# 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. A new method for displaying scanned real objects, called *view-based rendering*, is given. The method takes as input a collection of colored range images covering the object and creates 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. An interactive viewer was tested using real data of non-trivial objects that would be hard to accurately model using traditional model-based methods.

## 1 Introduction

In traditional *model-based rendering*, a geometric model of a scene, together with surface reflectance properties and lighting parameters, are 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 they are 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:

1. 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.

2. 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. The obvious solution is to create a model of the object capturing its shape and surface reflectance properties. However, generating realistic models of 3D objects is a nontrivial problem which 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 models at each point of this surface using the colors observed in the various images. Despite recent advances [5, 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 just 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 map 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, our method only requires images from the typically small set of viewpoints from which the range data were captured. 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 covers the previous work. Section 4 describes our view-based rendering approach. Section 5 presents details of our implementation, including data acquisition, view-based model generation, and use of graphics hardware for efficient implementation, and some results. Section 6 discusses some ideas for future work 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 from 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 [13] 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).

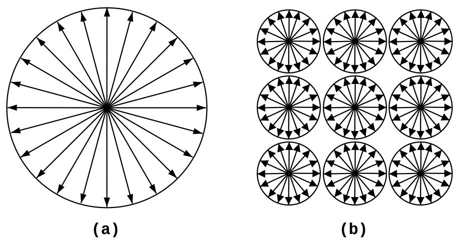

**Figure 1:** (a) A pencil of rays describes the colors of visible points from a given point. (b) The light field function describes the colors of all rays starting from any point.

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 a then a function

$$f : \mathbb{R}^3 \times S^2 \rightarrow \mathbb{R}^3.$$

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 obviously 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. 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.* [10], 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.

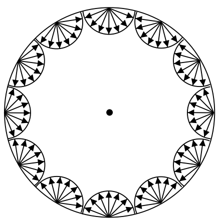

**Figure 2:** A spherical lumigraph surface.

The lumigraph contains all of the information 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 (or more precisely, 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 one looks for rays close to the one associated with some particular pixel. In the next two sections we examine two closeness measures. It is quite obvious that one should concentrate on sample rays that intersect the object surface close to where the query ray does. However, as we will show, merely considering distance between intersection points of rays with the object surface is only justified for a flat Lambertian surface, and in general ray direction should also be considered.

It is convenient to represent each image as the restriction of the light field function to a pencil of inward pointing rays based at a point. Thus we are given a collection $P_i$, $i = 1, \ldots, m$ of pencils of inward pointing rays, based at $m$ camera locations exterior to the lumigraph surface surrounding the object; and we know the values of $f$ on each pencil. Our goal is to estimate $f(\mathbf{x}, \hat{\mathbf{n}})$ for an inward pointing query ray based at a point $\mathbf{x}$ exterior to the lumigraph surface. We estimate $f(\mathbf{x}, \hat{\mathbf{n}})$ as a weighted average of values of $f$ on the rays of $P_i$ "near" $(\mathbf{x}, \hat{\mathbf{n}})$.

### 2.1.1 Ray-surface intersection

We first consider the case where the surface geometry is known, as illustrated in Figure 3(a). Let $\mathbf{x}_0$ denote the point where the query ray $(\mathbf{x}, \hat{\mathbf{n}})$ intersects the surface. To estimate $f(\mathbf{x}, \hat{\mathbf{n}})$, we locate those rays of $P_i$ that first intersect the object at $\mathbf{x}_0$. For a flat Lambertian surface, the value of $f$ at any of these rays gives a reasonably good approximation of $f(\mathbf{x}, \hat{\mathbf{n}})$, and we can recover $f$ provided at least one sample ray intersects the object surface at $\mathbf{x}_0$.

Figure 3(b) shows a case where the precise object geometry is not known, but we have an estimate of the average distance of the object surface from the lumigraph surface. That is, the object is approximated by a sphere, centered in the volume enclosed by the lumigraph surface whose radius is the average distance from the object surface to the center. In this case, we let $\mathbf{x}_0$ denote the intersection of $(\mathbf{x}, \hat{\mathbf{n}})$ with the approximating sphere and choose rays in $P_i$ that first intersect the approximating sphere at $\mathbf{x}_0$. The expected error in our estimate of $f(\mathbf{x}, \hat{\mathbf{n}})$ should now be greater than in case (a). Or, to obtain the same error, we will need many more sample rays (i.e. images).

