# Interactive Direct Lighting in Dynamic Scenes

Sebastian Fernandez Kavita Bala Moreno A. Piccolotto Donald P. Greenberg *

Program of Computer Graphics
Cornell University
January, 2000.

## Abstract

This paper presents an interactive renderer that computes direct illumination in dynamic scenes with soft shadows and complex BRDFs. The renderer permits the user to both navigate the scene interactively and modify the scene by moving objects, changing materials, and changing lighting conditions.

To support interactive viewing, we introduce a visibility caching technique in which the illumination of each patch in the scene is captured by a *local illumination environment*. This simplified environment enables interactive rendering by accelerating visibility computations. Since this is an object-space technique, local illumination environments can be reused from frame to frame.

To support interactive modification of the scene, we introduce a dynamic visibility algorithm that rapidly identifies which local illumination environments to update when the scene is modified. A 5D hierarchy stores illumination dependencies, and permits efficient identification of affected local illumination environments.

These techniques have been implemented in a parallel rendering system that uses a cluster of Intel processors connected on a fast network. The renderer produces images at interactive rates, achieving speedups of $10\times$ to $20\times$ over a standard parallelized ray tracer.

**CR Categories:** I.3.3 [Computer Graphics]: Picture/Image/Generation I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

**Keywords:** Illumination, Rendering, Parallel Computing, Ray Tracing, Rendering systems, Visibility Determination, Java.

## 1 Introduction

One long standing area of research in computer graphics is the rendering of high-quality images with soft shadows, complex BRDFs, and global illumination at interactive rates in dynamic scenes. Traditionally, rendering speeds are too slow to match these interactive needs. In this paper, we address the problem of interactive rendering of dynamic scenes with support for direct illumination effects such as soft shadows and arbitrary BRDFs because direct lighting with soft shadows yields the minimum feature set that provides acceptable cues to a user in an interactive application. Our system uses novel software techniques and exploits parallel processing to achieve this interactive rendering performance. There are several potential applications of this system, such as modeling, virtual reality walkthroughs, and gaming systems.

Our system permits the user to interactively navigate the scene and also to modify it by moving objects and changing materials. This paper makes several contributions that enable this interactive rendering performance:

- To support interactive viewing of the scene, we introduce the concept of a *local illumination environment*: the subset of the environment that contributes to the illumination at a patch in the scene. When a patch is rendered, shadow rays are traced through its local illumination environment, which is typically very simple. Thus, visibility computations for shadows rays are accelerated.

- The local illumination environments of some patches in the scene change when the scene is modified. The *ray segment tree*, a 5D hierarchical data structure, is used to track illumination dependencies and to allow rapid identification of local illumination environments affected by a scene modification.

- Both of these techniques have been implemented in a parallel rendering system, written completely in Java, and run on Intel processors connected by a fast network. We chose this environment to maintain flexibility, portability and extensibility of our code base.

## 2 Related Work

### 2.1 Direct illumination

Several researchers have considered the problem of accelerating the rendering of soft shadows [8]. Woo et al. [28] survey shadow algorithms, though this survey is somewhat dated. Several researchers have developed algorithms for computing accurate direct illumination. Teller [26] showed how to compute antiumbras and antipenumbras. Lischinski [19] introduced discontinuity meshing to produce global illumination solutions with accurate soft shadows. Haines [15] introduced shaft culling to speed up visibility computation in radiosity systems. Durand et al. [10] create a mesh of all illumination discontinuity events; this mesh can be used to compute illumination due to area light sources. Stark et al. [25] use splines to represent shadow irradiance in restricted environments. More recently, Hart et al. [16] propagate blocker information in image space to render analytically correct soft shadows.

Several researchers have recently focussed on the interactive computation of shadows: Soler et al.[24] rapidly approximate shadow computations using hardware convolution. Parker et al. [21] achieve interactive direct lighting by highly optimizing a parallelized ray tracer so that casting rays is extremely fast.

### 2.2 Dynamic scene updates

Recently, researchers have focussed on supporting interactive scene manipulation and editing with high-quality rendering. Several systems support scene editing with ray-traced imagery [7, 4, 17, 20, 22] while restricting user manipulations in different ways. The most recent and flexible of these systems [4], permits the

*This paper was submitted for review to SIGGRAPH 00.

user to modify material properties of objects, as well as move objects in the scene. However, all of these systems impose a severe restriction on the user: the viewpoint must stay fixed. Several researchers have considered the problem of dynamic editing with radiosity algorithms [6, 9, 11, 12]. Drettakis and Sillion [9] support dynamic editing by augmenting the four-dimensional link hierarchy in a hierarchical radiosity system. The drawback of these systems is that they only support pure diffuse surfaces.

Bala et al. [3] support scene editing with changing viewpoints in their ray tracing system. Their rendering system computes radiance interpolants that accelerate rendering by approximating radiance. They introduce ray segment trees to track dependencies of radiance interpolants and update them at interactive rates. However, their rendering times are not interactive.

Arvo and Kirk [2] use a five-dimensional representation of rays to accelerate ray-object intersections.

# 3 Rendering System

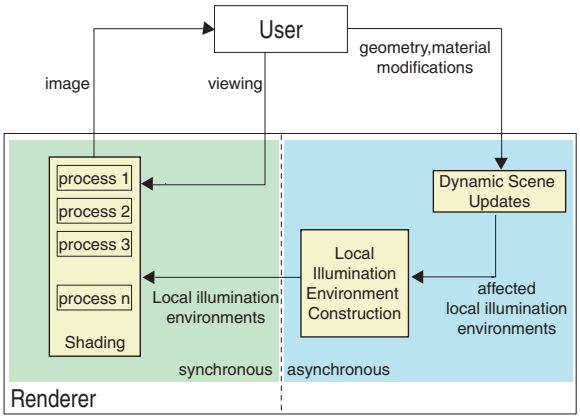


Figure 1: Rendering system.

This section describes the overall structure of the system. A user sends navigation commands and geometry/material/lighting changes to the rendering system, which interactively produces images that are displayed to the user.

The system is split into three modules. The first module is the dynamic visibility updater. It takes in bounding box descriptions of scene modifications and produces a set of patches that must have their local illumination environments updated. This set of patches is provided to the second module, the local illumination environment constructor.

The local illumination environment constructor uses this set of patches to recompute the necessary local illumination environments. This module is also continuously computing local illumination environments as a user navigates the scene. These local illumination environments are passed to the shaders.

