Image and Intervisibility Coherence in Rendering

Joseph Marks, Robert Walsh, Jon Christensen, and Mark Friedell

Aiken Computation Laboratory
Harvard University
Cambridge, Massachusetts 02138

Abstract

Researchers in computer graphics have long regarded the exploitation of image coherence as one of the fundamental opportunities for improving the efficiency of image rendering. We describe in this paper a theoretical and experimental investigation of the potential benefits of exploiting this phenomenon through the use of hybrid rendering strategies that combine area-sampling and point-sampling techniques. We also examine the impact of a related phenomenon, intervisibility coherence, on the calculation of form factors for radiosity-based rendering.

Although the prospect of exploiting image coherence to render many pixels simultaneously is intuitively appealing, this study indicates that the potential for reducing rendering time in this way is surprisingly limited in most circumstances. Nevertheless, the hybrid rendering algorithms we describe may be of some practical significance for high-resolution images that exhibit substantial image coherence. A similar observation holds for the exploitation of intervisibility coherence to speed up radiosity-based rendering: the potential for reducing rendering time by exploiting this kind of coherence appears limited, but the hybrid algorithm we describe may be of some practical use for very finely discretized scenes that exhibit substantial intervisibility coherence.

Keywords: image rendering, image coherence, intervisibility coherence, projective rendering, ray tracing, radiosity.

1. Introduction

Consider an arbitrary pixel in a typical synthetic image. There is a high probability that this pixel depicts part of the same primitive surface descriptor, e.g., a polygonal face, as that depicted by the pixel's 8-connected neighbors. This phenomenon is known as *image coherence*.

The computer-graphics literature chronicles many attempts to reduce rendering time by exploiting image coherence in several different ways. These efforts, which seek to render simultaneously all the pixels in a coherent region, previously have not been shown to be superior to other rendering algorithms, even when the image is highly coherent. Nonetheless, the prospect of rendering many pixels simultaneously is intuitively appealing and the exploitation of image coherence is frequently cited as a fundamental opportunity for improving the efficiency of image rendering.

The most aggressive attempts to exploit image coherence are area-sampling algorithms, which attempt to find the largest coherent regions in the image. Area-sampling algorithms have been developed by Warnock [WARNOCK], Weiler and Atherton [WEILER & ATHERTON], and Heckbert and Hanrahan [HECKBERT & HANRAHAN], among others. The Warnock algorithm was the first and is among the most famous algorithms in computer graphics. It is described briefly below in terms of two coroutines: *looker* and *thinker*. Rendering begins by setting *window* equal to the entire screen and invoking looker.

procedure looker:

1. Invoke thinker with the current window as its argument. If thinker succeeds, return immediately; otherwise, go on to step 2.

2. Subdivide evenly the window into 4 subwindows.
3. Call recursively the looker for each of the subwindows.

procedure thinker:

1. If the window is empty, tile the window with the background color and return success.
2. If the window contains a face, S, that completely fills the window and is everywhere in the window closer to the viewer than the plane of any other face in the window, tile the window with face S. Return success.
3. Return failure.

Variants of the Warnock algorithm, which we refer to as *window-sampling algorithms*, may be distinguished from the larger set of area-sampling algorithms by the way they subdivide regions of the screen. Window-sampling algorithms subdivide rectangular windows within the screen into smaller rectangular windows, while other area-sampling algorithms (e.g., [WEILER & ATHERTON]), may subdivide the screen in arbitrarily complex ways.

In Section 2, we offer a theoretical view of why previous efforts to exploit image coherence via window sampling have not yielded superior rendering algorithms, and we suggest a hybrid strategy that we believe leads to the fullest use of image coherence that is possible in practical circumstances.

The resulting improvements in rendering efficiency for various degrees of image coherence are reported in Sections 3 and 4, which describe two rendering algorithms that use our hybrid strategy. One algorithm is based on projective geometry and the other is based on ray tracing. Both algorithms harness existing algorithms to render the complex areas of an image, and use a window-sampling algorithm to render the coherent areas. When applied to images containing significant image coherence, this approach serves to accelerate the non-window-sampling half of the hybrid algorithm. In the worst case, when applied to images with little or no coherence, the hybrid algorithms reduce immediately to the non-window-sampling halfs of the respective algorithms with negligible performance degradation.

In Section 5, we outline how our hybrid approach has been used to exploit a similar form of coherence -- inter-visibility coherence -- in the calculation of form factors for radiosity-based rendering.

2. Theoretical Perspective

Figure 2.1 gives the times required by the Warnock window-sampling algorithm and the Romney scan-line algorithm to render the six test images presented in Figure 2.2. For all 6 scenes, the running time of the scan-line algorithm is shown to be less -- in most cases much less -- than the running time of the window-sampling algorithm. This is typical of the speed advantage over the Warnock algorithm of all rendering algorithms now in common use. Since the general inefficiency of the Warnock algorithm is not explained by approximate time-complexity measures [SUTHERLAND et al.], we assume that the complete time-complexity function for the Warnock algorithm includes a large constant of proportionality and/or significant lower-order terms.

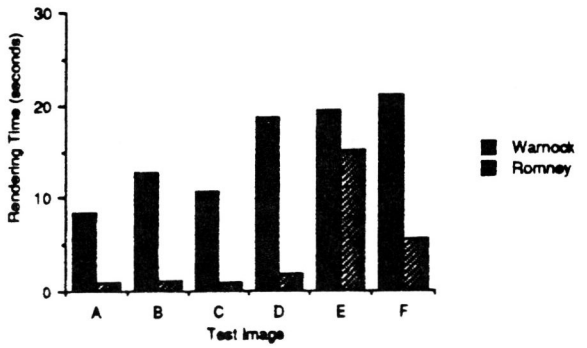
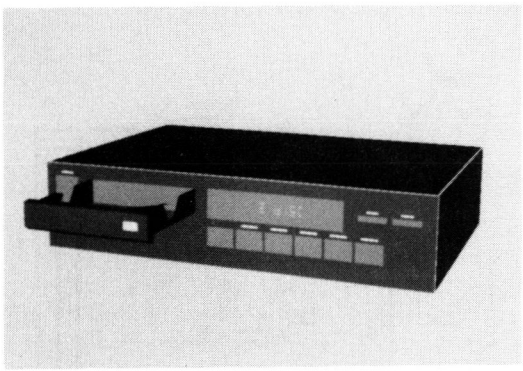


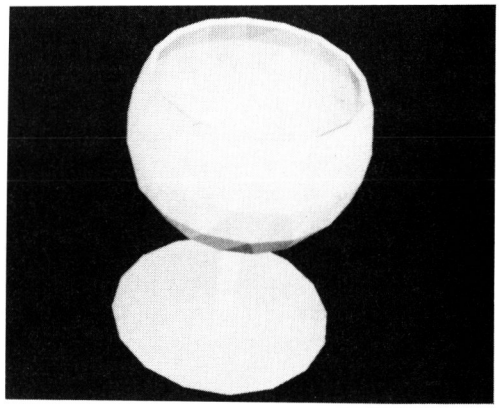
Figure 2.1 Comparative Rendering Times

In spite of the apparent disadvantages of window sampling, it is extremely efficient when rendering a large window within a coherent image region. Ideally, we would like to render these coherent regions with a window-sampling algorithm and render all other portions of the image with some other, more generally efficient algorithm. To do this in practice, we must be able to predict where the coherent regions of an image lie. If this can be done reliably with no appreciable cost, we would be able to construct hybrid rendering algorithms whose rendering-time speedup due to the exploitation of image coherence would approach the theoretical maximum.

