Physics-Based Differentiable Rendering: A Comprehensive Introduction

SHUANG ZHAO, University of California, Irvine
WENZEL JAKOB, École Polytechnique Fédérale de Lausanne
TZU-MAO LI, MIT CSAIL

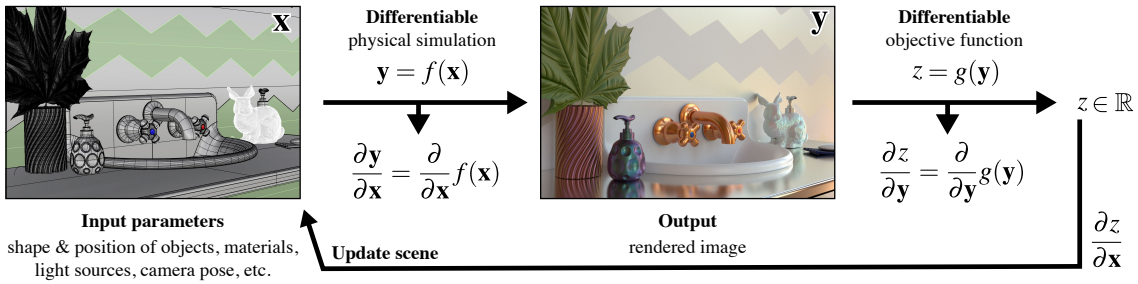
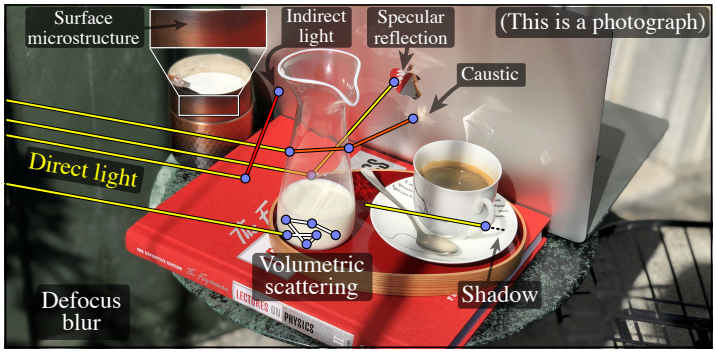


Fig. 1. Rendering a scene using a realistic light transport simulation can be interpreted as evaluating a function $f(\mathbf{x})$, whose input $\mathbf{x}$ encodes the shape and materials of objects. Owing to the complexity of f, an inverse $\mathbf{y} = f^{-1}(\mathbf{x})$ to retrieve scene parameters from an existing image $\mathbf{y}$ is normally not available. Differentiable renderers pursue such an inverse by differentiating f and casting the inversion into a gradient-based minimization process.

ACM Reference Format:
Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. 2020. Physics-Based Differentiable Rendering: A Comprehensive Introduction. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Courses (SIGGRAPH '20 Courses), August 17, 2020.* ACM, New York, NY, USA, 36 pages. https://doi.org/10.1145/3388769.3407454

1 INTRODUCTION

Differentiable rendering is a emerging tool enabling the *inverse analysis of images*—for instance, determining the shape and material of an object from one or multiple input photographs. This problem is surprisingly challenging—to understand why, consider the photograph on the right. Each pixel encodes a complex and messy superposition of different physical effects: light can interact with one object (e.g. the red book) and then propagate its color to neighboring objects. Specular surfaces create indirect views of other parts of the scene. Reflective and refractive objects focus light into caustics. Light can also be absent because it is blocked by another object. A milliliter of milk contains more than a billion of microscopic fat globules, and light entering such a volumetric material can interact with many thousands of them



SIGGRAPH '20 Courses, August 17, 2020, Virtual Event, USA
2020. ACM ISBN 978-1-4503-7972-4/20/08…$15.00
https://doi.org/10.1145/3388769.3407454

before exiting the glass some distance away, giving milk its typical "glow". The surface seen through a pixel may not even be in focus, in which case its interpretation is even less clear. Every position is effectively *coupled* to all other positions through the physics of light, and it is therefore not possible to analyze one object in isolation—we need to understand the properties of all other objects at the same time!

Existing physics-based rendering algorithms generate images by simulating the flow of light through detailed virtual scene descriptions. This process can also be interpreted as evaluating a function $f : \mathcal{X} \to \mathcal{Y}$, whose high-dimensional input encodes the shape and materials of objects (Figure 1), and whose output $\mathbf{y} = f(\mathbf{x})$ is a rendered image that reveals the complex radiative coupling between objects (shadows, interreflection, caustics, etc.). These effects are crucial in the pursuit of (photo-)realism, but they also obscure the individual properties of objects as shown on the previous page. For this reason, an inverse f^{-1} to retrieve parameters $\mathbf{x}$ from an existing image $\mathbf{y}$ is generally not available.

Methods in the area of **physics-based differentiable rendering** seek to provide such an inverse. However, instead of directly evaluating f^{-1}, they target a more general optimization problem that requires the choice of an objective function $z = g(\mathbf{y})$ to quantify the quality of a tentative solution. In the simplest case, we could, e.g., set $g(\mathbf{y}) = \|\mathbf{y} - \mathbf{y}_{\text{ref}}\|^2$ to measure the consistency of a simulation with an empirical observation $\mathbf{y}_{\text{ref}}$. Ordinarily, determining parameters $\mathbf{x}$ that optimize $g(f(\mathbf{x}))$ would simply be infeasible due to the nonlinear nature of the parameter space and its enormous dimension: every texel, voxel, and vertex in a mesh is a free parameter, and detailed scenes can easily have hundreds of millions of them!

The main distinction and benefit of differentiable rendering compared to "ordinary" rendering is the availability of derivatives ($\partial z/\partial \mathbf{x}$), which provide a direction of steepest descent in the high-dimensional parameter space (Figure 1). Using any suitable gradient-based optimization technique, a differentiable renderer is then able to successively improve the parameters with respect to the specified objective. The high level of generality of this approach has made physics-based differentiable rendering a key ingredient for solving challenging inverse-rendering problems in a variety of scientific disciplines. Another benefit of differentiable rendering techniques is that they can be incorporated into probabilistic inference and machine learning pipelines that are trained end-to-end. For instance, differentiable renderers allow "rendering losses" to be computed with complex light transport effects captured. Additionally, they can be used as generative models that synthesize photorealistic images.

Challenges. Compared to its "ordinary" counterpart, physics-based differentiable rendering introduces unique theoretical and practical challenges. For instance, practical problems can involve many (e.g., 10^6–10^{10}) parameters, making simple techniques for differentiation such as finite differences impractical. Standard frameworks for automatic differentiation (e.g., PyTorch or Tensorflow) that are widely used to train neural networks are poorly suited to the prohibitively large and unstructured graph structure underlying the computation of complex light transport effects. Finally, geometric derivatives involve a unique challenge: boundaries of objects introduce troublesome discontinuities during the computation of shadows and interreflections that lead to incorrect gradients if precautions are not taken. Thankfully, recent advances in physics-based differentiable rendering theory have enabled the scalable differentiation of radiometric measurements with respect to arbitrary scene parameters as well as unbiased Monte Carlo estimators. **In this course, we provide an in-depth introduction to general-purpose physics-based differentiable rendering.**

Course outline. This course is comprised of two major components: (i) a *theory*-focused component covering the mathematical foundation of physics-based differentiable rendering; and (ii) an *implementation*-focused component discussing the corresponding Monte Carlo solutions as well as how these estimators can be implemented correctly and efficiently. For physics-based differentiable rendering theory, we cover the following topics.

§2. **Motivation and Preliminaries**: We introduce notation and principles of physically-based rendering. Following this, we motivate differentiable rendering using a toy example of a scene containing two constant-colored triangles. We also introduce the mathematical preliminaries on the differentiation of integrals using Leibniz's rule and Reynolds transport theorem.

§3.1. **Differentiable rendering of surfaces**: We utilize the results from §2 to differentiate the rendering equation (RE) in §3.1.1 and §3.1.2, enabling general-purpose differentiable rendering of surfaces.

§3.2. **Differentiable rendering of participating media**: We discuss how the full radiative transfer equation (RTE) can be differentiated with respect to arbitrary scene parameters.

§3.3. **Radiative backpropagation**: We explain how differentiation can be cast into a light transport problem to enable efficient differentiation of scenes involving millions of parameters.

On the implementation side, we discuss the following aspects:

§5.1. **Monte Carlo solutions**: We present a few Monte Carlo solutions for derivative estimation including differentiable path tracing with edge sampling (§5.1.1), its reparameterized variant (§5.1.2), and differentiable volume path tracing (§5.1.3).

§5.2. **Automatic differentiation**: We discuss how automatic differentiation (autodiff), a key ingredient of many general-purpose differentiable renderers, should be implemented for rendering.

