Surface modeling and display from range and color data

Kari Pulli[1], Michael Cohen[2], Tom Duchamp[1], Hugues Hoppe[2], John McDonald[1], Linda Shapiro[1], and Werner Stuetzle[1]

[1] University of Washington, Seattle WA, USA
[2] Microsoft Research, Redmond WA, USA

Abstract. Two approaches for modeling surfaces from a collection of range maps and associated color images are covered. The first approach presents a method that robustly obtains a closed mesh that approximates the object geometry. The mesh can then be simplified and texture mapped for display. The second approach does not attempt to create a single object model. Instead, a set of models is constructed, one model for each view of the object. several of the view-based models are rendered separately, and their information is combined in a view-dependent manner for display.

1 Introduction

In this paper we propose two methods for modeling and displaying real objects using a set of range maps with associated color images as input. The first method follows the traditional surface reconstruction approach, where we attempt to create a single surface model that accurately describes the scanned object. Specifically, our algorithm [9] emphasizes robust recovery of the object topology from the input data. Holes due to missing data, i.e., unobserved surface regions, are automatically filled so that the model remains consistent with the input data. This approximate model can then be fitted more accurately to the input data, textured from the color images, and displayed. Our second method shows that it is not necessary to integrate the input data into a single surface model for display purposes. Instead, one can model each view separately and integrate the separate views at display time in the image space. We call this approach *view-based rendering* [8].

Section 2 describes our robust method for modeling object surfaces. Section 3 presents the view-based rendering method. Section 4 concludes the paper.

2 Robust approximate meshes

Overview. Our algorithm processes a cubical volume surrounding all the input data in a hierarchical fashion. For each cube-shaped partition, it checks whether the cube can be shown to be entirely inside or outside of the object. If neither, the cube is subdivided and the same test is recursively applied to the resulting smaller cubes. Our surface approximation is the closed boundary between cubes that lie entirely outside of the object and all the other cubes.

Assumptions. We assume that the range data is expressed as a set of range maps called views. Each view is an image in which each pixel stores a 3D point instead of a color value. Further, we assume that the calibration parameters of the sensor are known so that we can project any 3D point to the image plane of the sensor. We also assume that the line segments between the sensor and each measured point lie entirely outside of the object we are modeling. Finally, we assume that all range views have been registered to a common coordinate system.

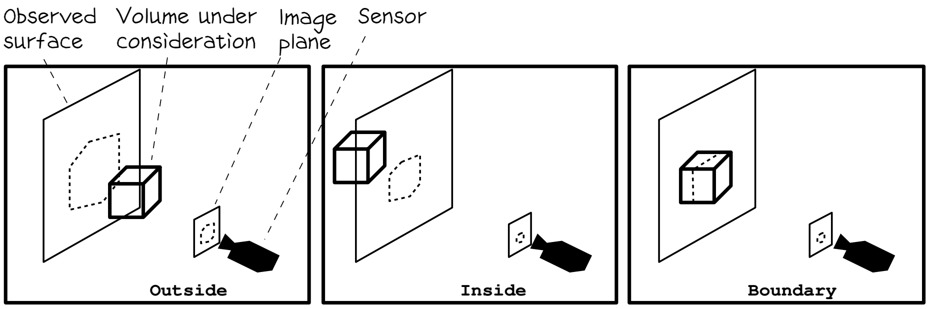

Fig. 1. The three cases of the algorithm. In case 1 the cube is in front of the range data, in case 2 it is entirely behind the surface (with respect to the sensor), while in case 3 the cube intersects the range data.

2.1 Processing a single range view

The initial volume is an axis-aligned cube that fully surrounds all the range data. We use interval analysis to evaluate the volumetric function on the cube. If the cube is neither completely inside nor completely outside the object, we recursively subdivide it into eight smaller cubes, which are added to the octree as children of the current cube. For each of these cubes, we classify their location with respect to the sensor and the range data. A cube can be classified in three ways (see Figure 1):

- In case 1 the cube lies between the range data and the sensor. The cube is assumed to lie outside of the object. It is not processed any further.
- In case 2 the whole cube is behind the range data. As far as this sensor is concerned, the cube will be assumed to lie inside of the object. It will not be further subdivided.
- In case 3 the cube intersects the range map. In this case we subdivide the cube into its eight children and recursively apply the algorithm up to a pre-specified maximum subdivision level. A case 3 cube at the finest level is assumed to be at the boundary of the object.

