Combining Edges and Points for Interactive High-Quality Rendering

Kavita Bala[*]
Computer Science Department
Cornell University

Bruce Walter
Program of Computer Graphics
Cornell University

Donald P. Greenberg
Program of Computer Graphics
Cornell University

Abstract

This paper presents a new interactive rendering and display technique for complex scenes with expensive shading, such as global illumination. Our approach combines sparsely sampled shading (points) and analytically computed discontinuities (edges) to interactively generate high-quality images. The *edge-and-point* image is a new compact representation that combines edges and points such that fast, table-driven interpolation of pixel shading from nearby point samples is possible, while respecting discontinuities.

The edge-and-point renderer is extensible, permitting the use of arbitrary shaders to collect shading samples. Shading discontinuities, such as silhouettes and shadow edges, are found at interactive rates. Our software implementation supports interactive navigation and object manipulation in scenes that include expensive lighting effects (such as global illumination) and geometrically complex objects. For interactive rendering we show that high-quality images of these scenes can be rendered at 8–14 frames per second on a desktop PC: a speedup of 20–60 over a ray tracer computing a single sample per pixel.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing; I.3.3 [Picture/Image Generation]: Display algorithms

Keywords: interactive software rendering, sparse sampling and reconstruction, silhouette and shadow edges

1 Introduction

This paper presents *edge-and-point rendering*, a new rendering and display technique that can generate high-quality images of complex scenes at interactive rates. It targets expensive effects such as global illumination, where pixel shading is too slow for interactive use. In this context, sparse sampling of shading is essential for interactive rendering, and a number of systems have tried to accelerate rendering through sparse sampling and reconstruction (see Section 2). In the absence of shading discontinuities, sparse samples can be effectively interpolated to produce good images. However, interpolation across discontinuities results in visually objectionable blurring. This paper shows that analytically computed discontinuities can be combined with sparse shading samples to produce high-quality images at interactive rates.

Edge-and-Point Image. Figure 1 illustrates how edge-and-point rendering works. A novel intermediate display representation called the *edge-and-point image* (EPI) supports the new rendering process. Discontinuity edges and point samples are combined and stored with subpixel precision in the pixels of the EPI. Shading for each pixel is reconstructed by interpolating between nearby samples while obeying an *edge-respecting invariant*: interpolation is only performed using samples that are not separated from the pixel by a discontinuity. This invariant prevents the incorrect blurring produced by many sparse sampling and reconstruction algorithms. The explicit representation of discontinuities also enables anti-aliasing without supersampling. Interactive performance is achieved because the compact EPI representation permits fast, simple, table-driven interpolation and anti-aliasing.

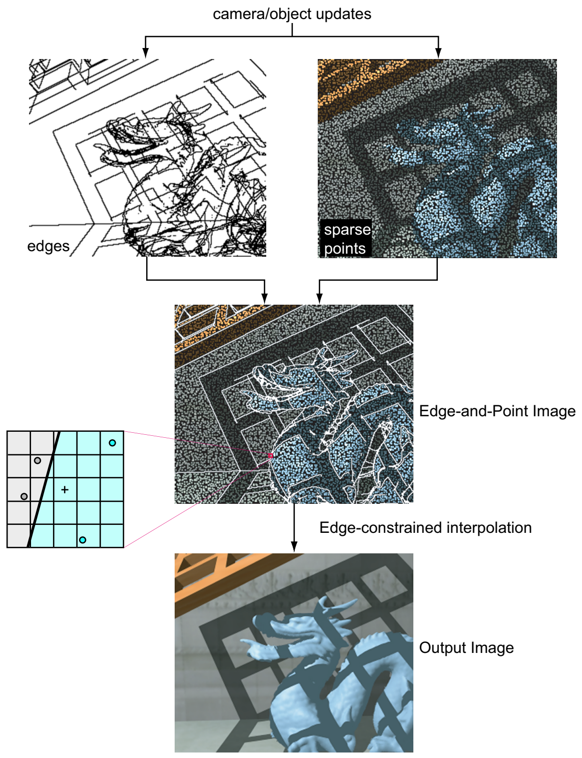


Figure 1: Edge-and-point rendering for the dragon with grid (∼ 871K polygons). After 3D edges and points are found, they are projected and combined into the edge-and-point image. The output image is computed by interpolating point samples, while respecting discontinuity edges. On the left a 5×5 neighborhood of pixels from the EPI is depicted. To reconstruct the center pixel, the blue samples are interpolated, while the unreachable gray samples are ignored.

The two most important shading discontinuities are typically geometric discontinuities, such as silhouettes, and shadow discontinuities. Our system analytically finds these discontinuities using efficient algorithms and data structures. Sparse shading samples are computed similar to the Render Cache [Walter et al. 1999]. Because edge-and-point rendering decouples shading from image reconstruction our implementation is flexible and can work with many different types of shaders, e.g., shaders for ray-traced glossy reflections and global illumination.

[*]email: kb@cs.cornell.edu, bjw@graphics.cornell.edu

Using the edge-and-point renderer, the user can navigate and manipulate objects in complex scenes while obtaining high-quality rendering feedback at interactive rates (8–14 frames per second on a desktop machine). The system collects samples for only 1–3% of the image pixels for any given frame, while reusing samples from frame to frame when possible.

2 Related Work

Sparse sampling and reconstruction. Several approaches adaptively sample and reconstruct smoothly varying shading. [Guo 1998] proposes a progressive technique that samples the image plane and tries to detect discontinuities. This system samples along hypothesized discontinuities and interpolates shading using these samples. The radiance interpolant system [Bala et al. 1999b] samples radiance in 4D ray space driven by conservative error analysis. Ray space is subdivided around discontinuities and radiance is interpolated where it varies smoothly. Both these approaches use sampling to detect discontinuities, thus requiring sampling patterns and densities that are difficult to maintain in an interactive context.

A progressive previewing technique that uses hardware to detect visibility and hard shadow edges in static scenes is presented in [Pighin et al. 1997]. A constrained Delaunay triangulation of sparse samples is used to reconstruct images. While this is an interesting approach, it is likely to be too expensive for the interactive rendering of complex scenes. Tapestry [Simmons and Séquin 2000] sparsely samples the scene and meshes samples using a Delaunay triangulation. Since this mesh is not in image space, it can be used as the viewpoint changes. Because this system does not try to detect discontinuities, it blurs edges.

The Render Cache [Walter et al. 1999] reprojects cached point samples from frame to frame. Interactive performance is achieved by interpolating available samples, but results tend to be blurry since no knowledge of edges exists. We have adapted the Render Cache for the point-based part of our algorithm.

The Shading Cache [Tole et al. 2002] displays scenes with expensive global illumination by interpolating sparse shading samples using graphics hardware. Because this technique requires at least one sample per visible polygon, it is too expensive for the geometrically complex scenes rendered in this paper.

Interactive ray tracing has become an active area of research [Parker et al. 1999; Wald et al. 2001]. The edge-and-point renderer is complementary to these approaches. Wald et al. [Wald et al. 2002] use fast distributed ray tracing and filtering to reduce image noise in parallel Monte Carlo simulations. A simple heuristic based on pixel depths and normals tries to detect discontinuities and avoid blurring over them.

Discontinuity meshing. The literature on finding discontinuities is extensive; visibility and shadow detection research is summarized in [Durand 1999]. Several approaches have tried to find all visibility events [Drettakis and Fiume 1994; Duguet and Drettakis 2002; Durand et al. 1997; Heckbert 1992; Lischinski et al. 1992]. However, the enumeration of all discontinuities is typically too slow for interactive use. Discontinuity meshing [Heckbert 1992; Lischinski et al. 1992] tesselates scenes based on radiosity discontinuities. Because it remeshes geometry around discontinuities, it often creates very large meshes with tiny mesh elements of poor aspect ratio. Our technique directly projects discontinuities onto the image plane, avoiding the need to construct a mesh.

