Nonlinear Beam Tracing on a GPU

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 are difficult to render efficiently via ray tracing but easy for our polygon rasterization based approach. See Section 5 for detailed performance statistics. Furthermore, the image quality generated by our approach is comparable to ray tracing; 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 teapot spout, lid, and handle, rendered under different view points.

Abstract

Beam tracing [Heckbert and Hanrahan 1984] combines the flexibility of ray tracing and the coherence and 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 global illumination 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. A major strength of our algorithm is that it naturally supports fully dynamic scenes as it does not rely on acceleration data structures required for ray tracing. Utilizing our approach, nonlinear global illumination effects can be rendered in real-time on a commodity GPU under a unified framework.

Keywords: nonlinear beam tracing, real-time rendering, GPU techniques

1 Introduction

Beam tracing was proposed by Heckbert and Hanrahan in 1984 [Heckbert and Hanrahan 1984] 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.

In this paper, we introduce nonlinear beam tracing to render nonlinear global illumination 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 our frustum sides are no longer planes, and a fast 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. Second, our algorithm naturally supports fully dynamic scenes since it does not rely on acceleration data structures usually required for ray tracing. Third, we provide a unified framework for rendering a variety of nonlinear global illumination effects

that have often been achieved via individual heuristics in previous graphics hardware rendering techniques.

2 Previous Work

Even though ray tracing computations are inherently parallel, its random-access to a shared scene database is difficult for efficient GPU implementation as it is primarily designed for streaming polygon rasterization [Purcell et al. 2002; Carr et al. 2002]. A further problem is that ray tracing is less suitable for dynamic scenes as most algorithms utilize pre-processed acceleration data structures for static models [Wald et al. 2003]. Beyond that, maintaining interactive frame rates often requires limited motion range/speed for incremental update on acceleration structures [Carr et al. 2006; Woop et al. 2006] or coherent bundles for eye rays and shadow rays only [Wald et al. 2006].

A variety of techniques have been proposed for simulating ray tracing effects suitable for feed-forward polygon rasterization [Akenine-Moller and Haines 2002]; typical example effects include reflection, refraction, and caustics. However, these techniques are often only applicable to various individual phenomena.

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 [Blinn and Newell 1976], planar reflectors [Heckbert and Hanrahan 1984; Diefenbach and Badler 1997; Guy and Soler 2004], static scene [Heidrich et al. 1999; Yu et al. 2005], limited motion for frame-to-frame coherence [Estalella et al. 2006], or reliance on extensive pre-computation for all possible motion parameters [Hakura et al. 2001]. [Szirmay-Kalos et al. 2005; Popescu et al. 2006] approximate near objects as depth imposters from which reflections are rendered via a technique similar to relief mapping [Policarpo and Oliveira 2006]. However, handling disocclusion and combine multiple depth-imposter objects can be difficult issues since they cannot be resolved via hardware z-buffering.

Several techniques exist to render fully dynamic reflections without the depth sprite issues mentioned above. [Ofek and Rappoport 1998; Roger and Holzschuch 2006; Mei et al. 2007] allow curved mirror reflection of near objects with unlimited motion. However, since these techniques require fine tessellation of scene geometry for rendering curvilinear effects, 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 [Heckbert and Hanrahan 1984]. Common techniques often handle only nearly planar refractors [Sousa 2005] or only far away scene objects [Genevaux et al. 2006], but not refraction of near objects through arbitrarily-shaped lens. [Wyman 2005] approximates such effects via image-based heuristics. This is similar to [Szirmay-Kalos et al. 2005], in which the technique represents refracted objects as depth sprites and shares similar problem in depth compositing. Caustics results from light focused by curved reflection or refraction and can be rendered by a standard two-pass process as demonstrated in [Purcell et al. 2003; Wand and Straßer 2003; Wyman and Davis 2006]

Recently, [Hou et al. 2006] has proposed a multi-perspective GPU technique which can render limited nonlinear effects. However, [Hou et al. 2006] 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 contrast, supports full nonlinear beam tracing effects including multiple reflections and 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.

Approximating ray tracing effects on GPU is an endless fascination; for more complete reviews we refer the readers to [Akenine-Moller and Haines 2002; Hou et al. 2006; Estalella et al. 2006; Krüger et al. 2006].

3 Overview of Our Approach

For clarity, we first provide an overview of our algorithm as summarized in Table 1. We present details for the individual parts of our algorithms in Section 4.

function NonlinearBeamTracing()
  $B_0 \leftarrow$ eye beam from viewing parameters
  $\{B_{ij}\} \leftarrow$ BuildBeamTree(B_0) // *on CPU*
  // *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*
  end for

function $\{B_{ij}\} \leftarrow$ BuildBeamTree(B_0)
  trace beam shapes recursively as [Heckbert and Hanrahan 1984]
  but allows nonlinear beams bouncing off curved surfaces
  record boundary rays $\{\overrightarrow{d_{ijk}}\}_{k=1,2,3}$ for each B_{ij}

function RenderBeam(B)
  if perturbation required // *bump map or other sources*
    perturb rays in B
    $\delta \leftarrow$ maximum angular deviation of B
      from original (smooth) beam parameterization
  else
    $\delta \leftarrow 0$
  end if
  foreach scene triangle $\triangle$
    $\tilde{\triangle} \leftarrow$ BoundingTriangle($\triangle$, B, δ) // *vertex shader*
    RenderTriangle($\tilde{\triangle}$, $\triangle$, B) // *fragment shader*
  end for

Table 1: Overview of our system.

