Fast Approximate Quantitative Visibility for Complex Scenes

Yiorgos Chrysanthou[1] Daniel Cohen-Or[2] Dani Lischinski[3]

[1]Department of Computer Science
UCL University of London
Gower Street, London WC1E 6BT, UK
E-mail: Y.Chrysanthou@cs.ucl.ac.uk

[2]Computer Science Department
School of Mathematical Sciences
Tel-Aviv University, Ramat-Aviv 69978, Israel
E-mail: daniel@math.tau.ac.il

[3]Institute of Computer Science
The Hebrew University, Givat Ram
Jerusalem 91904, Israel
E-mail: danix@cs.huji.ac.il

Abstract

Ray tracing and Monte-Carlo based global illumination, as well as radiosity and other finite-element based global illumination methods, all require repeated evaluation of quantitative visibility *queries, such as (i) what is the average visibility between a point (a differential area element) and a finite area or volume; or (ii) what is the average visibility between two finite areas or volumes.*

In this paper, we present a new data structure and an algorithm for rapidly evaluating such queries in complex scenes. The proposed approach utilizes a novel image-based discretization of the space of bounded rays in the scene, constructed in a preprocessing stage. Once this data structure has been computed, it allows us to quickly compute approximate answers to visibility queries. Because visibility queries are computed using a discretization of the space, the execution time is effectively decoupled from the number of geometric primitives in the scene. A potential hazard with the proposed approach is that it might require large amounts of memory, if the data structures are designed in a naive fashion. We discuss ways for substantially compressing the discretization, while still allowing rapid query evaluation. Preliminary results obtained via a partial implementation demonstrate the effectiveness of the proposed approach.

CR Categories and Subject Descriptors: *I.3.3 [Computer Graphics]: Picture/Image Generation — Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism* — color, shading, shadowing, and texture; radiosity; raytracing

Additional Key Words: *global illumination, shadow rays, form-factors, quantitative visibility, discrete scene representations*

The color images of this paper are also available at higher resolution at http://www.math.tau.ac.il/~daniel/cgi98/cgi98.html

1. Introduction

Efficient visibility computations have formed the core of many algorithms in the field computer graphics. Visibility information is necessary for displaying a 3D scene from a specified point of view with hidden surfaces removed, as well as for performing global illumination computations in order to produce photorealistic images.

The goal of this paper is to develop an algorithm that given a complex scene *rapidly* computes answers to *quantitative visibility queries*, such as: (i) what is the average visibility between a point (a differential area element) and a finite area or volume; (ii) what is the average visibility between two finite areas or volumes. We define average visibility as the ratio of unoccluded rays to the total number of rays in the ray pencil defined by the query; thus, the answer to such a query is a real number in the interval $[0, 1]$. In practice, computing this ratio exactly is infeasible; we are therefore interested in reliable and efficient approximative algorithms.

Why is this problem important? There are several very important applications requiring the efficient computation of quantitative visibility queries. Two examples of such applications are: (i) ray tracing and Monte-Carlo based global illumination algorithms for photorealistic image synthesis; (ii) radiosity and other finite-element based global illumination methods. These applications typically rely on point sampling (ray casting) in order to estimate the average visibility across a pencil of rays. This solution suffers from several important drawbacks. First, the resulting visibility estimates cannot be relied upon, since they are not accompanied by any kind of an error bound. Second, ray casting does not provide the means for trading off accuracy for speed: the only way to speed up the query is to reduce the number of rays in the pencil; however, this solution does not provide any means of error control. Third, the asymptotic cost of casting a ray is $O(\log N)$. Thus, ray casting can become quite computationally burdensome in very complex scenes, which might contain many millions of geometric primitives.

In this paper, we focus on the third drawback above. In order for the algorithm to handle extremely complex scenes, it must be relatively insensitive to the number of geometric primitives comprising the scene. Since ray-casting algorithms can produce visibility estimates in time roughly $O(\log N)$, our goal is to develop an algorithm whose observed complexity is sub-logarithmic in the number of scene primitives. The actual cost of a quantitative visibility query should depend on the desired accuracy: low accuracy requirements should result in very fast queries; when a more accurate approximation is required the query could take longer to compute.

