Vectorization for Fast, Analytic, and Differentiable Visibility

YANG ZHOU, University of California, Santa Barbara
LIFAN WU, NVIDIA
RAVI RAMAMOORTHI, University of California, San Diego
LING-QI YAN, University of California, Santa Barbara

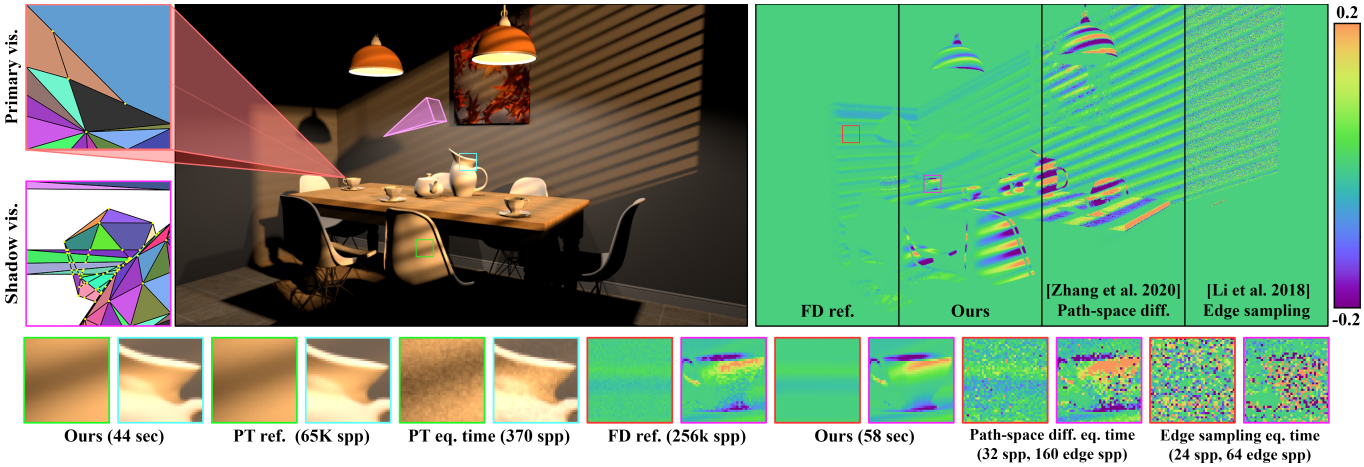


Fig. 1. We develop a new rendering method, vectorization, that computes analytic solutions to 2D point-to-region integrals in conventional ray tracing and rasterization pipelines. Our approach revisits beam tracing and maintains all the visible regions formed by intersections and occlusions in the beam (shown leftmost for primary visibility and shadows). This enables fast and analytic computation of direct lighting, including soft shadows (by tracing a beam from the shading point to the area light). Our result is computed in only 44 sec, and is noise-free, exactly matching the reference (compare to path tracing in the insets on the left). Crucially, this method enables automatic differentiation to compute exact derivatives with respect to any scene parameters (here the y-coordinate of the light), without any changes to our rendering pipeline. Our derivatives, including complex visibility gradients, are noise-free and one to two orders of magnitude faster than those obtained with previous differential methods (see insets; note that even with 256k samples per pixel, the finite difference reference fails to converge).

In Computer Graphics, the two main approaches to rendering and visibility involve ray tracing and rasterization. However, a limitation of both approaches is that they essentially use point sampling. This is the source of noise and aliasing, and also leads to significant difficulties for differentiable rendering. In this work, we present a new rendering method, which we call vectorization, that computes 2D point-to-region integrals analytically, thus eliminating point sampling in the 2D integration domain such as for pixel footprints and area lights. Our vectorization revisits the concept of beam tracing, and handles the hidden surface removal problem robustly and accurately. That is, for each intersecting triangle inserted into the viewport of a beam in an arbitrary order, we are able to maintain all the visible regions formed by intersections and occlusions, thanks to our Visibility Bounding Volume Hierarchy (VBVH) structure. As a result, our vectorization produces perfectly anti-aliased visibility, accurate and analytic shading and shadows, and most important, fast and noise-free gradients with Automatic Differentiation (AD) or Finite Differences (FD) that directly enables differentiable rendering without any changes to our rendering pipeline. Our results are inherently high-quality and noise-free, and our gradients are one to two orders of magnitude faster than those computed with existing differentiable rendering methods.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Visibility**.

Additional Key Words and Phrases: vectorization, beam tracing, anti-aliasing, differentiable rendering

ACM Reference Format:
Yang Zhou, Lifan Wu, Ravi Ramamoorthi, and Ling-Qi Yan. 2021. Vectorization for Fast, Analytic, and Differentiable Visibility. *ACM Trans. Graph.* 1, 1, Article 1 (January 2021), 20 pages. https://doi.org/10.1145/3452097

Authors' addresses: Yang Zhou, yzhou426@cs.ucsb.edu, University of California, Santa Barbara; Lifan Wu, lifanw@nvidia.com, NVIDIA; Ravi Ramamoorthi, ravir@cs.ucsd.edu, University of California, San Diego; Ling-Qi Yan, lingqi@cs.ucsb.com, University of California, Santa Barbara.

0730-0301/2021/1-ART1 $15.00
https://doi.org/10.1145/3452097

1 INTRODUCTION

In Computer Graphics, there are generally two approaches to render images: rasterization and ray tracing. Rasterization projects scene geometry onto the screen and breaks the geometry into pixels, while ray or path tracing casts rays into the scene and bounces them stochastically to find paths connecting the light and the camera.

It is well-known that rasterization is fast but prone to aliasing, and that ray tracing is high-quality but is slow and noisy. However, a bigger problem shared by both approaches is *point sampling*. The central point of each pixel is typically used to detect the coverage of geometry in rasterization, producing aliasing. The sampled paths in ray tracing are essentially points in the high dimensional path space, and such random point sampling of the path space introduces variance. Moreover, since the point samples are discrete, it is difficult to calculate the gradients of the rendering process with respect to scene parameters in order to enable differentiable rendering, especially with discontinuities that are commonly seen, such as boundaries of geometry and shadows.

An attempt to deal with the point sampling problem is beam tracing [Heckbert and Hanrahan 1984]. In beam tracing, a ray is extended to a frustum, and the goal is to find all the geometric primitives within the viewport of this frustum. However, beam tracing is far from practical. The biggest problem is that it does not attempt to deal with the complexity of geometry that intersects or resides in a beam. The original beam tracing [Heckbert and Hanrahan 1984] requires sorting a list of geometry intersecting the beam, which inevitably introduces the mutual occlusion problem as known in the painter's algorithm. Later variants of beam tracing try to steer away from this problem, either by simply forming beams with a starting point and a triangle, and assuming no visibility change inside these beams [Watt 1990; Duvenhage et al. 2010, 2014], or by splitting a beam into several smaller ones according to the edges of geometry primitives [Overbeck et al. 2007], introducing multiple BVH traversals of the scene that can significantly hinder high performance. Moreover, it is unknown how these methods would be extended to enable differentiable rendering.

In this paper, we revisit the concept of beam tracing, bringing out its merits that eliminate point sampling in the 2D point-to-region visibility problem, produce accurate and anti-aliased rendering results, and enable differentiable rendering automatically without any changes to the light transport computations. To achieve these goals, our high level idea is to maintain all the visible regions as we insert projected triangles into the viewport of the beam's frustum. This process is independent of the order of triangles inserted, and occlusions are handled precisely (see Fig. 1, leftmost, showing the precise beam frustum for primary visibility and shadows). Compared to rasterization that produces a bitmap, this process is more similar to drawing a vector graph, thus we name it *vectorization*.

The most difficult part of the vectorization process is the dynamic maintenance of the regions formed by the overlapped triangles projected onto the 2D image plane of a beam's viewport. To our knowledge, there is no convenient and efficient data structure previously that supports this, especially taking depths and occlusions into consideration. We present Visibility Bounding Volume Hierarchy (VBVH) that enables quickly locating intersecting primitives and dynamic splitting and merging to guarantee fast performance.

With our vectorization in §4, accurate visible regions are extracted as seen from the starting point of a beam. Then the rendering becomes a 2D point-to-region integration, which has analytic and accurate solutions and naturally removes aliasing. The benefits of such elimination of point sampling in the 2D domain are immediate, as we demonstrate in our applications in §5 on accurate primary visibility, soft shadows from area lights, anti-aliased shadows from point lights, and pure specular reflection. Our point-to-region integration also provides preliminary insights in solving the more general 4D region-to-region integration for direct illumination. More details and potential extensions will be discussed in Appendix A.

Moreover, with our vectorization, differentiation becomes straightforward without any other modifications to our vectorization process, using either Automatic Differentiation (AD) or Finite Differences (FD) (see Fig. 1, right, §6). This enables differentiable rendering with regard to any scene parameters without any noise, any additional cost from edge sampling [Li et al. 2018; Zhang et al. 2020] or reparameterization [Loubet et al. 2019]. Our differentiable renderer naturally supports zero-measure light sources (point lights) and BRDFs (pure specular reflection), which is challenging for existing methods. Thanks to the noise-free scene derivatives, we can utilize second-order optimization methods such as L-BFGS [Liu and Nocedal 1989] in inverse rendering applications and achieve faster convergence rates than first-order optimization methods such as stochastic gradient descent (SGD) or Adam [Kingma and Ba 2015]. With noisy gradients estimated by existing techniques, convergence using second-order optimization methods is not guaranteed.

To briefly summarize, our paper makes the following contributions to the rendering community:

- a novel vectorization pipeline different from both ray tracing and rasterization, eliminating point sampling in the 2D integration domain, producing fully analytic, accurate and anti-aliased point-to-region shading results,
- a hierarchical data structure for efficient, robust, depth-aware and order-independent insertion and maintenance of polygons, and most important,
- a differentiable rendering framework for visibility using only automatic differentiation or finite differences without special treatments to the rendering pipeline, resulting in noise-free gradients, which enables second-order optimization methods in inverse rendering for the first time, and orders of magnitude faster performance than existing approaches.

2 RELATED WORK

Beam tracing was proposed to exploit coherence among adjacent rays [Heckbert and Hanrahan 1984], so the original implementation of beam tracing will switch to ray tracing in the end. A similar concept to beam tracing is cone tracing [Amanatides 1984], which was originally designed as a naive version of ray differentials [Igehy 1999], but is currently often used for beam tracing with the shape of a cone rather than a frustum [Crassin et al. 2011]. Later extensions of beam tracing [Fortune 1999] consider geometric primitives intersected by a beam, but would require sorting the list of intersecting geometry from back to front. This introduces the mutual occlusion problem as known in the painter's algorithm. Overbeck et al. [2007] avoid the geometric complexity in a beam by splitting the beam into sub-beams along the edges of each triangle, but the number of sub-beams can be large and each of them will continue to be traversed in the scene. This significantly hinders high performance. Liu et al. [2011] extend beam tracing to handle nonlinear effects

such as curved reflection or refraction. None of the previous methods explicitly address the hidden surface removal problem, and it is also unknown how these methods would be extended to enable differentiable rendering. To our knowledge, our vectorization is the first method that can achieve both.

Visibility computation has been crucial for Computer Graphics since the beginning of the field. There were abundant studies in visibility surface determination in the 1970s, involving not only point-to-point visibility, but also point-to-region and region-to-region problems. Since our vectorization exploits beam tracing, it belongs to the point-to-region category. So global illumination is not handled by our method directly, because it involves the region-to-region problem discussed more in §7. For a complete overview of these visibility problems, we refer readers to the famous survey by Sutherland et al. [1974]. With the development of the hardware rasterization pipeline and ray tracing techniques, these traditional visibility methods were gradually replaced by point sampling. More recently, Durand [1999] provides a comprehensive review of the visibility problem in 3D. Nowrouzezahrai et al. [2014] propose a semi-analytical spherical integration scheme to determine the coverage of occluders. But their method is discrete, and is aimed at solving the binary visibility problem, instead of fully resolving all the different regions from different objects.

Hidden surface removal in Computational Geometry studies the point-to-region visibility problem in theory, and is closest to our vectorization. It deals with accurate occlusions of triangles in a 2D plane. Various methods are designed, and they can be categorized into two different types: online and offline. The online methods dynamically insert each triangle into the set of currently visible geometry, while the offline methods require storing all the triangles, processing them for more information, then inserting them sequentially. Our vectorization belongs to the online methods. Each intersecting triangle of a beam is processed immediately without additional storage, so the space complexity of our method is proportional to the number of visible triangles (§4.2).

A well-known hidden surface removal method is built on a data structure named the trapezoidal map [de Berg et al. 1997], where different regions on the planar subdivision are guaranteed to be trapezoids. It is online, and was originally invented for inserting line segments. Later, it has been extended to handle occlusions of triangles in the order of insertion or based on their depths [Mulmuley 1989, 1994; Goodrich 1992]. Other methods are also explored to solve the hidden line and hidden surface removal problem, such as the line-sweep based methods [Nurmi 1985; Sharir and Overmars 1992; McKenna 1987], which are vectorized and intrinsically similar to the rasterized scan conversion method in Computer Graphics. They are offline and require sorting all the vertices of 2D triangles.

However, these theoretical methods all suffer from severe numerical issues in practice. We emphasize that numerical robustness is a crucial problem rather than minor implementation detail in §3.2. And in §4.3, we demonstrate that our method achieves numerical robustness with a lightweight strategy, while only using finite precision representation. Finally, we present an extensive discussion in §7 on the relationship between our method and existing visibility methods as well as hidden surface removal methods.

Geometric anti-aliasing is related to several different branches of research. The first approach is to avoid the aliasing problem within pixels with a minimum amount of performance overhead. The related methods are usually seen in a rasterization pipeline. These methods are essentially still point sampling, either distributing more samples per pixel smartly [Wyman and McGuire 2017; Crassin et al. 2018] or reusing samples temporally (over time) [Yang et al. 2020]. The second approach is known as geometric level of detail [Loubet and Neyret 2017]. Methods in this category simplify the geometry to different levels, and choose a reasonably simplified geometry based on the distance from which it is viewed. These methods are also extended to guarantee smooth transition of both geometry and appearance under arbitrary view distances [Heitz et al. 2015; Zhao et al. 2016; Wu et al. 2019]. Finally, there are a few methods that derive specialized analytic formulae for certain simplified anti-aliasing problems so that stochastic sampling can be completely avoided [Auzinger et al. 2012; Manson and Schaefer 2013]. Our vectorization results in anti-aliased geometries, but is based on neither sampling nor simplification. Instead, our method accurately maintains all the geometries in an alias-free vector representation, regardless of their complexity.

Differentiable light transport computes the gradient of rendering results w.r.t. any input parameters of a scene, typically materials and transformations of objects, light sources and cameras. In an early work, Jalobeanu et al. [2004] introduced a differentiable rendering model. However, it is limited to differentiating pixel occupancy and hard shadows. To generalize its capability to compute derivatives with respect to arbitrary scene parameters, the key is to accurately detect and differentiate discontinuities, especially around geometric boundaries. This is because ray tracing is essentially point sampling, and point sampling is completely local, so no rays will know the existence of the boundaries, even if they are close.

