Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping

Paul Debevec, Yizhou Yu, and George Borshukov
Univeristy of California at Berkeley
debevec@cs.berkeley.edu

Abstract. This paper presents how the image-based rendering technique of view-dependent texture-mapping (VDTM) can be efficiently implemented using projective texture mapping, a feature commonly available in polygon graphics hardware. VDTM is a technique for generating novel views of a scene with approximately known geometry making maximal use of a sparse set of original views. The original presentation of VDTM in by Debevec, Taylor, and Malik required significant per-pixel computation and did not scale well with the number of original images. In our technique, we precompute for each polygon the set of original images in which it is visibile and create a "view map" data structure that encodes the best texture map to use for a regularly sampled set of possible viewing directions. To generate a novel view, the view map for each polygon is queried to determine a set of no more than three original images to blended together in order to render the polygon with projective texture-mapping. Invisible triangles are shaded using an object-space hole-filling method. We show how the rendering process can be streamlined for implementation on standard polygon graphics hardware. We present results of using the method to render a large-scale model of the Berkeley bell tower and its surrounding campus enironment.

1 Introduction

A clear application of image-based modeling and rendering techniques will be in the creation and display of realistic virtual environments of real places. Acquiring geometric models of environments has been the subject of research in interactive image-based modeling techniques, and is now becoming possible to perform with time-of-flight laser range scanners. Photographs can be taken from a variety of viewpoints using digital camera technology. The challenge, then, is to use the recovered geometry and the available real views to generate novel views of the scene quickly and realistically.

A desirable quality of such a rendering algorithm is to make judicious use of all the available views, including when a particular surface is seen from different directions in several images. This problem was addressed in [2], which presented view-dependent texture mapping as a means to render each pixel in the novel view as a blend of its corresponding pixels in the original views. However, the technique presented did not guarantee smooth blending between images as the viewpoint changed and did not scale well with the number of available images.

In this paper we adapt view-dependent texture mapping to guarantee smooth blending between images, to scale well with the number of images, and to make efficient use of polygon texture-mapping hardware. The result is an effective and efficient technique for generating virtual views of a scene when:

- A geometric model of the scene is available

- A set of calibrated photographs (with known locations and known imaging geometry) is available
- The photographs are taken in the same lighting conditions
- The photographs generally observe each surface of the scene from a few different angles
- Surfaces in the scene are not extremely specular

2 Previous Work

Early image-based modeling and rendering work [12, 5, 7], presented methods of using image depth or image correspondences to reproject the pixels from one camera position to the viewpoint of another. However, the work did not concentrate on investigating how to combining appearance information from multiple images to optimally produce novel views.

View-Dependent Texture Mapping (VDTM) was presented in [2] as a method of rendering interactively constructed 3D architectural scenes using images of the scene taken from several locations. The method attempted to make full use of the available imagery in novel view generation using the following principle: to generate a novel view of a particular surface patch in the scene, the best original image from which to sample reflectance information is the image that observed the patch from as close a direction as possible as the desired novel view. As an example, suppose that a particular surface of a building is seen in three original images from the left, front, and right. If one is generating a novel view from the left, one would want to use the surface's appearance in the left view as the texture map. Similarly, for a view in front of the surface one would most naturally use the frontal view. For an animation of moving from the left to the front, it would make sense to smoothly blend between the left and front texture maps during the animation in order to prevent the texture map suddenly changing from one frame to the next. As a result, the view-dependent texture mapping approach allowed renderings to be considerably more realistic than static texture-mapping allowed, since it better represented non-diffuse reflectance and can simulate the appearance of unmodeled geometry.

The implementation of VDTM in [2] computed texture weighting on a per-pixel basis, required visibility calculations to be performed at rendering time, examined every original view to produce every novel view, and only blended between the two closest viewpoints available. As a result, it was computationally expensive and did not always guarantee the image blending to vary smoothly as the viewpoint changed. This paper uses visibility preprocessing, polygon view maps, and projective texture mapping to overcome these limitations.

Other image-based modeling and rendering work has addressed the problem of blending between available views of the scene in order to produce renderings. In [6], blending was performed amongst a dense regular sampling of images in order to generate novel views. Since scene geometry was not used, a very large number of images was necessary in order to produce relatively low-resolution renderings. [4] was similar to [6] but used approximate scene geometry derived from object silhouettes. Both of these methods restricted the viewpoint to be outside the convex hull of an object or inside a convex empty region of space. The number of images necessary and the restrictions on navigation do not particularly recommend these methods for acquiring and navigating through a large environment, at least without specialized equipment. The work in this paper leverages the results of these methods to render each surface of a model as a light field constructed from a sparse set of views; since the model is assumed to conform well to the scene and the scene is assumed to be mostly diffuse, far fewer images are necessary to achieve good results.

