing the current viewing direction of the virtual camera. The textured mesh of each selected view is individually rendered from this viewpoint to obtain three separate images. The images are blended into a single image by the following weighting scheme. Consider a single pixel. We set $c = \sum_{i=1}^{3} w_i c_i / \sum_{i=1}^{3} w_i$ where c_i is the color value associated with that pixel in the i^{th} image and w_i is a weight designed to overcome the difficulties encountered in the naive implementation mentioned above. The weight w_i is the product of three weights $w_i = w_{\beta,i} \cdot w_{\phi,i} \cdot w_{\gamma,i}$, whose definition is illustrated in Figs. 5 and 9. Self-occlusions are handled by using soft z-buffering to combine the images pixel by pixel.

The first weight, w_β, measures the proximity of a chosen view to the current viewpoint, and therefore changes dynamically as the virtual camera moves. We first calculate the barycentric coordinate β of the current viewpoint with respect to the Delaunay triangle containing it (see Fig. 5(a)). β has three components between 0.0 and 1.0 that sum to 1.0, each associated with one of the triangle vertices. The components give the weights that linearly interpolate the vertices to produce the current viewpoint. We define the weight w_β of view i to be the component of β associated with that view.

The remaining two weights w_ϕ and w_γ are pixel dependent but are independent of the view direction of the virtual camera. The second weight w_ϕ is a measure of surface sampling density (see Figs. 5(b) and 9(b)) and it is constant within each triangle in a mesh. Consider a point on a triangle in the mesh of view i corresponding to a given pixel. A small region of area A about the point projects to a region of area $A \cos\phi$ on the "image plane" of the i^{th} sensor, where ϕ is the angle between the normal to the triangle and the ray from the point to the sensor. We set $w_\phi = \cos\phi$. Darsa *et al.* [5] use a similar weight.

The third weight w_γ which we call the *blend weight*, is designed to smoothly blend the meshes at their boundaries. As illustrated by Fig. 9(c), the blend weight $w_{\gamma,i}$ of view i linearly increases with distance from the mesh boundary to the point projecting onto the pixel. Whereas w_β is associated with a view, and w_ϕ with the triangles approximating the geometry of the view, w_γ is associated with color texture of the view. A similar weight was used by Debevec *et al.* [6].

Most self-occlusions are handled during rendering of individual views using back-face culling and z-buffering. When combining the view-based partial models, part of one view's model may occlude part of another view's model. Unless the surfaces are relatively close to each other, the occluded pixel must be excluded from contributing to the pixel color. This is done by performing "soft" z-buffering, in software. First, we consult the z-buffer information of each separately rendered view and search for the smallest value. Views with z-values within a threshold from the closest are included in the composition, others are excluded. The threshold is chosen to slightly exceed an upper estimate of the combination of the sampling, registration, and polygonal approximation errors.

Figure 6 illustrates a potential problem. In the picture the view-based surface approximation of the rightmost camera has failed to notice a step edge due to self-occlusion in the data, and has incorrectly connected two surface regions. When performing the soft z-buffering for the pixel corresponding to the dashed line, the wrongly connected step edge would be so much closer than the contribution from the other view that the soft z-buffering would throw away the correct sample. However, while doing the soft z-buffering we can treat the weights as confidence measures. If a pixel with a very low confidence value covers a pixel with a high confidence value, the low confidence pixel is ignored altogether.

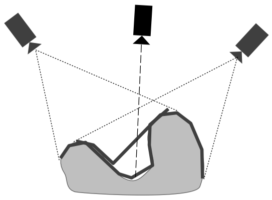

Figure 6: Problems with undetected step edges.

Rendering the registered geometry using graphics hardware and our soft z-buffering finds rays that intersect the surface where the query ray of a pixel does. Weights w_β and w_ϕ are used to favor good rays in the sense discussed in Section 2, while w_γ is used to hide the effects of inevitable inaccuracies due to the use of real scanned data.