Therefore, Li et al. [2018] develop an edge sampling approach to specifically sample the boundaries in the scene, but with a slow performance. To accelerate the computation, Loubet et al. [2019] come up with a reparameterization scheme, to fit a local change of parameters that approximately guarantees invariant boundaries, so the integration in the path space is then differentiable. This method is much faster, but is approximate and biased. Zhang et al. [2019] propose a differential theory for radiative transfer using the Reynolds transport theorem, and later extend it to work in path space [Zhang et al. 2020], which provides the state of the art performance, but requires precomputation and integration on boundaries. In concurrent work, Bangaru et al. [2020] present warped-area sampling that converts the integral at discontinuous boundaries to an area integral. Although their method is consistent and unbiased, it still has Monte Carlo noise in gradient images. Our method is compatible with the radiative backpropagation formulation proposed by Nimier-David et al. [2020], which can lead to better scalability.

All these previous methods require a certain level of modification to their rendering pipelines to enable differentiable rendering. In contrast, our vectorization directly supports differentiable visibility using automatic differentiation or simpler finite differences without any special treatments. Moreover, our method always generates noise-free gradients, and outperforms previous methods running an order of magnitude longer (Figs. 1, 10). The noise-free gradients

also enable using second-order optimization methods in inverse rendering for the first time.

Differentiable rasterization is widely used in 3D computer vision because of its relatively fast performance, but comes at the cost of inaccurate scene gradients due to the approximations to visibility [Liu et al. 2019; Ravi et al. 2020]. In concurrent work, Laine et al. [2020] present a high-performance differentiable rasterizer that can handle surface occlusion and edge anti-aliasing. However, it does not support visual effects caused by secondary rays such as soft shadows, while our method can handle these effects accurately.

Automatic differentiation (AD) provides an automatic way to keep track of the derivatives of variables of interest. It applies the chain rule to the user-specified computation. Therefore, AD can be treated as a black box without structural changes to the code, thus is easy to use, especially with support from mathematical libraries such as Eigen [Guennebaud et al. 2010] and Stan [Carpenter et al. 2015] on the CPU and Enoki [Jakob 2019] on the GPU. As mentioned earlier, our vectorization can be differentiated directly with the help of AD, and we choose Eigen in our implementation since other vector and matrix computations also depend on it.

Vector representation in images. It is also worth noticing that vector representation of images has already been applied, though these methods are not used to represent visibility. Apart from the classic curves and splines that are of infinite resolution, the Scalable Vector Graphics (SVG) [Ferraiolo et al. 2000] is able to represent vector images in different regions in an image. This is in essence similar to the result of our vectorization in the viewport. Also, the diffusion curve method [Orzan et al. 2008; Zhao et al. 2017] has been used to facilitate artists creating smoothly shaded vector images with gradient colors. Concurrent to our work, Li et al. [2020] present a differentiable rasterizer for vector graphics that allows the computation and backpropagation of per-pixel gradient with respect to the parameters of vector primitives.

3 OVERVIEW

In this section, we start by accurately defining the scope of the problems that our proposed method aims at. Then we briefly analyze the difficulties, to motivate our solution in §4.

3.1 The Point-to-Region Light Transport Problem

Many problems in light transport involve solving high dimensional integrals. In this work, we focus on direct illumination from area light sources, which can be formulated as a 4D region-to-region light transport integral:

$$I = \int_{\mathcal{P}} W(\boldsymbol{x}) \underbrace{\int_{A} L_e(\boldsymbol{y} \to \boldsymbol{x}) f_r(\boldsymbol{y} \leftrightarrow \boldsymbol{x} \leftrightarrow \boldsymbol{x}_c) G(\boldsymbol{x} \leftrightarrow \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y}) \, \mathrm{d}\boldsymbol{y}}_{=: L(\boldsymbol{x})} \mathrm{d}\boldsymbol{x}, \tag{1}$$

where $\mathcal{P}$ is the pixel coverage (i.e., pixel footprint), A is the set of area lights, $W(\cdot)$ is the pixel reconstruction filter, and $L(\boldsymbol{x})$ is the radiance received at a primary shading point $\boldsymbol{x}$. The received radiance $L(\boldsymbol{x})$ integrates over contributions from area light sources, in which $\boldsymbol{y}$ is a point on the area lights, L_e is the emitted radiance, f_r is the BRDF, $\boldsymbol{x}_c$ is the camera position, G is the geometry term, and V is the binary visibility function. Note that $L(\boldsymbol{x})$ is a 2D point-to-region integral involving visibility of scene geometry.

Finding an analytic solution to the full 4D region-to-region integral is challenging. So, in this paper, we present a *geometry vectorization* method that can compute its 2D point-to-region sub-problem, i.e., $L(\boldsymbol{x})$ in Eq. (1), in an analytic manner. Our method is based on beam tracing, which traces beams from a shading point $\boldsymbol{x}$ to the area lights. First, we reformulate the received radiance $L(\boldsymbol{x})$ as

$$\begin{aligned} L(\boldsymbol{x}) &= \int_{\mathcal{H}^2} L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) f_r(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \langle \boldsymbol{n}(\boldsymbol{x}), \boldsymbol{\omega}_i \rangle V(\boldsymbol{x}, \boldsymbol{\omega}_i) \, \mathrm{d}\boldsymbol{\omega}_i \\ &= \int_{Q(\boldsymbol{x})} L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) f_r(\boldsymbol{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \langle \boldsymbol{n}(\boldsymbol{x}), \boldsymbol{\omega}_i \rangle \, \mathrm{d}\boldsymbol{\omega}_i. \end{aligned} \tag{2}$$

We change the integral in Eq. (1) to the solid angle measure, where $\mathcal{H}^2$ is the hemisphere domain, $\boldsymbol{\omega}_i = (\boldsymbol{y} - \boldsymbol{x})/\|\boldsymbol{y} - \boldsymbol{x}\|$ is the incoming light direction, $L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) = L_e(\boldsymbol{y} \to \boldsymbol{x})$ is the incoming radiance, $\boldsymbol{\omega}_o = (\boldsymbol{x}_c - \boldsymbol{x})/\|\boldsymbol{x}_c - \boldsymbol{x}\|$ is the view direction, $\langle \cdot, \cdot \rangle$ represents the clamped dot product, $\boldsymbol{n}(\boldsymbol{x})$ is the shading normal, and $V(\boldsymbol{x}, \boldsymbol{\omega}_i)$ indicates whether a shadow ray that starts at $\boldsymbol{x}$ towards $\boldsymbol{\omega}_i$ is occluded or not. We further move the visibility function from the integrand to the integration domain by defining $Q(\boldsymbol{x}) = \{\boldsymbol{\omega} \mid V(\boldsymbol{x}, \boldsymbol{\omega}) = 1\}$ as a set of spherical polygons, which are formed by projecting all the visible regions of area lights onto the unit sphere centered at the shading point $\boldsymbol{x}$.

The key component of our method is computing all the visible regions of area lights $Q(\boldsymbol{x})$ within the beams using a binary Visibility Bounding Volume Hierarchy (§4). Finally in §5, we demonstrate how to compute an analytic solution to Eq. (2) given $Q(\boldsymbol{x})$. With our solution, a wide range of rendering applications can benefit from our analytic point-to-region integrals, as we demonstrate in §6. Although our method is not designed specifically for differentiable rendering, it allows elegant computations of noise-free scene derivatives with almost no additional effort using automatic differentiation, which is not possible previously.

3.2 Analysis and Motivation

From the problem formulation, we can see that it is ideal to directly solve the region-to-region light transport. However, how to find this 4D region-to-region integral analytically still remains unsolved, mainly due to the visibility function. To our knowledge, there is no efficient solution to the acquisition, representation, compression and analytic computation of such a complex 4D integral. A vast amount of literature [Durand 1999; Apostu 2012] has analyzed this problem, but still cannot fully solve it.

What we argue is that even for our point-to-region light transport, it is already of practical importance and is difficult to solve. We briefly provide evidence and insights on both, to motivate our solution next.

Practicality. In practice, most of the 4D region-to-region light transport problem can be factored out into several point-to-region sub-problems. This is not only because a pixel is usually small, but also thanks to the well-studied anti-aliasing problem in Computer Graphics.

With our analytic 2D point-to-region integral, we use a hybrid approach to estimate the 4D region-to-region integral. We first sample primary rays within a 2D pixel footprint $\mathcal{P}$, then trace beams

to compute the other 2D point-to-region integral. In terms of sampling primary rays, we use adaptive sampling similar to that in the Multi-Sampling Anti-Aliasing (MSAA) [Kirkland et al. 2002]. In most non-edge pixels, it suffices to sample only a single primary ray. Therefore, inspired by Wang et al. [2015], we group similar regions inside each pixel into at most 4 groups, and for each group, we perform beam tracing once. The performance overhead of adaptive sampling is no more than 1%, but the result is much better as compared to tracing one beam from the center of each pixel (demonstrated in Fig. 18). We provide further discussions for our hybrid approach to region-to-region integrals in Appendix A.

Difficulty. In Computational Geometry, there are two kinds of tasks: predicates and constructions. Predicates detect relationships between geometric primitives, including problems like detecting whether a point is on a line segment. When implementing them with floating-point numbers, the numerical inaccuracies are usually ignored or walk-arounded by ϵ-tweaking in engineering. On the other hand, constructions, such as the hidden surface removal problem, generate new geometric primitives based on predicates. They require much stronger guarantee on numerical robustness because otherwise, errors will accumulate and will lead to invalid control flow. We refer to the work of Kettner et al. [2008] for an extensive illustration of the failure cases in geometric computation due to floating-point numbers, and the work of Shewchuk [2013] for further discussions on geometric robustness.

While the Computational Geometry literature is usually theoretical and will simply ignore degeneracies, they happen frequently in Computer Graphics. For example, multiple triangles often share a common vertex in 3D models, which is a direct violation of their "general position assumption". In fact, we haven't found any existing implementation of hidden surface removal with finite precision arithmetic that will not crash in vectorizing the Stanford bunny model, including major software libraries such as CGAL [The CGAL Project 2020] and LEDA [Mehlhorn and Naher 1995].

The ultimate solution to the robustness problem is *Exact Geometric Computation* [Burnikel et al. 1995; Yap 1997; Li et al. 2005; Toth et al. 2017] taking advantage of exact arithmetic, but it will cause several orders of magnitude of performance degradation, which is unacceptable for us.

In summary, we emphasize that it is especially challenging to guarantee the robustness of algorithms that involve iterative geometric constructions depending on the logic decisions of geometric predicates. Such algorithms must ensure consistent evaluation of the predicates at all times, otherwise they may fail to execute properly. We keep this goal in the design of our hidden surface removal algorithm (§4.2). Meanwhile, numerical precision issues will always persist, because any finite number of bits will introduce imprecisions. We further alleviate such numerical issues by enforcing the convex invariant of our method, as described in detail in §4.3.

4 GEOMETRY VECTORIZATION

In this work, we focus on evaluating 2D point-to-region integrals in the following form (a generalized version of Eq. (2)):

$$I = L(\boldsymbol{x}) = \int_{Q(\boldsymbol{x})} h(\boldsymbol{\omega})\, \mathrm{d}\boldsymbol{\omega}, \tag{3}$$

Algorithm 1 Iterative computation of 2D visible polygons

function ComputeVisiblePolygons($\{\triangle_i\}_{i=1}^n$)
  $D = \{[-1,1]^2\}$ ▷ Initialize D with a square
  Initialize the VBVH tree(D) with a single root node
  for $i = 1$ to n **do** ▷ Add the triangles iteratively
    $\{p_k\}_{k=1}^m$ = Traverse($\triangle_i$, tree(D).root) ▷ Algo. 2
    for $k = 1$ to m **do** ▷ Iterate over the intersections
      $\widetilde{p}_k, r_k$ = Intersect($\triangle_i, p_k$)
        ▷ [O'rourke et al. 1998] chapter 7.6
      if $\widetilde{p}_k$ and r_k have different labels **then** ▷ Split
        Decompose r_k into convex regions $\{r_{k,j}\}$
        $D = D - p_k + \widetilde{p}_k + \{r_{k,j}\}$ ▷ Eq. 4
        Build a sub-tree T for $\widetilde{p}_k$ and $\{r_{k,j}\}$
          ▷ §4.2 splitting
        Substitute the node of p_k by the sub-tree T
        Update the bounding boxes on the affected nodes
      end if
    end for
    if All $\widetilde{p}_k$ have the same label **then** ▷ Merge
      $D = D - \{\widetilde{p}_k\} + \triangle_i$ ▷ Add the full triangle
      Find the leaf node of $\widetilde{p}_k$ that is closest to tree(D).root
        ▷ §4.2 merging
      Replace $\widetilde{p}_k$ by $\triangle_i$
      Remove the leaf nodes of other sub-regions
      Update the bounding boxes on the affected nodes
    end if
  end for
  return D
end function

where the integration domain $Q(\boldsymbol{x}) = \{\boldsymbol{\omega} \,|\, V(\boldsymbol{x}, \boldsymbol{\omega}) = 1\}$ is the visible polygons projected to the unit sphere centered at $\boldsymbol{x}$, and $h(\boldsymbol{\omega})$ represents a general spherical function whose explicit formulation depends on specific rendering applications (we will show a few examples in §5). Many problems in rendering, e.g. shading with area lights, can be formulated in this form of integral. The usual approach to solve this integral is Monte Carlo integration. Depending on the complexity of the integrand $h(\boldsymbol{\omega})$, it may require a number of samples. However, for some specific spherical functions used in rendering such as Linearly Transformed Cosines [Heitz et al. 2016], integrating them over spherical polygons yields analytic solutions, which is beneficial for noise-free anti-aliased rendering. Therefore, the integral evaluation boils down to computing the visible spherical polygons $Q(\boldsymbol{x})$.

Computing visible spherical polygons given scene geometries is challenging because of complex occlusions. We present an incremental approach (Algo. 1) to maintain a list of visible polygons in §4.1. To accelerate the intersection test of geometric primitives (triangles), which is a key step in the algorithm, we introduce a Visibility Bounding Volume Hierarchy (VBVH) in §4.2. Finally, we demonstrate how to make our algorithm more robust by mitigating the numerical issues caused by the limited precision of floating-point numbers in §4.3.

4.1 Computing Visible Polygons

The inputs of our method include a set of 3D triangles $\{\triangle_i^{3D}\}_{i=1}^n$ and a beam frustum. We assume that every pair of triangles are either disjoint or sharing vertices and edges, i.e., there is no penetration. The beam frustum is defined by a 4×4 perspective projection matrix. From a geometric point of view, a beam frustum can be interpreted as the viewport of a pinhole perspective camera (Fig. 2 (a0)), represented by an origin $\boldsymbol{x}$ and a rectangle at the far plane. Therefore, we can transform all the 3D triangles according to the perspective projection matrix. After the projection, all the relevant triangles (triangles out of the frustum are clipped) are in the normalized device coordinate (NDC) space, i.e., the x, y and z coordinates of triangle vertices are within $[-1, 1]$. The occlusion between the original 3D triangles (Fig. 2 (a2)) can be equivalently resolved using a set of 2D projected triangles $\{\triangle_i\}_{i=1}^n$ (taking the x and y coordinates, see Fig. 2 (b2)) and their depths $\{z_i\}_{i=1}^n$.