The shaders consist of several parallel renderers that use local illumination environments to render the pixels assigned to them. The renderers are not synchronized with the local illumination environment constructor, so the shaders do not have to wait on the results of local illumination environment construction. Also, computing any one pixel does not depend on the computation of any other pixel, so there is no communication between the renderers. The computed pixels are sent over a high-speed network where they are assembled into a single image.

The rest of this paper is organized as follows: Section 4 describes how local illumination environments are constructed. Section 5 describes how dynamic scene modifications are supported. Section 6 presents results, and finally, we conclude with a discussion of future work in Section 7.

# 4 Local Illumination Environments

The *local illumination environment* associated with a patch is the subset of the original scene, not necessarily determined by geometric proximity, that can significantly influence the illumination at the patch. Therefore, illumination at a patch computed using the local illumination environment does not differ significantly from that computed by using the entire scene. Rendering using the local illumination environment is substantially accelerated because the local illumination environment is often much simpler than the entire scene. As shown in Section 6, using local illumination environments allows significant speedup over renderers that use traditional acceleration structures. One important feature of local illumination environments is that they are *view-independent*; since a local illumination environment does not change with the viewpoint it can be reused from frame to frame.

For a direct lighting shading model, the local illumination environment for a patch consists of a set of light sources (emitters) that are visible to the patch, and for each visible emitter, a list of objects that may partially occlude the emitter. This list of objects is called a *blocker list*. Thus, each local illumination environment consists of a set of emitters, and their associated blocker lists (if any). Notice that a blocker list is associated with a patch-emitter pair.

Section 4.1 describes how local illumination environments are constructed. Section 4.2 describes how local illumination environments are used when rendering images. Section 4.3 describes several important design trade-offs in the design of the local illumination environment constructor module.

## 4.1 Constructing local illumination environments

For optimal performance, local illumination environments should be simple. Ideally, the local illumination environment for each patch should include only emitters that affect illumination at the patch. Similarly, for each visible emitter in a local illumination environment, the associated blocker list should only include the blockers that actually occlude the emitter. Computing this minimal blocker list is potentially expensive. Therefore, we sample the visibility between a patch and emitter to determine blocker lists, as done in [16].

The construction of the local illumination environments for all patches in the scene can be considered to be the successive computation of the blocker lists for each patch-emitter pair in the scene. Local illumination environments are constructed lazily. Section 4.3.3 describes how patch-emitter pairs are selected for blocker list construction as the user navigates the scene.

Assuming a patch-emitter pair is selected, its blocker list is determined by casting rays from the patch to the emitter to compute visibility, and adding occluding objects (if any) to the blocker list of the patch-emitter pair. There are three cases that arise when computing visibility between a patch-emitter pair:

- Fully occluded. If all rays from the patch to the emitter are occluded, the emitter is excluded from the local illumination environment for the patch. When shading this patch, no visibility computations are performed for that emitter.

- Fully visible. If all rays from the patch to the emitter are visible, the emitter is included with an empty blocker list

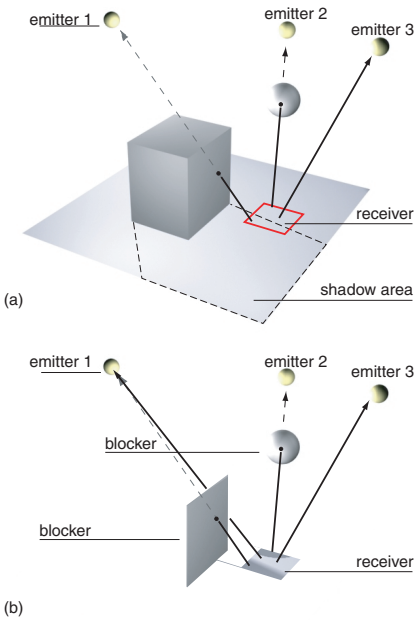


Figure 2: Constructing blocker lists. (a) Constructing the local illumination environment for the three patch-emitter pairs. (b) The local illumination environment for the patch.

in the local illumination environment of the patch. Again, when shading this patch, no visibility computations are performed for that emitter; only shading is computed.

- Partially occluded. If some of the rays from the patch to the emitter are occluded, the corresponding occluders (or blockers) are added to the blocker list for that patch-emitter pair. When shading the patch, visibility is computed only with respect to this blocker list.

Figure 2 depicts how blocker lists are computed for a patch in the scene. Figure 2-(a) shows some rays cast from the patch to each emitter; emitters 1 and 2 are partially blocked, while emitter 3 is fully visible to the patch. The blocker lists for the corresponding patch-emitter pairs are shown in Figure 2-(b). The blocker list for emitter 1 includes the vertical face of the cube shown. The blocker list for emitter 2 includes the sphere. Note that the algorithm does not restrict blockers to be polygonal patches; arbitrary ray tracing primitives are permitted. The blocker list for emitter 3 is empty, and emitter 3 is marked as fully visible in the local illumination environment for the patch.

#### 4.1.1 Patch subdivision

Rendering a fully occluded or fully visible patch-emitter pair is inexpensive since no visibility computations are necessary. Rendering a partially occluded patch-emitter pair requires computation proportional to the size of its blocker list because visibility computations are done with respect to each blocker in the list. Large input patches are more likely to have large blocker lists, e.g., the floor or ceiling in an indoor environment. Therefore, to improve rendering performance, patches whose local illumination environment include a large blocker list are automatically subdivided. Patches are defined with respect to the two-dimensional coordinates of a surface. Thus, if there exists an uniform mapping from the unit square to the surface of an object, patches can be defined over that surface. Patch subdivision is done through a quad-tree subdivision of the unit square representing the object surface. This quad-tree of patches is called a *patch tree*. There is one patch tree per surface in the environment.

A patch tree is subdivided if two conditions hold:

- the blocker lists for the patch tree are too complex (*i.e.*, they contain more than a specified number of blockers); and
- the surface area represented by that patch tree is above a certain threshold.

#### 4.1.2 Shaft sampling

Since visibility is computed by sampling, it is possible that a blocker is not included in a blocker list due to insufficient sampling. Thus, the accuracy of the blocker lists depends on the number of sample rays used to compute the lists. As the number of samples increases, the probability of missing a contributing blocker or emitter decreases. Section 4.3.2 describes how additional samples are automatically acquired so that blocker lists converge over time.