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 the last four lines of 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. 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, 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 tex-

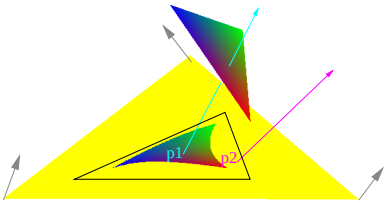

Figure 2: Rendering one scene triangle within one beam. The scene triangle is shown in RGB colors whereas the beam base triangle is shown in yellow. Due to nonlinear projection, the projected region may have curved edges. Our rendering is performed by a combination of a vertex program that estimates 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.

ture 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 *BT* (beam tracing) mesh. The resolution of the BT mesh trades off between rendering quality and speed. We associate a BT mesh with a reflector/refractor interface through either mesh simplification or procedural methods, depending on whether the interface is static (e.g. a mirror teapot) or dynamic (e.g. water waves).

4 Algorithm Details

In this section, we describe details of our algorithm as summarized in Section 3.

4.1 Beam Parameterization

As described in Section 3, we utilize beams with triangular cross sections for easy processing. For each point $\overrightarrow{P}$ 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.

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 $\triangle P_1P_2P_3$, our smooth parameterization of the ray origin and direction for any point P on the beam base plane is defined via the following formula:

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

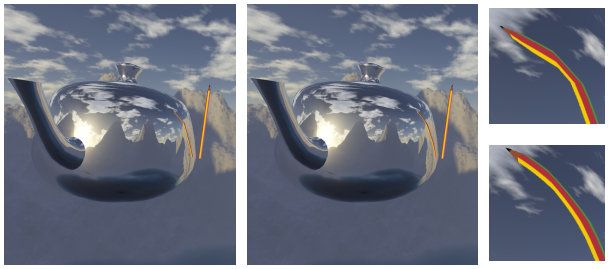

Figure 3: Comparison of beam parameterizations. Left: C^0 parameterization. Middle: S^3 parameterization. Right-top/right-bottom: close up views around the reflections of left/middle. For the C^0 case, notice the sudden direction change of the pencil reflection over the teapot.

$$\overrightarrow{b_{ijk}} = \overrightarrow{L_{ijk}}(\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}) \quad (1)$$

where $\overrightarrow{L}$ is a function satisfying the following linear property for any real number s, $\{\overrightarrow{P_1}, \overrightarrow{P_2}, \overrightarrow{P_3}\}$, and $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$:

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

We term this formulation our S^n parameterization. Note that this is an extension of triangular Bezier patches [Farin 1986] 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 [Hou et al. 2006] (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 [Vlachos et al. 2001] (see Equation 9 under Appendix B.1). Note that in general $\overrightarrow{O} \neq \overrightarrow{P}$ unless under the C^0 parameterization.

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

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 C^0 and S^3 parameterizations. The tradeoff between C^0 and S^3 parameterizations is that C^0 is faster to compute (both in vertex and pixel programs) but S^3 offers higher quality as it ensures C^1 continuity at BT mesh vertices. This quality difference is most evident for scene objects containing straight line geometry or texture, as exemplified in Figure 3.

For perturbed beams, we compute $\overrightarrow{d}$ and $\overrightarrow{O}$ via our C^0 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 $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$.

4.2 Shading

Here, we describe how our fragment program renders each pixel $\overrightarrow{P}$ within the bounding triangle $\tilde{\triangle}$ computed by our vertex program. For each pixel $\overrightarrow{P}$, 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 scene triangle. If $\overrightarrow{d_P}$ does not intersect the scene triangle, the pixel is killed immediately. Otherwise, its attributes (depth, color, or texture coordinates) are determined from the intersection point $\overrightarrow{Q}$ and the pixel is shaded followed by frame-buffer write with hardware z-test. This process is illustrated in Figure 2.

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

4.3 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) 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 two main execution stages, we have to design an algorithm that properly balances the workload between vertex and fragment stages. (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^n ($\delta = 0$) and bump-map ($\delta > 0$) parameterizations. Our algorithms rely on several math 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 $\overrightarrow{d}$, (2) *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 $\overrightarrow{H_i} = \overrightarrow{P_i} + t\overrightarrow{d_i}, i = 1, 2, 3$, (3) *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 $\triangle P_1P_2P_3$, and (4) *maximum offsets* μ_b and μ_d, which are maximum offsets between the S^n and C^0 parameterizations of ray origin and direction for all $\overrightarrow{P}$ on the base triangle $\triangle_b$. See Appendix A for detailed definitions.

4.3.1 Smooth beam with S^n parameterization

For a S^n parameterization in Equation 1 the bounding triangle can be accurately estimated by computing tangent lines to each projected edges of the scene triangle. (For example, for C^0 parameterization all projected edges are quadratic curves.) However, this strategy has several drawbacks. First, since the projection region may have weird shapes (it may not be a triangle or even a single region as shown in Figure 10), handling all these cases robustly can be difficult. Second, we have found through experiments that it is too computationally expensive for vertex programs. Third, it is not general as different n values for S^n would require different algorithms. Instead, we have designed an algorithm that produces less tight bounding triangles than this tangent line method but is more general (applicable to any S^n), more robust (works for any projected shape), and faster to compute. For clarity, we summarize the algorithm in Table 2.

function $\tilde{\triangle} \leftarrow$ BoundingTriangle($\triangle$, B, $\delta = 0$)
- $\{\widehat{d_i}\}_{i=1,2,3} \leftarrow$ extremal 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
- μ_b, $\mu_d \leftarrow$ maximum offsets 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*
  - **foreach** i,j = 1,2,3
    - $P_{ij} \leftarrow$ Transfer($\overrightarrow{Q_i}$, $\widehat{d_j}$, Π_b)
  - $\tilde{\triangle} \leftarrow$ ExpandedBoundingTriangle($\{P_{ij}\}_{i,j=1,2,3}$, $\triangle_b$)
- **else**
  - *// unique projection case*
  - $\Pi_a \leftarrow$ affine plane of B passing through the center of $\triangle$
  - $\triangle_a \leftarrow$ affine triangle on Π_a
  - **foreach** i,j = 1,2,3
    - $H_{ij} \leftarrow$ Transfer($\overrightarrow{Q_i}$, $\widehat{d_j}$, Π_a)
  - $\triangle E_1E_2E_3 \leftarrow$ ExpandedBoundingTriangle($\{H_{ij}\}_{i,j=1,2,3}$, $\triangle_a$)
  - $\{\omega_k\}_{k=1,2,3} \leftarrow \Omega(\{\overrightarrow{E_i}\}_{i=1,2,3}, \triangle_a)$ *// barycentric coordinates*
  - $\{\overrightarrow{F_i}\}_{i=1,2,3} \leftarrow \Omega^{-1}(\{\omega_k\}_{k=1,2,3}, \triangle_b)$
  - $\tilde{\triangle} \leftarrow \triangle F_1F_2F_3$
- **endif**