Using the coordinate transformation mentioned above, we convert the problem of finding 3D polygons that are visible within a beam frustum, to a simpler but equivalent problem of finding 2D visible projected polygons with the minimum depth values. This hidden surface removal problem is classical in computational geometry [Mulmuley 1989; Durand 1999]. However, previous methods focus on theoretical solutions and complexity analysis. They are prone to numerical errors, thus impractical to be used in rendering tasks. We present a practical algorithm to find visible polygons efficiently and alleviate the numerical issues as much as possible (§4.3). In addition, our algorithm directly supports automatic differentiation, enabling noise-free differentiable rendering with little additional implementation effort (§5.5).

We use an iterative approach to build a data structure $D = \{p_k\}_{k=1}^m$ recording all the visible polygons in 2D, as we add the triangles $\{\triangle_i\}_{i=1}^n$ one by one. To make the geometric computation easier, we require all of the polygons p_k to be *convex*. We denote D_i as the intermediate result after adding $\triangle_1, \triangle_2, \ldots, \triangle_i$. Initially, D_0 only has the square $[-1, 1]^2$, indicating the projection of the frustum viewport rectangle at the far plane (Fig. 2 (b0)). Without loss of generality, we assume the order of triangles to be random. The output of our algorithm is D_n, indicating all the 2D polygons with the minimum depths. They can be mapped to 3D visible polygons within the beam frustum by the inverse perspective projection transformation, and further projected on the unit hemisphere to get the integration domain $Q(\boldsymbol{x})$.

We now demonstrate the operation details at the i-th iteration that updates D_i by adding $\triangle_i$ (illustrated in Fig. 2). The pseudocode of our algorithm is demonstrated in Algo. 1.

Intersecting. Given a new triangle $\triangle_i$, we want to find the convex polygons $p_k \in D_{i-1}$ that overlap with $\triangle_i$, i.e., $\widetilde{p}_k = \triangle_i \cap p_k \neq \emptyset$ (line 5 of Algo. 1). We build a 2D Visibility Bounding Volume Hierarchy (VBVH) to speed up the intersection test and maintain this data structure as we add new triangles. We will discuss the details of the VBVH in §4.2.

Once we have the convex polygons p_k that overlap with $\triangle_i$, we can compute their overlapping regions $\widetilde{p}_k$ (line 7). We use the algorithm proposed by O'rourke et al. [1998] (please refer to chapter 7.6 of this book for algorithm details) to compute the intersection region $\widetilde{p}_k$ between a triangle and a convex polygon. Then, we calculate the residual region $r_k = p_k \setminus \widetilde{p}_k$ by 2D polygon differencing. The running time is linear in the number of polygon vertices.

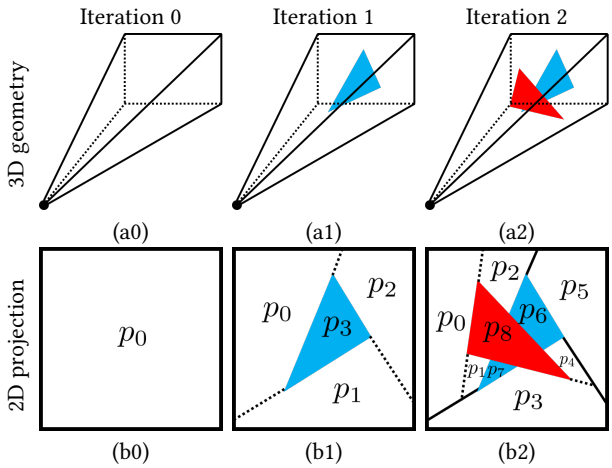


Fig. 2. Illustration of our geometry vectorization. The top row shows a beam frustum and the process of adding a blue and a red triangle. The bottom row demonstrates the corresponding 2D visible convex polygons as the triangles are added.

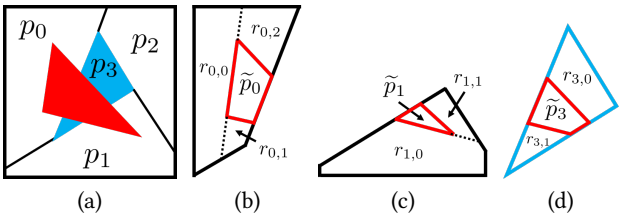


Fig. 3. Illustration of convex region splitting. (a) The red triangle overlaps with p_0, p_1 and p_3. (b–d) We compute the overlapping regions (red polygons) and split the residual non-convex polygons into convex ones.

Labeling. Each visible convex polygon p_k in D_{i-1} is associated with a *label* c_k, indicating that it belongs to the triangle $\triangle_{c_k}$. For every overlapping region $\widetilde{p}_k$, we compare the depth values of $\triangle_i$ and p_k (restricting on $\widetilde{p}_k$). Given the no-penetrating assumption, the new triangle $\triangle_i$ is either *behind* or *in front of* p_k. If $\triangle_i$ is in front of (closer to the beam origin) $\widetilde{p}_k$, we assign a new label i to the overlapping region. Otherwise the old label c_k is kept because the new triangle is invisible.

Note that sometimes it is unnecessary to use a different label for every individual triangle. For example, in the application of shading from area lights, the polygons representing occluders can share the same label. It can simplify the further polygon splitting operation and avoid keeping too many polygons in the data structure.

Splitting. For every convex polygon p_k in D_{i-1}, it can be decomposed into two disjoint regions $\widetilde{p}_k$ and $r_k = p_k \setminus \widetilde{p}_k$. If the two regions have the same label, no changes have to be made. If they have different labels (lines 9–16), we need to remove the original polygon p_k and add these two new regions $\widetilde{p}_k$ and r_k to the data structure. Note that we must ensure the added regions are convex. Clearly, $\widetilde{p}_k$ is a convex polygon since it is the intersection of a convex polygon and a triangle. The other part r_k may be non-convex, so we need to split it into multiple disjoint convex regions, i.e.,

Algorithm 2 Tree traversal to find convex regions intersecting $\triangle$

```
1: function TRAVERSE(△, node)
2:     if node is a leaf node then
3:         Let p_k be the convex region represented by this node
4:         if △ ∩ p_k ≠ ∅ then return {p_k} else return ∅
5:     end if
6:     if △ ∩ node.aabb = ∅ then return ∅
7:     return TRAVERSE(△, node.left) ∪ TRAVERSE(△, node.right)
8: end function
```

$r_k = \bigcup_j r_{k,j}$. We demonstrate an example of adding the red triangle and splitting regions in Fig. 3.

Though the general problem of splitting an arbitrary polygon into convex regions, called convex decomposition, is expensive to solve, we are facing a special case that leads to a simpler algorithm. The non-convex polygon r_k is formed by applying a mesh difference function on a convex polygon and a triangle in 2D. The geometric configurations are enumerable and splitting r_k will produce at most four convex sub-regions (Fig. 3 (b)). At last, the list of convex regions is updated as

$$D_i = D_{i-1} - \{p_k\} + \{\widetilde{p}_k\} + \{r_{k,j}\}. \tag{4}$$

Merging. If the new triangle $\triangle_i$ is completely visible, i.e., all the sub-regions $\widetilde{p}_k$ (note that $\triangle_i = \bigcup_k \widetilde{p}_k$) are visible and share the same label (the red triangle in Fig. 3), we can just add $\triangle_i$ to D_i by merging all of $\widetilde{p}_k$ ($\widetilde{p}_0$, $\widetilde{p}_1$ and $\widetilde{p}_3$ in Figs. 3 (b–d), lines 18–25). This approach can avoid unnecessary splittings and reduce the number of convex regions in the data structure.

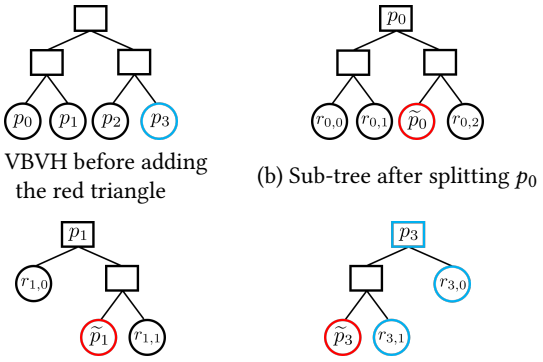
In theory, for any pair of adjacent convex polygons that share the same label, we can merge them if their union is still convex. This will lead to significant reduction of the number of polygons and make our algorithm more efficient. However, how to find the valid polygon pairs for merging without a large overhead is challenging. We leave it for future exploration.

4.2 Visibility Bounding Volume Hierarchy

To allow fast intersection tests, we use a binary Visibility Bounding Volume Hierarchy (VBVH) to store a set of 2D convex polygons. Each leaf node contains a visible convex polygon and its corresponding bounding box. For an interior node of a VBVH, we store the union of the bounding boxes of its children. The VBVH should be able to support the following operations.

Intersecting. To find the intersection regions of a triangle and a set of convex polygons, we perform a tree traversal (Algo. 2) on the VBVH starting from the root node. If a triangle intersects the bounding box of a VBVH node, we continue checking the intersection against its child nodes until they are disjoint. When we reach a leaf node, we compute the exact intersection region $\widetilde{p}_k = \triangle_i \cap p_k$, where p_k is the convex polygon represented by this leaf node.

Splitting. When we need to split a polygon into many sub-regions due to occlusion, we first find the corresponding leaf node of this polygon. Then, we substitute each leaf node by a sub-tree (Figs. 4 (b–d)), reflecting the convex polygon splitting (Figs. 3 (b–d)). The bounding boxes of the interior nodes are updated from bottom to top. We illustrate this operation in Fig. 4, which corresponds to the polygon splitting in Fig. 3. We also demonstrate the operation in lines 12–15 of Algo. 1.



(a) VBVH before adding the red triangle

(b) Sub-tree after splitting p_0

(c) Sub-tree after splitting p_1

(d) Sub-tree after splitting p_3

Fig. 4. Illustration of the region splitting operation on the VBVH. The rectangles represent interior nodes and the circles represent leaf nodes. (a) We show the VBVH before adding the red triangle (Fig. 2 (b1)). (b–d) For every region that requires splitting, we create the sub-trees according to the geometric configurations demonstrated in Figs. 3 (b–d). Finally, we substitute the leaf nodes shown in (a) by the corresponding sub-trees.

Merging. When the sub-regions of the newly added triangle share the same label and can be merged (the red leaf nodes in Figs. 4 (b–d)), we choose the node that is closest to the root node (e.g., $\widetilde{p}_0$ in Fig. 4 (b)) and replace its associated region by the full triangle (lines 20–22 of Algo. 1). Then, we remove all the other leaf nodes ($\widetilde{p}_1$ and $\widetilde{p}_3$ in Figs. 4 (c, d), line 23). The bounding boxes of the affected interior nodes should be also updated from bottom to top (line 24).

Discussion. Our method is efficient and able to finish (differentiable) rendering tasks in a few seconds. Also, though we did not specifically balance our VBVH, our closest-to-root heuristic in merging works well, and we haven't observed any performance issues in practice.

Because the order of the input triangles are random and each splitting operation on a polygon results in at most four sub-regions (meaning at most seven nodes on a binary tree), our VBVH will not have a prohibitively large number of nodes in total. The number of sub-regions also matches the prior hidden surface removal techniques using trapezoidal maps [Mulmuley 1989]. So, our algorithm shares the same complexity with those in theory. We refer to §6 for further discussions of time and space complexity.

We also briefly validate our VBVH in Fig. 5. As we can see, our method accurately and correctly handles cyclic occlusions and complex geometry inside one beam. For more validations and comparisons, please refer to §5 and §6.

4.3 Numerical Robustness

It is critical to ensure the numerical robustness of our method while maintaining its performance. Given its significant overhead, we cannot afford to rely on *Exact Geometric Computation*. Instead, we develop a lightweight strategy by enforcing the invariant of our system.

Our method works by partitioning space into convex regions. At each step, we compute intersection (difference) between the

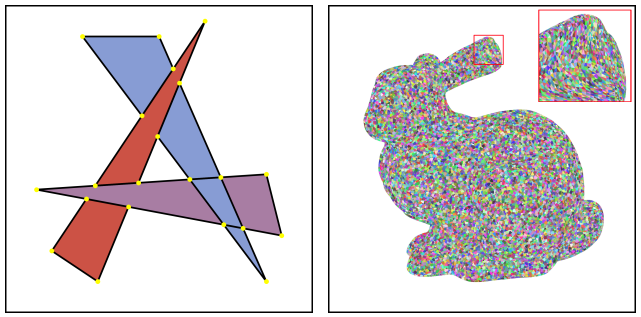


Fig. 5. Validation of our VBVH and color-coded visualization of regions using indices of triangles. (Left) Three cyclic overlapping triangles. (Right) An entire *Stanford bunny* in one pixel (with a close-up crop).

incoming triangle and existing regions. Our method will terminate successfully if the following two conditions are met, which we examine respectively:

(1) The intersection algorithm terminates successfully when inputs are valid convex polygons.

(2) All regions produced by Algo. 1 are convex.

Most polygon intersection algorithms, including the one we use, rely on the basic orientation predicate. The orientation predicate, denoted by $orient(a, b, c)$, detects whether three 2D points $a = (a_x, a_y)$, $b = (b_x, b_y)$, and $c = (c_x, c_y)$ form a left turn, a right turn, or a straight line (collinearity), and is defined as the sign of a determinant:

$$orient(a, b, c) = \text{sign}\left(\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{vmatrix}\right). \tag{5}$$

However, it is well understood that such a simple predicate cannot be robustly implemented by naïvely using floating-point numbers [Kettner et al. 2008]. A naïve floating-point implementation produces erroneous and inconsistent results when three points are close to collinear, which causes the intersection algorithm to either fail catastrophically (crash or hang), or generate garbage output (e.g. missing vertices).

To guarantee the correctness of the orientation predicate, and satisfy condition (1), we represent polygons by a simple fixed-point number scheme. Input floating-point coordinates are scaled by a constant factor C and rounded to integers. Typically, the input coordinates are within $[-1, 1]$, so we effectively discretize the range with a step size of $1/C$. The scaling factor C is chosen to provide enough precision, while not causing overflow. In our implementation, we set $C = 2^{20}$. By applying this scheme, the orientation predicate becomes exact, and the intersection algorithm is guaranteed to finish successfully.