Lastly, we demonstrate in §6 potential applications of physics-based differentiable rendering to several challenging inverse rendering problems.

1.1 Scope

In this course, we will mainly discuss how to correctly and efficiently differentiate rendering operations, with hard surface boundaries in the presence.

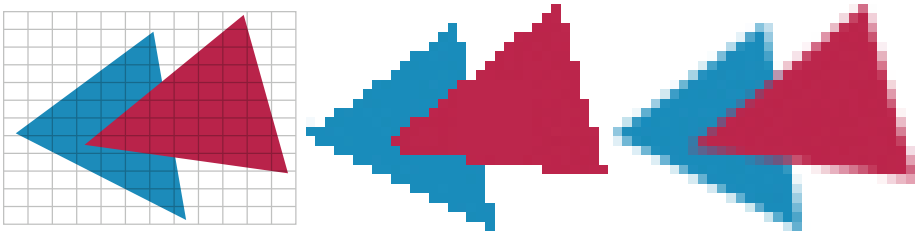
Geometry and representation. We focus our discussions on mesh representations for our scene geometry, as it is currently the standard of 3D photorealistic rendering. Most techniques and theories we present can potentially be generalized to handle other geometric representations that represent surfaces such as implicit functions, polynomials, or boundary representation.

Approximate versus unbiased methods. We mainly focus on how to correctly derive the derivatives and consistent/unbiased estimators of rendering in this course. We briefly discuss approximate methods in Section 2.6, and show that many of them are compatible with our theory. We note that it is still debatable whether approximation is necessary for fast differentiable rendering. From our experience, the main bottleneck of most differentiable rendering systems lie in the irregular memory access in the derivative computation, instead of the visibility computation, regardless of rasterization or ray-tracing. Further research is required to accelerate both approximated and unbiased methods.

Scene initialization, parametrization, topology, and prior. We focus on discussing the gradient computation of the rendering process. A real inverse rendering pipeline is more involved than just the gradient computation, requiring careful initialization and parametrization to avoid convergence to low-quality local minima, handling of discrete topology changes of meshes, and assigning priors to resolve ill-posed aspects of inverse problems. These issues, while being very important research problems, are beyond the scope of this course.

2 MOTIVATION AND MATHEMATICAL PRELIMINARIES

In this section, we will work on a simplified problem with rendering two 2D triangles with constant color (Fig. 2a). The two triangles can occlude each other. In this case our scene parameters are the 6 triangle vertices (12 real



(a) (b) no anti-aliasing (c) with anti-aliasing

Fig. 2. We start with a simplified rendering example to explain our motivation. Given two constant color triangles in 2D (a), we want to generate an image. A naïve approach is to evaluate the color for each pixel at the center (b). However, this approach is prone to *aliasing* in the signal processing sense, which produces artifacts such as jagged edges, fake fine details, and temporal flickering. Most renderers, whether differentiable or not, real-time or offline, rasterized or ray-traced, have to perform anti-aliasing (c). Anti-aliasing is formulated as an **integral** for each pixel, over the support of a low-pass filter. In the case of a uniform box filter, we want to compute the average color for each pixel.

numbers) and the colors of the two triangles (6 real numbers). Given these 18 real numbers as a vector $\boldsymbol{\pi}$, where we denote the vertices parameters as $\boldsymbol{\pi}_v$ and the color parameters as $\boldsymbol{\pi}_c$, we want to generate an image $I(\pi)$ and compute a loss function $L(I(\boldsymbol{\pi}))$ (e.g., comparing the image with a target, or feeding the image to a neural network classifier). Our goal to compute the gradient $\nabla_{\boldsymbol{\pi}} L(I(\boldsymbol{\pi}))$ so that we can minimize the loss using gradient-based optimization.

We will first show that rendering is usually modeled as an integration problem. Then we show that, while the integrands are discontinuous, the integrals are actually differentiable. Unfortunately, naïve combination of integral discretization and automatic differentiation does not compute the correct derivatives that converge in the limit. We discuss the mathematical tools that help us to correctly discretize the derivatives of integrals.

2.1 Rendering as an Integration Problem

Before we talk about the gradient, we need to discuss how the image I is defined. How do we generate an image from these two triangles? We can imagine that the two triangles define an underlying *imaging function* $m(x, y; \boldsymbol{\pi})$ that maps continuous 2D coordinates (x, y) to a RGB color, depending on which triangle the 2D coordinate hits. However an image is a discrete 2D grid. How do we go from the imaging function m to the image I? A naïve approach (Fig. 2b) is to evaluate m at the center of each pixel. This approach is prone to *aliasing*, which causes issues including jagged edges, temporal flickering, Moiré patterns, and breaking up fine details [7].

From the signal processing perspective, we are *sampling* this 2D domain with a discrete image, where the sampling rate is determined by the imaging function. Since the imaging function m is discontinuous, it has energy at all frequencies and is not bandlimited. Therefore, as long as we are evaluating at the center of the pixels, we will suffer from the aliasing problem no matter how large we select the resolution. To resolve the aliasing issue, we need to remove the high frequencies energy from the imaging function m. This is done by convolving the imaging function with a low pass filter (Fig. 2c). For each pixel I_i, we evaluate an *integral* centered around the pixel center (x_i, y_i):

$$I_i = \iint k(x, y) m(x_i + x, y_i + y; \boldsymbol{\pi}) \mathrm{d}x\mathrm{d}y = \iint f(x, y; \boldsymbol{\pi}) \mathrm{d}x\mathrm{d}y, \tag{1}$$

where k is the filtering kernel. For convenience we define the rendering integrand $f(x, y) = k(x, y) m(x_i + x, y_i + y)$.

Intuitively, to remove the artifacts introduced by aliasing, instead of only evaluating the center at each pixel, we evaluate the weighted average color over an area. The selection of the filtering kernel k is an art in its own

right, which involves subjective trade-offs between blurriness and ringing artifacts [29]. From now on we will assume we are using a simple kernel called *box filter*, which assign uniform weights inside the pixel area. The technique we introduce throughout this course works for any filter kernels, even non-differentiable ones.

There are at least two ways to solve the anti-aliasing integral (Eq. (1)), one is analytical and one is numerical. Catmull [4] proposed to solve the integral analytically, by clipping the polygons against each other and partition the integral into multiple regions, and analytically integrate the polygons. Arvo [2] derived an analytical approach to compute the derivatives of diffuse global illumination. The analytical approach is unfriendly to modern hardware architectures as it is hard to parallelize and involves extensive branching. It is also difficult to generalize analytical integral for a different imaging function m and a kernel k – What if we want to use primitives other than triangles such as ellipse or polynomials? What if, inside the polygons, we have color variation, and we don't have analytical solution for the color variation?

Most renderers, whether real-time, offline, physics-based, differentiable or not, need to deal with the aliasing issue. Most of them solve the anti-aliasing integral using numerical solution by evaluating the imaging function at various locations, a process often called *discretization*:

$$I_i \approx \frac{1}{N} \sum_{j=1}^{N} f(x_j, y_j; \boldsymbol{\pi}), \tag{2}$$

The naïve approach of evaluating at pixel center can also be seen as a (poor) approximation to the integral by setting $N = 1$ and $x_0 = y_0 = 0.5$.

We say a discretization is **consistent** if the discretization converges to the integral, i.e., $\lim_{N\to\infty} 1/N \sum_{j=1}^{N} f(x_j, y_j) = I_i$. The choice of the samples x_j, y_j does not need to be stochastic. Nevertheless, if we are randomly sampling x_j, y_j using probabilistic distribution, we say a discretization is **unbiased** if the expectation is the same as the integral, i.e., $\mathbb{E}[f(x_j, y_j)] = I_i$.

It turns out that, as we will show in Section 3, integrals are ubiquitous in rendering [6], and they do not just appear in anti-aliasing. For example, we can model motion blur as an integration over time during the camera shutter is open. We can model defocus blur for non-pinhole cameras as an integration over the aperture area. Given an area light, we compute the radiance that goes from the light source to the camera by integrating over the area of the light source. Kajiya [20] showed that we can model global illumination as an *recursive* integral, by integrating each 3D point in the scene recursively. Therefore, most rendering problems are about 1) How do we model the integrand f inside these multi-dimensional integrals, and 2) How do we evaluate, approximate, precompute, or compress the integral $\int f$ for each pixel. Differentiable rendering instead asks the third question: how do we differentiate these integrals?

For now let us focus on the anti-aliasing problem with the two constant color triangles in Figure 2.