3 Overview of the Method

Our method for VDTM first preprocesses the scene to computer which images saw which polygons from which directions. The preprocessing occurs as follows:

1. Compute Visibility: For each polygon, determine which images it is seen in, splitting polygons that are partially seen in one of the images.
2. Fill Holes: For each polygon not seen in any view, choose vertex colors for performing Gouraud shading.
3. Construct View Maps: For each polygon, store its closest viewing angle for each direction of a regularized viewing hemisphere.

The rendering loop is organized as follows:

1. Draw all polygons seen in none of the original views using the vertex colors determined during hole filling.
2. Draw all polygons which are seen in just one view.
3. For polygon seen in more than one view, calculate its viewing direction for the desired novel view. Calculate where the novel view falls within the view map, and then determine the three closest viewing directions and relative weights. Render the polygon using alpha-blending of the three textures with projective texture mapping.

4 Projective Texture Mapping

To take advantage of current graphics hardware, we make use of projective texture mapping. Projective texture mapping was introduced in [8] and is now part of the OpenGL graphics standard. Although the original paper used it only for shadows and lighting effects, it is extremely useful in image-based rendering because it can simulate the inverse projection of taking photographs with a camera. In order to do projective texture mapping, the user needs to specify a virtual camera position and orientation, and a virtual image plane with the textures. The texture is then cast onto a geometric model using the camera position as the center of projection. In later sections we will adapt projective texture-mapping to take advantage of multiple images of the scene (View-Dependent Texture-Mapping).

For a fixed image, we only want to map this image onto the polygons visible to the camera position from which the photograph was taken. We should not erroneously map it onto those occluded polygons. The OpenGL implementation of projective texture mapping does not automatically perform such visibility checks. It instead lets the texture pierce through the geometry and get mapped onto all backfacing and occluded polygons on the path of the ray(Fig. 1). So parts of the geometry that are occluded in the original image still receive valid texture coordinates and are incorrectly texture mapped instead of remaining in shadow. This effect is easily observed in Fig. 1. Thus, we need to obtain visibility information before texture-mapping.

We could solve this visibility problem in image-space using ray tracing or an item buffer. However, such methods would require us to compute visibility in image-space for each frame, which would be computationally expensive and not suited to interactive

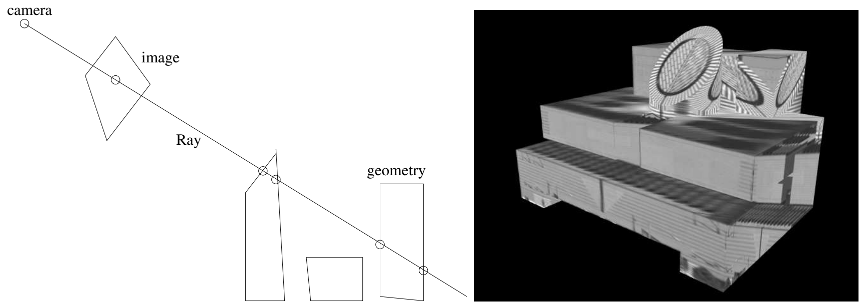


Fig. 1. The current hardware implementation of projective texture mapping in OpenGL lets the texture go through the geometry and get mapped onto all backfacing and occluded polygons on the path of the ray. Viewing the model from a viewpoint far from the original produces artifacts unless proper visibility pre-processing is performed.

applications. Hardware texture-mapping can be done in real-time if all the visibility information is known, which means we need a visibility preprocessing step in object-space to fully exploit the hardware performance. For any scene, this object-space preprocessing needs to be done only once.

In a object-space visibility preprocessing, we should subdivide partially visible polygons so that we only map the current image to their visible parts. Since the whole scene is covered by multiple images which may have overlapping areas and gaps, a polygon may be partially covered by multiple images. We need to clip the polygon against the image boundaries so that different parts of the polygon get textures from different images.

5 Determining Visibility

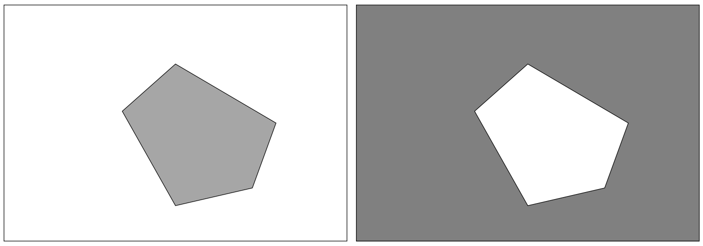


Fig. 2. Users can pick a desired texture region from each image by drawing a convex polygon inside the image frame. The chosen texture region may be either the interior or the exterior of the polygon.

It is desirable to allow users to pick a part of each image as the texture in texture mapping instead of forcing them to use every pixel from each photograph. In our algorithm, a convex planar polygon can be specified inside each image and the user can pick either the interior or the exterior of this polygon as the desired texture(Fig. 2). This gives rise to the necessity to clip polygons in the scene against the edges of this texture region.