An alternative technique to determine blocker lists is shaft culling [14]. Shaft culling produces conservative blocker lists by finding all potential blockers that lie in a shaft from the patch to the emitter. However, shaft culling is too conservative; for example, in Figure 2, shaft culling would probably include all the faces of the cube in the blocker list for emitter 1. Unlike shaft culling, our technique only samples the shaft; while this approach may miss some objects from the shaft, every object included in the blocker list is guaranteed to be contained within the shaft. Increasing the number of samples decreases the probability of missing a potential blocker. Thus, over time, this method will converge to the right solution while always including the smallest number of blockers possible given the information collected.

### 4.2 Shading with local illumination environments

Once patch-emitter blocker lists are constructed, the renderer computes direct illumination by using the local illumination environment for each visible patch in the image. A pixel is shaded by first tracing a ray through the scene to find the point of intersection. Once the point of intersection is determined, shadow rays are cast to each of the light sources. However, where a standard ray tracer would trace shadow rays to all light sources through the entire scene, our algorithm casts rays only to lights in the local illumination environment for the intersected patch and checks for intersection only with objects in its blocker lists.

Radiance is computed along the shadow rays that are not blocked, and multiplied by the BRDF at the surface of the patch. It is important to note that for specular BRDFs, a direct lighting local illumination environment is insufficient to produce a compelling image. Thus, if the scene includes an object with a specular BRDF, rays that intersect that object are traced in the same way that a standard ray tracer would.

Patches are not stored in the same acceleration structure used by higher level objects to determine eye ray visibility. Instead, once the surface coordinates of the intersection point are determined, the associated patch tree is traversed to determine the local illumination environment to use for that point. This is a more appropriate search structure for patches than a regular grid since they are defined over a surface. Traversing this structure adds an additional cost that is not incurred by a standard ray tracer; but, we have found that this cost is negligible.

This algorithm achieves significant speedups by eliminating visibility computations for fully visible and fully occluded patch-emitter pairs. As shown in Figure 10, a significant fraction of the of the patch-emitter pairs in the scene fall into one of these two categories.

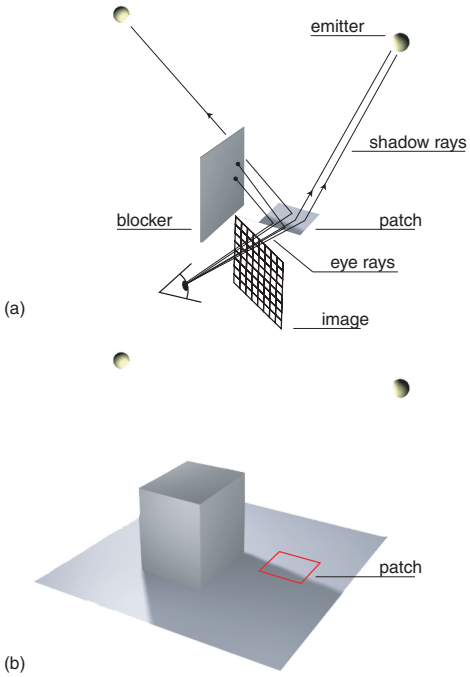


Figure 3: Shaders. (a) Renderer uses blocker lists when shading pixels. (b) Output image with direct illumination on all patches, including shadows.

Figure 3 depicts how blocker lists are used when rendering an image. Figure 3-(a) shows some eye rays traced through the image plane along with shadow rays traced by the renderer to compute illumination at each visible patch. Our algorithm accelerates the computation of shadow ray visibility. Eye ray visibility is computed as in a standard ray tracer. Figure 3-(b) shows the final image.

## 4.3 Design Issues

This section describes several design trade-offs that are addressed by the system.

### 4.3.1 Asynchronous

The local illumination environment constructor is an asynchronous process that computes local illumination environments as the user navigates the scene. Local illumination environments are initially computed for large patches and are refined over time as the system is used. The advantage of on-line computation of local illumination environments is that they are only computed for the portions of the scene that are currently visible to the user. It is also possible to dynamically garbage collect local illumination environments for surfaces that the user is no longer looking at. This method can lead to some artifacts as the user observes new portions of the scene that do not have their local illumination environments fully computed as yet. These artifacts typically clear up quickly.

### 4.3.2 Blocker list quality

Computing blocker lists is a stochastic process and as such, it is possible that some emitter or some blocker may be missed. These errors show up either as dark patches or missing shadows. Since we normally operate in an on-line mode and the blocker lists are continually refined, such errors go away in time. However, incomplete blocker lists are objectionable to the user.

To minimize these artifacts, we use the concept of blocker list quality. Blocker list quality is a measure of how confident we are that the contents of the blocker list are complete. Currently, we use a ratio of samples computed to patch area to approximate this measure. A blocker list is not used by the renderers until the blocker list quality is above a certain threshold. Instead we continue to use the blocker lists of the patch's parent (a pre-computation could be performed to make sure that all root patches have a high quality blocker list) until the blocker list quality is high enough.

### 4.3.3 Picking patch-emitter pairs

The local illumination environment constructor selects a patch by randomly choosing a receiver patch in proportion to the number of pixels the patch occupies on the screen. This is possible because blocker lists are constructed concurrently while rendering images. Thus, a random pixel on the screen is selected, and the closest visible patch from that pixel is picked for blocker list construction.

Once a patch is selected, there are two possible methods to select an emitter. The first method is to randomly pick one of the emitters in the scene. This method does not work well in environments with many emitters since most emitters are fully occluded with respect to a single patch. Therefore, a second method is considered that is similar in spirit to that proposed by [23]. A uniform subdivision grid over the entire scene is maintained with a list of emitters in each cell of the grid. Each cell in the grid represents the set of emitters possibly visible from some point within the cell.

Initially, emitters are picked at random, as in the first method, but these emitters are stored in the grid as they are found. As the grid becomes populated, the algorithm switches to randomly picking emitters from the appropriate cell, instead of from the global list of lights. The cell is selected according to the point on the patch from which a ray is cast. The benefit of using this grid is that it directs computation of blocker lists to emitters that are more likely to contribute to the illumination from the receiver patch.