To satisfy condition (2), we observe that violations can happen at two places. First, an input triangle of Algo. 1 may become degenerate or have its winding order flipped during the initial conversion to fixed-point representation. This case is easy to detect and fix. Second, when computing intersection, a new vertex may not be representable by an integer. Rounding the new vertex may change the topology of the output polygon, such as making it non-convex or self-intersecting. If such cases happen, we satisfy condition (2) by replacing the output polygon with its convex hull, which has a very close shape. The overhead of convex hull computation is negligible because typically the violating polygon only has a few vertices.

We demonstrate such an example in Fig. 6. The red polygon $F_0F_1F_2F_3$ is represented by floating-point numbers. With fixed-point numbers, the polygon vertices are rounding to their nearest grid points, resulting in the green polygon $G_0G_1G_2G_3$. However, the polygon becomes non-convex after the discretization. We resolve this problem by computing the convex hull of the non-convex polygon [O'rourke et al. 1998], yielding a triangle $G_0G_1G_3$. Therefore, we convert the red polygon ($F_0F_1F_2F_3$) represented by floating-point numbers to the green polygon ($G_0G_1G_3$) represented by fixed-point numbers. Though the two polygons are not exactly the same, the difference is negligible when we have a high resolution of discretization.

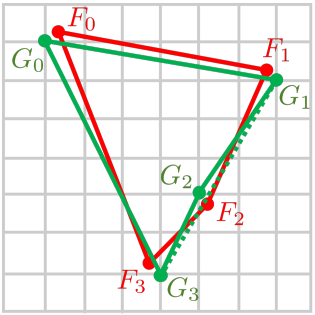


Fig. 6. We demonstrate a polygon represented with floating-point numbers (red) and another represented with fixed-point numbers (green). The green polygon is not exactly the same as the red one due to the vertex offset caused by rounding.

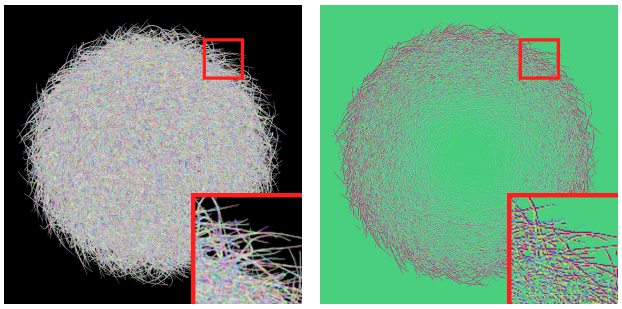
We have shown that our method always terminates successfully by satisfying the two invariant conditions. Therefore, we conclude that our method is theoretically accurate *within the limit of numerical precisions*. This immediately indicates that there will be no visible artifacts such as cracks generated by our method. And in practice, the numerical accuracy of our method is sufficient for all rendering applications we consider.

5 APPLICATIONS

Our vectorization leads to analytic solutions to 2D point-to-region integrals, avoiding Monte Carlo sampling. In this section, we present several practical applications that directly benefit from our technique. We start from a toy problem, demonstrating anti-aliased primary visibility with our analytic point-to-region integrals (§5.1). Then, we show three practical rendering tasks: accurate shading from area lights (§5.2), anti-aliased shadows from point lights (§5.3), and pure specular surface reflection (§5.4). Finally, we present our main application, noise-free differentiable rendering (§5.5) by extending our vectorization algorithm with automatic differentiation. Unlike existing differentiable rendering techniques [Li et al. 2018; Loubet et al. 2019; Zhang et al. 2020], our method does not require any special treatments to the forward rendering process.

5.1 Toy Problem: Anti-Aliased Primary Visibility

Primary visibility is used to determine the occupied area of visible polygons inside a pixel. For traditional rasterization or ray tracing methods, it is common to get aliased results due to the insufficient sampling rates, especially at the object's boundary regions that have many sub-pixel geometric details. While supersampling methods



(a) Pixel color (b) Derivative

Fig. 7. Primary visibility result of the *hairball* scene. Random colors are assigned to each triangle. The model has 2.88M triangles and is challenging for anti-aliased rendering. (a) Image rendered by our method, which is accurate even at the complex boundary regions. (b) Noise-free derivative of the pixel values with respect to the model moving away from the camera (mapped to false color for visualization).

can alleviate the aliasing problem at the cost of expensive computation, it is straightforward for our vectorization method to completely eliminate aliases and produce accurate pixel coverage values.

For each pixel on the image plane, we create a beam that originates from the camera and goes exactly through the pixel. Assuming the pixel is a square of $[-1,1]^2$, the pixel color is

$$I = \int_{[-1,1]^2} \alpha(\boldsymbol{u}) c(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}, \tag{6}$$

where $\alpha(\boldsymbol{u})$ is the binary function indicating whether a position $\boldsymbol{u}$ inside the pixel is covered by the projected triangles and $c(\boldsymbol{u})$ is the color of the triangle. Let $\{p_k\}_{i=1}^m$ be the set of 2D regions generated by our method (Algo. 1) and $\{c_k\}_{i=1}^m$ be the corresponding colors. [1] The pixel color can be reformulated as a summation

$$I = \sum_{k=1}^{m} c_k \underbrace{\int_{p_k} \mathrm{d}\boldsymbol{u}}_{=: \mathrm{Area}(p_k)}, \tag{7}$$

where the region areas $\mathrm{Area}(p_k)$ can be evaluated analytically given the polygon vertices. So we can compute the exact pixel colors and produce anti-aliased images. Figure 7 (a) shows a *hairball* model that is generally believed difficult for anti-aliased rendering, but our method can produce an accurate image efficiently (Fig. 7 (b)). In the supplementary video, we show an animation sequence in which the hairball model is moving away from the camera. It is temporally stable without flickering.

Note that this application is only a simple proof of concept. We acknowledge that for this specific problem (anti-aliasing), there are many other potential (differentiable) solutions [Laine et al. 2020]. A full comparison is beyond the scope of this paper.

5.2 Accurate Shading from Area Lights

As mentioned in §3, our vectorization allows accurate computation of the 2D point-to-region integral that represents the shading from an area light. We assume that the area light emits constant radiance L_e. According to Eq. (2), the received radiance at a shading point $\boldsymbol{x}$ from area lights [2] is

$$I = L(\boldsymbol{x}) = L_e \int_{Q(\boldsymbol{x})} f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \langle \boldsymbol{n}, \boldsymbol{\omega}_i \rangle \, \mathrm{d}\boldsymbol{\omega}_i. \tag{8}$$

To compute $Q(\boldsymbol{x})$, we create a beam from the shading point $\boldsymbol{x}$ to the area light sources (see the purple beam frustum on the wall in Fig. 1). We use Algo. 1 to find the visible regions representing the area lights. As mentioned in §4.1, we find the corresponding polygons in 3D by the inverse perspective projection and project them onto the unit sphere centered at $\boldsymbol{x}$, resulting in a set of spherical polygons $\{Q_k\}_{k=1}^m$. Then, we can rewrite the shading integral in Eq. (8) as

$$I = \sum_{k=1}^{m} L_e \underbrace{\int_{Q_k} f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \langle \boldsymbol{n}, \boldsymbol{\omega}_i \rangle \, \mathrm{d}\boldsymbol{\omega}_i}_{=: I_k}. \tag{9}$$

The integrals over spherical polygons I_k can be approximated analytically using Linearly Transformed Cosines (LTCs) [Heitz et al. 2016]. We fit the cosine-weighted BRDF (the integrand of I_k) by an LTC with the transformation matrix M. According to the work of Baum et al. [1989] and Heitz et al. [2016], the integral I_k has an analytic solution:

$$I_k = E(\widetilde{Q}_k) = \frac{1}{2\pi} \sum_{j=1}^{N} \arccos(\langle \boldsymbol{q}_j, \boldsymbol{q}_{j+1} \rangle) \left\langle \frac{\boldsymbol{q}_j \times \boldsymbol{q}_{j+1}}{\|\boldsymbol{q}_j \times \boldsymbol{q}_{j+1}\|}, \hat{\boldsymbol{z}} \right\rangle, \tag{10}$$

where E is the irradiance of the polygon $\widetilde{Q}_k = M^{-1} Q_k$, $\boldsymbol{q}_j$ is the j-th vertex[3] of the polygon $\widetilde{Q}_k$, and $\hat{\boldsymbol{z}} = (0,0,1)$. Combining Eqs. (9, 10), shading from area lights can be evaluated analytically.

So far, our focus has been on the analytic evaluation of the 2D shading integral at a single point $\boldsymbol{x}$. However, computing image pixel values requires another 2D integral over all the shading points inside a pixel, which makes the problem region-to-region. We use a practical hybrid approach (§3) to compute the 4D region-to-region integral for direct illumination (Fig. 8) and have more discussions about this hybrid method in Appendix A.

5.3 Anti-Aliased Shadows from Point Lights

Beside direct illumination from area lights, which can be naturally formulated as a region-to-region problem and be reduced to 2D point-to-region integrals, several rendering applications such as anti-aliased shadows from point lights (§5.3) and pure specular reflection (§5.4) can also benefit from this formulation. The shading of a polygonal surface patch that is seen through a pixel and is lit by a point light can be formulated as a 2D integral:

$$I = \int_{\mathcal{P}} W(\boldsymbol{x}) f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) \langle \boldsymbol{n}(\boldsymbol{x}), \boldsymbol{\omega}_i \rangle V(\boldsymbol{x}, \boldsymbol{\omega}_i) \, \mathrm{d}\boldsymbol{x}. \tag{11}$$

We follow the notation convention in Eqs. (1, 2). Note that the light direction $\boldsymbol{\omega}_i = (\boldsymbol{x}_l - \boldsymbol{x})/\|\boldsymbol{x}_l - \boldsymbol{x}\|$ depends on both the point light position $\boldsymbol{x}_l$ and the shading position $\boldsymbol{x}$. Assuming the pixel

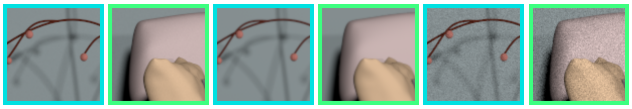
[1] We consider the background as one of these regions.

[2] We omit the shading point $\boldsymbol{x}$ in the integrand since it is fixed.

[3] We define $\boldsymbol{q}_{N+1} = \boldsymbol{q}_1$.

(a) Ours

(b) Path tracing



(c) PT reference (d) Ours (e) PT equal time

Fig. 8. Comparisons of shading from area lights using the *Living Room* scene. (a) Image rendered with our method. (b) Split image rendered with path tracing (having only direct illumination): (left) reference, (right) equal time. Image insets are shown in (c–e).

footprint is small enough compared to the size of the scene, we can approximate Eq. (11) by splitting the integral into two parts:

$$I \approx \underbrace{\int_{\mathcal{P}} W(\boldsymbol{x}) f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) \langle \boldsymbol{n}(\boldsymbol{x}), \boldsymbol{\omega}_i \rangle \, \mathrm{d}\boldsymbol{x}}_{=: I_{\text{unshadow}}} \cdot \underbrace{\int_{\mathcal{P}} V(\boldsymbol{x}, \boldsymbol{\omega}_i) \, \mathrm{d}\boldsymbol{x}}_{=: I_{\text{vis}}}. \tag{12}$$

The first integral represents the unshadowed contribution, where the integrand terms vary smoothly. It usually can be evaluated easily by taking one or a few samples on the pixel footprint. To compute the visibility integral in the second part, we trace a narrow beam from the light that encloses the patch, and compute the regions that are visible to the point light $Q(\mathcal{P}) = \{\boldsymbol{x} \mid V(\boldsymbol{x}, \boldsymbol{\omega}_i) = 1\}$ by our vectorization algorithm. Therefore, the second visibility integral becomes the area of the visible polygons,

$$I_{\text{vis}} = \int_{Q(\mathcal{P})} \mathrm{d}\boldsymbol{x}, \tag{13}$$

which has analytic solutions given polygon vertices.

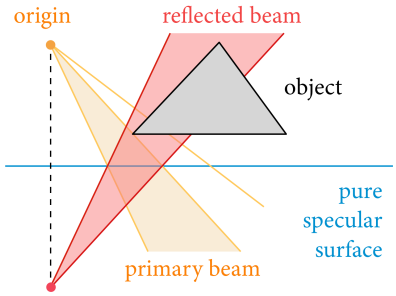


Fig. 9. Illustration of pure specular reflection. We first trace a primary beam (the orange one) through a pixel and compute all the visible polygons using our vectorization method. For the polygons that correspond to diffuse or glossy surfaces (covered by the unfilled orange beam), we evaluate the shading as described in §5.2 and §5.3. For each of the remaining polygons with a pure specular material (covered by the filled orange beam), we construct a reflected beam (the red one) by mirroring the beam origin and setting the specular polygon as the near-clipping plane.

In practice, instead of computing the visible polygons $Q(\mathcal{P})$ for each pixel footprint, we precompute a *vectorized shadow map* that records all visible polygons from all directions as viewed from the light position. For example, if we parameterize our vectorized shadow map as a cube map, we trace six beams and store all the visible polygons accordingly. For each pixel footprint, we query the vectorized shadow map and compute I_{vis} by finding its intersection area with the visible polygons. This significantly saves beam tracing time during rendering at the cost of extra storage for vectorized shadow maps.

5.4 Pure Specular Reflection

Our method can compute reflection from pure specular surfaces in a similar way to the original beam tracing [Heckbert and Hanrahan 1984]. The key operation is constructing the reflected beams during tracing, illustrated in Fig. 9. Initially, we trace a primary beam through a pixel and compute all the visible polygons within the beam using our vectorization method (§4). For the polygons with diffuse or glossy materials, we evaluate the shading as described in §5.2 and §5.3. For each of the remaining polygons with a pure specular material, we construct a reflected beam by mirroring the beam origin and setting the specular polygon as the near-clipping plane. We keep tracing reflected beams until there are no pure specular surfaces inside or reaching a maximum bounce count. Since we explicitly handle the Dirac delta BRDFs by tracing reflected beams, the equation of the final pixel color can be reduced to Eqs. (9, 12) and be computed analytically.