We propose a new image-based approach towards attacking the quantitative visibility problem. The proposed approach utilizes a novel discretization of the space of bounded rays in the scene, constructed in a preprocessing stage. Once computed, this data structure is able to quickly provide approximate answers to visibility queries. Because visibility queries are computed using a discretization of the space, the execution time is effectively decoupled from the number of geometric primitives in the scene. A potential hazard with the proposed approach is that it might require large amounts of memory, if the data structures are designed in a naive fashion. We discuss ways for substantially compressing the discretization, while still allowing rapid query evaluation. Preliminary results obtained via a partial implementation demonstrate the effectiveness of the proposed approach.

2. Background

As mentioned above, there are several applications that require the efficient computation of average visibility across a pencil of rays in a complex scene.

To compute the direct illumination at a point p in a scene, ray-tracing and Monte-Carlo algorithms must establish the extent to which each of the light sources is visible from p. For point light sources, shooting a single ray from p to each light source is sufficient, but for handling area light sources, the average visibility across the solid angle subtended by the source is required. Even with point light sources, in order to obtained anti-aliased shadow boundaries it is better to consider a finite area on the illuminated surface, instead of a single point. This area could be determined, for instance, by projecting the pixel area through which the surface is visible back onto the surface. Note that the computed visibility value need not be exact: computed pixel colors will be eventually quantized before they can be displayed, so the required accuracy is finite.

Ray-tracers typically approximate the visibility to extended light sources by point-sampling the solid angle that they subtend. Unfortunately, such approximations converge only as $O(\sqrt{n})$, where n is the number of ray samples. Thus, a large number of rays may be required in order to obtain a high enough confidence that a value is computed to within a small enough variance [10]. Casting shadow rays can be the most time-consuming part of the computation is scenes with sufficiently many area light sources. This process can be sped up by many of the general ray-tracing acceleration schemes that were devised over the years [4]. In particular, Haines [6] describes an approach, referred to as the *light buffer*, specifically to accelerate the casting of shadow rays.

There were also a few alternative approaches to shadow ray casting, such as casting beams [9] and cones [1], but these approaches are not as general as ray casting, since they impose limiting assumptions on the geometry of the objects in the scene.

Radiosity and other finite-element based global illumination methods typically perform very large amounts of point-to-area and area-to-area form-factor computations. Again, these algorithms require only limited accuracy for a given form-factor, and therefore, various approximations can be used. A common approximation is to compute the unoccluded form-factor separately from the visibility and take their product. Visibility is typically estimated by point-sampling the areas and casting rays between the points [8]. Casting more rays normally results in a more accurate estimate of the visibility, but there are no error bounds on the resulting approximation.

Many schemes for accelerating visibility computations in radiosity were proposed by researchers over the years. These include shaft culling [7], global visibility preprocessing [15], backprojections [2], and the visibility skeleton [3]. Sillion and Drettakis [12, 14, 13] represent the scene by a hierarchy of clusters. Each cluster is treated as an isotropic attenuating volume, allowing to approximate quantitative visibility queries very quickly. Our approach is similar in spirit to theirs, but instead of isotropically attenuating volumes we use a higher-dimensional directional data structure inspired by discrete light fields [5, 11].

3. Discrete visibility

As stated in Section 1, our approach consists of discretizing the space of bounded rays in the scene. Once such a discretization has been constructed, we can quickly classify any bounded ray in the scene as obstructed or unobstructed. Because we use the discretization of the space instead of the original geometry of the scene, the classification time no longer depends on the number of geometric primitives in the scene. The price that we pay for this increase in classification speed is that the result of the classification is no longer guaranteed to be precise: certain obstructed rays might be classified as unobstructed, and vice versa. However, since our ultimate goal is to handle quantitative visibility queries for finite pencils of rays, rather than binary ray queries, the accuracy of individual ray queries is not crucial.