function $\tilde{\triangle} \leftarrow$ ExpandedBoundingTriangle($\{P_{ij}\}_{i,j=1,2,3}$, $\triangle_a$)
- $\tilde{\triangle} \leftarrow$ BoundingTriangle($\{P_{ij}\}_{i,j=1,2,3}$)
- $\Pi_a \leftarrow$ plane containing $\triangle_a$
- $\theta_{min} \leftarrow \min(\text{angle}(\widehat{d_i}, \Pi_a), \forall i = 1, 2, 3)$
- $t \leftarrow$ the t value of $\triangle_a$ relative to $\triangle_b$
- expand $\tilde{\triangle}$ outward with distance $\epsilon = \frac{\mu_b + t\mu_d}{sin(\theta_{min})}$

Table 2: Bounding triangle estimation algorithm for our S^n 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 $\triangle$.

Our bounding triangle algorithm follows a similar philosophy of [Hou et al. 2006] so we begin with a brief description of their algorithm. For each scene triangle $\triangle$ with vertices $\{\overrightarrow{Q_i}\}_{i=1,2,3}$, [Hou et al. 2006] compute nine *transfer* (defined in Section 4.2) 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 $\{P_{ij}\}_{i,j=1,2,3}$. This algorithm only works for C^0 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 Table 2. (The amount of expansion ϵ is zero for C^0 pamaterization, and is non-zero for S^n parameterization to ensure that $\tilde{\triangle}$ is conservative even when $\overrightarrow{O_P} \neq \overrightarrow{P}$.) We term this algorithm *general case* as in Table 2. It can be easily shown that $\tilde{\triangle}$ computed above is guaranteed to be a bounding triangle for the projection of the scene triangle $\triangle$ as long its three vertices completely lie within the beam. (See Appendix B.5 for proof.) However, bounding triangle computed by this algorithm is too crude; as the scene triangle getting

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 4.

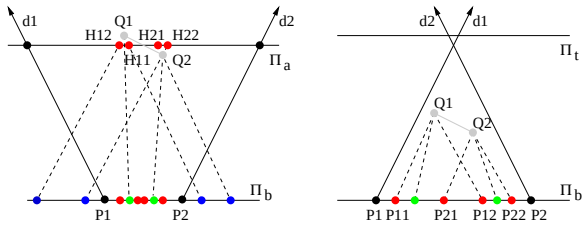

Figure 4: 2D illustration of our bounding triangle algorithm. Here we illustrate two scenarios of diverging (left) and converging (right) ray directions. The green points indicate projections of the scene triangle vertices $\overrightarrow{Q_1}$ and $\overrightarrow{Q_2}$. [Hou et al. 2006] computes 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 [Hou et al. 2006] 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 is similar to [Hou et al. 2006] and the overdraw ratio is limited due to the nature of converging rays.

We provide a better algorithm as follows. As illustrated in Figure 4, 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 center of the scene triangle $\triangle$. (If there are muliple affine planes $\Pi_a(t)$ that pass through the center, 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.5) $\triangle F_1F_2F_3$ is guaranteed to be a proper bounding triangle as long 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 Claim B.1. This ensures that $\triangle F_1F_2F_3$ is a bounding triangle as long $\triangle E_1E_2E_3$ is also a 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 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 4 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 C^0 parameterization, this frustum intersection test can be efficiently performed via the method described in [Stoll and Seidel 2006]. (For our S^3 parameterization it is a bi-cubic surface and analogous strategies can be applied.) Furthermore, triangles that intersect the beam boundary are relatively rare so this brute force solution does not significantly affect our run-time performance.

A comparison between [Hou et al. 2006] and our algorithm is demonstrated in Figure 5. Ideally, the overdraw ratio should remain

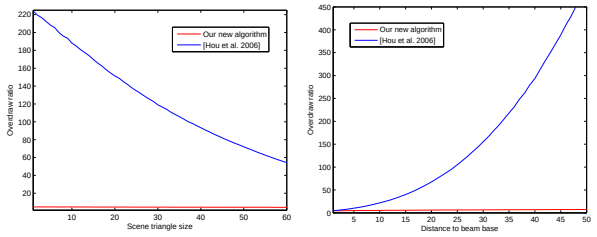

Figure 5: Overdraw ratio comparison. Here, we plot the pixel overdraw ratio of a single scene triangle being rendered within one beam. On the left, we fix the scene triangle in space but vary its size. On the right, we fix the scene triangle size but vary its distance to beam base.

constant with varying scene triangle size and distance to beam base. Note that our algorithm remains low and roughly constant while [Hou et al. 2006] exhibits exponential growth. Consequently our algorithm runs much faster than [Hou et al. 2006] as demonstrated in Section 5.

4.3.2 Perturbed beam with bump-map parameterization

With a set of properly computed extremal directions $\{\widehat{d_i}\}$, our general case algorithm for S^n can be directly applied for bump map parameterization (the proof for general case in Appendix B.5 can be directly applied to bump map parameterization). However, since bump map pixels can point in arbitrary directions, our bump map parameterization defies general analysis of Π_a and Π_T as described for our S^n parameterization. Consequently, our unique projection case algorithm is not applicable and rendering of bump mapped beams will be as slow as [Hou et al. 2006].