5.5 Differentiable Rendering

The accurate anti-aliased rendering applications are straightforward byproducts that benefit from our vectorization algorithm. But the most significant application of our method is to directly enable noise-free differentiable rendering with minimal extra effort. We can consider the rendering process as computing a radiometric measurement function $I(\theta)$ that maps from the scene parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_M\}$ including all the geometry, material and lighting

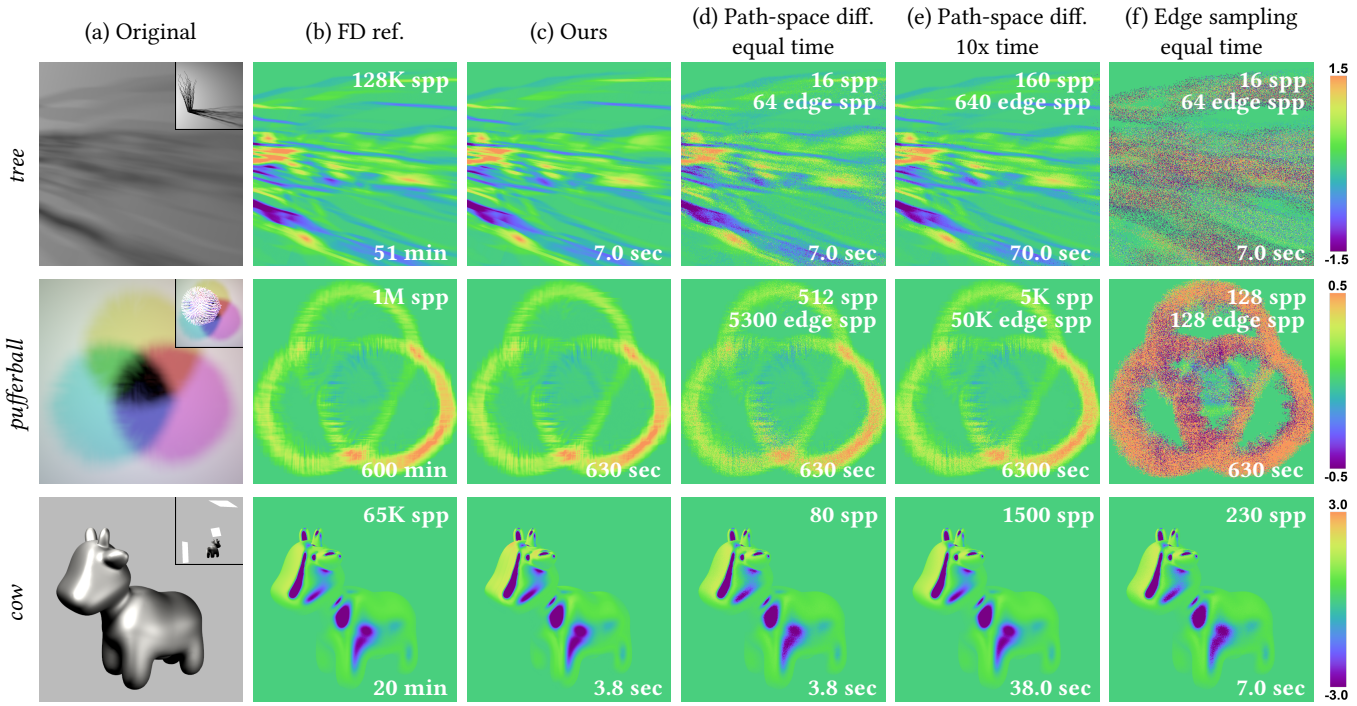


Fig. 10. Comparison of derivative computations of various differentiable rendering methods for the *tree* scene on top w.r.t the tree rotation around the up axis, the *pufferball* scene in the middle w.r.t. the object moving vertically downward, and the *Cow* scene at bottom w.r.t. the BSDF roughness increasing. The scene configurations are shown as the top-right insets in column (a). Our method is noise-free and produces accurate results, indistinguishable from the finite difference reference. In contrast, path-space differentiable rendering and edge sampling are noisy at equal time, and even path-space differentiable rendering with an order of magnitude more samples/time has residual noise (edge sampling would need about two orders of magnitude more samples to produce comparable results).

configurations onto the pixel value. In differentiable rendering, the main goal is to evaluate the derivatives of the pixel values with respect to scene parameters such as object positions and material roughness, i.e. $\partial I/\partial\theta_j$.

The biggest problems for differentiable rendering are geometric discontinuities. This is because the Monte Carlo rendering process is essentially generating point samples of the integrand, i.e., estimating the integral in Eq. (8). Differentiating this integral will introduce an additional boundary integral due to the geometric discontinuities. For this reason, most state-of-the-art work on differentiable rendering focuses on how to accurately handle the discontinuities. They either introduced expensive unbiased estimates of the boundary integrals [Li et al. 2018; Zhang et al. 2019, 2020], or used a biased reparameterization technique to avoid the expensive boundary integral computation [Loubet et al. 2019].

In contrast, with our vectorization method, the scene derivatives can be computed directly using automatic differentiation (AD), even for the difficult cases involving point lights and pure specular surfaces. We first demonstrate how to differentiate the shading from area lights (§5.2). The shading integral can be approximated analytically using LTCs (combining Eqs. (9, 10)):

$$I = L_e \sum_{k=1}^{m} E(M^{-1}Q_k), \tag{14}$$

where L is the light radiance, E is the analytic irradiance function of a polygon, M is the transformation matrix for LTCs, and Q_k is the spherical polygon corresponding to the visible area lights. Given the analytic expression, differentiating I is straightforward by using the chain rule. The derivative evaluation can be easily done via automatic differentiation. The derivative results are accurate, though with a very small deviation caused by the usage of fixed-point numbers (§4.3) and the LTC approximation (§5.2).

Extending our vectorization method to support differentiable rendering requires minimal effort. For the variables in Algo. 1, we only need to replace their regular data types (32-bit integer or float) by the data types that support automatic differentiation, which keep track of not only the original values but also the derivatives with respect to certain scene parameters. For example, we want to compute the derivatives with respect to geometric parameters such as the position of an object. Initially, we know the derivatives (local velocities) on the mesh vertices. After running our vectorization algorithm with AD, the derivatives are propagated to the visible spherical polygons so we have $\partial Q_k/\partial\theta$. Finally, evaluating Eq. (14) using AD gives our goal $\partial I/\partial\theta$, as the derivatives will be updated automatically. We show the derivative images with respect to geometric parameters in Figs. 1, 7, 10 (top and middle rows).

Thanks to the simplicity of our analytic expression (Eq. (14)), derivatives with respect to the lighting and geometric parameters are convenient to compute. However, it is not obvious how to compute

the derivative with respect to the material roughness r. Note that the LTC transformation matrix M is tabulated and precomputed by a set of roughness values in $[0, 1]$. [4] For an arbitrary roughness r that falls into the interval $[r_0, r_1]$, we linearly interpolate the LTC transformation matrix as

$$M(r) = \frac{r - r_0}{r_1 - r_0} M(r_1) + \frac{r_1 - r}{r_1 - r_0} M(r_0), \tag{15}$$

which leads to

$$\frac{\partial M(r)}{\partial r} = \frac{M(r_1) - M(r_0)}{r_1 - r_0}. \tag{16}$$

Therefore, we can have $\partial I / \partial r$ by computing Eq. (14) using AD with Eq. (16) as the initial derivative. We show the derivative images with respect to the surface roughness in Fig. 10 (bottom row).

Furthermore, existing differentiable rendering techniques usually assume that there are no point lights and pure specular surfaces in the scenes, because they introduce additional discontinuities due to the Dirac delta functions. It requires special treatments to handle zero-measure light sources and BRDFs in differentiable rendering (computing additional integrals on the discontinuous boundaries caused by the Dirac delta functions), and is too complicated to sample and evaluate efficiently. However, our method can still handle these cases naturally. In terms of differentiating the shading of point lights (§5.3) in Eq. (12), it is straightforward to compute $\partial I_{\text{vis}} / \partial \theta$ with automatic differentiation, since I_{vis} is an analytic equation. For the first integral I_{unshadow} on the RHS, we can differentiate it as

$$\frac{\partial I_{\text{unshadow}}}{\partial \theta} = \int_{\mathcal{P}} \frac{\partial [W(\boldsymbol{x}) f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) \langle \boldsymbol{n}(\boldsymbol{x}), \boldsymbol{\omega}_i \rangle]}{\partial \theta} \, \mathrm{d}\boldsymbol{x}, \tag{17}$$

because the integrand of I_{unshadow} is continuous in the integration domain $\mathcal{P}$. Therefore, we can estimate $\partial I_{\text{unshadow}} / \partial \theta$ along with I_{unshadow} by applying AD to the integrand function. Finally, differentiating the rendering result with pure specular reflection (§5.4) is also straightforward because we can reduce it to differentiating the analytic shading equations from area lights or point lights, which has just been discussed above. We show the derivative images with point lights and pure specular surfaces in Figs. 11 and 12, respectively.

6 RESULTS

In this section, we provide rendering results and detailed comparisons. We implement our algorithm in the Mitsuba renderer [Jakob 2010], and compare our vectorization algorithm against path tracing for performance validation. For differentiation, we compare with the edge sampling method [Li et al. 2018] and the path-space differentiable rendering method [Zhang et al. 2020].[5] All timings are measured on an 8-core Intel i9-9900K CPU, hyperthreaded to 16 threads, with 32 GB of main memory. Unless otherwise specified, all timings correspond to 720P renderings for rectangular images and 512×512 resolution for square images.

In Table 1, we report the complexity of each scene, along with the number of variables to differentiate and the rendering time w/ and w/o gradients using automatic differentiation.

[4] It is also tabulated in another dimension, the cosine value θ_o between the view direction and the normal. We do not discuss it here because it is not relevant.

[5] We do not compare to the reparameterization method [Loubet et al. 2019] since it uses a biased estimator for visibility derivatives. The code of concurrent work by Bangaru et al. [2020] is not available yet.

Table 1. Scene Configuration.[6]

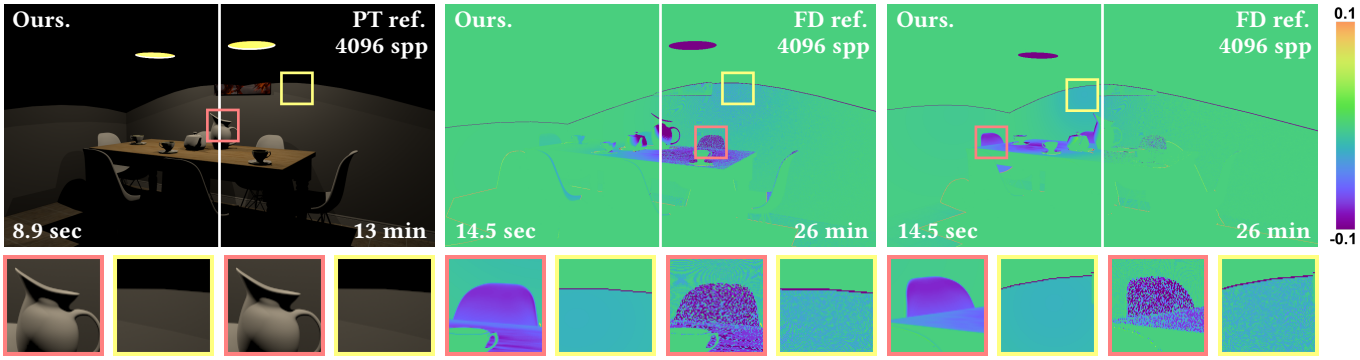
Scene	#Tris	#Lights	#Vars	Time w/ & w/o AD
Kaleidoscope	174	1	1	3.6 / 2.4 sec
Displacement	174	3	162	7.6 / 0.5 sec
Cow	23k	3	1	3.8 / 2.1 sec
Tree	24k	1	1	7.0 / 5.8 sec
Dining Room	270k	3	1	58 / 44 sec
Gears	290k	1	15	10.1 / 2.9 sec
Living Room	786k	3	1	31 / 24 sec
Dragon	832k	1	4	13 / 6.1 sec
Pufferball	1.06m	3	1	630 / 520 sec
Hairball	2.88m	0	1	90 / 80 sec

Shading from area lights. In Figs. 1 and 8, we show the direct illumination of the *Dining Room* scene and the *Living room* scene, respectively. Both scenes are illuminated with 3 area lights. These scenes are challenging in the complex visibilities for soft shadows, as seen from each shading point to the area lights. We apply our hybrid method to handle the primary visibility and soft shadows simultaneously. We trace a beam towards each pixel in the image plane to find primary visibility, then we trace multiple beams from the coverage of each pixel in the scene towards each of the area lights.

As we can tell from the insets comparing with path tracing, our method produces noise-free soft shadows correctly. Also note that both scenes contain hundreds of thousands of triangles. To our knowledge, this level of complexity for hidden surface removal has never been practically dealt with previously, in either the Computer Graphics or the Computational Geometry communities. In contrast, our vectorization handles the soft shadows from these scenes not only robustly but also efficiently. We show the performance of our method with a detailed break down of the time spent in each step in Table 2. As can be observed, the shadow computation takes up most of the computation time. This is reasonable because the beams toward area lights usually subtend much larger solid angles as compared to beams through pixels.

Shading from point lights. In Fig. 11, we show the direct illumination of the same *Dining Room* scene, this time illuminated with 2 point lights. We build a vectorized shadow map for each light that allows fast visibility query without tracing beams during rendering. Our method is able to obtain exact, anti-aliased shadows (see the insets of Fig. 11) without Monte Carlo noise and common shadow mapping artifacts such as jagged boundary and shadow acne. Zero-measure light sources are usually not supported by existing differentiable rendering techniques due to additional integrals caused by the discontinuous shadow boundaries [Li et al. 2018; Zhang et al. 2020]. However, it is straightforward for our method to support point lights via AD. We show two derivative images with respect to the vertical translation of each point light. We are able to capture the strong variation around the shadow boundary, as well as the smooth variation on the lit surfaces. Our results achieve better quality than the path traced reference using FD, which takes

[6] We consider all the emitting faces in the *kaleidoscope* scene as one area light. We only compute the primary visibility of the *Hairball* scene without lighting.

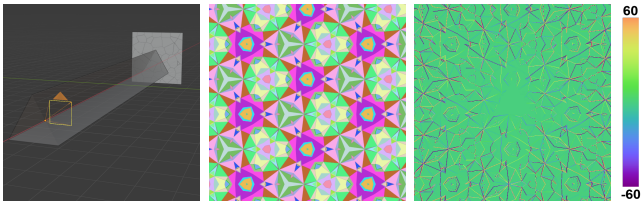


(a) Rendering (b) Derivative 1 (c) Derivative 2

Fig. 11. Comparisons of shading from point lights using the *Dining Room* scene. (a) Split image rendered with (left) our method, and (right) path tracing. (b) Split derivative image w.r.t. the y-coordinate of one light rendered with (left) our method (AD), and (right) path tracing FD. (c) Split derivative image w.r.t. the y-coordinate of the other light. Our derivative computation is roughly two orders of magnitude faster than the FD reference that uses path tracing.

Table 2. Performance breakdown of the different stages of regular rendering (without AD) by our method.

Scene	Total abs. time	Primary vis. vect.	Shadow vis. vect.	BVH traversal	Vertex processing	Shading	Other
Living Room	24.2 sec	12.6%	72.1%	7.4%	4.5%	1.5%	1.9%
Dining Room	44.6 sec	5.3%	76.3%	9.1%	7.4%	0.7%	1.2%
Pufferball	8.37 min	0.1%	87.0%	2.7%	8.7%	0.2%	1.3%



(a) Scene config. (b) Pixel color (c) Derivative

Fig. 12. Rendering of a virtual kaleidoscope that consists of multiple (up to 8) bounces of specular reflection. (a) The camera is placed within a prism of mirrors, facing the emitting pattern. (b) Image rendered by our method. (c) Accurate derivatives of the pixel values w.r.t. mirror rotation.