In designing our discrete visibility data structure, we are guided by the following two requirements: (i) the data structure should support fast classification of visibility along a bounded ray; and (ii) the data structure should be easy to compress. Consider the simplest possible discretization of the space of bounded rays. Since each bounded ray is defined by its two 3D endpoints, the space of all such rays is six-dimensional. We could simply discretize this six dimensional space, resulting in a 6D binary array. This discretization supports ray classification in constant time, but it's high dimensionality makes it difficult to get the memory requirements down to manageable amounts. It is also not clear how well we can compress such a volume, while still providing rapid retrieval of answers to ray queries.

Instead of using a 6D volume, we propose a novel 5D discretization of the space. The reduction of one dimension is traded off for a slight increase in the query retrieval time. The 5D data structure can be thought of as a 2D array of 3D arrays. Each element of the 2D array corresponds to a direction, which can be associated with a point on a unit sphere with polar coordinates (θ, ϕ). For each direction (θ, ϕ), we have a $n \times n \times d$ 3D array, which can be thought of as a stack of $n \times n$ binary images. Each image is obtained by projecting the contents of a slice of the scene perpendicular to the direction (θ, ϕ) (the projection is parallel to the direction).

Given a bounded visibility ray we first represent the ray in the form $(\theta, \phi, i, j, d_1, d_2)$, where (θ, ϕ) is the direction of the ray, (i, j) are the Cartesian coordinates or the point where the ray intersects a plane perpendicular to the direction (θ, ϕ) positioned in the middle between the ray endpoints, and (d_1, d_2) are the ray endpoints along the the given direction. Intuitively, this representation allows us to choose the element(s) in the 2D array, whose direction is closest to that of the ray. Once the discrete direction has been chosen, we can classify the visibility of the ray by examining the (i, j)-th element in all slices between d_1 and d_2 of the corresponding 3D array.

Generally, the direction of the visibility ray will not coincide with one of the directions that we have chosen for our discretization. In this case, we can simply use the nearest direction. A more sophisticated solution would be to examine the 3 closest directions on the sphere, and to combine the results using a weighted average. Having said that, in the remainder of this paper we will concentrate on the representation of the 3D array corresponding to one particular direction, since the compression and the visibility classification is performed in the same manner for each direction.

The 5D discretization described above is a binary array, in which the ray classification time is $O(d)$, where d is the number of scene slices along each direction. In the next section we describe a modification of this data structure that will allow us to both compress it effectively, and to classify ray queries faster (in time $O(\log d \log n)$, where n is the resolution of the images in each direction). We refer to the compressed 3D data structure corresponding to each direction as a *depth tree*.

4. Depth Tree

As said above we are now concentrating on the representation of the visibility corresponding to a particular direction. Given a ray (i, j, d_1, d_2) and a binary discretization of the scene, the ray visibility can be classified in time linear in the number of slices by examining the slices between d_1 and d_2. Instead we show how the binary discretization is translated to a binary-tree of depth images which can be compressed well and still, the visibility classification is faster.

