Bernstein Bounds for Caustics: Supplemental Document

ZHIMIN FAN, State Key Lab for Novel Software Technology, Nanjing University, China
CHEN WANG and YIMING WANG, State Key Lab for Novel Software Technology, Nanjing University, China
BOXUAN LI and YUXUAN GUO, State Key Lab for Novel Software Technology, Nanjing University, China
LING-QI YAN, University of California, Santa Barbara, United States of America
YANWEN GUO, State Key Lab for Novel Software Technology, Nanjing University, China
JIE GUO*, State Key Lab for Novel Software Technology, Nanjing University, China

ACM Reference Format:
Zhimin Fan, Chen Wang, Yiming Wang, Boxuan Li, Yuxuan Guo, Ling-Qi Yan, Yanwen Guo, and Jie Guo. 2025. Bernstein Bounds for Caustics: Supplemental Document. *ACM Trans. Graph.* 44, 4, Article 1 (August 2025), 5 pages. https://doi.org/10.1145/3618360

1 REMAINDER VARIABLES

The concept of remainder variables plays a vital role in our pipeline. In this section, we provide a formal discussion of its bounding correctness and details regarding the approximation for square root functions.

1.1 Correctness

Since the remainder variables are used for both refraction approximations and degree reductions in our pipeline, we consider a general setting. We aim at bounding the range of a complex function $g(u, \theta(h(u)))$. Both g and h are already rational, but θ is not rational. Denote $t = h(u)$. We have a cheap approximation $\hat{\theta}(t)$ of $\theta(t)$ with the error being

$$\delta(t) = \theta(t) - \hat{\theta}(t). \tag{1}$$

Theorem 1.1. *Assuming that we can compute a bound of the approximation error, $\underline{\delta}, \overline{\delta}$. Then, we can safely replace $\theta(t)$ with*

$$\tilde{\theta}(t, \xi) = \hat{\theta}(t) + (1 - \xi)\underline{\delta} + \xi\overline{\delta}, \tag{2}$$

which guarantees

$$\operatorname{ran} g(u, \theta(h(u))) \subseteq \operatorname{ran} g(u, \tilde{\theta}(h(u), \xi)) \tag{3}$$

*Here, $\xi \in \mathcal{U}$ is a (new) remainder variable, which is **independent** from t. We denote the range of a function g as* ran g.

Proof. By re-arrange the terms of Eq. (1) and Eq. (2), we have

$$\tilde{\theta}(t, \xi) = \theta(t) - \delta(t) + (1 - \xi)\underline{\delta} + \xi\overline{\delta}. \tag{4}$$

For each t, $\delta(t) \in [\underline{\delta}, \overline{\delta}]$, so $\exists \xi$ s.t. $\delta(t) = (1 - \xi)\underline{\delta} + \xi\overline{\delta}$. Therefore, for each u, $\exists \xi$ such that $\tilde{\theta}(h(u), \xi) = \theta(h(u))$. Hence,

$$\forall u, \quad \exists \xi, \quad g(u, \tilde{\theta}(h(u), \xi)) = g(u, \theta(h(u))). \tag{5}$$

□

In other words, the approximation with remainder variables never shrinks the range. Therefore, the range bound using the approximation is still valid for the original function. This largely extends the families of functions we can handle with controllable complexities.

For the refraction approximation, the above $\theta(t)$ corresponds to $\sqrt{\beta_i}$ and $h(u)$ corresponds to $\beta_i(\boldsymbol{u}_i)$.

1.2 Primal approximation for square roots

We opt for linear approximations to maintain simplicity and minimize degrees[1]. The primary objective is to derive appropriate coefficients for the linear function and remainder variables. When computing positional bounds, our approximation is

$$r(\beta_i) = a\beta_i + b. \tag{6}$$

Adding a remainder variable, it becomes

$$\tilde{r}(\beta_i, \xi_i) = r(\beta_i) + (1 - \xi_i)\underline{\delta}_i + \xi_i\overline{\delta}_i. \tag{7}$$

Assuming the range of β_i is $[\underline{\beta_i}, \overline{\beta_i}]$, we compute the slope using the interval endpoints

$$a = \frac{\sqrt{\overline{\beta_i}} - \sqrt{\underline{\beta_i}}}{\overline{\beta_i} - \underline{\beta_i}}. \tag{8}$$

Note that we assume $\beta_i > 0$. The part of its range smaller than 0 is meaningless, so we always clamp $\underline{\beta_i}$ to be greater than 0. Let $r(\underline{\beta_i}) = \sqrt{\underline{\beta_i}}$, we set b to

$$b = \sqrt{\underline{\beta_i}} - a\underline{\beta_i}. \tag{9}$$

The approximation error writes

$$\Delta = \sqrt{x} - ax - b. \tag{10}$$

[1] In principle, even a constant approximation could also be employed; however, we found that this approach results in exceedingly loose bounds.

