Nonlinear Beam Tracing on a GPU

Li-Yi Wei Baoquan Liu Xu Yang Chongyang Ma Ying-Qing Xu Baining Guo

Abstract—Beam tracing [8] combines the flexibility of ray tracing and the speed of polygon rasterization. However, beam tracing so far only handles linear transformations; thus, it is only applicable to linear effects such as planar mirror reflections but not to nonlinear effects such as curved mirror reflection, refraction, caustics, and shadows.
In this paper, we introduce nonlinear beam tracing to render these nonlinear effects. Nonlinear beam tracing is highly challenging because commodity graphics hardware supports only linear vertex transformation and triangle rasterization. We overcome this difficulty by designing a nonlinear graphics pipeline and implementing it on top of a commodity GPU. This allows beams to be nonlinear where rays within the same beam do not have to be parallel or intersect at a single point. Using these nonlinear beams, real-time GPU applications can render secondary rays via polygon streaming similar to how they render primary rays. A major strength of this methodology is that it naturally supports fully dynamic scenes without the need to pre-store a scene database. Utilizing our approach, nonlinear ray tracing effects can be rendered in real-time on a commodity GPU under a unified framework.

Index Terms—nonlinear beam tracing, real-time rendering, GPU techniques, reflection, refraction, caustics, shadows

1 Introduction

Beam tracing was proposed by Heckbert and Hanrahan in 1984 [8] as a derivative of ray tracing that replaces individual rays with beams. One primary advantage of beam tracing is efficiency, as it can render individual beams via polygon rasterization, a feat that can be efficiently performed by today's commodity graphics hardware. Beam tracing also naturally resolves sampling, aliasing, and LOD issues that can plague conventional ray tracing. However, a major disadvantage of beam tracing is that it so far handles only linear transformations; thus, it is not applicable to nonlinear effects such as curved mirror reflection. Note that such nonlinear effects are not limited to curved geometry as even planar refraction is nonlinear. Another common nonlinear effect is bump-mapped surfaces, as they require incoherent ray bundles that cannot be handled by linear beams.

We introduce nonlinear beam tracing to render nonlinear effects such as curved mirror reflection, refraction, caustics, and shadows. We let beams be nonlinear where rays within the same beam do not have to be parallel or intersect at a single point. Beyond smooth ray bundles, our technique can also be applied to incoherent ray bundles; this is useful for rendering bump mapped surfaces.

Realizing nonlinear beam tracing is challenging as commodity graphics hardware supports only linear vertex transform and triangle rasterization. We overcome this difficulty by designing a nonlinear graphics pipeline and implementing it on top of a commodity GPU. Specifically, our nonlinear pipeline can render a given scene triangle into a projection region with nonlinear edges. This is achieved by customizing the vertex program to estimate a bounding triangle for the projection region and the fragment program to render the projection region out of the bounding triangle. Our main technical innovation is a bounding triangle algorithm that is both tight and easy to compute, and remains so for both smooth and perturbed beams. Our additional technical contributions include a parameterization that allows efficient bounding triangle estimation and pixel shading while ensuring projection continuity across adjacent beams, a frustum culling strategy where the frustum sides are no longer planes, and a nonlinear beam tree construction with proper memory management supporting multiple beam bounces.

Our nonlinear beam tracing approach has several advantages. First, since we utilize polygon rasterization for rendering beams, the performance rides in proportion with advances in commodity graphics hardware. This also makes it easier to integrate our method with existing polygon-rasterization based methods such as games. Second, our algorithm naturally supports fully-dynamic or large-polygon-count scenes since it does not require pre-storage of a scene database. Third, we provide a unified framework for rendering a variety of nonlinear ray tracing effects that have often been achieved via individual heuristics in previous graphics hardware rendering techniques. Our methodology also provides an interesting research insight by pushing polygon rasterization to the logic extreme for rendering secondary ray effects. Our main goal is not to replace ray tracing but instead to investigate an alternative methodology to achieve similar effects.

Our method has several limitations. Quality-wise, nonlinear beam tracing is only an approximation to ray tracing and may shift the locations of the rendered reflections/refractions. The shift is usually not perceptually objectionable or even noticeable (e.g. see Figure 1) and our rendered motions remain smooth with proper parallax and occlusion effects. If necessary, higher accuracy can be obtained by using smaller beams at the expense of more computation. We render all reflections and refractions via surface textures and thus the quality is affected by texture map resolutions. Our technique also cannot handle singularity situations such as total internal reflection. Performance-wise, our approach has a worst case complexity of $O(MN)$ with respect to the number of beams N and scene triangles M. (The dependency on N could be significantly reduced by our frustum culling acceleration as detailed later.) Due to this, our approach is most suitable for large or smooth interfaces (e.g. reflection off the mirrored teapot as in Figure 1). For highly complex interfaces (e.g. reflection off the fractured brown object in Figure 9), our performance can be slow as we would need many beams for accurate rendering. However, we have found approximation via coarse beams quite adequate since human perceptions are not very sensitive to the accuracy of complex reflections or refractions.

Our paper has the following technical contributions:

Figure 1: Nonlinear beam tracing on a GPU. This scene contains a mirror teapot reflecting complex objects in fully dynamic motions. Such kinds of scenes can be efficiently rendered via our polygon rasterization based approach. See Section 9 for detailed performance statistics. Quality-wise, even though our approach is only an approximation to ray tracing, the image quality remains visually similar; for each pair of images the ray tracing result is on the left while our result is on the right. The small images are close-ups around high curvature areas, including spout, lid, and handle, rendered under different view points.

- A general nonlinear beam tracing framework over the predominately linear ones in prior methods (e.g. [8]).
- A bounding triangle estimation algorithm that is not only tight but also very easy to compute. This is a major improvement over the prior method [11] which can greatly overestimate the bounding triangle size.
- A beam parameterization that provides higher order continuity than the C^0-only one in [11].
- A nonlinear beam tree construction method that handles multiple beam bounces.

2 Previous Work

Ray Tracing Ray tracing is the primary method for rendering global illumination effects. Most existing fast ray tracing approaches store the scene geometry into a database, and pre-compute certain acceleration data structures for efficient rendering. As such, they are most suitable for rendering static scenes [10]. Extending this approach for dynamic scenes has proven to be a major challenge; despite impressive results by recent techniques as surveyed in [28], [13], [27], it remains difficult to match the performance of previous ray tracing approaches tailored for static scenes.

We provide an alternative methodology for rendering ray tracing effects for fully dynamic scenes: instead of storing scene geometry into a database and building acceleration data structures, we stream down scene geometry and render them via GPU polygon rasterization (a philosophy pioneered by [2]). This allows us to naturally handle fully-dynamic and large-polygon-count scenes. Furthermore, despite the theoretical advantage of ray tracing over polygon rasterization in terms of depth complexity, so far polygon rasterization on a GPU still outperforms either CPU- or GPU-based ray tracers for rendering primary rays. Our analysis in Section 9 demonstrates that this performance advantage of polygon rasterization carries over to secondary rays for large smooth surfaces as well.

Furthermore, since most gaming applications are based on polygon rasterization, it is more natural to combine our technique with the existing gaming pipeline rather than CPU/GPU-based ray tracers.

GPU Techniques Approximating ray tracing effects on a GPU has a long trail of research efforts, as surveyed in [1].

For reflection, the ultimate goal is to render curved reflection of nearby objects with fully dynamic motion in real time. However, previous work either assumes distant objects (i.e. environment map), planar reflectors [8], static scene [31], or limited motion for frame-to-frame coherence [4]. Another possibility (e.g. [25], [20]) is to approximate near objects as depth imposters from which reflections are rendered via a technique similar to relief mapping. However, handling disocclusion and combining multiple depth-imposters can be difficult issues since they cannot be resolved via hardware z-buffering.

To render fully dynamic reflections without the depth sprite issues mentioned above, one possibility is to perform fine tessellation of scene geometry as in [18], [17], allowing curved mirror reflection of near objects with unlimited motion. However, since these techniques require fine tessellation of scene geometry, they often maintain a high geometry workload regardless of the original scene complexity, which is often quite simple for interactive applications such as games.

Refraction poses a bigger challenge than reflection, as even planar refraction can produce nonlinear beams [8]. Common techniques often handle only nearly planar refractors [23] or only far away scene objects [7], but not refraction of near objects through arbitrarily-shaped lens. One possibility to approximate such effects is via image-based heuristics [29]. This is similar to [25], in which the technique represents refracted objects as depth sprites and shares similar problem in depth compositing. Ihrke et al. [12] renders refraction effects via propagating wave-fronts through pre-computed volumetric textures. This technique can produce impressive refraction effects, but it requires pre-computation of incoming radiance. Our technique is complement to [12] as we can efficiently compute incoming radiance from a complex and dynamic scene. Caustics results from light focused by curved reflection or refraction and can be rendered by a standard two-pass process as demonstrated in [30].

Hou et al. [11] is a multi-perspective GPU technique which can render limited nonlinear effects. However, this is not for general nonlinear beam tracing as it only handles single-bounce reflections and refractions. In addition, the technique is limited to a simple C^0 parameterization and can only render simple scenes in real-time due to the huge pixel overdraw ratio caused by a rough bounding triangle estimation. Our technique, in

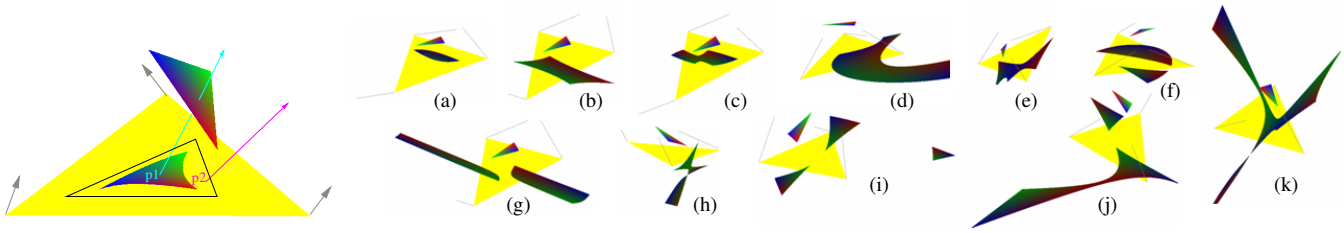


Figure 2: Nonlinear projection. Left: Rendering one scene triangle within one beam. The scene triangle is shown in RGB colors whereas the beam base triangle is shown in yellow. Due to nonlinear projection, the projected region may have curved edges. Our rendering is performed by a combination of a vertex program that estimates the bounding triangle (shown in black outline), and a fragment program that shades and determines if each bounding triangle pixel is within the true projection. In this example, p_1 lies within the true projection but not p_2. Right: Different cases of nonlinear beam projection. The projections are colored so that the same RGB colors are assigned from a scene vertex to the corresponding projected vertices. The number of projection regions are 1 for cases (a) to (e), 2 for cases (f, g, h, j, k), and 3 for case (i). Our algorithm can handle all these cases correctly.

contrast, supports full nonlinear beam tracing effects including multiple reflections/refractions and a general math framework that supports parameterizations with more than C^0 continuity as well as a tight bounding triangle estimation with a roughly constant overdraw ratio. This non-linear projection idea is also utilized in [16] for logarithmic shadow maps and [6] for single-center non-linear projections.

Dachsbacher et al. [3] introduces the notion of anti-radiance to avoid explicit visibility computation for global illuminations on a GPU. The technique can be considered complementary to our technique as it is most effective for diffuse objects in static scenes, whereas our technique is most effective for mirror reflections and refractions in dynamic scenes.

Linear Beam Tracing Linear beam tracing has been applied for computing visibility [19] or stitching together multiple pinhole cameras for interesting effects [21]. These methods are complementary to our approach, aiming at different applications.

3 Overview of Our Approach

For clarity, we first provide an overview of our algorithm as summarized in Algorithm 1. We present details for the individual parts of our algorithms in the subsequent sections.

function NonlinearBeamTracing()
  $B_0 \leftarrow$ eye beam from viewing parameters
  $\{B_{ij}\} \leftarrow$ BuildBeamTree(B_0) // *on CPU; Section 7*
  // *i indicates depth in beam tree with j labels nodes in level i*
  foreach (i, j) in depth first order
    RenderBeam(B_{ij}) // *on GPU; Sections 4, 5, 6, 8*
  end

function $\{B_{ij}\} \leftarrow$ BuildBeamTree(B_0) // *Section 7*
  trace beam shapes recursively as in [8]
  but allow nonlinear beams bouncing off curved surfaces

function RenderBeam(B) // *Sections 4, 5, 6, 8*
  foreach scene triangle $\triangle$
    $\tilde{\triangle} \leftarrow$ BoundingTriangle($\triangle$, B) // *vertex shader; Section 6*
    RenderTriangle($\tilde{\triangle}$, $\triangle$, B) // *fragment shader; Section 5*
  end

Algorithm 1: Overview of our system. See relevant sections for details.

Our nonlinear beam tracing framework is divided into two major parts: build beam tree on CPU and render beams on GPU, as illustrated in NonlinearBeamTracing(). All our beams take triangular cross sections and we compute and record the three boundary rays associated with each beam (BuildBeamTree()).

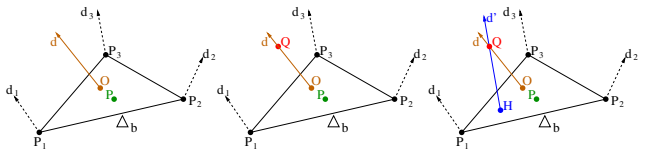
The core part of our algorithm renders scene triangles one by one via feed-forward polygon rasterization, as in RenderBeam(). As illustrated in Figure 2, due to the nonlinear nature of our beams, straight edges of a triangle might project onto curves on the beam base plane and the triangle might project onto multiple and/or non-triangular components. Such a nonlinear projection cannot be directly achieved by the conventional linear graphics pipeline. We instead implement a nonlinear graphics pipeline on a GPU by customizing both the vertex and fragment programs as follows. For each scene triangle, our vertex program estimates a bounding triangle that properly contains the projected region (BoundingTriangle()). This bounding triangle is then passed down to the rasterizer; note that even though the projected region may contain curved edges or even multiple components, the bounding triangle possesses straight edges so that it can be rasterized as usual. For each pixel within the bounding triangle, our fragment program performs a simple ray-triangle intersection test to determine if it is within the true projection; if so, the pixel is shaded, otherwise, it is killed (RenderTriangle()).

Rendering results for each beam is stored as a texture on its base triangle. The collection of beam base triangles is then texture mapped onto the original reflector/refractor surfaces for final rendering. For clarity, we call the collection of beam base triangles corresponding to a single reflector/refractor object the S (simplified) mesh. The resolution of the S mesh trades off between rendering quality and speed. We associate a S mesh with a reflector/refractor interface through either mesh simplification or procedural methods, depending on whether the interface is static or dynamic.