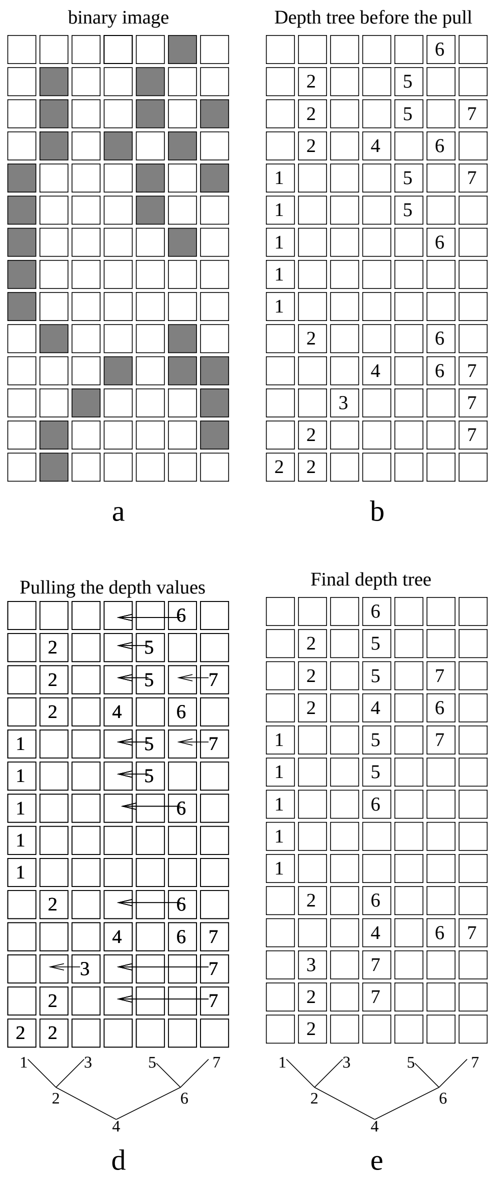


Figure 1. The binary and depth image.

In Figure 1 illustrates a 2D example where the slices are one dimensional - the vertical columns, and the depth axes is horizontal. We can see in (a) seven binary slices (columns), and in (b) their corresponding seven depth columns (slices), where each non-empty entry in the binary slices gets the depth values associated with a left-to-right view. Note that the depth image is still spatial coherent as the binary image, and can be well compressed along the axes perpendicular to the slices (columns in the figure). However, it would be much better to have a representation which has a 2D spatial coherence, since it could then be compressed even better. Note that depth image consists of many zero values which guarantee a 2D spatial coherency in the slices along the depth axes (or in the columns in the Figure 1(b)). Thus, the slices can be compressed using a quadtree which exploits the 2D spatial coherency. This mean that the access to a particular depth value requires $O(\log n)$, where n is the resolution of the slices. In Figure 3 shows six binary slices of the "5trees" scene containing five trees shown in Figure 2. Figure 4 shows the two depth images and their corresponding quadtrees.

As can be seen in Figure 1(d), the slices (or columns in Figure 1) can be regarded as nodes of a perfectly balanced binary-tree. This suggests that it may be possible to access the depth values in logarithmic time rather than linear. As we shall show below, the classification of a ray requires a search down the depth tree. That is $(O(\log d))$ steps, where d is the number of slices. Including the access into the quadtree the classification of a ray takes $O(\log d \log n)$. However, as we shall see, the depth trees are transformed such that most of the information in the tree is pushed as close as possible to the root of the tree so that on the average the number of steps down the binary-tree is better than logarithmic. See the fourth column in the binary-tree in Figure 1(f) - it is more loaded than its direct sons (notes 2 and 6), and much more loaded than the leaves (notes 1,3,5 and 7). Now we need to show (i) how to construct the depth tree and (ii) how to answer a query.

4.1. The construction of the depth tree

Given a binary image and a depth image the depth tree image is constructed by shifting data upwards the tree while leaving behind zero values. To explain it let's look at a given row in the depth images. Denote $h(x)$ and $v(x)$ the level and the value, respectively, at the x column in the binary-tree image. Note that the tree is balanced, x is the in-order index of the tree, and that $h(x)$ is number of least significant consecutive zeros in binary representation of x.

Initially $v(x) = x$ if and only if the voxel at x is not empty. All other values are null (See Figure 1(c)). Now, the value of $v(x)$ is shifted upwards the tree to the entry of x', $(v(x') = v(x))$ provided that (i)

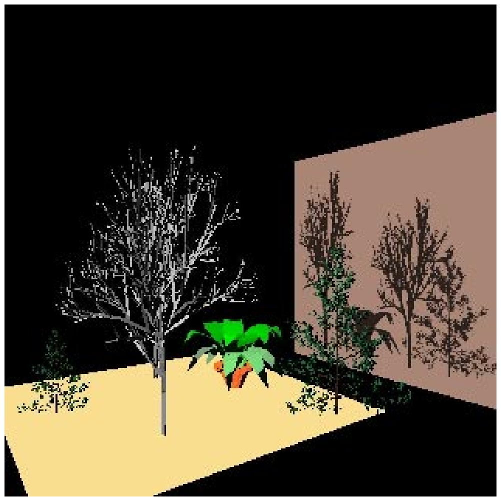


Figure 2. The "5trees" scene".