Complexity. Recent efforts in 3D scanning have created massive data sets [Levoy et al. 2000; Rushmeier et al. 1998]. Point-based approaches [Pfister et al. 2000; Rusinkiewicz and Levoy 2000; Wand et al. 2001; Zwicker et al. 2001] suggest an alternative representation for the interactive display of these very large data sets. Because they use precomputed sampling, these techniques are mainly useful for the display of static scenes. None of these approaches try to support expensive illumination effects such as shadows and global illumination.

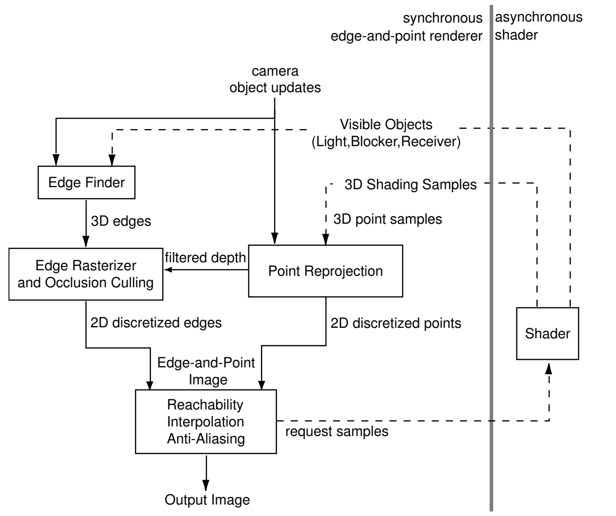


Figure 2: An overview of the edge-and-point rendering system.

Silhouette clipping [Sander et al. 2000] was proposed for the interactive viewing of complex objects with distinct silhouettes that would not be effectively represented by point-based approaches. It introduces cone-based hierarchical techniques to find silhouettes. A further extension uses hardware to anti-alias silhouette edges [Sander et al. 2001]. However, these systems are aimed at the interactive viewing of stand-alone objects and do not render scenes with expensive shading effects. [Johnson and Cohen 2001] extends the above cone-based data structure for a different application: local minimum distance computation. For a simple scene consisting of two smooth surfaces, a sphere and a torus, they also demonstrate the ability to compute shadow events, though these events are not used to compute shadow edges or used by a rendering system.

3 System Overview

Figure 2 shows an overview of our system. The system consists of two processes: the edge-and-point renderer and an external shader process. The shader asynchronously finds visible points and computes their color. These 3D point samples are cached and reprojected onto the image plane for each frame [Walter et al. 1999].

For each frame, the *edge finder* rapidly finds silhouettes and shadow edges for each visible object using hierarchical interval-based trees (Section 6). The *edge rasterizer* rasterizes these 3D edges, representing them to subpixel precision. Additionally, depth information from the reprojected point samples is used to perform conservative occlusion culling of the edges.

Both edges and point samples, represented at subpixel precision, are combined to form the edge-and-point image (Section 4). This compact representation discretizes positional information, enabling fast, high-quality interpolation of shading values using table-driven filtering algorithms (Section 5). These algorithms include *reachability determination*, which identifies the samples that are not separated by a discontinuity edge; *interpolation*, which reconstructs the pixel's shading using the computed reachability; and *anti-aliasing*.

Feedback from interpolation is used to decide where new point samples are needed and these requests are passed to the external shader. In addition to asynchronously computing shading samples, the shader also sends information to the edge finder about currently visible objects and shadows. This flexible architecture decouples the shader process from the edge-and-point renderer. Thus, almost any shader can be used to compute samples and visibility.

4 Edge-and-Point Image

The edge-and-point image (EPI) is the key intermediate display representation in the edge-and-point renderer. First, edges and shaded points are rasterized into the EPI. Each pixel in the EPI now contains its classification, including approximate edge location, if any, and the closest point, if any, that reprojected onto it. This produces a compact regular data structure whose size is independent of scene complexity. This representation allows fast image reconstruction. This section describes the EPI data structure and how it is computed for each frame.

4.1 Pixel classification

When the 3D edges from the edge finder are projected onto the image plane, they can vary in length from a fraction of a pixel to hundreds of pixels. Moreover, a single pixel may contain many edges or none. This edge information needs to be transformed into a compact regular format for rapid pixel processing. Therefore, pixels in the edge-and-point image are classified as empty, simple, or complex. Figure 3 shows examples of simple and complex pixels.

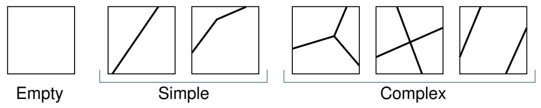


Figure 3: Examples of pixel classifications.

- **Empty**: A pixel with no edges is empty.
- **Simple**: A pixel is simple if the edge(s) through the pixel can be approximated well by a single edge spanning the pixel.
- **Complex**: Pixels that are neither empty nor simple are considered complex.

Most pixels are empty or simple. Our system reconstructs shading for simple pixels at the same cost as empty pixels as described in Section 5. Identification of simple pixels also permits anti-aliasing for higher image quality. Additionally, the ability to identify complex pixels allows us to concentrate more effort where needed.

4.2 Edge representation

For a simple pixel, its edge is approximated by locating its endpoints on the pixel boundary to eighth-of-a-pixel accuracy. The pixel boundary is subdivided into 32 segments (8 on each side) as shown in Figure 4 and the approximated edge is represented by 10 bits (5 bits for each end point). The approximated edge is not allowed to have both endpoints on the same side of the pixel; two of these disallowed combinations are used to encode empty and complex pixels. This subpixel precision is sufficient to permit good anti-aliasing while compactly encoding edges, thus, allowing fast edge-related operations using table lookups.

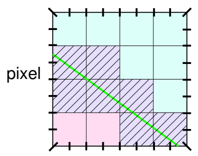


Figure 4: A single pixel crossed by an edge. Each side is divided into 8 segments and the interior into 16 regions. An edge divides the pixel into primary (blue) and secondary (pink) regions. The cross-hatched regions are classified as edge-ambiguous.

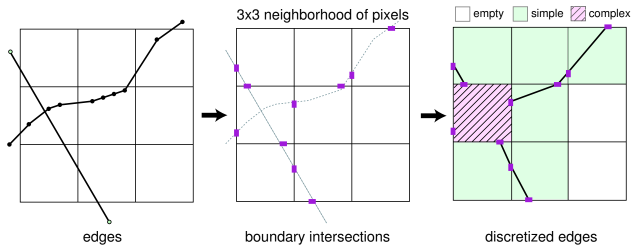


Figure 5: Edge rasterization for a 3×3 pixel neighborhood. 3D edge segments are shown delimited by points on the left. These edges are rasterized to compute their discretized intersections with pixel boundaries, shown in purple (middle). Pixels are classified from the boundary intersections as either empty (white), simple (green), or complex (cross-hatched pink), shown on the right. For simple pixels, an approximated edge is reconstructed.

4.3 Point representation

The EPI also compactly records information about shading samples. Each pixel is divided into 16 interior regions as shown in Figure 4. For each sample we record which subpixel region it lies in, requiring 4 bits per sample. Because both edges and points are represented with limited accuracy, it cannot always be determined on which side of an edge a point sample lies. Such samples are called *edge-ambiguous* and cannot be used for edge-respecting interpolation. For simple pixels, table lookups are used with the edge's 10-bit encoding and the sample's 4-bit subpixel position to determine on which side of the edge the sample lies, or if the sample is edge-ambiguous.