Figure 3(c) illustrates the case where we have absolutely no information about the object geometry. To estimate $f(\mathbf{x}, \hat{\mathbf{n}})$, we first locate the sample rays which intersect the lumigraph surface $M$ at points closest to the point at which $\mathbf{x}$ intersects $M$, and, among these, we choose the value of $f$ on the sample rays that are parallel to the query ray. If the surface is not oblique to the query ray at the (unknown) intersection

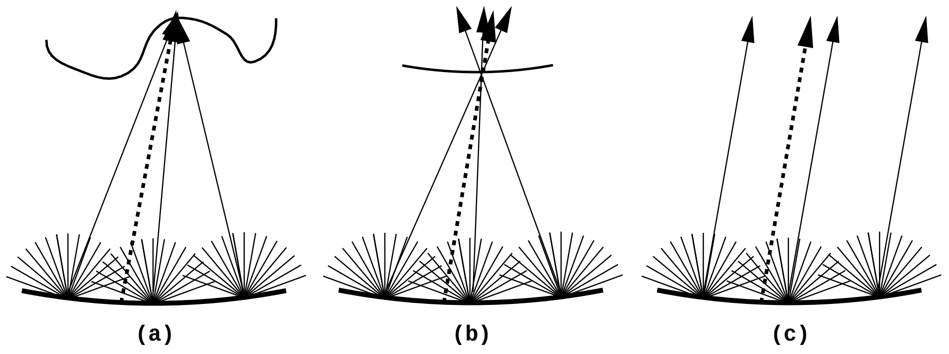

**Figure 3** (a) Choose the rays that intersect at the object surface. (b) Choose the rays that intersect the query ray at the average distance to the object. (c) Parallel rays correspond to an object being infinitely far.

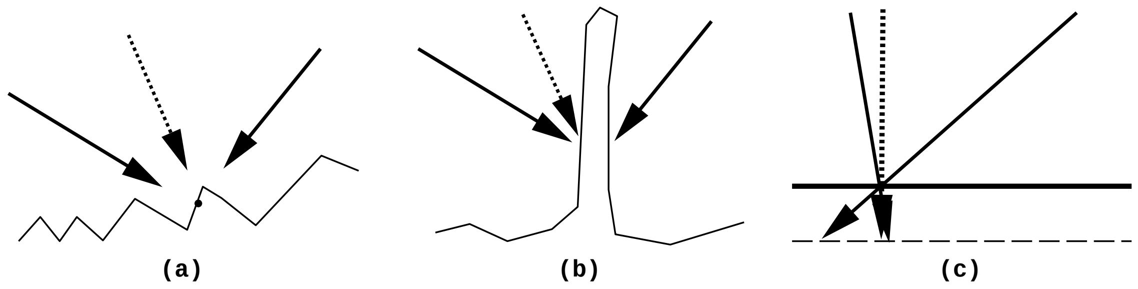

**Figure 4** The query ray is dotted, sample rays are solid. (a) Detailed surface geometry can cause occlusion that make the surface appear different from different directions. (b) Thin features can cause a discrepancy between surface distance and spatial distance of intersection points. (c) The more parallel the rays the less damaging an error in an estimate of surface distance.

point with the object surface, then the selected ray is likely to intersect the object surface at a point near $\mathbf{x}_0$. The more densely the lumigraph stores ray pencils, the smaller the distance between the intersections points with the object surface and so, the smaller the error in our estimate of $f(\mathbf{x}, \hat{\mathbf{n}})$ is likely to be.