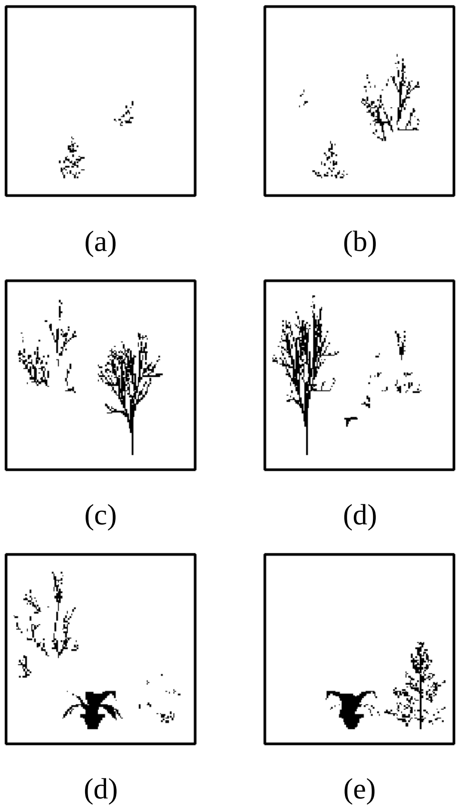


Figure 3. Five binary slices of the 5trees scence.

$x' < x$, (ii) $h(x) < h(x')$ and (iii) $v(x') = 0$. That is, a depth value shifts up to an empty entry of a higher level in the tree, but only form a right subtree. This *pull* operation starts from the root of the tree, pulls in as much information from its right subtree as possible under the above constrains. Figure 1(b) shows the initial values of the depth tree before applying the pull operations. Figure 1(c) shows the values which are shifted towards the root of the tree.

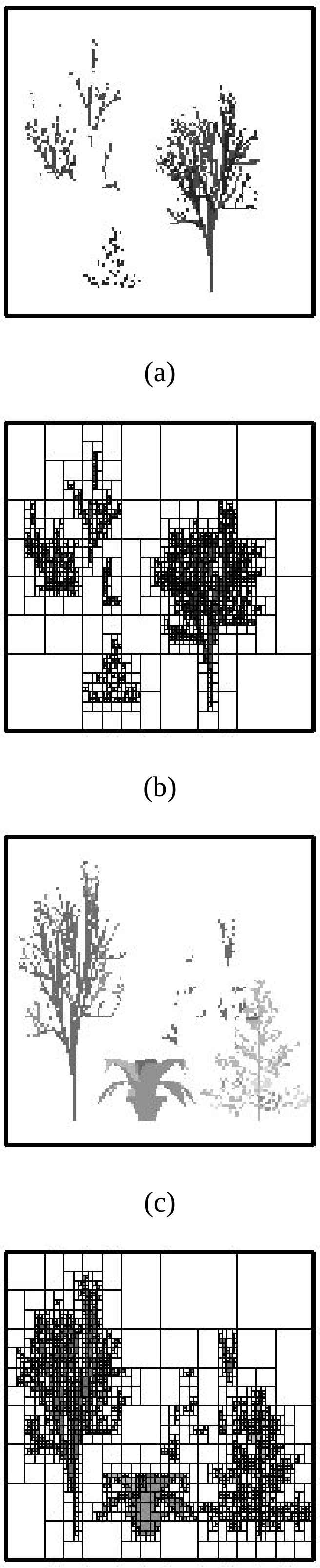
The final depth tree contains no "duplicate" information. Moreover, most of the information tends to be as close as possible to the root. Thus, the lower levels - the leaves or close to the leaves (more than half of the data) are mostly empty and can be well compressed. Note that columns 3 and 5 in Figure 1(d) are empty. This is typical to the leaves of the depth tree.

4.2. Retrieving the visibility

Once the depth tree has been constructed it can be used for fast ray classification. Given a ray (d_1, d_2) the visibility is retrieved by descending the tree top down. Denote by $d(tree)$ the depth of current tree node, and by $C(tree)$ the value stored in the node. Each node has two subtrees - the left subtree and the right subtree.

