Program of Computer Graphics Technical Report: PCG-02-3, January 2002

Combining Edges and Points for Interactive Anti-Aliased Rendering

Kavita Bala   Bruce Walter   Donald Greenberg [*]

Program of Computer Graphics
Cornell University
January, 2002

Abstract

This paper presents a new rendering and display paradigm that uses both discontinuities (edges) and sparsely sampled shading (points) to interactively generate anti-aliased images of complex scenes. Geometric and shadow discontinuities in the image are found at interactive rates using a novel data structure, the *Normal–Position Interval* tree, and algorithms based on interval arithmetic. After projecting discontinuities onto the image plane, shading information is interpolated from nearby point samples while never interpolating across a discontinuity edge. We introduce a new compact representation of the discontinuities and point samples called the *edge-and-point* image. An efficient interpolation algorithm uses this image to generate anti-aliased output images at interactive rates without using supersampling.

Our rendering technique is extensible, permitting the use of arbitrary shaders to collect radiance samples. Our software implementation supports interactive navigation and object manipulation in scenes that include complex lighting effects (such as global illumination) and geometrically complex objects. We show that high-quality anti-aliased images of these scenes can be rendered at several frames per second on typical desktop machines.

1 Introduction

Scalable rendering algorithms are needed for the high-quality interactive rendering of increasingly complex scenes. Polygon-based rendering scales poorly with scene and lighting complexity [15, 18, 19]. This paper introduces a scalable, interactive rendering technique that uses both radiance discontinuities and sparse samples to generate high-quality, anti-aliased images.

Sparse sampling of radiance is essential for interactive rendering when including shading effect such as accurate shadows, non-diffuse shading, and global illumination. In the absence of radiance discontinuities, such as object silhouettes and shadow boundaries, radiance can be efficiently approximated by interpolating among sparsely distributed samples in the image plane. However, discontinuities are perceptually important and expensive to find via sampling alone. The new rendering approach described in this paper directly finds important discontinuities (edges) such as those caused by geometric and shadow boundaries. These edges are projected onto the image, and interpolation is never performed between two samples that lie on opposite sides of a discontinuity edge. The explicit representation of discontinuities permits fast anti-aliasing without supersampling.

Currently the user can navigate and manipulate objects in complex scenes while obtaining high-quality rendering feedback at interactive rates (4–6 frames per second on a single desktop machine). Our system collects samples for only about 2 to 6% of the image pixels for any given frame, but samples are reused from frame to frame.

[*]Email: {kb, bjw, dpg}@graphics.cornell.edu.

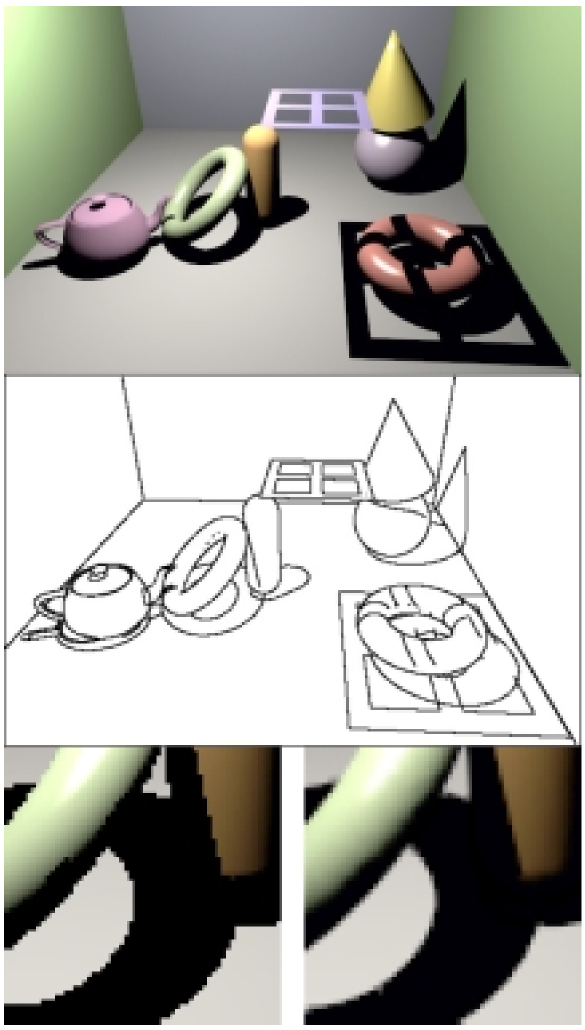
Figure 1: Example room with 150,000 polygons. The top image is an example output image from our system. The middle image shows the discontinuity edges found by our system. The bottom row compares a magnified single-sample-per-pixel image on the left and our anti-aliased output on the right.

Edge detection. The two most important shading discontinuities in images are: geometric discontinuities caused by changes in the visible surface, and shadow discontinuities caused by changes in the illumination of a visible surface. The silhouettes of objects correspond to the important geometric discontinuities. Hard shadow edges from point light sources, and soft shadow edges from area light sources are the important shadow discontinuities (See Figure 2).

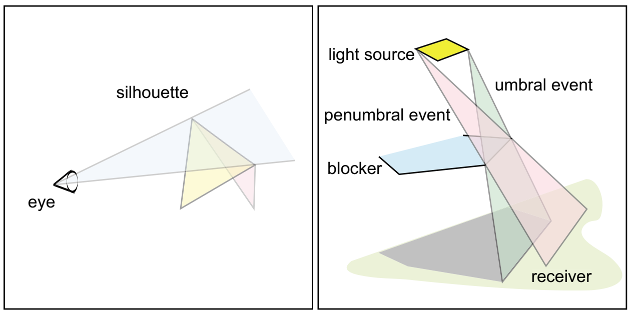

Figure 2: Sources of silhouette and shadow edges.

This paper introduces a hierarchical data structure called the *Normal–Position Interval Tree* and associated algorithms for efficiently finding these discontinuities in complex scenes. These discontinuity edges are directly projected onto the image plane and used to constrain interpolation of radiance samples.

Explicitly finding image discontinuities offers significant performance advantages. Systems that simply interpolate point samples must use adaptive point sampling around discontinuities to locate them. Therefore, these systems expend a large fraction of their computational effort sampling densely near discontinuities [8].

Reconstruction. We also introduce a novel display representation called the *edge-and-point image* that compactly stores *both* discontinuity edges and point samples. Edges are recorded using a discretized representation including subpixel information for anti-aliasing. This representation permits fast, simple, table-driven interpolation and anti-aliasing. In addition, the reconstruction algorithm is simple enough that it could be implemented in future hardware.

Our system is flexible and can work with many different types of shaders. We show results for various shaders that support hard and soft shadows and global illumination effects.

The remainder of this paper is organized as follows: Section 2 presents an overview of the rendering system. Section 3 discusses related work. Section 4 presents the data structures and algorithms for finding silhouette and shadow edges. Section 5 describes how the final anti-aliased image is produced by sampling and reconstructing radiance. Section 6 presents results and Section 7 concludes and describes future work.

