To appear in Journal of *Computers and Graphics*

Efficient Visibility Processing for Projective Texture-mapping

Yizhou Yu

Computer Science Division, University of California at Berkeley, USA

Abstract

Projective texture-mapping is a powerful tool for image-based rendering. However, visibility information must be obtained in order to project correct textures onto the geometric sturcture. This paper presents a novel visibility algorithm for determining which polygons are visible from each camera position. It has object-space precision, but operates in both image-space and object-space to get better performance. It generates much less polygons than traditional object-space algorithms to help accelerate the speed of texture-mapping. The techniques used to achieve these are *conservative image-space occlusion testing* and *object-space shallow clipping*. This algorithm has been successfully used to generate large-scale image-based rendering of a whole university campus, and image-based rendering of architectures under dynamic lighting.

1 Introduction

Projective texture mapping was first introduced in [1] and now has been implemented in OpenGL graphics package on SGI machines. Although it was only for shadows and lighting effects in the original paper, people have found it extremely useful in the fast developing field of *image-based rendering* because projective texture mapping simulates the setting of taking photographs with a camera. In order to do projective texture mapping, the user needs to specify a virtual camera position and orientation, 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 image-based rendering, real photographs with calibrated camera positions are often used to re-render a scene. If there are also some recovered 3D geometric structures for the same real scene, these photographs can be texture-mapped onto the recovered structures by projective texture mapping to generate extremely realistic renderings. 3D geometric models can be reconstructed from laser range data [2] or multiple photographs [3]. The system in [3] implemented this projective texture mapping in software. However, the hardware implementation on SGI machines are much faster.

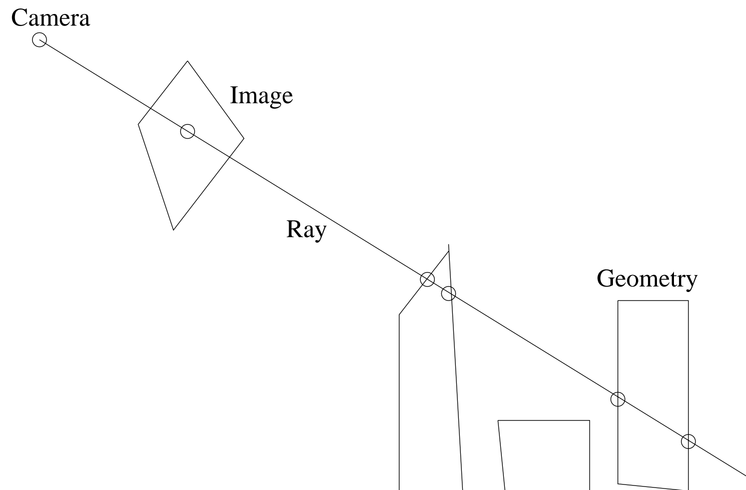


Figure 1: The current hardware implementation of projective texture mapping on SGI workstations let the texture pierce through the geometry and get mapped onto all backfacing and occluded polygons on the path of the ray

For a complex scene, multiple photographs taken at different camera positions are often needed to cover the whole scene. We want to texture-map those real images onto the recovered structure. For a fixed image, we only want to

map this image onto the polygons visible to the camera position where we took this image. We should not erroneously map it onto those occluded polygons. The current hardware implementation of projective texture mapping on SGI workstations cannot do this in hardware. It let 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 legible texture coordinates and are incorrectly texture mapped instead of remaining in shadow. This indicates we need to obtain visibility information before texture-mapping.

We could solve this visibility problem in image-space using ray tracing or item buffer. But that means if we want to render a large number of frames for an animation or do a real-time demonstration, we need to compute visibility in image-space for each frame, which would be impractically slow. 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 sequence of animation, this object-space preprocessing needs to be done only once. It also allows the view point to be changed dynamically during a real-time demonstration.

In an object-space visibility preprocessing, we should subdivide partially visible polygons so that we only map the current image to their visible parts. If a polygon is invisible from this camera position, it can still be visible from other camera positions since we have multiple images. So we need to feed a list of visible camera positions for each polygon to the rendering program and let it choose dynamically which ones should be used for the current view point.