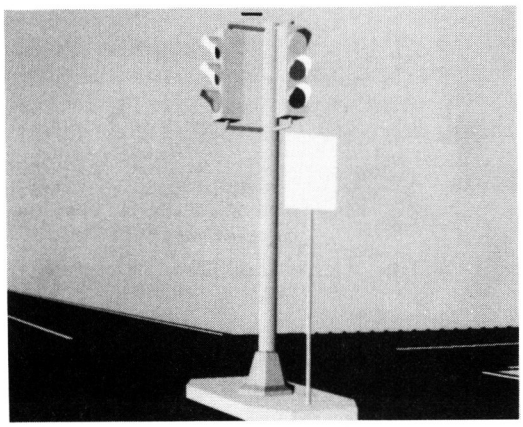
We address the problem of predicting the locations of coherent regions by assuming that rendering begins with a window-sampling technique. Then, for each window sample, we need to determine whether to continue with the window-sampling approach or to convert to an alternative rendering technique. It would be advantageous to continue the window-sampling approach if further subdivision would produce a sample window, C, such that the cost of rendering C via window sampling plus the cost of producing C would be less than the cost of rendering the pixels in C



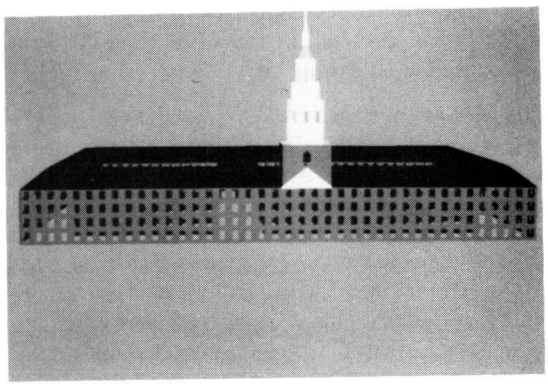
A



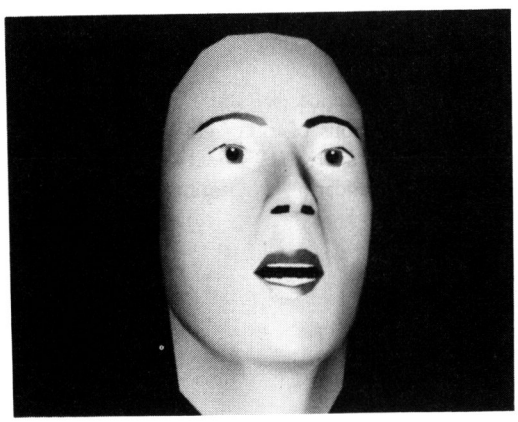
B



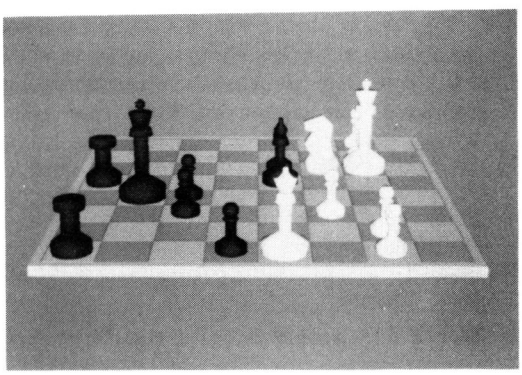
C



D



E



F

Figure 2.2 Six Test Scenes

with the alternative technique.

To make this determination, we formulate an analytic model of window sampling and subdivision and use it to derive an expression that predicts the number of window samples needed to render an image with a window-sampling algorithm. To validate our model, we compare predicted and actual samples for several test scenes. After validating the analytic model, we use it to derive an expression that estimates rendering cost per pixel for the most economically renderable subwindow that could be generated from a given sample window.

Our analytic model follows that in [SUTHERLAND et al.] and presumes a three-dimensional scene clipped to a viewing frustum and mapped by a perspective transformation into a three-dimensional image space E_s pixels wide by E_s pixels high. In this ideal scene, shown in Figure 2.3, there are D identical layers of identical square faces, each layer parallel to the image plane and measuring E_s x E_s. D is therefore the depth complexity of the scene. If we specify D and the total number of faces in the scene, F, we calculate the number of visible faces, F_v, as F/D.

Initially, we are interested in analyzing window-sampling processes of which the Warnock algorithm is the exemplar. To facilitate this analysis, we constrain the dimensions and the arrangement of the square faces in the scene. Within each layer of faces, we require a 1/2-pixel gap between the edges of adjacent faces. The edge length of a face, E_f, is therefore slightly less than $\sqrt{E_s^2/F_v}$. Further, we also require that at some granularity of window subdivision a window will contain exactly 1 complete face, aligned with the right and bottom borders of the window. This window will therefore contain a 1/2-pixel gap between left edge of the face and the left border of the window as well as a 1/2-pixel gap between the top edge of the face and the top border of the window. These conditions are illustrated in Figure 2.3. As described below, the required face registration plays a role in analyzing window subdivision as a function of face-window overlap, and the 1/2-pixel gaps are used in modeling window subdivision due to the presence of face edges.

Given an ideal scene, the behavior of a window-sampling process can be separated into two activities, *sorting* and *splitting*. If a window sample yields more than 1 visible polygon, the window will be subdivided into 4 subwindows, and all polygons in the sample will be sorted into the them; no polygon splitting will be necessary. If a sample yields exactly 1 visible face that fills the window, the window will be rendered completely. Otherwise, the window is not filled completely, the sample window will be subdivided as before, and each of the D faces in the sample will be split along the boundaries of the subwindows. (This analysis is made simpler by assuming that faces are literally split, although in practice the faces would probably be copied.) Sorting and splitting continue until each sample window is filled completely by a visible face or until the sample window covers 1 pixel or less.

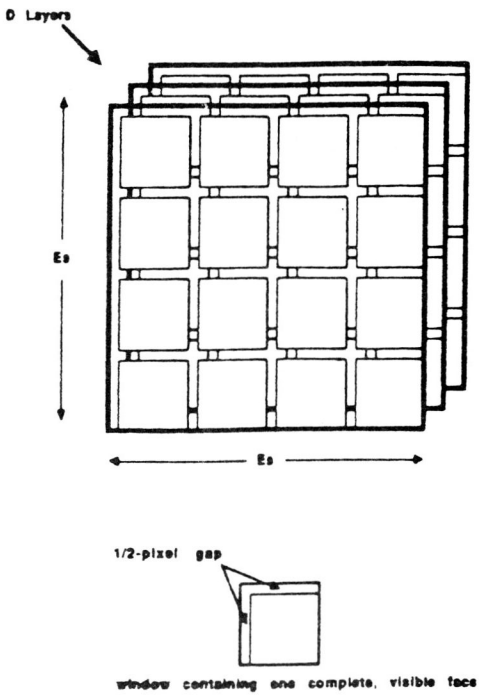


Figure 2.3 An Ideal Scene Description

To compute the number of window samples required to render an image of an ideal scene, we first determine the number of samples required during the sorting phase of rendering. This number will depend on F_v, and we refer to it as $S_{sort}(F_v)$. If a sample contains exactly 1 visible face, $S_{sort}(F_v)$ is 0. Otherwise, the total number of sorting samples required to render the scene will be, in addition to the sample for the original window, the sum of the sorting samples required to render each of the 4 identical subwindows. Therefore,