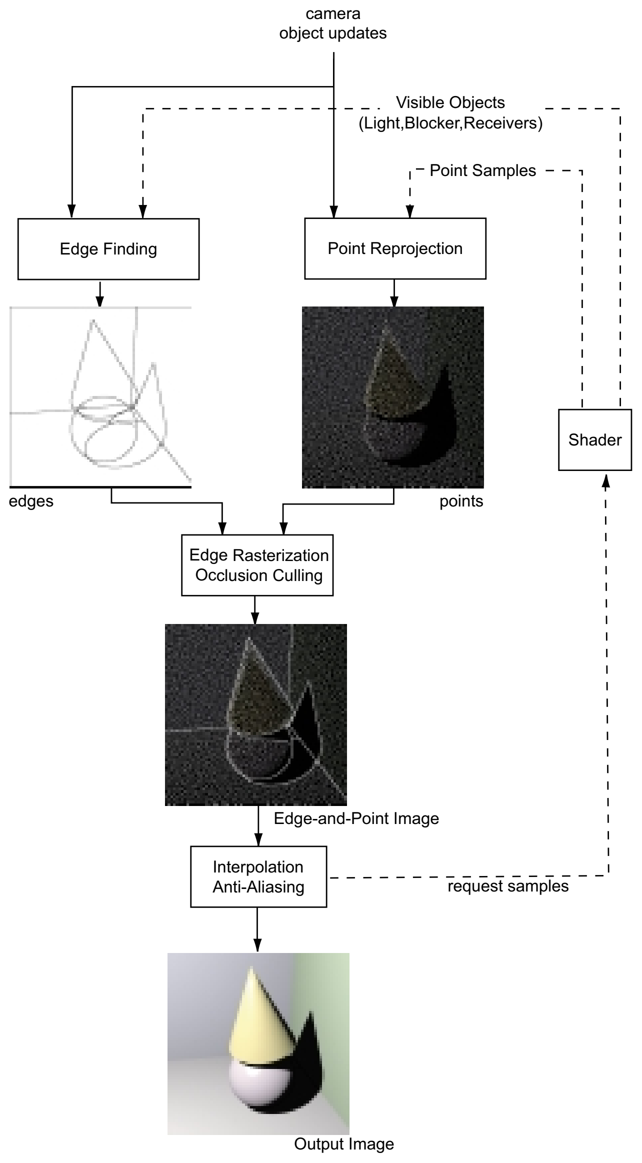

Figure 3: System Overview.

2 System Overview

An overview of our system is shown in Figure 3. The system consists of our edge-and-point renderer and an external shader process that asynchronously computes point samples. As the user changes the viewpoint or moves objects in the scene, these updates are passed to the edge-and-point renderer.

The edge finder finds silhouettes and shadow edges for each visible object in the scene. The silhouette edges of objects are view-dependent and are recomputed each frame. Shadow edges, which are view-independent, are computed once and reused except when the light, blocker or receiver associated with the shadow edge moves. Shadow edges are computed in two steps: first, shadow events are found by finding the silhouette of the blocker from the light's point of view for point lights or by finding umbral and penumbral events for area lights. These shadow events are then traced on receivers to find the associated shadow edges. The hierarchical construction of our trees permits the efficient computation of these edges even for complex objects. The computed 3D silhouette and shadow edges are then passed to the edge rasterizer.

The point reprojection module [26] reprojects the cache of recent point samples onto the image plane. The rasterizer rasterizes the 3D edges while also recording sub-pixel positioning information. In addition, depth information from the point samples is used to perform conservative occlusion culling on the edges. The result is the *edge-and-point image* which stores at most one edge and one point sample per pixel.

An interpolation method fills gaps in the point data while respecting the recorded discontinuity edges. Feedback from the interpolation is used to decide where new point samples are needed and these requests are passed to the external shader. Finally, a simple filter is used to anti-alias the pixels containing edges to produce the output image.

The shader asynchronously computes samples that are added to the point cache. The shader also communicates information about which objects are visible in a frame and which lights and blockers cause shadows on each visible receiver. Our system is flexible, decoupling the shader process from the discontinuity-driven sampling and reconstruction process. Thus, an arbitrary shader of choice can be used to compute samples and visibility information.

3 Related Work

Complexity. Recent efforts in 3D scanning have created massive data sets [12, 17]. Approaches such as Surfels and Qsplat [15, 18] suggest an alternative point-based representation for the interactive display of these massively large data sets. Because they use precomputed sampling these techniques are mainly useful for the display of static scenes. The randomized z-buffer [27] does not suffer from these limitations. However, these approaches do not track discontinuities or support illumination effects such as shadows and global illumination.

Level-of-detail algorithms, such as view-dependent progressive meshing [10], address the problem of scalable viewing for the perceptually-important silhouettes of complex objects. Silhouette clipping [19] is a recent approach for the interactive viewing of complex stand-alone objects; it differs from point-based approaches in handling objects with distinct silhouettes that would not be effectively represented by point-based approaches. The silhouette clipping approach introduces hierarchical techniques to rapidly find silhouettes of these complex objects for interactive viewing. A further extension of the system uses hardware to anti-alias silhouette edges [20]. However, these systems require large preprocessing times and again assume that the object being viewed is static. They also do not handle complex shading effects such as shadows and global illumination.

Our Normal–Position Interval Tree has similarities to the cone hierarchy used for silhouette clipping with some important differences: our tree includes the spatial extent of the geometry as well as the normals, making it possible to compute umbral and penumbral events. The construction time of our trees for similarly sized scenes is on the order of seconds rather than the hours reported in [19]. Silhouette, umbral and penumbral tests are simplified by using interval arithmetic.

The spatialized normal cone hierarchy [11] expands the silhouette clipping data structure for applications such as local minimum distance computation, and shadow event computation. Our tree structure is similar to the spatialized normal cone hierarchy, though it differs in its use of intervals to represent the range of normals and positions associated with an object.

Sparse sampling and reconstruction. Several researchers have tried to exploit adaptive sampling to sparsely sample regions with smoothly varying radiance. Guo proposed a progressive technique that samples the image plane and tries to detect the existence of discontinuities [8]. Once a discontinuity is hypothesized, this system samples along the discontinuity and interpolates radiance using these samples. This system is most effective in generating high-resolution still images.

The radiance interpolant system [2] samples radiance in 4D ray space and uses conservative error analysis techniques to identify where more samples are required. This system subdivides ray space around discontinuities and interpolates radiance where it varies smoothly. Both Guo's approach and the radiance interpolant system use sampling to detect where discontinuities arise and therefore spend significant computational resources sampling around discontinuities to locate them.

Pighin et al. [16] present a progressive previewing technique that uses hardware to detect visibility and hard shadow edges in static scenes. A constrained Delaunay triangulation of sparse samples is used to reconstruct images. Their system assumes static scenes lit by point lights. Tapestry [21] sparsely samples the scene and meshes the scene, also using a Delaunay triangulation. This mesh is not in image space and therefore, can be used as the viewpoint changes. However, this system does not try to detect discontinuities, thus producing results that could have significant blurring near such edges.