Since the whole scene is covered by multiple images which may have overlapping areas and gaps, the object-space visibility preprocessing also needs to deal with the image boundary problem. If a polygon is 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.

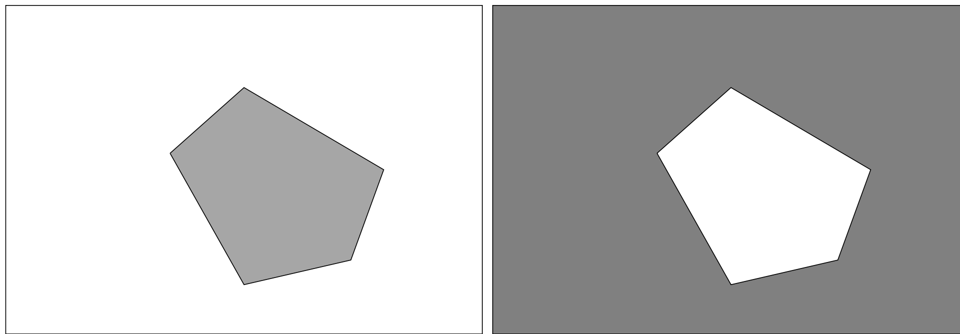


Figure 2: Users can pick a desired texture region from each image by drawing a convex polygon inside the image frame. The chosen texture region may be either the interior or the exterior of the polygon.

It is also 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.

This visibility problem is for a number of fixed camera positions. If we consider the visibility subproblem at one camera position, it is similar to the traditional problem of visible surface determination [12]. But object-space preprocessing means previous image-space algorithms are not appropriate. Previous object-space algorithms [12, 7] were designed only for one viewpoint. They do a lot of needless polygon clipping on completely and partially invisible polygons, therefore, increase the number of polygons dramatically. If we repeatedly applied them on the recovered geometry view by view, the size of the final polygon set would be unmanagable. We wish to minimize the number of polygons resulted from visibility processing because a large number of polygons will significantly slow down the speed of hardware texture mapping.

There is another class of visibility algorithms [10, 11]. They try to analyze visibility for all pairs of surface patches in an environment so that shadows and/or global illumination can be computed efficiently for area sources. They are

also object-space preprocessing. Since these algorithms deal with area sources while we are concerned about fixed camera positions, they are not suitable for our problem.

We will introduce our visibility algorithm in the next section. In Section 3, we will give the performance of our algorithm and compare it with some existing algorithm. In Section 4, we will introduce some interesting applications we have done. Section 5 has the conclusions.

2 The algorithm

What we need to do is to decide in which photographs a particular polygon from the geometric model is visible. If a polygon is partially visible in a photograph, we should clip it so that each resulting polygon is either totally visible or totally invisible. After this visibility processing, we can correctly and efficiently assign radiance values from the photographs to the visible polygons.

This algorithm is operated in both image space and object space. 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, and render the original large polygons from the geometric model using their id numbers as their colors and Z-buffer hardware.
5. At each camera position, scan-convert each frontfacing polygon in software so we can know which pixels are covered by it. If at some covered pixel location, the retrieved polygon id from the color buffer is different from the id of the polygon currently under consideration, a potentially occluding polygon is found and it is tested in object-space 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.

Clipping in object-space does not take much time because we use the original large polygons in the hardware Z-buffering step, which results in a very small set of occluders for each polygon. So this algorithm has nearly the speed of image-space algorithms and the accuracy of object-space algorithms as long as the original polygons in the model are all larger than a pixel. Using identification(id) numbers to retrieve objects from Z-buffer is similar to the item buffer technique introduced in [8].

There are some variants of this algorithm. First, we can replace scan conversion with object-space uniform sampling. On each polygon, draw uniform samples and project these sample points onto the image plane to check if there are any occluding polygons. But scan-conversion is obviously faster. Second, under some circumstances, we need some data structure to maintain the connectivity of the polygons and T intersections are not allowed.

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 detect all occluding polygons so that texture mapping are done correctly. To achieve this goal, the following two techniques are introduced, *conservative testing* and *shallow clipping*.