The cubes are classified as follows. We project the eight corners of the cube to the sensor's image plane, where the convex hull of the points forms a hexagon. The rays from the sensor to the hexagon form a cone, which we truncate so that it just encloses the cube. If all the data points projecting onto the hexagon are behind the truncated cone (i.e., are farther than the farthest corner of the cube from the sensor), the cube is outside. If all those points are closer than the closest cube corner, the cube is inside. Otherwise, it is the boundary case. Possible missing data is treated as points that are very close to the sensor. If we can label parts of the depth map as background, data at those locations are treated as being infinitely far away.

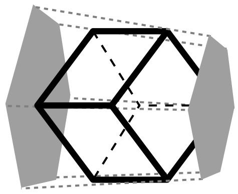

Fig. 2: An octree cube and its truncated cone.

Our labeling method is simple, but conservative. For instance, if all the points projecting onto the hexagon are actually behind the cube, but some of them are inside the truncated cone, we would erroneously label the cube to intersect the surface. This, however, is not a problem because the next subdivision level will most likely determine that all the children of the cube lie in the exterior of the object and therefore remove them.

2.2 Generalizations to multiple views

If multiple views are available, we have a choice of two processing orders. We can traverse the whole octree once and use the consensus of all the views to determine the labeling for each cube (simultaneous processing), or we can process one view at a time, building on the results of the previously processed views (sequential processing).

In simultaneous processing, we traverse the octree as we did in the case of a single view. However, the cube labeling process changes slightly.

- A cube is labeled inside the object only if it would be labeled inside with respect to each single view.
- A cube is labeled outside if it would be labeled outside with respect to any single view.
- Otherwise, the cube is labeled boundary and is further subdivided, unless the maximum subdivision level has been reached.

We use sequential processing if we later obtain a new view that we want to integrate into a previously processed octree. We recursively descend the octree and perform the occlusion test for each cube that has not been determined to lie outside of the object. If the new view determines that a cube is outside, it is relabeled and the subtrees below it are removed. Similarly, a boundary label overrides a previous inside label, in which case the cube's descendants must be recursively tested, potentially up to the maximum subdivision level.

Although both processing orders produce the same result, the simultaneous processing order is in general faster [10]. In sequential processing the silhouette of the object often creates a visual cone (centered at the sensor) that separates volumes known to be outside from those speculated to be inside. The algorithm would have to recurse up to the finest subdivision level to accurately determine this boundary. In simultaneous processing, however, another view could determine at a rather coarse level of subdivision that at least part of that boundary is actually outside of the object, and the finer levels of the octree for that subvolume need never be processed.

2.3 Mesh extraction

The labeling in the octree divides the space into two sets: the cubes known to lie outside of the object and the cubes that are assumed to be part of the object. Our surface estimate will be the closed boundary between these sets. This definition allows us to create a plausible surface even at locations where we failed to obtain data [1]. The boundary is represented as a collection of vertices and triangles that can be easily combined to form a mesh.

The octree generated by the algorithm has the following structure: outside cubes and inside cubes do not have any children, while the boundary cubes have a sequence of descendants down to the finest subdivision level. We traverse the octree starting from the root. At an outside cube we do nothing. At a boundary cube that is not at the finest level, we descend to the children. If we reach the maximum subdivision level and the cube is either at the boundary or inside we check the labeling of the six neighbors. If a neighboring cube is an outside cube, we create two triangles for the face they share. In an inside cube that is not at the maximum subdivision level, we check whether it abuts with an outside cube, and in such case create enough triangles (of same size as the ones created at the finest level) to cover the shared part of the face. The triangles are combined into a closed triangle mesh.

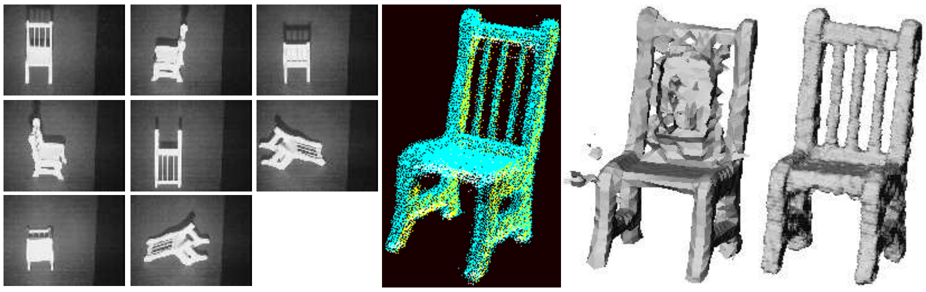

Fig. 3. Eight views of a chair data set, registered points, result from using a previous method, result from our method.