4 Implementation

4.1 View acquisition

Data acquisition. We obtain the range data from a stereo camera system that uses active light. Both cameras have been calibrated, and an uncalibrated light source sweeps a beam (a vertical light plane) past the object in discrete steps. For each pixel on the beam, we project its epipolar line to the right camera's image plane. The intersection of the epipolar line and the bright line gives a pixel that sees the same surface point as the original pixel from the left camera. We obtain the 3D coordinates of that point by triangulating the corresponding pixels. After the view has been scanned, we turn the lights on and take a color picture of the object. The object is then repositioned so we can scan it from a different viewpoint.

View registration. Registering the views using the range data aligns the range maps around the object. A transformation applied to the range data also moves the sensor with respect to an object centered coordinate system, giving us the relative camera positions and orientations. We perform the initial registration interactively by marking identifiable object features in the color images. This initial registration is refined using Chen and Medioni's registration method [3] modified to deal with multiple data sets simultaneously.

Triangle mesh creation. We currently create the triangle meshes interactively. The user marks the boundaries of the object by inserting points into the color image, while the software incrementally updates a Delaunay triangulation of the vertices. The system optimizes the z-coordinates of all the vertices so that the least squares error of the range data approximation is minimized. Triangles that are almost parallel to the viewing direction are discarded, since they are likely to be step edges, not a good approximation of the object surface. Triangles outside of the object are discarded as well.

We have begun to automate the mesh creation phase. First, we place a blue cloth to the background and scan the empty scene. Points whose geometry and color match the data scanned from the empty scene are classified as background. The adding of vertices is easily automated. For example, Garland and Heckbert [8] add vertices to image coordinates where the current approximation is worst. The drawback of this approach is that if the data contains step edges due to self-occlusions, the mesh is likely to become unnecessarily dense before a good approximation is achieved. To prevent this we perform a mesh simplification step using the mesh optimization methods by Hoppe *et al.* [10].

4.2 Rendering

We have built an interactive viewer for viewing the reconstructed images (see Figure 11). For each frame, we find three views whose view directions surround the current view direction on a unit sphere. The three views are then rendered separately from the viewpoint of the virtual camera as textured triangle meshes and weighted using the barycentric coordinates of the current view direction with respect to the chosen views.

Two of the weights, w_ϕ and w_γ are static for each view, as they do not depend on the viewing direction of the virtual camera. We apply both of these weights offline and code them into the alpha channels of the mesh color and the texture map. w_ϕ is the weight used to decrease the importance of triangles that are tilted with respect to the scanner. It is applied by assigning the RGBA color $(1, 1, 1, w_\phi)$ to each triangle. w_γ is the weight used to hide artifacts at the mesh boundary of a view. It is directly applied to the alpha channel of the texture map that stores the color information. We calculate the weights for each pixel by first projecting the triangle mesh onto the color image and painting it white on a black background. We then calculate the distance d for each white pixel to the closest black pixel. The pixels with distances of at least n get alpha value 1; all other pixels get the value $\frac{d}{n}$.

Figure 7 gives the pseudo code for the view composition algorithm. The function `min_reliable_z()` returns the minimum z for a given pixel, unless the closest pixel is a low confidence (weight) point that would occlude a high confidence point, in which case the z for the minimum high confidence point is returned.

```
FOR EACH pixel
  zmin            := min_reliable_z( pixel )
  pixel_color     := (0,0,0)
  pixel_weight    := 0
  FOR EACH view
    IF zmin <= z[view,pixel] <= zmin+thrsoft_z THEN
      weight        :=  wθ * wφ * wγ
      pixel_color  += weight * color[view,pixel]
      pixel_weight += weight
    ENDIF
  END
  color[pixel] := pixel_color / pixel_weight
END
```

Figure 7: Pseudo code for color blending.