To address this deficiency, we propose an algorithm that allows us to take advantage of the efficiency of our S^n bounding triangle algorithm. First, we pre-compute the maximum perturbation angle δ formed by each ray and its unperturbed cousin sharing the same origin as determined by our C^0 parameterization. Given a scene triangle $\triangle Q_1Q_2Q_3$, we first compute its C^0 bounding triangle via our S^n algorithm. We then extend this C^0 bounding triangle outward via a distance D computed via the following formula:

$$
\begin{aligned}
\tau &= \delta + \phi \\
\theta_1 &= \arcsin(\min(\overrightarrow{n} \cdot \overrightarrow{d_{Q_1}}, \overrightarrow{n} \cdot \overrightarrow{d_{Q_2}}, \overrightarrow{n} \cdot \overrightarrow{d_{Q_3}})) \\
\theta_2 &= \arcsin(\min(\overrightarrow{n} \cdot \overrightarrow{d_1}, \overrightarrow{n} \cdot \overrightarrow{d_2}, \overrightarrow{n} \cdot \overrightarrow{d_3})) \\
\theta_3 &= \begin{cases} \theta_2 & \text{if } \theta_2 > \tau \\ \theta_1 & \text{else if } \theta_1 > \tau \\ \tau & \text{else} \end{cases} \\
D &= \frac{\max(\|Q_iP_{ij}\|,\ \forall i,j = 1,2,3) \cdot \sin\tau}{\sin(\theta_3 - \tau)}
\end{aligned} \tag{4}
$$

where ϕ is the maximum angle formed between any two rays within the same beam, $\overrightarrow{n}$ is the normal of of beam base (facing the inside of 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^n bounding triangle algorithm). Intuitively, this algorithm extends the original C^0 bounding triangle by an amount D so that to cover all shifted projection locations caused by bump maps; see Appendix B.6 for math details. 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.

4.4 BT Mesh Construction

For a static reflector/refractor, we build its BT mesh via mesh simplification [Hoppe 1996]. 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 [Kircher and Garland 2006], 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 BT 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.

4.5 Beam Tree Construction

We build a beam tree via a method similar to [Heckbert and Hanrahan 1984], but unlike [Heckbert and Hanrahan 1984] that only handles linear transformations, we deal with general nonlinear beams. Another major difference is that we trace and split beams only along BT mesh boundaries to avoid excessively fractured beams that are harmful for GPU performance. This also avoids the need to dynamically re-parameterize BT meshes. Since BT meshes are coarser than the real reflector/refractor geometry, we trace beam shapes entirely on a CPU.

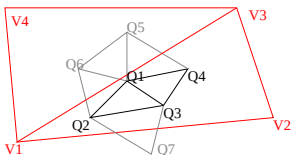

Figure 6: Beam-interface interaction. Here the beam $\triangle V_1V_2V_3$ bounces off BT mesh $\{Q_i\}_{i=1-7}$. We compute the reflected beam rays via interpolation (Q_1 to Q_4) and extrapolation (Q_5 to Q_7) without splitting beams.

Here we describe how to compute child beams from a parent beam through nonlinear interface interactions. Assume we have a beam $\triangle V_1V_2V_3$ bouncing off a reflector surface with BT mesh $\{Q_i\}_{i=1-7}$ as illustrated in Figure 6. We project each $\overrightarrow{Q_i}$ onto $\triangle V_1V_2V_3$, use the corresponding barycentric coordinates to interpolate the direction $\overrightarrow{d_i}$ via our S^n parameterization, and from $\overrightarrow{d_i}$ and the normal of $\overrightarrow{Q_i}$ compute the proper reflection/refraction direction. This works well for $\overrightarrow{Q_i}$ that projects entirely within $\triangle V_1V_2V_3$. For $\overrightarrow{Q_i}$ that projects outside $\triangle V_1V_2V_3$ (e.g. $\overrightarrow{Q_5}$ and $\overrightarrow{Q_6}$), one method is to subdivide the triangles but we have found this unnecessarily expensive as it introduces many small beams. Instead, we compute the reflection directions for these vertices via the same method for vertices with interior projection, essentially performing an extrapolation instead of interpolation. This method allows us to keep the triangle base of all beams invariant.

After building a beam tree via the process above, each BT mesh vertex $\overrightarrow{Q}$ might have been associated with more than one projection directions $\{\overrightarrow{d_{Q,i}}\}$ since it could be visible from multiple beams at the parent tree level. These directions computed from different parent beams will be utilized in different rendering passes during the traversal of the beam tree. (In our current implementation, we use a depth-first traversal for efficient texture storage, this will be detailed in Section 4.6.) However, some of these ray directions actually present inconsistency and need to be merged due to the following reasons. First, for concave reflections or refractions, a single vertex might have multiple projections within a group of connected beams or even within a single beam. Second, our extrapolation computation presented above might introduce muliple directions, such as Q_6 in Figure 6 which receives one interpolation from beam $\triangle V_1V_3V_4$ plus one extrapolation from beam $\triangle V_1V_2V_3$. This kind of inconsistency can cause discontinuity for higher level reflection and refraction across adjacent beams.

To avoid this ambiguity, we perform a weighted blending of computed directions via the following formula:

$$\begin{aligned} \overrightarrow{d_{Q,k}} &= \sum_i \sum_{j \in set_k} \tau_{ij} \overrightarrow{d_{ij}} \\ \tau_{ij} &= \text{projected area of } \triangle_i \text{ onto } \triangle_{b_j} \end{aligned} \quad (5)$$

where index i runs through all BT mesh triangles $\triangle_i$ adjacent to $\overrightarrow{Q}$, j runs through beams associated with k^{th} group of connected beams where $\triangle_i$ has non-empty projections, τ_{ij} is the projected area of $\triangle_i$ onto beam base triangle $\triangle_{b_j}$, and $\overrightarrow{d_{ij}}$ is the direction of $\overrightarrow{Q}$ computed with respect to $\triangle_{b_j}$. Each value in $\{\overrightarrow{d_{Q,k}}\}$ will be utilized in different rendering passes during the depth first traversal of the beam tree. Even though this solution is only approximate, it ensures both projection continuity and frame-to-frame coherence. We have found this better than many other alternative solutions with which we have experimented.