2.1 Conservative Image-space Occlusion Testing

From the hardware Z-buffering results, we want to safely detect all occluding polygons. We donot want to miss any occluding polygons. But if a nonoccluding polygon is erroneously reported, that is fine because we can do object-space testing to verify if it is really an occluding polygon and get rid of it if it is not.

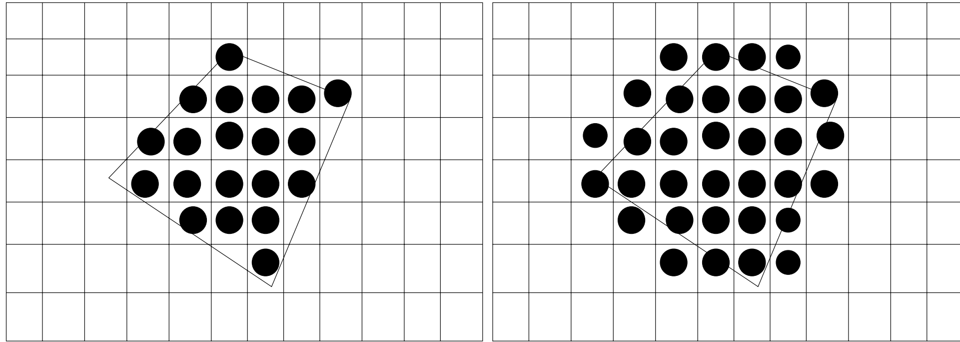


Figure 3: In the original scan-conversion of a polygon, only pixels whose centers are within the projected polygon are rendered, which is shown on left; our conservative occlusion testing checks all pixels with some overlap with the projected polygon, which is shown on right.

For the version using scan-conversion, if we consider each pixel is a tiny square, conservative testing is done by checking all pixels overlapping with the projected area on the image plane of the currently considered polygon instead of only checking the pixels whose centers are within the projected area as in the original version of scan conversion(Fig. 3). If any of these pixels have an id number different from the polygon's own id number, a potentially occluding polygon is found.

We can not use hardware Z-buffering for this scan-conversion because we have to check all pixels overlapping with the projected area of a polygon to see whether the considered polygon is completely or partially visible and which polygons occlude it. However, Z-buffering only gives the visible polygon at each pixel, and cannot directly tell us which pixels overlap with the polygon. Without software scan-conversion, we need to check the whole color buffer to obtain these pixels. Repeating this for every polygon is obviously impractical.

For the version using uniform object-space sampling, at the image-plane projection of each sample point, check all pixels in its neighborhood for occluding polygons rather than check only one pixel corresponding to the projected location.

The purpose of this image-space testing step is to quickly obtain the list of potentially occluding polygons.

2.2 Accurate Object-space Occlusion Testing

We assume all polygons are convex here since nonconvex polygons need to be tessellated before rendering with Z-buffer.

To test in object-space if a polygon really occludes another polygon from a camera position, we should first test if they are coplanar because Z-buffer only gives one id number at each pixel despite the fact that there might be coplanar polygons at the same depth. If the polygons are coplanar, we consider them both visible. Otherwise, we should form a viewing pyramid for the first polygon with the apex at the camera position and check if the second polygon is outside this pyramid(Fig. 4). Each bounding face of the pyramid has two edges which are half lines extending to infinity. The semi-infinite bounding faces are called *wedges*. Each wedge lies on a plane. We can check if a polygon falls totally inside a pyramid by using the plane equations of its wedges. To test if the polygon falls outside the pyramid, we need to check each wedge with the polygon. Using the plane equations are not safe

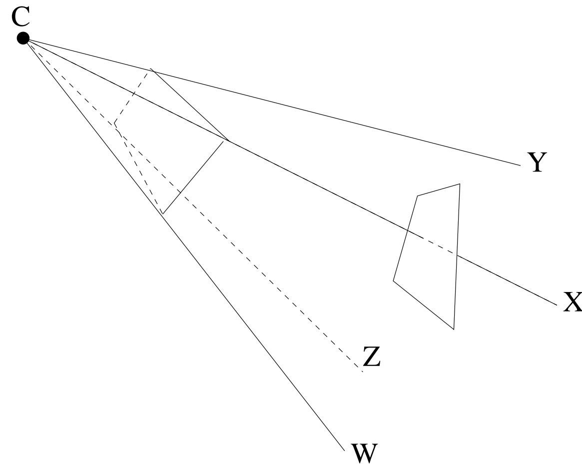


Figure 4: To verify if a polygon really occludes another one in object-space, we need to form a pyramid for the first polygon with the apex at the camera position and check if the second polygon is outside this pyramid

any more, see Fig. 5 for a counterexample. We have the following conditions to verify whether a wedge intersects a polygon.