2.4 Reconstruction results

We have tested our method with both real and synthetic data. One of the real data sets consisting of eight views of a miniature chair is shown in Fig. 3, along with the data points, and failed surface reconstruction from another algorithm [5] that was our original motivation for this method. Even though we have cleaned the data and removed most of the outliers, some noisy measurements close to the surface remain, especially between the spokes of the back support of the chair. The algorithm from [5] works with an unorganized point cloud and does not use any extra knowledge (such as viewing directions, etc.) in addition to the points. It works quite nicely if the data does not contain outliers and uniformly samples the underlying surface. Unfortunately real data, such as this data set, often violate both of those requirements. In contrast, our method is able to correctly recover the topology of the chair as shown in the rightmost image.

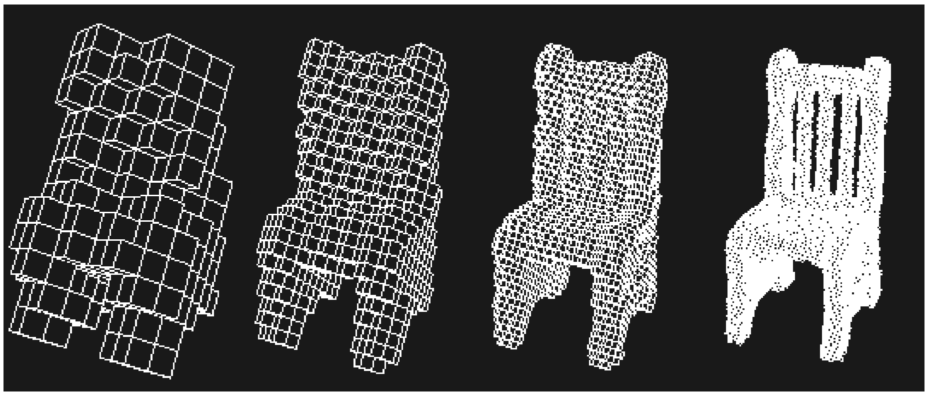

Fig. 4. Chair after 4, 5, 6, and 7 subdivisions.

Figure 4 shows intermediate results of our method using the chair data set, displaying the octree after 4, 5, 6, and 7 subdivisions. The final mesh in Fig. 3 was obtained from the level 7 octree. We smooth the mesh before displaying using Taubin's method [11]. Notice that the spokes and the holes between them have been robustly recovered despite the large number of outliers and some missing data. This is partially due to the implicit removal of outliers that the algorithm performs: in most cases, there is a view that determines that the cube containing an outlier lies outside of the object. Fig. 5 also shows the results from a synthetic data set. The smoothed result appears above, while the result after applying Hoppe's mesh optimization algorithm [6] appears below.

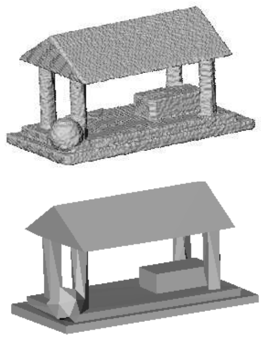

Fig. 5: Simulated temple data set, our results and a simplified mesh.

2.5 Discussion

The simplicity of the algorithm leads to a fast implementation. This again allows interactive selection of the subdivision level. For example, with the chair data set one could first subdivide the octree six times. Within ten seconds or so the user can see the current approximation, which is not good enough since all the holes in the object haven't been opened yet. An extra subdivision step (25 sec. later) shows that now the topology of the current approximation agrees with that of the real object.

It is important to notice that holes often can be correctly detected and modeled using only indirect evidence, i.e., using the fact that the scanner can see through the hole. Curless and Levoy suggest using a backdrop [1], placing planes behind the holes that can be detected. However, all range scanning methods based on optical triangulation have limits on how narrow cavities can be scanned. Better results can be obtained if a color (or even intensity) image can be obtained with the range scan (typically easily available with scanners using optical triangulation), the background may be segmented out using image segmentation methods, or even user interaction.

Some methods that employ signed surface distance functions are unable to correctly reconstruct thin objects. Curless and Levoy [1], for example, build a distance function by storing positive distances to voxels in front of the surface and negative distances to voxels behind the surface. In addition to the distance, weights are stored to facilitate combining data from different views. With this method, views from opposite sides of a thin object interfere and may cancel each other, preventing reliable extraction of the zero set of the signed distance function. Our volumetric method carves away the space between the sensor and the object and does not construct a signed distance function. In case of a thin sheet of paper, our algorithm would construct a thin layer of octree cubes (voxels) straddling the paper.