4.6 Texture Storage for Multiple Bounces

For single-bounce beam tracing we simply record the rendering results over a texture map parameterized by a BT 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 simply keep n copies of texture for each beam for n-bounce beam tracing.

4.7 Acceleration

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 [Ofek and Rappoport 1998; Roger and Holzschuch 2006; Mei et al. 2007], which requires 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 nonlinear rendering framework.

4.7.1 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

Figure 7: 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.

resolve these issues, we adopt a simple heuristic via a cone which completely contains the original beam viewing frustum. The cone center $\overrightarrow{P_C}$, direction $\overrightarrow{d_C}$, and angle θ_C are computed as follows:

$$\begin{aligned} \overrightarrow{d_C} &= \frac{(\widehat{d_1}+\widehat{d_2}+\widehat{d_3})}{\|(\widehat{d_1}+\widehat{d_2}+\widehat{d_3})\|} \\ \theta_C &= \arccos(\min(\widehat{d_1}\cdot\overrightarrow{d_C}, \widehat{d_2}\cdot\overrightarrow{d_C}, \widehat{d_3}\cdot\overrightarrow{d_C})) \\ \overrightarrow{P_C} &= \overrightarrow{P_G} - \overrightarrow{d_C}\cdot\frac{\max(\|\overrightarrow{O_P}-\overrightarrow{P_G}\|, \forall\overrightarrow{P}\in\triangle P_1P_2P_3)}{\sin\theta_C} \end{aligned} \tag{6}$$

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

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

In our current implementation the bounding cones and culling operations are all performed on a CPU.

4.7.2 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.

5 Results and Discussion

Quality and speed A major strength of our approach is its flexibility, as we could render nonlinear global illumination effects for fully dynamic scenes. Figure 1 demonstrates an example scene consisting of a mirror teapot reflecting a galloping horse and a group of flapping butterflies. The horse motion involves general mesh deformation and the butterflies are moving independently from each other, so the scene contains complex motions and occlusions. Consequently, such kinds of scenes are difficult to render effectively via traditional methods such as ray tracing that often requires static scenes or limited motions for building acceleration data structures, or depth-sprites methods that may have difficulty with occlusion/disocclusion events.

scene	# scene △	# beam	BT texture	FPS
Figure 1	78.1K	941	2048^2	11.2
Figure 7	5.3K	960	1024^2	14.0
Figure 8 (a)	4.0K	958_1+24458_2	2048^2	11.0
Figure 8 (b)	2.9K	105_1+802_2	2048^2	15.0
Figure 9 (a)	4.3K	450	1024^2	21.0
Figure 9 (b)	14.3K	450	1024^2	17.0
Figure 9 (c)	10.2K	1922	1024^2	16.0
Figure 9 (d)	28K	800	1024^2	29.0

Table 3: Scene statistics. For multiple reflection and refractions we indicate the # of beams per tree level (level 0 contains a single eye beam). All frame-rates are measured on an NVIDIA Geforce 8800 GTX with a viewport size 1280×1024. For fair comparison with other techniques, we have turned off geometry LOD control for timing measurement.

(a) (b)

Figure 8: Multiple bounce beam tracing for multiple reflections and refractions. We intentionally leave the inter-reflections un-modulated by teapot surface colors for clarity.

Quality-wise, our algorithm produces similar results to offline ray tracing as shown in Figure 1. In particular, our rendering is almost identical to ray tracing in low-curvature areas, and the difference only shows up in high-curvature regions such as the teapot lid, spout, and handle. However, even in these cases our algorithm produces similar results.

Speed-wise, our algorithm achieves 11.2 fps for the scene with 78.1K triangles in Figure 1. The performance of our algorithm primarily depends on the number of scene triangles and number of beams (vertex workload), as well BT mesh texture resolution (pixel workload). See Table 3 for detailed scene statistics. In comparison, the same scene in Figure 1 would run at 0.49 FPS for [Hou et al. 2006], which, to our knowledge, is the major recent algorithm most similar to ours. We also compare our performance with several recent works offering less flexibility: (1) [Wald et al. 2006] (7.8 fps for a 78K scene) which renders only eye rays without reflection or refraction, (2) [Estalella et al. 2006] (12 fps for a 30K scene) which assumes coherent motion, (3) [Roger and Holzschuch 2006] (14.3 fps for a 67K scene from their Figure 10) which requires fine tessellation of scene objects, (4) [Gunther et al. 2006] (5.7 fps for a 30K scene) which handles skinned, but not general, animations,

(a) (b) (c) (d)

Figure 9: Dynamic interface with a variety of global illumination phenomena. (a) a waving lake reflecting seagulls fying by, (b) refraction of a group of fish in the lake, (c) caustics cast over a whale and lake bottom, and (d) nonlinear shadow of a seagull cast onto the lake bottom.

and (5) [Mei et al. 2007] (5 fps for a 74K scene) which requires fine tessellation of scene objects into point clouds and is restricted to rigid bodies due to the use of KD-tree acceleration structure similar to ray tracing. As shown, our algorithm competitively performs with these methods even though they are targeted for more specific applications and run on different hardware architectures.

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 7. 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 4. Consequently we limit δ to be no more than 30 degrees in our current implementation. We leave it as future work to devise a better bounding triangle algorithm for perturbed beams.

Multiple beam bounces Unlike [Hou et al. 2006] which offers only single bounces, our technique allows rendering multiple-bounce reflections and refractions as demonstrated in Figure 8. A potential issue is that the number of beams may increase exponentially with the increasing number of beam bounces. Fortunately, for practical applications we have found two levels of beam-bounces usually enough. Furthermore, due to the use of depth-first 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 9 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 BT 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 [Wyman and Davis 2006; Hou et al. 2006]. 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 [Wyman and Davis 2006].