4 Beam Parameterization

As described in Section 3, we utilize beams with triangular cross sections for easy processing. For each point on the beam base triangle, our beam parameterization defines an associated ray with a specific origin and direction. Our beam parameterization for ray origins and directions must satisfy the following requirements: (1) interpolating vertex values, (2) edge values depend only on the two adjacent vertices, (3) at least C^0 continuity across adjacent beams to ensure pattern continuity, (4) no excessive undulations inside each beam, (5) easy to compute in a per pixel basis for efficient fragment program execution, and (6) easy bounding triangle estimation for efficient vertex program execution. We provide two flavors of parameterizations: a smooth parameterization for coherent beams and a bump-map parameterization for perturbed beams.

Smooth S^n Parameterization



(a) parameterization (b) projection (c) transfer

Figure 3: Illustrations for several basic definitions in our algorithm. In (a), the point P on the beam base $\triangle_b$ has a corresponding ray origin $\overrightarrow{O}$ and direction $\overrightarrow{d}$. In (b), P is the projection of scene point $\overrightarrow{Q}$ under the beam parameterization in (a). In (c), H is the transfer of $\overrightarrow{Q}$ in the direction $\overrightarrow{d'}$, regardless of the beam parameterization.

Given a beam B with normalized boundary rays $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$ defined on three vertices of its base triangle $\triangle P_1P_2P_3$ (Figure 3a), our smooth parameterization of the ray origin and direction for any point P on the beam base plane is defined via the following formula:

$$\begin{aligned}
\overrightarrow{O} &= \sum_{i+j+k=n} \overrightarrow{a_{ijk}}\, \omega_1^i \omega_2^j \omega_3^k \text{ (ray origin)} \\
\overrightarrow{d} &= \sum_{i+j+k=n} \overrightarrow{b_{ijk}}\, \omega_1^i \omega_2^j \omega_3^k \text{ (ray direction)} \\
\omega_i &= \frac{area(\triangle PP_jP_k)}{area(\triangle P_1P_2P_3)}_{(i,j,k)=(1,2,3),(2,3,1),(3,1,2)} \\
\overrightarrow{a_{ijk}} &= \overrightarrow{L_{ijk}}(\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}) \\
\overrightarrow{b_{ijk}} &= \overrightarrow{L_{ijk}}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3})
\end{aligned} \tag{1}$$

where each $\overrightarrow{L_{ijk}}$ belongs to a general function class $\overrightarrow{L}$ satisfying the following linear property for any real number s, $\{\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}\}$, and $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$:

$$\begin{aligned}
&\overrightarrow{L}(\overrightarrow{P_1} + s\overrightarrow{d_1}, \overrightarrow{P_2} + s\overrightarrow{d_2}, \overrightarrow{P_3} + s\overrightarrow{d_3}) \\
= \;&\overrightarrow{L}(\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}) + s\overrightarrow{L}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3})
\end{aligned} \tag{2}$$

We term this formulation our S^n parameterization. Note that this is an extension of triangular Bezier patches [5] with two additional constraints: (1) the identical formulations of $\overrightarrow{a_{ijk}}$ and $\overrightarrow{b_{ijk}}$ in Equation 1, and (2) the linear property for $\overrightarrow{L}$ as in Equation 2. These are essential for an efficient GPU implementation and a rigorous math analysis as will be detailed later. Even with these two restrictions, our S^n parameterization is still general enough to provide desired level of continuity. In particular, the C^0 parameterization in [11] (Equation 3) is just a special case of our S^n parameterization with $n = 1$, and a S^3 parameterization can be achieved via the cubic Bezier interpolation as proposed in [26] (see Equation 6). Note that in general $\overrightarrow{O} \neq \overrightarrow{P}$ unless under the S^1 parameterization.

$$\begin{aligned}
\overrightarrow{O} &= \omega_1\overrightarrow{P_1} + \omega_2\overrightarrow{P_2} + \omega_3\overrightarrow{P_3} \\
\overrightarrow{d} &= \omega_1\overrightarrow{d_1} + \omega_2\overrightarrow{d_2} + \omega_3\overrightarrow{d_3}
\end{aligned} \tag{3}$$

Even though in theory general C^k continuity might be achieved via a large enough n in our S^n parameterization, we have found out through experiments that an excessively large n not only slows down computation but also introduces excessive undulations. Consequently, in our current implementation, we have settled down for our S^1 and S^3 parameterizations. The tradeoff between S^1 and S^3 parameterizations is that S^1 is faster to compute (both in vertex and pixel programs) but S^3 offers higher quality as it ensures C^1 continuity at S mesh vertices. This quality difference is most evident for scene objects containing straight line geometry or texture, as exemplified in Figure 4.

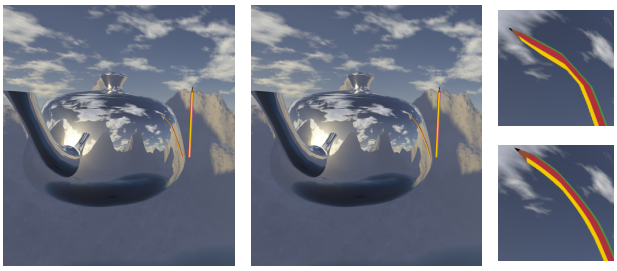


Figure 4: Comparison of beam parameterizations. Left: S^1 parameterization. Middle: S^3 parameterization. Right-top/right-bottom: close up views around the reflections of left/middle. For the S^1 case, notice the sudden direction change of the pencil reflection over the teapot.

Bump-map Parameterization For perturbed beams, we compute $\overrightarrow{d}$ and $\overrightarrow{O}$ via our S^1 parameterization followed by standard bump mapping. Due to the nature of perturbation we have found a parameterization with higher level continuity unnecessary.

For efficient implementation and easy math analysis, we assume all rays $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$ point to the same side of the beam base plane. The exception happens only very rarely for beams at grazing angles. We enforce this in our implementation by proper clamping of $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$, i.e. setting the z-component d_z of $\overrightarrow{d}$ (in the local frame of the beam base) to $\max(\delta, d_z)$ for a small positive value δ.

5 Core Computations & Definitions

Our beam parameterization above is utilized in two major computations of our algorithm: *intersection*, where a point $\overrightarrow{Q}$ is found by intersecting a scene triangle with a ray parameterized via a point $\overrightarrow{P}$ on a beam base, and *projection*, where an inverse computation is performed to find the corresponding $\overrightarrow{P}$ on a beam base for a space point $\overrightarrow{Q}$ (Figure 3b). Wel also define an additional operation, termed *transfer*, that does not depend on the beam parameterization (Figure 3c).

Intersection For each point $\overrightarrow{P}$ on a beam base, we first compute its ray origin $\overrightarrow{O_P}$ and direction $\overrightarrow{d_P}$ via Equation 1 (followed by bump map if necessary). We then perform a simple ray-triangle intersection between the ray $(\overrightarrow{O_P}, \overrightarrow{d_P})$ and the space triangle in question. This intersection operation can be easily implemented in our fragment program to render each point $\overrightarrow{P}$ within the bounding triangle $\tilde{\triangle}$ computed by our vertex program, as illustrated in Figure 2. If the ray does not intersect the scene triangle, the pixel associated with $\overrightarrow{P}$ is killed immediately. Otherwise, its attributes (depth, color, or texture coordinates) are determined from the intersection point $\overrightarrow{Q}$ and $\overrightarrow{P}$ is shaded followed by frame-buffer write.

function RenderTriangle($\tilde{\triangle}$, $\triangle$, B)
  foreach pixel/point $P \in \tilde{\triangle}$
    compute ray origin $\overrightarrow{O_P}$ and direction $\overrightarrow{d_P}$
    $\overrightarrow{Q} \leftarrow$ intersection of ray $(\overrightarrow{O_P}, \overrightarrow{d_P})$ with $\triangle$
    if $\overrightarrow{Q}$ does not exist // *no intersection*
      kill P
    else
      shade P according to $\overrightarrow{Q}$ and $\triangle$
      // *via standard methods depending on barycentric coordinates*
    end
  end

Algorithm 2: Pixel shading.

Projection Computing $\overrightarrow{P}$ on a beam base for a given space

point $\overrightarrow{Q}$ would require solving a cubic polynomial for the S^1 parameterization (and higher order polynomials for our general S^n parameterization). This can be computationally expensive, but fortunately we never need to compute projections on a GPU as it is only needed for determining the boundary directions $\{\overrightarrow{d_i}\}_{i=1,2,3}$ of each beam during the beam tree construction phase on a CPU (Section 7.2).

Transfer In our bounding triangle estimation algorithm below, we utilize a specific operation termed *transfer*, defined as follows. A scene point $\overrightarrow{Q}$ has a transfer $\overrightarrow{H}$ on a plane Π_b in the direction $\overrightarrow{d}$ if there exists a real number t so that $\overrightarrow{Q} = \overrightarrow{H} + t\overrightarrow{d}$. In other words, unlike our projection operation, transfer is a pure geometric operation and has no association with the beam parameterization.

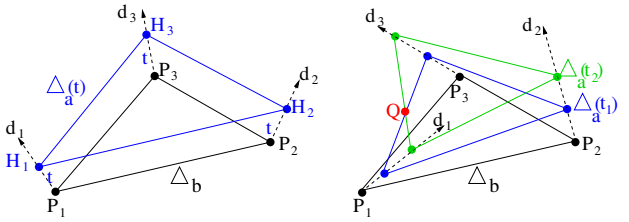


Figure 5: Geometric interpretation of our beam parameterization. The beam is defined by its base triangle $\triangle P_1P_2P_3$ and boundary directions $\{\overrightarrow{d_i}\}_{i=1,2,3}$. (Left): an affine triangle $\triangle_a(t)$ is defined by $\{\overrightarrow{H_i} = \overrightarrow{P_i} + t\overrightarrow{d_i}\}_{i=1,2,3}$. In this case, $\{\overrightarrow{d_i}\}_{i=1,2,3}$ point away from each other so $\triangle_a(t)$ with different t values will never intersect each other. (Right): when $\{\overrightarrow{d_i}\}_{i=1,2,3}$ not pointing away from each other, $\triangle_a(t)$ with different t values might intersect each other, such as $\triangle_a(t_1)$ and $\triangle_a(t_2)$, causing any $\overrightarrow{Q} \in \triangle_a(t_1) \cap \triangle_a(t_2)$ to have multiple projections on the beam base $\triangle_b$. The threshold plane Π_T, not shown here, is below $\triangle_b$ in the left case and high on the sky in the right case (when $\{\overrightarrow{d_i}\}_{i=1,2,3}$ eventually diverge).

Geometric Interpretation Even though we compute intersections and projections algebraically, we have found the following geometric interpretation very useful for understanding our algorithm. We rely on the following two math quantities for computing intersections and projections (see Figure 5): (1) *affine plane* $\Pi_a(t)$, which is t distances away from the beam base plane Π_b and contains affine triangle $\triangle_a(t)$ with vertices $\overrightarrow{H_i} = \overrightarrow{P_i} + t\overrightarrow{d_i}, i = 1,2,3$, and (2) *threshold plane* Π_T, which is a special affine plane $\Pi_a(t_{res})$ with a particular value t_{res} so that any space point $\overrightarrow{Q}$ above Π_T is guaranteed to have at most one projection within the beam base triangle $\triangle P_1P_2P_3$. (Geometrically, $\{\overrightarrow{d_i}\}_{i=1,2,3}$ point away from each other above Π_T.) See below for detailed definitions.

Affine planes/triangles have this "barycentric invariance property" that if $\overrightarrow{H} \in \Pi_a$ has the same set of barycentric coordinates (relative to $\triangle_a$) as $\overrightarrow{P} \in \Pi_b$ (relative to $\triangle_b$), then $\overrightarrow{P}$ is a projection of $\overrightarrow{H}$ on Π_b. This property allows us to solve the projection opereration geometrically as follows. Say we want to compute the projection $\overrightarrow{P}$ of a space point $\overrightarrow{Q}$. We start with $\triangle_a(t=0) \equiv \triangle_b$ and gradually lift $\triangle_a(t)$ away from $\triangle_b$ (by increasing t) until $\Pi_a(t)$ contains $\overrightarrow{Q}$. We can then compute the barycentric coordinates of $\overrightarrow{Q}$ relative to $\triangle_a(t)$, and use that to interpolate the projection $\overrightarrow{P}$ from $\triangle_b(t)$.

When the three boundary directions $\{\overrightarrow{d_i}\}_{i=1,2,3}$ of a beam point away from each other, $\triangle_a(t)$ will rise monotonically from $\triangle_b$ with increasing t (Figure 5 left). Thus, each space point $\overrightarrow{Q}$ has a unique projection $\overrightarrow{P}$ on the beam base. Otherwise, $\triangle_a(t)$ may go up and down several times (at most 3 for S^1) before it eventually goes away, and this is the exact cause for multiple projections, as illustrated in Figure 5 right where $\overrightarrow{Q}$ can have different barycentric coordinates relative to $\triangle_a(t_1)$ and $\triangle_a(t_2)$ (and thus different $\overrightarrow{P_1}$ and $\overrightarrow{P_2}$ on Π_b). Fortunately, it can be shown (Appendix B.2) that when $\triangle_a(t)$ is above the threshold plane Π_T, it will only rise monotonically. These geometric properties are very important in both our bounding triangle estimation (Section 6) and beam tree construction (Section 7.2), to be detailed later.

Threshold plane Π_T We define the *threshold plane* Π_T of a beam B as an affine plane $\Pi_a(t_{res})$ with a particular value t_{res} so that any space point $\overrightarrow{Q}$ above Π_T is guaranteed to have at most one projection within the beam base triangle $\triangle P_1P_2P_3$. Intuitively, the beam frustum above the threshold plane Π_T has diverging ray directions.

We now define the threshold plane for our S^1 parameterization (Equation 1). Let $\overrightarrow{P_{i,t}} = \overrightarrow{P_i} + t\overrightarrow{d_i}, \ i = 1,2,3$. Define the following three quadratic functions

$$\begin{aligned} f_1(t) &= (\overrightarrow{P_{2,t}} - \overrightarrow{P_{1,t}}) \times (\overrightarrow{P_{3,t}} - \overrightarrow{P_{1,t}}) \cdot \overrightarrow{d_1} \\ f_2(t) &= (\overrightarrow{P_{3,t}} - \overrightarrow{P_{2,t}}) \times (\overrightarrow{P_{1,t}} - \overrightarrow{P_{2,t}}) \cdot \overrightarrow{d_2} \\ f_3(t) &= (\overrightarrow{P_{1,t}} - \overrightarrow{P_{3,t}}) \times (\overrightarrow{P_{2,t}} - \overrightarrow{P_{3,t}}) \cdot \overrightarrow{d_3} \end{aligned}$$

Solving the three equations separately $f_1(t) = f_2(t) = f_3(t) = 0$ we obtain a set of (at most 6) real roots $\Im$. Denote $t_{max} = \max(0, \Im)$. The threshold plane Π_T is then defined as the affine plane $\Pi_a(t_{res})$ so that $t_{res} > t_{max}$ and $\min\left(\|\overrightarrow{P_{res}} - \overrightarrow{P_{max}}\|, \forall \overrightarrow{P_{res}} \in \triangle_a(t_{res}), \overrightarrow{P_{max}} \in \triangle_a(t_{max})\right) > 0$. Note that $\Pi_a(t_{res})$ always exists as long as t_{res} is large enough. Intuitively, for $t > t_{res}$ and $\Delta t > 0$, we have $\overrightarrow{P_{1,t}} + \Delta t \cdot \overrightarrow{d_1}$, $\overrightarrow{P_{2,t}} + \Delta t \cdot \overrightarrow{d_2}$ and $\overrightarrow{P_{3,t}} + \Delta t \cdot \overrightarrow{d_3}$ on the same side of $\Pi_a(t)$. See Appendix B.2 for proof.