3 View-based rendering

Rendering statically textured surface models produces images that are much less realistic than photographs, which can capture intricate geometric texture and global illumination effects with ease. This is one of the main reasons why image-based rendering algorithms have become popular. This section proposes a new rendering method that does not require creating 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 [4,7], 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 section describes our new *view-based rendering* algorithm and shows results on non-trivial real objects.

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. 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 using soft z-buffering.

3.1 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 with the additional viewpoints. The resulting image is a collage of several individual images. Because individual meshes are likely to overlap, the compound errors from the actual range measurements, view registration, and polygonal approximation make 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 6(a).

(a) (b)

Fig. 6. (a) The result of combining three views by repeatedly rendering the view-based meshes from the viewpoint of the virtual camera. (b) Using the weights and soft z-buffering produces a much better result.

We can improve on this by giving different weights to the views, with the viewpoint closest to the viewpoint of the 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.

In the next section we describe how we can produce much better images (see Fig. 6(b)) using a more sophisticated approach.

3.2 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 described 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 Figs. 7 and 8. Self-occlusions are handled by using soft z-buffering to combine the images pixel by pixel.

The first weight, w_θ, 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 details and the surface reflectance change with the viewing direction; the weight w_θ is designed to favor views with viewing directions similar to that of the virtual camera. Figure 7 illustrates how w_θ is calculated. All the views are placed on a unit sphere, and the sphere is triangulated. The current viewpoint is placed onto that sphere, and we determine which triangle contains it. Only the views corresponding to the corners of that triangle will get a nonzero w_θ. Specifically, we calculate the barycentric coordinates for the current view point within the triangle. Each of the three components of the barycentric coordinates corresponds to a corner of the triangle, and the w_θ of the corner views is the corresponding component. The three w_θ's are all in the range $[0.0, 1.0]$ and they sum up to 1.0.

The second weight, w_φ, is a static measure of surface sampling density. As a surface perpendicular to the camera is rotated by an angle ϕ, the surface area projecting to a pixel increases by $1/\cos\phi$ and the surface sampling density

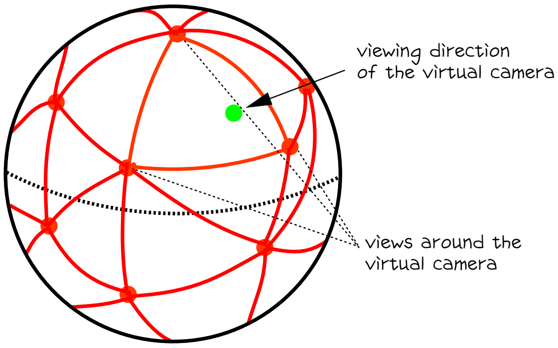

Fig. 7. The views are placed on a unit sphere based on their viewing directions, and the sphere is triangulated using the views as triangle vertices. The views corresponding to the corners of the triangle surrounding the direction of the virtual camera are used, and their weights w_θ are the barycentric coordinates the virtual camera within that triangle.

decreases by $\cos\phi$. In our system, a weight $w_\varphi = \mathbf{n} \cdot \mathbf{d}$ is applied to each mesh triangle, where $\mathbf{n}$ is the external unit normal of the triangle and $\mathbf{d}$ is a unit vector pointing from the centroid of the triangle to the sensor. Figure 8(b) shows w_φ with gray level encoding: the lighter the triangle, the higher w_φ. The scanning geometry ensures that this value is in the range $(0.0, 1.0]$.

The third weight w_γ which we call the *blend weight*, is designed to smoothly blend the meshes at their boundaries. As illustrated by Figure 8 (c), the blend weight linearly increases with distance from the mesh boundary. Like w_φ, the weight w_γ does not depend on the viewing direction of the virtual camera. A similar weight was used by Debevec *et al.* [2].

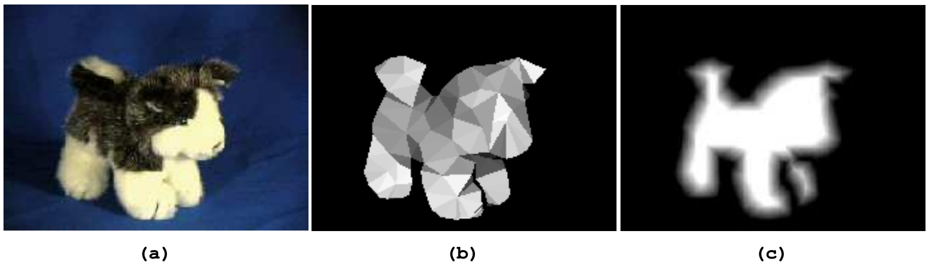

Fig. 8. (a) An image of a toy dog. (b) Weight w_φ is applied to each face of the triangle mesh. (c) Weight w_γ smoothly decreases the influence of the view towards the mesh boundaries.