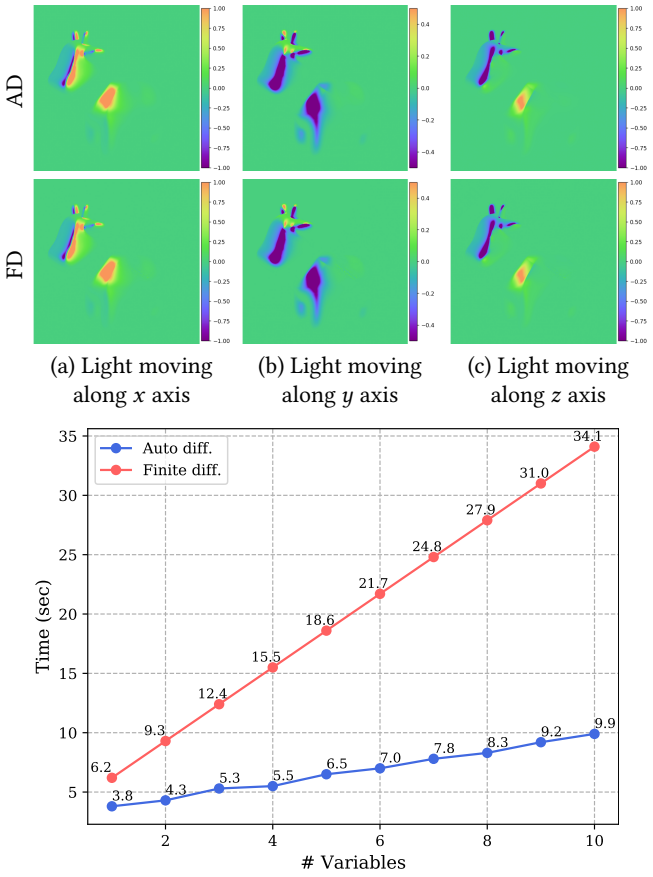
orders of magnitude more time to compute, while still suffering from Monte Carlo noise and numerical cancellation error.

Pure specular reflection. We demonstrate the ability to compute pure specular reflection (i.e. Dirac delta BRDFs) using a virtual kaleidoscope scene in Fig. 12. The camera is placed inside a solid prism of mirror at one end and a textured area light source with colorful patterns is placed at the other end. The image is rendered with up to 8 bounces, and most of the pixel colors are computed by specular reflection. The rendering is perfectly anti-aliased thanks to our analytic shading formula. Our method also allows accurate derivative computation of light transport with pure specular reflection, which is not naturally supported by the existing differentiable rendering techniques [Li et al. 2018; Zhang et al. 2020]. In Fig. 12 (c), we show the derivative image with respect to the rotation of the mirror, which effectively captures the edges of the reflected color patterns from the textured area light.

Analytic Derivatives. To further demonstrate the correctness and efficiency of our vectorization in terms of differentiable rendering, we introduce two different ways of applying our method using Automatic Differentiation (AD) and Finite Differences (FD), respectively. We compare with path traced references using FD, as well as the state-of-the-art techniques by Li et al. [2018] and Zhang et al. [2020] (the unidirectional estimator).

In Fig. 10, we use AD to compute the derivative images. We show the *Tree* scene as an example of differentiating the rotation of objects, and further demonstrate the *Pufferball* scene with object translation, and the *Cow* scene with changing roughness of the BSDF. At equal time, all previous (and concurrent) differentiable rendering methods still have noise in their gradients. Also note that, it is theoretically impossible to compare the running times of other methods at equal quality to ours, since any path traced results, including the "converged" references, will still be noisy. To deal with this, we let other methods run for a reasonable amount of time – an order of magnitude longer than ours – to show that they still generate visible noise.

It is convenient to directly use our method to get the entire gradients with all derivatives at once with AD, since our method is always noise-free. It is also straightforward to subtract two different rendering results of ours to get noise-free derivatives using FD. Fig. 13 compares our AD and FD approaches. We can see that they are almost identical. With this observation, it is natural to ask when we should apply AD and why we cannot always use FD, since it is even simpler to implement. We have plotted two different curves at the bottom of Fig. 13, corresponding to the running time of AD and FD w.r.t. the number of variables to be differentiated. As we can see, finite differences can give us only one entry of the gradient at a time, so FD scales linearly with the number of differentiable variables and the time increases faster. In contrast, AD does have a small overhead when there is only one variable (canceled out by the fact that FD needs to render twice), but is more efficient in computing



(a) Light moving along x axis (b) Light moving along y axis (c) Light moving along z axis

Fig. 13. Comparison of derivative computations of the *Cow* scene by our analytic AD and FD methods. Top row is AD, and bottom row is FD. Below, we show a graph of running time versus number of variables for AD and FD. As expected, FD time increases faster and linearly with the number of variables, since new function evaluations are required for each variable.

the full gradient as the number of variables increases. Like FD, time still increases linearly with the number of variables but the slope is much smaller. It is also worthwhile noting that finite differences will always introduce some bias with finite steps.

Time and space complexity. In general, it is an extremely difficult task to analyze the complexity of the hidden surface removal problem. The upper bound of the time complexity is proven to be 3-SUM hard [Gajentaan and Overmars 1995], which indicates that the worst case performance can be $O(n^2)$, where n is the number of vertices in the scene.

However, we should emphasize the difference between the worst case and the common cases in practice. The $O(n^2)$ worst case only happens if every pair of triangles intersect, which is far from a practical setup in Computer Graphics. We should consider the output-sensitivity of our algorithm by taking the actual number of intersections into account. In a realistic case, given a triangle to be inserted, our VBVH allows rapid pruning of disjoint nodes in the tree, and fast location of a small set of overlapping regions, whose size is proportional to the area of the triangle being inserted. In Fig. 14, we run our vectorization using a beam covering the entire *Stanford*

Table 3. Summary of inverse rendering examples.

Scene	#Params	Source of params	Time per iter.
Tree	1	Transform	8.4 sec
Kaleidoscope	1	Transform	3.6 sec
Dragon	4	Transform+BSDF	11.2 sec
Gears	15	Transform	10.1 sec
Displacement	162	Vertex Position	56.0 sec

dragon model starting from 800K triangles and with subdivision or simplification to obtain different triangle counts. We plot the performance of our vectorization w.r.t. the number of triangles. We see that the complexity is not only far from the theoretical quadratic bound, but also more efficient than the usually expected $O(n \log n)$ curve. Instead, it is almost linear $O(n)$, in fact bounded above by the red dashed linear curve shown in the plot. In practice, many triangles are completely blocked by the previously inserted triangles in the vectorization process, thus are rejected at an early stage. This leads to approximately linear performance, and enables us to achieve superior performance on complex models with millions of triangles.

For space complexity, the worst case of our vectorization is again $O(n^2)$. However, that also only happens in pathological cases where the next inserted triangle always falls in the newly split regions from the last step, forming a highly unbalanced tree. For a relatively balanced VBVH, which is the much more likely case in practice, the storage is dominated by the leaf nodes, and the space complexity becomes $O(n+k)$, where k is the number of edge intersections. Our vectorization uses convex polygons instead of triangles as geometric primitives, which reduces the constant factor of the space complexity. Moreover, as mentioned in §2, our vectorization can operate in-place, which further prevents unnecessary storage.

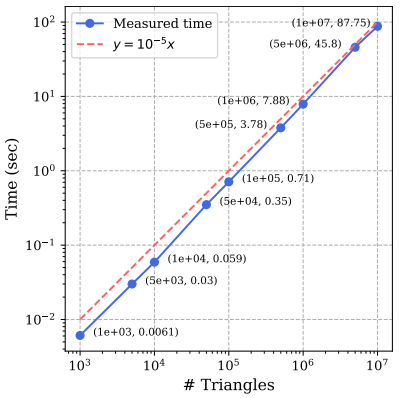


Fig. 14. Running time for geometric vectorization of the *Stanford Dragon* scene with varying numbers of triangles. The complexity is essentially linear.

The input order of triangles intersecting a beam can be arbitrary. Similar to ray tracing, it is possible that a ray will intersect all the triangles from back to front, which will cause a performance degradation. Designing better scene BVHs and using randomized beam intersecting algorithms will alleviate this problem, but this is orthogonal to our vectorization. Note that for a fixed input order of triangles, our method will always result in a unique VBVH representation. For different input orders, our vectorizations can be different, but will always be handled robustly, correctly and accurately.

Inverse Rendering. The goal of inverse rendering is to optimize scene parameters based on one or more rendered images of the scene. Given a set of target images and a set of initial parameters,

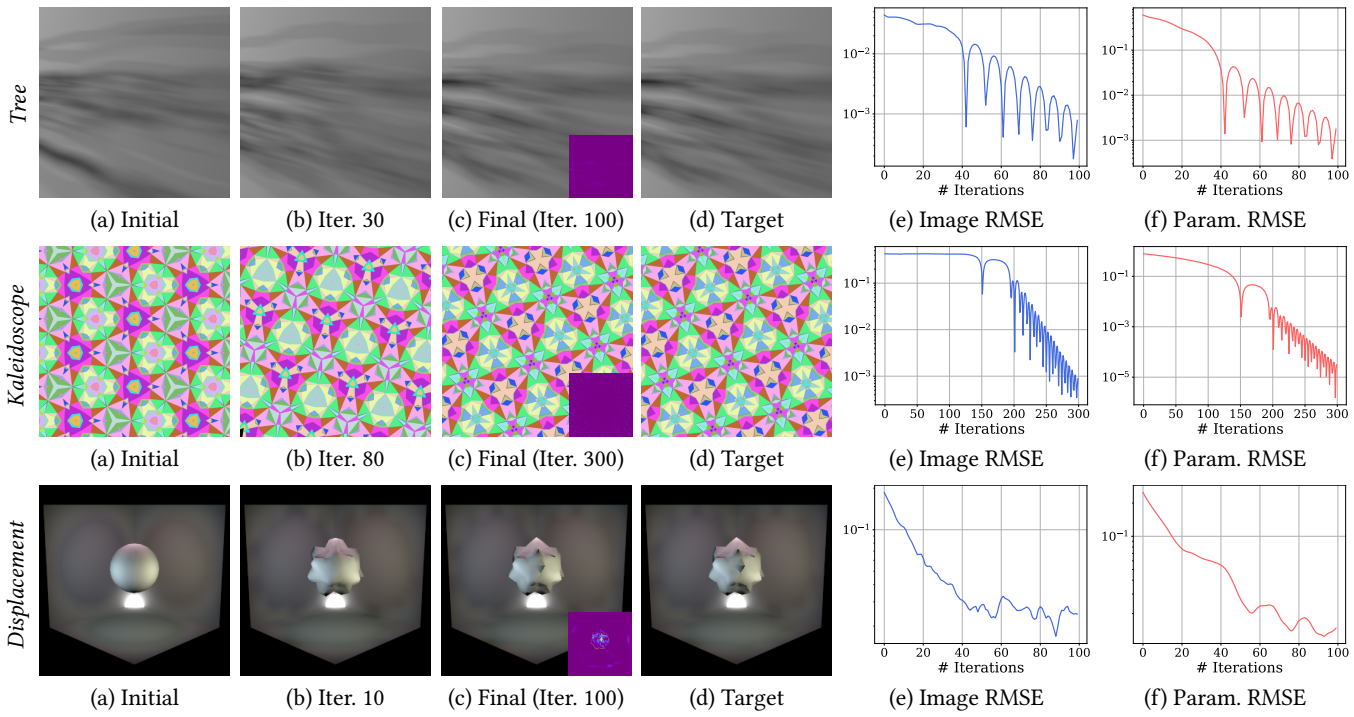


Fig. 15. Inverse rendering examples using our analytic gradients. In the *tree* example, we optimize for the object's rotation angle around the y axis. In the *kaleidoscope* example, we optimize for the mirror's rotation angle around the z axis. In the *displacement* example, we optimize for per-vertex displacement along normal direction. Please refer to the supplementary video for the full sequences. The bottom-right insets in column (c) show (color-mapped) 10× absolute difference images between our final optimization results and the target images. The graphs (e) and (f) show the image and parameter RMSE as functions of the number of iterations.

the inverse rendering process automatically optimizes the parameters such that the final rendered images match the target images. Inverse rendering can be implemented by differentiable rendering techniques that produce gradients, which are necessary for gradient-based optimization. With the ability to produce noise-free gradients, our method is not only compatible with first-order optimization methods, but also enables second-order optimization methods for the first time. In Fig. 15, we show the image sequences of multiple inverse rendering examples, as well as the error plots. In the accompanying video, the entire sequences will be played in full. At each iteration, we use the root mean square error (RMSE) to measure the difference of the currently rendered image and parameters, respectively, against the target image (image RMSE) and the target parameters used to generate the image (parameter RMSE). Please refer to Table 3 for a summary of statistics.

In the *tree* example, we optimize the rotation angle of the tree around y axis. The tree has very complex geometric structures and casts shadows with high-frequency details. We are able to converge to the target angle even with a single view that only captures the shadows. The *kaleidoscope* example attempts to match the target pattern that primarily consists of pure specular reflection by rotating the mirror. The *displacement* example demonstrates the ability of our method to handle many parameters. We displace each of the 162 vertices of the sphere along its normal direction to match the desired shape. All examples use a single view for optimization except the *displacement* example, which uses 8 distinct views.

In order to compare the convergence rates using different methods, we provide two more examples in Fig. 16 and plot their error curves in Fig. 17. We evaluate the effect of pairing different optimizers with different differentiable rendering techniques. For optimizers, Adam [Kingma and Ba 2015] and L-BFGS [Liu and Nocedal 1989] are used for representing first-order optimizers and second-order optimizers, respectively. For differentiable rendering techniques, we compare our method to the state-of-the-art path-space differentiable rendering (PSDR) [Zhang et al. 2020]. In these examples, both differentiable rendering techniques perform each optimization iteration in equal time. Each optimizer is configured with the same set of hyperparameters (e.g. learning rate for Adam, and maximal line search steps for L-BFGS), regardless of the paired differentiable rendering technique.

In the *dragon* example, we jointly optimize the translation (x, y, z) of the light source, as well as the roughness of the model. We observe similar performance between methods that use Adam. However, our method achieves much faster convergence speed when paired with L-BFGS. On the other hand, we observe inconsistent behavior and failure to converge when pairing PSDR with L-BFGS. This is because quasi-Newton optimizers such as L-BFGS are inherently noise-sensitive. The noisy gradients produced by Monte Carlo sampling corrupt the Hessian estimation, which causes the optimization to be trapped in a non-optimum. Although recently some stochastic quasi-Newton methods were proposed [Byrd et al. 2016; Moritz et al.