$$S_{sort}(F_v)=1+4S_{sort}\left(\frac{F_v}{4}\right)$$

$$=\sum_{i=0}^{\log_4 F_v-1} 4^i+0$$

$$=\frac{F_v-1}{3}. \qquad (2.1)$$

In addition to the $(F_v-1)/3$ sorting samples, there is some number of splitting samples for each of the F_v windows containing 1 visible face and a 1/2-pixel gap along the left and top borders of the window. This number, which depends of E_f and is referred to as $S_{split}(E_f)$, is 1

when E_f is 1. Otherwise, $S_{split}(E_f)$ is 2 (1 for the initial sample and 1 for the lower-right window, which is rendered), plus 2 times the number of samples required to render a subwindow with a 1/2-pixel gap along 1 edge (the lower-left and upper-right subwindows), plus the number of samples required for the upper-left subwindow (whose gap configuration is identical to the original sample window). By referring to the number of samples required to render a window with a 1/2-pixel gap along one edge of length n as $S_{split_{1-edge}}(n)$, we can write

$$S_{split}(E_f)=2+2S_{split_{1-edge}}\left[\frac{E_f}{2}\right]+S_{split}\left[\frac{E_f}{2}\right]$$

$$=2\sum_{i=1}^{\log_2 E_f} S_{split_{1-edge}}\left[\frac{E_f}{2^i}\right]+2log_2E_f+1 \quad (2.2)$$

When n is 1, $S_{split_{1-edge}}(n)$ is 1. Otherwise, it is 3 (the sample for the original window and 1 sample each for the upper-right and lower-right subwindows, which will be rendered), plus the number of samples required for the upper-left and lower-left subwindows, which also have gaps along their left edges. Therefore,

$$S_{split_{1-edge}}(n)=3+2S_{split_{1-edge}}\left[\frac{n}{2}\right]$$

$$=3\sum_{i=0}^{\log_2 n-1} 2^i+n$$

$$=4n-3.$$

Substituting this expression for $S_{split_{1-edge}}(n)$ into Equation 2.2 yields

$$S_{split}(E_f)=2\sum_{i=1}^{\log_2 E_f}\left[4\frac{E_f}{2^i}-3\right]+2log_2E_f+1$$

$$=8E_f-4log_2E_f-7 \quad (2.3)$$

By combining Equation 2.1 with Equation 2.3 and substituting $\sqrt{E_s^2/F_v}$ for E_f, we find $S(F_v)$, the total number of samples required to render an image with F_v visible faces:

$$S(F_v)=\frac{F_v-1}{3}+F_v\left[8\sqrt{\frac{E_s^2}{F_v}}-4\log_2\sqrt{\frac{E_s^2}{F_v}}-7\right]$$

$$S(F_v)=\frac{F_v-1}{3}+8E_s\sqrt{F_v}-4F_v\log_2\sqrt{\frac{E_s^2}{F_v}}-7F_v \quad (2.4)$$

In real scenes, the distribution of scene complexity may vary widely from the uniform distribution of ideal scenes. Consequently, we wish to test the ability of our model to accurately predict the behavior or real rendering processes. To do so, we empirically verify the predictive power of Equation 2.4 with an instrumented implementation of the Warnock algorithm that counts the number of window samples required during rendering. This implementation was used to render the 6 images shown in Figure 2.2, and the predicted and actual operation counts are shown in Figure 2.4.

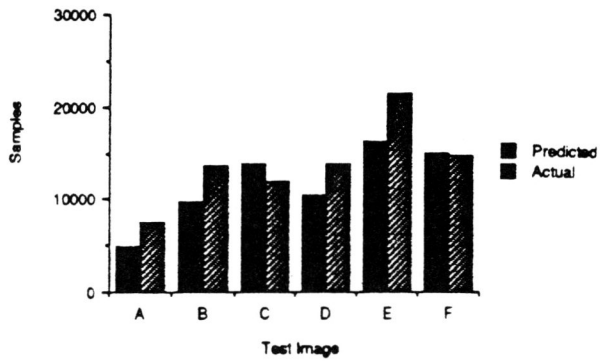


Figure 2.4 Predicted and Actual Window Samples

We now re-examine the behavior of window-sampling processes with the goal of estimating for any given window, W, the *rendering cost per pixel* of the most economically renderable subwindow that can be derived from W by subdivision. In this discussion, F_W, F_{v_W}, E_W, and E_{f_W} are defined for W as F, F_v, E, and E_f are defined for the complete scene. Our analysis assumes the same face-computation cost for sorting a face into a subwindow or for splitting it into 4 subwindows.

During the sorting phase of rendering, the number of face computations required to render a window, W, containing F_w faces, $C_{sort_W}(F_W)$, is 0 if F_W is D. Otherwise, it is F_W (to sort each face into the appropriate subwindow) plus the number of face computations required for each of the 4 subwindows:

$$C_{sort_W}(F_W)=F_W+4C_{sort}\left[\frac{F_W}{4}\right]$$

$$=\sum_{i=1}^{\log_4\frac{F_W}{D}} F_W+0$$

$$=F_W\log_4\frac{F_W}{D}$$

As the splitting phase of rendering begins, each of the subwindows of W containing exactly 1 visible face is further subdivided into 4 windows. One of these -- the lower-right one -- is rendered completely when it is sampled. The number of pixels in each of these completely rendered windows is ${E_{f_W}}^2/4$.

For these windows, the largest to be rendered as part of a single sampling operation, we can compute the number of face-processing operations per pixel. There were $F_W \log_4 \frac{F_W}{D}$ operations needed during the sorting phase of rendering, which produced F_{v_W} windows. Therefore, each of these F_{v_W} windows can be "charged" with $(F_W \log_4 \frac{F_W}{D})/F_{v_W}$ operations performed during the sorting phase of rendering. During the splitting phase, each of the D faces in each of the F_{v_W} windows is processed as it is split into 4 subwindows. The lower-right subwindow, containing $E_{f_W}/4$ pixels is then rendered. Therefore, the number of face-processing operations per pixel is

$$\frac{\frac{F_W \log_4 \frac{F_W}{D}}{F_{vW}} + D}{\frac{{E_{f_W}}^2}{4}}. \tag{2.5}$$

Substituting F_W/D for F_{v_W} and $\sqrt{(E_W^2 D)/F_W}$ for E_{f_W} in expression 2.5 yields

$$\frac{4F_W \log_4 \frac{F_W}{D} + 4F_W}{E_W^2}. \tag{2.6}$$

Consider now a different window-sampling algorithm that requires processing time for each sample that is proportional to the total number of faces in the scene. (This is the case for the hybrid beam-ray tracing algorithm presentation in Section 4.)