6 Bounding Triangle Estimation

Here, we describe our bounding triangle estimation algorithm as implemented in our vertex program. Our bounding triangle algorithm needs to satisfy two goals simultaneously: (1) easy computation for fast/short vertex program and (2) conservative and yet tight bounding region for low pixel overdraw ratio, defined as the relative number of pixels of the bounding triangle over the projected area. Since GPU has multiple main execution stages, we have to design an algorithm that properly balances the workload between vertex and fragment units. (The performance impact remains even for a unified shader architecture such as NVIDIA Geforce 8800.)

In the following, we present our bounding triangle algorithm for S^1 and bump-map parameterizations. (Extension to S^n is presented in Section 6.2.) Our algorithms rely on the two math quantities *affine plane* and *threshold plane* as described in Section 5.

6.1 Bounding $\triangle$ for S^1 parameterization

Our bounding triangle algorithm follows a similar philosophy of [11] so we begin with a brief description of their algorithm. For each scene triangle $\triangle$ with vertices $\{\overrightarrow{Q_i}\}_{i=1,2,3}$, [11] compute nine *transfer* (defined in Section 5) points $\{P_{ij}\}_{i,j=1,2,3}$ on the beam base plane Π_b where P_{ij} is the transfer of $\overrightarrow{Q_i}$ in the direction of $\overrightarrow{d_j}$. A bounding triangle $\tilde{\triangle}$ is then computed from $\{P_{ij}\}_{i,j=1,2,3}$. We term this algorithm *general case* as in Algorithm 3. It can be easily shown that $\tilde{\triangle}$ computed above

function $\tilde{\triangle} \leftarrow$ BoundingTriangle($\triangle$, B) // *vertex shader*

$\{\overrightarrow{d_i}\}_{i=1,2,3} \leftarrow$ boundary rays of B
$\{\overrightarrow{P_i}\}_{i=1,2,3} \leftarrow$ base vertices of B
$\Pi_b \leftarrow$ base plane of B
$\triangle_b \leftarrow$ base triangle of B
$\Pi_T \leftarrow$ threshold plane of B
// *all quantities above are pre-computed per* B
$\{\overrightarrow{Q_i}\}_{i=1,2,3} \leftarrow$ vertices of scene triangle $\triangle$
if $\{\overrightarrow{Q_i}\}_{i=1,2,3}$ not all inside or outside B
  // *brute force case*
  $\tilde{\triangle} \leftarrow$ entire base triangle of B
else if $\triangle$ not above Π_T
  // *general case, e.g. Figure 6 right*
  foreach i,j = 1,2,3
    $P_{ij} \leftarrow$ Transfer($\overrightarrow{Q_i}$, $\overrightarrow{d_j}$, Π_b)
  $\tilde{\triangle} \leftarrow$ BoundingTriangle($\{P_{ij}\}_{i,j=1,2,3}$)
else
  // *unique projection case, e.g. Figure 6 left*
  $\Pi_a \leftarrow$ affine plane of B passing through the top of $\triangle$
  $\triangle_a \leftarrow$ affine triangle on Π_a
  foreach i,j = 1,2,3
    $H_{ij} \leftarrow$ Transfer($\overrightarrow{Q_i}$, $\overrightarrow{d_j}$, Π_a)
  $\triangle E_1E_2E_3 \leftarrow$ BoundingTriangle($\{H_{ij}\}_{i,j=1,2,3}$)
  foreach i = 1,2,3
    $\{\omega_k\}_{k=1,2,3} \leftarrow \Omega(\{\overrightarrow{E_i}\}, \triangle_a)$ // *barycentric coordinates*
    $\{\overrightarrow{F_i}\} \leftarrow \Omega^{-1}(\{\omega_k\}_{k=1,2,3}, \triangle_b)$
  end for
  $\tilde{\triangle} \leftarrow \triangle F_1F_2F_3$
end if

Algorithm 3: Bounding triangle estimation algorithm for our S^1 parameterization. $\Omega(\overrightarrow{P}, \triangle)$ indicates the computation of barycentric coordinates of a point $\overrightarrow{P}$ on a triangle $\triangle$, and $\Omega^{-1}(\{\omega_k\}_{k=1,2,3}, \triangle)$ is the inverse operation of computing a point from a set of barycentric coordinates $\{\omega_k\}_{k=1,2,3}$ and a triangle $\triangle$.

is guaranteed to be a bounding triangle for the projection of the scene triangle $\triangle$ as long as its three vertices completely lie within the beam. (Otherwise, $\triangle$ might straddle across the beam boundary, causing strange projections (e.g. with infinite size) for which we have not yet been able to find a mathematically conservative bounding triangle.) (See Appendix B.4 for proof.) However, the bounding triangle computed by [11] is too crude; as the scene triangle gets smaller or further away from Π_b, the overdraw ratio can become astronomically large, making rendering of large scene models impractical. The reasoning behind this phenomenon is illustrated in Figure 6.

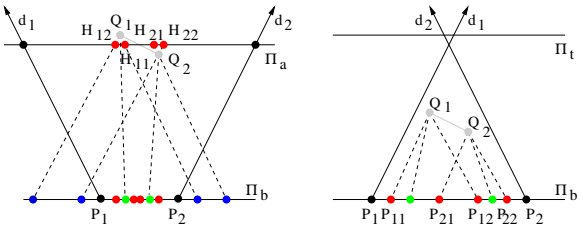


Figure 6: 2D illustration of our bounding triangle algorithm. Here we plot two scenarios of diverging (left) and converging (right) ray directions (corresponding to the unique-projection and general cases in Algorithm 3). The green points indicate projections of the scene triangle vertices $\overrightarrow{Q_1}$ and $\overrightarrow{Q_2}$. [11] computes the bounding triangle from the blue points while our algorithm from the red points. Note that in the left (diverging) case, the overdraw ratio can be arbitrary large for [11] when the scene triangle is small or far away from the beam base plane. Our algorithm, in contrast, always maintains a roughly constant overdraw ratio. On the right (converging) case, our algorithm reduces to [11] and the overdraw ratio is limited due to the nature of converging rays. (In the left case Π_a should really pass through $\overrightarrow{Q_1}$ instead of the middle of the scene triangle, but we plot it this way for clarity of illustration.)

We provide a better algorithm as follows. As illustrated in Figure 6, the large overdraw ratio in our general case algorithm only happens when the rays are diverging. We address this issue by first computing nine transfers $\{H_{ij}\}_{i,j=1,2,3}$ and the associated bounding triangle $\triangle E_1E_2E_3$ similar to our general case algorithm, but instead the base plane Π_b these computations are performed on an affine plane Π_a that passes through the top of the scene triangle $\triangle$. (If there are muliple affine planes $\Pi_a(t)$ that pass through the top, we choose the one with largest t value.) We then transport vertices of $\triangle E_1E_2E_3$ on Π_a to the vertices of $\triangle F_1F_2F_3$ on Π_b so that they share identical barycentric coordinates. It can be proven (as in Appendix B.4) that $\triangle F_1F_2F_3$ is guaranteed to be a proper bounding triangle as long as the scene triangle $\triangle$ is above the threshold plane Π_T. (Intuitively, the algorithm works because of the barycentric invariance property of affine planes as described in Section 5. This ensures that $\triangle F_1F_2F_3$ is a bounding triangle as long as $\triangle E_1E_2E_3$ is also one.) Furthermore, since Π_a is much closer to $\triangle$ than Π_b, the bounding region is usually much tighter and thus reduced overdraw ratio. Since this algorithm is faster than our general case algorithm but works only when each space point $\overrightarrow{Q} \in \triangle$ has at most one projection in the base triangle (due to the fact that $\triangle$ is above Π_T), we term it our *unique projection case* algorithm.

Since our unique projection case algorithm is not guaranteed to work when the scene triangle $\triangle$ is not above the threshold plane Π_T, in this case, we simply keep our general case algorithm. Unlike the diverging case, the region below the threshold plane is mostly converging and the volume is bounded, so the overdraw ratio is usually small as illustrated in Figure 6 right. Our algorithm also fails to be conservative when the three vertices of the scene triangle lie on different sides of the beam boundary. When this happens, we simply use the entire beam base as the bounding triangle. Since each frustum side of our beams is a bilinear surface for our S^1 parameterization, this frustum intersection test can be efficiently performed via the method described in [24].

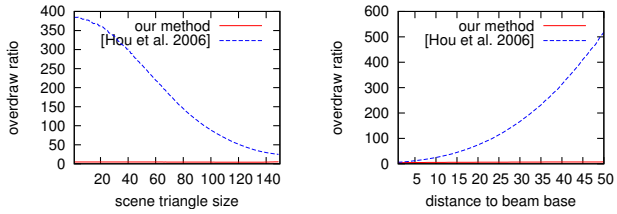


Figure 7: Overdraw ratio comparison. Here, we plot the pixel overdraw ratio of a single scene triangle being rendered within one beam. On the left, we fix the scene triangle in space but vary its size. On the right, we fix the scene triangle size but vary its distance to beam base.

A comparison between [11] and our algorithm is demonstrated in Figure 7. Ideally, the overdraw ratio should remain constant with varying scene triangle size and distance to beam base. Note that our algorithm remains low and roughly constant while [11] exhibits exponential growth. Consequently our algorithm runs much faster than [11] as demonstrated in Section 9.

6.2 Bounding $\triangle$ for S^n parameterization

In addition to *affine plane* and *threshold plane*, our algorithm for S^n parameterization depends on two additional quantities: (1) *extremal directions* $\{\widehat{d_1}, \widehat{d_2}, \widehat{d_3}\}$, which are expansions of the original beam directions $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$ so that they "contain" all interior ray directions $\overrightarrow{d}$, and (2) *maximum offsets* μ_b and μ_d,

which are maximum offsets between the S^n and S^1 parameterizations of ray origin and direction for all $\overrightarrow{P}$ on the base triangle $\triangle_b$.

Extremal directions $\{\widehat{d_i}\}_{i=1,2,3}$ Given a beam B with base triangle $\triangle P_1P_2P_3$, its *extremal directions* is a set of ray directions $\{\widehat{d_1}, \widehat{d_2}, \widehat{d_3}\}$ emanating from $\{P_1, P_2, P_3\}$ so that for each P within $\triangle P_1P_2P_3$ there exists a set of real numbers $\widehat{\omega_1}, \widehat{\omega_2}, \widehat{\omega_3}$, $0 \leq \widehat{\omega_1}, \widehat{\omega_2}, \widehat{\omega_3}$ and $\widehat{\omega_1} + \widehat{\omega_2} + \widehat{\omega_3} = 1$ that can express $\overrightarrow{d}(P)$ as follows:

$$\overrightarrow{d}(P) = \widehat{\omega_1}\widehat{d_1} + \widehat{\omega_2}\widehat{d_2} + \widehat{\omega_3}\widehat{d_3}$$

Extremal directions exist as long as all the beam directions point away to the same side of the beam base plane. Note that for S^1 parameterization we could have $\widehat{d_i} = \overrightarrow{d_i}$, but this is not true for general S^n parameterization and bump mapped surfaces. Extremal directions can be computed analytically for S^n parameterization and from the maximum perturbation angle δ for bump map parameterization.

Maximum offsets μ_b **and** μ_d We define μ_b and μ_d as the maximum offsets between the S^n and S^1 parameterizations of ray origin and direction for all $\overrightarrow{P}$ on the base triangle $\triangle_b$:

$$\begin{aligned} \mu_b &= \max(\|\overrightarrow{O_P} - \sum_{i=1}^{3} \omega_i \overrightarrow{P_i}\|, \forall \overrightarrow{P} \in \triangle_b) \\ \mu_d &= \max(\|\overrightarrow{d_P} - \sum_{i=1}^{3} \omega_i \overrightarrow{d_i}\|, \forall \overrightarrow{P} \in \triangle_b) \end{aligned} \quad (4)$$

where $\overrightarrow{O_P}$ and $\overrightarrow{d_P}$ are ray origin and direction computed according to our S^n parameterization, $\sum_{i=1}^{3} \omega_i \overrightarrow{P_i}$ and $\sum_{i=1}^{3} \omega_i \overrightarrow{d_i}$ are same quantities computed via our S^1 parameterization, and $\{\omega_i\}_{i=1,2,3}$ are the barycentric coordinates of $\overrightarrow{P}$ with respect to $\triangle_b$. Since our S^n parameterizations are just polynomials, μ_b and μ_d can be calculated by standard optimization techniques.

Threshold plane Π_T We also need to modify the threshold plane definition from our S^1 parameterization (Section 5) for our S^n parameterization (Equation 1) by replacing the condition $\min\left(\|\overrightarrow{P_{res}} - \overrightarrow{P_{max}}\|, \forall \overrightarrow{P_{res}} \in \triangle_a(t_{res}), \overrightarrow{P_{max}} \in \triangle_a(t_{max})\right) > 0$ with $\min\left(\|\overrightarrow{P_{res}} - \overrightarrow{P_{max}}\|, \forall \overrightarrow{P_{res}} \in \triangle_a(t_{res}), \overrightarrow{P_{max}} \in \triangle_a(t_{max})\right) > \mu_b + t_{max}\mu_d$ (μ_b and μ_d are defined in Equation 4).

This algorithm described in Section 6.1 only works for S^1 parameterization but we can extend it for our general S^n ($n > 1$) parameterization by replacing $\overrightarrow{d_j}$ with $\widehat{d_j}$ followed by proper ϵ-expansion of $\tilde{\triangle}$ as described under ExpandedBoundingTriangle() in Algorithm 4. (The amount of expansion ϵ is zero for S^1 pamaterization, and is non-zero for S^n parameterization to ensure that $\tilde{\triangle}$ is conservative even when $\overrightarrow{O_P} \neq \overrightarrow{P}$.) The algorithm is summarized in Algorithm 4.