# 5 Dynamic Visibility

As described in Section 4, local illumination environments are view-independent; *i.e.*, when the viewpoint changes, blocker lists can be reused from frame to frame. However, when objects in the scene move, some of the blocker lists become incorrect. For example, Figure 4 depicts three patches in a scene, along with some of the rays traced by the local illumination environment constructor to determine blocker lists for each patch. When an object with bounding box **b** is added to the scene, the blocker lists for patches $P_1$ and $P_2$ are no longer correct, while the blocker list for patch $P_0$ is unaffected by the addition of the object. If the blocker lists for patches $P_1$ and $P_2$ are not updated, the rendered image will incorrectly miss the shadow cast on the patches by the new object.

This section presents an algorithm to efficiently identify all blocker lists that might be affected by a change to the scene.

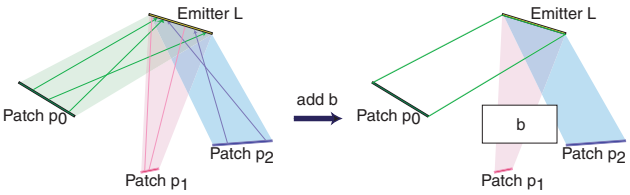


Figure 4: Blocker lists for patches $P_1$ and $P_2$ are no longer correct when **b** is added to the scene.

When the scene is changed, the algorithm has to satisfy two goals:

- **Correctness**. For correctness, all the blocker lists affected by the scene change must be identified;
- **Efficiency**. For efficiency, only the blocker lists affected by the change to the scene must be updated.

The blocker list for each patch-emitter pair in the scene depends on some region of space; Figure 4 shows these regions as shaded areas. To correctly update blocker lists, the algorithm must conservatively characterize these regions. Section 5.1 describes a dual space [3], called *ray segment space*, in which each dependency region of a blocker list can be represented conveniently. Section 5.2 describes a data structure, called a *ray segment tree*, that stores the dependency regions. Section 5.3 describes how this data structure is used to rapidly identify the blocker lists affected by a change in the scene.

## 5.1 Ray segment space

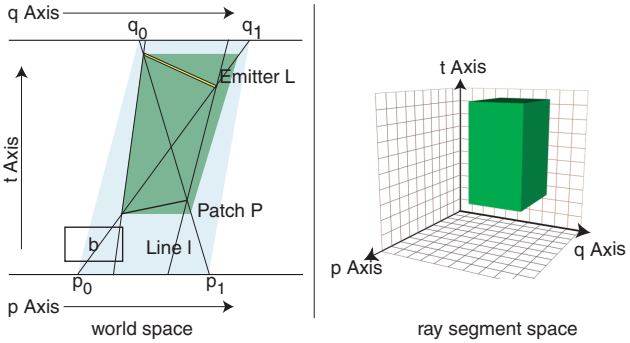


Figure 5: Ray segment space. The dependency region of the patch $P$ on emitter $L$ is represented by the green box in $(p, q, t)$-space.

A dual space is used to represent the regions of world space that can affect a blocker list. For simplicity, we first consider a 2D scene in which we must represent all *lines* through patch $P$ and emitter $L$. Each line *l* in 2D world space from $P$ to $L$ can be parameterized by the two intercepts $(p, q)$ it makes with a pair of parallel axes surrounding the scene, as shown in Figure 5. This parameterization is the 2D equivalent of the two-plane parameterization described in [13, 27] [1]. The four extremal lines shown in Figure 5, from each end of patch $P$ to each end of emitter $L$, conservatively bound all lines from $P$ to $L$. Thus, all lines from $P$ to $L$ can be conservatively represented by a range $(p_0, q_0)$—$(p_1, q_1)$, as shown in the light blue region of Figure 5. The problem with this conservative representation is that it also includes regions of space that do not affect the blocker lists. For example, in the figure, when an object with bounding box **b** is added to the scene, the blocker list for the patch-emitter pair $P$-$L$ is marked as potentially affected, even though **b** does not block any of the rays from $P$ to $L$.

For efficiency, a more accurate representation of the dependency regions is required. This accuracy is achieved by adding an additional dimension *t* that represents the distance along a ray (as shown in the figure). This dual three-dimensional $(p, q, t)$-space, called *ray segment space*, has the following property: a box in this space represents a shaft of bounded rays (or ray segments) in world space, connecting two parallel line segments. Together, all the rays from the patch to the emitter can be conservatively represented by a box $(p_0, q_0, t_0)$—$(p_1, q_1, t_1)$, shown on the right in Figure 5. The dependency region represented by this box in world space is shown on the left in green.

Extending this discussion to 3D scenes, each line in 3D can be represented by the four intercepts $(p, q, r, s)$ it makes with two parallel rectangles surrounding the scene [2]. With the additional $t$ dimension, the ray segment space is a five-dimensional space. Thus, each blocker list depends on a region of world space that can be represented by a five-dimensional box in ray segment space.

## 5.2 Ray segment trees

The blocker list dependency regions are stored in a data structure, called the *ray segment tree* [3], that is used to rapidly identify the blocker lists affected by a change in the scene. When the local illumination environment constructor determines a blocker list for a particular patch-emitter pair, the dependency region of that blocker list in ray segment space is computed as described in the previous section. These dependencies, represented as boxes in ray segment space, are then stored in a *ray segment tree* that is constructed over ray segment space. The root of the ray segment tree represents all ray segments that pass between two parallel rectangles surrounding the scene. Since there are six such pairs of parallel rectangles, six ray segment trees are built: one for each pair of parallel rectangles surrounding the scene.

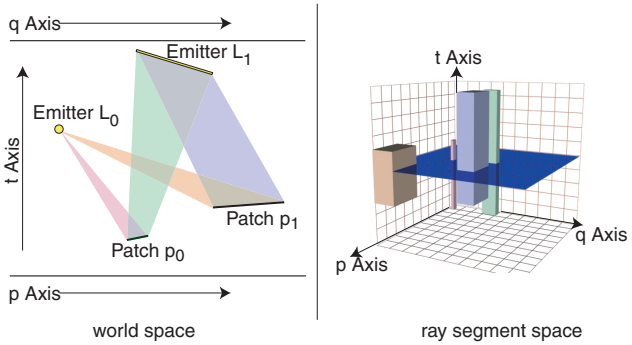


Figure 6: Ray segment trees. Each of the dependency regions in world space is represented by a box in ray segment space. Ray segment trees are constructed over this space.

[1] In this parameterization, four such pairs of axes are required to represent all lines that pass through a scene. However, other parameterizations of lines are also possible as described in [5, 18].