2.2 Computing Gradients by Differentiating the Integrals.

Remember that our goal is to compute the gradient of the image I that contains the two triangles, over some scalar loss function L, that is, $\nabla_{\boldsymbol{\pi}} L\left(I\left(\boldsymbol{\pi}\right)\right)$. Using the chain rule, we know that for each component $\pi \in \mathbb{R}$ in the parameter vector $\boldsymbol{\pi}$

$$\frac{\partial}{\partial \pi} L\left(I\left(\boldsymbol{\pi}\right)\right) = \sum_i \frac{\partial L}{\partial I_i(\boldsymbol{\pi})} \frac{\partial I_i(\boldsymbol{\pi})}{\partial \pi}. \tag{3}$$

To be more concrete, if our loss function is the sum of pixel-wise squared difference with another target image $\hat{I}$, that is,

$$L = \left(I(\boldsymbol{\pi}) - \hat{I}(\boldsymbol{\pi})\right)^2, \tag{4}$$

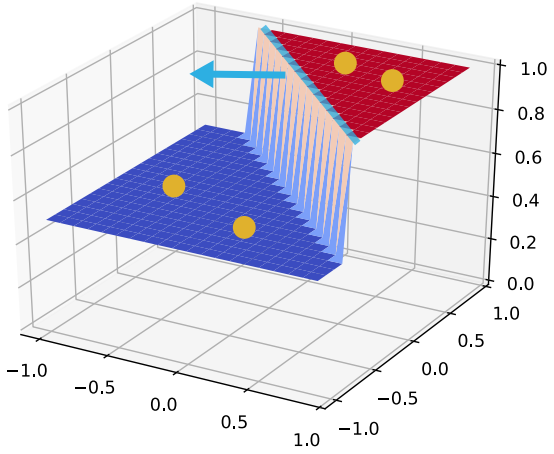


Fig. 3. A discontinuous integrand does not lead to non-differentiable integral. Here we show a discontinuous function $z = x + y > \theta \; ? \; 1.0 \; : \; 0.5$ and we want to integrate it over $[-1, 1] \times [-1, 1]$. The integrand z is not differentiable with respect to the boundary parameter θ. Therefore, all the yellow samples will return zero if we ask them the derivative with respect to θ. However, changing θ (the blur arrow) will lead to a **continuous** change to the volume below the curve. This is because derivative of a sample can only detect local changes, and the chance a sample lands exactly at the boundary is zero.

then the gradient is

$$\sum_i 2 \left(I_i(\boldsymbol{\pi}) - \hat{I}_i(\boldsymbol{\pi}) \right) \frac{\partial I_i(\boldsymbol{\pi})}{\partial \pi}. \tag{5}$$

The loss can be any differentiable function, including a neural network. We can compute L's gradient with respect to the image ($\partial L / \partial I_i(\boldsymbol{\pi})$) efficiently using the backpropagation or, equivalently, reverse-mode automatic differentiation algorithm (Section 5.2).

Now we want to compute the derivative of a pixel color with respect to the scene parameters $\partial I_i(\boldsymbol{\pi}) / \partial \boldsymbol{\pi}$. A common misconception of the non-differentiability of rendering is that the derivative $\partial I_i(\boldsymbol{\pi}) / \partial \pi$ is discontinuous and not differentiable. However, recall that I_i is an integral that evaluates the average color within the filter support. Therefore, the movement of the triangle vertices will in fact lead to continuous and differentiable changes to the average color (Fig. 3). **The *integrand* of rendering is discontinuous and not differentiable, but the *integral* is actually differentiable!** Importantly, we did not make rendering an integration problem to make it differentiable. Instead, rendering is an integration problem in the first place. All the approaches in any rendering methods, real-time or offline, are different approximations or discretizations of the rendering integral.

How do we compute the derivatives of an integral? Recall that we wanted to compute the integral numerically (Eq. (2)). Unfortunately, we cannot just automatically differentiate the numerical integrator. For the vertex position parameters, the numerical integrator will always evaluate to 0 (the function return the color based on which triangle (x, y) hits). However, the derivative of the *integral* with respect to a vertex position parameter $\pi_v \in \boldsymbol{\pi}_v$ is

not 0 (Fig. 3).

$$\frac{\partial}{\partial \pi_v} \iint f(x, y; \boldsymbol{\pi}) \mathrm{d}x\mathrm{d}y \not\approx \frac{1}{N} \sum_{j=1}^{N} \frac{\partial}{\partial \pi_v} f(x_j, y_j; \boldsymbol{\pi}) = 0.^{1} \tag{6}$$

In general, the discretization and the gradient operator do not commute for discontinuous integrands. This is because derivatives are measuring local changes, and uniform discretization has zero chance of detect the local changes around discontinuities. To see this, let us simplify the problem and consider the following 1D derivative integral with a step function:

$$\frac{\partial}{\partial p} \int_0^1 (x < p \ ? \ 1 : 0) \, \mathrm{d}x. \tag{7}$$

If we discretize this function and evaluate derivative with respect to p, the derivative is always 0. However, the integral evaluate to p for $0 \leq p \leq 1$ and thus the derivative is 1. This is because the discretization relies on point sampling of the integrand. However, the point sampling can only access the integrand *locally* through its infinitesimal neighborhood. Given a sample x_j and its small neighborhood $(x_j - \epsilon, x_j + \epsilon)$, the only location where change happens is at $x = p$. However, the chance of us hitting $x = p$ randomly is exactly zero. Therefore we will never detect the change of the integral through discretization.

Our strategy to differentiate the integrals is then to *explicitly* put samples at the discontinuities to detect the local change. In the following we provide the mathematical tools that allows us to do this.

2.3 Distributional Derivatives and Dirac Delta

One way to formalize the statement above mathematically is to use the concept of *distributional derivatives*, which defines the derivative for even discontinuous functions. In the case above, the distributional derivative of the integrand with respect to p is a *Dirac delta* δ:

$$\frac{\partial}{\partial p} x < p \ ? \ 1 : 0 = \delta(x - p). \tag{8}$$

$\delta(x - p)$ is not a function but a *distribution*. We cannot just feed a Dirac delta an input value and get an output value back. We can only evaluate Dirac delta through an integral (usually called the *test function*). We define the Dirac delta's integral as $\int_0^1 \delta(x - p) = 1$ if $0 \leq p \leq 1$, otherwise it is 0.

Mathematically, doing algebra with Dirac delta inside an integral and derive the correct measure is not entirely trivial. This is mostly due to the fact that Dirac delta is a distribution and not a function. For example, the multiplication of two Dirac deltas is not well-defined. While it is possible to derive differentiable rendering using Dirac delta [24, 38], in this course note we use an alternative strategy that leads to simpler derivations in higher-dimensional space.

2.4 Leibniz's Rule for Differentiating 1D Integrals

Our alternative strategy is to, conceptually, split the integral such that all the discontinuities are at the boundaries. We will demonstrate this for 1D functions, and generalize this to higher-dimensional space in the next subsection.

For the step function example we can split it into two constant integrals

$$\frac{\partial}{\partial p} \int_0^1 (x < p \ ? \ 1 : 0) \, \mathrm{d}x = \frac{\partial}{\partial p} \int_0^p 1 \mathrm{d}x + \frac{\partial}{\partial p} \int_p^1 0 \mathrm{d}x. \tag{9}$$

Importantly, the splitting is done only for derivation purposes, and we do not actually clip the polygons for computation.

[1] Here $\not\approx$ means that when we choose x_j, y_j randomly, the sum will not converge to the integral even if N goes to infinity. In contrast, the original approximation (Eq. (2)) does converge to the corresponding integral when we choose the samples randomly.

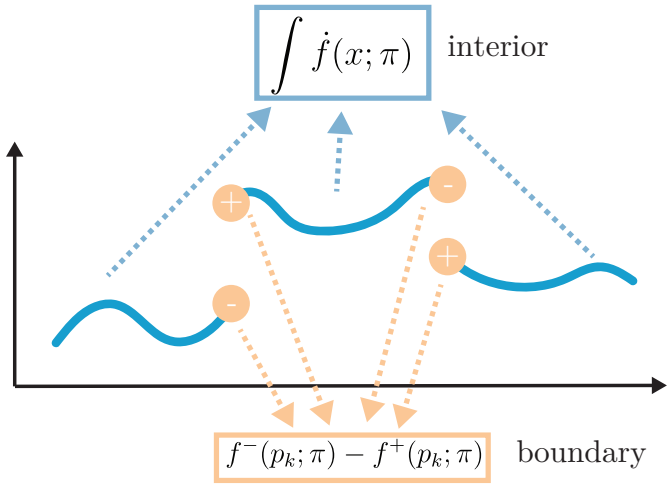


Fig. 4. To compute the derivative of an 1D integral with a discontinuous integrand, given a set of boundary points (orange) that include all discontinuities, we can compute the derivatives by separately computing the derivatives of the continuous part (the **interior** derivative), and the discontinuous part (the **boundary** derivative). The set of boundary points can include continuous points, since the difference $f^- - f^+$ evaluates to zero for those points.