*Corresponding author.

Authors' addresses: Zhimin Fan, zhiminfan2002@gmail.com, State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China; Chen Wang, 02yimingwang@gmail.com; Yiming Wang, 02yimingwang@gmail.com, State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China; Boxuan Li, tianyuxiaoty@outlook.com; Yuxuan Guo, 213210060@seu.edu.cn, State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China; Ling-Qi Yan, lingqi@cs.ucsb.edu, University of California, Santa Barbara, Santa Barbara, United States of America; Yanwen Guo, ywguo@nju.edu.cn, State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China; Jie Guo, guojie@nju.edu.cn, State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China.

The position x with maximal approximation error can be easily obtained by computing the zeros of the derivatives to β_i:

$$x = \frac{1}{4a^2}. \tag{11}$$

Therefore, the range of δ is

$$\underline{\delta} = 0, \quad \overline{\delta} = \Delta. \tag{12}$$

1.3 Derivative-aware approximation for square roots

Our goal is to keep the bound valid for any rational function F of the primal value and the first-order differentials of $\sqrt{\beta_i}$:

$$F\left(u, \sqrt{\beta_i}, \frac{\partial\sqrt{\beta_i}}{\partial\beta_i}\right), \tag{13}$$

which we approximate using

$$F\left(u, \tilde{r}(\beta_i, \xi_i, \zeta_i), \frac{\partial\tilde{r}(\beta_i, \xi_i, \zeta_i)}{\partial\beta_i}\right). \tag{14}$$

Here, u represents variables that do not depend on β_i. Note that we only consider the derivative to β_i because it depends on our approximation. We have to keep the bound of both the primal and the differential correct at the same time, so we introduce a two-stage compensation using two remainder variables ξ_i and ζ_i. In particular, we first correct the differential using

$$\tilde{r}_\mathrm{d}(\beta_i, \xi_i) = a\beta_i + b + \left((1-\xi_i)\underline{\delta}_i + \xi_i\overline{\delta}_i\right)(\beta_i - c), \tag{15}$$

where ξ_i controls the slope of the line. Then, we correct the primal value

$$\tilde{r}(\beta_i, \xi_i, \zeta_i) = \tilde{r}_d(\beta_i, \xi_i) + (1-\zeta_i)\underline{\delta}'_i + \zeta_i\overline{\delta}'_i. \tag{16}$$

Note that ζ_i does not change the differential, so we still have

$$\frac{\partial\tilde{r}(\beta_i, \xi_i, \zeta_i)}{\partial\beta_i} = a + (1-\xi_i)\underline{\delta}_i + \xi_i\overline{\delta}_i. \tag{17}$$

Now, we derive the coefficients. The slope of the line (for the linear approximation) is computed from the average of the slope at two endpoints:

$$a = \frac{1}{2}\left(\frac{1}{2\sqrt{\underline{\beta_i}}} + \frac{1}{2\sqrt{\overline{\beta_i}}}\right), \tag{18}$$

using the fact that $(\sqrt{x})' = \frac{1}{2\sqrt{x}}$. The error range is

$$\underline{\delta}_i = -\frac{\Delta_\mathrm{d}}{2}, \quad \overline{\delta}_i = \frac{\Delta_\mathrm{d}}{2}, \tag{19}$$

where

$$\Delta_\mathrm{d} = \frac{1}{2\sqrt{\underline{\beta_i}}} - \frac{1}{2\sqrt{\overline{\beta_i}}}. \tag{20}$$