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 [Watt 1990]. 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 9 (d).

6 Conclusions and Future Work

We present nonlinear beam tracing for rendering common global illumination effects on a GPU such as reflection, refraction, caustics, and shadows. Due to its polygon rasterization nature, our technique has several unique advantages. First, it supports fully dynamic scenes without any preprocessing or limitation on object motions. Second, since scene geometry is streamed rather than stored in memory, our technique is memory efficient and devoid of the issues of a shared scene database. Consequently, our algorithm can easily be parallelized since individual beams can be rendered independently across multiple GPUs. More important, the speed of our methodology scales directly with future improvements of commodity graphics hardware.

Our algorithm is less suitable for reflectors and refractors with complex shapes that require many small beams for accurate rendering. However, since human perception cannot easily judge the rendering correctness for such complex situations, we have found our rough BT mesh approximation more than enough. For applications that require a high degree of accuracy, 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 ray engine [Carr et al. 2002].

We are in an exciting era of rapid hardware architectural innovation, as the GPUs are embracing general stream computation while the CPUs are incorporating multiple cores. We believe the design of future real-time global-illumination algorithms will heavily depend on and benefit from these architectural innovations, and our nonlinear beam tracing framework is just a start.

Acknowledgement [Removed for paper submission]

References

AKENINE-MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering, 2nd edition*. A.K. Peters Ltd.

BLINN, J. F., AND NEWELL, M. E. 1976. Texture and reflection in computer generated images. *Commun. ACM 19*, 10, 542–547.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 37–46.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, 203–209.

DIEFENBACH, P. J., AND BADLER, N. I. 1997. Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, 59–ff.

ESTALELLA, P., MARTIN, I., DRETTAKIS, G., AND TOST, D. 2006. A gpu-driven algorithm for accurate interactive reflections on curved objects. In *Rendering Techniques'06 (Proc. of the Eurographics Symposium on Rendering)*.

FARIN, G. 1986. Triangular bernstein-bezier patches. *Comput. Aided Geom. Des. 3*, 2, 83–127.

GENEVAUX, O., LARUE, F., AND DISCHLER, J.-M. 2006. Interactive refraction on complex static geometry using spherical harmonics. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 145–152.

GUNTHER, J., FRIEDRICH, H., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Interactive ray tracing of skinned animations. *Vis. Comput. 22*, 9, 785–792.

GUY, S., AND SOLER, C. 2004. Graphics gems revisited: fast and physically-based rendering of gemstones. *ACM Trans. Graph. 23*, 3, 231–238.

HAKURA, Z. S., SNYDER, J. M., AND LENGYEL, J. E. 2001. Parameterized environment maps. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, 203–208.

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 119–127.

HEIDRICH, W., LENSCH, H. P. A., COHEN, M., AND SEIDEL, H.-P. 1999. Light field techniques for reflections and refractions. In *Rendering Techniques '99: Proceedings of the 10th Eurographics Workshop on Rendering (EGRW-99)*, D. Lischinski and G. W. Larson, Eds., 187–196.

HOPPE, H. 1996. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 99–108.

HOU, X., WEI, L.-Y., SHUM, H., AND GUO, B. 2006. Real-time multi-perspective rendering on graphics hardware. In *Rendering Techniques'06 (Proc. of the Eurographics Symposium on Rendering)*, 93–102.

KIRCHER, S., AND GARLAND, M. 2006. Editing arbitrarily deforming surface animations. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 71.

KRÜGER, J., BÜRGER, K., AND WESTERMANN, R. 2006. Interactive screen-space accurate photon tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, 319–329.

MEI, C., POPESCU, V., AND SACKS, E. 2007. A hybrid backward-forward method for interactive reflections. In *Second International Conference on Computer Graphics Theory and Applications*.

OFEK, E., AND RAPPOPORT, A. 1998. Interactive reflections on curved objects. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 333–342.

POLICARPO, F., AND OLIVEIRA, M. M. 2006. Relief mapping of non-height-field surface details. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 55–62.

POPESCU, V., MEI, C., DAUBLE, J., AND SACK, E. 2006. Reflected-scene impostors for realistic reflections at interactive rates. *Computer Graphics Forum (Proceedings of Eurographics 2006) 25*, 3 (sep).

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July), 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 41–50.

ROGER, D., AND HOLZSCHUCH, N. 2006. Accurate specular reflections in real-time. *Computer Graphics Forum (Proceedings of Eurographics 2006) 25*, 3 (sep).

SOUSA, T. 2005. Generic refraction simulation. In *GPU Gems II*, 295–305.

STOLL, C., AND SEIDEL, H.-P. 2006. Incremental raycasting of piecewise quadratic surfaces on the gpu. In *IEEE Symposium on Interactive Raytracing 2006 Proceedings*, 141–150.

SZIRMAY-KALOS, L., ASZODI, B., LAZANYI, I., AND PREMECZ, M. 2005. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum (Proceedings of Eurographics 2005) 24*, 3, 171–176.

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved pn triangles. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, 159–166.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3, 485–493.

WAND, M., AND STRASSER, W. 2003. Real-time caustics. *Computer Graphics Forum 22*, 3, 611–620.

WATT, M. 1990. Light-water interaction using backward beam tracing. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 377–385.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *HWWS '06: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 67–77.

WYMAN, C., AND DAVIS, S. 2006. Interactive image-space techniques for approximating caustics. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 153–160.

WYMAN, C. 2005. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE 2005*.

YU, J., YANG, J., AND MCMILLAN, L. 2005. Real-time reflection mapping with parallax. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 133–138.

A Detailed Math Formulas

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:

$$\overrightarrow{d}(P) = \widehat{\omega_1}\widehat{d_1} + \widehat{\omega_2}\widehat{d_2} + \widehat{\omega_3}\widehat{d_3} \tag{7}$$

Extremal directions exist as long all the beam directions point away to the same side of the beam base plane. Note that for C^0 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.