After the splitting, the derivative can then be evaluated through the fundamental theorem of calculus, i.e.,

$$\frac{\partial}{\partial p}\int_0^p 1\mathrm{d}x = 1, \frac{\partial}{\partial p}\int_p^1 0\mathrm{d}x = 0, \tag{10}$$

thus the derivative evaluate to 1.

The general version of differentiation with respect to both the integral boundaries and the integrand is the Leibniz's rule. Formally, let $\pi \in \mathbb{R}$ and consider the following Riemann integral over some interval $(a(\pi), b(\pi)) \subset \mathbb{R}$:

$$\int_{a(\pi)}^{b(\pi)} f(x;\pi)\,\mathrm{d}x, \tag{11}$$

where the integrand f is differentiable everywhere with respect to x and π. Then, *Leibniz's rule for differentiation under the integral sign* [10] states that the derivative of Eq. (11) with respect to π is

$$\frac{\partial}{\partial\pi}\int_{a(\pi)}^{b(\pi)} f(x;\pi)\,\mathrm{d}x = \underbrace{\int_{a(\pi)}^{b(\pi)} \dot{f}(x;\pi)\,\mathrm{d}x}_{\text{interior}} + \underbrace{\dot{b}(\pi)\,f(b(\pi);\pi) - \dot{a}(\pi)\,f(a(\pi),\pi)}_{\text{boundary}}, \tag{12}$$

where $\dot{f} := {}^{\partial f}/_{\partial\pi}$, $\dot{a} := {}^{\mathrm{d}a}/_{\mathrm{d}\pi}$, and $\dot{b} := {}^{\mathrm{d}b}/_{\mathrm{d}\pi}$. This equation involves a *interior* and a *boundary* term where the former equals the original integral (11) with the integrand differentiated while the latter given by the integrand evaluated at the integral boundaries $a(\pi)$ and $b(\pi)$ modulated by their change rates $\dot{a}(\pi)$ and $\dot{b}(\pi)$. The proof of the equation easily follows from the fundamental theorem of calculus.

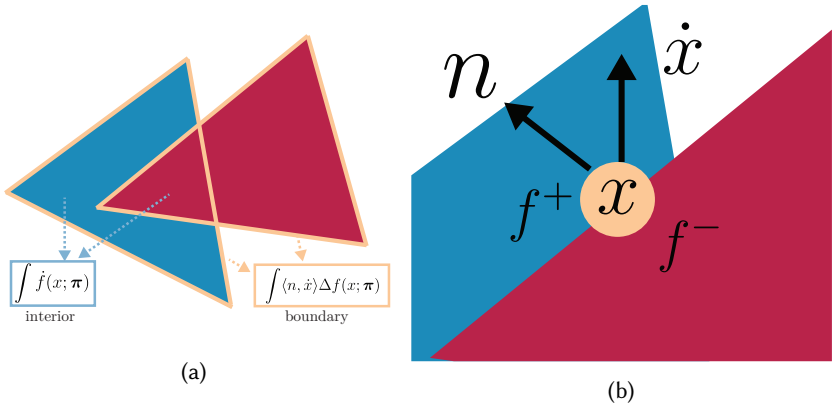


Fig. 5. (a) illustrates Reynolds transport theorem for differentiating 2D integrals. In contrast to the 1D case, our boundary term becomes an integral as well. As in the 1D case, our boundary points can include points that land in the continuous regions of the integrand, such as the occluded part of the blue triangle. For the continuous regions, the difference $\delta f(x;\pi)$ evaluates to 0. b shows that, for each point on the boundary, we need to compute the 2D movement of the boundary with respect to the differentiating parameter $\dot{x}$. We then need to project $\dot{x}$ to the normal direction and multiply it with the difference. This is to account for the infinitesimal *width* of our line integral, so that we can properly measure the infinitesimal area changes at the boundary.

Therefore, given an 1D integral $\int_0^1 f(x;\pi)dx$, and a set of points $p_0, p_1, ..., p_{M-1}$ represent the point of discontinuities with respect to the parameter π, the derivative of the integral is

$$\frac{\partial}{\partial \pi} \int_0^1 f(x;\pi)\, \mathrm{d}x = \underbrace{\int_0^1 \dot{f}(x;\pi)\, \mathrm{d}x}_{\text{interior}} + \underbrace{\sum_{k=0}^{M} \left(f^-(p_k;\pi) - f^+(p_k;\pi) \right)}_{\text{boundary}}, \tag{13}$$

where we define $f^-(x;\pi)$ and $f^+(x;\pi)$ as the right-hand and left-hand limits of f at x respectively, i.e., $f^-(x;\pi) = \lim_{x' \to x^-} f(x';\pi)$ and $f^+(x;\pi) = \lim_{x' \to x^+} f(x';\pi)$. For notational convenience, when f is not differentiable at $(x;\pi)$, we define $\dot{f}(x;\pi) = 0$. Intuitively, we are explicitly summing over the difference at the two sides of the discontinuities. Fig. 4 illustrates this process. We can derive the exact same results using distributional derivatives 2.3: the derivatives at discontinuities at p_k result in Dirac delta signals, which we can explicitly integrate into the discrete sum.

We can then estimate the derivative integral (Eq. (13)) by discretizing the interior integral, and sum over all the differences. Note that, first, the set of discontinuity points p_k can include points that turn out to be continuous, since for continuous regions the difference $f^-(x;\pi) = f^+(x;\pi)$. This is important for handling occlusion later in 2D when we do not know if a triangle boundary is occluded or not. Second, we do not need to actually decompose the integral, since the continuous regions can be assembled back to the full integral.

Next we will generalize our approach to 2D and higher-dimensional spaces. To do this we will use a mathematical tool in fluid mechanics that was used for tracking the evolution of a surface, called Reynolds transport theorem.

2.5 Reynolds Transport Theorem for Differentiating Multi-dimensional Integrals

In two-dimensional space, our discontinuities become lines instead of points (point discontinuities in 2D will be smoothed out by the integrals and do not affect the derivatives). In our two triangles example (Fig. 2), the

discontinuities are defined by the edges of the triangles. Similar to the 1D case, we want to explicitly sample the discontinuities, and evaluate the difference at the two sides. The point evaluation over the discontinuities in 1D becomes integrals over the lines in 2D. We can formalize this idea using Reynolds transport theorem, a generalization of the Leibniz's rule for differentiation (12).

Formally, let f be a (potentially discontinus) scalar-valued function defined on some n-dimensional manifold $\Omega(\pi)$ parameterized with some $\pi \in \mathbb{R}$. Additionally, let $\Gamma(\pi) \subset \Omega(\pi)$ be an $(n-1)$-dimensional manifold given by the union of the *external* boundary $\partial\Omega(\pi)$ and the *internal* one containing the discontinuous locations of f. Then, it holds that

$$\frac{\partial}{\partial \pi}\left(\int_\Omega f \,\mathrm{d}\Omega\right) = \underbrace{\int_\Omega \dot{f} \,\mathrm{d}\Omega}_{\text{interior}} + \underbrace{\int_\Gamma \langle \boldsymbol{n}, \dot{\boldsymbol{x}} \rangle \, \Delta f \,\mathrm{d}\Gamma}_{\text{boundary}}, \tag{14}$$

where $\dot{\boldsymbol{x}} := \frac{\partial \boldsymbol{x}}{\partial \pi}$ (note that $\dot{\boldsymbol{x}}$ is a n-dimensional vector, representing the relative boundary movement with respect to the parameter π), $\mathrm{d}\Omega$ and $\mathrm{d}\Gamma$ respectively denote the standard measures associated with Ω and Γ; and $\langle \cdot, \cdot \rangle$ indicates vector dot (inner) product. As in the 1D case, when f is not differentiable at $(x; \pi)$, we define $\dot{f}(x; \pi) = 0$. Further, $\boldsymbol{n}$ is the normal direction at each $\boldsymbol{x} \in \Gamma(\pi)$, and Δf is given by[2]

$$\Delta f(\boldsymbol{x}) := \lim_{\epsilon \to 0^-} f(\boldsymbol{x} + \epsilon \boldsymbol{n}) - \lim_{\epsilon \to 0^+} f(\boldsymbol{x} + \epsilon \boldsymbol{n}), \tag{15}$$

for all $\boldsymbol{x} \in \Gamma(\pi)$.

Fig. 5 illustrates the idea. As in the 1D case (Eq. (13)), we have the interior term and the boundary term. The interior term is just the derivative of the continuous part of the integrand. The boundary term is now an *integral* that integrates over the difference at the $(n-1)$-dimensional discontinuities Γ (all the discontinuities with lower dimensionality do not affect the derivative). For each point on the discontinuity, we compute the difference of the integrand at the two sides of the boundary (Δf). To take the infinitesimal area of the boundary changes with respect to the differentiating parameters into account, we multiply the difference with the speed of the boundary movement with respect to the parameters ($\dot{\boldsymbol{x}}$), projected to the normal direction (Fig. 5b). This multiplication is in contrast to the 1D case, where the boundary can only move left or right and is always aligned with the *normal* direction. See Flanders' article for a simple proof of Reynolds transport theorem [10].