We align the primal value of the line with the true value of square roots at the left endpoint, so we have

$$b = \sqrt{\underline{\beta_i}} - a\underline{\beta_i}. \tag{21}$$

Likewise, we set the slope adjusting term $\left((1-\xi_i)\underline{\delta}_i + \xi_i\overline{\delta}_i\right)(\beta_i - c)$ to zero at the left endpoint, so

$$c = \underline{\beta_i}. \tag{22}$$

Lastly, the approximation error for the primal value is

$$\sqrt{x} - \sqrt{\underline{\beta_i}} - \frac{x-\underline{\beta_i}}{2\sqrt{x}} = \frac{1}{2}\left(1 - \sqrt{\underline{\beta_i}/x}\right)^2 \le \frac{1}{2}\left(1 - \sqrt{\underline{\beta_i}/\overline{\beta_i}}\right)^2, \tag{23}$$

where $\frac{x-\underline{\beta_i}}{2\sqrt{x}}$ comes from the product of the corrected slope $\frac{1}{2\sqrt{x}}$ and the difference of the point x and the left endpoints. The above expression is always non-negative and has an upper bound

$$\Delta_\mathrm{p} = \frac{1}{2}\left(1 - \sqrt{\underline{\beta_i}/\overline{\beta_i}}\right)^2. \tag{24}$$

Therefore,

$$\underline{\delta}'_i = 0, \quad \overline{\delta}'_i = \Delta_\mathrm{p}. \tag{25}$$

2 ADDITIONAL DETAILS ABOUT IRRADIANCE

In this section, we discuss some details regarding the computation and use of the irradiance bound.

2.1 Derivation of irradiance expressions

We discuss the remaining three terms of the irradiance formulation.

The first term is a (regular) geometric term [Veach 1998], which is only related to the light source and the vertex positions of the first triangle:

$$\frac{\mathrm{d}\Omega_0}{\mathrm{d}A_1} = \frac{\mathrm{d}\Omega_0}{\mathrm{d}A_1^\perp}\frac{\mathrm{d}A_1^\perp}{\mathrm{d}A_1} = \frac{\cos\theta_0}{|\boldsymbol{d}_0|^2}. \tag{26}$$

Here, θ_0 represents the (planar) angle between the light source to point $\boldsymbol{x}_1$ and the (geometrical) normal of the mirror. Considering $\cos\theta_0$ includes a square root function, we factor it out to make the remaining expression rational and correct the range bound by multiplying the range of $\cos\theta_0$ via interval multiplication. This keeps the bound correct and only becomes loose when $\cos\theta_0$ has a large deviation, which seldom happens in practice.

The Jacobian of the transformation between the barycentric coordinates and the vertex positions is

$$\left|\frac{\partial\boldsymbol{x}_1}{\partial\boldsymbol{u}_1}\right| = |\vec{\boldsymbol{e}}_{1,1} \times \vec{\boldsymbol{e}}_{1,2}|, \tag{27}$$

$$\left|\frac{\partial\boldsymbol{u}_k}{\partial\boldsymbol{x}_k}\right| = \frac{1}{\left|\frac{\partial\boldsymbol{x}_k}{\partial\boldsymbol{u}_k}\right|} = \frac{1}{|\vec{\boldsymbol{e}}_{k,1} \times \vec{\boldsymbol{e}}_{k,2}|}. \tag{28}$$

2.2 Total irradiance contributions

We leverage position and irradiance bounds to analyze the total irradiance contribution $E(\mathcal{T}, \boldsymbol{u}_k, \mathcal{T}_k)$ using the following theorem:

Theorem 2.1. *Suppose the union of disjoint rectangles*[2] $\boldsymbol{U}_1^1, \ldots, \boldsymbol{U}_1^n$ *covers* $\mathcal{U}^2$. *Each* $\boldsymbol{U}_1^i$ *corresponds to a position bound* $\boldsymbol{U}_k^i$ *and an irradiance bound* $\boldsymbol{E}_k^i$. *The total irradiance* $E(\mathcal{T}, \boldsymbol{u}_k, \mathcal{T}_k)$ *satisfies*

$$E(\mathcal{T}, \boldsymbol{u}_k, \mathcal{T}_k) \le \tilde{E}(\mathcal{T}, \boldsymbol{u}_k, \mathcal{T}_k) = m \max_{\boldsymbol{u}_k \in \boldsymbol{U}_k^i} \overline{E}_k^i. \tag{29}$$

[2]We consider this partition of domain since we need a piecewise constant bound in certain cases to improve tightness.

