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 ofview-dependent texture-mapping (VDTM) can be efficiently implementedusing projective texture mapping, a feature commonly available inpolygon graphics hardware. VDTM is a technique for generating novelviews of a scene with approximately known geometry making maximal useof a sparse set of original views. The original presentation of VDTMby Debevec, Taylor, and Malik required significant per-pixelcomputation and did not scale well with the number of original images.In our technique, we precompute for each polygon the set of originalimages in which it is visible and create a "view map" data structurethat encodes the best texture map to use for a regularly sampled setof possible viewing directions. To generate a novel view, the viewmap for each polygon is queried to determine a set of no more thanthree original images to blend together to render the polygon.Invisible triangles are shaded using an object-space hole-fillingmethod. We show how the rendering process can be streamlined forimplementation on standard polygon graphics hardware, and presentresults of using the method to render a large-scale model of theBerkeley bell tower and its surrounding campus environment.

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 practical to perform with techniques such as laser scanning or interactive photogrammetry. Acquiring the corresponding appearance information (under given lighting conditions) is easily performed with a digital camera. The remaining challenge is to use the recovered geometry and the available real views to generate novel views of the scene quickly and realistically.

In addressing this problem, it is important to make judicious use of all the available views, especially when a particular surface is seen from different directions in multiple images. This problem was addressed in [2], which presented view-dependent texture mapping as a means to render each pixel of a 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 views.

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

- A reasonably accurate 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 [16, 5, 8], 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 how to combine 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 taken from multiple locations. The method attempted to make full use of the available imagery 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, as in morphing, 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 allows renderings to be considerably more realistic than static texture-mapping allows, since it better represents non-diffuse reflectance and can simulate the appearance of unmodeled geometry.

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 is performed amongst a dense regular sampling of images in order to generate novel views. Since scene geometry is not used, a very large number of views is necessary to produce even low-resolution renderings. [4] is similar to [6] but uses irregularly sampled views and leverages approximate scene geometry derived from object silhouettes. View-dependent texture-mapping, used with a dense sampling of images and with simple geometry, reduces to the light field approach. The representation used in the presented methods restricts the viewpoint to be outside the convex hull of an object or inside a convex empty region of space. This restriction, and the number of images necessary, could complicate using these methods for acquiring and rendering a large environment. The work in this paper leverages the light field 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 predominantly diffuse, far fewer images are necessary to achieve coherent results.

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 (several minutes per frame) and did not always guarantee the image blending to vary smoothly as the viewpoint changed. Subsequent work [9, 7] presented more efficient methods for optically compositing multiple re-rendered views of a scene. In this work we associate appearance information with surfaces, rather than with viewpoints, in order to better interpolate between widely spaced viewpoints in which each sees only a part of the scene. We use visibility preprocessing, polygon view maps, and projective texture mapping to implement our technique.

3 Overview of the Method

Our method for VDTM first preprocesses the scene to determine which images observe which polygons from which directions. This preprocessing occurs as follows:

1. **Compute Visibility**: For each polygon, determine in which images it is seen. Split polygons that are partially seen in one of the images. (Section 5).
2. **Fill Holes**: For each polygon not seen in any view, choose appropriate vertex colors for performing Gouraud shading. (Section 5).
3. **Construct View Maps**: For each polygon, store the index of the image closest in viewing angle for each direction of a regularly sampled viewing hemisphere. (Section 7).

The rendering algorithm (Section 8) runs 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 each 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 their relative weights. Render the polygon using alpha-blending of the three textures with projective texture mapping.

4 Image-Based Rendering with Projective Texture Mapping

Projective texture mapping was introduced in [10] and is now part of the OpenGL graphics standard. Although the original paper used it only for shadows and lighting effects, it is directly applicable to image-based rendering because it can simulate the inverse projection of taking photographs with a camera. In order to perform projective texture mapping, the user specifies a virtual camera position and orientation, and a virtual image plane with the texture. The texture is then cast onto a geometric model using the camera position as the center of projection. The focus of this paper is to adapt projective texture-mapping to take advantage of multiple images of the scene via view-dependent texture-mapping.

Of course, we should only map a particular image onto the portions of the scene that are visible from its original camera viewpoint. The OpenGL implementation of projective texture mapping does not automatically perform such visibility checks; instead a texture map will project through any amount of geometry and be mapped onto occluded polygons as seen in Fig. 1. Thus, we need to explicitly compute visibility information before performing projective texture-mapping.