Theorem *A wedge intersects a polygon if there are vertices of the polygon on both sides of the plane to which the wedge belongs, and either of the following two conditions is satisfied*

i) one of the edges of the wedge goes through the interior of the polygon,

ii) one of the edge segments of the polygon intersects the interior of the wedge.

Fig. 6 shows all the cases that a polygon can intersect a wedge. We can consider it as a simple proof of the above theorem.

If a polygon is in front of another one and its corresponding pyramid intersects the other polygon, it becomes a real occluder. We need to clip the occluded polygon against the occluding polygon.

2.3 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 cases, one should first form a viewing pyramid as in Section 2.2, 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 using the above theorem.

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. 7). Only partially visible polygons are clipped. Those totally invisible ones are left intact. This is the major reason that our algorithm can 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 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.

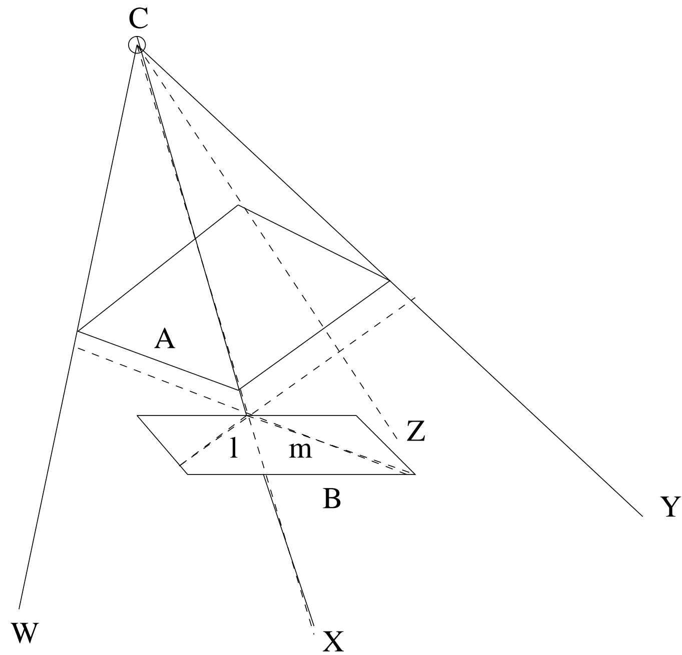


Figure 5: To test if the polygon falls outside the pyramid, we need to check each wedge with the polygon. Using the plane equations are not safe any more. This is a counterexample. Polygon B is actually outside the pyramid for polygon A. But it has two intersections l and m with the planes defining two sides of the pyramid.

2.4 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.