Proof. We enumerate all $\boldsymbol{u}_1$ that corresponds to a given $\boldsymbol{u}'_k$:

$$\begin{aligned} E(\mathcal{T}, \boldsymbol{u}'_k, \mathcal{T}_k) &= \sum_{\boldsymbol{u}'_k = \boldsymbol{u}_k(\boldsymbol{u}_1)} E_k(\boldsymbol{u}_1) \le \sum_{\boldsymbol{u}'_k = \boldsymbol{u}_k(\boldsymbol{u}_1)} \max_{\boldsymbol{u}_1 \in \boldsymbol{U}_1^i} \overline{E}_k^i \\ &\le \sum_{\boldsymbol{u}'_k = \boldsymbol{u}_k(\boldsymbol{u}_1)} \max_{\boldsymbol{u}'_k \in \boldsymbol{U}_k^i} \overline{E}_k^i \le m \max_{\boldsymbol{u}'_k \in \boldsymbol{U}_k^i} \overline{E}_k^i. \end{aligned} \tag{30}$$

Here, m is the maximal number of solutions. □

2.3 Deciding γ for sampling

We first consider how to find γ given an expected size W of $|\mathrm{S}|$. For a given W, we find γ by solving $\sum_{t=1}^{T} \min\left(\gamma \tilde{E}(\mathcal{T}), 1\right) = W$ using bisection since the left-hand side is monotonic to γ. Note that the time complexity of evaluating the left-hand side is $O(1)$ using a prefix sum. Therefore, the total time complexity of finding γ is $O(\log V)$, with V being the reciprocal accuracy.

In practice, caustics are often unevenly distributed spatially, with certain regions requiring significantly more budgets than others. Consequently, using a global constant W would be sub-optimal. Our bound of variance allows users to specify an expected $\overline{\mu_2}^*$, which we leverage to solve

$$\sum_{t=1}^{T} \frac{\tilde{E}^2(\mathcal{T})}{\min\left(\gamma \tilde{E}(\mathcal{T}), 1\right)} = \overline{\mu_2}^* \tag{31}$$

via bisection to find γ. This automatically adjusts the sample count to control the variance, achieving a satisfactory rendering result[3].

Our previous discussion utilizes population variance, i.e., the expected value of sample variance. However, sample variance can be significantly higher than population variance, especially with low sample counts, leading to undesirable fireflies even in low-energy regions. We recommend specifying the maximal sample variance $\overline{S^2}^*$. The solution for γ is analogous.

In general, automatically determining γ based on a variance limit enhances interpretability. However, our findings indicate that this approach does not yield a better rendering result compared to manually tuned γ. Consequently, we employ the automatic method in only one scene (Slab) for demonstration purposes.

3 ADDITIONAL DISCUSSIONS

In this section, we provide additional discussions of several issues regarding complexities, design choices, and extensions.

3.1 Complexity analysis

Since the sampling overhead is small, as we have analyzed in the performance statistics section, we only consider the complexity of the precomputation pass. Generally, the time complexity of the precomputation is a product of the number of primitive tuples and the complexity of bounding one primitive tuple.

[3]We decide the sample count a priori. This differs from adaptive sampling, which decides the number of samples after the generation of a part of the samples, which introduces correlation and could be unstable due to relying on variance estimations. At the same time, ours is uncorrelated and does not introduce bias. Moreover, our multi-sample estimator could reach zero variance even when the distribution is not accurate (i.e., tight) at the cost of a sufficient (but still finite) number of samples.

The number of primitive tuples. Assuming N the number of triangles in the scene and the number of specular bounces is $k-1$, the worst case number of tuples is N^{k-1}. However, due to our use of incremental tuple constructions, the number of primitive tuples is approximately NC^{k-2}, with C being a constant indicating how many triangles an outgoing beam will intersect on average, which ranges from 10 to 20 in our test scenes. When non-planar receivers are used, an additional C should be multiplied.

The time complexity of bounding one primitive tuple. For a single piece, the computational complexity is primarily determined by tensor convolution with the largest size. For example, for single refraction, we require convolving two 8×8 matrices. For single refraction, we need to convolve two $3 \times 3 \times 12 \times 12$ tensors, with the first two dimensions corresponding to the remainder variables. We use brute-force convolution since we found Fourier transforms easily lose accuracy. However, when scaled to double bounces, this quickly becomes impractical. Generally, the degree of position rational functions is approximately A^{k-1}, while the degree of irradiance is $4A^{k-1}$. Here, the constant A takes the value of 2 for reflection, 3 for refraction, and is doubled for interpolated normals. As aforementioned, we use degree reduction to keep the length of each dimension of the tensor below 40. Remainder variables are also replaced by a new one. Therefore, the final computation time for double refraction, on average, is only several milliseconds (single core) for each piece.