6.3 Bounding $\triangle$ for bump-map parameterization

We pre-compute the maximum perturbation angle δ formed by each ray and its unperturbed cousin sharing the same origin as determined by our S^1 parameterization. Given a scene triangle $\triangle Q_1Q_2Q_3$, we first compute its S^1 bounding triangle via our S^1 algorithm. We then extend this S^1 bounding triangle outward via a distance D computed via the following formula:

$$\begin{aligned} \tau &= \delta + \phi \\ \theta_1 &= \arcsin(\min(\overrightarrow{n} \cdot \overrightarrow{d_{Q_1}}, \overrightarrow{n} \cdot \overrightarrow{d_{Q_2}}, \overrightarrow{n} \cdot \overrightarrow{d_{Q_3}})) \\ \theta_2 &= \arcsin(\min(\overrightarrow{n} \cdot \overrightarrow{d_1}, \overrightarrow{n} \cdot \overrightarrow{d_2}, \overrightarrow{n} \cdot \overrightarrow{d_3})) \\ \theta_3 &= \begin{cases} \theta_2 & \theta_2 > \tau \\ \theta_1 & \theta_2 \leq \tau \text{ and } \theta_1 > \tau \\ \tau & \text{otherwise} \end{cases} \\ D &= \frac{\max(\|Q_iP_{ij}\|,\ \forall i,j = 1,2,3) \cdot \sin\tau}{\sin(\theta_3 - \tau)} \end{aligned} \quad (5)$$

where ϕ is the maximum angle formed between any two rays within the same beam, $\overrightarrow{n}$ is the normal of the beam base (facing the inside of the beam), $\overrightarrow{d_{Q_i}} = \frac{\overrightarrow{Q_i} - \overrightarrow{P_{Q_i}}}{\|\overrightarrow{Q_i} - \overrightarrow{P_{Q_i}}\|}$ with $\overrightarrow{P_{Q_i}}$ being the un-perturbed projection of $\overrightarrow{Q_i}$, $\overrightarrow{d_i}$ the direction at $\overrightarrow{P_i}$, and $\overrightarrow{P_{ij}}$ the transfer of $\overrightarrow{Q_i}$ at direction $\overrightarrow{d_j}$ (as in our S^1 bounding triangle algorithm). Intuitively, this algorithm extends the original S^1 bounding triangle by an amount D so that to cover all shifted projection locations caused by bump maps; see Appendix B.5 for proof. Note that obtaining $\{\overrightarrow{P_{Q_i}}\}_{i=1,2,3}$ requires solving three cubic polynomials; fortunately, this is only needed when $\theta_2 \leq \tau$, a rare condition in practical situations.

function $\tilde{\triangle} \leftarrow$ BoundingTriangle($\triangle$, B) *// vertex shader*

```
{d̂_i}_{i=1,2,3} ← extremal rays of B
{P⃗_i}_{i=1,2,3} ← base vertices of B
Π_b ← base plane of B
△_b ← base triangle of B
μ_b, μ_d ← maximum offsets of B
Π_T ← threshold plane of B
// all quantities above are pre-computed per B
{Q⃗_i}_{i=1,2,3} ← vertices of scene triangle △
if {Q⃗_i}_{i=1,2,3} not all inside or outside B
    // brute force case
    △̃ ← entire base triangle of B
else if △ not above Π_T
    // general case, e.g. Figure 6 right
    foreach i,j = 1,2,3
        P_ij ← Transfer(Q⃗_i, d̂_j, Π_b)
    △̃ ← ExpandedBoundingTriangle({P_ij}_{i,j=1,2,3}, Π_b)
else
    // unique projection case, e.g. Figure 6 left
    Π_a ← affine plane of B passing through the top of △
    △_a ← affine triangle on Π_a
    foreach i,j = 1,2,3
        H_ij ← Transfer(Q⃗_i, d̂_j, Π_a)
    △E_1E_2E_3 ← ExpandedBoundingTriangle({H_ij}_{i,j=1,2,3}, Π_a)
    foreach i = 1,2,3
        {ω_k}_{k=1,2,3} ← Ω({E⃗_i}, △_a) // barycentric coordinates
        {F⃗_i} ← Ω^{-1}({ω_k}_{k=1,2,3}, △_b)
    end for
    △̃ ← △F_1F_2F_3
end if
```

function $\tilde{\triangle} \leftarrow$ ExpandedBoundingTriangle($\{P_{ij}\}_{i,j=1,2,3}$, Π_a)

```
△̃ ← BoundingTriangle({P_ij}_{i,j=1,2,3})
θ_min ← min(angle(d̂_i, Π_a), ∀i = 1,2,3)
t ← the t value of Π_a relative to Π_b
expand △̃ outward with distance ε = (μ_b + tμ_d) / sin(θ_min)
```

Algorithm 4: Bounding triangle estimation algorithm for our S^n parameterization. The algorithm is very similar to the S^1 case described in Algorithm 3, with the differences that (1) $\{\widehat{d_i}\}_{i=1,2,3}$ is used instead of $\{\overrightarrow{d_i}\}_{i=1,2,3}$ and (2) ExpandedBoundingTriangle() is used instead of BoundingTriangle().

7 Beam Construction

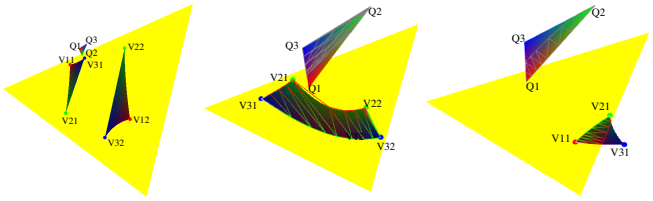
Here we describe how we construct beam related data structures: S mesh, beam tree, and textures. All these operations happen on a CPU.

7.1 S Mesh Construction

For a static reflector/refractor, we build its S mesh via mesh simplification [9]. We have found this sufficient even though in theory a view dependent technique might yield superior quality. For a dynamic reflector/refractor, care must be taken to ensure animation coherence. Even though in theory this can be achieved via [14], in our current implementation we simply use a procedural method to ensure real-time performance. For example, water waves are often simulated in games via a planar base mesh with procedural, sinusoidal-like vertex motions plus further mesh subdivision for rendering. In our system we could simply use this base mesh as the S mesh and texture-map the beam tracing results back onto the subdivided mesh. Similar strategies could be applied for other kinds of simulation techniques in games and interactive applications.

7.2 Beam Tree Construction

We build a beam tree via a method similar to [8]; for each primary beam B that interacts with a S mesh triangle $\triangle Q_1Q_2Q_3$, we derive the spawned secondary beam(s) with projection direction(s) $\{\overrightarrow{d_{ij}}\}$ for each $\overrightarrow{Q_i}$ with respect to B. Note that unlike [8] which handles only linear beams and consequently each $\overrightarrow{Q_i}$ has a unique value $\overrightarrow{d_i}$, we have deal with general nonlinear beam interactions which may produce multiple values $\{\overrightarrow{d_{ij}}\}$. The reason behind this is simple; similar to the diverse situations that might happen when a scene triangle is projected onto a nonlinear beam (as illustrated Figure 2), a secondary beam vertex $\overrightarrow{Q}$ might also have multiple projections onto the primary beam B (e.g. Figure 5 right).



simple case complex case 1 complex case 2

Figure 8: Beam construction cases. The secondary beam base triangle is shown as a colored space triangle and the primary beam base is shown in yellow. In the simple case (left) each projection region is triangular and is entirely within the primary beam base (note that it is OK to have multiple projection regions as shown here). In the complex cases a projection region can be non-triangular (middle) or can be clipped (right). Also shown are the triangulations upon the projection regions and the corresponding mappings onto the secondary beam base triangle in space.

To handle all these situations, we have designed a beam construction algorithm as summarized in Algorithm 5. Given a primary beam B and a secondary beam base triangle (i.e. the triangle of a beam that interacts with a primary beam for secondary reflection/refraction effects) $\triangle Q_1Q_2Q_3$, we first compute the projections $\{V_{ij}\}$ of each $\overrightarrow{Q_i}$ onto B. The projections $\{V_{ij}\}$ correspond to vertices of the components of the projection regions from $\triangle Q_1Q_2Q_3$, as illustrated in Figure 8. We group $\{V_{ij}\}$ into disjoint components $\{R_k\}$, cull those

function $\{B'\} \leftarrow$ ConstructBeams($\triangle Q_1Q_2Q_3$, B) // *on CPU*
  // $\triangle Q_1Q_2Q_3$: *secondary beam base triangle*
  // B: *primary beam*
  // $\{B'\}$: *secondary beams*
  $\{B'\} \leftarrow$ empty set
  foreach i = 1,2,3
    $\{V_{ij}\} \leftarrow$ Project($\overrightarrow{Q_i}$, B)
  group $\{V_{ij}\}_{i=1,2,3}$ into disjoint regions $\{R_k\}$
  cull and clip $\{R_k\}$ against beam base triangle $\triangle_b$ of B
  foreach R_k
    if # vertices of R_k equals 3 **and** R_k not clipped
      // *simple case; each* $\overrightarrow{Q_i}$ *has unique* $V_i \in R_k$ *and hence unique* $\overrightarrow{d_i}$
      $B' \leftarrow$ beam with base $\triangle Q_1Q_2Q_3$ and directions $\{\overrightarrow{d_i}\}$
      add B' to $\{B'\}$
    else // *complex case; some* $\overrightarrow{Q_i}$ *may have multiple directions* $\{\overrightarrow{d_{ij}}\}$
      triangle-strip R_k into multiple regions $\{R_{km}\}$ so that each $\overrightarrow{Q} \in \triangle Q_1Q_2Q_3$ has at most one projection within each R_{km}
      handle each R_{km} as the simple case above
    end if
  end for

Algorithm 5: Our beam construction algorithm.

outside the base triangle $\triangle_b$ of B, and clip those crossing the boundary of $\triangle_b$. We then handle each remaining R_k separately. In the simple case where R_k (1) has exactly three vertices and (2) is not clipped (e.g. Figure 8 left), it can be shown (see Appendix B.6) that (1) each $\overrightarrow{Q_i}$ has a unique projection V_i corresponding to one of the vertices of R_k and (2) R_k corresponds to the entire $\triangle Q_1Q_2Q_3$. Consequently the projection direction $\overrightarrow{d_i}$ of $\overrightarrow{Q_i}$ can be computed from V_i relative to the primary beam B. In the complex cases where R_k either (1) has number of vertices other than three (Figure 8 middle) or (2) is clipped (Figure 8 right), it could happen that (1) at least one $\overrightarrow{Q_i}$ projects onto multiple vertices of R_k and consequently has multiple directions $\overrightarrow{d_{ij}}$ defined or (2) R_k corresponds to a curved-boundary subset of $\triangle Q_1Q_2Q_3$. In these cases, we will have to associate $\triangle Q_1Q_2Q_3$ with multiple beams $\{B'\}$ so that each one of them has uniquely defined directions over the corresponding beam base vertices. We achieve this by triangle-stripping R_k into multiple components $\{R_{km}\}$ so that each $\overrightarrow{Q} \in \triangle Q_1Q_2Q_3$ has at most one projection within each R_{km}, and handle R_{km} via our simple case algorithm.

Note several things here. (1) Since our eye-beam is single-perspective, all our secondary beams (for first level reflections and refractions) belong to the simple cases and the complex cases can arise only for third or higher level beams. (2) Our method clips R_k against the boundary of $\triangle_b$, so it guarantees consistency across multiple primary beams via the vertices created through clipping, e.g. when $\triangle Q_1Q_2Q_3$ projects onto the bases of two adjacent primary beams (e.g. Figure 8). (3) Since each R_{km} corresponds to a subset of $\triangle Q_1Q_2Q_3$, this beam splitting process does not require re-building of S meshes or the associated textures as each new beam can be rendered into a subset of the original texture atlases.

7.3 Texture Storage for Multiple Bounces

For single-bounce beam tracing we simply record the rendering results over a texture map parameterized by a S mesh. However, for multi-bounce situations (e.g. multiple mirrors reflecting each other), one texture map might not suffice as each beam base triangle might be visible from multiple beams. For the worst case scenario where M beams can all see each other, the number of different textures can be $O(M^n)$ for n-bounce beam tracing. Fortunately, it can be easily proven that the

maximum amount of texture storage is $O(Mn)$ if we render the beam tree in a depth-first order. In our current implementation we pack all beams belonging to one (bounce) level into one large texture, and keep n textures for n-bounce beam tracing.

8 Acceleration and Antialiasing

A disadvantage of our approach is that given M scene triangles and N beams, our algorithm would require $O(MN)$ time complexity for geometry processing. Fortunately, this theoretical bound could be reduced in practice by view frustum culling and geometry LOD control, as discussed below. Furthermore, unlike [18], [22], [17], which require fine tessellation of scene geometry for curvilinear projections, our approach achieves this effect even for coarsely tessellated reflector/refractor geometry due to the nature of our method.

Frustum Culling For a linear view frustum, culling can be efficiently performed since each frustum side is a plane. However, in our case, each beam has a nonlinear view frustum where each frustum side is a non-planar surface. Furthermore, for beam bases lying on a concave portion of the reflector the three boundary surfaces might intersect each other, making the sidedness test of traditional culling incorrect. To resolve these issues, we adopt a simple heuristic via a cone which completely contains the original beam viewing frustum. Please see Appendix A for more details. In our current implementation the bounding cones and culling operations are performed per object on a CPU as well as per triangle on a GPU.

Geometry LOD Control Geometry LOD can be naturally achieved by our framework; all we need to do is to estimate proper LOD for each object when rendering a particular beam, and send down the proper geometry. This not only reduces aliasing artifacts but also accelerates our rendering speed. Geometry LOD is more difficult to achieve for ray tracing as it requires storing additional LOD geometry in the scene database.

In our current implementation, we use a simple heuristic by projecting the bounding box of an object onto a beam, and utilize the rasterized projection area to estimate proper LOD.

Anti-aliasing Anti-aliasing is efficiently performed by the GPU in every major step of our system. In addition to the geometry LOD control as mentioned above, we also perform full screen anti-aliasing for each rendered beam. The computed beam results are stored in textures, which in turn are anti-aliased via standard texture filtering.

9 Results and Discussion

9.1 Quality and speed

We compare the quality and speed of our method against previous methods via (1) the BART animation scenes [15], a commonly used benchmark for dynamic ray tracing, and (2) a collection of scenes designed by ourselves that highlight potential usage of our technique in gaming and interactive applications. All results reported here are produced with our S^1 parameterization; the performance of our S^3 parameterization is about one order of magnitude slower.