We know from Equation 2.1 that $(F_{v_W}-1)/3$ samples will be required during the "sorting" phase of rendering to yield F_{v_W} windows containing 1 visible face. As "splitting" begins, each of the F_{v_W} windows will be subdivided and the lower-right subwindow will be rendered completely. This will require 5 more samples for each F_{v_W} window. Hence, the number of samples per rendered pixel is

$$\frac{\frac{F_{v_W}-1}{3F_{v_W}} + 5}{\frac{{E_{f_W}}^2}{4}}.$$

Each sample requires time proportional to the total number of faces in the scene, F. Therefore, using the identities $F_{v_W} = \frac{F_W}{D}$ and $E_{f_W} = \sqrt{\frac{E_W^2 D}{F_W}}$, the rendering cost per pixel is proportional is

$$F \frac{16F_W - D}{E_W^2 D}. \tag{2.7}$$

3. Potential Speedups for Projective-Geometry-Based Rendering

To measure the potential for rendering-time speedups in practical projective-geometry-based rendering algorithms, we have prepared a hybrid algorithm in which area sampling is performed by the Warnock looker and thinker coroutines, and point sampling is accomplished by a Z-buffer. We chose a Z-buffer point-sampling algorithm for this experiment because of its simplicity. (We use the term *projective-geometry rendering algorithm* to refer to a conventional, non-ray-tracing rendering algorithm that operates on a scene which has been mapped by a perspective transformation into a three-dimensional image space.)

In our hybrid algorithm, the looker process has been modified slightly. When invoked, the looker counts the number of faces in its window, estimates the depth complexity of the window, and estimates the cost of rendering a portion of window via continued window sampling by evaluating the expression

$$\frac{F_W \log_4 \frac{F_W}{D} + F_W}{E_W^2}, \tag{3.1}$$

whose value is proportional to Expression 2.6. F_W is the number of faces in the window, D is the depth complexity of the scene, and E_W is the edge length of the sample window in pixels. The value of Expression 3.1 is compared to a threshold which represents the cost of rendering the same pixels by point sampling via the Z-buffer. If the computed cost exceeds the threshold, the window is passed to the Z-buffer routine.

The threshold for selecting point sampling is determined empirically to capture an accurate balance between window sampling and point sampling. This balance point reflects the efficiencies of the constituent window-sampling and point-sampling algorithms and their implementations in the hybrid renderer. If the threshold is set arbitrarily to 0, the hybrid algorithm will immediately convert to point sampling and render the entire image with the Z-buffer; if the threshold is set to infinity, the Warnock algorithm will

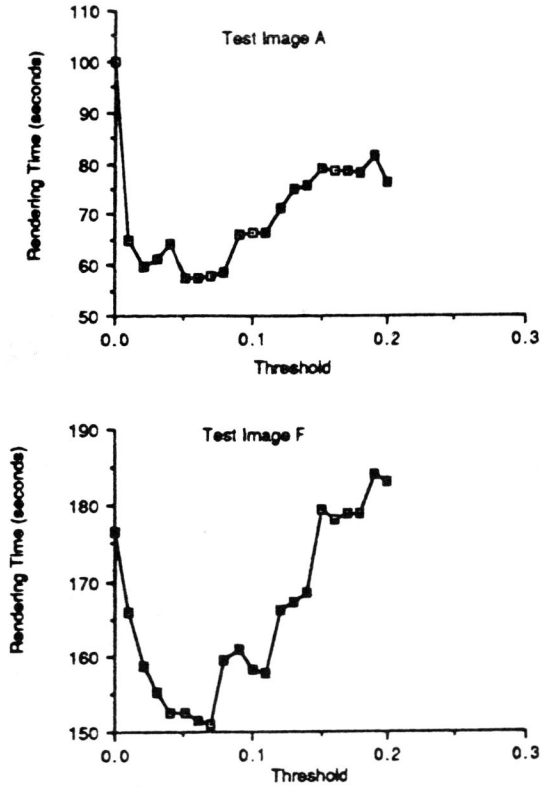


Figure 3.1 Rendering Time vs. Threshold Value

be used exclusively. To determine an appropriate threshold value, the times to render a few typical test scenes are recorded as the threshold is varied along a ramp of values. If the insights of Section 2 are valid, we would expect the rendering time to initially decrease as the threshold value is increased from 0 (and some window sampling is "blended in"), then increase as the utility of additional window sampling is exhausted. This behavior is shown in Figure 3.1 for test scenes A and F, which represent the maximum and minimum performance improvements for the test scenes in Figure 2.2. Using these empirically derived performance data, we select a threshold value that corresponds to a minimal rendering time over the set of test scenes. In this case, we choose 0.04. With that value, the hybrid algorithm produces a speedup over the Z-buffer algorithm of 30% for scene A and 8% for scene F. These speedups correspond well with the relative presence of image coherence in the test images.

By comparing the value of Expression 3.1 to a fixed threshold, we assume a fixed rendering cost per pixel for the Z buffer. Because the Z-Buffer technique incorporates a high overhead cost per pixel, this is a reasonable assumption for small values of F_W. Further, since the cost of rendering as a function of the number of faces increases more rapidly with window sampling that it does with a Z buffer, our simplification produces the desired result for large values of F_W, i.e., we select point sampling via the Z buffer.

The rendering times given in Figure 3.1 are for images at 1024 x 780 spatial resolution and are exclusive of pixel tiling. We assume that tiling is performed by a processor in the graphics controller that accepts rectangle-fill commands (windows from the thinker coroutine) and vector-draw commands (scan-line spans from the Z-buffer).

4. Potential Speedups for Ray-Tracing-Based Rendering

The recent focus of computer-graphics research on ray-tracing-based rendering has produced dramatic improvements in the efficiency of this class of rendering algorithms. In fact, many now believe that no further significant improvements in efficiency are achievable without the use of some form of parallelism.

Beam tracing is a form of parallelism that seeks to exploit image coherence by tracing simultaneously all rays through a coherent area of the screen. This is possible because all rays through the coherent area will intersect the same surfaces, by definition. Although beam tracing has obvious appeal for rendering large coherent areas of an image, previous beam-tracing algorithms suffer from the performance degradation observed in projective-geometry-based rendering algorithms that attempt to use area sampling on insufficiently large screen areas. When this point is reached with beam tracing, however, a point-sampling process -- ray tracing -- can be used. Thus, a strategy for hybrid beam-ray tracing is to trace beams through the more coherent areas of an image and to trace rays elsewhere.

To understand better the prospects for exploiting coherence in ray-tracing-based rendering, we have constructed a hybrid algorithm that combines beam tracing with ray tracing using a extension of Warnock's approach to area sampling. This beam-tracing algorithm is different from the beam-tracing algorithm due to Heckbert and Hanrahan[HECKBERT & HANRAHAN], which is a generalization of the Weiler-Atherton hidden-surface algorithm[WEILER & ATHERTON].

Window	A rectangular region of the screen.
Beam	An infinite pyramidal volume. A truncated beam is formed by removing the top of a beam through intersection with a planar surface.
Lateral Rays	The four rays that form the edges of a beam.
Original Beam	The lateral rays of an original beam are rays from the eye through the corners of a window.
Reflected Beam	A truncated beam that is formed when another beam is incident upon a reflecting face.
Refracted Beam	A truncated beam that is formed when a beam is incident upon a refracting face. When the refractive index is not 1.0, the refracted beam will be distorted.
Beam Segment	The segment of a beam between two planar surfaces that intersect the beam.
Beam Path	A sequence of connected beam segments produced by intersections with reflecting and refracting surfaces that ends with a truncated beam.
Hit	A hit occurs when a beam intersects completely with a face (the hit face), and the resultant beam segment is empty.
Partial Hit	A partial hit occurs when no face intersects a beam completely.
Blocked Hit	A blocked hit occurs when a beam intersects completely with a face, but the resultant beam segment contains one or more additional faces.