For rendering performance, we wish to minimize the number of polygons resulting from visibility processing. Traditional object-space algorithms for hidden surface removal [3, 11] often generate too many polygons or run very slowly. We propose an efficient visibility algorithm for the above purposes.

This algorithm operates in both image space and object space to get better performance. It is summarized as follows:

1. Give each original polygon an id number. If a polygon is subdivided later, all the smaller polygons generated share the same original id number.
2. If there are intersecting polygons, subdivide them along the intersecting line.
3. Clip the polygons against all image boundaries and user-specified planar polygons so that any resulting polygon is either totally inside or totally outside the desired texture regions.
4. Set each camera position as the viewpoint in turn, Z-buffer the original large polygons from the geometric model using their id numbers as their colors.
5. At each camera position, uniformly sample each frontfacing polygon and project these sample points onto the image plane. Retrieve the polygon id at each projection of the sample points from the color buffer. If the retrieved id is different from the current polygon id, a potentially occluding polygon is found and it is tested in object-space whether they are coplanar and whether it is really an occluding polygon.
6. Clip each polygon with its list of occluders in object-space.
7. Associate with each polygon a list of photographs to which it is totally visible.

Using identification (id) numbers to retrieve objects from Z-buffer is similar to the item buffer technique introduced in [10]. The image-space steps in the algorithm can quickly obtain the list of occluders for each polygon.

The objective of this algorithm is to minimize the number of polygons resulted from clipping to accelerate texture mapping at a later stage while safely detecting all occluding polygons so that texture mapping is done correctly. In the above algorithm, safe detection is enhanced by not only checking the pixels at the projections of the sample points on each polygon, but also checking the pixels in a neighborhood of each projection.

5.1 Polygon Shallow Clipping

The method of clipping a polygon against image boundaries is the same as that of clipping a polygon against a real occluding polygon. In either case, we should form a pyramid for the occluding polygon or image frame with the apex at the camera position(Fig. 3(a)), and then clip the polygon with the bounding faces of the pyramid. Before clipping with each bounding face, we should also verify if that bounding face really intersects the polygon.

Our algorithm does *shallow clipping* in the sense that if polygon A occludes polygon B, we only use A to clip B, and any polygons behind B are unaffected(Fig. 3(b)). Only partially visible polygons are clipped. Those totally invisible ones are left intact. This greatly helps minimize the number of resulting polygons.

If a polygon P has a list of occluders $O = \{p_1, p_2, ..., p_m\}$, we use a recursive approach to do the clipping: Obtain the overlapping area on the image plane between each member of O and polygon P and choose the polygon p in O with maximum overlapping area to clip P into two parts P' and S where P' is the part of P that is occluded by p, and

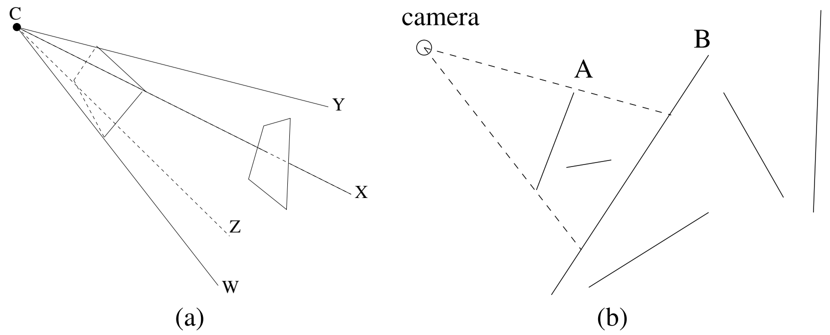


Fig. 3. (a) To clip a polygon against an occluder, we need to form a pyramid for the occluder with the apex at the camera position, and then clip the polygon with the bounding faces of the pyramid. (b) Our algorithm does *shallow clipping* in the sense that if polygon A occludes polygon B, we only use A to clip B, and any polygons behind B are unaffected.

S is a set of convex polygons which make up the part of P not occluded by p. Recursively apply the algorithm on each member of S, i.e. first detect its occluders and then do clipping.