Figure 9 compares rendering quality of BART animation scenes between ground truth (via ray tracing) and our technique. As shown, even though our technique does not yield identical results to ray tracing, the image quality remains perceptually similar.

	kitchen	robots	museum
Hou et al. [11] NVIDIA 8800 GTX	517 fps (1^{st}) 0.042 fps (2^{nd})	157 fps (1^{st}) 0.06 fps (2^{nd})	647 fps (1^{st}, 10K $\triangle$) 1.76 fps (2^{nd}, 10K $\triangle$) 512 fps (1^{st}, 26K $\triangle$) 0.36 fps (2^{nd}, 26K $\triangle$) 273 fps (1^{st}, 76K $\triangle$) 0.10 fps (2^{nd}, 76K $\triangle$)
Horn et al. [10] (our measurement) NVIDIA 8800 GTX static, with shadow no antialiasing	120 fps (1^{st}) 1.79 fps (2^{nd})	122 fps (1^{st}) 2.92 fps (2^{nd})	130 fps (1^{st}, 10K $\triangle$) 3.51 fps (2^{nd}, 10K $\triangle$) 128 fps (1^{st}, 26K $\triangle$) 2.43 fps (2^{nd}, 26K $\triangle$) 120 fps (1^{st}, 76K $\triangle$) 2.03 fps (2^{nd}, 76K $\triangle$)
Umenhoffe et al. [25] NVIDIA 8800 GTX 1 depth layer only, no shadow	0.63 fps (2^{nd})	0.25 fps (2^{nd})	13.64 fps (2^{nd}, 10K $\triangle$) 6.44 fps (2^{nd}, 26K $\triangle$) 3.02 fps (2^{nd}, 76K $\triangle$)
our method NVIDIA 8800 GTX	517 fps (1^{st}) 7.1 fps (2^{nd})	157 fps (1^{st}) 6.6 fps (2^{nd})	647 fps (1^{st}, 10K $\triangle$) 17.3 fps (2^{nd}, 10K $\triangle$) 512 fps (1^{st}, 26K $\triangle$) 9.36 fps (2^{nd}, 26K $\triangle$) 273 fps (1^{st}, 76K $\triangle$) 3.34 fps (2^{nd}, 76K $\triangle$)

Table 1: Comparison using the BART animation scenes. The technique [10] assumes static scenes. The performance of [25] depends on the number of depth layers (produced via depth peeling) used. Here, we use the fastest possibility of just 1 depth layer, which is usually not enough to resolve all dis-occlusions but the performance is still slower than ours. The performances of the museum scene depend on the number of triangles of the brown object. When possible we also cite the performances according to the different number of ray bounces (primary 1^{st}, secondary 2^{nd}, etc). For all scenes, our technique renders shadows via cubemap shadow maps. All cited algorithms produce image size no larger than ours.

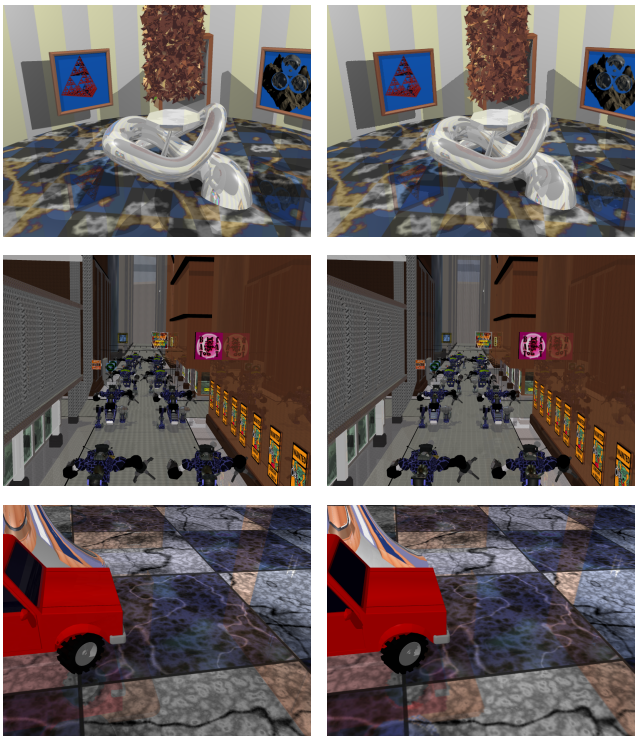
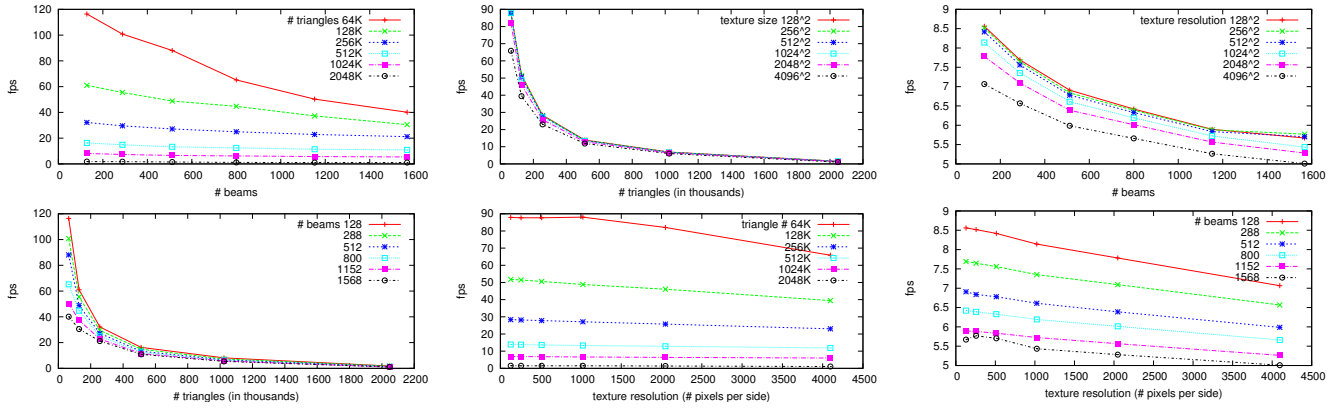


Figure 9: Comparison of BART animation scenes between ground truth via ray tracing (left) and our technique (right). From top to bottom: museum, robot, and kitchen. Please refer to Table 1 for scene statistics.

Table 1 compares performance of BART animation scenes between our approach and a variety of previous techniques, including both ray-based [10], [25] and polygon-based [11]. As shown in Table 1, our technique outperforms [10], one the fastest ray tracers existing today. Note that [10] actually utilizes polygon rasterization for primary rays similar to our



fixed: 1024^2 texture resolution | fixed: 512 beams | fixed: $1024K$ triangles

Figure 10: Performance analysis showing algorithmic dependence on triangle count, beam count, and texture resolution. For each column of figures, we fix one of the three variables while vary the other two.

scene	# scene △	# beam	S texture	FPS	CPU/GPU
Figure 1	78.1K	941	2048^2	11.2	1/88
Figure 11	5.3K	960	1024^2	14.0	1/70
Figure 12 reflection	4.0K	$958{+}2458'$	2048^2	11.0	25/65
Figure 12 refraction	2.9K	$105{+}802'$	2048^2	15.0	11/55
Figure 13 reflection	4.3K	450	1024^2	21.0	1/46
Figure 13 refraction	14.3K	450	1024^2	17.0	1/57
Figure 13 caustics	10.2K	1922	1024^2	16.0	1/61
Figure 13 shadow	28K	800	1024^2	29.0	1/33
kitchen	110.5K	336	1024^2	7.1	16/125
robot	71.7K	52	2048^2	6.6	13/139
museum	26K	1281	512×2048	9.36	2/105

Table 2: Scene statistics. For multiple reflection and refractions we indicate the # of beams per tree level (the root level containing a single eye beam and is not shown for brevity). All frame-rates are measured on an NVIDIA Geforce 8800 GTX with a viewport size 1280×1024. The last column shows the timing breakdown between CPU/GPU in msecs. For fair comparison with other techniques, we have turned off geometry LOD control for timing measurement.

approach. Thus, the fact that our method outperforms [10] on the same hardware via a nontrivial benchmark (BART) clearly demonstrates the advantage of our method. Our method is also faster than ray tracing depth-sprite method [25], which would have problems handling scenes with many depth layers such as reflecting the fractured brown objects upon the mirror tube object in the BART museum scene.

The performance of our algorithm primarily depends on the number of scene triangles and number of beams (vertex workload), as well as the S mesh texture resolutions (pixel workload). (See Table 2 for detailed scene statistics.) Consequently, our performance will be slow if a lot of small beams are rendered. For example, currently we render reflections off the fractured brown object using only a few coarse beams; if we were to render reflection off each fragments via individual beams, our performance will cease to be interactive. Due to the use of polygon-streaming, our performance is relatively insensitive to viewport resolution. Our method is also relatively insensitive to the designed stress factors in BART [15], such as animation (factors 1, 2), coherence (factor 4), object distribution (factors 3, 6, 7), and working set size (factor 5).

To further analyze the performance of our algorithm, we have also measured our algorithmic dependence on triangle count, beam count, and texture resolution via a simplified version of the BART museum scene. As shown in Figure 10, our algorithm performance is most sensitive to number of scene triangles. The performance also depends on number of beams but to a lesser degree, since frustum culling would prevent the worst case $O(MN)$ situation from happening. Finally, our algorithm

Figure 11: Bump map effects via beam tracing. These two images demonstrate different amounts of bumpiness; however, since they share identical δ value, they have the same performance as well.

Figure 12: Multi-bounce beam tracing for multiple reflections (left) and refractions (right). For more results and comparisons see Figure 15.

performance also depends on texture resolution, but to a lesser degree than the number of scene triangles and number of beams.

9.2 Applications

Perturbed beams In addition to smooth surfaces, our technique can also be used to render reflections off bump-mapped surfaces via our perturbed beam parameterization, as demonstrated in Figure 11. However, unlike smooth beams, the overdraw ratio of our perturbed beams can degrade quickly with the increasing of the δ angle as evident in Equation 5. We leave it as future work to devise a better bounding triangle algorithm for perturbed beams.

Multiple beam bounces Unlike [11] which offers only single bounces, our technique allows rendering multiple-bounce reflections and refractions as demonstrated in Figure 12. A potential issue is that the number of beams may increase exponentially with the increasing number of beam bounces. Fortunately, we have found two levels of beam-bounces usually enough, as they provide bigger bang for the buck than subsequent levels. Furthermore, due to the use of depth-first

Figure 13: Dynamic interface with a variety of phenomena. From left to right: a waving lake reflecting seagulls flying by, refraction of a group of fish in the lake, caustics cast over a whale and lake bottom, and nonlinear shadow of a seagull cast onto the lake bottom.

traversal, the memory usage of our algorithm only increases linearly with the level of beam bounces.

Dynamic interface Our technique can also be used to render dynamic reflectors and refractors. Figure 13 shows a waving lake surface reflecting the sky and seagulls, as well as refracting the fishes and lake bottom, plus caustics effects. In this demo, the motion of the water surface is procedurally generated via sinusoidal motions and consequently our S mesh simply moves along with that. We render reflections and refractions via methods previously discussed, and we achieve caustic effects by a two pass algorithm similar to [30], [11]. In the first pass, we render the caustics-receiver coordinates via our nonlinear beam tracing, and use this information for photon gathering in a second pass. In this demo, we utilize the image-space gathering algorithm as in [30].

Nonlinear shadow A nonlinear shadow happens when an object cast shadows through a reflection or refraction interface, such as a flying bird over a water surface. Nonlinear shadows can be easily generated as a byproduct of caustics as described above. Unlike caustics, however, we have found that the nonlinear shadow is most visible when the reflection or refraction interface is relatively smooth; otherwise the shadow will be too fragmented to be noticed. An example of nonlinear shadow generated by our approach is shown in Figure 13 .

10 Conclusions and Future Work

Recent years have witnessed significant progress of ray tracing acceleration techniques, especially for dynamic scenes. The basic premise of ray tracing is to treat the scene as a (more passive) database and the rays as (more active) queries, and the key to acceleration is on how to organize the scene database and/or ray queries for efficiency. Our approach essentially attempts to swap the active/passive roles of rays and scenes, by treating rays as a more passive entity (e.g. stored in textures) while the scene a more active entity (e.g. streaming instead of database storage). As such, our goal is more of investigating an alternative possibility rather than a definite solution. As shown in our results, our methodology offers both advantages and disadvantages over ray tracing. In particular, our approach is most suitable for rendering reflections/refractions off large/smooth interfaces with at most one or two levels of ray bounces. Beyond that, the excessive number of beams would render our brute force approach too inefficient.

One potential future research area is to look at alternative methods of grouping rays into beams. Specifically, instead of ray origins as in our current approach, we could group rays based on both origins and directions and render them via a method similar to our bump-map algorithm. This can be considered as a form of acceleration for the ray engine [2].

Acknowledgement We would like to thank Baining Guo for his advice on this project, Yuan Tian, Junzhi Lu, and other artists for their help on building models and animations, Ligang Liu for answering questions on C^n continuity parameterizations, Daniel Horn for answering questions about his GPU ray tracer, Pat Brown for fixing the vertex program length issue in NVIDIA OpenGL driver (version 97.02), Shay Barlow for lending his voice, Dwight Daniels for help on writing, and the anonymous reviewers for their comments. Environment maps courtesy of NVIDIA Corporation (http://www.codemonsters.de/html/textures_cubemaps.html).

References

[1] Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering, 2nd edition*. A.K. Peters Ltd., 2002.

[2] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02*, pages 37–46, 2002.

[3] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Fredo Durand. Implicit visibility and antiradiance for interactive global illumination. In *SIGGRAPH '07*, page 61, 2007.

[4] Pau Estalella, Ignacio Martin, George Drettakis, and Dani Tost. A gpu-driven algorithm for accurate interactive reflections on curved objects. In *EGSR '06*, June 2006.

[5] G Farin. Triangular bernstein-bezier patches. *Comput. Aided Geom. Des.*, 3(2):83–127, 1986.

[6] Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier, and Bernard Peroche. Fast non-linear projections using graphics hardware. In *I3D '08*, 2008.

[7] Olivier Genevaux, Fredericric Larue, and Jean-Michel Dischler. Interactive refraction on complex static geometry using spherical harmonics. In *I3D '06*, pages 145–152, 2006.

[8] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84*, pages 119–127, 1984.

[9] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96*, pages 99–108, 1996.

[10] Daniel Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07*, 2007.

[11] Xianyou Hou, Li-Yi Wei, Harry Shum, and Baining Guo. Real-time multi-perspective rendering on graphics hardware. In *EGSR '06*, pages 93–102, June 2006.

[12] Ivo Ihrke, Gernot Ziegler, Art Tevs, Christian Theobalt, Marcus Magnor, and Hans-Peter Seidel. Eikonal rendering: efficient light transport in refractive objects. In *SIGGRAPH '07*, page 59, 2007.

[13] Alexander Keller and Per Christensen. *Symposium on Interactive Ray Tracing 2007*. Sep 2007.

[14] Scott Kircher and Michael Garland. Editing arbitrarily deforming surface animations. In *SIGGRAPH '06*, page 71, 2006.

[15] Jonas Lext, Ulf Assarsson, and Tomas Möller. A benchmark for animated ray tracing. *IEEE Comput. Graph. Appl.*, 21(2):22–31, 2001.

[16] Brandon Lloyd, Naga K. Govindaraju, David Tuft, Steve Molnar, and Dinesh Manocha. Practical logarithmic rasterization for low-error shadow maps. In *Graphics Hardware '07*, 2007.

[17] C. Mei, V. Popescu, and E. Sacks. A hybrid backward-forward method for interactive reflections. In *Second International Conference on Computer Graphics Theory and Applications*, 2007.

[18] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *SIGGRAPH '98*, pages 333–342, 1998.

[19] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *EGSR '07*, June 2007.