The RenderCache [26] reuses and reprojects pixels from frame to frame to achieve interactive performance. However, since discontinuity events are not available to constrain the reprojection, the resulting images are blurred near edges. We have adapted the RenderCache for the point-based part of our algorithm.

Discontinuity meshing. The literature on finding discontinuities in a scene is extensive; a summary of visibility and shadow detection research can be found in [4]. Several approaches have tried to find all visibility events in scenes [3, 5, 9, 13, 24, 25].

Discontinuity meshing [9, 13] determines where radiosity varies discontinuously and tesselates scene geometry based on these discontinuities. Discontinuity meshing techniques avoid the cost of finding discontinuities using adaptive sampling. However, because they remesh geometry around the discontinuities, they usually create excessively large meshes to account for all discontinuities in typical scenes. The enumeration of all discontinuities is also typically too slow for interactive use in complex scenes because these algorithms exhaustively find all shadow events for all potentially interacting polygons.

Our approach is to achieve interactive performance by focusing on the shadow discontinuities that are typically the most important perceptually: those corresponding to hard shadows and umbral and penumbral vertex–edge events. We describe how to find these events efficiently in high-complexity scenes using Normal–Position Interval Trees.

Approximate soft shadows. Soler and Sillion [23] make simplifying assumptions about the blocker, receiver configuration to approximate soft shadows. However, their approach does not capture important effects such as the "hardening on contact"[28] that arises when a blocker and receiver are in contact.

4 Finding Edges

The first step in rendering an image is to rapidly find silhouettes and shadow edges for all visible objects. Each frame, the edge-and-point renderer uses the camera position and a list of visible objects to find the silhouette and shadow edges for the visible objects. Silhouette edges are view-dependent and are recomputed for each frame. Shadow edges are view-independent and are reused from frame to frame when possible. This section describes the data structures and algorithms for finding discontinuities efficiently.

4.1 Background

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 2. Thus, the test for the existence of a silhouette is the following:

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

Here, f_0 and f_1 are the two polygons adjacent to the edge e, 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 $\mathbf{E}$.

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*, which when intersected with the receivers cause shadow discontinuities. In polygonal environments with area lights, two types of shadow events cause shadow discontinuities [6, 7, 9, 13]: vertex–edge events, and edge–edge–edge events. Point lights are a degenerate case of area lights and only produce vertex–edge events. Vertex–edge events are planes (actually, wedges) defined by a vertex of the light and an edge of the blocker, or vice-versa. We follow Lischinski et al. [13] 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. Edge–edge–edge (EEE) events have an edge on the light, and two edges from two different blockers. In this paper, we restrict ourselves to finding VE and EV events, which are more common than EEE events and result in visually more important discontinuities.

Umbral and penumbral events are shown in Figure 2 on the right. An umbral or penumbral event occurs 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. For point lights, the umbral and penumbral events coincide. Collectively, the wedges of the shadow events bound *shadow volumes*: regions of space in which the visibility of the light (blocked, partially blocked, or unblocked) is the same. Shadow edges occur where the edge of a shadow volume intersects a receiver object.

Note that for objects composed of polygons, each vertex and edge of the light and blocker is associated with a *set* of normals; this set includes all normals that can be found by linearly interpolating between normals of the adjacent faces. A plane is considered tangential if its normal is one of those associated with the contact vertex or edge.

4.2 Normal–Position Interval Trees

Visibility and shadow edge detection are accelerated using a hierarchical data structure that we call the Normal–Position Interval Tree (NPIT). There are two such data structures associated with each object. The *edge tree* is a hierarchical representation of all the edges of the object, and is used to accelerate silhouette detection and shadow event computation. The *face tree* is a hierarchical representation of all the faces of the object, and is used to accelerate the computation of shadow edges.

Each node of the tree uses intervals to conservatively record the range of normals and positions of the elements (edges or faces) represented by that node. The range of normals is represented by three intervals: one for each of the three Cartesian components of the normal. The subscript i is used to denote an interval, so the normals are represented by a triple of intervals (N_i^x, N_i^y, N_i^z), which we call an *interval vector*. The interval vector $([x_0, x_1], [y_0, y_1], [z_0, z_1])$ represents all vectors (x, y, z) such that $x \in [x_0, x_1]$, $y \in [y_0, y_1]$, and $z \in [z_0, z_1]$. Similarly, the range of positions associated with the tree node is represented by an interval vector. The normal and position interval vectors are used during hierarchical traversals of the tree.

The edge and face trees are each constructed as follows: at each node of the tree, the tree is divided on one of the three axes (x, y, z) so that the combined size of its children is minimized. The spatial extent of a tree node's children need not be disjoint; an edge (or face) is always put in only one of the two children. Tree nodes are not subdivided below a threshold number of elements; performance is not sensitive to this setting. This construction has low preprocessing overhead (on the order of seconds) and seems to be as effective as a more specialized hierarchical construction [19] taking much longer (hours) to construct.

Note that the face tree is essentially a bounding volume hierarchy that is used to accelerate the intersection of shadow events with object faces. However, the intervals for normals are used to further accelerate this process by culling back-facing triangles.

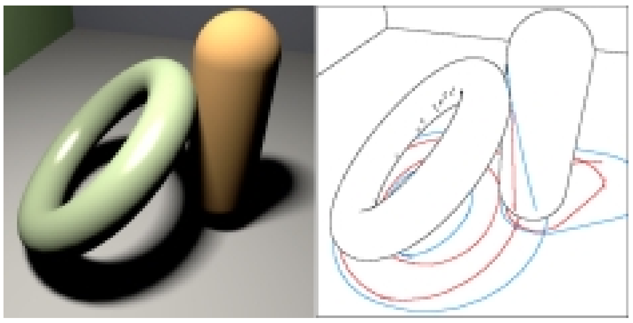

Figure 4: Umbral and Penumbral edges: umbral edges in red, penumbral edges in blue.

4.3 Silhouette Detection

For each frame, the edge tree associated with each visible object is recursively traversed to find all silhouette edges of the object with respect to the current viewpoint **E**. Starting at the root, each node of the tree is tested for potential silhouette edges. If the edges covered by a node of the tree are all forward-facing or backward-facing, the recursive walk is terminated. When a leaf node is reached, all its edges are tested using Equation 1 to determine if they are on the object silhouette.

For internal (non-leaf) nodes, the normal and position interval vectors are used to determine the possible presence of silhouettes in the subtree as follows. The view vector is conservatively represented by the interval vector $\mathbf{V}_i = \mathbf{P}_i - \mathbf{E}$, where $\mathbf{P}_i$ is the position interval vector representing all the edges in that node of the tree. The dot product of the normal interval vector $\mathbf{N}_i$ and the view interval vector is computed using interval arithmetic [14, 22], resulting in an interval $[q_0, q_1]$:

$$\begin{aligned} [q_0, q_1] &= \mathbf{N}_i \cdot_i \mathbf{V}_i \\ &= N_i^x *_i V_i^x +_i N_i^y *_i V_i^y +_i N_i^z *_i V_i^z \end{aligned}$$

where $*_i$ and $+_i$ are the usual interval arithmetic operators. If an edge contained in the subtree is on the silhouette, corresponding to a surface normal and view vector whose dot product is zero, then the resulting interval $[q_0, q_1]$ must include zero. In this case, the subtree is recursively explored.