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 9 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.

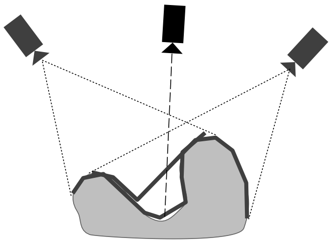

Fig. 9: Problems with undetected step edges.

3.3 Implementation

Triangle mesh creation. We currently create the triangle meshes manually. For each view, 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. The segmentation of the image into object and background is facilitated by placing a blue cloth to the background and scanning the empty scene. Points whose position and color match the data scanned from an empty scene can be classified as background. The adding of vertices is easily automated. For example, Garland and Heckbert [3] 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. For this reason we will perform a mesh simplification step using the mesh optimization methods by Hoppe *et al.* [6].

Rendering. We have built an interactive viewer for viewing the reconstructed images (see Figure 10). 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_θ) are then rendered separately from the viewpoint of the virtual camera as textured triangle meshes.

Fig. 10. Our viewer shows the three view-based models rendered from the viewpoint of the virtual camera. The final image is on the bottom right.

Two of the weights, w_φ and w_γ 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. 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 $(1, 1, 1, w_\varphi)$ 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 11 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

```
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        := wθ * wφ * wγ
      pixel_color  += weight * color[view,pixel]
      pixel_weight += weight
    ENDIF
  END
  color[pixel] := pixel_color / pixel_weight
END
```

Fig. 11: Pseudo code for color blending.

is $w_\varphi \cdot w_\gamma$. It is also possible to apply the remaining weight, w_θ, using graphics hardware. After we render the views, we have to read in the information from the frame buffer. Many graphics libraries, such as OpenGL, allow scaling each pixel while reading the frame buffer into memory. If we scale the alpha channel by w_θ, the resulting alpha value contains the final weight $w_\theta \cdot w_\varphi \cdot w_\gamma$.

3.4 Discussion

View-based rendering takes a step from model-based rendering towards image-based rendering. The advantage as compared to model-based rendering is that making view-based models is much easier than making full models for a large class of objects, such as the flower basket illustrated in Figs. 6 and 10. Additionally, it provides view-dependant texturing of the object, which enables one to create believable impressions of fine geometric detail through texturing without actually having to model the fine geometry. Using static texturing the illusion breaks once the object is rotated to a different angle from where the fine detail was originally seen. The advantage of view-based rendering as compared to image-based rendering methods [4,7] is that a much sparser sample set of images has to be obtained and less data has to be stored, since the geometric information enables us to view the same data from a continuous set of viewpoints close to the one where the view was really taken.

We have demonstrated interactive viewing of our reconstructed models from arbitrary viewpoints at speeds of up to eight frames per second.

4 Conclusion

We have presented two alternative approaches for surface reconstruction and display using range and color data. One of the methods involves creating a single object model that can then be viewed from an arbitrary viewpoint, the other method creates a set of view-based models that are texture mapped using color images, and composites a few of the models in image space to a final image. We are currently working on view-based texturing of complete surface models, so we can better compare the relative advantages of these two competing approaches.

References

1. B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH '96*, pages 303–312, August 1996.
2. 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.
3. 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.

4. 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.
5. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH '92*, pages 71–78, July 1992.
6. 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.
7. M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996.
8. K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, L. Shapiro, and W. Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. Technical Report UW-CSE-97-04-01, Univ. of Washington, Seattle WA 98105, 1997. Available through ftp://ftp.cs.washington.edu/tr/1997/04/UW-CSE-97-04-01.d.
9. K. Pulli, T. Duchamp, H. Hoppe, John McDonald, L. Shapiro, and W. Stuetzle. Robust meshes from multiple range maps. In *Proc. IEEE Int. Conf. on 3-D Imaging and Modeling*, May 1997.
10. R. Szeliski. Rapid octree construction from image sequences. *CVGIP: Image Understanding*, 58(1):23–32, July 1993.
11. G. Taubin. A signal processing approach to fair surface design. In *Proceedings of SIGGRAPH '95*, pages 351–358, August 1995.