[2] Six such pairs of parallel rectangles are required to represent all the rays that intersect a scene.

Each node in the ray segment tree stores ten coordinates $(p_0, q_0, r_0, s_0, t_0)$—$(p_1, q_1, r_1, s_1, t_1)$ that define a 5D bounding box in ray segment space. The root node of the tree corresponds to the 5D box $(0, 0, 0, 0, 0)$—$(1, 1, 1, 1, 1)$, and represents all ray segments that pass through the scene from the front face to the back face of the ray segment tree. When a ray segment tree node is subdivided along each of the five axes, thirty-two children are created. The ray segments represented collectively by the children include all rays segments represented by the parent. When a ray segment tree node is subdivided, each of the dependency regions stored in the parent node are copied down to the appropriate children; the dependency regions are recomputed with respect to the new children, to represent the part of the dependency region that lies in the appropriate child.

Figure 6 shows four patch-emitter pairs in 2D world space and their corresponding dependency regions, shown as shaded polygons on the left. On the right, the corresponding boxes that represent each of those dependency regions in the root node of the corresponding ray segment tree are shown, appropriately shaded. The blue plane shown in ray segment space is one of the three cutting planes that subdivide the ray segment tree node shown.

### 5.3 Identifying affected blocker lists

When the scene changes, for example, by the movement of objects, addition of new objects, or deletion of existing objects, the ray segment trees are traversed to find the blocker lists that are affected. Note that our system treats object movements as object deletions from the old location and object additions to the new location; the cost of performing these operations is small enough that it is not worth optimizing further.

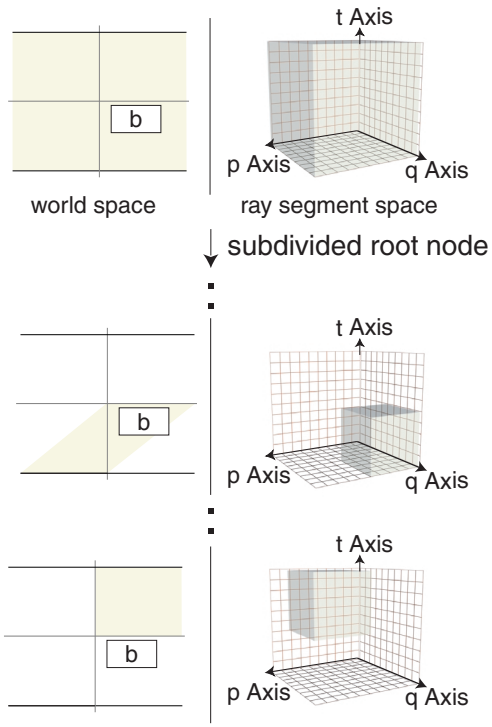


Figure 7: Traversing ray segment trees. When box $b$ is added to the scene, the ray segment trees are rapidly traversed to find the affected blocker lists.

Figure 7 depicts the action of the algorithm when a 2D object with bounding box $b$ is edited. Starting at the root node of the ray segment tree, $b$ is tested against the shaft that represents the node. If there is an intersection, $b$ is tested against each of the children of the ray segment tree node that intersects $b$. The algorithm walks down the ray segment tree, recursively testing against the appropriate ray segment tree nodes until it reaches the leaves. Then, the algorithm tests against each blocker list stored in the leaf, to find the patch-emitter pairs that are affected by the edit.

## 6 Results

### 6.1 Implementation

The system currently runs on a cluster of eight shared-memory quad-processor Pentium-III 550 MHz systems. The computers are connected together by a Myrinet gigabit network, although the renderer uses only about 40 Mbps of network bandwidth.

Seven machines are used as shaders, and one machine is used for the local illumination environment constructor. The model and the blocker lists are duplicated on all eight machines, but within each machine, all four processors share the model and blocker lists. One dual processor Pentium-III 550 MHz system is used for the display of images and for interpreting user input.

All of the code is written in Java 2 and is run using Sun's JRE 1.2.2. The code has not yet been optimized and we have consistently chosen generality, flexibility, and portability over performance. As a result, it is relatively easy to include new types of primitives and materials with different BRDFs. Communication is done over standard Java TCP/IP sockets.

The ray tracer is a stochastic ray tracer augmented to support local illumination environments and dynamic visibility. Area sampling of the light sources is not purely random: the same sample points on the emitters are chosen every time, introducing banding, but removing pixel noise.

### 6.2 Scene Descriptions

Three models are used to evaluate the performance of these algorithms: Model 1, shown in Figure 8-(a), has approximately 9k patches, one area light source, and purely diffuse materials. Model 2, shown in Figure 8-(d), is similar to Model 1, except that it includes a glossy floor and a mirror-like reflective table-top. Model 3, shown in Figure 9, has approximately 20k patches and six area light sources.

Rendering results presented in this section report performance for a 512x512 image. The synchronous component of the renderer uses 28 processors, while the local illumination environment constructor uses 4 processors. The dynamic visibility module has not been parallelized; therefore, all timing results for dynamic visibility are on a single processor. All results are presented for one representative image in the scene, shown in Figure 8-(c).

### 6.3 Material properties

Figure 8 shows Model 1 evaluated with some of the different BRDFs supported by our shaders. Figure 8-(a) includes purely diffuse surfaces. Figure 8-(b) includes a mirror-like reflective table-top and a diffuse floor. Figure 8-(c) includes a glossy table. Finally, Figure 8-(d) includes a mirror-like reflective table-top and a glossy textured floor. Notice the high quality of the soft shadows cast by the chairs on the floor, and the reflections of objects in the table-top.