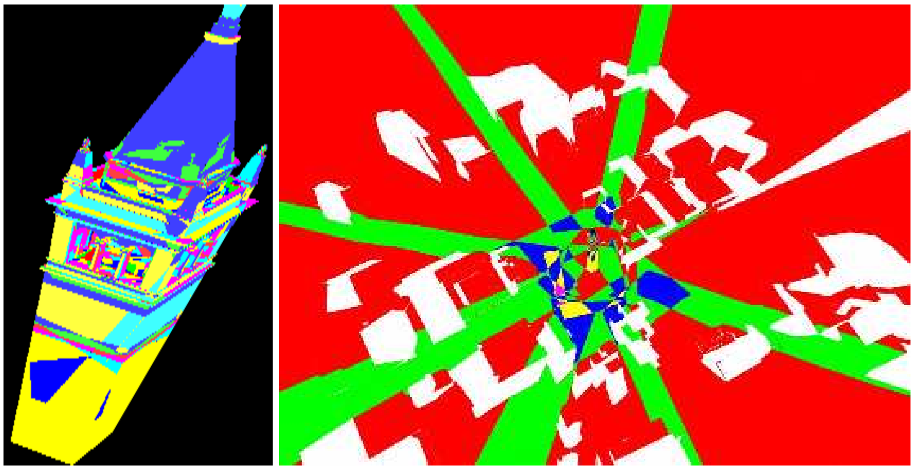


Fig. 4. Visibility results for a bell tower model with 24 camera positions and for a university campus model with 10 camera positions. The shade of each polygon encodes the number of camera positions from which it is visible.

5.2 Thresholding Polygon Size

By experiments, we found most polygons resulting from clipping are tiny polygons. To further reduce the number of polygons, we set a threshold on the size of polygons. If the object-space area of a polygon is below the threshold, it is not subdivided any more and is assigned a constant color based on the textures on its surrounding polygons. If a polygon is very small, it is not noticable whether it has a texture on it or just a constant color. The rendered images can still maintain good quality.

Fig. 4 shows visibility processing results for two geometric models.

6 Hole Filling

No matter how many photographs we have, there still might be some polygons invisible to all cameras. Unless some sort of coloring is assigned to them, they will appear as black holes as we move the view point away from the positions where we took the photographis. This is an inherent problem in projective texture-mapping. One possible solution is to compute an aspect graph of the whole scene, which is computationally expensive, and obtain the minimum number of camera positions we need to have the whole scene covered. However, the required number of camera positions may be too large for a complex scene and it may be impossible to take photographs from some of the calculated camera positions.

Instead of taking more and more photographs, we decided to compose some colors for those black holes from its surrounding area, a precessed called *hole filling*. Previous hole-filling algorithms [12, 2] have operated in image space, which can cause flickering in animations since the manner of the hole filling will change with each frame. Object-space filling can guarantee the color for each invisible polygon is consistent at different frames. By doing this, we can guarantee those invisible polygons are filled with some colors close to the colors of their surrounding visible polygons and they will become less noticeable.

Fig. 5. The image on the left has some black regions which are invisible to all the cameras. The image on the right shows the rendering result with all the holes filled. See also Fig. 9.

The steps in hole filling are:

1. Determine polygon connectivity. At each shared vertex, set up a linked list for those polygons sharing that vertex. In this way, from a polygon, we can access all its neighboring polygons.
2. Sample a color for each visible polygon by projecting its centroid onto the image planes and sampling the colors there.
3. Iterative step: for each invisible polygon, if it has not been filled with some colors, each of its vertices is assigned the color of the closest polygon which is visible or has been color-filled in the previous iterations.

The reason to have an iterative step is that an invisible polygon may not have a visible polygon in its neighborhood. So its filling color should not be decided until some of its neighboring polygons are filled with some colors.

There may be slight misalignment between the geometry and the photographs. Therefore, the textures of the edges of some objects may be projected onto the background. This can only occur at the boundaries of visible and invisible areas. In order not to fill invisible areas with these incorrect textures, we avoid sampling colors at places close to those boundaries.

Fig. 5 shows the result for holefilling. The invisible polygons, filled will Gouraud-shaded low-frequency image content, are largely unnoticeable in animations.

7 Constructing and Querying Polygon View Maps

The goal of view-dependent texture-mapping is to always use surface appearance information sampled from the images which observed the scene closest in angle to the novel viewing angle. As such, the effects of specular reflectance and incorrect model geometry will be minimized. Note than in any particular virtual novel view, different surfaces in the scene may have different "best views"; an obvious case of this is when the novel view encompasses an area not entirely observed in any one view.

In order to avoid the perceptually distracting effect of surfaces suddenly switching between different best views as the user navigates through the scene, we wish to blend between the available views as the angle of view changes. This section shows how for each polygon we will create a *view map* that encodes how to blend between at most three available views for any given novel viewpoint, with guaranteed smooth image weight transitions as the viewpoint changes. The view map for each polygon takes little storage and is simple to compute as a preprocessing step. A view maps may be queried very efficiently given a desired novel viewpoint to return the set of images with which to texture-map the polygon and their relative weights.

To build a polygon's view map, we construct a local coordinate system for the polygon that represents the space of all viewing directions. We then regularly sample the set of viewing directions, and assign to each of these samples the closest original view in which the polygon is visible. These view maps are stored and used at rendering time to determine three best original views and their blending factors by a quick look-up based on current viewpoint.