Figure 4.1 Glossary of Terms for Beam-Ray Tracing

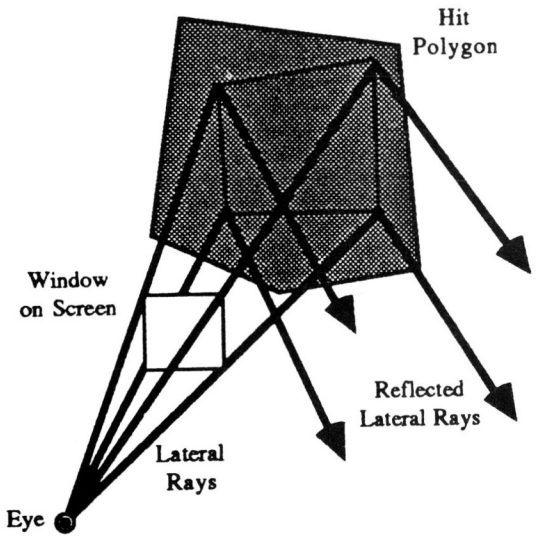


Figure 4.2 An Illustration of Beam Tracing

A glossary of terms for hybrid beam-ray tracing is given in Figure 4.1. Figure 4.2 illustrates the optimal situation for beam tracing. The beam passing through a given window on the screen intersects completely with a face in the scene, and the resulting beam segment contains no other faces. (To simplify the presentation, faces are assumed to be convex. This is not essential to the algorithm, but it does permit a simplification of the intersection computation: if the four lateral rays of a beam intersect a convex face, then the whole beam intersects the face.) The diffuse-reflection contribution from the hit face can now be calculated for the entire window. If the hit face is specularly reflective, the reflected beam is found by reflecting the incident lateral rays of the beam. The specular-reflection contribution from the hit face is then computed by recursively tracing the reflected beam through the scene. A similar procedure can be followed for refracted beams, although additional steps are required to cope with the possible distortion of the refracted beam. For a discussion of how to extend the hybrid beam-ray tracing algorithm to simulate more sophisticated illumination effects, including general refraction, shadows, and texture-mapping, see [MARKS et al.].

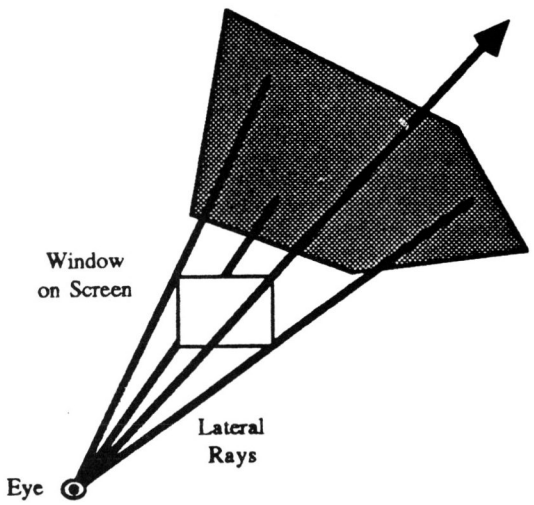


Figure 4.3 A Partial Hit

For the optimal case afforded by a coherent area of the image, beam tracing a large window on the screen can be much more efficient than ray tracing it. For example, the cost of tracing a beam like the one in Figure 4.2 is the cost of ray tracing four lateral rays and performing spatial queries to verify that the beam segments on the subsequent beam path are empty. Although a spatial query of this kind is a complex operation, not every polygon in the scene need be tested for intersection with a given beam segment: a data structure like the spatial enumeration of [FUJIMOTO et al.] is used to restrict attention to only those polygons in the neighborhood of the beam segment. Further efficiencies result from the fact that the spatial query need only determine whether an intersection has occurred, and need not return a geometric description of the intersection. Computing the reflected beams generated at each reflecting surface incurs only the cost of reflecting four incident lateral rays. Ray tracing a N x N pixel

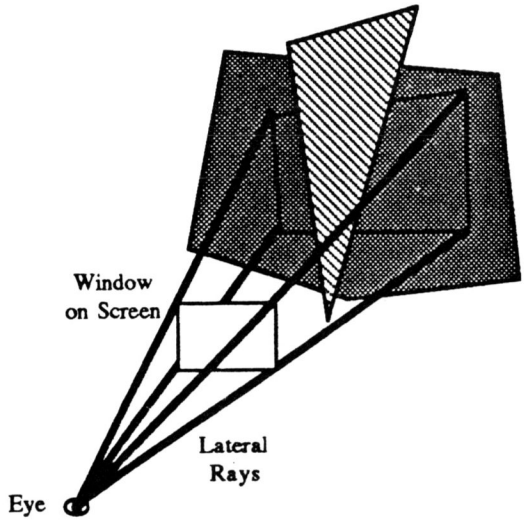


Figure 4.4 A Blocked Hit

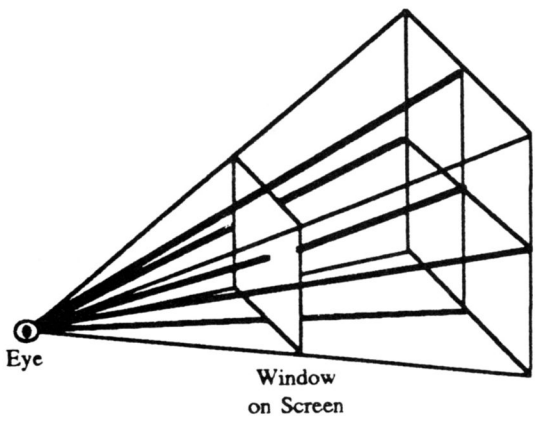


Figure 4.5 Subdivision of a Window on the Screen

window would require tracing and reflecting N^2 rays; antialiasing by oversampling might require tracing many more rays.

Unfortunately, the optimal case is not the only one that can occur. Figure 4.3 illustrates the case of a partial hit. The lateral rays of the beam do not all have the same surface of nearest intersection, hence the beam does not completely intersect any face. Figure 4.4 illustrates the case of a blocked hit. The lateral rays share the same surface of nearest intersection, so the beam intersects the face completely. But the beam does not intersect the face exclusively: the beam segment contains another face. If the newest beam segment on a beam path experiences a partial or a blocked hit, it is clear that this and subsequent beam segments will not generate coherent contributions to the window associated with the beam path.

The two options for coping with partial and blocked hits are subdivision of the beam path and ray tracing. Figure 4.5 illustrates the procedure for subdividing a beam path. The associated window is subdivided regularly into four smaller windows. Rays are traced through the corners of each of these smaller windows. These rays are the lateral rays that define four new beam paths, one for each of the smaller windows. Some of the new beam paths will generate hits immediately, whereas some will require more subdivision to generate further hits.