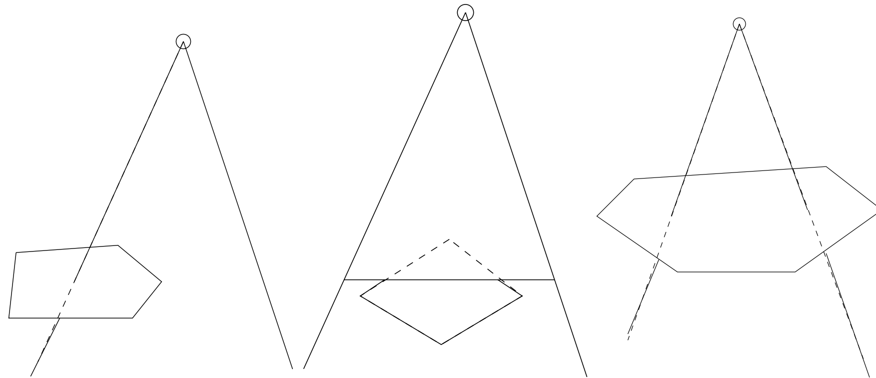


Figure 6: All possible situations that a polygon can intersect a wedge

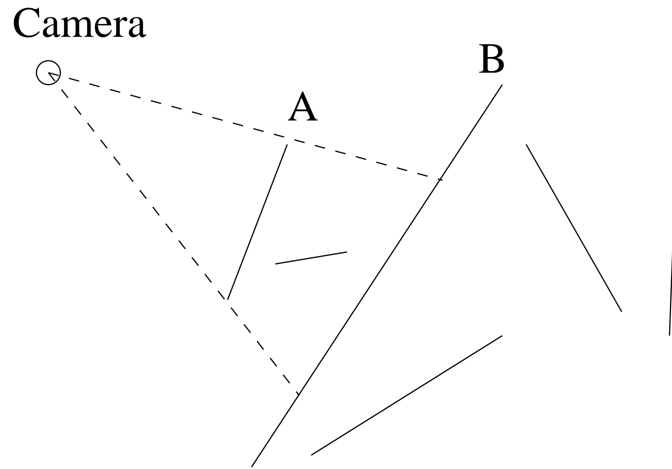


Figure 7: Our algorithm does *shallow clipping* in the sense that if polygon A occludes polygon B, we only use A to clip B, and any polygons behind B are unaffected.

3 Experimental Results

Fig. 8(a) shows the texture-mapping result if we do not obtain visibility information of the geometric model. The backfacing and occluded faces obtained incorrect textures.

We have implemented the above visibility algorithm on SGI machines. Although it operates in both image-space and object-space, it actually outputs object-space geometric data which are the polygons after subdivision and a list of camera views to which each polygon is visible. These data can be used to generate real-time projective texture-mapping.

To demonstrate our algorithm's efficiency in both polygon clipping and running time, we compare it with a pure object-space algorithm which is a modified version of Weiler-Atherton hidden surface removal algorithm [7, 12]. The hierarchical Z-buffer algorithm in [9] is good for complex models. But it is essentially an image-space algorithm and is not appropriate for our purpose. Since we have more than one camera positions in our situation, we apply Weiler-Atherton algorithm repeatedly at each camera position and the input data for the processing at each camera position is the output data from the processing at the previous camera position.

The comparison was performed on three data sets. One is a very detailed model of a bell tower(Fig. 8(b)). Another is a coarse model of a university campus, including both the buildings and the terrain(Fig. 8(c)). These two models were recovered by the system in [3]. The third data set is a cluster of polygons uniformly distributed in a bounding box. The polygons have random sizes and orientations. They do not give any obvious object shape. The number of polygons in the original models and the number of camera views used for visibility processing are shown in Table 1.

For fair comparison, in both algorithms we only process those frontfacing polygons falling into field of view at each camera position. The visibility processing results for the first two models are shown in Fig. 9. Both the running time on a SGI O_2 workstation and the number of polygons generated after visibility processing are compared in Table 2-4 where our algorithm is named Hybrid. The comparison is done for three thresholds on polygon size. Any polygon smaller than the threshold is no longer subdivided as proposed in Section 2.4. From these results, we can see our algorithm is significantly better in both running time and the number of polygons generated. The running time of each part of our algorithm is shown in Table 5. The time spent on Z-buffering is ignorable. Scan-conversion is the most time-consuming part which keeps the time spent on object-space operations much lower than it should be for an algorithm with object-space precision. We can speed up this part by identifying all the completely or partially visible polygons from the hardware Z-buffering results, performing scan-conversion only on these polygons and ignoring those completely invisible polygons. However, as preprocessing, the compactness of the final polygon set

is much more important than speed.

	Model I	Model II	Model III
# polygons	2409	660	1000
# views	24	10	16

Table 1: Statistics for three geometric data sets: number of polygons and number of camera positions for visibility processing.

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	19018	7511	3584
WA	62556	12461	4779

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	112.14s	81.15s	61.47
WA	875.40s	234.97	175.53

Table 2: Comparison between the Hybrid and Weiler-Atherton(WA) algorithms on Model I with three different thresholds on polygon size. The first half shows the number of polygons generated. The second half shows the running time in seconds on a SGI O_2 workstation.