The local coordinate system, in which the set of viewing directions for each polygon is represented, is constructed according to equation 1.

$$\begin{aligned} \mathbf{x} &= \begin{cases} \mathbf{y}^{\mathbf{W}} \times \mathbf{n} & \text{, if } \mathbf{y}^{\mathbf{W}} \text{ and } \mathbf{n} \text{ are not collinear,} \\ \mathbf{x}^{\mathbf{W}} & \text{otherwise} \end{cases} \\ \mathbf{y} &= \mathbf{n} \times \mathbf{x} \end{aligned} \tag{1}$$

where $\mathbf{x}^{\mathbf{W}}$ and $\mathbf{y}^{\mathbf{W}}$ are world coordinate system axes, and $\mathbf{n}$ is the triangle unit normal.

In order to represent a viewing direction in the local coordinate system we use the following mapping. It is constructed as seen in Fig. 6. We first obtain $\mathbf{v}$, the unit vector in the direction from the polygon centroid $\mathbf{c}$ to the original view position. We then rotate this vector into the $\mathbf{x} - \mathbf{y}$ plane of the local coordinate system for the polygon.

$$\mathbf{v}_r = (\mathbf{n} \times \mathbf{v}) \times \mathbf{n} \tag{2}$$

This vector is then scaled by the arc length $l = \cos^{-1}(\mathbf{n}^T\mathbf{v})$ and projected onto the $\mathbf{x}$ and $\mathbf{y}$ axes giving the desired view mapping.

$$
\begin{aligned}
x &= (l\mathbf{v}_r)^T\mathbf{x} \\
y &= (l\mathbf{v}_r)^T\mathbf{y}
\end{aligned}
\tag{3}
$$

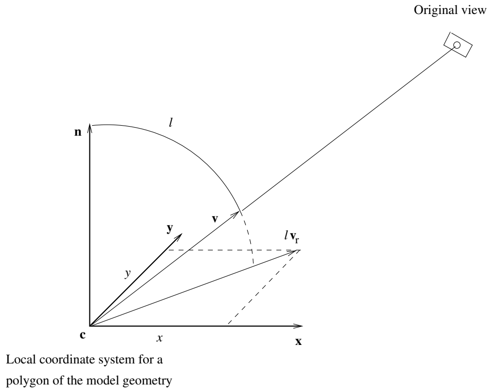


Fig. 6. The local polygon coordinate system for constructing view maps.

We pre-compute for each polygon of the model the mapping coordinates $\mathbf{p}_i = (x_i, y_i)$ for each original view i in which the polygon is visible. These points $\mathbf{p}_i$ represent a sparse sampling of view direction samples.

To make up for the expected sparsity of samples, we regularize the sampling of viewing directions as in Fig. 7. For every viewing direction on the grid, we assign to it the original view nearest to its location. This new regular configuration is what we store and use at rendering time. For current virtual viewing direction we compute its mapping $\mathbf{p}_{virtual}$ in the local space of each polygon. Then based on this value we do a quick look-up in the stored regularly resampled view map. We find the grid triangle inside which $\mathbf{p}_{virtual}$ falls and use the original views associated with its vertices to in the rendering (4, 5, and 7 in the example from Fig. 7). The blending weights are computed as the barycentric coordinates of $\mathbf{p}_{virtual}$ in the triangle in which it lies. In this manner the weights of the various viewing images are guaranteed to vary smoothly as the viewpoint changes.

8 Efficient 3-pass View-Dependent Texture-Mapping

This section explains details of the implementation of the view-dependent texture-mapping algorithm.

For each polygon visible in more than one original view we pre-compute and store the viewmaps described in section 7. Then before a frame is rendered for each polygon we find the coordinate mapping of the current viewpoint $\mathbf{p}_{virtual}$ and do a quick lookup to determine which triangle of the grid it lies inside of. As explained in 7 this gives the three best original views and their prescribed weights $\alpha_1, \alpha_2, \alpha_3$.