As the amount of subdivision required before producing a hit increases, so does the rendering cost per pixel for the associated window. If the new beam paths resulting from a subdivision are unlikely to generate further hits while their associated windows are still large enough to realize savings in computation over ray tracing, it would be best to abandon the area-sampling approach. Any ray-tracing algorithm can be used as the alternative. Choosing between continued subdivision and ray tracing is done by evaluating Expression 2.7,

$$F\frac{16F_W-D}{E_W^2D},$$

to estimate the cost-per-pixel of continued beam tracing. Here F is the total number of faces in the scene, D is the depth complexity of the scene, F_W is the number of faces in the window, and E_W is the edge length of the window in pixels. If the value of Expression 2.7 exceeds an empirically determined threshold (which is determined using the same method that we used for the projective-geometry renderer), beam tracing in this window is discontinued, and further rendering within the window is performed by the ray tracer. The number of faces in a window is found when the spatial query is performed for the beam.

The hybrid beam-ray tracing algorithm described above, which was implemented on a Sun 3 workstation, was used to render the images in Figures 4.6 and 4.9 at three different resolutions. (A simple Lambertian illumination model was used so that the program timings reflect more closely the cost of hidden-surface processing.) For comparison purposes, the same images were rendered using the embedded ray tracer alone. The rendering times of both algorithms are illustrated in Figures 4.8 and 4.11. The timings reported here are for the complete rendering process, from reading the scene description to writing the image into a frame buffer.

The very simple scene in Figure 4.6 was chosen because the image exhibits a very high degree of image coherence. Figure 4.7 illustrates how the hybrid algorithm takes advantage of this image coherence by rendering most of the image using beam tracing. At an image resolution of 1024 x 780, 84% of the pixels were rendered by beam tracing. Although the cost-estimation procedure for deciding between further beam subdivision and ray tracing correctly identifies most of the coherent areas of the image,

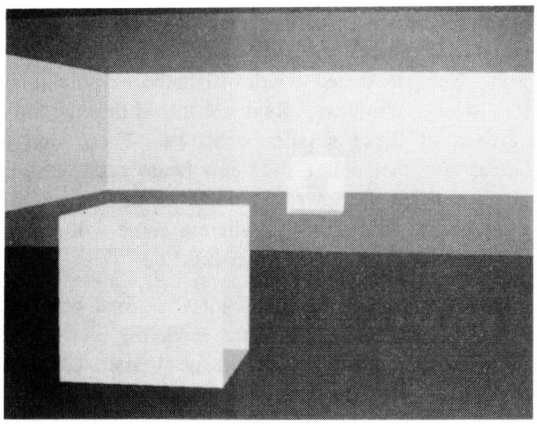


Figure 4.6 Image G: Concentric Cubes Reflected in a Mirror (18 faces)

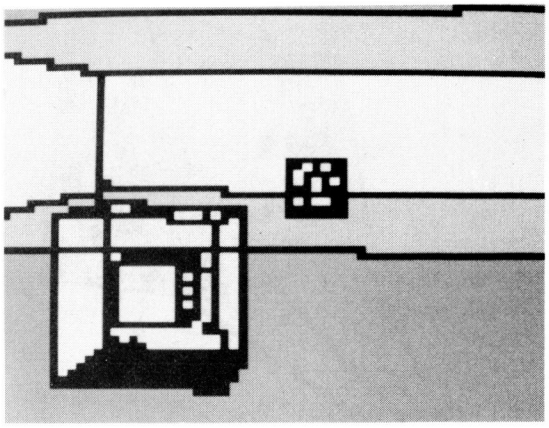


Figure 4.7 Image G: regions requiring some ray tracing are blacked out

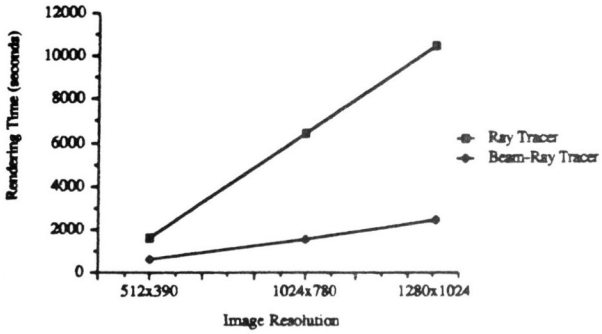


Figure 4.8 Rendering Time vs. Picture Resolution for Image G

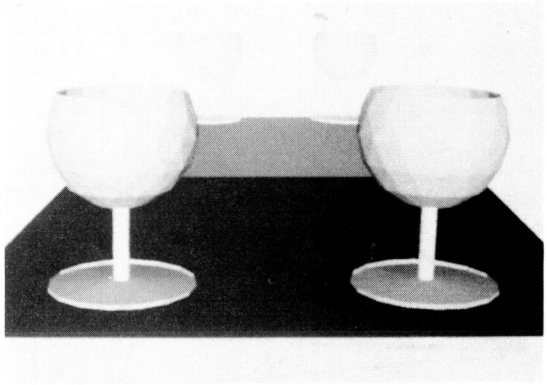


Figure 4.9 Image H: Two Goblets Reflected in a Mirror (946 faces)

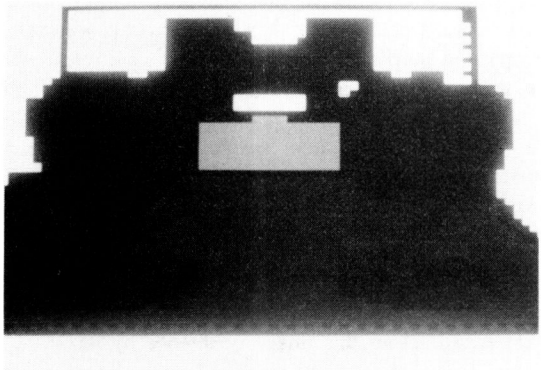


Figure 4.10 Image H: regions requiring some ray tracing are blacked out

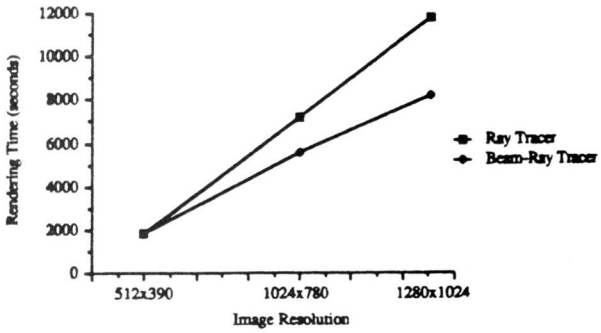


Figure 4.11 Rendering Time vs. Picture Resolution for Image H

it occasionally misses some. Note in Figure 4.8 that exploitable coherence increases with image resolution.

The image in Figure 4.9 exhibits less image coherence (see Figure 4.10); at an image resolution of 512 x 390, only 42% of the pixels were rendered by beam tracing. At this resolution, the rendering-time speedup due to the exploitation of coherence is negligible. At greater resolution, however, coherence becomes more significant as a greater number of pixels are more efficiently rendered by beam tracing (the Warnock hybrid of section 3 shows similar improvment at higher resolutions).

Figure 4.12 shows a more complex image with multiple reflections. At a resolution of 1024 x 780 it is rendered 27% faster by the hybrid algorithm than by ray tracing alone.