Point processing. Point processing in the edge-and-point renderer is based on the Render Cache [Walter et al. 1999; Walter et al. 2002]. Point samples are produced by an external shading process that computes the color and first point of intersection for a viewing ray. The Render Cache manages a fixed-sized cache of recent shading samples, which are stored as colored points in 3D. For each frame, the cached points are reprojected using the current camera parameters, filtered to reduce occlusion errors, and interpolated to fill in small gaps between reprojected samples.

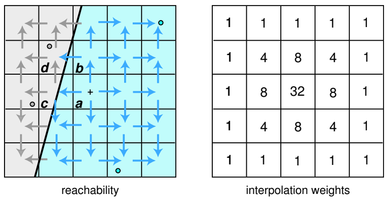
4.4 EPI pixels

Each pixel of the EPI includes at most one discretized edge and one point sample. Thus, the EPI size depends only on image resolution and is independent of scene complexity. A pixel in the edge-and-point image uses 10 bytes of storage: 10 bits for edge encoding, 4 bits for sample subpixel location, 24 bits for sample color, and 36 bits for other Render Cache data.

4.5 Constructing the edge-and-point image

The first step of EPI construction is edge rasterization. Edges are rasterized by recording their intersections (shown in purple in Figure 5) with all pixel boundaries. This can be done quickly, because it is equivalent to recording the edges' intersections with regularly spaced parallel horizontal and vertical lines. Each pixel boundary segment is divided into 8 pieces and its intersections are stored in an 8-bit mask; since neighboring pixels share intersections, only 16 bits per pixel are needed to store the intersections.

Eliminating occluded edges makes interpolation more effective. Since points approximate the visible surfaces, depth information from point reprojection can be used to cull occluded edges. However, points and edges do not exactly coincide and there are gaps in the point data, so the depth buffer must be filtered first. We compute a fairly conservative depth buffer by using the maximum depth value in a 4×4 neighborhood and adding an additional offset that depends on the local point density. These filtered depths are used



1	1	1	1	1
1	4	8	4	1
1	8	32	8	1
1	4	8	4	1
1	1	1	1	1

Figure 6: Reachability computation and interpolation weights for a 5×5 neighborhood of pixels. Weights are renormalized after excluding unreachable pixels.

as a read-only z-buffer during edge rasterization. This occlusion culling typically removes about half the edges.

Edge reconstruction. After rasterization, pixels are classified and edges are reconstructed from the boundary intersections, using these simple rules. If a pixel has zero intersections then it is considered empty. If a pixel has two intersections on its boundary and they are not on the same side of the pixel, then it is simple and a single edge is reconstructed (shown on the right in Figure 5). Otherwise, the pixel is complex.

5 Reconstruction

In image reconstruction, the EPI is used to compute the color for each pixel. The first step is to compute a *reachability map* that determines which samples may be used to interpolate color for each pixel. Then pixels are interpolated using this map. Finally, a separate pass reconstructs complex pixels more accurately.

5.1 Pixel regions

Each simple pixel is divided into two regions by its edge (Figure 4), termed the primary and secondary regions. To correctly anti-alias a simple pixel, color must be determined for both regions. For performance, color is initially reconstructed only for the primary region. Color for the secondary region is approximated using the reconstructed color from a neighboring pixel. If a pixel has a sample, the region containing the sample is considered primary. Otherwise, the larger region is primary. For empty and complex pixels, the entire pixel is considered the primary region.

5.2 Reachability map

When reconstructing radiance for the primary region of a pixel, only reachable samples are used to interpolate the radiance at that pixel. A sample in pixel region b is considered *reachable* from a pixel region a if there exists a reasonably direct path that does not cross any edges from a to b. Let us denote the primary region of pixel x as P_x and the secondary as S_x. The reachability between two pixels a and b is denoted as $a \leftrightarrow b$ and can be encoded in 4 bits: ($P_a \leftrightarrow P_b$, $P_a \leftrightarrow S_b$, $S_a \leftrightarrow P_b$, $S_a \leftrightarrow S_b$), where $\leftrightarrow$ denotes reachability. Reachability is computed as shown in Figure 6. There are three main operations in reachability computation:

1. Neighbor reachability: The reachability between pixels that share a boundary (e.g., pixels c and d in Figure 6) can be found using a lookup table based on the edge encodings for the pixels. Only 4 bits out of each edge's 10-bit encoding are used in this lookup. This optimization is possible because two adjacent pixels always have the same intersection on their shared boundary; therefore, the subpixel location of the boundary crossing is not needed. The 4 bits per edge used in the lookup encode if its pixel is empty, simple or complex, and if simple, which two sides of the pixel the edge intersects. Thus, a 256-entry table indexed by two 4-bit edge encodings returns the 4-bit reachability mask between the two pixels.

2. Chaining: Reachability between two pixels a and d through an intermediate pixel b (Figure 6), denoted $a \overset{b}{\leftrightarrow} d$, is obtained by combining $a \leftrightarrow b$ with $b \leftrightarrow d$ using a lookup table. The lookup table is indexed by the two 4-bit reachability masks and returns the 4-bit mask $a \overset{b}{\leftrightarrow} d$. Chaining considers potential reachability paths through both the primary and secondary regions of intermediate pixel b.

3. Combining: Reachability between two pixels along multiple paths (e.g., $a \overset{b}{\leftrightarrow} d \cup a \overset{c}{\leftrightarrow} d$) is combined by simply taking a bit-wise boolean OR of the reachability masks of the paths.

Example. To compute reachability between pixels a and d in Figure 6, we first find the reachability between neighbors using the neighbor reachability lookup table: $a \leftrightarrow b = (1,0,0,0)$, $a \leftrightarrow c = (0,1,0,0)$, $c \leftrightarrow d = (1,0,0,1)$, $b \leftrightarrow d = (0,1,0,0)$. Pixels a and b are both empty and therefore are considered to have only a primary region. The gray region is primary in pixel d because it contains a sample. Since pixel c does not have a sample, the larger region (gray) is considered its primary. The reachability between a and d can be computed along two paths, through b and c respectively. Using the chaining lookup table we find: $a \overset{b}{\leftrightarrow} d = (0,1,0,0)$ and $a \overset{c}{\leftrightarrow} d = (0,1,0,0)$. Combining these gives the final reachability $a \leftrightarrow d$: $(0,1,0,0)$. Note that the primary-to-primary reachability is 0; that is, the gray sample in d is unreachable from a.

Complex pixels are always considered unreachable and reachability is not propagated through them.

Reachability is first propagated to a pixel's immediate neighbors across their shared boundary along the arrows shown in Figure 6 and then from them to their neighbors. At each step, reachability is propagated using the three operations above, always away from the center pixel. Together these operations allow reachability to be quickly computed in the 5×5 neighborhood.

Since the goal is to reconstruct radiance in the primary region of a pixel, and samples always lie in a pixel's primary region, reachability is finally needed only between primary regions. However, since two primary regions might be reachable via another pixel's secondary region, reachability is computed between both primary and secondary regions. Once it is computed, the primary-to-primary reachability from a pixel to every other pixel in its 5×5 neighborhood is stored in a 24-bit mask. Since reachability is symmetric, a is reachable from d if and only if d is reachable from a, reachability of pixels on prior scanlines is reused for efficiency.