### 2.1.2 Ray direction

The estimate of the lighting function can be improved by taking into account the direction and more heavily weighting sample rays whose direction is near that of the query ray. There are three justifications for this claim. 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. This property is often modeled in model-based rendering by the bidirectional reflectance distribution function (brdf); 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. This is shown by the point marked in Figure 4(a). 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. 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. Moreover, if the object has thin features, as illustrated in Figure 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 Figure 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 Previous 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 [13] and Gortler *et al.* [10] independently developed image synthesis systems that are modeled on the 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 [10]. 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 [13] built a system that moves the camera in a regular pattern into a known set of positions. The camera images are then projected back to the planes of the lumigraph cube. Gortler *et al.* [10] moved a hand-held video camera around an object placed on the capture stage. The capture stage was patterned with a set of concentric circles that enabled camera pose estimation for each image. The rays from the images were projected to the lumigraph walls, and the lumigraph was 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 and use it to find sample rays that intersect the object near the query ray. 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 [12] and has since been used in several other systems [15, 18, 8]. 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 Figure 5 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 also have almost constant color, so a projection error of a few pixels typically does not cause visible artifacts.

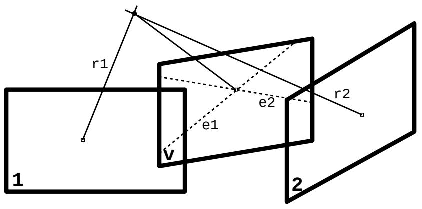

**Figure 5:** Two matching rays correspond to the pixel of the virtual camera where the projections of the rays intersect.

Chen and Williams [2] used similar methods to trade unbounded scene complexity to bounded scene 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. Other systems that simplify geometry using images include the work of Shade *et al.* [17]. They 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.* [7] developed a system that creates geometric models of buildings from digitized images with user interaction. 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.

We defer the discussion of two recent papers [6, 14] to Section 6. They use several of the techniques that we developed for our system, but they were developed simultaneously and independently.

## 4 View-based rendering

The input to our view-based rendering system is a set of color images of the objects. Along with each color image we obtain a range map for the part of the object surface that is visible in the image. Registering the range maps into a common coordinate system gives us the relative camera locations and orientations of the color images with respect to the object. We replace the dense range maps by sparse triangle meshes that closely approximate them. We then texture map each triangle mesh using the associated color image. This preprocessing step is described in more detail in Section 5. To synthesize an image of the object from a fixed viewpoint we individually render the meshes constructed from the three nearest viewpoints and blend them together with a pixel-based weighting algorithm and using soft z-buffering.

### 4.1 A simple approach

To better understand the virtues of our approach, it is helpful to consider a more simple 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 even move the virtual camera around the stored viewpoint by rendering the mesh from the new viewpoint. But as the viewpoint changes, parts of the surface not seen from the original viewpoint may 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 to the additional viewpoints. The resulting image is a collage of several individual images. Because individual meshes are likely to overlap, the compounded errors from the actual range measurements, view registration, and polygonal approximation make it arbitrary which surface is closest to the camera and therefore rendered. Also, the alignment of the color information is not perfect, and there may be additional slight changes in the lighting conditions between the views. These errors cause the unnatural features visible in Figure 10(a).

We can improve on this by giving different weights to the views, with the viewpoint closest to the viewpoint of virtual camera receiving higher weight than the others. The effect of self-occlusion can be minimized by using z-buffering and back face culling when rendering the individual views. Even with these improvements, several problems remain. The pixels where only some of the views contribute appear darker than others. Even if we normalize the colors by dividing the color values by the sum of the weights of the contributing views, changes in lighting and registration errors create visible artifacts at mesh boundaries. There are also problems with self-occlusion. Without z-buffering the color information from surfaces that should be hidden by other surfaces is blended with the color of the visible surfaces, causing parts of the front-most surface to appear partially transparent. A third problem is related to the uniform weighting of the images generated by the meshes. The color and surface geometry is sampled much more densely at surface locations that are perpendicular to the sensor than at tilted surfaces. Additionally, the range information is usually less reliable at tilted surfaces.

## 4.2 Three weights and soft z-buffering

To synthesize an image of the object from a fixed viewpoint, we first select $n$ stored views whose viewing directions roughly agree with the direction from the viewpoint to the object. Each selected textured mesh is individually rendered from this viewpoint to obtain $n$ separate images. The images are blended into a single image by the following weighting scheme. Consider a single pixel. Let $r$ be the red channel value (green and blue are processed in the same manner) associated to it. We set

$$r = \frac{\sum_{i=1}^{n} w_i r_i}{\sum_{i=1}^{n} w_i}$$

where $r_i$ is the color value associated to 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 $w_i = w_{\theta,i} \cdot w_{\phi,i} \cdot w_{\gamma,i}$ , whose definition is illustrated in Figure 6. Self-occlusions are handled by using soft z-buffering to combine the images pixel by pixel.