4.4 Shadow Event Detection

Shadow discontinuities are more complex than silhouette discontinuities and are computed in two steps:

- The shadow events produced by lights and blockers are found.
- The shadow events are intersected with receivers to find shadow edges.

The blocker's edge tree is used for the shadow event computation and the receiver's face tree is used for the intersection of the events with the receiver to find shadow edges.

The hard shadows caused by the occlusion of a point light source by a blocker create perceptually important D_0 radiance discontinuities. They can be found relatively simply by computing the blocker's silhouette from the point of view of the light. These shadow events are then used to compute the shadow edges that appear on receiver objects, as described in Section 4.5.

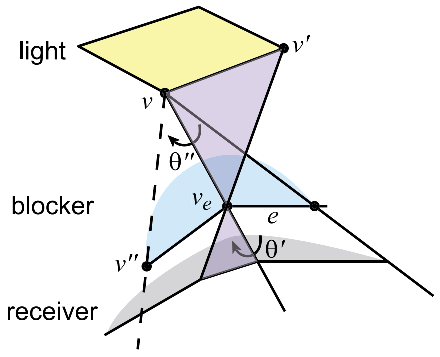

Figure 5: Finding an umbral EV event.

Soft shadows arise from the occlusion of an area light source and generate D_1 discontinuities. It is particularly important to identify soft shadow edges near the point where a blocker contacts a receiver. At that point, soft shadows exhibit a perceptually important "hardening" effect in which the radiance gradient increases. Identifying penumbral and umbral shadow boundaries allows this effect to be rendered correctly, as shown in Figure 4.

For area lights, our algorithm finds the penumbral and umbral VE and EV events using a recursive traversal of the blocker's edge tree. At each node, the light vector is conservatively characterized by the interval vector $\mathbf{P}_i -_i \mathbf{L}_i$, where $\mathbf{P}_i$ is the position interval vector for the blocker, $\mathbf{L}_i$ is the position interval vector for the light, and $-_i$ is the interval vector subtraction operator. Using a position interval vector for the light is important because area lights cover a range of positions and thus introduce variability in the light vector. When internal nodes are reached during the traversal, the algorithm checks whether the surface normal vector might be perpendicular to the light vector within that node. This is conservatively determined by testing whether the dot product, computed using interval arithmetic, contains zero.

4.4.1 VE events

At the leaves of the tree, a more precise test is performed to identify shadow events. Recall that each VE event defines a plane. 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 is one of the normals of 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.

4.4.2 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 locally convex surfaces, EV events contribute to the penumbral shadow volume boundary, but not to the umbral shadow volume boundary. Conversely, for locally concave surfaces (from the viewpoint of the vertex v), EV events contribute to the umbral boundary but not to the penumbral boundary.

Figure 5 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. Our algorithm extends an earlier efficient algorithm for finding penumbral events [30] to support meshes, concave surfaces, and umbral events. The algorithm efficiently finds all shadow events that participate in the shadow volume boundary, while largely avoiding the generation of unnecessary edges that would slow down the later rendering stages.

4.5 Finding Shadow Edges

Given the VE and EV events, shadow edges are computed by intersecting the event wedges with the receiver geometry. This process is accelerated by using the receiver's face tree. Each shadow event is walked down the tree to find intersections of the event with the geometry represented by that node of the tree. At internal nodes of the face tree, three tests are used to avoid unnecessary traversals. The first test is to see whether the event intersects the node at all, which requires that the positions represented by the position interval vector lie on both sides of the event. Given that $\mathbf{N}_{ve}$ is the normal of the event plane, the following interval is computed:

$$[q_0, q_1] = (\mathbf{P}_i - v) \cdot_i \mathbf{N}_{ve}$$

If any of the faces represented by the node lie on both sides of the plane, the resulting interval $[q_0, q_1]$ must include zero. If so, the algorithm recurses to the children of the tree node.

The second test uses the normal interval vector stored with the node to check whether all the faces associated with that node are backward-facing with respect to the light. If so, then a shadow need not be cast on them and the search space is pruned. The third test uses planes that are perpendicular to the event plane and contain the rays bounding the event wedge to eliminate nodes whose faces all lie fully on the wrong side of these planes.

At the leaf node of the face tree, all the polygons stored in the leaf are tested against the candidate shadow event; if an intersection is found, a shadow edge is generated.

4.6 Discussion

Our approach is reminiscent of discontinuity meshing [13] but has important advantages: since we do not mesh discontinuities—we use them only to constrain interpolation—there is no resulting explosion of small mesh elements. In addition, our hierarchical data structures and algorithms permit the efficient computation of all shadow events even for blockers of high polygon count; this is essential for the rendering of complex scenes at interactive rates.

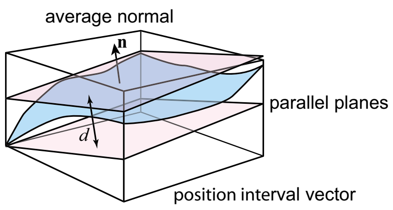

Figure 6: Planes approximating edges.

4.7 Features and Implementation Issues

There are a few additional features and implementation details that are important to interactive detection of silhouettes and shadow edges.

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 sampling to the edge finder. When a particular receiver r is visible in a frame, the shadow edges cast onto r by all the lights and blockers that are known to have r as a receiver are found. These shadow edges are stored with r and reused for other frames as long as the light, blocker and receiver do not change.

For a given light-blocker pair, all VE and EV events associated with the pair are computed. These shadow events are then intersected with every receiver associated with that pair to find the receiver's shadow edges. However, some shadow edges may be occluded by receivers nearer to the light, and thus need not be included in the edge-and-point image. A shadow edge is removed from the output if the entire extent of its generating shadow event casts shadow edges on a different receiver that is closer to the blocker. This optimization removes many unnecessary edges.

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, but shadows on other receivers that share a light-blocker pair with that receiver might also be affected because they are no longer occluded by the moved object. To be conservative, all these shadows are recomputed. For example, if the torus next to the teapot in Figure 1 is moved, the shadows cast by the teapot on *both* the torus and the floor are recomputed. Thus, shadow events that cast shadows of the teapot on the torus will correctly appear on the floor when the torus moves out of the way.

To maintain interactive performance, when the user moves an object, queries into the NPIT data structure are transformed appropriately to avoid rebuilding these data structures. For silhouettes, the viewpoint is transformed into the object's coordinate system and the silhouette edges found. For shadows, the light is transformed into the blocker's coordinate system. The resulting shadow events are transformed to the receiver's coordinate system (if necessary) to find shadow edges. These silhouette and shadow edges are transformed back into world space for display.

Edge approximation. When an object projects to a small part of the image, finding all its silhouette edges and rasterizing them may be too expensive. If, during the computation of silhouette edges, it is discovered that an entire edge tree node projects to a small screen area, the system computes an approximation of the silhouette edges occurring within that node.