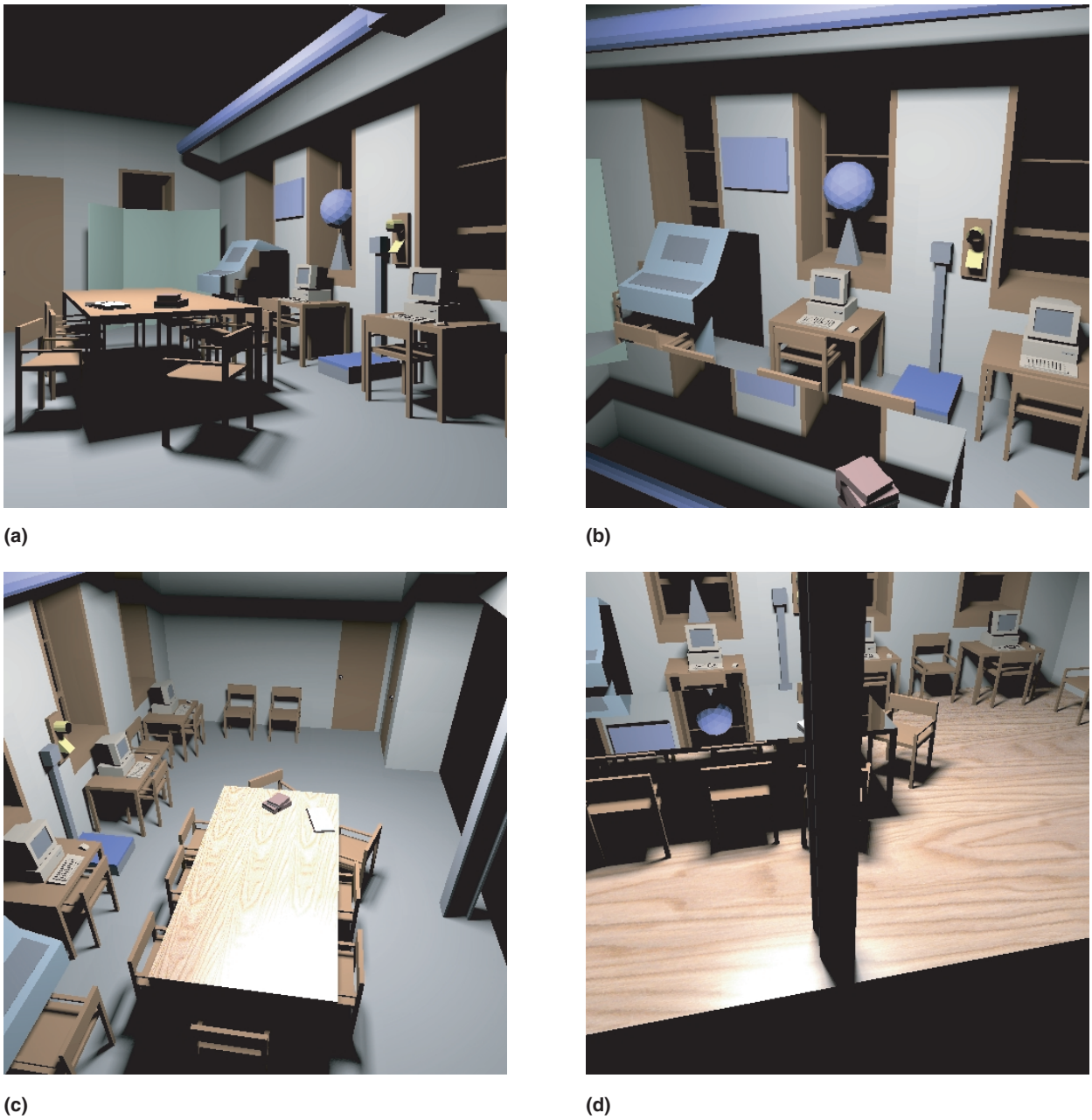


Figure 8: Museum scene with different BRDFs: (a) diffuse materials, (b) reflective mirror-like table-top, (c) glossy table-top, (d) reflective table-top, glossy textured floor.

Figure 9: Model 3 with six area light sources.

### 6.4 Lighting and scene complexity

Our techniques increase in effectiveness with the complexity of occlusions in a scene. Rendering performance is also improved when high-quality images with soft shadows are required. Figures 8 and 9 show images of Model 1, Model 2 and Model 3 produced by our renderer. The images have smooth soft shadows; each image was produced by casting 100 shadow rays per emitter.

| Shadow rays | 0 | 1 | 20 | 100 |
|---|---|---|---|---|
| Our system (sec/frame) | 0.18 | 0.24 | 0.36 | 0.67 |
| Standard rt (sec/frame) | 0.18 | 0.39 | 2.76 | 12.78 |
| **Speedup** | 1 | 1.63 | 7.67 | 19.07 |

Table 1: Performance and speedup with respect to shadow rays. As the number of shadow rays increases, our system is up to $20\times$ faster than the standard ray tracer.

Table 1 presents timing results for Model 1 as the lighting complexity of the scene increases. As larger numbers of shadow rays are traced for higher quality soft shadows, our system performs increasingly better than a standard ray tracer. When no shadow rays are traced, both ray tracers only compute visibility along each eye ray. Since our algorithm does not accelerate this computation, there is no speedup for these images. However, when more shadow rays are cast, our algorithm is up to $20\times$ faster than the standard ray tracer. Even greater speedups are obtained when more than 100 shadow rays are cast, but this would not substantially improve image quality. Similar frame rates, 1 to 2 frames per second, are obtained for the more complex scene in Model 3.

### 6.5 Patch subdivision

Figure 10 shows the effect of increasing patch subdivision while constructing blocker lists. As patches are refined, their blocker lists typically decrease in size, thus improving rendering performance. In the figure a color-coded image of Model 1 is shown, where the colors represents visibility with respect to the single emitter in the scene. Green represents full visibility, blue represents full occlusion, and shades of gray represent increasing numbers of blockers in a partially visible patch. Black represents only one blocker and white represents ten or more blockers. As the minimum patch size decreases from Figure 10-(a) to Figure 10-(d), the number of gray patches decreases, while the number of green and blue patches increases. For the gray patches that remain, the length of the blocker lists decreases, as can be seen by the darkening of the white patches.

| Minimum patch size | 1 % | 0.1% | 0.01% | 0.001% |
|---|---|---|---|---|
| Number of patches | 4.5k | 5.5k | 9k | 19k |
| Our system (sec/frame) | 2 | 0.94 | 0.67 | 0.64 |
| Standard rt (sec/frame) | 12.78 | 12.78 | 12.78 | 12.78 |
| **Speedup** | 6.4 | 13.6 | 19 | 20 |

Table 2: Performance and speedup with respect to minimum patch size. The minimum patch size is specified as a percentage of the largest polygon in the scene. As the minimum patch size decreases, the number of patches increase, and performance with respect to the standard ray tracer improves.