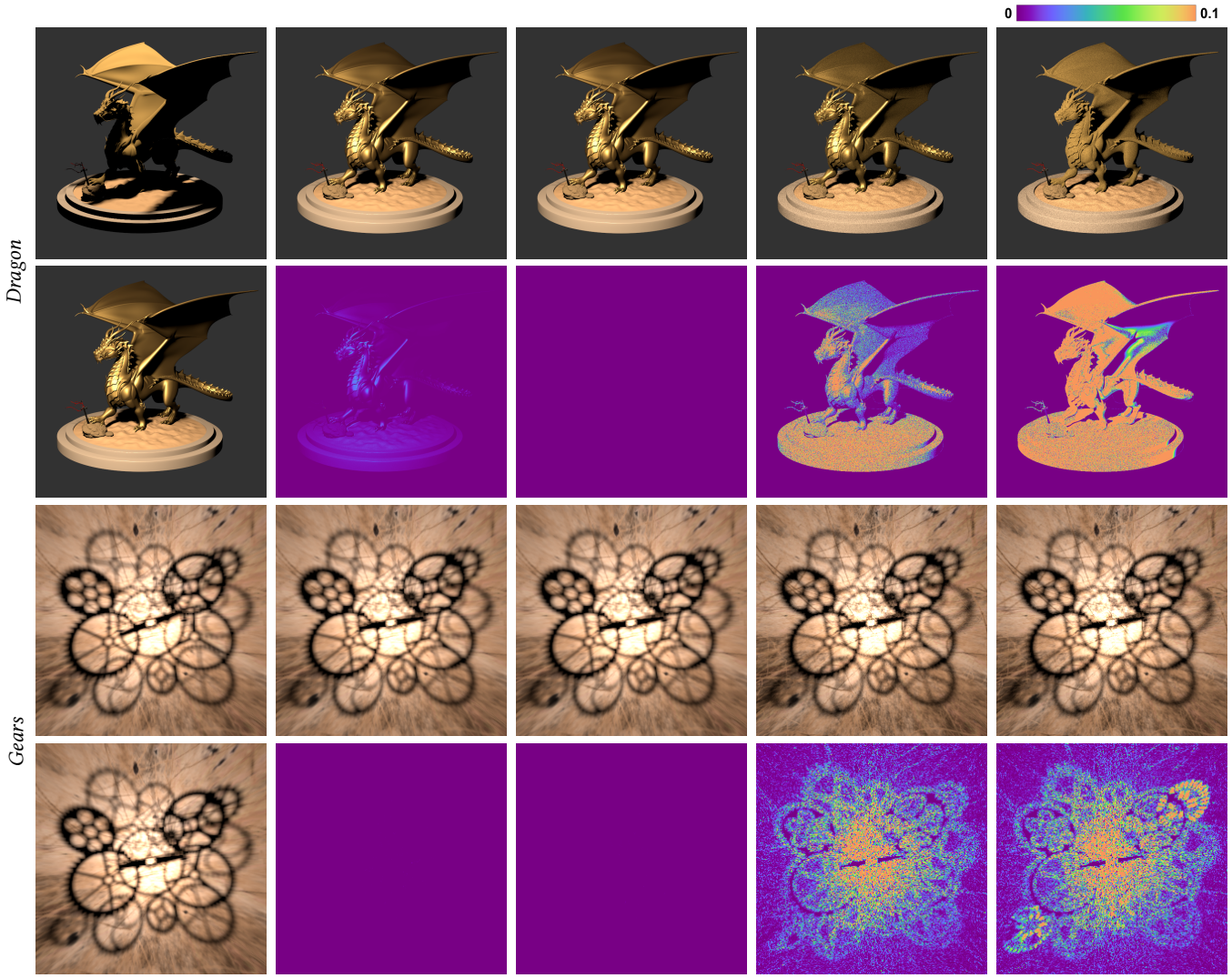


Fig. 16. Comparison of inverse rendering convergence results by different methods. For each example, the initial image (top) and the target image (bottom) are shown in (a). In columns (b) to (e), we show the optimized results by each method (top), as well as the (color-mapped) 10× absolute difference images between the optimized results and the target images (bottom). Our method does not suffer from Monte Carlo noise and achieves lower error. Please refer to the supplementary video for the full optimization sequences, esp. the subtle rotations of the gears.

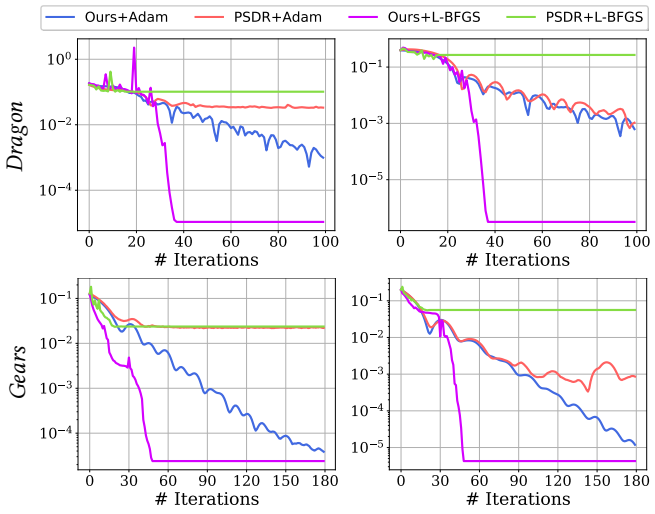
2016], the area still remains active, and we leave a comprehensive exploration in this direction as future work.

In the *gears* example, we optimize the rotation of all gears based on the shadow cast on the background. While both differentiable rendering techniques converge with Adam, PSDR struggles to match the fine-grained shadow silhouette exactly due to Monte Carlo noise. Our method does not suffer from this problem, and achieves lower error. When switching to L-BFGS, the convergence of PSDR, again, is not guaranteed. In contrast, our method leads to consistent and faster convergence rate than that of the first-order alternatives.

7 DISCUSSION AND ANALYSIS

The hidden surface removal problem is one of the most fundamental problems in computer graphics. A wide range of existing works have attempted to solve it analytically. In other words, they are *object-space* methods as classified by Sutherland et al. [1974]. We group these methods into several categories and discuss the relationship between our method and them.

Theoretical algorithms. The hidden surface removal problem is of theoretical interest to researchers because a naïve solution runs in quadratic time which is unacceptable. Various exact algorithms have been proposed to demonstrate certain algorithmic advantanges, such as lower complexity bound, or output-sensitivity. The standard line-sweeping technique can be used to obtain efficient algorithms



(a) Image RMSE (b) Param. RMSE

Fig. 17. Image and parameter RMSE as functions of the number of iterations achieved by different methods. In both examples, ours+L-BFGS leads to faster convergence rate, while PSDR+L-BFGS fails to converge.

for hidden line and hidden surface removal [Nurmi 1985; Katz et al. 1992; McKenna 1987]. Several exact algorithms [Mulmuley 1989, 1994; Goodrich 1992] rely on the data structure called trapezoidal map [de Berg et al. 1997], which maintains a planar subdivision consisting of trapezoidal regions, and allows incremental insertion of line segments. Sharir and Overmars [1992] derived an output-sensitive algorithm by merging visibility maps of triangles.

However, these algorithms are categorized by us as "theoretical" because the discussions are usually limited to theoretical analysis. No implementations or only toy implementations with finite-precision number types have been reported to our knowledge. They are challenging, if possible at all, to implement in practice due to three reasons:

- They assume the *real RAM* computational model, which can compute with real numbers in infinite precision in constant time [Shamos 1978]. This is never possible in reality even with the use of *Exact Geometry Computation* (not constant time).
- They either fail or require significant overhaul in the presence of degeneracies, which is unacceptable in most computer graphics application.
- Some of the algorithms are based on too intricate data structures. Even if they achieve good theoretical complexity, the high (constant) overhead renders them useless in practice.

In constrast, we demonstrate the practical efficiency and robustness of our method in numerous applications.

Weiler-Atherton algorithm. Weiler and Atherton [1977] propose a hidden surface removal algorithm by recursively subdivding the image space into polygonal windows. Despite its conceptual similarity to our method, we make three distinctions between them:

- Weiler-Atherton algorithm requires a fairly complex polygon intersection algorithm that must be able to handle concave polygons with holes. Our method by-design only needs to process convex polygons, which allows us to use a much more efficient intersection algorithm.
- Our strategy of convex partition is necessary to the construction of VBVH because in this way when a leaf node is split, it produces no more than 4 new nodes and all of them are also convex. Weiler-Atherton algorithm produces polygons with holes, whose bounding volumes are meaningless. An arbitrarily complex splitting strategy must be applied afterwards.
- There is no discussion about numerical robustness of Weiler-Atherton algorithm. Given its complex nature, it seems difficult that an implementation with finite-precision number types can perform robustly when the input is complex, especially when the intersection algorithm is used in recursive subdivision: an input polygon of an intermediate step can very well be degenerate or invalid (e.g. self-intersecting or flipped winding order) due to numerical error in previous steps. In constrast, we show how our method can be implemented robustly in §4.3.

Occlusion tree. Another similar method is the occlusion tree proposed by Bittner et al. [1998; 2002a]. The occluison tree is a BSP tree that parititions a viewport into a set of visible polygons that are stored at the leaf nodes. The occlusion tree allows a similar incremental insertion scheme where a new polygon is recursively tested against the tree and split accordingly. The key difference between the occlusion tree and our VBVH lies in the methods to partition space. The occlusion tree partitions space by half-planes induced by the edges of the polygons, which may lead to a high tree depth and consequently high traversal cost. Meanwhile, the VBVH does not suffer from this issue as it groups polygons by bounding boxes. It remains interesting to compare the culling efficiency by using half-planes versus bounding boxes in the conext of analytic visibility computation. Bittner [2002b] has also extended the occlusion tree to handle region-to-region visibility, which could inspire our future directions.

Scanline rendering. Early scanline rendering methods [Catmull 1978, 1984] also introduce techniques for analytic hidden surface removal. Since they are designed for scanline rendering, they take advantange of the spatial coherence between neighbor pixels (scanlines). However, the visible polygon computation within each pixel usually reduces to brute-force clipping. They are not applicable to many of our applications other than primary visibility, such as the analytic computation of soft shadow and its derivative. On the other hand, it is possible to apply their ideas on top of our method to potentially speed up the computation of primary visibility.

Potentially Visible Set. Many methods focus on pre-computing the *Potentially Visible Sets* (PVSs) of the scene, which describe for each view cell the set of objects that may be visible [Airey et al. 1990]. The PVSs are then used to accelerate visibility culling or occlusion culling at rendering. Based on the quality of the PVSs, the methods can be classified as conservative [Hudson et al. 1997; Coorg and Teller 1997; Bittner et al. 1998; Chandak et al. 2009], aggresive [Gotsman et al. 1999; Nirenstein and Blake 2004], or exact [Nirenstein et al. 2002; Haumont et al. 2005]. Our method is an *online* method that produces exact visibility, without the need of preprocessing. It is also possible to incorporate their ideas to accelerate

the rendering of complex scenes, and solve the region-to-region visibility problem.

Approximate visibility map. One natural idea to improve the performance of analytic visibility computation is to introduce some types of approximation. Stewart and Karkanis [1998] describe a method to compute an approximate visibility map by first constructing a coarse visibility map from rasterization, and then refining the coarse map to obtain a better approximation. Erickson [2000] proposes a hybrid image-space/object-space hidden surface removal method that constructs a sampled visibility map based on trapezoid decomposition. One serious disadvantage of those approximation methods in our context is that their result is no longer differentiable with respect to scene parameters. It could be interesting to explore approximations that maintain differentiability.

GPU analytic visibility. Despite the prevalence of rasterization and depth buffering, there are a few attempts to compute analytic visibility on the GPU [Auzinger et al. 2013a,b]. Auzinger et al. [2013b] propose an algorithm to compute edge intersections and hidden surface removal in parallel. Their method is able to achieve interactive performance for relatively simple scenes. While the method exploits the massive parallel architecture of the GPU, it is also brute-force in the sense that it is unable to take advantage of more advanced acceleration structures. Nevertheless, we are eager to explore possible ways to adapt our method to the GPU.

8 LIMITATIONS

Our method is now focused on direct illumination, but can be combined with Monte Carlo techniques to handle more general light transport phenomena such as global illumination. As a proof of concept, we demonstrate how to replace the next-event estimation (NEE) in a path tracer with our vectorization in Appendix B. In the future, we are enthusiastic to extend our hybrid method to handle general high-dimensional region-to-region integrals in light transport to enable efficient global illumination.

As mentioned in §4.1, our geometry vectorization assumes no penetrating triangles. If they do exist, a preprocessing step is required to split them. We note that all existing differentiable rendering techniques implicitly make the same assumption because otherwise penetrating triangles introduce new boundary edges that need to be explicitly handled in the boundary integral calculation to correctly differentiate visibility [Li et al. 2018; Zhang et al. 2020]. Theoretically, it is possible to extend our geometry vectorization to handle penetrating triangles, for example, by computing the intersection between a 3D triangle and its overlapping 3D visible polygons instead of their xy projections. Accordingly, each VBVH node can store an extra depth slab for quick rejection in the z direction.

Our analytic shading from area lights relies on the approximation using LTCs [Heitz et al. 2016]. LTCs have been shown to be expressive enough to closely fit a wide range of BRDFs including anisotropic ones. Although not demonstrated in our applications, a prefiltering approach is also readily available to support shading with textured area lights. Therefore, we conclude that our analytic shading is general for practical usage. While our method can accurately integrate visibility functions and BRDFs, analytic integration of arbitrary integrands of other types, such as pixel filters, remains an interesting future directon.

Our method computes specular reflection by recursively spawning reflected beams for specular faces. This could be impractical for finely tessellated meshes as the number of beams may increase exponentially as the tracing proceeds, similar to the original beam tracing [Heckbert and Hanrahan 1984]. One possible acceleration is to incorporate the depth-first traversal and the geometry level-of-detail control introduced in [Liu et al. 2011].

9 CONCLUSION

We have presented a vectorization method for fast, analytic, and differentiable visibility. For the first time, we have enabled beam tracing to accurately handle the complex geometries robustly within its viewport, thanks to our VBVH structure. With our accurate beam tracing scheme, we can have analytic solutions to the 2D point-to-region visibility problem, leading to a practical hybrid algorithm to solve the region-to-region problem in 4D. We demonstrate the robustness and correctness of our method, and provide efficient solutions to many rendering applications.

Our beam tracing immediately gives us noise-free results. But more importantly, it allows us to elegantly compute the visibility differentiation problem w.r.t. any parameters that were previously considered difficult, simply using automatic differentiation without any modification of our pipeline. Compared to existing differentiable rendering methods, our method is able to quickly generate noise-free gradients, which can be easily used in inverse rendering and enable second-order optimization techniques for the first time.

Although our method currently computes exact visibility, it can be valuable to study approximate or progressive techniques to increase efficiency while maintaining perceptually satisfactory quality. Applying frequency analysis in beam tracing could also be explored, since the beam's coverage is essentially an accurate version of a ray differential. Other improvements, such as hardware implementation and automatic balancing of our VBVH with sub-tree rotations, will further improve the practicality of our method.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was funded in part by an NVIDIA Fellowship, the Ronald L. Graham Chair and the UC San Diego Center for Visual Computing.

REFERENCES

John M Airey, John H Rohlf, and Frederick P Brooks Jr. 1990. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH computer graphics* 24, 2 (1990), 41–50.