To make this approximation closely match the true silhouette, the system creates a rotated coordinate system for each tree node (see Figure 6); the z axis in this coordinate system is the average normal of all contained edges. In the figure, the two parallel planes perpendicular to the average normal capture the range of positions occupied by the edges in that node of the tree. The advantage of the new coordinate system is that the separation d between these two planes is usually much smaller than the dimensions of the original node. When a possible silhouette is detected within the node, the two planes are being viewed nearly edge-on. If they are separated by less than one pixel when projected onto the image plane, the recursive traversal is terminated and the planes themselves are used to generate the silhouette edge.

Sharp edges. When adjacent faces of an object have a large variation in their normal—for example, the faces of a cube—our system considers that their shared edge is intended to be sharp and to display a shading discontinuity. When the NPIT is computed, sharp edges are flagged and included in the edge-and-point image if they are forward-facing. Shadow edge computation is not affected by this consideration.

Chains. It is useful for the rasterizer to receive edges stitched together into connected chains of many edges, in order to support incremental rasterization and accurate subpixel information where a sequence of edges cross a pixel. We use simple hash tables to construct chains of silhouette and shadow edges. Care is taken to compute the shadow edges on receivers in a consistent manner that avoids floating point inconsistencies.

5 Rasterization and Interpolation

Once a set of 3D discontinuity edges are identified, the next tasks are to project them onto the image plane, efficiently combine them with the point samples, and rapidly generate images using an edge-respecting interpolation scheme.

We have chosen to adapt the RenderCache algorithm [26], very briefly summarized here, for our point sample processing. Point samples are produced by an independent shading system that need only be capable of computing the color and first intersection for a viewing ray. The RenderCache caches recent samples as colored points in 3D using a fixed-size cache. 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. The RenderCache also prioritizes future sampling to concentrate effort in image regions with sparse or stale data. To keep the framerate interactive, it restricts the maximum effective sample density to one sample point per pixel and uses relatively small 3×3 pixel filter kernels.

We use the reprojection and sampling components of the RenderCache essentially unchanged, but we have adapted its image filters, specifically the depth (occlusion) cull and interpolation operations, to respect discontinuity edges. To maintain an interactive framerate though, these operations must be fast. We have also added a new anti-aliasing algorithm, implemented as a filter, after the interpolation step.

5.1 Edge Rasterization

After the discontinuity edges have been identified, the system projects them onto the image plane and converts them to a convenient form for use in interpolation. The simplest approach would be to rasterize the edges onto a bitmap, but we want to preserve some subpixel information for anti-aliasing purposes. As the edges are rasterized, the discretized subpixel locations where they cross pixel boundaries are recorded as shown in Figure 7. Within a pixel, an edge is approximated as a straight line that starts and ends on the pixel boundary.

This approach has the advantages of being simple, compact, and easy to compute with. The disadvantages are that only one edge can be recorded per pixel and pixels with sharp edge corners are

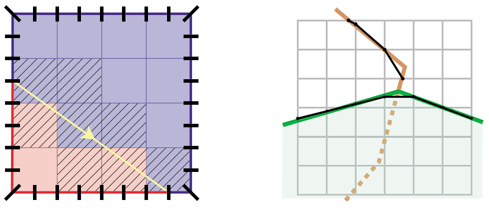

Figure 7: A magnified view of a pixel (left) shows that its boundary is discretized into 32 regions and its interior into 16 regions. An edge (yellow) divides these regions into those inside (red) and outside (blue) the edge. Some interior regions (cross-hatched) are also classified as ambiguous with respect to this edge. On the right, is an edge rasterization example for a 6×6 pixel region. Shown is a silhouette edge for a foreground object (green) and a shadow edge on a background object (brown). Where they compete for the same pixel, the z-buffer ensures that the foreground edge wins (black). Depth information from the points allows the hidden portion of the background edge to be culled.

not accurately recorded. This is acceptable because we are targeting interactive applications where speed is more important than the accurate resolution of more difficult, and hopefully rare, cases. Our results also validate this approximation.

Pixel boundaries are discretized into 32 locations, allowing the encoding of each pixel's edge using 10 bits (5 each for start and end). This is enough subpixel precision for good anti-aliasing and small enough that table lookups can be used for many edge-related operations. Pixels with no edge have both their start and end points set to zero.

During edge rasterization a z-buffer ensures that when multiple edges compete for a pixel, the frontmost edge wins. The z-buffer from the point reprojection step is reused to cull most of the occluded edges. But first, a small 3×3 filter is used to fill in small gaps in the point's z-buffer. Points and edges will not exactly coincide even when they map to the same pixel, so we use a fairly conservative z-offset when drawing edges to ensure they are not falsely occluded. This z-offset is dependent on the image resolution, camera parameters, and, for shadow edges, the surface normal.

5.2 Reachability Map

Once the edges have been rasterized, we can compute reachability. A sample is considered *reachable* if there exists a fairly direct path to it from the point or region of interest that does not cross any edge segments. It is more efficient to first compute reachability in a separate pass for use by the later depth cull and interpolation steps.

An edge divides a pixel into two regions: inside and outside of the edge,[1] as shown in Figure 7. For anti-aliasing, we potentially need to interpolate two different values for each pixel, however we have chosen to reconstruct only one value per pixel and correct for the missing values in a later filter.

During reprojection, we record which of 16 subpixel regions that sample mapped to. However, if an edge crosses through a subpixel region then this is not enough information to reliably determine on which side of the edge the sample belongs. For each pixel, we use a lookup table to get a 16 bit mask recording which subpixel regions are ambiguous and then invalidate ambiguous samples.

For each pixel, we need to decide whether to reconstruct its inside or outside value. If a sample point mapped to this pixel, then we use whichever region the sample point fell into. Otherwise we choose an arbitrary interior region and use its inside/outside classification. Using these definitions reachability becomes symmetric. If pixel a is reachable from pixel b then b is also reachable from a. Thus we need only compute and store the reachability of half of a pixel's neighbors, since we can look up the others using symmetry. For a 3×3 filter, we need only store four bits for the reachability of four of its neighbors, plus one bit saying whether we reconstructed the value for its inside or outside region.

[1]For pixels with no edge, we arbitrarily classify the entire pixel as inside.

A pixel's boundary is divided into 32 regions. Another table lookup is used to get a 32 bit mask encoding which portions of its boundary are reachable from its inside and/or outside region. By comparing boundary masks for the shared boundaries between neighboring pixels, we can rapidly determine which parts (inside and/or/neither outside) of the neighbors are reachable. We first propagate the reachability to a pixel's immediate neighbors and then from them to its diagonal neighbors. Once we know which regions of all the neighboring pixels are reachable, a lookup table allows a quick determination of whether any sample points they contain are reachable.

5.3 Image Filtering

The depth cull works by comparing each valid sample's z value to the average z value for all valid samples in its neighborhood. This step is necessary to reduce occlusion errors due to gaps in the reprojected points. It also has the unwanted side effect of eliminating valid samples that lie just outside of depth discontinuities (i.e., silhouette edges). However, because our edges include silhouette edges, this artifact can be avoided by averaging the only z values of samples that are reachable.