We could solve the visibility problem in image-space using ray tracing, an item buffer, or a shadow buffer (as in [2]). However, such methods would require us to compute visibility in image-space for each novel view, which is computationally expensive

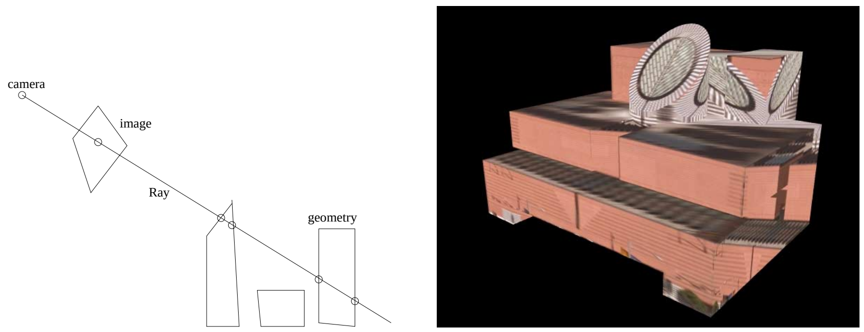


Fig. 1. The current hardware implementation of projective texture mapping in OpenGL lets the texture pass through the geometry and be mapped onto all backfacing and occluded polygons on the path of the ray, as can be seen in this rendering of a building on the right. Thus it is necessary to perform visibility pre-processing so that only polygons visible to a particular camera are texture-mapped with the corresponding image.

and not suited to interactive applications. Projective texture-mapping is extremely efficient if we know beforehand which polygons to texture-map, which suggests that we employ a visibility preprocessing step in object-space to determine which polygons are visible to which cameras. The next section describes such a visibility preprocessing method.

5 Determining Visibility

The purpose of our visibility algorithm is to determine for each polygon in the model in which images it is visible, and to split polygons as necessary so that each is fully visible or fully invisible to any particular camera. Polygons are clipped to the camera viewing frustums, to each other, and to user-specified clipping regions. This algorithm operates in both object space [3, 15] and image space and runs as follows:

1. Assign 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 line of intersection.
3. Clip the polygons against all image boundaries and any user-specified clipping regions so that all resulting polygons lie either totally inside or totally outside the view frustum and clipping regions.
4. For each camera position, rendering the original polygons of the scene with Z-buffering using the polygon ID numbers as their colors.
5. For each frontfacing polygon, uniformly sample points and project them onto the image plane. Retrieve the polygon ID at each projected point from the color buffer. If the retrieved ID is different from the current polygon ID, the potentially occluding polygon is tested in object-space to determine whether it is an occluder or coplanar.
6. Clip each polygon with each of its occluders in object-space.
7. Associate with each polygon a list of photographs to which it is totally visible.

Using identification numbers to retrieve objects from the Z-buffer is similar to the item buffer technique introduced in [14]. The image-space steps in the algorithm can quickly obtain the list of occluders for each polygon. Errors due to image-space sampling are largely avoided by checking the pixels in a neighborhood of each projection in addition to the pixels at the projected sample points.

Our technique also allows the user the flexibility to specify that only a particular region of an image be used in texture mapping. This is accomplished by specifying an additional clipping region in step 3 of the algorithm.

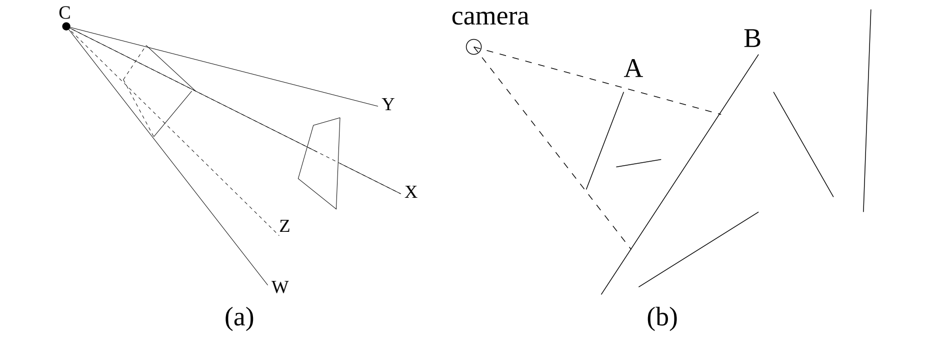


Fig. 2. (a) To clip a polygon against an occluder, we need to form a pyramid for the occluder with the apex at the camera position, and then clip the polygon with the bounding faces of the pyramid. (b) Our algorithm does *shallow clipping* in the sense that if polygon *A* occludes polygon *B*, we only use *A* to clip *B*, and any polygons behind *B* are unaffected.