There are two cases to consider: the first case is when d_1, the start-point of the ray is to the left of $d(tree)$. If d_2, the end-point is to the right of the first occluder, then the ray is occluded. Otherwise we recurse in the left subtree, since there is no more useful information here. The second case is when d_1 is on or to the right $d(tree)$. If the first occluder is null (there are no occluders to the right of the current node) then the ray is unoccluded. Otherwise, there is an occluder at $C(tree)$. Now, the ray may starts to the right of it ($d_1 > C(tree)$) so we should recurse down to the right subtree. If the ray ends to the left the first occluder ($d_2 < C(tree)$) then it is unoccluded. Finally if the ray's endpoints are on either side of the occluder ($d_1 < C(tree) < d_2$) then the ray is occluded. Figure 5 shows the pseudo-code for the above procedure.

This visibility classifications requires at most $\log d$ steps, where each step requires at most one access to $C(tree)$ ($O(\log n)$) and only few comparisons. Note that the classifications procedure traverses the tree down to node of d_1. If that value is at a leaf then it is $\log d$ levels away, since our tree is perfectly balanced.



(a)

(b)

(c)

(d)

Figure 4. (a) and (c) are two depth images (2 and 4) of the 5trees scence after the pull operation, and their corresponding quadtrees in (b) and (d).

However, the value of d_1 could be higher up the tree or we might acquire sufficient information that allow to conclude the visibility and stop before this value is actually reached.

5. Results

The data structure that we have presented allows a fast point sampling of large and complex scenes by accessing a precomputed compressed representation of the visibility. We have applied our algorithm to complex scenes like the “pine tree” scene in Figure 6 which consists of 7,779 polygons, or the “5trees” scene in Figure 7, which consists of 25,816 polygons. These type of scenes are complex enough so that a the generation of their shadows is too expensive to be created in real-time by casting shadow rays. Our experimental tests where applied to one direction only. For the given direction the scenes where discretized, compressed, and the visibility queries where applied only for parallel rays.

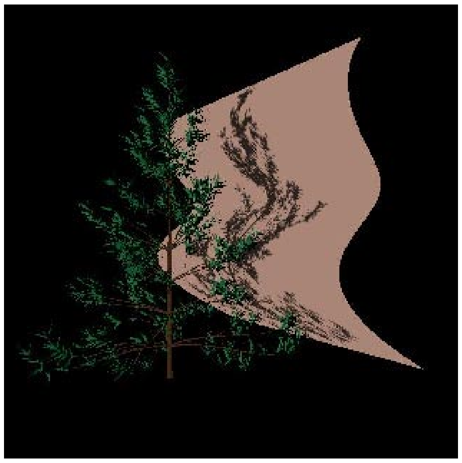


Figure 6. A pine tree casting shadow on a curved surface.

These scenes were discretized employing conventional rasterization hardware, which also yields the depth images on the z-buffer. For each slice, the scene was rendered with a front and back clipping planes defining the slice width. The non empty pixels of the slices (see Figure 3) where given their depth value by rendering the scene where the back clipping plane lay on the slice and no front clipping plane. Our implementation of the technique is not optimized. We use no culling technique which could trivially reject most of the scene at each slice, so we re-render everything for every image. Also our implementation of the quad-tree is rather naive. But still the construction of a 511x512x512 buffer and its compression takes about 5 minutes only. Almost half of that time is spent on rendering and most of the rest on the construction of the quad-trees.

slice resolution	128	256	512
number of slices			
127	130	410	758
255	164	510	906
511	202	610	1,119

Table 1. The number of KBytes of memory required for “pinetree” scene, 7,779 polygons.

slice resolution	128	256	512
number of slices			
127	178	258	829
255	250	716	1,118
511	336	962	1,506

Table 2. The number of KBytes of memory required for the “5 trees” scene, 25,816.

Tables 1 and 2 show the number of Kbytes required for the “pinetree” and the “5 trees” scenes. Note the direct effect of the resolution of the slice, and the sublinear dependency in the number of slices. This suggests that it pays off to have more slices. On the other hand, as can be seen in Figure 7, the slice resolution is not that visually critical to the quality of results. One should account the fact that typically such a ray query is only one sample out of many in a ray pencil so that it must not be accurate or in other words, the slice resolution must not be high. Note also that the size of the compressed depth tree is sublogarithmic in the volume resolution. A 512^3 volume of 128Mbytes is compressed down to 1,506Kbytes, about the same size as a 256^3 volume of 16Mbytes which is compressed down to 1,118Kbytes.