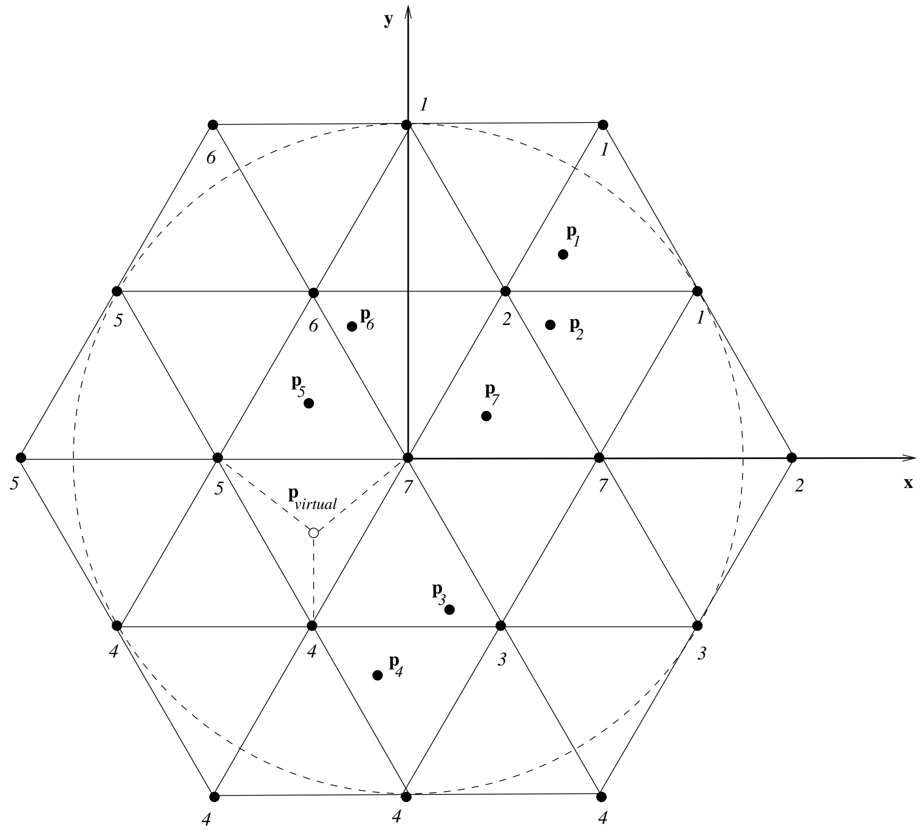


Fig. 7. The space of viewing directions for each polygon is regularly sampled, and the closest original view is stored for each sample. To determine the weightings of original views to be used in a new view, the barycentric coordinates of the novel view within its containing triangle are used. This guarantees smooth changes of the set of three original views used for texture mapping when moving the virtual viewpoint.

Since each VDTM polygon must be rendered with three texture maps, the rendering is performed in three passes. Texture mapping is enabled in modulate mode, where the new fragment color C is obtained by multiplying the existing fragment color C_f and the texture color C_t. The Z-buffer test is set to *less than or equal* (`GL_LEQUAL`) instead of the default *less than* (`GL_LESS`) to allow a polygon to blend with itself as it is drawn multiple times with different textures. The first pass proceeds by selecting an image camera, binding the corresponding texture, loading the corresponding texture matrix transformation $\mathbf{M}_{texture}$ in the texture matrix stack and sending for display the part of the model geometry for which the first best camera is the selected one with modulation color $(\alpha_1, \alpha_1, \alpha_1)$. These steps are repeated for all image cameras. The results of this pass can seen on the tower in Fig. 9 (b). The first pass fills the depth buffer with correct depth values for the entire view. Before proceeding with the second pass we enable blending in the frame buffer, i.e. instead of replacing the existing pixel values with incoming values, we add those values together. The second pass then selects cameras and renders polygons for which the second best camera is the selected one with colors $(\alpha_2, \alpha_2, \alpha_2)$. The results of the second pass can seen on the tower in Fig. 9 (c). The third pass proceeds similarly rendering polygons for which the third best camera is the currently selected one with colors $(\alpha_3, \alpha_3, \alpha_3)$. The results of this last pass can seen on the tower in Fig. 9 (d). Polygons visible only in one original view are compiled in separate list and rendered in this first pass with modulation color $(1.0, 1.0, 1.0)$ to achieve additional speed.

The polygons that are not visible in any image cameras are compiled in a separate OpenGL display list and their vertex colors are specified according to the results of the hole-filling algorithm. Those polygons are rendered in another pass with Gouraud shading after the texture mapping is disabled.

The block diagram in Fig. 8 summarizes the display loop steps.

9 Discussion and Future Work

The method we have presented proved effective at taking a relatively large-scale image-based scene and rendering it realistically at interactive rates on standard graphics hardware. Using relatively unoptimized code, we were able to achieve 20 frames per second on the Silicon Graphics InfiniteReality hardware for the full tower and campus models. Nonetheless, many aspects of this work should be regarded as preliminary in nature. One problem with the technique is that it ignores the spatial resolution of the original images in its selection process – an image that shows a particular surface at very low resolution but at just the right angle would be given greater weighting than a high-resolution image from a slightly different angle. Having the algorithm blend between the images using a multiresolution image pyramid would allow low-resolution images to influence only the low-frequency content of the renderings. However, it is less clear how this could be implemented using standard graphics hardware.

While the algorithm guarantees smooth texture weight transitions as the viewpoint moves, it does not guarantee that the weights will transition smoothly across surfaces of the scene. As a result, seams can appear in the renderings where neighboring polygons are rendered with very different combinations of images. The problem is most likely to be noticeable near the frame boundaries of the original images, or near a shadow boundary of an image, where polygons lying on one side of the boundary include an image in their view maps but the polygons on the other side do not. [2] suggested feathering the influence of images in image-space toward their boundaries and near shadow boundaries to reduce the appearance of such seams; with some consideration this technique should be adaptable to the object-space method presented here.