When we render a triangle mesh with the described colors and texture maps, the hardware calculates the correct weights for us. The alpha value in each pixel is $w_\phi \cdot w_\gamma$. It is also possible to apply the remaining weight, w_β, using graphics hardware. After we render the views, we have to read in the information from the frame buffer. OpenGL allows scaling each pixel while reading the frame buffer into memory. If we scale the alpha channel by w_β, the resulting alpha value contains the final weight $w_\beta \cdot w_\phi \cdot w_\gamma$.

4.3 Results

We have implemented our object visualization method on an SGI Maximum Impact with a 250 MHz MIPS 4400. We first obtain a polygonal approximation consisting of 100–250 triangles for each view. The user is free to rotate, zoom, and pan the object in front of the virtual camera. For each frame, we choose three views. The texture-mapped polygonal approximations of the views are rendered from the current viewpoint separately into 256×256 windows. The images are combined pixel by pixel into a composite image.

Figure 10 compares the simple approach of Section 3.1 to our view-based rendering method that uses three weights and soft z-buffering (Section 3.2). In Fig. 10(a) three views have been rendered repeatedly into the same frame from the viewpoint of the virtual camera. The mesh boundaries are clearly visible and the result looks like a badly made mosaic. In Fig. 10(b) the views have been blended smoothly pixel by pixel. Both the dog and the flower basket are almost free of blending artifacts such as background color showing at mesh boundaries and false surfaces due to undetected step edges in the triangle meshes.

Our current implementation can deliver about 8 frames per second. The execution time is roughly divided into the following components. Rendering the three texture mapped triangle meshes takes 37%, reading the color and z-buffers into memory takes 13%, building the composite image takes 44%, and displaying the result takes 6% of the total execution time.

4.4 Additional hardware acceleration

The only parts of our algorithm not currently supported by graphics hardware are the weighted pixel averaging and the soft z-buffering. The weighted averaging would be easy to implement by allowing more bits for the accumulation buffer, interpreting the alpha channel value as a weight instead of the opacity value, and providing a command that divides the RGB channels by the alpha channel value. An approximate implementation of the soft z-buffering in hardware would require adding, replacing, or ignoring the weighted color and the weight (alpha value) depending on whether the new pixel's z value is within, much closer, or much farther from the old z-value, respectively. For exact implementation two passes are required: first find minimum reliable z, then blend using soft threshold based on that minimum z.

5 Related work

Chen [1] and McMillan and Bishop [15] modeled environments by storing the light field function around a point. The rays visible from a point are texture mapped to a cylinder around that point, and any horizontal view can be created by warping a portion of the cylinder to the image plane. Both systems allow limited rotations about a vertical axis, but they do not support continuous translation of the viewpoint.

Levoy and Hanrahan [12] and Gortler *et al.* [9] developed image synthesis systems that use a lumigraph and that support continuous translation and rotation of the view point. In fact, the term "lumigraph" that we use to describe the 4D slice of the light field is borrowed from [9]. Both systems use a cube surrounding the object as the lumigraph surface. To create a lumigraph from digitized images of a real object, Levoy and Hanrahan moved the camera in a regular pattern into a known set of positions, and projected the camera images back to the lumigraph cube. Gortler *et al.* moved a hand-held video camera around an object placed on the capture stage. The capture stage is patterned with a set of concentric circles for estimating the camera pose for each image. The rays from the images are projected to the lumigraph walls, and the lumigraph is interpolated from these samples and stored as a grid of 2D images. In both systems, new images are synthesized from a stored grid of 2D images by an interpolation procedure, but Gortler *et al.* use additional geometric information to improve on ray interpolation. They create a rough model from the visual hull of the object. One advantage of the lumigraph methods is that they allow capturing the appearance of any object regardless of the complexity of its surface. A disadvantage is the difficulty of storing and accessing the enormous lumigraph representation.