ing directions of the virtual camera and the stored view. The first weight, $w_\theta$, measures the proximity of the stored view to the current viewpoint, and therefore changes dynamically as the virtual camera moves. Both the appearance of minute geometric surface detail and the surface reflectance change with the viewing direction; the weight $w_\theta$ is designed to favor views with viewing directions similar to that of the virtual camera. Specifically, we use $w_\theta = \max(0, \cos\theta)$, where $\theta$ is the angle between the view-

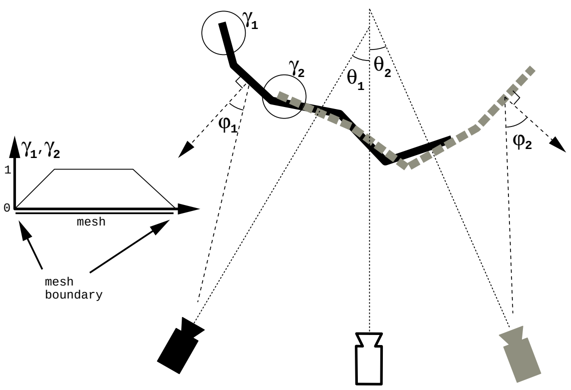

**Figure 6:** The three weights $w_\theta$, $w_\varphi$, $w_\gamma$ used in combining the images. The virtual camera is in the middle, the other cameras have each a textured polygon mesh. Weight $w_\theta$ is the cosine of the angle between the viewing directions of the virtual and real camera, $w_\varphi$ is the cosine of the angle between the normal of the surface and vector from the surface to the sensor, and $w_\gamma$ is a weight that decreases close to the mesh boundary in order to seamlessly blend views together.

The second weight, $w_\varphi$, is a static measure of surface sampling density. As a surface perpendicular to the camera is rotated by an angle $\phi$, the surface area projecting to a pixel increases by $1/\cos\phi$ and the surface sampling density decreases by $\cos\phi$. In our system, a weight $w_\varphi = \vec{n} \cdot \vec{d}$ is applied to each mesh triangle, where $\vec{n}$ is the external unit normal of the triangle and $\vec{d}$ is a unit vector pointing from the centroid of the triangle to the sensor (see Figure 9(b)). The scanning geometry ensures that this value is in the range $(0.0, 1.0]$.

The third weight $w_\gamma$ which we call the *blend weight*, is designed to smoothly blend the meshes at their boundaries. As illustrated by Figure 9 (c), the blend weight linearly increases with distance from the mesh boundary. Like $w_\varphi$, the weight $w_\gamma$ does not depend on the viewing direction of the virtual camera. A similar weight was used by Debevec *et al.* [7].

Most self-occlusions are handled during rendering of individual views using backface 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 7 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 wrongly 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, we ignore the low confidence pixel altogether.

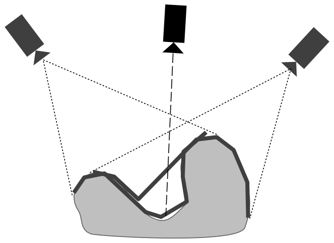
**Figure 7:** Problems with undetected step edges.

## 5 Implementation

### 5.1 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. We use Curless and Levoy's spacetime analysis [4] to more accurately locate the beam at each step. 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. The coordinates are calculated in the sensor coordinate system of the left camera. After the view has been scanned, we turn the lights on, and take a color picture of the object, again with the left camera. 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. The corresponding 3D points are rotated and translated so that the distances between points corresponding to the same features are minimized. The initial registration is refined using Chen and Medioni's registration method [3] that has been modified to deal with multiple data sets simultaneously.

**Triangle mesh creation.** We currently create the triangle meshes manually. 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. When the user adds a vertex, 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 can be classified as background. The adding of vertices is easily automated. For example, Garland and Heckbert [9] 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 will perform a mesh simplification step using the mesh optimization methods by Hoppe *et al.* [11].