Discretizing the integrals. We now know how to compute the rendering derivative of our triangle rendering example. Given our anti-aliasing integral (Eq. (1)), we separate its derivative integral into an interior integral and a boundary integral using Reynolds transport theorem (Eq. (14)). We can then estimate the two integrals by discretizing them:

$$\begin{aligned} \int_\Omega \dot{f} \,\mathrm{d}\Omega &\approx \frac{1}{N_i} \sum_{j=1}^{N_i} \dot{f}(x_j) \\ \int_\Gamma \langle \boldsymbol{n}, \dot{\boldsymbol{x}} \rangle \, \Delta f \,\mathrm{d}\Gamma &\approx \frac{1}{N_b} \sum_{j=1}^{N_b} \langle \boldsymbol{n}, \dot{\boldsymbol{x}} \rangle \, \Delta f(x_j) \end{aligned}. \tag{16}$$

For the interior integral, we can just use the same discretization strategy in normal rendering. For example, we can randomly sample inside the filter support. We can also divide the filter support into a grid, and sample the center at each grid. Both rasterization *and* ray tracing can be used for computing the interior integral. For the constant color parameters, $\frac{\partial f}{\partial \pi_c} = 1$. For the vertex positions parameters, $\frac{\partial f}{\partial \pi_v} = 0$.

[2]When $\boldsymbol{x}$ approaches $\Gamma(\pi)$ from the exterior of the integral domain $\Omega(\pi)$, the corresponding one-sided limit in Eq. (15) is set to zero.

For the boundary integral, we use the fact that the boundary domain Γ can be include continuous points. For the points that land in the continuous regions, Δf evaluates to 0 due to the definition of continuity. Therefore we define Γ to be the set of points on all the triangle edges. To select a point on the edge, we first randomly pick an edge (there are 6 edges out of 2 triangles), we then uniformly pick a point on the edge. To evaluate the difference Δf, in practice we select a small ϵ and query f from the two sides. Furthermore, we are estimating many pixel integrals I_i in the same time. When we pick a point on the edge, we will gather all the pixels whose filter support intersect with the point we pick, and scatter the derivative to the corresponding pixel integral discretization.

When there are more triangles in the scene, this can be accelerated by pruning out the set of edges that are not the *silhouette* and importance sampling. We will discuss more about the discretization of the boundary integral in Section 5.1.1.

2.6 Relation to Rasterization-based Differentiable Renderers

Our discretization strategy for the boundary integral above can only be efficiently computed using a ray-tracing based visibility engine, since it requires random query to the pixel integrand f. Using hardware rasterization to compute it is possible but likely to be wasteful.

However, many existing *rasterization*-based differentiable renderers can be seen as a particular discretization to the boundary integral (Eq. (16)). For example, OpenDR [26] approximates the boundary term $\langle \boldsymbol{n}, \dot{\boldsymbol{x}} \rangle \Delta f$ using pixel finite difference, and choose between central difference, forward difference and backward difference depending on the result of the object ID detection. Kato et al. [21], on the other hand, apply an edge rasterization pass by rasterizing all the edges, while interpolating the color buffer to compute the difference. de La Gorce et al. [9], inspired by discontinuity edge overdraw [39], also perform an edge rasterization pass, but compute the difference by assuming the mesh is a closed manifold, and compute the shading twice at the two sides of the silhouette.

Liu et al.'s work [25] does not fully fit in our theory. They tackled the differentiable rendering problem by changing the forward rendering formulation to be differentiable without the boundary integral, through introducing two modifications: 1) They analytically approximate the spatial anti-aliasing integral (Eq. (1)) using a sigmoid function by assuming only one triangle is inside the pixel. 2) They make all the surfaces in the scene transparent using a form of weighted order independent transparency [28].

In contrast, our theory can support transparent surfaces as well without any modification. It works for both order independent transparency and the more traditional OVER operator [37] – they are just different rendering integrands f. We do not need to resort to spatial analytical approximation, which can produce artifacts when many triangles are in a single pixel or when texture is presented. Rendering transparent surfaces efficiently is a longstanding problem in real-time rendering, and it usually makes efficient occlusion culling techniques not applicable. On the other hand, transparency can make inverse rendering more tractable and less susceptible to local minima.

These methods, while being different approximations to the anti-aliasing and boundary integrals, are not *consistent* nor *unbiased* discretization. This can have negative impacts on the convergence of stochastic gradient descent, which assumes that the approximated gradients converge to the correct gradient.

The performance of rasterization v.s. ray tracing is a longstanding debate [8, 16]. Differentiable rendering further introduces more performance tradeoffs with different memory access patterns and arithmetic intensity characteristics. While the performance comparison between rasterization and ray tracing under the differentiable rendering context deserves at least a full stand-alone research article, we list a few remarks here:

- A common misconception is that ray tracing can only be used for computing *advanced* effects such as shadow and global illumination. In contrast, the techniques we introduced here can be used for rendering local shading as well.

- Visibility query is usually *not* the biggest bottleneck in current differentiable rendering systems. Instead, most of the rendering time is spent on incoherent memory access, irregular scattering, and atomic operations during the derivative computation.
- The interior integral and the boundary integral do not need to be computed using the same visibility algorithm. One can compute the interior integral using rasterization, while computing the boundary integral using ray tracing.
- Many acceleration strategies that are used in rasterization, such as the hierarchical Z-buffer [12] for occlusion culling, can also be used for ray tracing based boundary evaluation (we can reject edges using occlusion culling techniques).
- The set of the boundary is often sparse in the image space compared to the number of pixels. It is conceivable that we can compute the boundary integral with much less samples (N_b) compared to the number of pixels while achieving very low error.
- Ray tracing is getting hardware support as well. The performance gap between coherent ray tracing and rasterization is closing.

Nevertheless, our goal with the theory presented in this course is to guide future developments of differentiable rendering with a principled mathematical model. We hope that our theory can be the foundation of future approximation methods.

In the next section, we will generalize our theory to handle the recursive rendering equation that computes global illumination, and the radiative transfer equation that models participating media.

3 PHYSICS-BASED DIFFERENTIABLE RENDERING THEORY

Given the mathematical tools described in §2, we now provide an in-depth introduction to the theory of physics-based differentiable rendering. Specifically, we discuss in §3.1 the differentiation of the rendering equation (RE) [20] with respect to arbitrary scene parameters. Then, we present in §3.2 the differentiation of the radiative transfer equation (RTE) [5], another key equation to physics-based rendering.

3.1 Differentiable Rendering of Surfaces

Physics-based rendering of surfaces has been a central topic in computer graphics for decades and is governed by the well-known *rendering equation* (RE) [20]. The RE is an integral equation stating that the (steady-state) radiance L at any surface point $\boldsymbol{x}$ with direction $\boldsymbol{\omega}_\mathrm{o}$ is given by:

$$L(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) = L_\mathrm{e}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) + \int_{\mathbb{S}^2} L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})\, \mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i}), \tag{17}$$

where L_i indicate the incident radiance, f_s denotes the cosine-weighted BSDF, and $\mathrm{d}\sigma$ is the solid-angle measure.

The RE has no analytical solution in general, and numerous numerical methods have been developed. Some of the widely adopted examples include unbiased methods like unidirectional and bidirectional path tracing, as well as biased ones such as photon mapping and lightcuts. [SZ: citations]

3.1.1 Direct Illumination. Before differentiating the full RE (17), we first consider the case of direct illumination as a warm-up. Specifically, the radiance L_r resulting from exactly one reflection at a surface point $\boldsymbol{x}$ with direction $\boldsymbol{\omega}_\mathrm{o}$ equals

$$L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) = \int_{\mathbb{S}^2} L_\mathrm{e}(\boldsymbol{y}, -\boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})\, \mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i}), \tag{18}$$

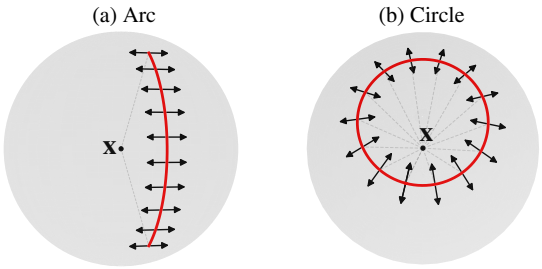


Fig. 6. The normal directions of arcs and circles (that are respectively the projections of line segments and spheres) as spherical curves.

where $\boldsymbol{y}$ is the closest intersection of a light ray originated at $\boldsymbol{x}$ with direction $\boldsymbol{\omega}_\mathrm{i}$. That is, $\boldsymbol{y} = \mathrm{rayTrace}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})$. Unlike the RE (17) that takes the form of an integral equation, Eq. (18) is a simple spherical integral as its right-hand side involves only known quantities.