5. Potential Speedups for Radiosity-Based Rendering

The term "radiosity" is used to describe a variety of shading techniques that model the flow of radiant flux between diffusely reflecting surfaces. The essence of the technique for computing this radiant transfer is to decompose each face in the scene description into a mesh of small *patches* and to compute a *form factor* for each pair of patches in the scene. The form factor between patches P_i and P_j gives the fraction of flux leaving P_i that strikes P_j.

For any non-trivial scene, the calculation of form factors is obviously a very expensive operation. A form factor depends on the shapes, sizes, and angular orientations of the two patches, as well as on the distance between them. The form factor for P_i and P_j, denoted by $FF_{i,j}$, is

$$FF_{i,j}=\frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{\cos\phi_i\cdot\cos\phi_j}{\pi r^2}\delta_{dA_i dA_j}\,dA_i\,dA_j \qquad (5.1)$$

where A_i is the area of P_i, A_j is the area of P_j, dA_i is the differential area of A_i, dA_j is the differential area of A_j, and $\delta_{dA_i dA_j}$ is 1 if dA_i and dA_j are intervisible and 0 if they are not intervisible. These quantities are illustrated in Figure 5.1.

The exact evaluation of the double integral in equation 5.1 is very difficult, and an approximation to it must be used in practice. Currently the most popular method of computing this approximation is the hemicube algorithm[COHEN et al]. A hemicube (essentially a five-sided box) is centered over each patch, and each side of the hemicube is treated as a virtual "screen." All other patches in the scene are projected onto each hemicube side in turn, and hidden-patch elimination is performed using a modified Z-buffer algorithm. Based on the number and location of "pixels" in which a given patch is visible, an estimate can be made of the form-factor from that patch to the patch at the center of the hemicube.

A recently proposed alternative to the hemicube algorithm is the technique of ray casting between patches to compute form factors [CHRISTENSEN et al., WALLACE et al.]: we call this technique *flux tracing*. By using a hybrid method that is similar to the beam-ray-tracing algorithm of Section 4 the efficiency of the flux-tracing algorithm can be enhanced for scenes that exhibit substantial *intervisibility coherence*. Intervisibility coherence refers to the phenomenon that patches A and B are likely to be intervisible if A is contiguous to patch α, B is contiguous to patch β, and α and β are intervisible.

We determine form factors between patches by separately calculating intervisibility and energy transfer. In calculating intervisibility, we wish to compute $\delta_{A_i A_j}$ in Equation 5.1 for each pair of patches, P_i and P_j. To accomplish this efficiently, we consider each pair of *faces*, F_a and F_b, and, assuming they are quadrilaterals, we form from them a decahedron. This decahedron is simply the convex hull around the eight vertices of F_a and F_b, as shown in Figure 5.2. We then perform a spatial query to determine whether the decahedron is empty. If it is, every patch in face F_a is intervisible with every patch in face F_b. Should the decahedron not be empty, we subdivide the larger of the two faces -- say F_a -- into four subfaces and apply this technique recursively to each subface and F_b. Figures 5.3 - 5.6 illustrate the process.

At some point during the recursive process, it will become more efficient to perform pairwise comparisons between patches in faces F_a and F_b than to perform exhaustive queries of decahedral volumes. This task is performed by the flux-tracing algorithm, to which we alluded earlier. To determine the intervisibility of patches P_i and P_j, we cast a ray between the centers of the two patches to see if it is blocked by any face in the scene. If the ray is not blocked, the patches are treated as if they were completely intervisible; if the ray is blocked by a face, the patches are treated as if they were mutually obscured. This represents a conversion from an area-sampling process to a point-sampling process and establishes the hybrid nature of the algorithm.

Once the intervisibility of patches in the scene is determined, we compute the energy transfers between pairs of faces. If patches P_i and P_j are not intervisible, the energy transfer is clearly 0. If the patches are intervisible, we estimate the form factor, FF_{ij}^{est}, to be

$$FF_{ij}^{est}=\frac{1}{A_i}\cdot\frac{\cos\phi_i\cdot\cos\phi_j}{\pi r^2}\cdot\delta_{A_i A_j}. \qquad (5.2)$$

Figures 5.7 and 5.8 were produced using this algorithm. The scene shown in Figure 5.7 contains 29 faces that decompose into 832 patches. The scene in Figure 5.8