## 5.2 Rendering

We have built an interactive viewer for viewing the reconstructed images (see Figure 11). For each frame, we calculate the dot product of the camera viewing directions for the stored views and the viewing direction of the virtual camera. The three views with highest dot product values (the weight $w_\theta$) are then rendered separately from the viewpoint of the virtual camera as textured triangle meshes.

Two of the weights, $w_\varphi$ and $w_\gamma$ are static for each view, they do not depend on the viewing direction of the virtual camera. We can apply both of these weights offline and code them into the alpha channels of the mesh color and the texture map. $w_\varphi$ 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 $(1, 1, 1, w_\varphi)$ to each triangle. $w_\gamma$ 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 8 presents 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.

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 $w_\varphi \cdot w_\gamma$. It is also possible to apply the remaining weight, $w_\theta$, 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 $w_\theta$, the resulting alpha value contains the final weight $w_\theta \cdot w_\varphi \cdot w_\gamma$.

```
FOR EACH pixel
  $z_{min}$          := min_reliable_z( pixel )
  pixel_color    := (0,0,0)
  pixel_weight   := 0
  FOR EACH view
    IF $z_{min}$ <= z[view,pixel] <= $z_{min}$+$thr_{soft\_z}$ THEN
      weight       := $w_\theta$ * $w_\varphi$ * $w_\gamma$
      pixel_color  += weight * color[view,pixel]
      pixel_weight += weight
    ENDIF
  END
  color[pixel] := pixel_color / pixel_weight
END
```

**Figure 8:** Pseudo code for color blending.

## 5.3 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 closest views based on how close their viewing directions are to the direction of the virtual camera. The texture-mapped polygonal approximations of the views are rendered from the current view point separately into $256 \times 256$ windows. The images are combined pixel by pixel into a composite image that uses z-buffer information for self-occlusion and the three weights.

Figure 10 compares the simple approach of Section 4.1 to our view-based rendering method that uses three weights and soft z-buffering (Section 4.2). In Figure 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 Figure 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.

## 6 Discussion

### 6.1 Concurrent work

Two recent papers related to our work will be presented at the 1997 Symposium on Interactive 3D Graphics. This research was independent of and performed concurrently with our own.

Mark *et al.* [14] 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 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. The normal vectors and z-values (computed for each pixel in the reference images during rendering) are used to locate false connections across a step edge between an occluding and occluded surface. Their logic for discarding unreliable pixels on triangles spanning a step edge in favor of pixels from more reliable triangles is very similar to ours.

Darsa *et al.* [6] 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 away from the center, parts of the scene hidden by some nearby objects become visible. The textured meshes from several cubes can be used to fill in the missing data. The authors discuss different weighting schemes for merging meshes from several cubes. Their quality weight is based on the relative orientation of a triangle in the mesh with respect to the center of the camera that created view, and is essentially the same as our $w_\varphi$. They also use another weight related to the distance from the current position to the cube centers, which is different from but analogous to our $w_\theta$.

### 6.2 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. Implementing 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.

### 6.3 Per view weighting

Currently the dynamic weight that we use is $w_\theta$, the cosine of the angle between the viewing directions of the virtual camera and the stored view. If we always combine three stored views, there is another, more natural choice. The stored views can be placed on a unit sphere based on their viewing directions, and the sphere can be tessellated by Delaunay triangulation. For any viewpoint of the virtual camera, the stored views corresponding to the triangle enclosing the point of the virtual camera's viewing direction are selected. The new $w_\theta$'s are the barycentric coordinates of the current viewing direction within that triangle. We are currently investigating this approach.

## 6.4 Summary

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, giving us both geometry and color information. An image can be rendered from an arbitrary viewpoint by blending the information from several of these views. The blending operation is improved by the use of three weights that determine the color value of a given pixel. Soft z-buffering allows only points within a threshold to be included in blending. We have demonstrated interactive viewing of two non-trivial real objects using our method.

## 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. Better optical triangulation through spacetime analysis. In *Proc. IEEE Int. Conf on Computer Vision (ICCV)*, pages 987–994, June 1995.

[5] 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.

[6] 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*, April 1997.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proc. 1997 Symposium on Interactive 3D graphics*, April 1997.

[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.