To measure timings we sent $1x10^6$ (1 million) random rays through the two different scenes at differ-

```
classify_ray(tree, $(d_1, d_2)$)
{
    **if** $(d_2 < d(tree))$  /* totaly to the left of the root */
        **return** classify_ray(tree->left, $(d_1, d_2)$);
    **else**
        $d_{occ} = C(tree)$;  /* get position of first occluder */
    **if** ( $d_{occ}$ == 0 or $d_2 < d_{occ}$)  /* either there is no occluder */
            /* or the ray ends before it */
        **if** $(d_1 < d(tree))$  /* starts to the left of the root */
            **return** classify_ray(tree->left, $(d_1, d_2)$);
        **else**
            **return** UNOCCLUDED;
    **else if** $(d_1 > d_{occ})$  /* totaly to the right of the first occluder */
            **return** classify_ray(tree->right, $(d_1, d_2)$);
        **else**  /* $d_1$ and $d_2$ are on either side of the occluder */
            **return** OCCLUDED;
}
```

Figure 5. Pseudo-code for classifying a ray against the depth tree.

ent resolutions and they alway seem to take the same amount of time (see Figure 8). For example, the “pine-tree” at resolution $128x128$ with only three slices took 4.4sec for all the million rays, and the “5trees” at resolution $512x512$ with 511 slices took only 5.5sec. We can see that as we increase the resolution the increase in the time taken to classify the rays is much less than logarithmic, even though our data structures have a logarithmic bound. This is because most rays are classified very high up near the root of the tree and only few are traced down to the leaves.

6. Future work

We have presented a mechanism for rapidly replying to ray visibility queries. Since it is based on the discretization of both the scene and the ray, the visibility of a single ray is just an approximation. As we mentioned in the paper, our final goal is quantitative visibility queries for finite pencils of rays, rather than binary ray queries. When the pencil of rays is very large it would require many ray queries where the accuracy of each individual ray is less significant. Thus, we intend to extend this work and construct a hierarchy of discretization of the scene on which we would be able to query on low resolution rays. Such coarse rays will yield a quantitative visibility that approximates the results of several finer rays. However, the success of replacing many finer rays by a single coarser ray is very much scene dependent. Thus, we can associate with each ray a confidence value which will indicate the probability that the approximation is successful. Such mechanism will provide means for trading off speed for accuracy.

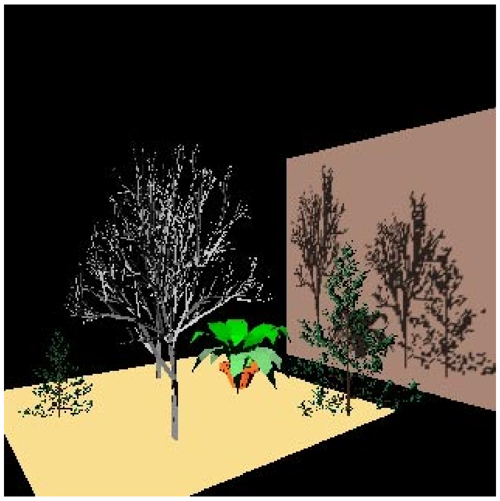
Acknowledgments

This work was supported in part by a grant from the Ministry Of Science Israel, and the French Ministry of Research and Technology (AFIRST).

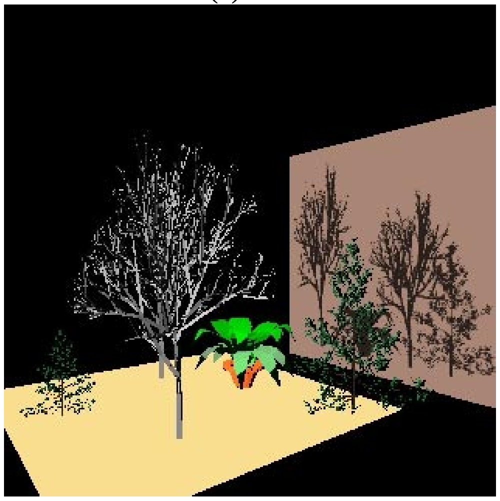
References

[1] J. Amanatides. Ray tracing with cones. *Computer Graphics*, 18(3):129–135, July 1984.