Similarly the interpolation must be constrained to use only reachable samples. This is straightforward; the interpolation already needs to check if each neighbor contains a valid sample. An additional lookup for reachability is sufficient to make the interpolation respect any recorded discontinuity edges. This reconstructs a color for every pixel which has a valid reachable sample in its neighborhood. Any other pixels retain their color from the previous frame and are given maximum priority for having a new sample computed when the sparse locations for new samples are selected.

Although our system normally uses 3×3 interpolation kernels, it is straightforward to extend our method for larger kernels. We have experimented with a 5×5 kernel which permits lower sampling densities, but incurs a higher filtering cost.

For pixels that contain an edge, we would like to anti-alias them by using a weighted combination of its inside and outside colors that reflects the relative areas of its inside and outside regions. Because only one color per pixel is interpolated, the missing color is approximated by looking up the color of a neighboring pixel. An edge-indexed table lookup is used to choose the neighbor most likely to contain a good approximation and the relative weights to use. We have found this filter to be cheap and effective.

6 Implementation and Results

In this section we present results of our edge-and-point based rendering system. Our system runs as a single thread on a dual Pentium IV 1.7 GHz machine. The edge-and-point renderer runs on one processor while the other processor is typically used to shade samples. When using expensive shaders, for example to include global illumination effects, the system can be configured to distribute the shader computation across multiple machines. We use a Java-based distributed ray tracing engine for the computation of shading, though any other renderer that could produce point samples would work as well.

6.1 Interactive Performance

6.1.1 Finding edges

We have run our edge-finding algorithm in scenes containing up to 150,000 polygons. The preprocessing time required for the construction of our NPIT data structures ranges from about a second to at most a minute for these scenes.

For the scenes shown in this paper our silhouette computation time ranged from 10 to 60 milliseconds for each frame. When the scene is loaded, initial shadow computation takes 0.25 to 2 seconds. As the user moves around or manipulates objects in later frames, the shadow computation time ranges from 10 to 100 milliseconds depending on how many new light-blocker-receiver tripes are found by the shader.

We compared our results to a system that did not use the hierarchical NPIT for silhouette and shadow computation: our silhouette algorithm is 8 to 30 times faster, while our shadow algorithm is 5 to 50 times faster. Scenes with objects of greater geometric complexity and lighting with soft shadows have the largest speedups.

6.1.2 Rasterization and Interpolation

Our experiments have confirmed that the cost for the point reprojection and the image filters including reachability, interpolation and anti-aliasing is essentially linear in the number of pixels. For a 512×512 image, these take a total of 120 milliseconds per frame and are usually the most expensive part of our system. These parts of our system have been written in C++ and optimized for Pentium IV processors and the SSE instruction set. The rest of the system is less speed critical and is written in Java.

In some scenes with many edges, edge rasterization becomes the bottleneck because we are using a simple rasterization algorithm written in Java. Adapting one of the many more efficient algorithms to produce the required subpixel information and porting it to C++ should eliminate this bottleneck. Currently our system produces frames at 4 to 6 frames per second for a variety of scenes at 512×512 resolution.

Our framerate is essentially independent of the speed of the shader being used, although the quality of the images degrades somewhat as the number of newly shaded samples per frame decreases (e.g., from using more expensive shaders such as soft shadows or global illumination). For the examples shown, our sampling ratio is typically between 16 and 50 times smaller than the number of pixels in our output images. Thus only 2 to 6% of the points samples are being updated each frame. This helps us achieve interactive performance even when using expensive shaders.

6.2 Test Scenes

6.2.1 Non-convex with Occlusion Culling

The chain data set in Figure 8 shows how effective the edge-and-point rendering algorithm is for rendering scenes with non-convex glossy objects. The scene is lit by two lights and has about 75,000 polygons. The edge image on the right shows the effectiveness of our occlusion culling algorithm. The edge algorithm finds a few extraneous edges due to some noise in the geometry of the torus; however, our interpolation algorithm is robust and handles these edges without introducing any visible artifacts.

6.2.2 Soft shadows

The room image shown in Figure 9 shows the ability of our system to handle scenes with non-convex objects casting soft shadows on each other. Some of the objects are in contact with each other as can be seen by the meeting of the penumbral and umbral shadow edges at these contact points.

6.2.3 Anti-aliasing

The Mackintosh room shown in Figures 10 and 11 demonstrates the ability of our algorithm 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; on the bottom is the result produced by our system.

The Alto library model shown in Figure 12 is a visually complex environment with 26,000 triangles and 6 light sources. It contains a high density of edges, especially in the ceiling slats, and in many places exceeds our limit of one representable edge per pixel. Nevertheless, our algorithm does quite well. On the right we show three magnified results for a section of the ceiling. From top to bottom these are: a conventional 1-sample-per-pixel image, a 4-sample-per-pixel supersampled image, and our output image. Although our algorithm is limited to a maximum of one sample per pixel, it achieves quality comparable to and sometimes better than the supersampled image. Moreover, because we use sparse sampling in both image space and time (only 2.5% of the points samples are updated per frame), our method produces much better interactive performance (and superior image quality) than the conventional method collecting only one sample per pixel.

6.2.4 Optimizing for Stills

Although we have concentrated on interactive applications, with a few modifications, our system could also be used for the rapid production of high-quality still images. In this case, there is no need to recompute existing point samples or check them for occlusion errors. And since framerate is not important, it would be desirable to use larger interpolation filters. As an initial exploration, we computed an image of Mackintosh room in Figure 11 using a larger 5×5 interpolation kernel. On the right we show two magnified versions both computed using the same point data. The upper image is blurry because it does not use discontinuity edges; the lower image generated using our method is much sharper. This scene also demonstrates our ability to work with different shaders as it includes dynamically computed global illumination using a variant of the irradiance caching scheme [29].

7 Conclusions

This paper introduces a scalable, interactive rendering technique that combines radiance 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. Point samples that are separated by these projected discontinuity edges are never interpolated together. We have demonstrated that our system renders high-quality anti-aliased images at interactive rates of several frames per second in scenes including geometrically complex objects and lighting effects such as shadows and global illumination. The user can also dynamically move objects within the environment.

This paper describes our novel interval-arithmetic based algorithms that use the *Normal–Position Interval Tree* to find silhouette and shadow edges at interactive rates in complex scenes. These data structures and algorithms are very effective in finding these edges and are 5 to 50 times faster than a brute-force algorithm.

We also present 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 detection of discontinuities permits the generation of anti-aliased output images without supersampling at interactive rates of 4 to 6 fps on a modern desktop computer. Our system collects samples for only about 2 to 6% of the image pixels for any given frame.