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	5623	5541	5255
WA	10194	9878	8675

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	14.36s	14.63s	13.96s
WA	22.85s	21.60s	18.64s

Table 3: Comparison between the Hybrid and Weiler-Atherton(WA) algorithms on Model II with three different thresholds on polygon size. The first half shows the number of polygons generated. The second half shows the running time in seconds on a SGI O_2 workstation.

4 Applications

We designed this visibility algorithm in terms of the demand from real-time image-based rendering. So far, we have successfully used this algorithm to process data in two applications. The first is for view-dependent projective texture-mapping. By visibility processing and projecting the real images like those shown in Fig. 10(a)-10(d), we are able to have a recovered large-scale 3D model covered with seamless textures, and re-render the model from novel view points(Fig. 11). This application has produced a fly-by animation for UC Berkeley campus [5]. Since our algorithm generates much less polygons than previous algorithms, we are able to produce a corresponding real-time(60Hz) demonstration on SGI Onyx2 InfiniteReality Engine. The second application uses visibility processing to assign correct radiance values from photographs to their corresponding geometric surfaces and then recover the reflectance of the surfaces [4]. Thus we are able to re-render the model under novel lighting conditions such as a novel solar position for an outdoor scene(Fig. 10(e)).

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	15745	8596	6573
WA	40973	16238	9040

	Th = 0.0001	Th = 0.001	Th = 0.01
Hybrid	90.43s	62.96s	47.23s
WA	239.74s	182.32s	141.92s

Table 4: Comparison between the Hybrid and Weiler-Atherton(WA) algorithms on Model III with three different thresholds on polygon size. The first half shows the number of polygons generated. The second half shows the running time in seconds on a SGI O_2 workstation.

	Th = 0.0001	Th = 0.001	Th = 0.01
HW Z-buffering	0.69s	0.68s	0.72s
Scan-conversion	61.92s	50.80s	44.34s
Object-space ops	49.53s	29.67s	16.41s

	Th = 0.0001	Th = 0.001	Th = 0.01
HW Z-buffering	0.09s	0.08s	0.09s
Scan-conversion	8.16s	8.30s	7.84s
Object-space ops	6.11s	6.25s	6.03s

Table 5: Statistics of the running time of each part of the Hybrid algorithm on Model I and II, including Z-buffering time, scan-conversion time and time for object-space testing and clipping operations.

5 Conclusions

This paper presents a novel visibility algorithm for projective texture-mapping which is a powerful tool for image-based rendering. It is a hybrid algorithm operating in both image-space and object-space. It has object-space precision, but part of it operates in image-space in order to get better performance. It generates much less polygons than traditional object-space algorithms to help accelerate the speed of texture-mapping. The techniques used to achieve these are *conservative image-space occlusion testing* and *object-space shallow clipping*. This algorithm has been successfully used to generate large-scale image-based rendering of a whole university campus, and image-based rendering of architectures under dynamic lighting.

Acknowledgments

This research was supported by a Multidisciplinary University Research Initiative on three dimensional direct visualization from ONR and BMDO, grant FDN00014-96-1-1200, the California MICRO program, Philips Corporation and Microsoft Graduate Fellowship. The author wishes to thank George Borshukov and Paul E. Debevec for providing the geometric models and texturing-mapping results used in this paper.

References

[1] M. Segal, C. Korobkin, R. van Sidenfelt, J. Foran, and P. Haeberli, Fast Shadows and Lighting Effects Using Texture Mapping. *Computer Graphics*, 26(2), 249-252 (1992).

[2] B. Curless, and M. Levoy, A volumetric method for building complex models from range images. In *Proc. of SIGGRAPH 96*, 303-312 (1996).

[3] P.E. Debevec, C.J. Taylor, and J. Malik, Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach. In *Proc. of SIGGRAPH 96*, 11-20 (1996).

[4] Y. Yu, and J. Malik, Recovering Photometric Properties of Architectural Scenes from Photographs. In *Proc. of SIGGRAPH 98*, 207-217 (1998).