The method of clipping a polygon against image boundaries is the same as that of clipping a polygon against an occluding polygon. In either case, we form a pyramid for the occluding polygon or image frame with the apex at the camera position (Fig. 2(a)), and then clip the polygon with the bounding faces of the pyramid. 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. 2(b)). Only partially visible polygons are clipped; invisible ones are left intact. This greatly reduces 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: First, we obtain the overlapping area on the image plane between each member of O and polygon P; we then 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 S is a set of convex polygons which make up the part of P not occluded by p. We recursively apply the algorithm on each member of S, first detecting its occluders and then performing the clipping.

To further reduce the number of resulting polygons, we set a lower threshold on the size of polygons. If the object-space area of a polygon is below the threshold, it is assigned a constant color based on the textures of its surrounding polygons. If a polygon is very small, it is not noticeable whether it is textured or simply a constant color. Fig. 3 shows visibility processing results for two geometric models.

6 Object-Space Hole Filling

No matter how many photographs we have, there may still be some polygons invisible to all cameras. Unless some sort of coloring is assigned to them, they will appear as undefined regions when visible in novel views.

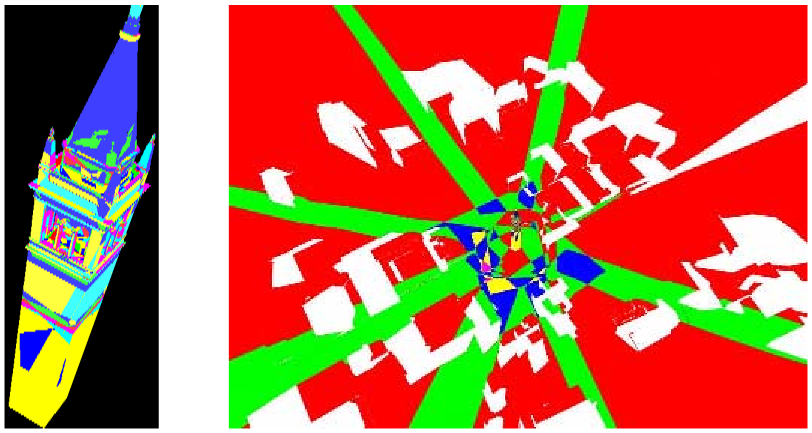


Fig. 3. Visibility results for a bell tower model with 24 camera positions and for the university campus model with 10 camera positions. The shade of each polygon encodes the number of camera positions from which it is visible; the white regions in the overhead view of the second image are “holes” invisible to all cameras.

Instead of relying on photographic data for these regions, we instead assign colors to them based on the appearance of their surrounding surfaces, a processed called *hole filling*. Previous hole-filling algorithms [16, 2, 7] have operated in image space, which can cause flickering in animations since the manner in which a hole is filled will not necessarily be consistent from frame to frame. Object-space hole-filling can guarantee that the derived appearance of each invisible polygon is consistent between viewpoints. By filling these regions with colors close to the colors of the surrounding visible polygons, the holes can be made difficult to notice.

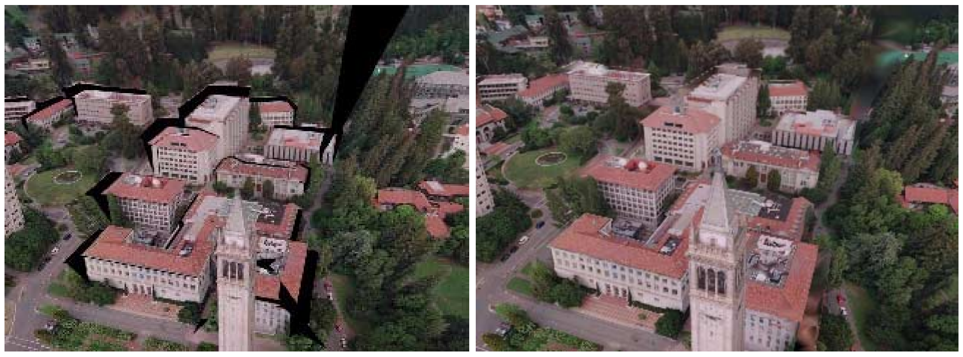


Fig. 4. The image on the left exhibits black regions which were invisible to all the original cameras but not to the current viewpoint. The image on the right shows the rendering result with all the holes filled. See also Fig. 8.