5.3 Interpolation

Once reachability is computed, the colors for pixels are reconstructed using constrained interpolation. For each pixel's primary region, color is reconstructed using all the reachable samples in its 5×5 neighborhood. The particular filter weights we use are shown in Figure 6; reconstruction is not sensitive to the choice of weights, provided the filter is sharply peaked in the center. Any pixel that is either unreachable or does not contain a sample point is given a weight of zero and then the weights are renormalized.

Prioritized sampling. Future sampling is prioritized using feedback from interpolation. Our sampling strategy is essentially the same as in the Render Cache, except that edges influence the sampling distribution. As in the Render Cache, the priority for pixels without points is inversely related to the total interpolation filter weight before renormalization. More samples are concentrated in regions where the interpolation filter found low point densities, and pixels without reachable samples are given the highest priority. The presence of edges reduces the number of nearby reachable samples, and therefore increases the sampling near edges.

Anti-aliasing. A separate pass anti-aliases simple pixels using an area-weighted combination of the colors of its primary and secondary regions. For efficiency, the shading of the secondary region is approximated by looking up the color of a neighboring pixel's primary region. An edge-indexed table is used to choose the neighbor most likely to contain a good approximation for the secondary

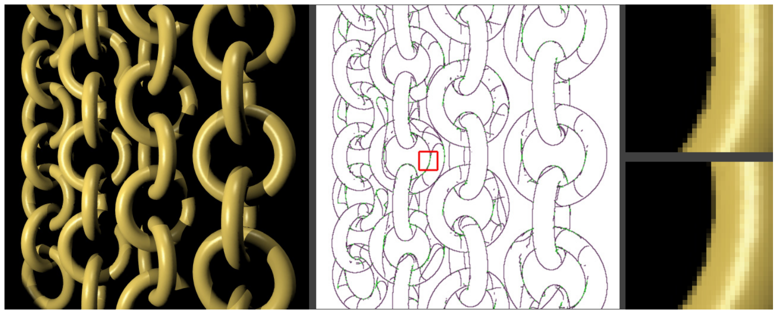


Figure 7: Chains scene with tesselated torii. The edge image shows simple pixels in purple and complex pixels in green. The images on the right show reconstruction for the part of the image shown in the red box without (top) and with (bottom) the complex filter.

color and to obtain the relative weights to use. This filter is cheap and effective.

The EPI representation lets us distinguish between simple and complex pixels, thus permitting better reachability propagation, interpolation and anti-aliasing for simple pixels.

5.4 Complex pixels

The mechanisms described thus far do not generate enough information to accurately shade complex pixels. One simple approach is to interpolate a color for them while ignoring reachability. At the very least, this color is consistent with the local region of the image. It is computed during interpolation by simply considering all neighbors reachable from complex pixels (without affecting reachability for other pixels).

A better approximation is computed at a slightly higher cost by rasterizing the edges at a higher resolution. We add horizontal and vertical boundaries through the center of each pixel and record intersections with them during edge rasterization. Intersections are still recorded to eighth-of-a-pixel precision along each boundary, and these additional boundaries are ignored by all the prior reconstruction steps. However, the new boundaries divide each pixel into 4 subpixels, enabling edge reconstruction in each subpixel.

For each complex pixel, we subdivide its 3×3 neighborhood into a 6×6 subpixel neighborhood. Edge reconstruction and reachability are computed for subpixels similarly to the prior stages. For a complex pixel, typically some of its 4 subpixels will be simple or empty and can have colors interpolated for them. Any subpixel region that is still complex or has no reachable samples uses the complex pixel's estimated color from the simple interpolation. The complete pixel is then shaded by combining the colors of the subpixel regions.

This additional filter for complex pixels subtly but noticeably improves image quality. An example is seen in Figure 7. Even more accurate reconstruction is possible by supersampling complex pixels. While this option is less attractive for an interactive system, it could be useful for progressive rendering.

6 Finding Edges

Edge-and-point rendering can be used with a variety of techniques for discontinuity finding. We have developed fast algorithms for computing discontinuities arising from silhouette and shadow edges, including soft shadow edges. For each frame, the silhouettes of the currently visible objects are computed from the current camera position. This recomputation is needed because silhouettes are view-dependent. Shadow edges, which are view-independent, are computed once and reused when possible.

6.1 Silhouettes

A point on a surface is on an object's silhouette if the normal at that point is perpendicular to the view vector. In a polygonal scene, an edge is on an object's silhouette if one of the edge's adjacent faces is forward-facing while the other face is backward-facing, as shown on the left in Figure 8. Thus, an edge e is a silhouette edge if the following test succeeds:

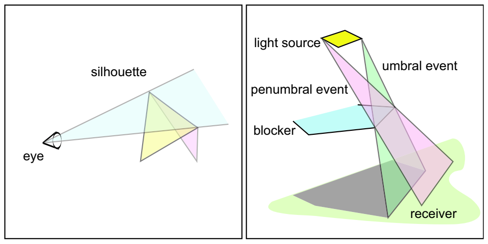


Figure 8: Sources of silhouette and shadow edges.

$$\operatorname{sign}(\mathbf{N}_{f_0} \cdot \mathbf{V}_{f_0}) \neq \operatorname{sign}(\mathbf{N}_{f_1} \cdot \mathbf{V}_{f_1}) \quad (1)$$

Here, f_0 and f_1 are the two polygons adjacent to the edge e, $\mathbf{N}_f$ is the normal of polygon f, and the view vector $\mathbf{V}_f$ is a vector from a vertex of the face f to the current viewpoint.

6.2 Normal discontinuities

A simple form of geometric discontinuity is an edge that is intended to be perceived as sharp (e.g., the edges of a box). These edges cause shading discontinuities. The edge finder reports these edges if they are forward-facing with respect to the viewpoint. For a given mesh, the user can specify a cutoff angle that determines which edges are considered sharp. In our scenes, the cutoff angle is 85°.

6.3 Shadows

A shadow on a receiving object (the receiver) occurs when a light is occluded by an intervening object (the occluder or blocker). A blocker creates shadow *events* with respect to a light. In polygonal scenes, these shadow events intersect receivers to create *shadow edges*. In polygonal scenes with area lights, two types of shadow events cause shadow discontinuities [Gigus et al. 1991; Heckbert 1992; Lischinski et al. 1992]: vertex–edge events, and edge–edge–edge (EEE) events. Vertex–edge events are planes (actually, wedges) defined by a vertex of the light and an edge of the blocker, or vice-versa. The algorithm described here only finds vertex–edge events, which are more common than EEE events and result in visually more important discontinuities.

Umbral and penumbral events are shown in Figure 8 on the right. They occur when the plane through the vertex v and edge e is tangential to the light and the blocker, and the light and blocker lie on the same side or opposite side of the plane respectively. For point lights, the umbral and penumbral events coincide.

The hard shadows caused by the occlusion of a point light source by a blocker create perceptually important D_0 radiance discontinuities. Soft shadows arise from the occlusion of an area light source and generate D_1 discontinuities [Heckbert 1992]. It is important to identify soft shadow edges near the point where a blocker contacts a receiver. At that point, soft shadows show a "hardening" effect in which the radiance gradient increases [Wanger et al. 1992]. Identifying penumbral and umbral shadow boundaries allows this effect to be rendered correctly (see Figure 9).

Shadow discontinuities are computed in two steps. First, shadow events produced by lights and blockers are found. Then, shadow edges are computed by intersecting the shadow events with receivers.