[5] P.E. Debevec, Y. Yu, and G.D. Borshukov, Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. In *9th Eurographics Workshop on Rendering*, 105-116 (1998).

[6] P.S. Heckbert, Fundamentals of texture mapping and image warping. *Master's thesis, Computer Science Division, UC Berkeley* (1989).

[7] K. Weiler, and P. Atherton, Hidden Surface Removal Using Polygon Area Sorting. In *Proc. of SIGGRAPH 77*, 214-222 (1977).

[8] H. Weghorst, G. Hooper, and D.P. Greenberg, Improved Computational Methods for Ray Tracing. In *ACM TOG*, 3(1), 52-69 (1984).

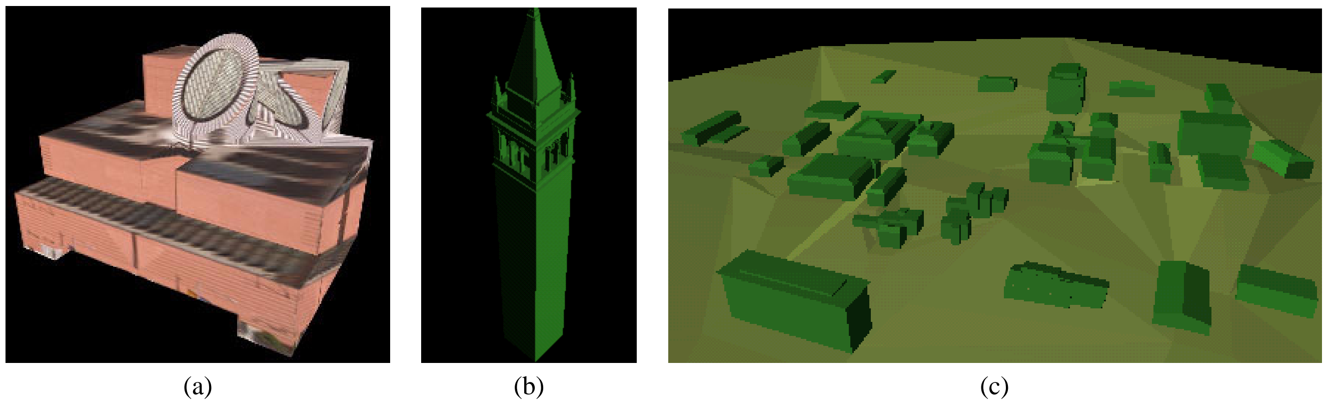
[9] N. Greene, M. Kass, and G. Miller, Hierarchical Z-Buffer Visibility. In *Proc. of SIGGRAPH 93*, 231-238 (1993).

[10] S. Teller, and P. Hanrahan, Global Visibility Algorithms for Illumination Computations. In *Proc. of SIGGRAPH 93*, 239-246 (1993).

[11] G. Drettakis, and E. Fiume, A Fast Shadow Algorithm for Area Light Sources Using Backprojection. In *Proc. of SIGGRAPH 94*, 223-230 (1994).

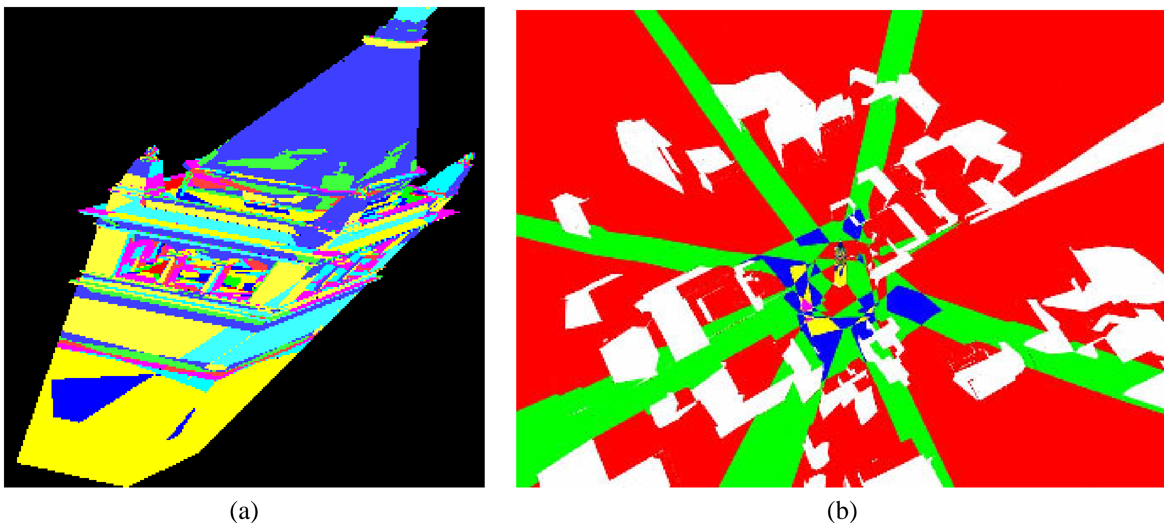
[12] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: principles and practice*, 2nd Edition, Addison-Wesley (1996).