We now consider the problem of calculating the derivative of $L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})$ with respect to some abstract scene parameter $\pi \in \mathbb{R}$. Given $\boldsymbol{x}$ and $\boldsymbol{\omega}_\mathrm{o}$, let $f_\mathrm{direct}(\boldsymbol{\omega}_\mathrm{i};\, \boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) := L_\mathrm{e}(\boldsymbol{y}, -\boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})$. It holds that

$$[L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})]^\bullet := \frac{\mathrm{d}}{\mathrm{d}\pi}[L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})] = \frac{\mathrm{d}}{\mathrm{d}\pi}\left(\int_{\mathbb{S}^2} f_\mathrm{direct}(\boldsymbol{\omega}_\mathrm{i};\, \boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})\, \mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i})\right). \tag{19}$$

By applying Reynolds transport theorem (14) to this equation, we have

$$[L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})]^\bullet = \underbrace{\int_{\mathbb{S}^2} [f_\mathrm{direct}(\boldsymbol{\omega}_\mathrm{i};\, \boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})]^\bullet\, \mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i})}_{\text{interior}} + \underbrace{\int_{\Delta\mathbb{S}^2} \langle \boldsymbol{n}_\perp, \dot{\boldsymbol{\omega}}_\mathrm{i} \rangle\, \Delta f_\mathrm{direct}(\boldsymbol{\omega}_\mathrm{i};\, \boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})\, \mathrm{d}\ell(\boldsymbol{\omega}_\mathrm{i})}_{\text{boundary}}. \tag{20}$$

where $\mathrm{d}\ell$ is the curve-length measure. In this equation, the *interior* term is an integral over the unit sphere $\mathbb{S}^2$, which is independent of the scene parameter π, making the integral variable $\boldsymbol{\omega}_\mathrm{i}$ also be π-independent. The *boundary* term, on the other hand, emerges due to the (jump) discontinuity points of f_direct (with respect to $\boldsymbol{\omega}_\mathrm{i}$) and captures how they "move" π varies. These discontinuity points forms 1D *discontinuity curves*, which we denote as $\Delta\mathbb{S}^2$, over the unit sphere. For any $\boldsymbol{\omega}_\mathrm{i} \in \mathbb{S}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})$, $\boldsymbol{n}_\perp(\boldsymbol{\omega}_\mathrm{i})$ is a vector in the tangent space of $\mathbb{S}^2$ at $\boldsymbol{\omega}_\mathrm{i}$ perpendicular to the discontinuity curve (see Figure 6). with unit-normal field $\boldsymbol{n}_\perp$ (see Figure 6).

Assuming the (cosine-weighted) BSDF $f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o}$ to be continuous with respect to $\boldsymbol{\omega}_\mathrm{i}$, which is usually the case except for perfectly specular BSDFs, the discontinuities of the integrand f_direct fully emerge from those of the incident emission $L_\mathrm{e}(\boldsymbol{y}, \boldsymbol{\omega}_\mathrm{i})$, which is generally discontinuous due to occulsions [SZ: Add a figure.] Therefore,

$$\Delta f_\mathrm{direct}(\boldsymbol{\omega}_\mathrm{i};\, \boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) = f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})\, \Delta L_\mathrm{e}(\boldsymbol{y}, -\boldsymbol{\omega}_\mathrm{i}). \tag{21}$$

3.1.2 Differential Rendering Equation. Based on the analyses in §3.1.1, we now differentiate the full rendering equation (RE) (17) using Reynolds transport theorem (14). This yields another integral equation, which we call the *differential rendering equation*:

$$[L(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})]^\bullet = [L_\mathrm{e}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})]^\bullet + \underbrace{\int_{\mathbb{S}^2} [L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})]^\bullet \mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i})}_{\text{interior}} + \underbrace{\int_{\Delta\mathbb{S}^2} \langle \boldsymbol{n}_\perp, \dot{\boldsymbol{\omega}}_\mathrm{i} \rangle\, f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i}, \boldsymbol{\omega}_\mathrm{o})\, \Delta L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, \mathrm{d}\ell(\boldsymbol{\omega}_\mathrm{i})}_{\text{boundary}}. \tag{22}$$

Similar to its original counterpart (17), the differential rendering equation (22) has no analytical soltuion in general. We will discuss Monte Carlo solutions to this equation in §5.1.

3.2 Differentiable Rendering of Participating Media

Besides surface reflection and refraction, another important type of light-transport effects is volumetric: light can be absorbed and scattering within participating media comprised of microscopic particles. To model volumetric scattering of light, the radiative transfer theory (RTT) [5] has been introduced by physicists half a century ago. We review RTT in §3.2.1 and present its differential variant in §3.2.2.

3.2.1 Radiative Transfer Theory Preliminaries. *Radiative transfer* [5] uses energy conservation principles to model light transport in participating media. At its core is the *radiative transfer equation* (RTE): Consider a medium confined in a volume $\Omega \subseteq \mathbb{R}^3$ with boundary $\partial\Omega$. The *steady-state* RTE is a linear integral equation on the radiance field L in the interior of the volume that can be expressed in operator form as [45]:

$$L = (\mathcal{K}_\mathrm{T}\,\mathcal{K}_\mathrm{C} + \mathcal{K}_\mathrm{S})\,L + L^{(0)}. \tag{23}$$

In this equation, the *transport operator* $\mathcal{K}_\mathrm{T}$ maps any function $g : (\Omega \setminus \partial\Omega) \times \mathbb{S}^2 \to \mathbb{R}_+$ to a new function

$$(\mathcal{K}_\mathrm{T}\,g)(\boldsymbol{x},\boldsymbol{\omega}) = \int_0^D T(\boldsymbol{x}',\boldsymbol{x})\,g(\boldsymbol{x}',\boldsymbol{\omega})\,\mathrm{d}\tau, \tag{24}$$

where: $\boldsymbol{x}' \coloneqq \boldsymbol{x} - \tau\boldsymbol{\omega}$; D is the distance from $\boldsymbol{x}$ to the medium's boundary in the direction of $-\boldsymbol{\omega}$;

$$D = \inf\{\tau \in \mathbb{R}_+ \;:\; \boldsymbol{x} - \tau\boldsymbol{\omega} \in \partial\Omega\}; \tag{25}$$

and $T(\boldsymbol{x}',\boldsymbol{x})$ is the *transmittance* between $\boldsymbol{x}'$ and $\boldsymbol{x}$, that is, the fraction of light that transmits straight between $\boldsymbol{x}'$ and $\boldsymbol{x}$ without being absorbed or scattered away:

$$T(\boldsymbol{x}',\boldsymbol{x}) = \exp\left(-\int_0^\tau \sigma_\mathrm{t}(\boldsymbol{x} - \tau'\boldsymbol{\omega})\,\mathrm{d}\tau'\right), \tag{26}$$

with σ_t denoting the medium's *extinction coefficient*.

In Eq. (23), the *collision operator* $\mathcal{K}_\mathrm{C}$ maps the interior radiance field L to[3]

$$(\mathcal{K}_\mathrm{C}\,L)(\boldsymbol{x},\boldsymbol{\omega}) = \sigma_\mathrm{s}(\boldsymbol{x}) \underbrace{\int_{\mathbb{S}^2} f_\mathrm{p}(\boldsymbol{x},-\boldsymbol{\omega}_\mathrm{i},\boldsymbol{\omega})\,L(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i})\,\mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i})}_{=:\,L^\mathrm{ins}(\boldsymbol{x},\boldsymbol{\omega})}, \tag{27}$$

where σ_s and f_p respectively denote the medium's *scattering coefficient* and *single-scattering phase function*, and L^ins is usually termed as the *in-scattered* radiance.[4] Additionally, the *interfacial scattering operator* $\mathcal{K}_\mathrm{S}$ follows the rendering equation (17) and is defined as

$$(\mathcal{K}_\mathrm{S}\,L)(\boldsymbol{x},\boldsymbol{\omega}) = T(\boldsymbol{x}_0,\boldsymbol{x}) \int_{\mathbb{S}^2} f_\mathrm{s}(\boldsymbol{x}_0,-\boldsymbol{\omega}_\mathrm{i},\boldsymbol{\omega})\,L(\boldsymbol{x}_0,\boldsymbol{\omega}_\mathrm{i})\,\mathrm{d}\sigma(\boldsymbol{\omega}_\mathrm{i}), \tag{28}$$

where $\boldsymbol{x}_0 \coloneqq \boldsymbol{x} - D\boldsymbol{\omega}$ is a point on the medium's boundary. This term accounts for contributed radiance due to reflection and transmission at the medium's boundary.