6.4 Accelerating silhouette computation

A brute-force identification of silhouette edges and shadow events can be done using Equation 1 and (for shadow events) Appendix A. Our implementation further accelerates the computation by using a tree to store and look up the edges of an object. The leaves of this tree store edges; the internal nodes of the tree conservatively represent the normals and positions of all the contained edges. Unlike the cone-based representations in [Sander et al. 2000; Johnson and Cohen 2001], normals and positions are represented by *interval vectors*, vectors of 3 intervals[1].

The edge tree is constructed top-down. At each node of the tree, the node is subdivided on one of the three axes of the normal vector so that the combined size of its children on those axes is minimized. This construction has low preprocessing overhead (about 7 minutes for our largest scene, the dragon in Figure 13); it appears to be as effective as the more specialized hierarchical construction in [Sander et al. 2000] that takes hours to construct.

Tree traversal for silhouette and shadow events

Silhouettes and hard shadow events are found by traversing the tree. During the traversal, Equation 1 is evaluated at each node reached. Using interval arithmetic, the dot product of the normal interval vector and the interval vector for the view vector, $\mathbf{V}$, results in an interval $[r_0, r_1]$[2]. If $[r_0, r_1]$ does not contain zero, all edges represented by the node are either forward- or backward-facing and are therefore not silhouette edges. In this case, the sub-tree is pruned from the search; otherwise, the subtree is recursively explored. When a leaf node is reached, all its edges are tested using Equation 1 to determine if they are on the object silhouette.

For point lights, shadow events are essentially silhouette computations from the light's point of view. Therefore they are found by the same tree traversal. For area lights, our algorithm finds the penumbral and umbral VE and EV events using a recursive traversal of the blocker's edge tree. Using intervals permits us to uniformly use the same test (Equation 1) for area lights and point lights at internal nodes. Leaf node tests for umbral and penumbral events are described in Appendix A.

Shadow edge computation

Given the VE and EV event wedges, shadow edges are computed by intersecting these events with the receiver geometry. This process is accelerated using a bounding volume hierarchy for the receiver faces. Each shadow event is walked down the receiver's face tree to find intersections of the event with the geometry represented by that node of the tree. The hierarchy is extended to include a conservative normal interval vector at each internal node. These normal interval vectors are used to eliminate entire sub-trees of receiver polygons that are back-facing with respect to the light.

[1]An interval vector used to represent a range of normals has three intervals each representing the range of the corresponding Cartesian component of the normals. Operators on interval vectors are defined using the usual interval arithmetic operations [Moore 1979]; e.g., the dot product of two interval vectors is computed as a component-wise interval multiplication and an interval addition of the resulting intervals to obtain the final interval.

[2]The interval vector for $\mathbf{V}$ is computed using the position interval vector of the node.

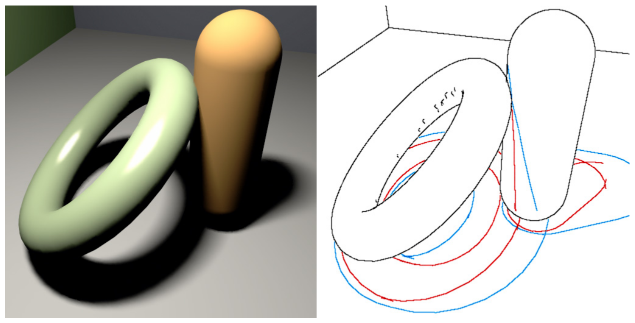


Figure 9: Umbral and Penumbral edges: umbral edges in red, penumbral edges in blue. Notice the coincidence of umbral and penumbral edges where the torus contacts the capsule, resulting in the correctly rendered "hardening-at-contact" shadow effect.

Non-smooth surfaces

The edge tree has similarities to the cone-based hierarchies of [Johnson and Cohen 2001; Sander et al. 2000]; we chose the interval representation for its simplicity. Both [Johnson and Cohen 2001; Sander et al. 2000] note that their cone-based data structures work less well for meshes produced from scanned data, because these meshes often contain "sharp" edges whose normals vary significantly. Such edges make the data structure less effective at pruning the search space.

To address this problem, we place sharp edges higher in the tree. During top-down tree construction, an edge is stored at a node if its normals span more than half of the node's entire range of normals. This optimization makes normals represented by the child nodes more compact. When an internal node is reached while traversing the tree, all its stored edges are tested. This simple extension to the data structure substantially improves its performance on complex meshes: for the David head (Figure 12), performance improves by a factor of 2.

6.5 Complex Scenes

A few additional features are important for the interactive detection of silhouettes and shadow edges in complex scenes.

Finding light-blocker-receiver triples. To determine the shadow edges for a receiver, the lights and blockers that cast shadows on that receiver must be identified. Exhaustive computation of shadow events for all potential light-blocker pairs would be too slow. Instead, the shading process asynchronously communicates the light-blocker-receiver triples that it discovers during shading to the edge finder. All shadow edges associated with a receiver are stored with the receiver, and reused as long as the light, blocker and receiver do not move.

Clipping. A nearby receiver may prevent a shadow event from propagating to more distant receivers. For each shadow event, we sort the shadow edges it generates and clip them against each other, retaining only the unoccluded edges. Clipping is simple and efficient because it is performed in one dimension, along the edge of the VE/EV event.

Moving objects. If a light or blocker are moved, the shadow events associated with that light-blocker pair are recomputed and the shadows for receivers associated with that pair are also recomputed. When a receiver moves, its shadows are affected; other receivers that share a light-blocker pair with that receiver and are farther from the light might also be affected by the move because they are no longer occluded by the moved object. To be conservative, all these shadows are recomputed.

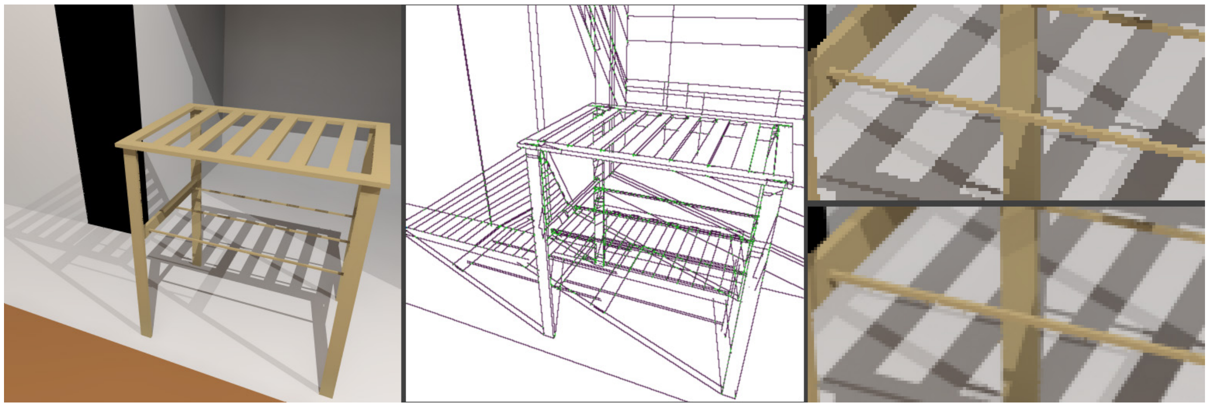


Figure 10: The left image shows a tea-stand with fine geometry and complex shadows. The middle image shows the edges found; complex pixels are shaded in green. The images on the right compare a standard 1-sample-per-pixel sampling (top) against our technique (bottom).

Queries into the tree data structure are performed in object coordinates, thus eliminating the need to rebuild these data structures when an object is moved.

6.6 Discussion

Unlike discontinuity meshing techniques, our goal is to achieve interactive performance. Therefore, we focus on the shadow edges that are typically the most important perceptually: those corresponding to silhouettes, hard shadow events, and umbral and penumbral VE/EV events. Additionally, computed shadow edges are directly rendered onto the image plane; we never explicitly mesh discontinuities.