The "algebraic" approach to image-based rendering using pairs of images and pixel correspondences in the two images was introduced by Laveau and Faugeras [11]. It has since been used in several other systems [15, 18, 7]. Given correct dense pixel correspondences one can calculate the 3D coordinates of surface points visible in both images, and then project these to the image plane of the virtual camera. However, the projection can also be calculated directly without 3D reconstruction. This is illustrated in Fig. 8 which shows the stored images 1 and 2, and the image plane of the virtual camera v. Since the pixel marked in image 1 corresponds to the one marked in image 2, their associated rays r_1 and r_2 are assumed to intersect at the same location on the object surface. That point projects to the image v at the intersection of the epipolar lines e_1 and e_2, which are the projections of r_1 and r_2 onto image v. The color of the destination pixel would be a combination of the colors of the input pixels. The pixel correspondence mapping between the input images is not easy to do reliably, especially within regions of homogeneous color. But fortunately, the regions where such pixels project have almost constant color, so a projection error of a few pixels typically does not cause visible artifacts.

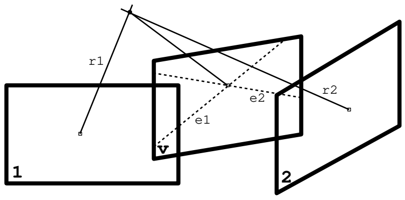

Figure 8: Two matching rays correspond to the pixel of the virtual camera where the projections of the rays intersect.

Chen and Williams [2] used similar methods to trade unbounded scene complexity to bounded image complexity. They render a large number of views of a complicated scene and obtain accurate pixel correspondences from depth values that are stored in addition to the color at each pixel. The missing views needed for a walk-through of the virtual environment are interpolated from the stored ones. Max and Ohsaki [14] used similar techniques for rendering trees from precomputed Z-buffer views. However, rather than morphing the precomputed images, they reproject them pixel by pixel. Shade *et al.* [17] partition the geometric primitives in the scene, render images of them, and texture map the images onto quadrilaterals, which are displayed instead of the geometry. Debevec *et al.* [6] developed a system that fits user-generated geometric models of buildings to digitized images by interactively associating image features with model features and fitting model parameters to images. The buildings are view-dependently texture mapped using the color images. The interpolation between different texture maps is improved by determining more accurate surface geometry using stereo from several input images and morphing the texture map accordingly.

Two recent papers use similar techniques to ours. Mark *et al.* [13] investigate the use of image-based rendering to increase the frame rate for remotely viewing virtual worlds. Their proposed system would remotely render images from geometric models at 5 frames/sec and send them to a local computer that warps and interpolates two consecutive frames at about 60 frames/sec. The 3D warp is done as in [2]. Using the z-values at each pixel a dense triangle mesh is constructed for the two views between which the interpolation is performed. Normal vectors and z-values at each pixel are used to locate false connections across a step edge between an occluding and occluded surface. Darsa *et al.* [5] describe another approach for rapidly displaying complicated environments. The virtual environment is divided into cubes. From the center of each cube, six views (one for each face of the cube) are rendered. Using the z-buffer, the geometry of the visible scene is tessellated into a sparse triangle mesh, which is texture mapped using the rendered color image. A viewer at the center of a cube can simply view the textured polygon meshes stored at the cube walls. If the viewer moves, parts of the scene previously hidden become visible. The textured meshes from several cubes can be used to fill the holes. The authors discuss different weighting schemes for merging meshes from several cubes.

6 Discussion

We have described a new rendering method called view-based rendering that lies in between purely model-based and purely image-based methods. The input to our method is a small set of range and color images, containing both geometric and color information.

An image can be rendered from an arbitrary viewpoint by blending the information obtained from several of these views. This blending operation is accomplished by three weights determined by the view direction of the virtual camera, the surface sampling density and orientation, and the distance from the mesh boundary. As a robust solution to the visibility problem, we propose the use of a soft z-buffering technique to allow only points within a threshold to be included in blending. We have demonstrated interactive viewing of two non-trivial real objects using our method.

Our view-based rendering has several advantages over the traditional model-based approach of rendering full objects. It is much easier to model each view separately than it is to create a model of the whole object, especially if the object has convoluted geometry. Our approach automatically gives view-dependent texturing of the object, which produces more realistic images than can typically be obtained by static texturing.