Table 2 presents system performance as the minimum patch size decreases. From left to right, patch sizes are decreased by factors of ten resulting in the number of patches shown in the table. The patch sizes are specified as a percentage of the largest polygon in the scene. As the number of patches increases, our technique produces increasingly greater speedups, from $6\times$ to $20\times$, while improving frame rates. Since decreasing the minimum patch size to 0.001% does not substantially accelerate the ray tracer, we select a minimum patch size of 0.01% for Model 1. This prevents unnecessary subdivision of patches. This minimum patch size is currently picked manually, but a simple cost model should be able to predict an appropriate value.

Since most local illumination environments are simple, with only a few blockers (if any), the memory required to store local illumination environments is very small: less than 5 MB for these scenes.

### 6.6 Dynamic Visibility

Figure 11 shows rendered images produced by our system as the user moves objects in the scene. Notice the correct shadows cast by the cube on the floor in Figure 11-(a), and the reflection in Figure 11-(b). As the box is moved in Figures 11-(c) and (d), the shadows cast by the left wall on the box and the shadows cast by the box on the floor and the right wall are correctly updated.

| Minimum patch size | 1 % | 0.1% | 0.01% |
|---|---|---|---|
| **Time in seconds** | 0.1-0.9 | 0.12-1.0 | 0.3-1.5 |
| **Memory in MBs** | 4.5 | 6 | 20 |

Table 3: Performance of the dynamic visibility module with decreasing patch size. The dynamic visibility module responds at interactive rates. Note that these timing results were obtained on a single processor.

Table 3 presents timing and memory results for Model 1 as the cube is moved by the user through the scene. As the number of patches increases, the time to update affected blocker lists increases from 0.1 up to 1.5 seconds. Memory requirements also

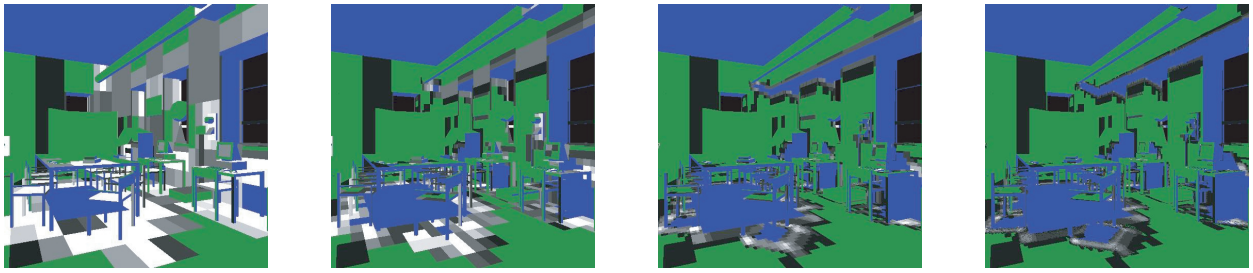


Figure 10: Color-coded images show complexity of local illumination environments. Green represents full visibility, blue represents full occlusion, and shades of gray represent increasing numbers of blockers in a partially visible patch.

increase to store more dependencies. However, even for the patch size that results in optimal rendering performance (0.01%), the dynamic visibility module finds all affected blocker lists at interactive rates. Also, these timing results were obtained on a single processor. Therefore, straightforward parallelization of this computation should enable real-time updates.

The memory usage of the algorithm is modest; less than 20 MB. Similar results are obtained for the movement of the box in Figures 11-(c) and (d).

## 7 Conclusions

This paper presents an interactive renderer that computes direct illumination in dynamic scenes with soft shadows and supports complex BRDFs. A user can both navigate and modify the scene interactively. To achieve this rendering performance, this paper introduces the concept of the local illumination environment, which caches visibility information and is reused from frame to frame. The local illumination model for each patch is the scene is computed asynchronously and is used by the shaders to accelerate the tracing of shadow rays. Automatic patch subdivision accelerates rendering performance by simplifying local illumination environments.

The ray segment tree, a five-dimensional hierarchical data structure, is used to track dependencies of the local illumination environments. When the scene is modified, these trees are rapidly traversed to identify local illumination environments that are affected by the modification and need to be computed.

We have presented results for a parallel renderer written entirely in Java. Our results show that the system achieves interactive performance, with speedups from $10\times$ to $20\times$ in scenes containing soft shadows. The memory usage of the algorithms described is modest.

There are several avenues of research that can be explored in this system. This implementation only used direct lighting local illumination environments. We would like to extend this concept to indirect lighting by capturing indirect sources of illumination in the local illumination environment. The approach of Drettakis and Sillion [9] can be adapted to this purpose to support dynamic visibility. Accelerating the eye ray visibility determination would further improve rendering frame rates. A variety of techniques seem applicable [1].

## References

[1] Stephen J. Adelson and Larry F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3):43–52, May 1995.

[2] James Arvo and David Kirk. Fast ray tracing by ray classification. In *Computer Graphics (SIGGRAPH 1987 Proceedings)*, pages 196–205, July 1987.

[3] Kavita Bala, Julie Dorsey, and Seth Teller. Interactive ray-traced scene editing using ray segment trees. In *Tenth Eurographics Workshop on Rendering*, pages 39–52, June 1999.

[4] Normand Brière and Pierre Poulin. Hierarchical view-dependent structures for interactive scene manipulation. In *Computer Graphics (SIGGRAPH 1996 Proceedings)*, pages 83–90, August 1996.

[5] Emilio Camahort, Apostolos Lerios, and Donald Fussell. Uniformly sampled light fields. In *Ninth Eurographics Workshop on Rendering*, June 1998.

[6] Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Computer Graphics (ACM SIGGRAPH '90 Proceedings)*, 24(4):135–144, August 1990.

[7] Robert L. Cook. Shade trees. In *Computer Graphics (SIGGRAPH 1984 Proceedings)*, volume 18, pages 223–231, July 1984.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 137–45, July 1984.

[9] George Drettakis and Francois X. Sillion. Interactive update of global illumination using a line-space hierarchy. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 57–64, August 1997.

[10] Fredo Durand, George Drettakis, and Calude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

[11] David Forsyth, Chien Yang, and Kim Teo. Efficient radiosity in dynamic environments. In *Proceedings 5th Eurographics Workshop on Rendering*, June 1994.

[12] David W. George, Francois X. Sillion, and Donald P. Greenberg. Radiosity Redistribution for Dynamic Environments. *IEEE Computer Graphics and Applications*, 10(4):26–34, July 1990.