7 Implementation and Results

In this section we present results from our edge-and-point based rendering system. All result images are 512×512. We use three different shaders: direct illumination with ambient, direct with glossy reflections, and direct with global illumination. The edge-and-point renderer runs on a single 2.8 GHz Pentium 4. All the shaders use ray tracing. The direct illumination shader runs on a single 2.8 GHz Pentium 4. The glossy reflection shader uses ray tracing to sample the glossy reflection and runs on four 1.7 GHz processors. The global illumination shader uses a dynamic variant of irradiance caching [Ward et al. 1988] modified for interactive use. The global illumination shading is computed on four 1.7 GHz processors, and 6 processors asynchronously compute the irradiance samples.

7.1 Scene descriptions

Scene	Polygons	Edges	Shader
Rooms	15,006	23,004	global illumination
Chains	73,728	110,592	direct
Mack	101,306	153,526	global illumination
David	250,014	374,635	glossy reflection
Dragon	872,184	1,305,164	direct

Table 1: Scene statistics and the shaders used for each scene.

Table 1 lists the scenes used to acquire results. The Rooms scene (Figure 12) shows a textured multi-room scene with six lights. The Chains scene (Figure 7), with two lights, has tesselated non-convex objects casting complex shadows on each other. The edge image shows our ability to correctly find silhouettes and shadows.

The Mackintosh room (Figures 10 and 11), has three lights and is rendered with the global illumination shader. The David head from Stanford's Digital Michelangelo project (Figure 12), has 250k polygons and is lit by 1 light; the scene is rendered using glossy reflections. The Dragon scene (Figure 13) has a grid casting a shadow pattern on a dragon with 871k polygons (from Georgia Tech's Large Geometry Models Archive). All these scenes also demonstrate self-shadowing of complex objects.

7.2 Performance

Table 2 gives the breakdown of time (in milliseconds) spent on the important components of the algorithm for a typical frame: from left to right, edge finding, edge rasterization, point reprojection, reachability, interpolation, anti-aliasing and complex filtering. The total time is slightly higher than the sum of all these modules since it includes all processing.

Silhouette finding is efficient, ranging from 0.5 ms up to 22 ms for the most complex model, the dragon, which has 1.3M edges. Edge rasterization is slowest for the David, because it has the most silhouette and shadow edges (see Table 3). Point reprojection costs are essentially unchanged from the Render Cache.

Reachability, interpolation, and anti-aliasing have constant cost per frame, because the compact EPI and their implementation using fast table lookups makes these stages independent of scene and edge complexity. Their run time only depends on image resolution. The cost of the complex filter depends on the number of complex pixels (see Table 3).

Table 2 presents the frame rate in frames per second; numbers in parentheses give the slightly improved frame rates achieved with complex filtering turned off. The system achieves interactive frame rates of about 8–14 frames per second.

The framerate is essentially independent of the speed of the shader being used, although the quality of the images does depend on the number of newly shaded samples per frame. The sampling sparseness ratio is also reported for the examples given. This is the ratio between the number of pixels in an image and the number of new samples acquired per frame. The sparseness ratio is between 40 to 125; thus only 1 to 3% of the point samples are updated each frame. The ability to tolerate such sparse sampling is crucial for good interactive image quality when using expensive shaders.

Finally, we compare our results with a ray tracer that samples every pixel in each frame. Frames for this ray tracer were computed using our shaders and the same number of processors as used by our system (i.e., the display processor is also used for shading by this ray tracer). Compared to this ray tracer we achieve speedups of 20 to 65. We also produce higher quality images with anti-aliasing (see comparison on the right of Figure 10).

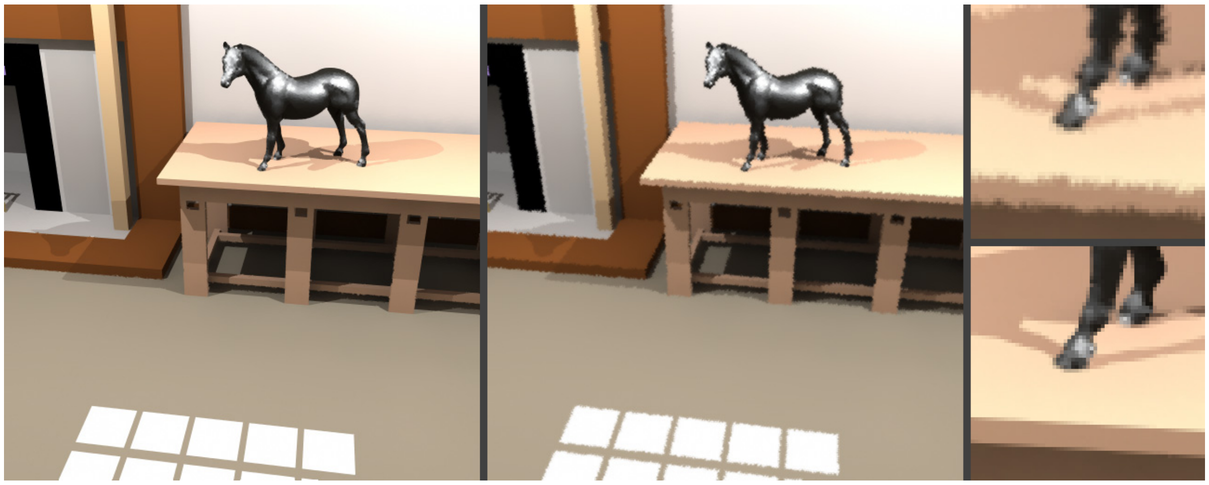


Figure 11: Mackintosh Room comparing edge-preserving interpolation (left and bottom right) vs. standard interpolation similar to the Render Cache (middle and top right) from the same samples. For both images, only 20% of the pixels have samples. The images on the right show a magnified comparison.

Scene	Find Edges	Rasterize Edges	Point Proc.	Reach	Interp	Anti Alias	Complex Filter	Total (ms)	**FPS**	Sparseness Ratio	Full RT (s)	**Speedup**
Rooms	0.5	17	26	11	14	2	17	95	**10.5 (13.3)**	97 (124)	4.8	**51 (64)**
Chains	4	13	26	11	14	2	6	88	**11.4 (12.3)**	52 (58)	2.0	**23 (25)**
Mack	2.5	11	26	11	14	2	6	77	**13.0 (14.3)**	92 (101)	4.5	**58 (64)**
David	9	39	27	11	14	2	16	127	**7.9 (9.7)**	65 (80)	5.1	**40 (50)**
Dragon	22	29	26	11	14	2	6	124	**8.1 (8.8)**	40 (40)	2.3	**19 (20)**

Table 2: Performance results. Frame breakdown of different components and total time (in milliseconds), frame rates, and speedup numbers.

EPI complexity. Table 3 presents the detailed number of silhouette and shadow edges for a typical frame of each scene. The number of shadow events for a light-blocker pair is proportional to silhouette edge size. Theoretically, the total number of shadow events is proportional to $n_{lights} \times n_{sil}$, where n_{sil}, the number of silhouette edges, is usually $O(\sqrt{n})$ in the number of polygons [Sander et al. 2000].

Shadow edges are computed for each light-blocker-receiver triple, and depend on the receivers associated with each light-blocker. Thus, a large blocker casting a shadow on a finely tesselated receiver could generate several shadow edges per event. Conversely, a finely tesselated blocker could cast only a small number of shadow edges per event. In theory, each shadow event could cause $O(n)$ shadow edges. In practice, the number of edges is substantially smaller than $(n_{lights} n\sqrt{n})$, as can be seen from our results.