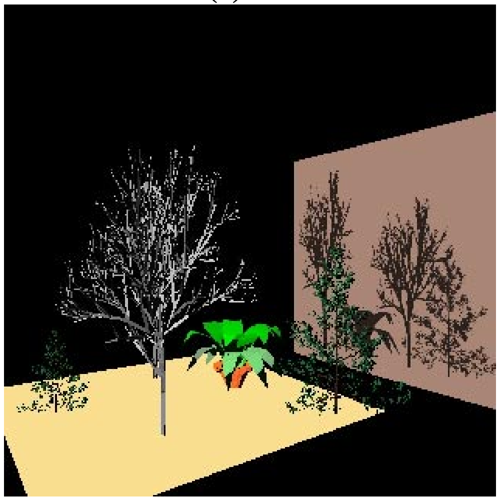
[2] G. Drettakis and E. Fiume. A fast shadow algorithm for area light sources using backprojection. In *Computer Graphics* Proceedings, Annual Conference Series, pages 223–230, July 1994.



(a) 128



(b) 256



(c) 512

Figure 7. The effect of the slice resolution.

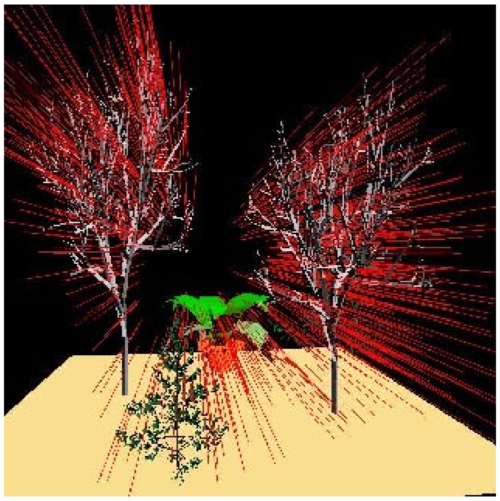


Figure 8. A visualisation of the random rays through the model, the intersected rays are colored in red.

[3] F. Durand, G. Drettakis, and C. Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *Computer Graphics* Proceedings, Annual Conference Series, pages 89–100, Aug. 1997.

[4] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, Inc., San Diego, California, 1989.

[5] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Computer Graphics* Proceedings, Annual Conference Series, pages 43–54, Aug. 1996.

[6] E. A. Haines and D. P. Greenberg. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, Sept. 1986.

[7] E. A. Haines and J. R. Wallace. Shaft culling for efficient ray-traced radiosity. In *Proceedings of the Second Eurographics Workshop on Rendering (Barcelona, Spain, May 13–15, 1991)*, May 1991.

[8] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, July 1991.

[9] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–128, July 1984.

[10] M. E. Lee, R. A. Redner, and S. P. Uselton. Statistically optimized sampling for distributed ray tracing. *Computer Graphics*, 19(3):61–68, July 1985.

[11] M. Levoy and P. Hanrahan. Light field rendering. In *Computer Graphics* Proceedings, Annual Conference Series, pages 31–42, Aug. 1996.

[12] F. Sillion. Clustering and volume scattering for hierarchical radiosity calculations. In *Proceedings of the Fifth Eurographics Workshop on Rendering (Darmstadt, Germany, June 13–15, 1994)*, pages 105–117, June 1994.

[13] F. Sillion and G. Drettakis. Feature-based control of visibility error: A multi-resolution clustering algorithm for global illumination. In *Computer Graphics* Proceedings, Annual Conference Series, pages 145–152, Aug. 1995.

[14] F. X. Sillion. A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):240–254, Sept. 1995.

[15] S. Teller and P. Hanrahan. Global visibility algorithms for illumination computations. In *Computer Graphics* Proceedings, Annual Conference Series, pages 239–246, Aug. 1993.