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. **Determine colors of visible polygons**. Compute an "average" color for each visible polygon by projecting its centroid onto the image planes of each image in which it appears and sample the colors at those coordinates.
3. **Iteratively assign colors to the holes**. For each invisible polygon, if it has not yet been assigned a color, assign to each of its vertices the color of the closest polygon which is visible or that has been filled in a previous iteration.

The reason for the iterative step is that an invisible polygon may not have a visible polygon in its neighborhood. In this way its vertex colors can be determined after its neighboring invisible polygons are assigned colors.

Due to slight misalignments between the geometry and the original photographs, the textures of the edges of some objects may be projected onto the background. For example, a sliver of the edge of a building may project onto the ground nearby. In order to avoid filling the invisible areas with these incorrect textures, we do not sample polygon colors at regions directly adjacent to occlusion boundaries.

Fig. 4 shows the results of hole filling. The invisible polygons, filled will Gouraud-shaded low-frequency image content, are largely unnoticeable in animations. Because we assume that the holes will be relatively small and that the scene is mostly diffuse, we do not use the view-dependent information to render the holes.

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. In this way, the errors in rendered appearance due to specular reflectance and incorrect model geometry will be minimized. Note that in any particular novel view, different visible surfaces 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 from frame to frame, we wish to blend between the available views as the angle of view changes. This section shows how for each polygon we 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 polygon's view map may be queried very efficiently: given a desired novel viewpoint, it quickly returns 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. The view maps are stored and used at rendering time to determine the three best original views and their blending factors by a quick look-up based on the current viewpoint.

The local polygon coordinate system is constructed as in Equation 1:

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

$$\mathbf{y} \quad = \quad \mathbf{n} \times \mathbf{x} \tag{1}$$

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

We transform viewing directions to the local coordinate system as in Fig. 5. 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.

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

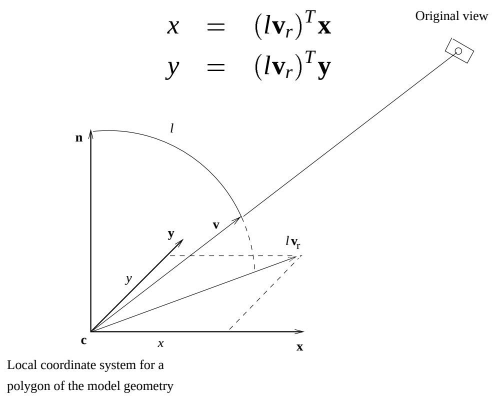


Fig. 5. The local polygon coordinate system for constructing view maps.

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 extrapolate the sparse set of original viewpoints, we regularize the sampling of viewing directions as in Fig. 6. 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 the 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 lookup into the regularly resampled view map. We find the grid triangle inside which $\mathbf{p}_{virtual}$ falls and use the original views associated with its vertices in the rendering ($\mathbf{p}_4$, $\mathbf{p}_5$, and $\mathbf{p}_7$ in the example of Fig. 6). 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 the implementation of the view-dependent texture-mapping rendering algorithm.

For each polygon visible in more than one original view we pre-compute and store the viewmaps described in Section 7. Before rendering begins, for each polygon we