The detailed edge counts give a flavor of the number of edges interactively computed and manipulated by our system. The David head has the largest number of edges because of its finely tesselated, irregular geometry, which results in significant self-shadowing.

Table 3 also gives a detailed breakdown of the pixel classifications. The fraction of complex pixels is small for all of the scenes. Most pixels are empty or simple, and can be efficiently and accurately reconstructed by the edge-and-point renderer.

Scene	# Sil Edges	# Shadow Edges	Pixel Categorization		
			Empty	Simple	Complex
Rooms	1,814	7,356	87.0%	8.9%	4.1%
Chains	3,644	14,138	92.0%	6.9%	1.1%
Mack	3,669	27,917	89.1%	9.7%	1.2%
David	39,116	163,338	92.5%	3.5%	4.0%
Dragon	26,269	92,884	92.9%	5.9%	1.2%

Table 3: Number of silhouette and shadow edges and pixel categorization for a typical 512×512 frame in each scene.

Edge finding. The preprocessing time to construct the edge trees ranges from about a second to a maximum of 7 minutes for the dragon scene (as compared to hours for [Sander et al. 2000]).

Silhouette finding and shadow event computation is fast (see Table 2). The edge tree appears to be about as effective in pruning the search space as the cone-based data structure of [Sander et al. 2000]; therefore detailed results are omitted.

The torus scene (30,000 polygons) in Figure 9 shows soft shadow edges for an area light. Computing shadow events for area lights is approximately twice as expensive as a silhouette computation since both umbral and penumbral events must be computed.

Shadow edge computation time varies substantially depending on the viewpoint and the light-blocker-receiver triples being processed. Therefore, we summarize the total time for shadow computation over all scenes. When our scenes are loaded, initial shadow computation takes 0.1 to approximately 2 seconds (for the David head). As the user moves around or manipulates objects in later frames, the shadow computation time ranges from 1 to 50 milliseconds depending on how many new light-blocker-receiver triples are found by the shader. When the grid is moved in the Dragon scene, shadows on the dragon are updated in 50 ms. The average cost of intersecting a shadow event with the scene to compute shadow edges ranges from 8–28 μs per event depending on the scene.

7.3 Sparse sampling and anti-aliasing

The Mackintosh scene shows the ability of the renderer to find and anti-alias fine shadow details. On the right in Figure 10, magnified images of the tea-stand are shown. On the top is the result produced by a regular ray tracer using 1 sample per pixel; on the bottom is the anti-aliased result produced by our system.

Figure 11 compares our image with an image produced using interpolation that does not respect edges (like the Render Cache).

Both images use the same samples, and only 20% of the pixels contain samples. We can see that using edge-respecting interpolation produces a significantly better image, while using the same samples. Even at this low sampling density we are able to effectively anti-alias the image.

Our system can reconstruct good images with very sparse sampling densities as can be seen in the video. For example, in the Mackintosh room, only 1 out of every 100 pixels had a new sample shaded per frame. By reusing and interpolating sparse samples we produce high-quality images at interactive frame rates.

7.4 Memory Usage

The per-pixel memory costs for various data structures are as follows: the EPI uses 10 bytes (Section 4.4), reachability masks require 3 bytes, and boundary intersections require 2–4 bytes depending on whether complex filtering is turned on. For 512×512 result images, these structures use 3.75–4.25 MB of memory. The various lookup tables use less than 16 KB. Point-based processing requires the same space as the Render Cache ($\sim$ 9 MB). The edge trees are linear in the size of the objects.

7.5 Discussion

From Table 2 we see that the cost of the 3 main filters (reachability, interpolation, and anti-aliasing) is independent of scene complexity and the number of edges. One of the strengths of the EPI representation is this decoupling of the cost of reachability and interpolation from scene complexity.

Edge finding and rasterization are dependent on scene complexity but are typically fast enough. It should be possible to further accelerate these components using programmable graphics hardware.

Scenes with a very large fraction of complex pixels violate the assumptions made by sparse sampling and reconstruction algorithms. Fundamentally, such scenes must be reconstructed at higher resolutions for reasonable image quality. While our algorithm will slow down for these scenes, our image quality should not be worse than traditional sampling and reconstruction algorithms for these scenes. We expect our techniques would also be useful in the context of non-interactive progessive viewing for these scenes.

When objects are moved, some shading samples become invalid and new samples must be acquired. In addition, shadow edges must be recomputed. In our results, the time required to obtain updated shading samples is the bottleneck; shadow edges are computed fast enough. Better techniques for sample invalidation (e.g., [Bala et al. 1999a]) would speed up shading updates for dynamic scenes.

8 Conclusions

This paper introduces a new interactive rendering technique that combines shading discontinuities (edges) and sparse samples (points) to generate high-quality, anti-aliased images. Discontinuity edges are found interactively and are projected on the image plane. Shading is reconstructed from sparse samples using edge-respecting interpolation. We have demonstrated that our system renders high-quality anti-aliased images at interactive rates of many frames per second in scenes including geometrically complex objects and lighting effects such as shadows and global illumination with very low sampling ratios (1%–3% of the pixels per frame). The user can also dynamically move objects within the environment.

We describe a new compact representation of discontinuities and point samples called the *edge-and-point* image; this image is used by efficient interpolation algorithms to reconstruct radiance. The explicit detection of discontinuities permits the generation of anti-aliased output images, without supersampling, at interactive rates of 8–14 fps on a modern desktop computer.

Future Work. A promising avenue of future research is to accelerate our edge-and-point renderer using the new programmable graphics hardware. Our table-driven filters are simple enough for this to be feasible. Hardware assisted silhouette and shadow edge detection should also be possible.

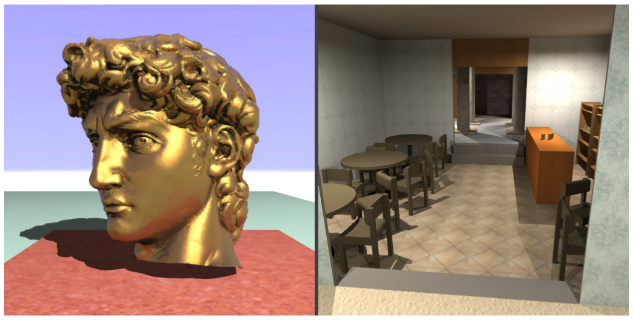


Figure 12: David head and Rooms scene.

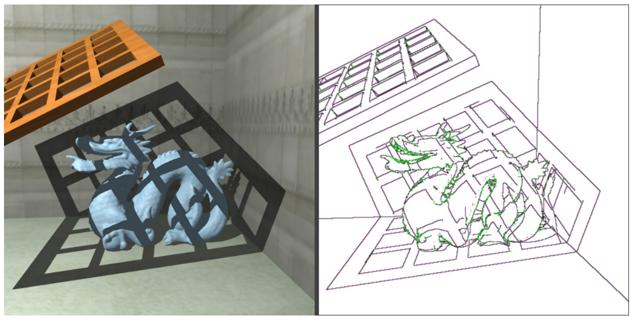


Figure 13: Grid casting shadow on Dragon.

Our system identifies two important types of discontinuities; other discontinuities such as mirror reflections, specular transparency, and textures can be perceptually important for some scenes. It appears possible to extend interval-based edge finding for reflection and transparency edges. For some textures it may also be beneficial to include edges from the texture in the EPI.