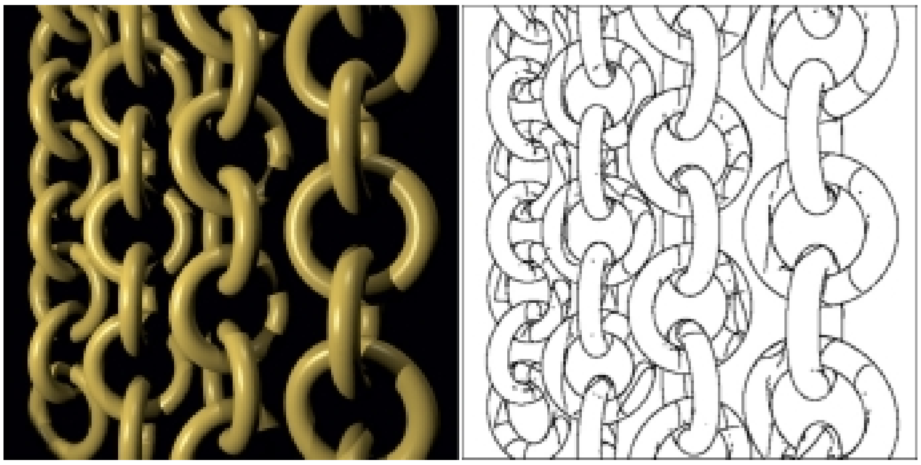

Figure 8: Chains scene inspired by Stuart Warmink.

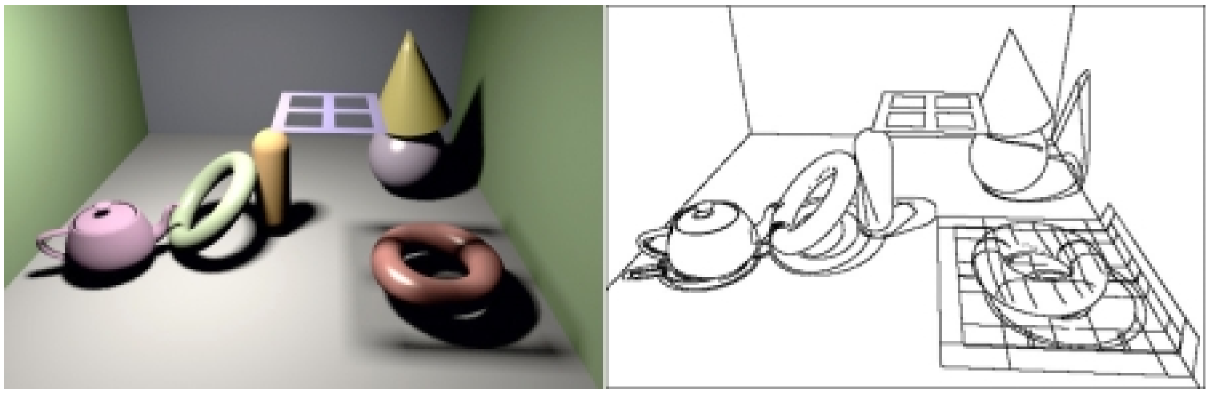

Figure 9: This room demonstrates soft shadows with contact hardening.

Although the speed of this new rendering technique is already acceptable, it is currently often limited by the speed of edge rasterization and sample interpolation. Thus, we envision implementing the rasterization of the edge-and-point image in future hardware, which should be quite feasible and also result in significant speedups. Because our system is compatible with the use of a variety of different shaders to collect point samples, this rasterization functionality would also be broadly applicable and is thus a promising new direction for graphics hardware support.

Future Work. Our system identifies two important types of discontinuities in an image as being perceptually important; other discontinuities such as reflections are also perceptually important and the hierarchical data structures and algorithms described in this paper could be extended to find them. To improve our support for interactive performance, we could add a cost-benefit model that identifies the most important discontinuities in an image given framerate constraints. For example, in scenes with large numbers of lights, perceptual metrics to find discontinuities for only the important lights would be useful. It would also be useful for the shader to use a more sophisticated technique to determine which samples must be invalidated when objects move [1]. We also expect that the shadow edge finding algorithms from our paper could be applied to shading in non-photorealistic rendering.

References

[1] 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.

[2] Kavita Bala, Julie Dorsey, and Seth Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3):213–256, July 1999.

[3] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24-29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 223–230. ACM SIGGRAPH, ACM Press, July 1994.

[4] Fredo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Grenoble University, July 1999.

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

[6] Ziv Gigus, John Canny, and Raimund Seidel. Efficiently computing and representing aspect graphics of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.

[7] Ziv Gigus and Jitendra Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2), February 1990.

[8] Baining Guo. Progressive radiance evaluation using directional coherence maps. In *Computer Graphics (SIGGRAPH 1998 Proceedings)*, pages 255–266, August 1998.

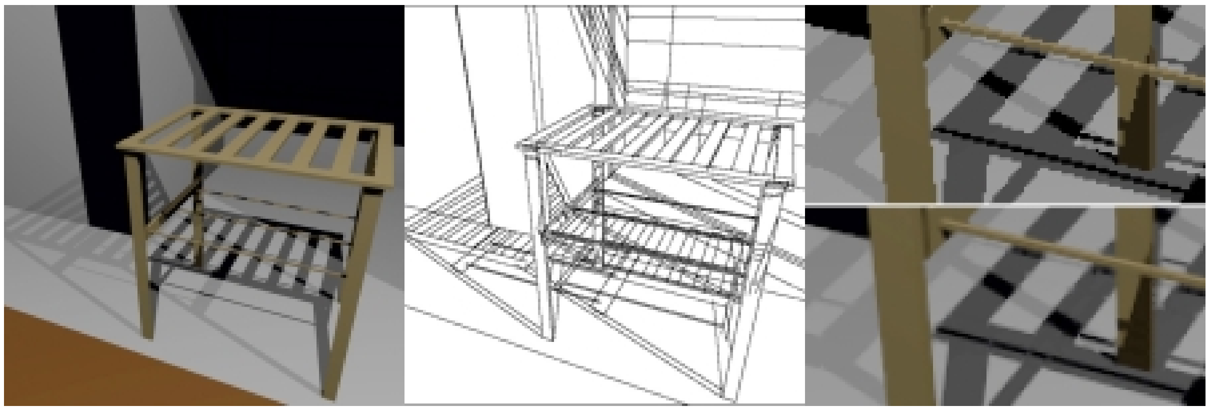

Figure 10: The left image shows a tea-stand with fine geometry and complex shadows. The middle image shows the edges found by our system. The images on the right compare a standard 1-sample-per-pixel sampling (top) against our results (bottom). Because the surface surrounding the fireplace is black, it reflects no illumination from the lights. Our system automatically detects this and does not compute shadow edges on this surface.