The last term on the right-hand side (RHS) of the RTE (23) is

$$L^{(0)}(\boldsymbol{x},\boldsymbol{\omega}) \coloneqq (\mathcal{K}_\mathrm{T}\,\sigma_\mathrm{a}\,Q)(\boldsymbol{x},\boldsymbol{\omega}) + T(\boldsymbol{x}_0,\boldsymbol{x})\,L_\mathrm{e}(\boldsymbol{x}_0,\boldsymbol{\omega}), \tag{29}$$

[3] In this course, we follow the convention that all directions point away from $\boldsymbol{x}$ when expressing BSDFs and phase functions, yielding the negative sign before $\boldsymbol{\omega}_\mathrm{i}$ in Eq. (27).

[4] Precisely, L^ins captures the in-scattered radiance before volumetric absorption.

$$\dot{L} = \underbrace{(\mathcal{K}_\mathrm{T}\,\mathcal{K}_\mathrm{C}\,L)^{\cdot}}_{\S 3.2.3} + \underbrace{(\mathcal{K}_\mathrm{S}\,L)^{\cdot}}_{\S 3.2.5} + \underbrace{\dot{L}^{(0)}}_{\S 3.2.6}.$$

Fig. 7. An **outline** of the derivations of the differential radiative transfer equation.

which indicates the contribution of radiant emission in the medium and from its boundary. In this equation, $\sigma_\mathrm{a} \coloneqq \sigma_\mathrm{t} - \sigma_\mathrm{s}$ is the *absorption coefficient*, and Q represents the medium's *radiant emission*.

3.2.2 Differential Radiative Transfer. We now present a differential theory of radiative transfer introduced by Zhang et al. [45] by showing how the interior radiance L can be differentiated with respect to some scene parameter $\pi \in \mathbb{R}$. To this end, we derive the partial derivative $\dot{L} \coloneqq {}^{\partial L}/_{\partial \pi}$ by differentiating each of the operators on the right-hand side (RHS) of Eq. (23).

Assumptions. As most participating media and translucent materials are non-emissive, we neglect the volumetric emission term Q in Eqs. (29) when deriving $\dot{L}$. Additionally, we assume that: (i) the RTE and RE parameters σ_t, σ_s, f_p, f_s, and L_e are *continuous* spatially and directionally; and (ii) there are no *zero-measure* light sources (e.g., point and directional) or *ideal specular surfaces* (e.g., perfect mirrors).

Overview. Figure 7 outlines the structure of our derivations. As a preview, based on the assumptions above, the scene partial derivative $\dot{L}$ will take the form of Eq. (45).

3.2.3 Differentiation of the transport and collision operators. We start with the first term on the RHS in Figure 7, $\partial_\pi \mathcal{K}_\mathrm{T}\,\mathcal{K}_\mathrm{C}\,L$, which can be shown to equal [45]:

$$\begin{aligned}(\partial_\pi \mathcal{K}_\mathrm{T}\,\mathcal{K}_\mathrm{C}\,L)\,(\boldsymbol{x},\boldsymbol{\omega}) = &\int_0^D T(\boldsymbol{x}',\boldsymbol{x})\,\sigma_\mathrm{s}(\boldsymbol{x}')\,\dot{L}^\mathrm{ins}(\boldsymbol{x}',\boldsymbol{\omega})\,\mathrm{d}\tau \\ &+ \int_0^D T(\boldsymbol{x}',\boldsymbol{x})\,[\dot{\sigma}_\mathrm{s}(\boldsymbol{x}') - \Sigma_\mathrm{t}(\boldsymbol{x},\boldsymbol{\omega},\tau)\,\sigma_\mathrm{s}(\boldsymbol{x}')]\,L^\mathrm{ins}(\boldsymbol{x}',\boldsymbol{\omega})\,\mathrm{d}\tau + \dot{D}\,T(\boldsymbol{x}_0,\boldsymbol{x})\,\sigma_\mathrm{s}(\boldsymbol{x}_0)\,L^\mathrm{ins}(\boldsymbol{x}_0,\boldsymbol{\omega}), \quad (30)\end{aligned}$$

where $\boldsymbol{x}' \coloneqq \boldsymbol{x} - \tau\boldsymbol{\omega}$ is a point on the ray originated from $\boldsymbol{x}$ with direction $-\boldsymbol{\omega}$; $\boldsymbol{x}_0 \coloneqq \boldsymbol{x} - D\boldsymbol{\omega}$ is the location where the ray intersects the medium boundary; L^ins is the in-scattered radiance from Eq. (27); and $\Sigma_\mathrm{t}(\boldsymbol{x},\boldsymbol{\omega},\tau)$, defined as

$$\Sigma_\mathrm{t}(\boldsymbol{x},\boldsymbol{\omega},\tau) \coloneqq \int_0^\tau \dot{\sigma}_\mathrm{t}(\boldsymbol{x} - \tau'\boldsymbol{\omega})\,\mathrm{d}\tau', \quad (31)$$

appears when we differentiate the transmittance $T(\boldsymbol{x}',\boldsymbol{x})$.

In Eqs. (30) and (31), $\dot{\sigma}_\mathrm{t}$ and $\dot{\sigma}_\mathrm{s}$ are essentially *material derivatives* given by [22]

$$\dot{\sigma}(\boldsymbol{x}) = \frac{\partial \sigma}{\partial \pi}(\boldsymbol{x}) + \langle \dot{\boldsymbol{x}}, \nabla\sigma(\boldsymbol{x})\rangle, \quad (32)$$

for $\sigma \in \{\sigma_\mathrm{t}, \sigma_\mathrm{s}\}$ with ∇ being the gradient operator.

Evaluating the RHS of Eq. (30) also requires $\dot{D}$, $L^\mathrm{ins}(\boldsymbol{x}_0,\boldsymbol{\omega})$, and $\dot{L}^\mathrm{ins}(\boldsymbol{x}',\boldsymbol{\omega})$. In what follows, we derive formulas for these terms.

In-scattered radiance at the boundary. The last term required for evaluating $\partial_\pi \mathcal{K}_\mathrm{T}\,\mathcal{K}_\mathrm{C}\,L$ in Eq. (30) involves the in-scattered radiance L^ins at boundary point $\boldsymbol{x}_0$. Caution is needed here since light transport behaves differently at the two sides of the boundary.

$L^\mathrm{ins}(\boldsymbol{x}_0,\boldsymbol{\omega})$ is given by the limit of $L^\mathrm{ins}(\boldsymbol{x}',\boldsymbol{\omega})$ as $\boldsymbol{x}'$ approaches the boundary location $\boldsymbol{x}_0$ from the interior of the medium along the direction of $-\boldsymbol{\omega}$. Namely, $L^\mathrm{ins}(\boldsymbol{x}_0,\boldsymbol{\omega}) = \lim_{\tau\to D^-} L^\mathrm{ins}(\boldsymbol{x}',\boldsymbol{\omega})$ with $\boldsymbol{x}' \coloneqq \boldsymbol{x} - \tau\boldsymbol{\omega}$. This limit

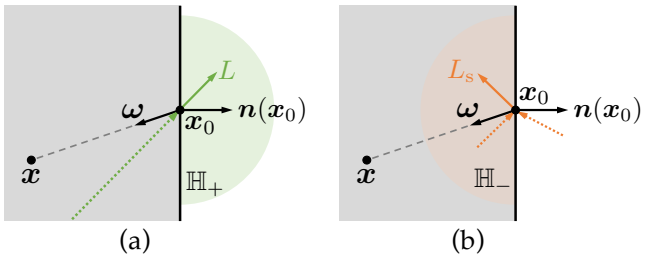


Fig. 8. Calculating the in-scattered radiance L^{ins} at location $\boldsymbol{x}_0 \in \partial\Omega$ with direction $\boldsymbol{\omega}$ pointing toward the interior of the medium (illustrated in gray). (a) When $\boldsymbol{\omega}' \in \mathbb{H}_+$ (i.e., pointing toward the exterior of the medium), the interior radiance $L(\boldsymbol{x}_0, \boldsymbol{\omega}')$ involving a line integral (indicated as the dashed line in green) from the interior is used. (b) When $\boldsymbol{\omega}' \in \mathbb{H}_-$, on the contrary, the interfacial radiance $L(\boldsymbol{x}_0, \boldsymbol{\omega}')$ from the interior is used. This radiance is in turn determined by interior radiances reflected and refracted by the interface (shown as dashed lines in orange).