The advantages over image-based rendering are twofold and are a direct consequence of having explicit geometric information. First, significantly fewer input images are needed for view-based rendering than for image-based rendering. Second, we can construct composite objects from several view-based models. In contrast, realistic composite images can be generated from image-based models only if their bounding boxes do not intersect.

The disadvantage is that our system shows the object in fixed lighting. Relighting of synthetically created view-based models is possible if we store additional information such as normals and reflectance properties for each pixel of the texture maps. For real objects, normals could be approximated but obtaining reflectance properties is not trivial.

References

[1] S. E. Chen. Quicktime VR - an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings*, pages 29–38. ACM SIGGRAPH, Addison Wesley, August 1995.

[2] S. E. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[3] Y. Chen and G. Medioni. Object modelling by registration of multiple range images. *Image and Vision Computing*, 10(3):145–155, April 1992.

[4] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH 96 Conference Proceedings*, pages 303–312. ACM SIGGRAPH, Addison Wesley, August 1996.

[5] L. Darsa, B. C. Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 25–34, April 1997.

[6] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH 96 Conference Proceedings*, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996.

[7] T. Evgeniou. Image based rendering using algebraic techniques. Technical Report A.I. Memo No. 1592, Massachusetts Institute of Technology, 1996.

[8] M. Garland and P. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1995.

[9] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *SIGGRAPH 96 Conference Proceedings*, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996.

[10] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[11] S. Laveau and O. D. Faugeras. 3-d scene representation as a collection of images and fundamental matrices. Technical Report RR 2205, INRIA, France, 1994. Available from ftp://ftp.inria.fr/INRIA/tech-reports/RR/RR-2205.ps.gz.

[12] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996.

[13] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 7–16, April 1997.

[14] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Eurographics Rendering Workshop 1995*, pages 74–81;359–360. Eurographics, June 1995.

[15] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *SIGGRAPH 95 Conference Proceedings*, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995.

[16] K. Pulli, T. Duchamp, H. Hoppe, J. McDonald, L. Shapiro, and W. Stuetzle. Robust meshes from multiple range maps. In *Proc. IEEE Int. Conf. on Recent Advances in 3-D Digital Imaging and Modeling*, May 1997.

[17] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996.

[18] T. Werner, R. D. Hersch, and V. Hlaváč. Rendering real-world objects using view interpolation. In *Proc. IEEE Int. Conf on Computer Vision (ICCV)*, pages 957–962, June 1995.

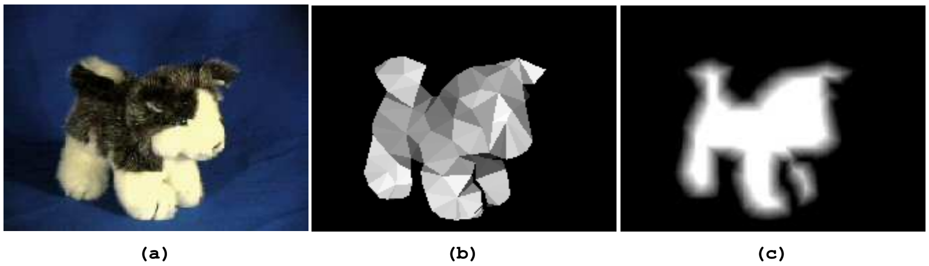

Figure 9 (a) A color image of a toy dog. (b) Weight w_ϕ is applied to each face of the triangular mesh. (c) Weight w_γ smoothly decreases towards the mesh boundary.

Figure 10 (a) The result of combining three views by repeatedly rendering the view-based meshes from the viewpoint of the virtual camera as described in Section 3.1. (b) Using the weights and soft z-buffering described in Section 3.2 produces a much better result.

Figure 11 Our viewer shows the three view-based models rendered from the viewpoint of the virtual camera. The final image is on the bottom right.