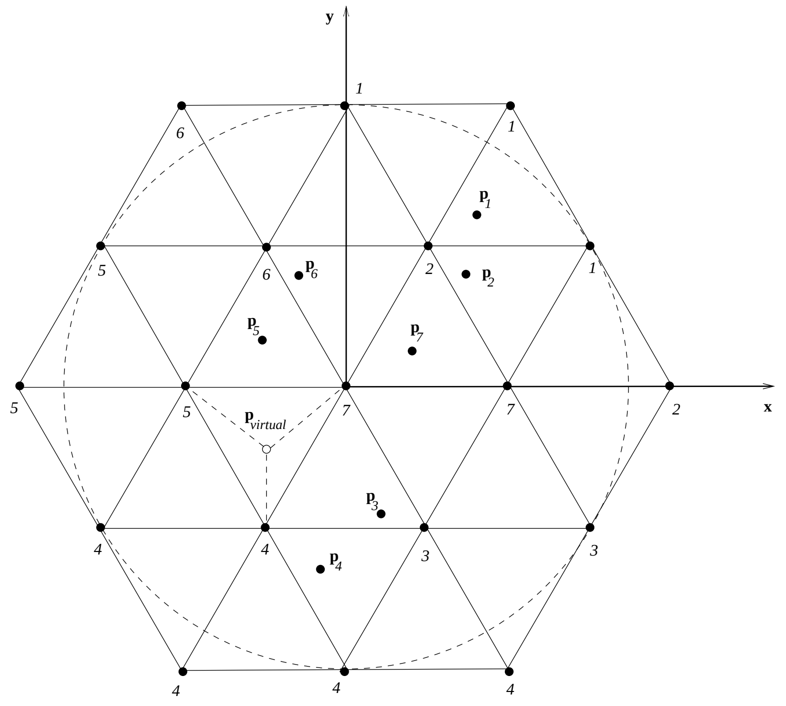


Fig. 6. A View Map. The space of viewing directions for each polygon is regularly sampled, and the closest original view is stored for each sample. To determine the weightings of original views to be used in a new view, the barycentric coordinates of the novel view within its containing triangle are used. This guarantees smooth changes of the set of three original views used for texture mapping when moving the virtual viewpoint. Here, for viewpoint $\mathbf{p}_{virtual}$, the polygon corresponding to this view map will be texture-mapped by an almost evenly weighted combination of original views $\mathbf{p}_4$, $\mathbf{p}_5$, and $\mathbf{p}_7$, since those are the views assigned to the vertices of $\mathbf{p}_{virtual}$'s view map triangle.

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 within. As explained in Section 7 this returns the three best original views and their relative weights $\alpha_1, \alpha_2, \alpha_3$.

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 pixel color C is obtained by multiplying the existing pixel 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 rendering 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. 8 (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 modulation color $(\alpha_2, \alpha_2, \alpha_2)$. The results of the second pass can seen on the tower in Fig. 8 (c). The third pass proceeds similarly, rendering polygons for which the third best camera is the currently selected one with modulation color $(\alpha_3, \alpha_3, \alpha_3)$. The results of this last pass can seen on the tower in Fig. 8 (d). Polygons visible from only one original viewpoint are compiled in separate list and rendered during the first pass with modulation color $(1.0, 1.0, 1.0)$.

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 before the first pass with Gouraud shading after the texture mapping is disabled.

The block diagram in Fig. 7 summarizes the display loop steps.