With degree reductions, the computation time for bounds on longer chains can still be maintained in a reasonable range. However, the limitation arises from the increasing number of tuples and looser bound, which can lead to overall performance issues.

Storage. The data used to compute the bound can be free immediately after we finish the computation of each piece, so the memory overhead is primarily the storage of bounds. This depends on the scene, which has been shown in the rendering statistics.

3.2 Design choices

Coefficients after domain subdivision. Transforming a rational function from its form before subdivision to its form after subdivision is fundamentally a linear transformation of polynomial variables. This transformation exhibits lower complexity, as it involves matrix multiplication for tensors, compared to recomputing from scratch through tensor convolution. Despite this efficiency, we still choose to recompute the functions. There are two primary reasons for this decision. First, in the case of single scattering, the degrees of the functions are relatively low, making the difference in computational approaches negligible. Second, for multiple scattering, certain approximations depend on the range of intermediate variables. After subdivision, these ranges change, necessitating recomputation to enhance accuracy. In other words, recomputation is essential if we aim to ensure that the bounds converge to the true value with infinite subdivision depth when such approximations are employed.

Power basis vs. Bernstein basis. The power basis often faces numerical issues, particularly the phenomenon where the transformation matrix between the two bases is ill-conditioned. Consequently,

for multiple scattering, we compute all polynomials using the Bernstein basis. However, for single scattering, we initially utilize the power basis and subsequently transform it to the Bernstein basis before calculating the bounds. This approach is advantageous because the coefficient matrix is upper triangular in the power basis, which helps to reduce computational costs.

Strictly conservative bounds. Our method focus a theoretically conservative bound. The sole exception is that we do not account for multiple solutions within a primitive tuple, which decision is based on our empirical findings and could be explored by future works. Nevertheless, the strict validity of the bounds may not be the most efficient approach. In practice, we believe that eliminating certain complexities, such as remainder variables and high degrees, could significantly accelerate the algorithm, enhancing its suitability for complex scenes.

3.3 Extensions

Glossy materials. Extending our method to accommodate glossy materials is a straightforward process. While we currently do not incorporate glossy receivers into our sampling probability design, integrating product importance sampling [Herholz et al. 2016] into our pipeline is feasible, albeit with an $O(|U|)$ cost due to the need to enumerate tuples and multiply the BSDF value by the probability [Fan et al. 2023]. For glossy chains, the addition of new dimensions to support normal offsets is all that is required.

Normal mapping. Conceptually, we should employ tessellation to generate a standard triangle mesh; however, it is not necessary for the tessellated mesh to be fully stored. The development of a specialized and more efficient method for high-resolution normal and displacement mapping remains a topic for future research. Exploring alternative rational coordinate mappings for other rational surfaces, such as splines, presents an intriguing avenue for future research.

4 IMPLEMENTATION DETAILS

In this section, we present the implementation details along with the accompanying pseudo code. All computations are performed in double-precision[4].

4.1 Pseudo code

Coefficients. Given a primitive tuple and a domain of u_1, the first step is to derive rational functions, which we implement as polynomials. First, we take a single reflection as an example.

```
def get_coefs_R(tseq, u1m, u1M, v1m, v1M):
    # ... Extract data from tseq (triangle tuple)
    # Note that P11 corresponds to e_{1,1} in the main paper. Likewise, N11 corresponds to n_{1,1}-n_{1,0}
    # The variables below are already (vector) polynomials
    # m: the lower bound. M: the upper bound
    p10 = P10 + P11 * u1m + P12 * v1m
    p11 = P11 * (u1M - u1m)
    p12 = P12 * (v1M - v1m)
    n10 = N10 + N11 * u1m + N12 * v1m
    n11 = N11 * (u1M - u1m)
    n12 = N12 * (v1M - v1m)

    x1 = p10 + p11 * u + p12 * v
    n1 = n10 + n11 * u + n12 * v

    d0 = x1 - x0
    d1 = d0 * (n1.dot(n1)) - n1 * (n1.dot(d0)) * 2

    # suffix p means numerators
    u2p = d1.cross(p22).dot(x1 - p20)
    v2p = (x1 - p20).cross(p21).dot(d1)
    k2 = d1.cross(p22).dot(p21)
    C = d0.dot(d0) # Denominator of the geometric term

    # Test of signs
    s0 = n1.dot(d0) * (-1) # Incident light comes from the front side
    s1 = (x1 - p20).cross(p21).dot(p22) * k2 # Ray goes forward, not backward

    return u2p, v2p, k2, C, s0, s1
```

We note that the above sign tests should be designed according to specific application cases, particularly regarding whether the mismatch between shading normals and geometric normals should be taken into account.