John Amanatides. 1984. Ray tracing with cones. In *ACM SIGGRAPH Computer Graphics*, Vol. 18. 129–135.

Oana Livia Apostu. 2012. *Analytic visibility in Plücker space: From theory to practical applications.* Ph.D. Dissertation. Université de Limoges.

Thomas Auzinger, Michael Guthe, and Stefan Jeschke. 2012. Analytic Anti-Aliasing of Linear Functions on Polytopes. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 335–344.

Thomas Auzinger, Przemyslaw Musialski, Reinhold Preiner, and Michael Wimmer. 2013a. Non-Sampled Anti-Aliasing. In *Vision, Modeling & Visualization*. The Eurographics Association.

Thomas Auzinger, Michael Wimmer, and Stefan Jescke. 2013b. Analytic Visibility on the GPU. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 409–418.

Sai Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased Warped-Area Sampling for Differentiable Rendering. *ACM Trans. Graph.* 39, 6 (2020), 245:1–245:18.

D. R. Baum, H. E. Rushmeier, and J. M. Winget. 1989. Improving Radiosity Solutions through the Use of Analytically Determined Form-Factors. *SIGGRAPH 89*, 325–334.

Jiří Bittner. 2002a. Efficient construction of visibility maps using approximate occlusion sweep. In *Proceedings of the 18th Spring Conference on Computer Graphics*. 167–175.

Jirı Bittner. 2002b. *Hierarchical techniques for visibility computations*. Ph.D. Dissertation. Czech Technical University.

Jiri Bittner, Vlastimil Havran, and Pavel Slavik. 1998. Hierarchical visibility culling with occlusion trees. In *Proceedings. Computer Graphics International (Cat. No. 98EX149)*. IEEE, 207–219.

Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. 1995. Exact geometric computation in LEDA. In *Proceedings of the 11th annual symposium on computational geometry*. 418–419.

Richard H Byrd, Samantha L Hansen, Jorge Nocedal, and Yoram Singer. 2016. A stochastic quasi-Newton method for large-scale optimization. *SIAM Journal on Optimization* 26, 2 (2016), 1008–1031.

Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv preprint arXiv:1509.07164* (2015).

Edwin Catmull. 1978. A hidden-surface algorithm with anti-aliasing. *ACM SIGGRAPH Computer Graphics* 12, 3 (1978), 6–11.

Edwin Catmull. 1984. An analytic visible surface algorithm for independent pixel processing. *ACM SIGGRAPH Computer Graphics* 18, 3 (1984), 109–115.

Anish Chandak, Lakulish Antani, Micah Taylor, and Dinesh Manocha. 2009. FastV: From-point Visibility Culling on Complex Models. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1237–1246.

Satyan Coorg and Seth Teller. 1997. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 symposium on Interactive 3D graphics*. 83–ff.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, Vol. 30. 1921–1930.

Cyril Crassin, Chris Wyman, Morgan McGuire, and Aaron Lefohn. 2018. Correlation-aware semi-analytic visibility for antialiased rendering. In *Proceedings of the Conference on High-Performance Graphics*. 1–4.

Mark de Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. Computational geometry. In *Computational geometry*. 1–17.

Frédo Durand. 1999. *3D Visibility: analytical study and applications*. Ph.D. Dissertation.

Bernardt Duvenhage, Kadi Bouatouch, and Derrick Kourie. 2010. Exploring the use of glossy light volumes for interactive global illumination. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. 139–148.

Bernardt Duvenhage, Kadi Bouatouch, and Derrick G Kourie. 2014. Light beam tracing for multi-bounce specular and glossy transport paths. In *Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology*. 199–208.

Jeff Erickson. 2000. Finite-resolution hidden surface removal. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms*.

Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. 2000. *Scalable vector graphics (SVG) 1.0 specification*.

Steven Fortune. 1999. Topological beam tracing. In *Proceedings of the fifteenth annual symposium on Computational geometry*. 59–68.

Anka Gajentaan and Mark H Overmars. 1995. On a class of O (n2) problems in computational geometry. *Computational geometry* 5, 3 (1995), 165–185.

Michael T Goodrich. 1992. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graphical Models and Image Processing* 54, 1 (1992), 1–12.

Craig Gotsman, Oded Sudarsky, and Jeffrey A Fayman. 1999. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics* 23, 5 (1999), 645–654.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Denis Haumont, Otso Makinen, and Shaun Nirenstein. 2005. A low dimensional framework for exact polygon-to-polygon occlusion queries. (2005).

Paul S Heckbert and Pat Hanrahan. 1984. Beam tracing polygonal objects. *SIGGRAPH 84*, 119–127.

Eric Heitz, Jonathan Dupuy, Cyril Crassin, and Carsten Dachsbacher. 2015. The SGGX microflake distribution. *ACM Transactions on Graphics* 34, 4 (2015), 1–11.

Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. 2016. Real-time polygonal-light shading with linearly transformed cosines. *ACM Transactions on Graphics* 35, 4 (2016), 41.

Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming Lin, Kenneth Hoff, and Hansong Zhang. 1997. Accelerated occlusion culling using shadow frusta. In *Proceedings of the thirteenth annual symposium on Computational geometry*. 1–10.

Homan Igehy. 1999. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 179–186.

Wenzel Jakob. 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki.

André Jalobeanu, Frank O Kuehnel, and John C Stutz. 2004. Modeling images of natural 3d surfaces: Overview and potential applications. In *2004 Conference on Computer Vision and Pattern Recognition Workshop*. IEEE, 188–188.

Matthew J Katz, Mark H Overmars, and Micha Sharir. 1992. Efficient hidden surface removal for objects with small union size. *Computational Geometry* 2, 4 (1992), 223–234.

Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. 2008. Classroom examples of robustness problems in geometric computations. *Computational Geometry* 40, 1 (2008), 61–78.

Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

Dale Kirkland, Bill Armstrong, Michael Gold, Jon Leech, and Paula Womack. 2002. ARB Multisample. *OpenGL Extension Registry* (2002), 83–94.

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Transactions on Graphics* 39, 6 (2020).

Chen Li, Sylvain Pion, and Chee-Keng Yap. 2005. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming* 64, 1 (2005), 85–111.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Transactions on Graphics* 37, 6 (2018), 222:1–222:11.

Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. 2020. Differentiable Vector Graphics Rasterization for Editing and Learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 193:1–193:15.

Baoquan Liu, Li-Yi Wei, Xu Yang, Chongyang Ma, Ying-Qing Xu, Baining Guo, and Enhua Wu. 2011. Non-Linear Beam Tracing on a GPU. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 2156–2169.

Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1-3 (1989), 503–528.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2019).

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics* 38, 6 (2019), 1–14.

Guillaume Loubet and Fabrice Neyret. 2017. Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets. In *Computer Graphics Forum*, Vol. 36. 431–442.

Josiah Manson and Scott Schaefer. 2013. Analytic rasterization of curves with polynomial filters. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 499–507.

Michael McKenna. 1987. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics (TOG)* 6, 1 (1987), 19–28.

Kurt Mehlhorn and Stefan Naher. 1995. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM* 38, 1 (1995), 96–103.

Philipp Moritz, Robert Nishihara, and Michael Jordan. 2016. A linearly-convergent stochastic L-BFGS algorithm. In *Artificial Intelligence and Statistics*. 249–258.

Ketan Mulmuley. 1989. An Efficient Algorithm for Hidden Surface Removal. *SIGGRAPH 89*, 379–388.

Ketan Mulmuley. 1994. An efficient algorithm for hidden surface removal, II. *J. Comput. System Sci.* 49, 3 (1994), 427–453.

Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. 2020. Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering. *ACM Trans. Graph.* 39, 4, Article 146 (2020), 15 pages.

Shaun Nirenstein and Edwin Blake. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. (2004).

Shaun Nirenstein, Edwin Blake, and James Gain. 2002. Exact from-region visibility culling. Eurographics.

Derek Nowrouzezahrai, Ilya Baran, Kenny Mitchell, and Wojciech Jarosz. 2014. Visibility Silhouettes for Semi-Analytic Spherical Integration. *Computer Graphics Forum* 33, 1 (2014), 105–117.

Otto Nurmi. 1985. A fast line-sweep algorithm for hidden line elimination. *BIT Numerical Mathematics* 25, 3 (1985), 466–472.

Joseph O'rourke et al. 1998. *Computational geometry in C*. Cambridge university press.

Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. 2008. Diffusion curves: a vector representation for smooth-shaded images. *ACM Transactions on Graphics* 27, 3 (2008), 1–8.

Ryan Overbeck, Ravi Ramamoorthi, and William R Mark. 2007. A real-time beam tracer with application to exact soft shadows. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*. 85–98.

Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. 2020. Accelerating 3D Deep Learning with PyTorch3D. *arXiv:2007.08501* (2020).

Michael Ian Shamos. 1978. Computational geometry. *Ph. D. thesis, Yale University* (1978).

Micha Sharir and Mark H Overmars. 1992. A simple output-sensitive algorithm for hidden surface removal. *ACM Transactions on Graphics (TOG)* 11, 1 (1992), 1–11.

Jonathan Richard Shewchuk. 2013. Lecture Notes on Geometric Robustness. (2013).

A James Stewart and Tasso Karkanis. 1998. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. In *Eurographics Workshop on Rendering Techniques*. Springer, 57–68.

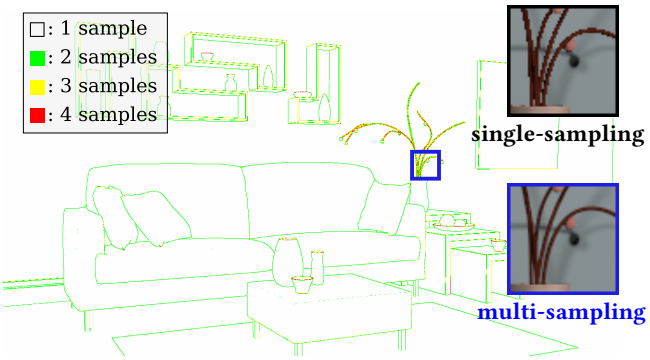


Fig. 18. Visualization of our multi-sampling scheme applied to the *Dining Room* scene, with a maximum sample count of 4. We compare the images rendered using single-sampling and multi-sampling on the right. Our multi-sampling scheme results in better anti-aliasing quality.

Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. 1974. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)* 6, 1 (1974), 1–55.
The CGAL Project. 2020. *CGAL User and Reference Manual* (5.0.2 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0.2/Manual/packages.html
Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. 2017. *Handbook of discrete and computational geometry*. Chapman and Hall/CRC.
Yuxiang Wang, Chris Wyman, Yong He, and Pradeep Sen. 2015. Decoupled coverage anti-aliasing. In *Proceedings of the 7th Conference on High-Performance Graphics*. 33–42.
Mark Watt. 1990. Light-water interaction using backward beam tracing. *ACM SIGGRAPH Computer Graphics* 24, 4 (1990), 377–385.
Kevin Weiler and Peter Atherton. 1977. Hidden surface removal using polygon area sorting. *ACM SIGGRAPH computer graphics* 11, 2 (1977), 214–222.
Lifan Wu, Shuang Zhao, Ling-Qi Yan, and Ravi Ramamoorthi. 2019. Accurate appearance preserving prefiltering for rendering displacement-mapped surfaces. *ACM Transactions on Graphics* 38, 4 (2019), 1–14.
Chris Wyman and Morgan McGuire. 2017. Hashed alpha testing. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 1–9.
Lei Yang, Shiqiu Liu, and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. *STAR* 39, 2 (2020).
Chee-Keng Yap. 1997. Towards exact geometric computation. *Computational Geometry* 7, 1-2 (1997), 3–23.
Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-Space Differentiable Rendering. *ACM Transactions on Graphics* 39, 6 (2020), 143:1–143:19.
Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shuang Zhao. 2019. A Differential Theory of Radiative Transfer. *ACM Transactions on Graphics* 38, 6 (2019), 227:1–227:16.
Shuang Zhao, Frédo Durand, and Changxi Zheng. 2017. Inverse diffusion curves using shape optimization. *IEEE transactions on visualization and computer graphics* 24, 7 (2017), 2153–2166.
Shuang Zhao, Lifan Wu, Frédo Durand, and Ravi Ramamoorthi. 2016. Downsampling Scattering Parameters for Rendering Anisotropic Media. *ACM Transactions on Graphics* 35, 6 (2016).

A HYBRID APPROACH TO REGION-TO-REGION INTEGRALS

In our hybrid approach to solving 4D region-to-region integrals for direct illumination, we first sample primary rays adaptively within a 2D pixel footprint $\mathcal{P}$, then trace beams to compute the other 2D point-to-region integral analytically. We visualize the sampling rate of each pixel using our adaptive multi-sampling strategy in Fig. 18, and show better image quality compared to tracing one beam from the center of each pixel (i.e., the single-sampling strategy). As we can see, most pixels do not require multi-sampling, thus leading to a small performance overhead. With the rapid development of Temporal Anti-Aliasing (TAA) [Yang et al. 2020], it is arguable that the temporal reuse strategies may further decrease the number of point-to-region integrals needed to approximate the region-to-region transport, ideally down to one per pixel.

Fig. 19. One-bounce global illumination of the *Dining Room* scene by our hybrid path tracer. Our method no longer produces noise-free results.

Alternatively, one can get an unbiased estimate by drawing point samples in the 4D space using Monte Carlo path tracing. However, point sampling yields large variance and produces noisy images. Our method, in contrast, generates noise-free shading and soft shadows from area lights. We have demonstrated that our hybrid approach that exploits geometry vectorization leads to better rendering quality than traditional Monte Carlo path tracing (Figs. 1 and 8). In general, we can extend our hybrid method to solve integrals with higher dimensions for handling global illumination, as demonstrated in Appendix B.

B PRELIMINARY EXPERIMENTS ON ONE-BOUNCE GLOBAL ILLUMINATION

In order to support global illumination, we have integrated our geometry vectorization into a standard unidirectional path tracer to replace the next-event estimation (NEE). Effectively, the hybrid path tracer computes the n-D light transport integral by sampling paths in the first $(n-2)$-D space and computing the last 2D point-to-region integral analytically. In Fig. 19, we show the one-bounce global illumination of the *Dining Room* scene. While our hybrid path tracer converges to ground truth with enough samples, in general it is not able to produce noise-free images (see the insets of Fig. 19). This is because our method currently cannot fully handle the high-dimensional path integral. The benefit of analytic direct lighting diminishes as noise is reintroduced by Monte Carlo sampling.

The experimental integration could be improved in several possible ways. First, currently the result of each analytic direct lighting computation is only used by one path sample, which seems to be wasteful given the cost of the computation. It might be worthwhile to amortize the computation cost by caching. Alternatively, our vectorization could be used to accelerate other types of visibility computation. For example, given an incident direction and a glossy surface, we could trace a beam that encloses the reflection lobe, which will provide free visibility for all subsequent reflected rays. We leave those possible improvements for future work.