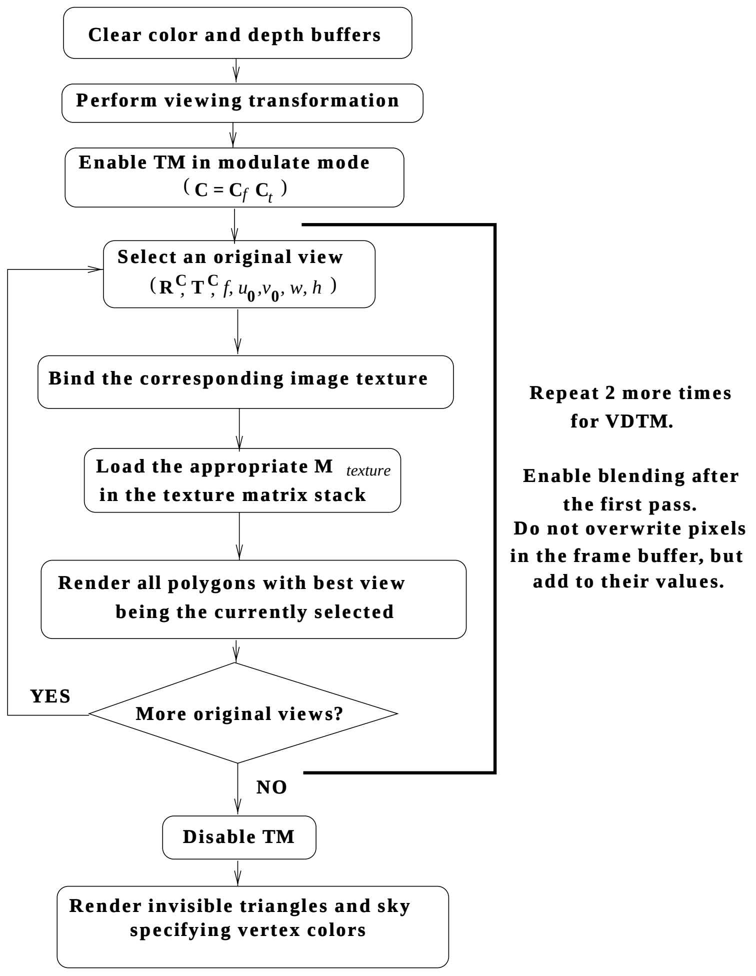

Clear color and depth buffers
Perform viewing transformation
Enable TM in modulate mode ($\mathbf{C} = \mathbf{C}_f\ \mathbf{C}_t$)
Select an original view ($\mathbf{R}^{\mathbf{C}}, \mathbf{T}^{\mathbf{C}}, f, u_0, v_0, w, h$)
Bind the corresponding image texture
Load the appropriate $\mathbf{M}_{texture}$ in the texture matrix stack
Render all polygons with best view being the currently selected
More original views?
YES
NO
Disable TM
Render invisible triangles and sky specifying vertex colors
Repeat 2 more times for VDTM.
Enable blending after the first pass.
Do not overwrite pixels in the frame buffer, but add to their values.

Fig. 7. The multi-pass view-dependent projective texture mapping rendering loop.

9 Discussion and Future Work

The presented method was effective at realistically rendering a relatively large-scale image-based scene at interactive rates on standard graphics hardware. Using relatively unoptimized code, we were able to achieve 20 frames per second on a Silicon Graphics InfiniteReality 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] and [9] suggest 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 [13], image caching [11], and impostor manipulation [11, 12] techniques could be adapted to the current framework.

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 (such as those acquired by laser scanning); 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 geometry-enhanced light field. Such a technique will require new rendering methods in order to render the distilled representations in real time. Lastly, extensions of techniques such as model-based stereo [2] might be able to perform a better job of interpolating between the various views than linear interpolation.

Images and Animations

Images and Animations of the Berkeley campus model may be found at:

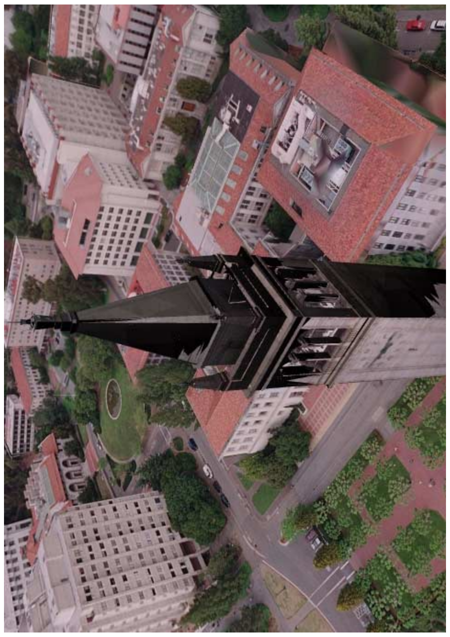
`http://www.cs.berkeley.edu/~debevec/Campanile`

Acknowledgments

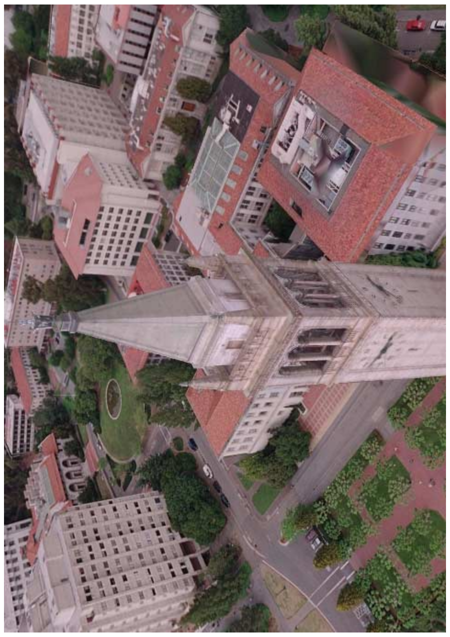
The authors wish to thank Jason Luros, Vivian Jiang, Chris Wright, Sami Khoury, Charles Benton, Tim Hawkins, Charles Ying, Jitendra Malik, and Camillo Taylor for their contributions to the Berkeley Campus Animation. This research was supported by Silicon Graphics and a Multidisciplinary University Research Initiative on 3D Direct visualization from ONR and BMDO, grant FDN00014-96-1-1200.