[13] Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[14] E. Haines and J. Wallace. Shaft culling for efficient ray-traced radiosity. In *Proc. $2^{nd}$ Eurographics Workshop on Rendering*, May 1991.

[15] Eric A. Haines and John R. Wallace. Shaft culling for efficient ray-traced radiosity. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, New York, 1994. Springer-Verlag.

[16] David Hart, Philip Dutre', and Donald P. Greenberg. Direct illumination with lazy visibility evaluation. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, Annual Conference Series, pages 147–154, August 1999.

[17] David Jevans. Object space temporal coherence for ray tracing. In *Proceedings of Graphics Interface 1992*, pages 176–183, Toronto, Ontario, May 1992. Canadian Information Processing Society.

[18] Mark Levoy and Pat Hanrahan. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

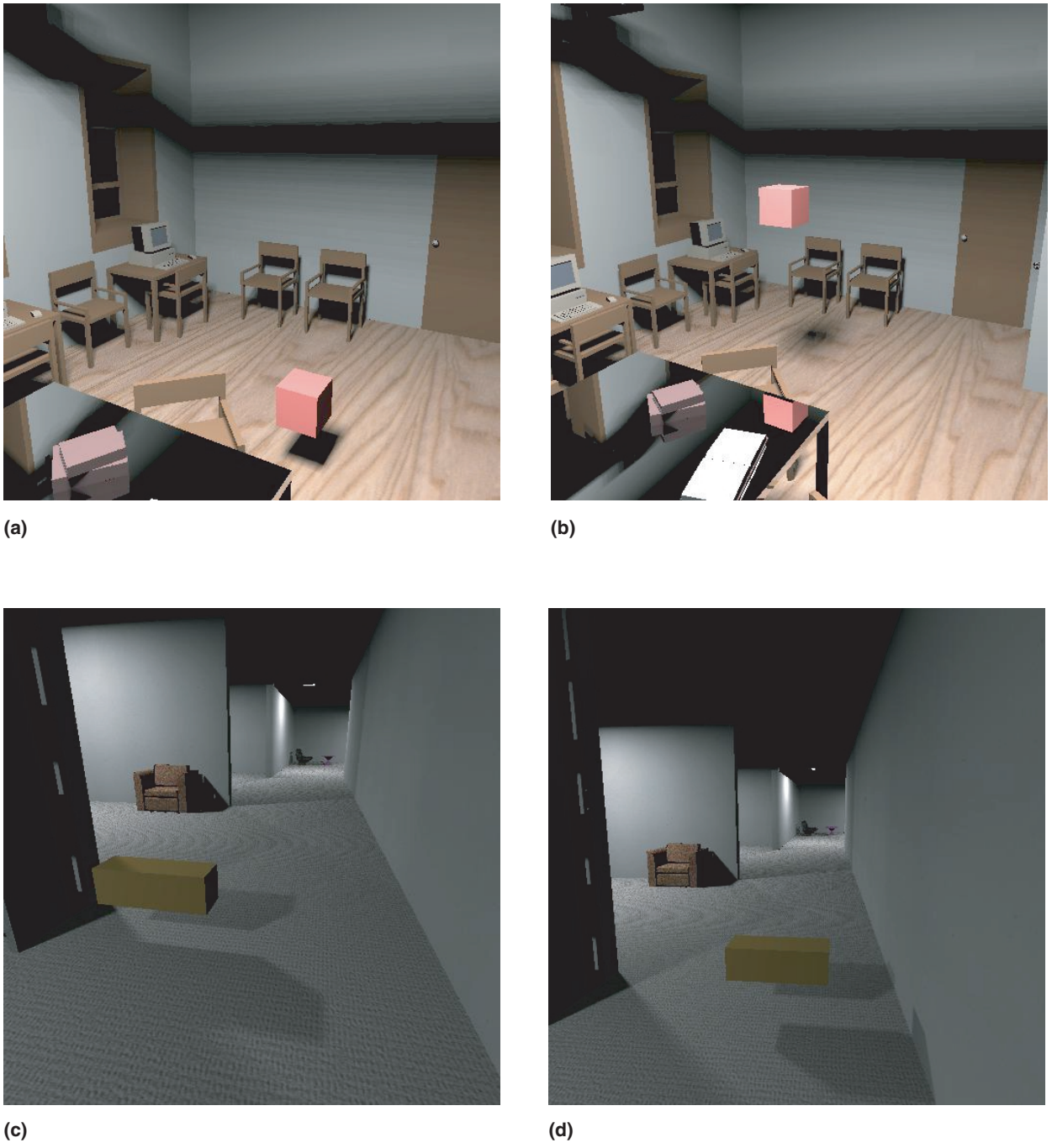


Figure 11: Effect of dynamic object movement: (a)-(b) red cube is moved in Model 1, (c)-(d) brown box is moved in Model 3. Notice the shadows cast by the moving cubes on the environment.

[19] Daniel Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity Meshing for Accurate Radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, November 1992.

[20] Koichi Murakami and Katsuhiko Hirota. Incremental ray tracing. In K. Bouatouch and C. Bouville, editors, *Photorealism in Computer Graphics*. Springer-Verlag, 1992.

[21] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Chuck Hansen. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, April 1999.

[22] Carlo H. Séquin and Eliot K. Smyrl. Parameterized ray tracing. In *Computer Graphics (SIGGRAPH 1989 Proceedings)*, pages 307–314, July 1989.

[23] Peter Shirley and Changyaw Wang. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1), January 1996.

[24] Cyril Soler and Francois X. Sillion. Fast calculation of soft shadow textures using convolution. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 321–332, August 1998.

[25] Michael M. Stark, Elaine Cohen, Tom Lyche, and Richard F. Riesenfeld. Computing exact shadow irradiance using splines. In *SIGGRAPH 99 Conference Proceedings*, August 1999.

[26] Seth Teller. Computing the antipenumbra cast by an area light source. *Computer Graphics (Proc. Siggraph '92)*, 26(2):139–148, 1992.

[27] Seth Teller, Kavita Bala, and Julie Dorsey. Conservative radiance interpolants for ray tracing. In *Seventh Eurographics Workshop on Rendering*, pages 258–269, June 1996.

[28] Andrew Woo. Fast ray-polygon intersection. In Andrew S. Glassner, editor, *Graphics Gems*. Academic Press, San Diego, 1990.