Affine plane Π_a Given a beam B with base triangle $\triangle P_1P_2P_3$ and associated rays $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \overrightarrow{d_3}\}$, we define *affine triangle* $\triangle_a(t)$ as a triangle $\triangle H_1H_2H_3$ where $\overrightarrow{H_i} = \overrightarrow{P_i} + t\overrightarrow{d_i}$ for a real number t. Note that an affine triangle is uniquely defined by the value t. Define *affine plane* $\Pi_a(t)$ as a plane which contains the affine triangle $\triangle_a(t)$. Note that the base plane of B is a special case of affine plane with $t = 0$. Affine planes have this nice property that if $\overrightarrow{H} \in \Pi_a$ has the same set of barycentric coordinates as $\overrightarrow{P} \in \Pi_b$, then $\overrightarrow{P}$ is a projection of $\overrightarrow{H}$ on Π_b. See Appendix B.2 for math details.

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^n 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 $t_{res} = \max(0, \Im)$. The threshold plane Π_T is then defined as the affine plane $\Pi_a(t_{res})$. See Appendix B.3 for math details.

Maximum offsets μ_b and μ_d We define μ_b and μ_d as the maximum offsets between the S^n and C^0 parameterizations of ray origin and direction for all $\overrightarrow{P}$ on the base triangle $\triangle_b$:

$$\begin{aligned} \mu_b &= \max(\|\overrightarrow{OP} - \sum_{i=1}^{3} \omega_i \overrightarrow{P_i})\|, \forall \overrightarrow{P} \in \triangle_b) \\ \mu_d &= \max(\|\overrightarrow{d_P} - \sum_{i=1}^{3} \omega_i \overrightarrow{d_i})\|, \forall \overrightarrow{P} \in \triangle_b) \end{aligned} \tag{8}$$

where $\overrightarrow{OP}$ 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 C^0 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.

B Supplementary Math Proofs

Here, we provide detailed math proofs for the correctness of our algorithms presented in Section 4. Note: Since this portion is not essential for the understanding and implementation of our algorithms, we intend this portion to be part of the electronics version only, not on the printed proceedings.

B.1 S^3 parameterization

Our formula for S^3 is derived directly from [Vlachos et al. 2001]:

$$\begin{aligned}
&\overrightarrow{L_{ijk}}(\overrightarrow{X_1},\overrightarrow{X_2},\overrightarrow{X_3}) = \frac{3!}{i!j!k!}\overrightarrow{c_{ijk}}(\overrightarrow{X_1},\overrightarrow{X_2},\overrightarrow{X_3})\\
&\overrightarrow{c_{300}} = \overrightarrow{X_1}, \overrightarrow{c_{030}} = \overrightarrow{X_2}, \overrightarrow{c_{003}} = \overrightarrow{X_3}\\
&w_{ij} = (\overrightarrow{X_j} - \overrightarrow{X_i})\cdot\overrightarrow{n_i},\ \overrightarrow{n_i} \text{ indicates normal at } \overrightarrow{P_i}\\
&\overrightarrow{c_{210}} = (2\overrightarrow{X_1} + \overrightarrow{X_2} - w_{12}\overrightarrow{n_1})/3\\
&\overrightarrow{c_{120}} = (2\overrightarrow{X_2} + \overrightarrow{X_1} - w_{21}\overrightarrow{n_2})/3\\
&\overrightarrow{c_{021}} = (2\overrightarrow{X_2} + \overrightarrow{X_3} - w_{23}\overrightarrow{n_2})/3\\
&\overrightarrow{c_{012}} = (2\overrightarrow{X_3} + \overrightarrow{X_2} - w_{32}\overrightarrow{n_3})/3\\
&\overrightarrow{c_{102}} = (2\overrightarrow{X_3} + \overrightarrow{X_1} - w_{31}\overrightarrow{n_3})/3\\
&\overrightarrow{c_{201}} = (2\overrightarrow{X_1} + \overrightarrow{X_3} - w_{13}\overrightarrow{n_1})/3\\
&\overrightarrow{e} = (\overrightarrow{c_{210}} + \overrightarrow{c_{120}} + \overrightarrow{c_{021}} + \overrightarrow{c_{012}} + \overrightarrow{c_{102}} + \overrightarrow{c_{201}})/6\\
&\overrightarrow{v} = (\overrightarrow{X_1} + \overrightarrow{X_2} + \overrightarrow{X_3})/3\\
&\overrightarrow{c_{111}} = \overrightarrow{e} + (\overrightarrow{e} - \overrightarrow{v})/2
\end{aligned} \tag{9}$$

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

B.2 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 $\triangle_a(s)$ that contains $\overrightarrow{Q}$.*

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

B.3 Threshold plane

Claim B.3 *For a beam with S^n 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. ∎

B.4 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 $\triangle_b$:

$$\mu_t = \max(\|\overrightarrow{O_{P_t}} - \overrightarrow{P_t}\|, \forall\overrightarrow{P_t} \in \triangle_a) \tag{10}$$

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 8, 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.5 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.)

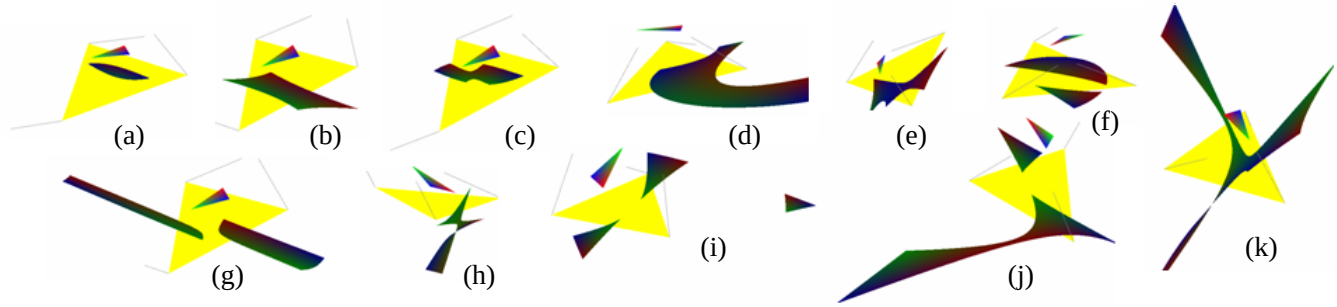

Figure 10: Different cases of nonlinear beam projection. Within each case, the scene triangle is shown in RGB colors whereas the beam base triangle is shown in yellow. The number of projection regions are 1 for cases (a) to (e), 2 for cases (f, g, h, j, k), and 3 for case (i). Our algorithm can handle all these cases correctly.