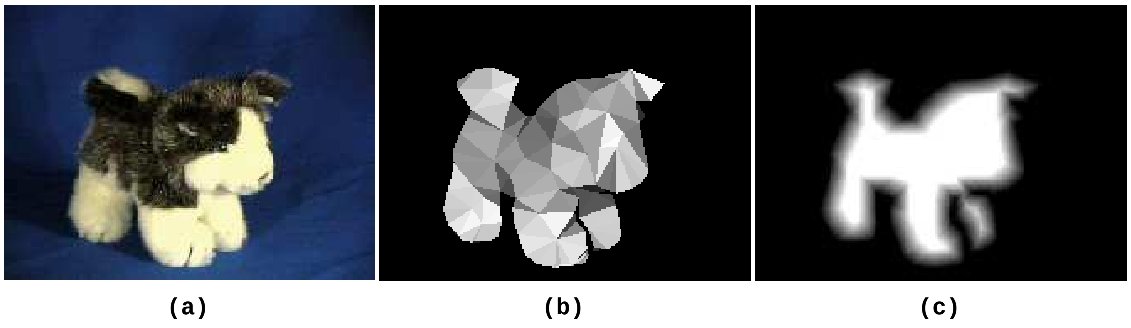

**Figure 9** (a) A color image of a toy dog. (b) Weight $w_\varphi$ is applied to each face of the triangle mesh. (c) Weight $w_\gamma$ smoothly decreases the influence of the view towards the mesh boundaries.

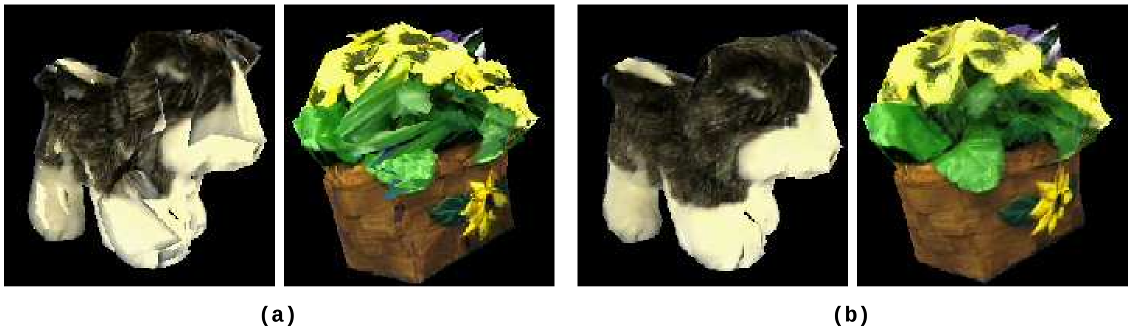

**Figure 10** (a) The result of combining three views by repeatedly rendering the view-based meshes from the viewpoint of the virtual camera as described in Section 4.1. (b) Using the weights and soft z-buffering described in Section 4.2 produces a much better result.

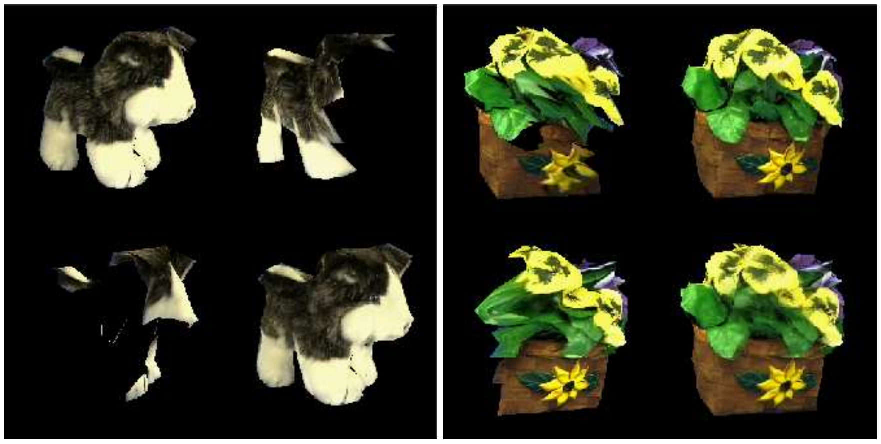

**Figure 11** Our viewer shows the three view-based models rendered from the viewpoint of the virtual camera. The final image is on the bottom right.