The algorithm as we have presented it requires all the available images of the scene to fit within the main memory of the rendering computer. For a very large-scale environment, this is unreasonable to expect. To solve this problem, spatial partitioning schemes such as those presented in [9] could be adapted for this purpose.

As we have presented the algorithm, it is only appropriate for models that can be broken into polygonal patches. The algorithm can also work for curved surfaces; these surfaces would be need to be broken down by the visibility algorithm until they are seen without self-occlusion by their set of cameras.

Lastly, it seems as if it would be more efficient to analyze the set of available views of each polygon and distill a unified view-dependent function of its appearance, rather than the raw set of original views. One such representation is the Bidirectional texture function, presented in [1], or a yet-to-be-presented form of compressed light field. Both techniques will require new rendering methods in order to render the distilled representations in real time. Extensions of techniques such as model-based stereo [2] might be

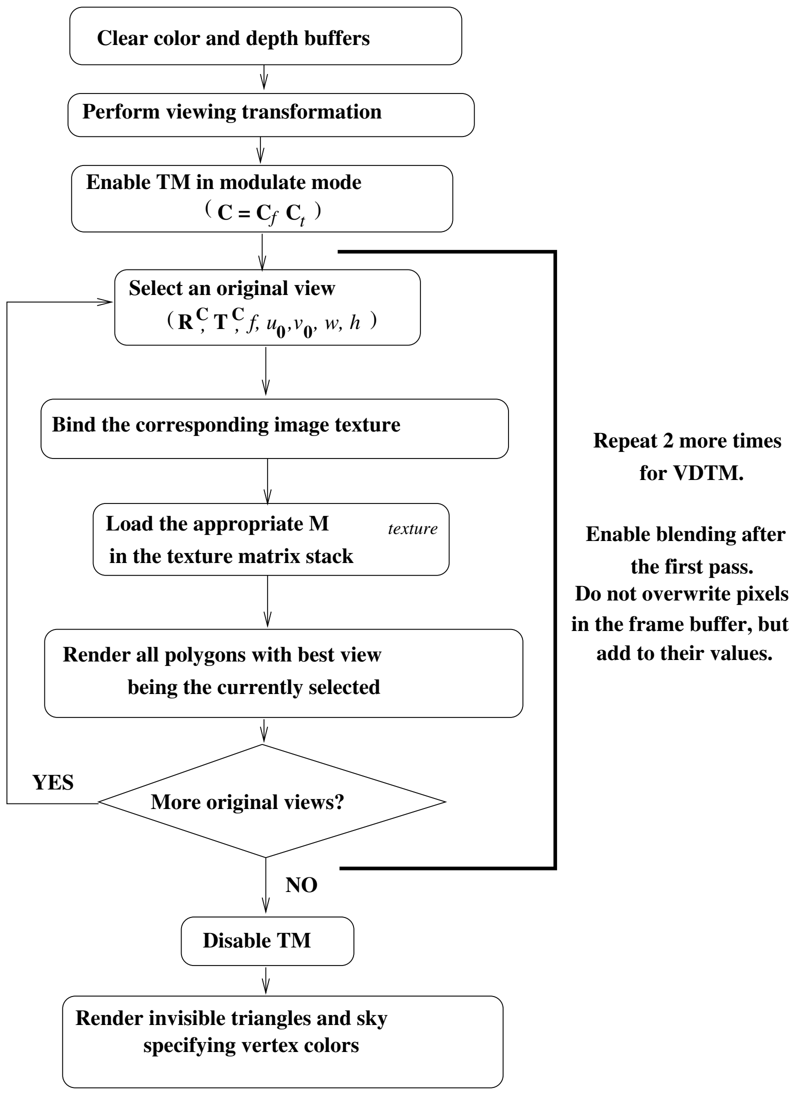


Fig. 8. Multi-pass rendering display loop.