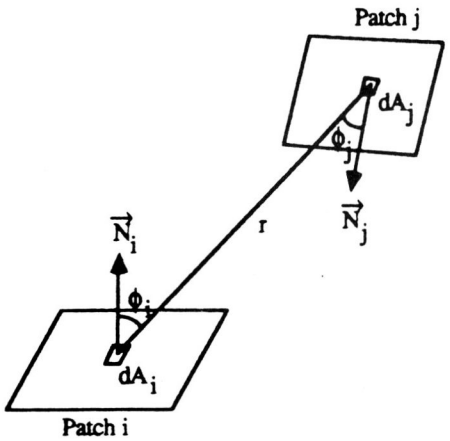


Figure 5.1 Computing Form Factors

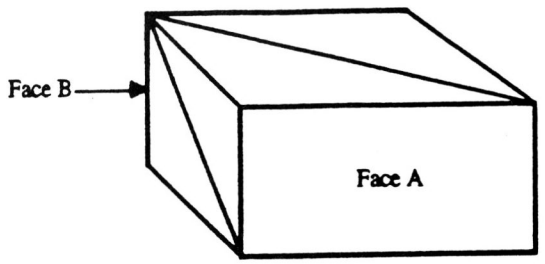


Figure 5.2 Decahedral Volume

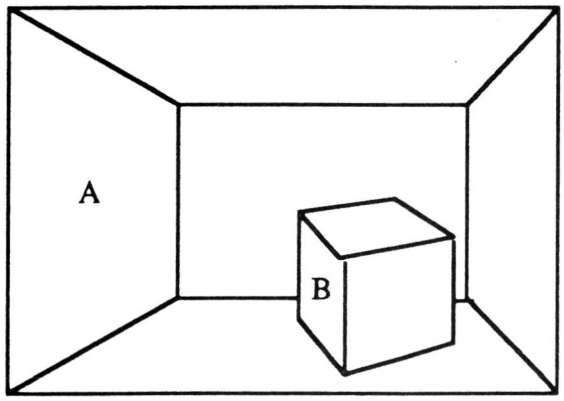


Figure 5.3 Scene with Substantial Coherence

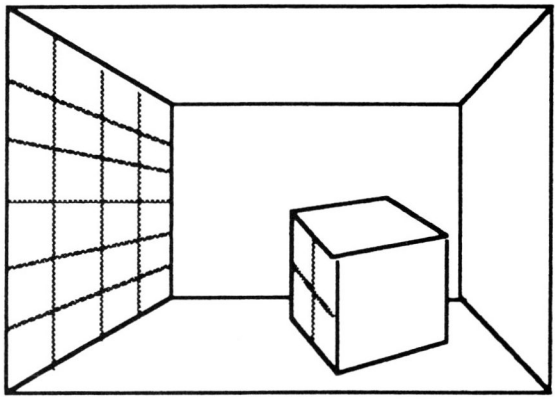


Figure 5.4 Decomposing Faces into Patches

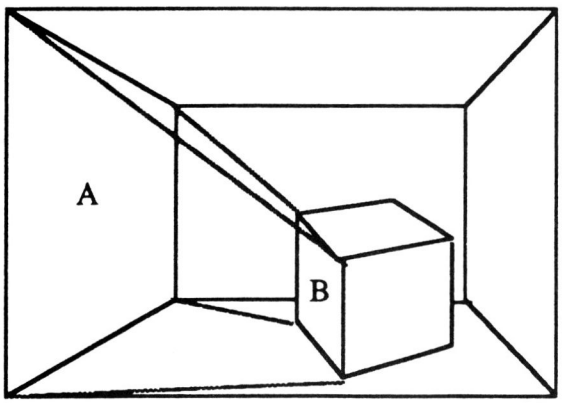


Figure 5.5 Determining the Intervisibility of Patches by Volume Sampling

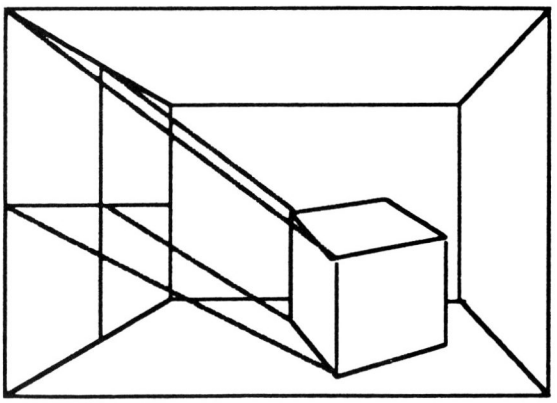


Figure 5.6 Volume Subdivision

contains 324 faces that decompose into 1,348 patches.

Our present implementation uses relatively crude heuristics to choose between the continued subdivision of a decahedral volume and flux tracing. Nevertheless, we successfully exploit intervisibility coherence to reduce by 45%, from 272,608 to 149,728, the number of ray castings needed to render the image in Figure 5.7, while performing a little more than 800 decahedral-volume queries. This reduces the time taken to compute form factors by approximately 28%, from 196 seconds to 141 seconds on a Sun 4 workstation.

As intervisibility coherence decreases, however, the computational savings of our hybrid approach will also decrease. For example, in Figure 5.8 the number of ray castings is only reduced from 510,748 to 507,676 by performing 524 decahedral-volume queries, and the time taken to compute form factors actually increases slightly by about 3%, from 777 seconds to 805 seconds on a Sun 4.

6. Conclusions

The exploitation of image coherence is widely regarded as one of the fundamental opportunities for improving the efficiency of rendering operations. Through a combination of theoretical and experimental work, we have characterized the prospects for improving practical rendering algorithms by exploiting image coherence. We have also identified the phenomenon of intervisibility coherence in radiosity-based rendering and experimented with an approach for exploiting coherence of this kind.

Our study indicates that for unusually coherent images the maximum achievable speedup with our hybrid strategy is approximately a factor of 2, and that for images of moderate complexity the speedup is on the order of a few percent. While this may be significant in some circumstances, the benefits of exploiting image coherence in this way are surprisingly less than would be expected to accrue from a process that renders many pixels simultaneously. Likewise the benefits of exploiting intervisibility coherence in radiosity-based rendering are surprisingly less than would be expected to accrue from a process that computes the intervisibility of many pairs of patches simultaneously.

The hybrid rendering algorithms presented in this paper serve primarily as experiments to improve our understanding of the phenomena of image and intervisibility coherence, and are not necessarily recommended for production rendering. Nonetheless, the algorithms presented in Sections 2 and 3 can be used to accelerate existing algorithms for coherent images (especially at high resolution) , and the algorithm in Section 5 improves upon a promising new technique for scenes with substantial intervisibility coherence.

7. References

[CHRISTENSEN et al.]
Christensen, J., Marks, J., Walsh, R. and Friedell, M. Flux Tracing: A Flexible Infrastructure for Global Shading. Technical Report TR-16-89, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, July 1989.

[COHEN et al.]
Cohen, F. C., Greenberg, D. P., Immel, D. S. and Brock, P. J. "An Efficient Radiosity Approach for Realistic Image Synthesis." *IEEE Computer Graphics and Applications* (March 1986).

[FUJIMOTO et al.]
Fujimoto, A., Tanaka, T. and Iwata, K. "ARTS: Accelerate Ray-Tracing System." *IEEE Computer Graphics and Applications* (April 1986).

[HECKBERT & HANRAHAN]
Heckbert, P. and Hanrahan, P. "Beam Tracing Polygonal Objects." *Computer Graphics*, 18, 3.

[MARKS et al.]
Marks, J., Walsh, R. and Friedell, M. Hybrid Beam-Ray Tracing. Technical Report TR-03-88, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University 1987.

[SUTHERLAND et al.]
Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. "A Characterization of Ten Hidden-Surface Algorithms." *Computing Surveys*, 6, 1.

[WALLACE et al.]
Wallace, J. R., Elmquist, K. A. and Haines, E. A. "A Ray Tracing Algorithm for Progressive Radiosity." *Computer Graphics*, 23, 3.

[WARNOCK]
Warnock, J. A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures. Technical Report TR 4-15, Computer Science Department, University University of Utah 1969.

[WEILER & ATHERTON]
Weiler, K. and Atherton, P. "Hidden Surface Removal Using Polygon Area Sorting." *Computer Graphics*, 11, 3.

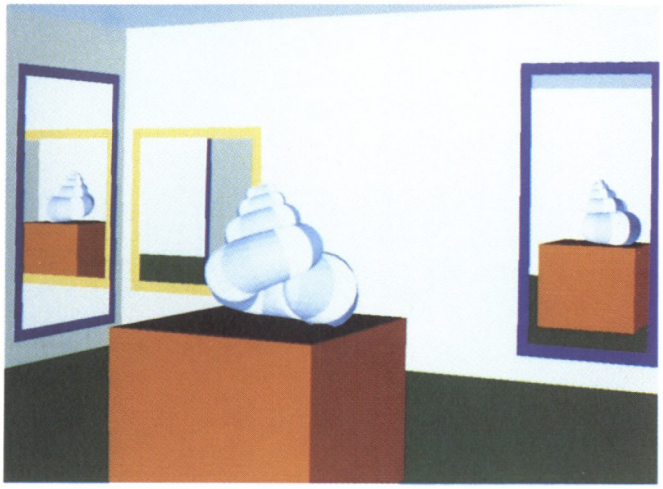


Figure 4.12 Image I: Sea Shell and Mirrors (2018 faces)

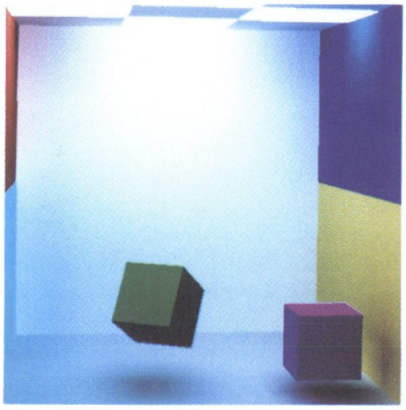


Figure 5.7 Garish Room with Four Luminous Ceiling Panels and Two Weightless Cubes

Figure 5.8 The Programmer's Cell