Position bounds. Then, we can obtain and check the bound of each rational function.

```
def positional_check_R(u1m, u1M, v1m, v1M, u2p, v2p, k2, s0, s1):
    # m: the lower bound. M: the upper bound
    # p.fbound(q): Bernstein bound of rational function p/q
    # p.bound():   Bernstein bound of polynomial p
    k2 = k2.align_degree_to(u2p)
    # Compute the bound for rational functions
    u2m, u2M = u2p.fbound(k2)
    v2m, v2M = v2p.fbound(k2)
    # Compute the bound for polynomial functions
    u2pm, u2pM = u2p.bound()
    v2pm, v2pM = v2p.bound()
    # suffix q means denominators
    u2qm, u2qM = k2.bound()
    s0m, s0M = s0.bound()
    s1m, s1M = s1.bound()

    # ... Try reciprocal if u2qm * u2qM < 0, e.g., computing k2.fbound(u2p)

    # If any of the following flags is true, return immediately
    # bad: the whole region (box) contains no valid paths
    bad_u2 = u2qm * u2qM > 0 and (u2M < 0 or u2m > 1 or v2M < 0 or v2m > 1)
    bad_u = u1m + v1m > 1
    bad_s = s0M < 0 or s1M < 0

    # If any of the following flags is true, we are in favor of continuing subdivisions. U_TOL is a tolerance threshold, which we set to 1 here
    # pbad: the region (box) contains some invalid paths
    pbad_u2 = u2qm * u2qM > 0 and (u2m < -U_TOL or u2M > 1 + U_TOL or v2M < -U_TOL or v2m > 1 + U_TOL)
    pbad_s = s0m < 0 or s1m < 0
    pbad_u1 = u1M + v1M > 1 + U_TOL
```

[4]Since degree reductions maintain polynomial degrees below 40, they contribute to improved numerical stability. It is important to note that these reductions not only enhance performance but also improve accuracy. We have observed instances where the original polynomials fail to yield a finite bound, whereas the reduced ones succeed.

Irradiance derivation. Computing the irradiance for single reflection is straightforward. We take a more complicated case, explicit differentiation for double refractions, for example.

```
def irradiance_explicit_TT(tseq, u1m, u1M, v1m, v1M):
    # ... Get polynomials with derivative-aware approximations
    u3p_du, u3p_dv, v3p_du, v3p_dv, k3_du, k3_dv = u3p.du(), u3p.dv(), v3p.du(), v3p.dv(), k3.du(), k3.dv()
    # ... Scale differentials using (u1m, u1M, v1m, v1M)

    # We store polynomial coefficients as 6D tensors, with the last 2D representing u, v
    # The first 4D is for the remainder variables
    # reduce_all(p, i): reduce p to linear in u and v, and put the new remainder variable to the i-th dimension
    u3p_du = reduce_all(u3p_du, 0)
    u3p_dv = reduce_all(u3p_dv, 0)
    v3p_du = reduce_all(v3p_du, 0)
    v3p_dv = reduce_all(v3p_dv, 0)
    k3_du = reduce_all(k3_du, 2)
    k3_dv = reduce_all(k3_dv, 2)
    k3 = reduce_all(k3, 1)
    u3p = reduce_all(u3p, 3)
    v3p = reduce_all(v3p, 3)

    # We must interleave the dimensions of different remainder variables to keep the bound valid
    u3u1p = u3p_du * k3 - k3_du * u3p
    u3v1p = u3p_dv * k3 - k3_dv * u3p
    v3u1p = v3p_du * k3 - k3_du * v3p
    v3v1p = v3p_dv * k3 - k3_dv * v3p
    denom = k3 * k3

    u3u1p = reduce_all(u3u1p, 0)
    v3v1p = reduce_all(v3v1p, 1)
    u3v1p = reduce_all(u3v1p, 2)
    v3u1p = reduce_all(v3u1p, 3)

    # The main part of irradiance is fp/fq
    fq = u3u1p * v3v1p
    fq = reduce_all(fq, 0)
    fq1 = u3v1p * v3u1p
    fq = fq - fq1
    fq = fq * C
    fp = denom * denom
    return fp, fq
```

Subdivision. The subdivision process can be characterized as a breadth-first search. The criteria for pruning and subdivision are elaborated in the preceding paragraph on position checks and implementation section within the main body of the paper. During the subdivision of a box domain, we partition it at its center along both the u_1 and v_1 axes, resulting in four smaller boxes.