References

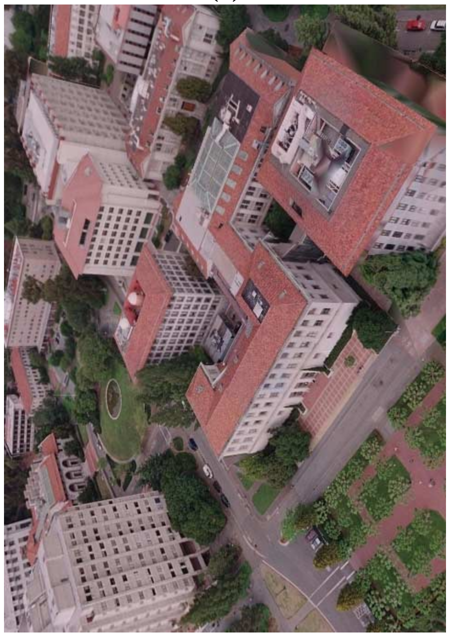
1. DANA, K. J., GINNEKEN, B., NAYAR, S. K., AND KOENDERINK, J. J. Reflectance and texture of real-world surfaces. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.* (1997), pp. 151–157.
2. DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. Modeling and rendering architecture from photographs: A hybrid 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. MARK, W. R., MCMILLAN, L., AND BISHOP, G. Post-rendering 3D warping. In *Proceedings of the Symposium on Interactive 3D Graphics* (New York, Apr.27–30 1997), ACM Press, pp. 7–16.
8. MCMILLAN, L., AND BISHOP, G. Plenoptic Modeling: An image-based rendering system. In *SIGGRAPH '95* (1995).
9. PULLI, K., COHEN, M., DUCHAMP, T., HOPPE, H., SHAPIRO, L., , AND STUETZLE, W. View-based rendering: Visualizing real objects from scanned range and color data. In *Proceedings of 8th Eurographics Workshop on Rendering, St. Etienne, France* (June 1997), pp. 23–34.
10. 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.
11. SHADE, J., LISCHINSKI, D., SALESIN, D., DEROSE, T., AND SNYDER, J. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings* (1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 75–82.
12. SILLION, F., DRETTAKIS, G., AND BODELET, B. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum (Proc. Eurographics 97) 16*, 3 (Sept. 4–8 1997), C207–C218.
13. TELLER, S. J., AND SEQUIN, C. H. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH '91* (1991), pp. 61–69.
14. WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (January 1984), 52–69.
15. WEILER, K., AND ATHERTON, P. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77* (1977), pp. 214–222.
16. WILLIAMS, L., AND CHEN, E. View interpolation for image synthesis. In *SIGGRAPH '93* (1993).



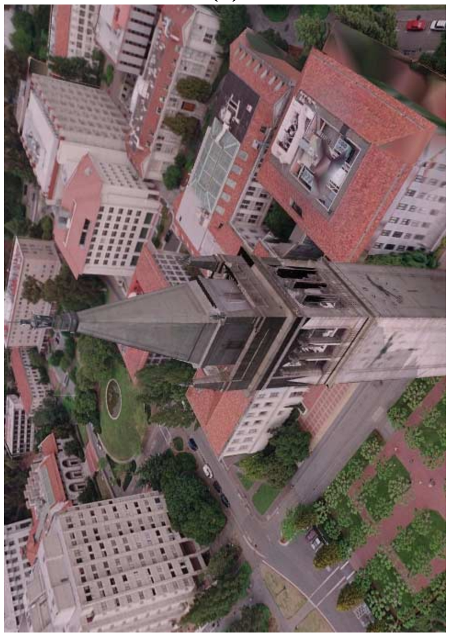
(b)



(d)



(a)



(c)

Fig. 8. The different view-dependent projective texture-mapping passes in producing a frame of the Berkeley campus virtual fly-by. The complete model contains approximately 100,000 triangles. **(a)** The campus buildings and terrain after hole-filling; 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 after all three VDTM passes.