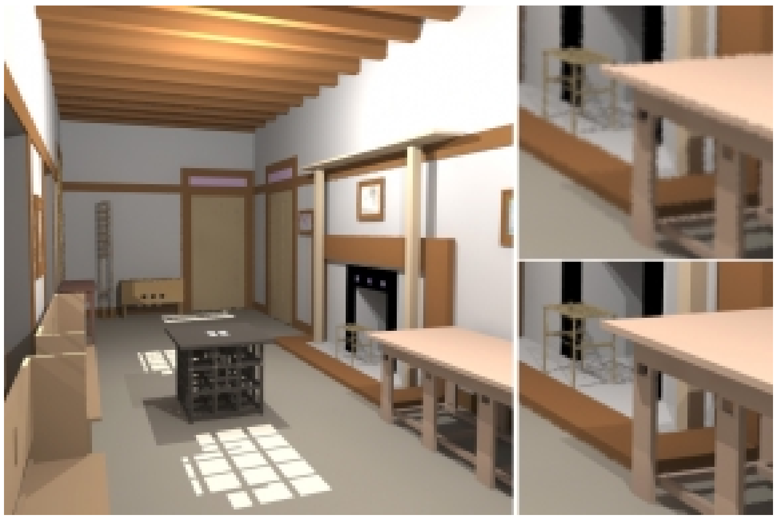

Figure 11: Mackintosh Room with 5$\times$5 interpolation filter. The images on the right show interpolation without discontinuities (top) and our algorithm with edges (bottom).

[9] Paul Heckbert. Discontinuity meshing for radiosity. *Third Eurographics Workshop on Rendering*, pages 203–226, May 1992.

[10] Hughes Hoppe. View-dependent refinement of progressive meshes. In *Computer Graphics (SIGGRAPH ’97 Proceedings)*, pages 189–198, 1997.

[11] David Johnson and Elaine Cohen. Spatialized normal cone hierarchies. In *ACM Symposium on Interactive 3D Graphics*, pages 129–134, March 2001.

[12] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Computer Graphics (SIGGRAPH ’00 Proceedings)*, August 2000.

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

[14] Ramon E. Moore. *Methods and Applications of Interval Analysis*. Studies in Applied Mathematics (SIAM), Philadelphia, 1979.

[15] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Computer Graphics (SIGGRAPH ’00 Proceedings)*, August 2000.

[16] F. Pighin, Dani Lischinski, and David Salesin. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In *Rendering Techniques 1997*, pages 115–126, June 1997.

[17] Holly Rushmeier, Fausto Bernardini, Joshua Mittleman, and Gabriel Taubin. Acquiring input for rendering at appropriate level of detail: Digitizing a pieta. In *Ninth Eurographics Workshop on Rendering*, pages 81–92, June 1998.

[18] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Computer Graphics (SIGGRAPH ’00 Proceedings)*, August 2000.

[19] P. V. Sander, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In *Computer Graphics (SIGGRAPH ’00 Proceedings)*, August 2000.

[20] P. V. Sander, S. J. Gortler, H. Hoppe, and J. Snyder. Discontinuity edge overdraw. In *ACM Symposium on Interactive 3D Graphics*, March 2001.

[21] Maryann Simmons and Carlo H. Squin. Tapestry: A dynamic mesh-based display representation for interactive rendering. In *Eleventh Eurographics Workshop on Rendering*, pages 329–340, 2000.

[22] John M. Snyder. Interval analysis for computer graphics. *Computer Graphics (ACM SIGGRAPH ’92 Proceedings)*, 26(4):121–130, July 1992.

[23] 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.

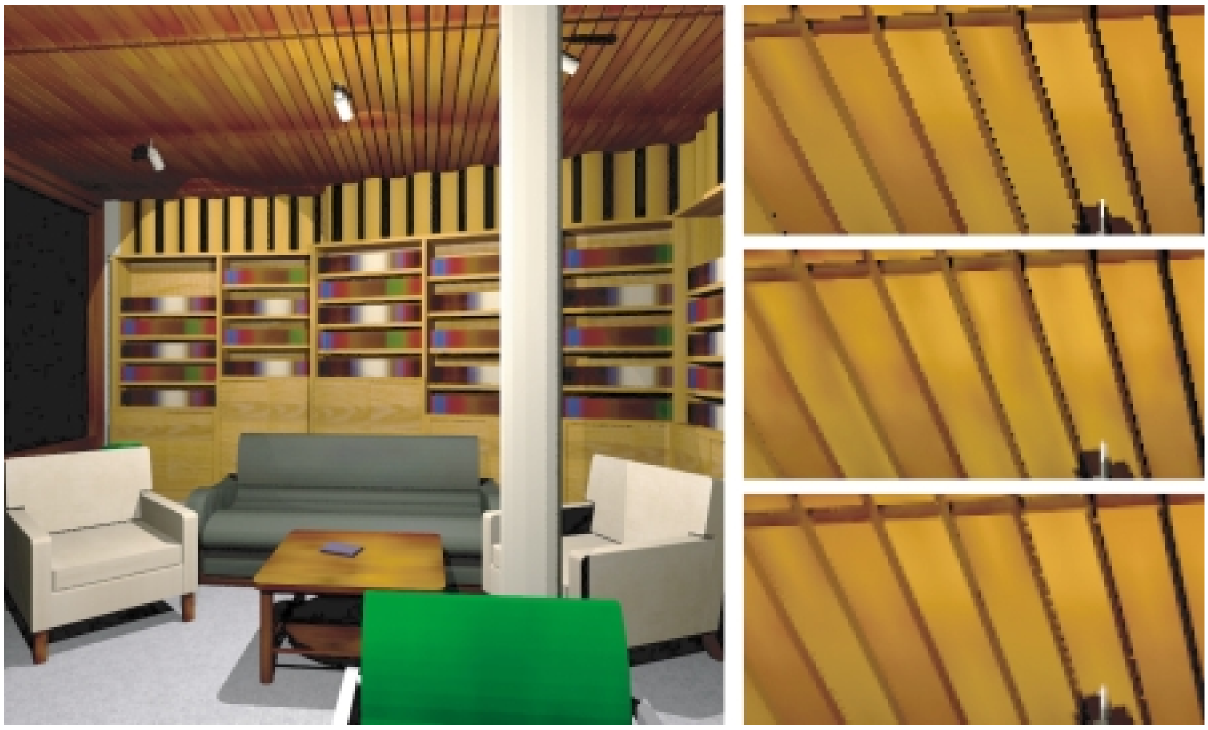

Figure 12: The left image shows our results for the Alto library model. On the right, from top to bottom, magnified images of a standard 1-sample-per-pixel image, a 4× supersampled image, and our edge anti-aliased results are shown. The books on the shelves appear blurry because they are modeled using low-resolution textures.

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

[25] James Stewart and Tasso Karkanis. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. In *Ninth Eurographics Workshop on Rendering*, pages 57–68, June 1998.

[26] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Tenth Eurographics Workshop on Rendering*, June 1999.

[27] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pages 361–370, 2001.

[28] Len Wanger, Jim Ferwerda, and Donald Greenberg. Perceiving spatial relationships in computer-generated images. *IEEE Computer Graphics and Applications*, 12(3):44–58, 1992.

[29] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A Ray Tracing Solution for Diffuse Interreflection. *Computer Graphics (ACM SIGGRAPH '88 Proceedings)*, 22(4):85–92, August 1988.

[30] Kwan-Hee Yoo, Dae Seoung Kim, Sung Yong Shin, and Kyung-Yong Chwa. Linear-time algorithms for finding the shadow volumes from a convex area light source. *Algorithmica*, 20(3):227–241, 1998.