[20] V. Popescu, C. Mei, J. Dauble, and E. Sack. Reflected-scene impostors for realistic reflections at interactive rates. *EG '06*, 25(3), 2006.
[21] Voicu Popescu, Paul Rosen, and Nicoletta Adamo-Villani. The graph camera. In *SIGGRAPH Asia '09*, pages 1–8, 2009.
[22] David Roger and Nicolas Holzschuch. Accurate specular reflections in real-time. *EG '06*, 25(3), 2006.
[23] Tiago Sousa. Generic refraction simulation. In *GPU Gems II*, pages 295–305, 2005.
[24] Carsten Stoll and Hans-Peter Seidel. Incremental raycasting of piecewise quadratic surfaces on the gpu. In *Symposium on Interactive Raytracing 2006*, pages 141–150, 2006.
[25] Tamas Umenhoffer, Gustavo Patow, and Laszlo Szirmay-Kalos. Robust multiple specular reflections and refractions. In *GPU Gems 3*, 2007.
[26] Alex Vlachos, Jorg Peters, Chas Boyd, and Jason L. Mitchell. Curved pn triangles. In *I3D '01*, pages 159–166, 2001.
[27] Ingo Wald, Thiago Ize, and Steven G. Parker. Special section: Parallel graphics and visualization: Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Comput. Graph.*, 32(1):3–13, 2008.
[28] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *EG '07 STAR*, 2007.
[29] Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE 2005*, 2005.
[30] Chris Wyman and Scott Davis. Interactive image-space techniques for approximating caustics. In *I3D '06*, pages 153–160, 2006.
[31] Jingyi Yu, Jason Yang, and Leonard McMillan. Real-time reflection mapping with parallax. In *I3D '05*, pages 133–138, 2005.

APPENDIX A
DETAILED MATH FORMULAS

Bounding Cone The cone center $\overrightarrow{P_C}$, direction $\overrightarrow{d_C}$, and angle θ_C are computed as follows:

$$\begin{aligned}\overrightarrow{d_C} &= \frac{(\widehat{d_1}+\widehat{d_2}+\widehat{d_3})}{\|(\widehat{d_1}+\widehat{d_2}+\widehat{d_3})\|}\\ \theta_C &= \arccos(\min(\widehat{d_1}\cdot\overrightarrow{d_C}, \widehat{d_2}\cdot\overrightarrow{d_C}, \widehat{d_3}\cdot\overrightarrow{d_C}))\\ \overrightarrow{P_C} &= \overrightarrow{P_G} - \overrightarrow{d_C}\cdot\frac{\max(\|\overrightarrow{O_P}-\overrightarrow{P_G}\|, \forall\overrightarrow{P}\in\triangle P_1P_2P_3)}{\sin\theta_C}\end{aligned}$$

where $\{\widehat{d_1}, \widehat{d_2}, \widehat{d_3}\}$ is the set of extremal directions and $\overrightarrow{P_G}$ is the barycenter of beam base $\triangle P_1P_2P_3$. It can be easily proven that this cone properly contains every point $\overrightarrow{Q}$ inside the original view frustum and it works for both S^n and bump-map parameterizations, as detailed in Appendix B.7.

Once the bounding cone is computed, we can use it for frustum culling as follows. For each scene object, we compute a bounding sphere with center $\overrightarrow{O}$ and radius R, and use the following steps to judge if the sphere and cone intersect. First, we check if $\overrightarrow{P_C}$ is inside the sphere. If not, let $\theta_S = \arcsin(\frac{R}{|\overrightarrow{O}-\overrightarrow{P_C}|})$, which is the critical angle formed between $\overrightarrow{O}$-$\overrightarrow{P_C}$ and the cone boundary when the sphere and cone barely touch each other. Then, it naturally follows that the sphere and cone intersect each other if and only if

$$\frac{\overrightarrow{O}-\overrightarrow{P_C}}{\|\overrightarrow{O}-\overrightarrow{P_C}\|}\cdot\overrightarrow{d_C} > \cos(\theta_C+\theta_S)$$

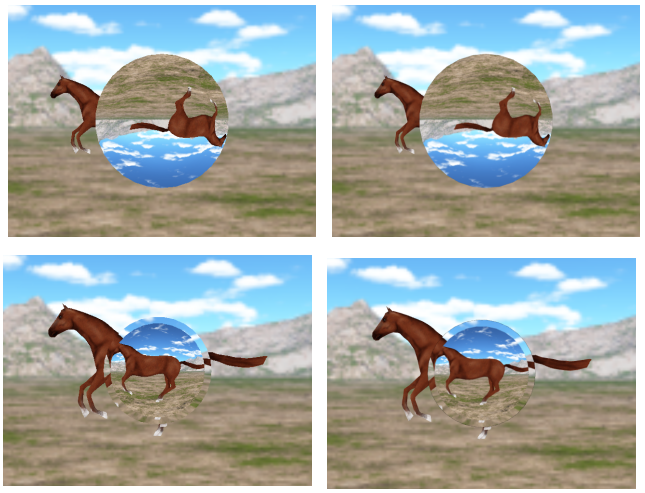
S^3 parameterization

Our formula for S^3 is derived directly from [26]:

$$\begin{aligned}&\overrightarrow{L_{ijk}}(\overrightarrow{X_1},\overrightarrow{X_2},\overrightarrow{X_3}) = \frac{3!}{i!j!k!}\overrightarrow{c_{ijk}}(\overrightarrow{X_1},\overrightarrow{X_2},\overrightarrow{X_3})\\ &\overrightarrow{c_{300}} = \overrightarrow{X_1}, \overrightarrow{c_{030}} = \overrightarrow{X_2}, \overrightarrow{c_{003}} = \overrightarrow{X_3}\\ &w_{ij} = (\overrightarrow{X_j}-\overrightarrow{X_i})\cdot\overrightarrow{n_i},\ \overrightarrow{n_i} \text{ indicates normal at } \overrightarrow{P_i}\\ &\overrightarrow{c_{210}} = (2\overrightarrow{X_1}+\overrightarrow{X_2}-w_{12}\overrightarrow{n_1})/3\\ &\overrightarrow{c_{120}} = (2\overrightarrow{X_2}+\overrightarrow{X_1}-w_{21}\overrightarrow{n_2})/3\\ &\overrightarrow{c_{021}} = (2\overrightarrow{X_2}+\overrightarrow{X_3}-w_{23}\overrightarrow{n_2})/3\\ &\overrightarrow{c_{012}} = (2\overrightarrow{X_3}+\overrightarrow{X_2}-w_{32}\overrightarrow{n_3})/3\end{aligned}$$

ray tracing our result

Figure 14: Comparison of bump mapping effects between ground truth and our technique.



ray tracing our result

Figure 15: Comparison of our beam tree construction strategies. In this example we render multiple refractions through a convex (top) and concave (bottom) lens. Note that our method produces similar results to ray tracing for both convex and concave refractions. The convex case will trigger complex cases for our beam construction, while the concave case will result in mostly simple cases (except beams near the lens boundary).

$$\begin{aligned}&\overrightarrow{c_{102}} = (2\overrightarrow{X_3}+\overrightarrow{X_1}-w_{31}\overrightarrow{n_3})/3\\ &\overrightarrow{c_{201}} = (2\overrightarrow{X_1}+\overrightarrow{X_3}-w_{13}\overrightarrow{n_1})/3\\ &\overrightarrow{e} = (\overrightarrow{c_{210}}+\overrightarrow{c_{120}}+\overrightarrow{c_{021}}+\overrightarrow{c_{012}}+\overrightarrow{c_{102}}+\overrightarrow{c_{201}})/6\\ &\overrightarrow{v} = (\overrightarrow{X_1}+\overrightarrow{X_2}+\overrightarrow{X_3})/3\\ &\overrightarrow{c_{111}} = \overrightarrow{e}+(\overrightarrow{e}-\overrightarrow{v})/2\end{aligned} \quad (6)$$

It can be easily shown that this formulation satisfies Equations 1 and 2.

APPENDIX B
MATH PROOFS

Here, we provide detailed math proofs for the correctness of our algorithms presented earlier.

B.1 Affine plane

Claim B.1: Given a beam base triangle $\triangle P_1P_2P_3$ on base plane Π_b and an associated affine triangle $\triangle H_1H_2H_3$ on an affine plane Π_a

where $\triangle P_1P_2P_3$ and $\triangle H_1H_2H_3$ share the same set of ray directions $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$. Denote $\Omega(P, \triangle)$ as the barycentric coordinates of a point P on a triangle $\triangle$, and $\Omega^{-1}(\{\omega_k\}_{k=1,2,3}, \triangle)$ the inverse operation of computing a point from a set of barycentric coordinates $\{\omega_k\}_{k=1,2,3}$ and a triangle $\triangle$. Then for any scene point $\overrightarrow{Q}$ with a projection $\overrightarrow{H}$ on Π_a there exists point $\overrightarrow{P}$ on Π_b where (1) $\overrightarrow{P}$ is a projection of $\overrightarrow{Q}$ on Π_b and (2) $\Omega(\overrightarrow{P}, \triangle P_1P_2P_3) = \Omega(\overrightarrow{H}, \triangle H_1H_2H_3)$.

Proof: Let $\overrightarrow{Q}$ be a scene point with a projection $\overrightarrow{H}$ on $\Pi_a(s)$ with parameter s. Then from Equation 1 and 2 and the fact that $\overrightarrow{H_i} = \overrightarrow{P_i} + s\overrightarrow{d_i}$, we have

$$\begin{aligned}
\overrightarrow{Q} &= \overrightarrow{O_H} + t\overrightarrow{d},\ \overrightarrow{O_H} \text{ is the ray origin for } \overrightarrow{H} \\
&= \sum_{i+j+k=n} \left(\overrightarrow{L_{ijk}}(\overrightarrow{H_1}, \overrightarrow{H_2}, \overrightarrow{H_3}) + t\overrightarrow{L_{ijk}}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3})\right) \omega_1^i\omega_2^j\omega_3^k \\
&= \sum_{i+j+k=n} \left(\overrightarrow{L_{ijk}}(\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}) + (t+s)\overrightarrow{L_{ijk}}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3})\right) \omega_1^i\omega_2^j\omega_3^k \\
&= \overrightarrow{O_P} + (t+s)\overrightarrow{d},\ \overrightarrow{O_P} \text{ is the ray origin for } \overrightarrow{P}
\end{aligned}$$

Obviously, P is a projection of $\overrightarrow{Q}$ on Π_b and $\Omega(\overrightarrow{P}, \triangle P_1P_2P_3) = \Omega(\overrightarrow{H}, \triangle H_1H_2H_3)$. □

Claim B.2: For an arbitrary space point $\overrightarrow{Q}$ with projection P within the beam base triangle where $\overrightarrow{Q} = \overrightarrow{O_P} + s\overrightarrow{d}$, there exists an affine triangle $\triangle_a(s)$ which contains a point $\overrightarrow{H}$ with corresponding ray origin $\overrightarrow{O_H} = \overrightarrow{Q}$.

Proof: Simply set $t = 0$ in the proof for Claim B.1. □

B.2 Threshold plane

Claim B.3: For a beam with S^1 parameterization, any space point $\overrightarrow{Q}$ is guaranteed to have a unique projection within the beam base triangle $\triangle P_1P_2P_3$ if $\overrightarrow{Q}$ is (1) within the beam frustum and (2) above the threshold plane Π_T.

Proof: From the property of scalar triple product, we know that

$$\overrightarrow{d_2} \times \overrightarrow{d_3} \cdot \overrightarrow{d_1} = \overrightarrow{d_3} \times \overrightarrow{d_1} \cdot \overrightarrow{d_2} = \overrightarrow{d_1} \times \overrightarrow{d_2} \cdot \overrightarrow{d_3}$$

Using the property of quadratic function, we know that $f_1(t)$, $f_2(t)$ and $f_3(t)$ are all positive or are all negative when $t > t_{res}$. Then we can deduce that $\overrightarrow{P_{1,t}} + \overrightarrow{d_1}$, $\overrightarrow{P_{2,t}} + \overrightarrow{d_2}$ and $\overrightarrow{P_{3,t}} + \overrightarrow{d_3}$ on the same side of the affine plane $\Pi_a(t)$. Using similar arguments, we can prove that for any $\Delta t \neq 0$, we have $\overrightarrow{P_{1,t}} + \Delta t \cdot \overrightarrow{d_1}$, $\overrightarrow{P_{2,t}} + \Delta t \cdot \overrightarrow{d_2}$ and $\overrightarrow{P_{3,t}} + \Delta t \cdot \overrightarrow{d_3}$ on the same side of the affine plane $\Pi_a(t)$.

According to Claim B.2, to prove that any space point $\overrightarrow{Q}$ has a unique projection on beam base where $\overrightarrow{Q}$ is (1) within the beam frustum and (2) above the threshold plane Π_T, all we need to prove is that only one unique affine triangle that contains $\overrightarrow{Q}$. From Claim B.2, there must exists an affine triangle $\triangle_a(t_1)$ with $t_1 > t_{res}$ that contains $\overrightarrow{Q}$. We now only need to prove that it is impossible to have another affine triangle $\triangle_a(t_2)$ with $t_2 \neq t_1$ that also contains $\overrightarrow{Q}$.

Assuming the contrary that $\triangle_a(t_2)$ contains $\overrightarrow{Q}$. Since $\triangle_a(t_1)$ and $\triangle_a(t_2)$ are both affine triangles, we have

$$\overrightarrow{P_{i,t_2}} = \overrightarrow{P_{i,t_1}} + (t_2 - t_1) \cdot \overrightarrow{d_i}, i = 1, 2, 3$$

Since $t_1 > t_{res}$, we know from above that $\overrightarrow{P_{1,t_1}} + (t_2 - t_1) \cdot \overrightarrow{d_1}$, $\overrightarrow{P_{2,t_1}} + (t_2 - t_1) \cdot \overrightarrow{d_2}$ and $\overrightarrow{P_{3,t_1}} + (t_2 - t_1) \cdot \overrightarrow{d_3}$ are on the same side of $\Pi_a(t_1)$. However, since $\overrightarrow{Q}$ can be expressed as a non-negative barycentric interpolation from $P_{i,t_2}, i = 1, 2, 3$, it must be outside $\Pi_a(t_1)$. This results in a contradiction. □

Note that we do not need a similar claim for general S^n parameterization to prove the correctness of Algorithm 4; see the proof in Appendix B.4 for more details.