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

$$\begin{aligned}\sum_{i=1}^{3}\Omega(\overrightarrow{P})_i\overrightarrow{P_i} &= \overrightarrow{P}=\sum_{j=1}^{n}k_j\overrightarrow{X_j}=\sum_{j=1}^{n}\sum_{i=1}^{3}k_j\Omega(\overrightarrow{X_j})_i\overrightarrow{P_i}\\ &= \sum_{i=1}^{3}\left(\sum_{j=1}^{n}k_j\Omega(\overrightarrow{X_j})_i\right)\overrightarrow{P_i} \qquad (11)\end{aligned}$$

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:

$$\begin{aligned}\overrightarrow{P} &= \sum_{i=1}^{3}\Omega(\overrightarrow{P})_i\overrightarrow{P_i}=\sum_{i=1}^{3}\left(\sum_{j=1}^{n}k_j\Omega(\overrightarrow{X_j})_i\right)\overrightarrow{P_i}\\ &= \sum_{j=1}^{n}k_j\sum_{i=1}^{3}\Omega(\overrightarrow{X_j})_i\overrightarrow{P_i}=\sum_{j=1}^{n}k_j\overrightarrow{X_j} \qquad (12)\end{aligned}$$

∎

Proof for general case Here we prove the general case of our algorithm (shown in Table 2).

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 $0\le k_i$ 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 (Table 2).

Proof Since $\overrightarrow{P}$ is a projection of $\overrightarrow{Q}$ inside the base triangle, obviously there exists t so that $\overrightarrow{Q}=\overrightarrow{OP}+t\overrightarrow{dP}$. For simplicity, let's denote $\overrightarrow{dP}$ 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 Table 2. From the definition of ϵ and Claim B.4, we know

$$\overrightarrow{P}=\sum_{j=1}^{3}\Omega(P,\triangle P_{1,\overrightarrow{d},\epsilon}P_{2,\overrightarrow{d},\epsilon}P_{3,\overrightarrow{d},\epsilon})_j\overrightarrow{P_{j,\overrightarrow{d},\epsilon}} \qquad (13)$$

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

$$\overrightarrow{P_{j,\overrightarrow{d}}}=\sum_{i=1}^{3}\Omega(P_{j,\overrightarrow{d}},\triangle P_{j1}P_{j2}P_{j3})_i\overrightarrow{P_{ji}} \qquad (14)$$

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 14 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

$$\overrightarrow{P_{j,\overrightarrow{d},\epsilon}}=\sum_{i=1}^{3}\Omega(P_{j,\overrightarrow{d},\epsilon},\triangle F_1F_2F_3)_i\overrightarrow{F_i} \qquad (15)$$

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

Combining Equations 13 and 15, 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 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 $\{k_i\}_{i=1,2,3}$, $0\le k_i$ 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 (Table 2). 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 Table 2.

Consequently, we know $\triangle F_1F_2F_3$ is a proper bounding triangle.

B.6 Bounding triangle for bump map parameterization

In the proof below we assume the bump map is applied after a C^0 parameterization as general S^n parameterization is unnecessary.

Figure 11: Comparison of bump mapping effects between ground truth and our technique. For each pair of images the ray tracing result is on the left while our result is on the right.

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 $\overrightarrow{QP}$ 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 11 for a quality comparison between ground truth and our technique.

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

$$\begin{aligned} (\overrightarrow{O_P} - \overrightarrow{P_C}) \cdot \overrightarrow{d_C} &= \|\overrightarrow{O_P} - \overrightarrow{P_C}\| \cdot \cos \angle \overrightarrow{O_P}P_CP_G \\ &\geq \|\overrightarrow{O_P} - \overrightarrow{P_C}\| \cdot \cos \theta_C \end{aligned} \tag{16}$$

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

$$\begin{aligned} (\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}) \cdot \overrightarrow{d_C} &\geq \sum_{i=1}^{3} (\widehat{\omega_i}(\|\widehat{d_i}\| \cdot \cos \theta_C)) \\ &\geq (\|\sum_{i=1}^{3} (\widehat{\omega_i}\widehat{d_i})\|) \cdot \cos \theta_C \end{aligned} \tag{17}$$

From Equations (16) and (17), we know that

$$\begin{aligned} \frac{\overrightarrow{Q} - \overrightarrow{P_C}}{\|\overrightarrow{Q} - \overrightarrow{P_C}\|} \cdot \overrightarrow{d_C} &= \frac{\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ = & \frac{\overrightarrow{O_P} - \overrightarrow{P_C}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ & + \frac{t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \overrightarrow{d_C} \\ \geq & \frac{\|\overrightarrow{O_P} - \overrightarrow{P_C}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \cos \theta_C \\ & + \frac{\|t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \cos \theta_C \\ \geq & \frac{\|\overrightarrow{O_P} - \overrightarrow{P_C}\| + \|t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|}{\|\overrightarrow{O_P} - \overrightarrow{P_C} + t\sum_{i=1}^{3} \widehat{\omega_i}\widehat{d_i}\|} \cdot \cos \theta_C \\ \geq & \cos \theta_C \end{aligned} \tag{18}$$

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