To improve the rendering of complex scenes, it would be useful to identify the perceptually important discontinuities in an image. For example, in scenes with large numbers of lights, discontinuities from unimportant lights could be eliminated [Ward 1994].

Acknowledgements. The authors thank Sebastian Fernandez, Jim Ferwerda, Andrew Myers, Steve Westin, and the anonymous reviewers for their feedback. We also thank Stanford's Digital Michelangelo project, the Georgia Tech Large Geometry Models Archive, Gene Gregor, and Reynald Dumont for their models. We acknowledge the support of NSF grant ACI-0205438 and Intel Corporation for generously providing funding and equipment.

References

BALA, K., DORSEY, J., AND TELLER, S. 1999. Interactive ray-traced scene editing using ray segment trees. In *10th Eurographics Workshop on Rendering*, 39–52.

BALA, K., DORSEY, J., AND TELLER, S. 1999. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics 18*, 3, 213–256.

DRETTAKIS, G., AND FIUME, E. 1994. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH '94*, 223–230.

DUGUET, F., AND DRETTAKIS, G. 2002. Robust epsilon visibility. In *SIGGRAPH '02*, 567–575.

DURAND, F., DRETTAKIS, G., AND PUECH, C. 1997. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *SIGGRAPH '97*, 89–100.

DURAND, F. 1999. *3D Visibility: Analytical Study and Applications*. PhD thesis, Grenoble University.

GIGUS, Z., CANNY, J., AND SEIDEL, R. 1991. Efficiently computing and representing aspect graphics of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 6, 542–551.

GUO, B. 1998. Progressive radiance evaluation using directional coherence maps. In *SIGGRAPH '98*, 255–266.

HECKBERT, P. 1992. Discontinuity meshing for radiosity. In *3rd Eurographics Workshop on Rendering*, 203–226.

JOHNSON, D., AND COHEN, E. 2001. Spatialized normal cone hierarchies. In *Symposium on Interactive 3D Graphics*, 129–134.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The digital Michelangelo project: 3D scanning of large statues. In *SIGGRAPH '00*, 131–144.

LISCHINSKI, D., TAMPIERI, F., AND GREENBERG, D. P. 1992. Discontinuity Meshing for Accurate Radiosity. *IEEE Computer Graphics and Applications 12*, 6, 25–39.

MOORE, R. E. 1979. *Methods and Applications of Interval Analysis*. Studies in Applied Mathematics (SIAM), Philadelphia.

PARKER, S., MARTIN, W., SLOAN, P.-P., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, 119–126.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: Surface elements as rendering primitives. In *SIGGRAPH '00*, 335–342.

PIGHIN, F., LISCHINSKI, D., AND SALESIN, D. 1997. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In *8th Eurographics Workshop on Rendering*, 115–126.

RUSHMEIER, H., BERNARDINI, F., MITTLEMAN, J., AND TAUBIN, G. 1998. Acquiring input for rendering at appropriate level of detail: Digitizing a pieta. In *9th Eurographics Workshop on Rendering*, 81–92.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH '00*, 343–352.

SANDER, P. V., GORTLER, S. J., HOPPE, H., AND SNYDER, J. 2000. Silhouette clipping. In *SIGGRAPH '00*, 327–334.

SANDER, P. V., GORTLER, S. J., HOPPE, H., AND SNYDER, J. 2001. Discontinuity edge overdraw. In *Symposium on Interactive 3D Graphics*, 167–174.

SIMMONS, M., AND SÉQUIN, C. H. 2000. Tapestry: A dynamic mesh-based display representation for interactive rendering. In *11th Eurographics Workshop on Rendering*, 329–340.

TOLE, P., PELLACINI, F., WALTER, B., AND GREENBERG, D. 2002. Interactive global illumination. In *SIGGRAPH '02*, 537–546.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. In *Proc. of Eurographics*, 153–164.

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive Global Illumination using Fast Ray Tracing. In *13th Eurographics Workshop on Rendering*, 15–24.

WALTER, B., DRETTAKIS, G., AND PARKER, S. 1999. Interactive rendering using the Render Cache. In *10th Eurographics Workshop on Rendering*, 19–30.

WALTER, B., DRETTAKIS, G., AND GREENBERG, D. 2002. Enhancing and optimizing the Render Cache. In *13th Eurographics Workshop on Rendering*, 37–42.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRAER, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *SIGGRAPH '01*, 361–370.

WANGER, L., FERWERDA, J., AND GREENBERG, D. 1992. Perceiving spatial relationships in computer-generated images. *IEEE Computer Graphics and Applications 12*, 3, 44–58.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A Ray Tracing Solution for Diffuse Interreflection. In *SIGGRAPH '88*, 85–92.

WARD, G. J. 1994. Adaptive shadow testing for ray tracing. In *5th Eurographics Workshop on Rendering*, 11–20.

YOO, K.-H., KIM, D. S., SHIN, S. Y., AND CHWA, K.-Y. 1998. Linear-time algorithms for finding the shadow volumes from a convex area light source. *Algorithmica 20*, 3, 227–241.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *SIGGRAPH '01*, 371–378.

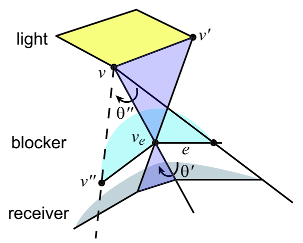


Figure 14: Finding an umbral EV event.

A Umbral and Penumbral Events

We extend an efficient algorithm for finding penumbral events [Yoo et al. 1998] to support meshes, concave surfaces, and umbral events. Our algorithm efficiently finds all shadow events that participate in the shadow volume boundary. We follow [Lischinski et al. 1992] in using the term "VE event" to refer to a vertex–edge event whose vertex lies on the light; an "EV event" is correspondingly defined by an edge of the light and a vertex of the blocker.

VE events. To identify VE events involving a particular edge e of the blocker and a vertex v of the light, the silhouette test of Equation 1 is first performed to ensure that the normal of the plane defined by the VE event is included in the normals represented by the edge. The vertices of the light adjacent to v are tested to determine whether they all lie on the same side of the plane; if so, an event has been found. To determine whether the event is umbral or penumbral, the orientation of the blocker surface is compared to the position of these adjacent light vertices.

EV events. EV events can be found efficiently by piggybacking onto the discovery of VE events, thus avoiding a separate tree traversal. Once a VE event connecting vertex v and edge e is found, the shadow volume boundary is traversed in both directions (along e) looking for adjacent EV events. This traversal is simplified using the following observation: for surfaces that appear locally convex from the viewpoint of the vertex v, EV events contribute to the penumbral shadow volume boundary, but not to the umbral shadow volume boundary. Conversely, for locally concave surfaces, EV events contribute to the umbral boundary but not to the penumbral boundary.

Figure 14 illustrates the algorithm used to find EV events in the case of a concave umbral event. Here, the shadow volume boundary is being traversed towards the vertex v_e along the blocker surface. The first step is to determine whether the surface is locally convex or concave. This is done by finding the angle θ'' between the v–e plane and the plane defined by the three vertices: v, v_e, and the blocker vertex v'' adjacent to v_e that minimizes this angle. If the angle is less than π, the blocker surface is locally concave from the viewpoint of v.

Given this geometry, there are two possibilities for the next event along the shadow volume boundary: it is either the EV event shaded in purple, defined by the vertices v, v' and v_e, or else it is the VE plane defined by v, v'', and v_e. Which is correct is determined by computing the angle θ' between the v–e plane and the possible EV plane; if this angle is larger than θ'', the EV event is part of the shadow volume boundary.

Penumbral EV events are found similarly, except that they are only found for convex surfaces, and the angles involved are greater than π.