[13] Y. Yu, and H. Wu, A Rendering Equation for Specular Transfers and Its Integration into Global Illumination. In *Proc. of EUROGRAPHICS 97*, *Computer Graphics Forum*, 16(3), 283-292 (1997).



(a) (b) (c)

Figure 8: (a)Viewing the model from a viewpoint far from the original produces artifacts unless proper visibility pre-processing is performed, (b)A detailed bell tower model, (c)A model for a university campus, including both the buildings and the terrain.



(a) (b)

Figure 9: Visibility results for a bell tower model with 24 camera positions and for a university campus model with 10 camera positions. The color of each polygon encodes the number of camera positions from which it is visible: white= 0, red= 1, green= 2, blue= 3, yellow= 4, cyan= 5, magenta= 6 and unsaturated colors for larger numbers.

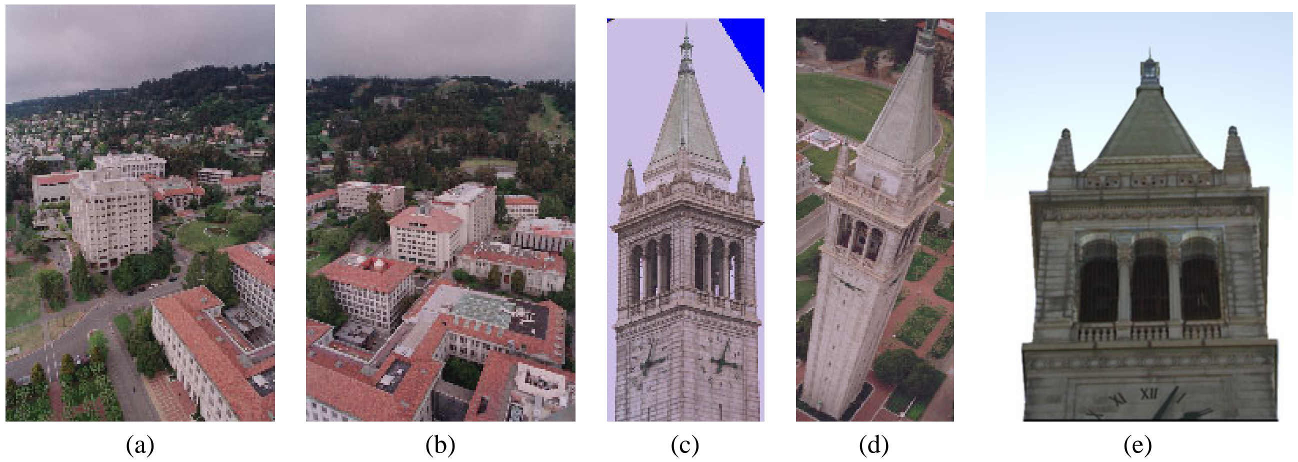


Figure 10: (a)-(d) Some of the photographs of the university campus and bell tower used for projective texture-mapping, (e) A re-rendering of the bell tower under a new lighting condition with the sun directly behind it so the frontfacing side is shaded and has a light blue tint.

Figure 11: Two re-rendered images of the university campus at two novel view points. The textures are actually from different photographs, but they seamlessly cover the geometry using visibility processing.