B.3 Maximum offsets

Similar to μ_b for the base triangle $\triangle_b$, we could define a maximum offset μ_t for an affine triangle $\triangle_a$ where t is the value of $\triangle_a$ relative to $\triangle_b$:

$$\mu_t = \max(\|\overrightarrow{O_{P_t}} - \overrightarrow{P_t}\|, \forall \overrightarrow{P_t} \in \triangle_a) \tag{7}$$

Claim B.4:

$$\mu_t \leq \mu_b + t\mu_d$$

Proof: Let $\overrightarrow{P_t}$ be any point on Π_a who shares indentical barycentric coordinates with $\overrightarrow{P}$ on Π_b. From Equation 1, we know

$$\overrightarrow{O_{P_t}} = \sum_{i+j+k=n} \overrightarrow{a_{ijk}}(t)\omega_1^i\omega_2^j\omega_3^k$$

Since

$$\begin{aligned}
\overrightarrow{a_{ijk}}(t) &= \overrightarrow{L}(\overrightarrow{P_1} + t\overrightarrow{d_1}, \overrightarrow{P_2} + t\overrightarrow{d_2}, \overrightarrow{P_3} + t\overrightarrow{d_3}) \\
&= \overrightarrow{L}(\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}) + t\overrightarrow{L}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}) \\
&= \overrightarrow{a_{ijk}} + t\overrightarrow{b_{ijk}}
\end{aligned}$$

We have

$$\begin{aligned}
\overrightarrow{O_{P_t}} &= \sum_{i+j+k=n} \overrightarrow{a_{ijk}}\omega_1^i\omega_2^j\omega_3^k + t\sum_{i+j+k=n} \overrightarrow{b_{ijk}}\omega_1^i\omega_2^j\omega_3^k \\
&= \overrightarrow{O_P} + t\overrightarrow{d_P}
\end{aligned}$$

From above and Equation 4, we know

$$\begin{aligned}
\|\overrightarrow{O_{P_t}} - \overrightarrow{P_t}\| &= \|\overrightarrow{O_P} + t\overrightarrow{d_P} - \sum_{i=1}^{3} \omega_i(\overrightarrow{P_i} + t\overrightarrow{d_i})\| \\
&\leq \|\overrightarrow{O_P} - P\| + t\|\overrightarrow{d_P} - \sum_{i=1}^{3} \omega_i\overrightarrow{d_i}\| \\
&\leq \mu_b + t\mu_d
\end{aligned}$$

Consequently we have $\mu_t \leq \mu_b + t\mu_d$. □

B.4 Bounding triangle for S^n parameterization

Claim B.5: Let $\overrightarrow{P}$ and $\{\overrightarrow{X_i}\}_{i=1:n}$ be points on the same plane containing triangle $\triangle P_1P_2P_3$. Then

$$\overrightarrow{P} = \sum_{i=1}^{n} k_i\overrightarrow{X_i} \leftrightarrow \Omega(\overrightarrow{P}) = \sum_{i=1}^{n} k_i\Omega(\overrightarrow{X_i})$$

where $\sum_{i=1}^{n} k_i = 1$ and $\Omega(\overrightarrow{P})$ indicates the barycentric coordinates of $\overrightarrow{P}$ with respect to $\triangle P_1P_2P_3$. ($\Omega(\overrightarrow{P})$ is a shorthand of $\Omega(\overrightarrow{P}, \triangle P_1P_2P_3)$ for simplicity.)

Proof: First, we prove the $\rightarrow$ direction. From above, we know that

$$\begin{aligned}
\sum_{i=1}^{3} \Omega(\overrightarrow{P})_i\overrightarrow{P_i} &= \overrightarrow{P} = \sum_{j=1}^{n} k_j\overrightarrow{X_j} = \sum_{j=1}^{n}\sum_{i=1}^{3} k_j\Omega(\overrightarrow{X_j})_i\overrightarrow{P_i} \\
&= \sum_{i=1}^{3} \left(\sum_{j=1}^{n} k_j\Omega(\overrightarrow{X_j})_i\right)\overrightarrow{P_i}
\end{aligned} \tag{8}$$

From the uniqueness property of barycentric coordinates $\{\Omega(\overrightarrow{P})_i\}_{i=1,2,3}$ under non-degenerated configurations of $\{\overrightarrow{P_i}\}_{i=1,2,3}$ we know

$$\Omega(\overrightarrow{P})_i = \sum_{j=1}^{n} k_j\Omega(\overrightarrow{X_j})_i \text{ for i = 1,2,3}$$

Thus

$$\Omega(\overrightarrow{P}) = \sum_{j=1}^{n} k_j\Omega(\overrightarrow{X_j})$$

and the $\rightarrow$ direction is proven.

Now we prove the $\leftarrow$ direction as follows:

$$\begin{aligned}
\overrightarrow{P} &= \sum_{i=1}^{3} \Omega(\overrightarrow{P})_i\overrightarrow{P_i} = \sum_{i=1}^{3} \left(\sum_{j=1}^{n} k_j\Omega(\overrightarrow{X_j})_i\right)\overrightarrow{P_i} \\
&= \sum_{j=1}^{n} k_j \sum_{i=1}^{3} \Omega(\overrightarrow{X_j})_i\overrightarrow{P_i} = \sum_{j=1}^{n} k_j\overrightarrow{X_j}
\end{aligned} \tag{9}$$

□

Proof for general case Here we prove the general case of our algorithm (shown in Algorithm 4).

Claim B.6: Let $P \in \triangle P_1P_2P_3$ be a projection of a space point $\overrightarrow{Q} \in$ scene triangle $\triangle Q_1Q_2Q_3$. Then there exist k_i for $i = 1, 2, 3$ with $0 \leq k_i$ and $\sum_{i=1,2,3} k_i = 1$ so that

$$\overrightarrow{P} = \sum_{i=1}^{3} k_i \overrightarrow{F_i}$$

where the set $\{\overrightarrow{F_i}\}$ are the vertices of the expanded bounding triangle $\tilde{\triangle}$ as computed in our general case algorithm (Algorithm 4).

Proof: Since $\overrightarrow{P}$ is a projection of $\overrightarrow{Q}$ inside the base triangle, obviously there exists t so that $\overrightarrow{Q} = \overrightarrow{O_P} + t\overrightarrow{d_P}$. For simplicity, let's denote $\overrightarrow{d_P}$ as $\overrightarrow{d}$ below. Let $\overrightarrow{P_{i,\overrightarrow{d}}}$ be the transfer of $\overrightarrow{Q_i}$ in $\overrightarrow{d}$. Let $\triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon}$ be the ϵ expansion of $\triangle P_{1,\overrightarrow{d}} P_{2,\overrightarrow{d}} P_{3,\overrightarrow{d}}$ where ϵ is computed as in function ExpandedBoundingTriangle() in Algorithm 4. From the definition of ϵ and Claim B.4, we know

$$\overrightarrow{P} = \sum_{j=1}^{3} \Omega(P, \triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon})_j \overrightarrow{P_{j,\overrightarrow{d},\epsilon}} \tag{10}$$

where $\Omega(P, \triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon})_j$ all ≥ 0 and sum to 1.

Furthermore, since $\overrightarrow{d}$ is bound within the directions $\{\widehat{d_1}, \widehat{d_2}, \widehat{d_3}\}$, we have

$$\overrightarrow{P_{j,\overrightarrow{d}}} = \sum_{i=1}^{3} \Omega(P_{j,\overrightarrow{d}}, \triangle P_{j1}P_{j2}P_{j3})_i \overrightarrow{P_{ji}} \tag{11}$$

where $\Omega(P_{j,\overrightarrow{d}}, \triangle P_{j1}P_{j2}P_{j3})_j$ all ≥ 0 and sum to 1 if $\{\overrightarrow{Q_1}, \overrightarrow{Q_2}, \overrightarrow{Q_3}\}$ all inside the beam, and $\overrightarrow{P_{ji}}$ is the transfer of $\overrightarrow{Q_j}$ in direction $\widehat{d_i}$.

From Equation 11 and the fact that $\triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon}$ is an ϵ expansion of $\triangle P_{1,\overrightarrow{d}} P_{2,\overrightarrow{d}} P_{3,\overrightarrow{d}}$ and $\triangle F_1F_2F_3$ is an ϵ expansion of the bounding triangle for $\{\overrightarrow{P_{ji}}\}_{i,j=1,2,3}$, we have

$$\overrightarrow{P_{j,\overrightarrow{d},\epsilon}} = \sum_{i=1}^{3} \Omega(P_{j,\overrightarrow{d},\epsilon}, \triangle F_1F_2F_3)_i \overrightarrow{F_i} \tag{12}$$

where $\Omega(P_{j,\overrightarrow{d},\epsilon}, \triangle F_1F_2F_3)_i$ all ≥ 0 and sum to 1.

Combining Equations 10 and 12, we have

$$\overrightarrow{P} = \sum_{i=1}^{3}\sum_{j=1}^{3} \Omega(P, \triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon})_j \Omega(P_{j,\overrightarrow{d},\epsilon}, \triangle F_1F_2F_3)_i \overrightarrow{F_i}$$

Thus we can have $k_i = \sum_{j=1}^{3} \Omega(P, \triangle P_{1,\overrightarrow{d},\epsilon} P_{2,\overrightarrow{d},\epsilon} P_{3,\overrightarrow{d},\epsilon})_j$. □

Consequently, from Claim B.6 we know $\triangle F_1F_2F_3$ is a proper bounding triangle.

Proof for unique projection case

We first prove the $\mathbf{S^1}$ case. Let H be the projection on any affine plane Π_a of a space point $\overrightarrow{Q}$ *in* $\triangle Q_1Q_2Q_3$ where $\triangle Q_1Q_2Q_3$ is above the threshold plane Π_T. From Claim B.3, we know that $\overrightarrow{Q}$ has a unique projection within the beam base triangle.

From the proof for the general case, we know there exists a set $\{k_i\}_{i=1,2,3}$, $0 \leq k_i$ and $\sum k_i = 1$, so that $\overrightarrow{H} = \sum_{i=1,2,3} k_i \overrightarrow{E_i}$ where $\overrightarrow{E_i}$ is computed as in our unique projection case algorithm (Algorithm 4). Note that the proof for general case only holds when $\{\widehat{d_1}, \widehat{d_2}, \widehat{d_3}\}$ all point to the same side of Π_a; this is only guaranteed when Π_a is above the threshold plane Π_T. This is why our unique projection case algorithm only works for $\overrightarrow{Q}$ above Π_T.

Let $P = \Omega^{-1}(\Omega(H, \Pi_a), \Pi_b)$. From Claim B.1, we know $\overrightarrow{P}$ is the projection of $\overrightarrow{Q}$ onto the base plane Π_b. From Claim B.5 we know

$$\overrightarrow{P} = \sum_{i=1}^{3} k_i \overrightarrow{F_i}$$

where $\overrightarrow{F_i}$ is computed as described in our unique projection case algorithm in Algorithm 4. Consequently, we know $\triangle F_1F_2F_3$ is a proper bounding triangle.

We now prove the $\mathbf{S^n}$ case. For the same space point $\overrightarrow{Q}$ as above, there exists potentially multiple (but at least one) points $\{\overrightarrow{X_i}\}$ where each $\overrightarrow{X_i}$ is within one affine triangle $\triangle_a(t_i)$ and has an ray origin $\overrightarrow{O_{X_i}} = \overrightarrow{Q}$. From our definitions of threshold plane, it can be easily seen that each $t_i \geq t_{max}$. (If $t_i < t_{max}$, then $\|\overrightarrow{O_{X_i}} - \overrightarrow{X_i}\| < \mu_b + t_{max}\mu_d$ according to Claim B.4. Consequently, $\overrightarrow{Q} = \overrightarrow{O_{X_i}}$ cannot be above $\Pi_a(t_{res}) = \Pi_T$, a contradiction.) Consequently, we know that our unique projection bounding triangle algorithm must work for each one of them (by treating each $\overrightarrow{X_i}$ as a degenerated triangle), under S^1 parameterization as proven above.

Now all we need to do is to show that our S^n bounding triangle computed on an affine plane $\Pi_a(t_{top})$ above $\overrightarrow{Q}$ would include all S^1 projections $\{\overrightarrow{H_{X_i}}\}$ for all $\overrightarrow{X_i}$. Note that by definition $\{\overrightarrow{H_{X_i}}\}$ are projections of $\overrightarrow{Q}$ on $\Pi_a(t_{top})$. This can be proven by the fact that the ϵ expansion of the bounding triangle computation on $\Pi_a(t_{top})$ is large enough to contain all S^1 bounding triangles for each $\overrightarrow{X_i}$. (Note that $\mu_b + t_{top}\mu_d \geq \max(\|\overrightarrow{Q} - \overrightarrow{X_i}\|, \forall i)$.)

B.5 Bounding triangle for bump map parameterization

In the proof below we assume the bump map is applied after a S^1 parameterization as general S^n parameterization is unnecessary.

For a spatial point $\overrightarrow{Q} \in \triangle Q_1Q_2Q_3$, let its non-perturbed projection be $\overrightarrow{P}$ and its perturbed projection be $\overrightarrow{P}'$ where $\overrightarrow{Q} = \overrightarrow{P} + t\overrightarrow{d}$. For bounding triangle proof, we only care for $\overrightarrow{P}'$ which is inside beam base triangle $\triangle P_1P_2P_3$. However, $\overrightarrow{P}$ can be inside or outside $\triangle P_1P_2P_3$. Below, we first prove the case where $\overrightarrow{P} \in \triangle P_1P_2P_3$. We deal with the $\overrightarrow{P} \notin \triangle P_1P_2P_3$ case later.

For the $\overrightarrow{P} \in \triangle P_1P_2P_3$ case, we will prove (1) $\angle PQP' \leq \tau$, (2) $\angle QP'P \geq \theta_3 - \tau$, and (3) $|PQ| \leq \max(\|Q_iP_{ij}\|,\ \forall i, j = 1, 2, 3)$. Since

$$\|PP'\| = \frac{\|PQ\| \cdot \sin \angle PQP'}{\sin \angle QP'P}$$

from the above three conditions we know

$$\|PP'\| \leq \frac{\max(\|Q_iP_{ij}\|,\ \forall i, j = 1, 2, 3) \cdot \sin(\tau)}{\sin(\theta_3 - \tau)} = D$$

For condition (1), it can be easily shown that $\angle PQP' \leq \tau$ from the definition of τ.

For condition (2), all we need to prove is that $\angle QP'P \geq \theta_1 - \tau$ and $\angle QP'P \geq \theta_2 - \tau$. Let's consider $\overrightarrow{n} \cdot \overrightarrow{d}$. From the definition of $\overrightarrow{d_{Q_i}}$, we know

$$\overrightarrow{n} \cdot \overrightarrow{d} \geq \min(\overrightarrow{n} \cdot \overrightarrow{d_{Q_i}},\ i = 1, 2, 3) = sin(\theta_1)$$

Consequently the angle between $\overline{QP}$ and the beam base plane must be $\geq \theta_1$. From trigonomotry, we know $\angle QP'P \geq \theta_1 - \tau$. Following a similar argument we can also show $\angle QP'P \geq \theta_2 - \tau$.

We now prove condition (3). Let $\triangle P_{Q_1,\overrightarrow{d}} P_{Q_2,\overrightarrow{d}} P_{Q_3,\overrightarrow{d}}$ be the parallel projection of $\triangle Q_1Q_2Q_3$ onto the beam base plane in direction $\overrightarrow{d}$. From this, we know $|PQ| \leq \max(\|Q_iP_{Q_i,\overrightarrow{d}}\|,\ i = 1, 2, 3)$. Furthermore, from the fact that $\overrightarrow{d} = \sum_{i=1}^{3} \omega_i \overrightarrow{d_i}$ with $\omega_i \geq 0$, we know $\|Q_iP_{Q_i,\overrightarrow{d}}\| \leq \max(\|Q_i, P_{ij}\|, j = 1, 2, 3)$. Consequently, we have $|PQ| \leq \max(\|Q_iP_{ij}\|,\ \forall i, j = 1, 2, 3)$ as in condition (3).