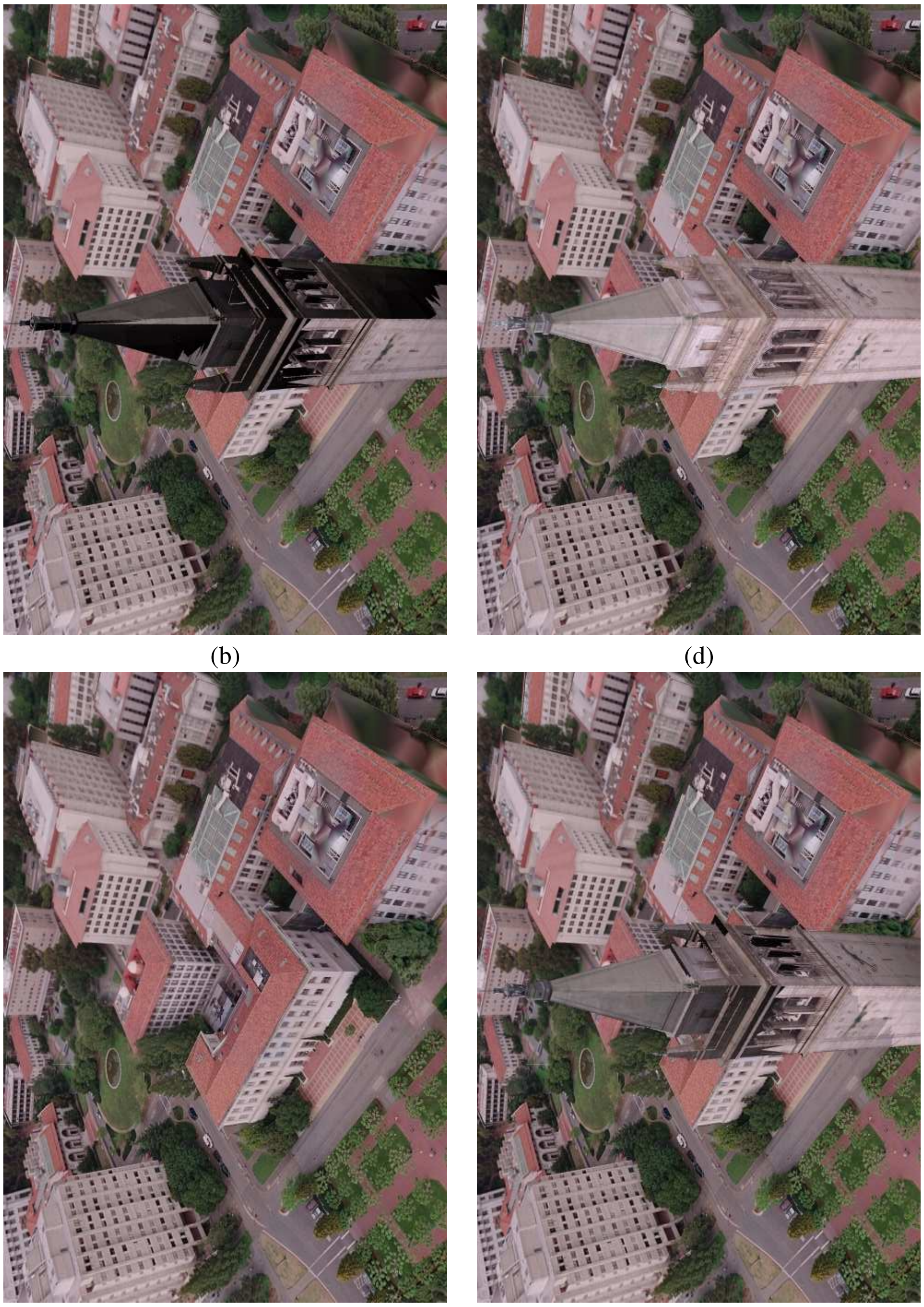
able to perform a better job than linear blending of interpolating between the various views.

10 Images and Animations

Images and Animations of the Berkeley campus model may be found at:
`http://www.cs.berkeley.edu/~debevec/Campanile`

References

1. DANA, K. J., GINNEKEN, B., NAYAR, S. K., AND KOENDERINK, J. J. Reflectance and texture of real-world surfaces. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.* (1997), pp. 151–157.
2. DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH '96* (August 1996), pp. 11–20.
3. FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics: principles and practice*. Addison-Wesley, Reading, Massachusetts, 1990.
4. GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. The Lumigraph. In *SIGGRAPH '96* (1996), pp. 43–54.
5. LAVEAU, S., AND FAUGERAS, O. 3-D scene representation as a collection of images. In *Proceedings of 12th International Conference on Pattern Recognition* (1994), vol. 1, pp. 689–691.
6. LEVOY, M., AND HANRAHAN, P. Light field rendering. In *SIGGRAPH '96* (1996), pp. 31–42.
7. MCMILLAN, L., AND BISHOP, G. Plenoptic Modeling: An image-based rendering system. In *SIGGRAPH '95* (1995).
8. SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92* (July 1992), pp. 249–252.
9. TELLER, S. J., AND SEQUIN, C. H. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH '91* (1991), pp. 61–69.
10. WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (January 1984), 52–69.
11. WEILER, K., AND ATHERTON, P. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77* (1977), pp. 214–222.
12. WILLIAMS, L., AND CHEN, E. View interpolation for image synthesis. In *SIGGRAPH '93* (1993).



(b) (d)

(a) (c)

Fig. 9. The different rendering passes in producing a frame from the photorealistic renderings of the Berkeley campus virtual fly-by. **(a)** The campus buildings and terrain; these areas were seen from only one viewpoint and are thus rendered before the VDTM passes. **(b)** The Berkeley tower after the first pass of view-dependent texture mapping. **(c)** The Berkeley tower after the second pass of view-dependent texture mapping. **(d)** The complete rendering of the scene.