Bound storage. Conceptually, for each primitive tuple, we store a bitmap of its irradiance contribution on the receiver plane.

```
# um, uM, vm, vM: the position bound
# val: the irradiance (upper) bound
def splat(um, uM, vm, vM, val):
    # RES: resolution
    a = max(0, min(RES, int(max(0.0, vm) * RES)))
    b = max(0, min(RES, int(min(1.0, vM) * RES) + 1))
    c = max(0, min(RES, int(max(0.0, um) * RES)))
    d = max(0, min(RES, int(min(1.0, uM) * RES) + 1))
    e = np.ones((b - a, d - c)) * val
    buf[a:b, c:d] = maximum(buf[a:b, c:d], e)
```

However, such a dense representation easily encounters memory issues, so we use a sparse representation, where each grid stores a list of primitive indices and corresponding irradiance values.

BVH. We implement the Bounding Volume Hierarchy (BVH) as a binary tree, using the longest edge of each node's bounding box to determine the splitting axis. The midpoint of this axis is used to partition the node into left and right sub-trees, continuing the recursive subdivision until a triangle is reached.

4.2 Experiment settings

We present our experiment details in Tables 1 and 2. Note that all reported γ and $\overline{S^2}^*$ do not include the intensity of light sources. In other words, the actual value should be $1/I_0$ and I_0^2 times our reported values, respectively. We generate reference images using specular polynomials [Fan et al. 2024] for single-bounce scenarios and Stochastic Progressive Photon Mapping (SPPM) [Hachisuka and Jensen 2009] for multiple bounces, with the exception of the Living Room scene, for which we employ Manifold Path Guiding (MPG) [Fan et al. 2023] at very high sample rates.

Table 1. Detailed setup for rendering experiments. For each scene, we show the path tracing depth, the threshold for the constraint checking threshold in the solver, the maximal subdivision level, and the choice of sampling parameters.

Scene	Max Depth	Threshold	γ	$\overline{S^2}^*$
Dragon (1 min)	10	10^{-5}	1000	N/A
Dragon (23 min)	10	10^{-5}	10	N/A
Plane	10	10^{-5}	100	N/A
Sphere	5	10^{-5}	10	N/A
Pool	5	10^{-5}	50	N/A
Slab	5	10^{-6}	N/A	10^{-4}
Diamonds	20	3×10^{-6}	2	N/A

Table 2. Summary of video scenes and parameters.

Scene	Max Subdiv. Level	γ
Plane I	12	200
Plane II	12	900
Sphere	3	10
Diamonds I	3	2
Diamonds II	3	2
Slab I	1	200
Slab II	3	200

REFERENCES

Zhimin Fan, Jie Guo, Yiming Wang, Tianyu Xiao, Hao Zhang, Chenxi Zhou, Zhenyu Chen, Pengpei Hong, Yanwen Guo, and Ling-Qi Yan. 2024. Specular Polynomials. *ACM Trans. Graph.* 43, 4, Article 126 (July 2024), 13 pages.

Zhimin Fan, Pengpei Hong, Jie Guo, Changqing Zou, Yanwen Guo, and Ling-Qi Yan. 2023. Manifold Path Guiding for Importance Sampling Specular Chains. *ACM Trans. Graph.* 42, 6, Article 257 (Dec 2023), 14 pages.

Toshiya Hachisuka and Henrik Wann Jensen. 2009. Stochastic Progressive Photon Mapping. *ACM SIGGRAPH Asia 2009 papers* (Dec. 2009), 1–8.

Sebastian Herholz, Oskar Elek, Jiří Vorba, Hendrik Lensch, and Jaroslav Křivánek. 2016. Product Importance Sampling for Light Transport Path Guiding. *Computer Graphics Forum* 35, 4 (2016), 67–77.

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. AAI9837162.