Since $\overrightarrow{P}$ is inside the bounding triangle $\tilde{\triangle}$ computed by S^n algorithm and $\|PP'\| \leq D$, we know that $\tilde{\triangle}$ expanded by D must contain all perturbed projections $\overrightarrow{P}'$. Thus we complete the proof for the $\overrightarrow{P} \in \triangle P_1P_2P_3$ case.

For the $\overrightarrow{P} \notin \triangle P_1P_2P_3$ case, even though an analytical approach is possible, we have found it too complex and unnecessary. Even though our proof above does not hold for $\overrightarrow{P} \notin \triangle P_1P_2P_3$ in theory, for real applications we have found it work well since our D is already over-conservative. Specifically, when δ is small, $\overrightarrow{P}$ and $\overrightarrow{P}'$ tend to be close to each other so the case is well under cover by D. When δ is large, the reflections or refractions tend to be stochastic anyway so

whether or not the bounding triangle is truly conservative does not have perceivable impact on image quality. See Figure 14 for a quality comparison between ground truth and our technique.

B.6 Beam construction

Here we provide more details for our beam construction algorithm as described in Section 7.2. For clarity, we assume S^1 parameterization in the following expositions but the same principles apply to our S^n parameterization as well.

Given a primary beam B and a secondary beam base triangle $\triangle Q_1Q_2Q_3$, we first compute the projections $\{V_{ij}\}$ of each $\overrightarrow{Q_i}$ onto B. The projections $\{V_{ij}\}$ correspond to vertices of the components of the projection regions from $\triangle Q_1Q_2Q_3$, as illustrated in Figure 8. Note that each vertex $\overrightarrow{Q_i}$ of $\triangle Q_1Q_2Q_3$ can have up to a maximum number of projections as a function of the beam parameterization; for example, the maximum number is 3 for our S^1 parameterization. (In theory, it is possible to have singularity situations where a vertex $\overrightarrow{Q_i}$ have an infinite number of projections, but they are unlikely to happen in real applications both statistically and due to finite precision floating point numbers.)

Our next step is to group $\{V_{ij}\}$ into connected components. We achieve by sorting $\{V_{ij}\}$ according to the corresponding t values; i.e. the t value that connects $\overrightarrow{Q_i}$ with V_{ij} in our parameterization (see Section 4 and Section 5). In the sorted list of $\{V_k\}$ (from $\{V_{ij}\}$) we insert a "cut" between each pair of neighbors V_k and V_{k+1} (with corresponding t values t_k and t_{k+1}) whenever the affline plane $\Pi_a(\frac{t_k+t_{k+1}}{2})$ does not intersect $\triangle Q_1Q_2Q_3$; these "cuts" partitions $\{V_k\}$ into disjoint connected components $\{R_k\}$.

Geometrically, this can be interpreted as an affline plane $\Pi_a(t)$ that passes through $\overrightarrow{Q_i}$ and with V_{ij} determined by the relative barycentric coordinate of $\overrightarrow{Q_i}$ within $\triangle_a(t)$, i.e. $V_{ij} = \Omega^{-1}(\Omega(Q_i, \triangle_a(t)), \triangle_b)$. ($\triangle_b$ is the base triangle of B.) In the simplest case where $\Pi_a(t)$ just "ascends" monotonically away from the base plane Π_b of B with t from 0 to ∞, we will have a single triangular projection region as illustrated in Figure 2. However, in more complex situations $\Pi_a(t)$ may "flip" or "turn around", passing through $\triangle Q_1Q_2Q_3$ multiple times with result patterns shown in Figure 8. Since $\{V_k\}$ has a finite number of vertices (at most 9 for S^1 parameterization), we can have only a finitely number of paritition $\{R_k\}$ as well.

In the simple case where R_k has exactly 3 vertices and is not clipped against $\triangle_b$, we know each point $\overrightarrow{Q} \in \triangle Q_1Q_2Q_3$ can have a most one projection inside R_k as R_k is produced by having $\Pi_a(t)$ passing through $\triangle Q_1Q_2Q_3$ only once. Consequently, we can safely assign a unique projection direction $\overrightarrow{d_i}_{i=1,2,3}$ for each vertex $\overrightarrow{Q_i}_{i=1,2,3}$ of $\triangle Q_1Q_2Q_3$.

Other wise, R_k belongs to the complex cases with either a number of vertices other than 3 or is clipped. The former case signals that $\Pi_a(t)$ passes through multiple times the subset of $\triangle Q_1Q_2Q_3$ corresponding to R_k, causing at least one point $\overrightarrow{Q} \in \triangle Q_1Q_2Q_3$ with more than one projections inside R_k. In the latter case, the clippling line inside R_k may correspond to a curve on $\triangle Q_1Q_2Q_3$. In both cases we cannot assign a unique direction $\overrightarrow{d_i}_{i=1,2,3}$ for each vertex $\overrightarrow{Q_i}_{i=1,2,3}$ of $\triangle Q_1Q_2Q_3$ as the interpolated direction $\overrightarrow{d}$ for the interior points will not be correct.

The theoretically correct solution to tackle the complex cases is to use "line beam" as follows. A line beam is formed from a base as a line with 2 vertices instead of a triangle with 3 vertices as with our ordinary beams. Each vertex of a line beam has an associated direction similar to an ordinary triangle beam. A line beam $B(t)$ is formed as the intersection of $\Pi_a(t)$ with $\triangle Q_1Q_2Q_3$. Due to the special properties of $\Pi_a(t)$, $B(t)$ will project onto a straight line segment in Π_b. For a complex region R_k, we can decompose it into iso-lines corresponding to proper projections of line cameras $\{B(t)\}$, and render these line cameras via our beam tracing framework. Note that unlike triangle beams, line beams are immune to ambiguity or clipping and each interior point of a line beam has a uniquely defined direction.

Even though using line beams is theoretically correct, we have found it too computationally expensive for practical applications. Instead, in our current implementation we approximate the line beams with triangle-strip beams. And with reasonably-tessellated triangle-strips, we have found the image quality to be visually indistinguishable from ground truth. An example is shown in Figure 15. In our current implementation we use a heuristic of 4-texel wide triangles.

As an additional optimization, for each R_k that belongs to the complex cases, we could further decide which sub-regions of R_k are actually ambiguity-free and consequently do not need fine tessellation. This is achieved as follows. When the affine plane $\Pi_a(t)$ sweeps through $\triangle Q_1Q_2Q_3$ with $t \in [t_k^{min}, t_k^{max}]$ that corresponds to R_k, it must intersect $\triangle Q_1Q_2Q_3$ at one vertex or two edges at any given time. Let Q_iQ_j and Q_iQ_k be the two edges of intersection ($i = 1, 2, 3$, $j = (i+1)\ mod\ 3$, $k = (i+2)\ mod\ 3$), and $\overrightarrow{Q_i} + s_1(t)(\overrightarrow{Q_j} - \overrightarrow{Q_i})$ and $\overrightarrow{Q_i} + s_2(t)(\overrightarrow{Q_k} - \overrightarrow{Q_i})$ the two specific intersections where $s_1(t)$ and $s_2(t)$ are two scalars in the range $[0, 1]$. For regions R_k where $s_1(t)$ and $s_2(t)$ move in the same direction (i.e. both increasing or decreasing), there is no projection ambiguity and we could forgo the fine triangle-stripping. We decompose $[t_k^{min}, t_k^{max}]$ into such simple and complex intervals via basic calculus, i.e. the derivatives of $s_1(t)$ and $s_2(t)$. It can be shown that doing so would incur solving quartic polynomials since $s_1(t)$ and $s_2(t)$ have this general rational polynomial form: $\frac{At^3+Bt^2+Ct+D}{Et^2+Ft+G}$.

B.7 Bounding cone

For each point inside the beam viewing frustum $\overrightarrow{Q} = \overrightarrow{O_P} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}$ where $t > 0$, we know that from Equation 6,

$$\|\overrightarrow{P_G} - \overrightarrow{P_C}\| = \frac{\max(\|\overrightarrow{O_P} - \overrightarrow{P_G}\|, \forall \overrightarrow{P} \in \triangle P_1P_2P_3)}{\sin\theta_C}$$

Then

$$\begin{aligned}\sin\angle O_PP_CP_G &\leq \frac{\|\overrightarrow{O_P} - \overrightarrow{P_G}\|}{\|\overrightarrow{P_G} - \overrightarrow{P_C}\|} \\ &= \frac{\|\overrightarrow{O_P} - \overrightarrow{P_G}\|}{\max(\|\overrightarrow{O_P} - \overrightarrow{P_G}\|, \forall \overrightarrow{P} \in \triangle P_1P_2P_3)} \cdot \sin\theta_C \\ &\leq \sin\theta_C\end{aligned}$$

Thus

$$\begin{aligned}(\overrightarrow{O_P} - \overrightarrow{P_C}) \cdot \overrightarrow{d_C} &= \|\overrightarrow{O_P} - \overrightarrow{P_C}\| \cdot \cos\angle\overrightarrow{O_P}P_CP_G \\ &\geq \|\overrightarrow{O_P} - \overrightarrow{P_C}\| \cdot \cos\theta_C \end{aligned} \quad (13)$$

And for θ_C, from Equation 6 we have $\widehat{d_i} \cdot \overrightarrow{d_C} \geq \|\widehat{d_i}\| \cdot \cos\theta_C$

Then

$$\begin{aligned}(\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}) \cdot \overrightarrow{d_C} &\geq \sum_{i=1}^{3}(\widehat{\omega_i}(\|\widehat{d_i}\| \cdot \cos\theta_C)) \\ &\geq (\|\sum_{i=1}^{3}(\widehat{\omega_i}\widehat{d_i})\|) \cdot \cos\theta_C \end{aligned} \quad (14)$$

From Equations (13) and (14), we know that

$$\begin{aligned}&\frac{\overrightarrow{Q} - \overrightarrow{P_C}}{\|\overrightarrow{Q} - \overrightarrow{P_C}\|} \cdot \overrightarrow{d_C} = \frac{\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ = &\frac{\overrightarrow{O_P} - \overrightarrow{P_C}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ &+ \frac{t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ \geq &\frac{\|\overrightarrow{O_P} - \overrightarrow{P_C}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \cos\theta_C \\ &+ \frac{\|t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \cos\theta_C \\ \geq &\frac{\|\overrightarrow{O_P} - \overrightarrow{P_C}\| + \|t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3}\widehat{\omega_i}\widehat{d_i}\|} \cdot \cos\theta_C \\ \geq &\cos\theta_C \end{aligned} \quad (15)$$

As a result, each $\overrightarrow{Q}$ inside the view frustum is also inside the bounding cone.

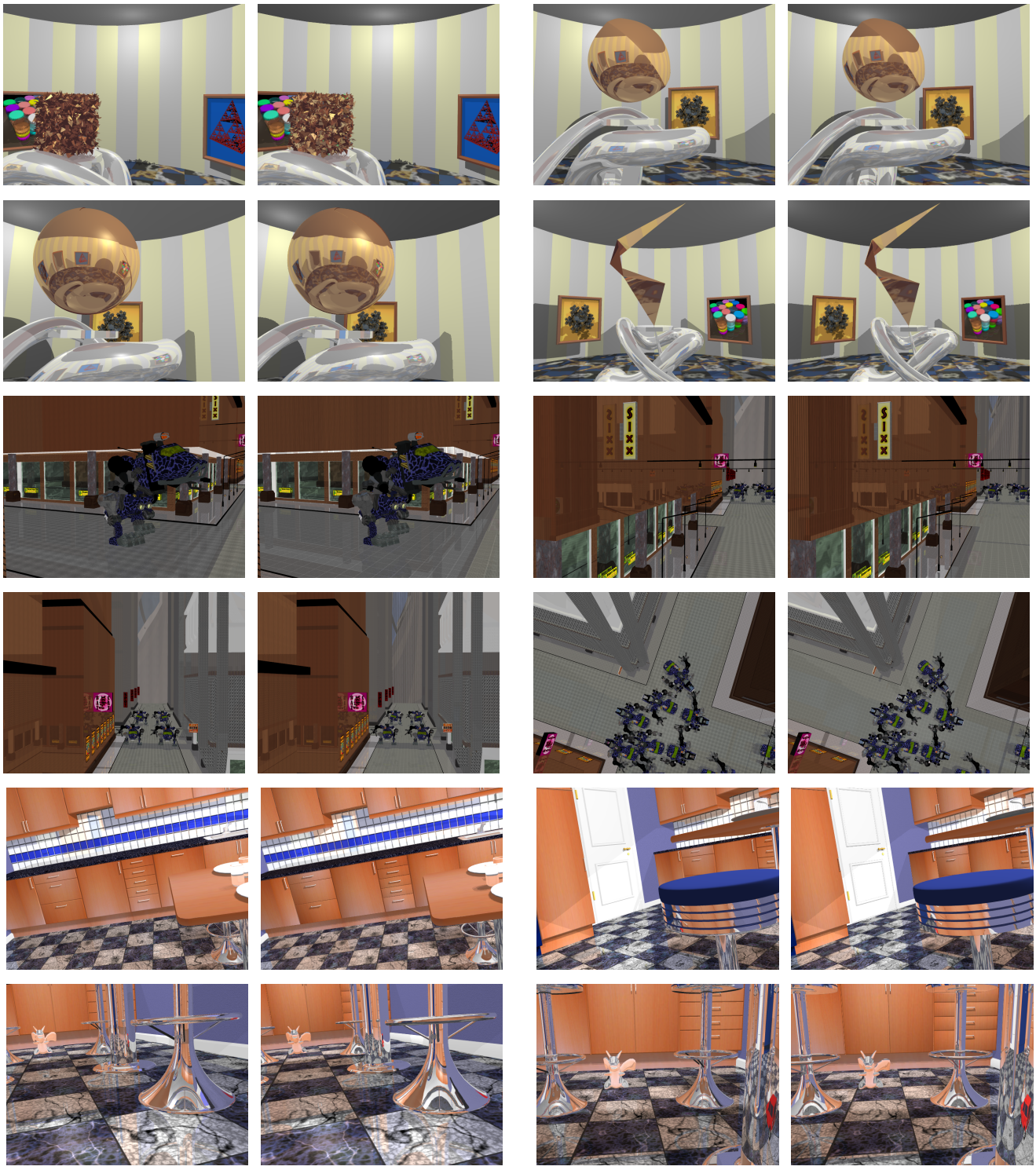


Figure 16: More comparison of BART animation scenes between ground truth via ray tracing (left) and our technique (right). From top to bottom: museum, robot, and kitchen. All rendering parameters are identical to Figure 9.