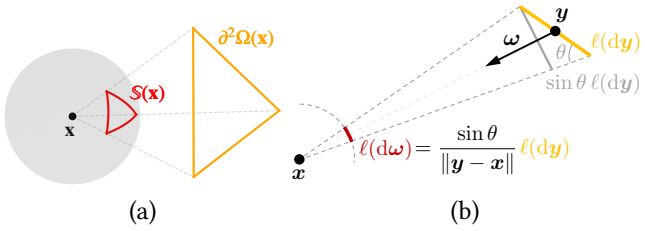


Fig. 9. (a) Definitions of $\mathbb{S}(\boldsymbol{x})$ and $\partial^2\Omega(\boldsymbol{x})$. (b) Deriving the change-of-measure ratio $\sin\theta/\|\boldsymbol{y}-\boldsymbol{x}\|$ in Eq. (36) by projecting a differential curve $\mathrm{d}\boldsymbol{y}$ to the surface of a unit sphere around $\boldsymbol{x}$.

can be further expressed as

$$L^{\text{ins}}(\boldsymbol{x}_0, \boldsymbol{\omega}) = \int_{\mathbb{H}_+} f_{\text{p}}(\boldsymbol{x}_0, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}_0, \boldsymbol{\omega}')\, \mathrm{d}\boldsymbol{\omega}' + \int_{\mathbb{H}_-} f_{\text{p}}(\boldsymbol{x}_0, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}_0, \boldsymbol{\omega}')\, \mathrm{d}\boldsymbol{\omega}', \tag{33}$$

where $\mathbb{H}_+$ and $\mathbb{H}_-$ are the two hemispheres separated by the boundary at $\boldsymbol{x}_0$ (see Figure 8) given by

$$\mathbb{H}_+ = \{\boldsymbol{\omega}' \in \mathbb{S}^2 \;:\; \langle \boldsymbol{n}(\boldsymbol{x}_0), \boldsymbol{\omega}' \rangle > 0\}, \qquad \mathbb{H}_- = \{\boldsymbol{\omega}' \in \mathbb{S}^2 \;:\; \langle \boldsymbol{n}(\boldsymbol{x}_0), \boldsymbol{\omega}' \rangle < 0\}, \tag{34}$$

where $\boldsymbol{n}(\boldsymbol{x}_0)$ is the boundary normal pointing toward the exterior of the medium (i.e., $\langle \boldsymbol{n}(\boldsymbol{x}_0), \boldsymbol{\omega} \rangle < 0$).

3.2.4 Differentiation of the in-Scattered radiance. Evaluating the RHS of Eq. (30) also requires the derivative of the in-scattered radiance L^{ins} of Eq. (??). Recall that L^{ins} is expressed as an integral of $f_{\text{p}}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}, \boldsymbol{\omega}')$ over directions $\boldsymbol{\omega}'$. Note that $L(\boldsymbol{x}, \boldsymbol{\omega}')$ may have discontinuities in $\boldsymbol{\omega}'$ (for fixed $\boldsymbol{x}$) due to visibility changes. Thus, to differentiate L^{ins}, we must consider how the discontinuities change with respect to the scene parameter π.

Let $\mathbb{S}(\boldsymbol{x}) \subset \mathbb{S}^2$ be a set of *spherical curves* capturing all discontinuities of $f_{\text{p}}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}, \boldsymbol{\omega}')$ with respect to $\boldsymbol{\omega}'$ (see the red curves in Figure 9). At an interior point $\boldsymbol{x} \in \Omega \setminus \partial\Omega$ and direction $\boldsymbol{\omega} \in \mathbb{S}^2$, the derivative of L^{ins} can be expressed as

$$\dot{L}^{\text{ins}}(\boldsymbol{x}, \boldsymbol{\omega}) = \int_{\mathbb{S}^2} \partial_\pi \left[f_{\text{p}}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}, \boldsymbol{\omega}') \right] \mathrm{d}\boldsymbol{\omega}' + \underbrace{\int_{\mathbb{S}(\boldsymbol{x})} \langle \boldsymbol{n}_\perp, \dot{\boldsymbol{\omega}}' \rangle\, f_{\text{p}}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, \Delta L(\boldsymbol{x}, \boldsymbol{\omega}')\, \mathrm{d}\ell(\boldsymbol{\omega}')}_{:= B^{\text{ins}}(\boldsymbol{x}, \boldsymbol{\omega})}. \tag{35}$$

The first term on the RHS of this equation is an integral over the unit sphere $\mathbb{S}^2$, which is independent of the scene parameter π, making the integral variable $\boldsymbol{\omega}'$ also be π-independent. To evaluate this term, the derivative of the phase function f_p is needed. In practice, f_p usually has analytical expressions, allowing $\dot{f}_p(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})$ to be obtained using symbolic differentiation.

The second term in Eq. (35) arises from the boundary integral term in Eq. (??) after Reynolds transport theorem (14) is applied. It emerges due to the discontinuities of $f_\mathrm{p}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}, \boldsymbol{\omega}')$ with respect to $\boldsymbol{\omega}'$, corresponding to how the discontinuities "move" as the scene parameter π varies. We denote this integral as $B^\mathrm{ins}(\boldsymbol{x}, \boldsymbol{\omega})$, wherein $\dot{\boldsymbol{\omega}}'$ is the change rate (with respect to π) of the discontinuity location $\boldsymbol{\omega}'$, and $\ell(\boldsymbol{\omega}')$ indicates the curve length measure.

In Eq. (35), $\boldsymbol{n}_\perp$ is a vector in the tangent space of $\mathbb{S}^2$ at $\boldsymbol{\omega}'$ perpendicular to the discontinuity curve (see Figure 6). Further, ΔL is the radiance difference across a discontinuity curve in $\mathbb{S}(\boldsymbol{x})$. We discuss both terms in more detail later in this subsection.

While B^ins integrates over a set of spherical curves on $\mathbb{S}^2$, it is computationally more convenient to rewrite this integral in terms of boundary curves in the 3D space. As shown in Figure 10, the 3D boundary curves comprise all the *geometric edges* that cause discontinuities of $L(\boldsymbol{x}, \boldsymbol{\omega}')$ in $\boldsymbol{\omega}'$ when viewed from $\boldsymbol{x}$, including (i) *boundary* edges associated with only one face; (ii) *silhouette* edges shared by a front-facing and a back-facing face; and (iii) front-facing *sharp* edges across which the surface normals are discontinuous.

Let $\partial^2\Omega(\boldsymbol{x}) \subset \partial\Omega$ denote all the boundary curves when viewed from $\boldsymbol{x}$. By changing the measure of curve length from $\mathbb{S}^2$ to the 3D Euclidean space, we rewrite the last term $B^\mathrm{ins}(\boldsymbol{x}, \boldsymbol{\omega})$ in Eq. (35) as

$$B^\mathrm{ins}(\boldsymbol{x}, \boldsymbol{\omega}) = \int_{\partial^2\Omega(\boldsymbol{x})} \left\langle \boldsymbol{n}_\perp, \partial_\pi(\boldsymbol{y} \to \boldsymbol{x}) \right\rangle f_\mathrm{p}(\boldsymbol{x}, \boldsymbol{x} \to \boldsymbol{y}, \boldsymbol{\omega}) \Delta L(\boldsymbol{x}, \boldsymbol{y} \to \boldsymbol{x})\, V(\boldsymbol{x}, \boldsymbol{y})\, \frac{\sin\theta}{\|\boldsymbol{y} - \boldsymbol{x}\|}\, \mathrm{d}\ell(\boldsymbol{y}), \tag{36}$$

where $\boldsymbol{y} \to \boldsymbol{x}$ indicates the normalized direction from $\boldsymbol{y}$ to $\boldsymbol{x}$, $\partial_\pi(\boldsymbol{y} \to \boldsymbol{x})$ is the same as $\dot{\boldsymbol{\omega}}'$, $V(\boldsymbol{x}, \boldsymbol{y})$ denotes the mutual visibility between $\boldsymbol{x}$ and $\boldsymbol{y}$, and θ is the angle between the tangent direction at $\boldsymbol{y}$ and $\boldsymbol{y} \to \boldsymbol{x}$ (see Figure 9-b). As a result, $\ell(\boldsymbol{y})$ here is the length measure in Euclidean space.

Lastly, we discuss how to compute the ingredients needed for evaluating Eq. (35), namely, $\boldsymbol{n}_\perp$, $\dot{\boldsymbol{\omega}}' = \partial_\pi(\boldsymbol{y} \to \boldsymbol{x})$, and ΔL.

Normal. The normal $\boldsymbol{n}_\perp$ in Eqs. (35) and (36) emerges from the application of Reynolds transport theorem (14) and represents the normal vector of the discontinuity boundary on the integration manifold (i.e., $\mathbb{S}^2$). Therefore, $\boldsymbol{n}_\perp$ must be in the tangent space of $\boldsymbol{\omega}' \in \mathbb{S}^2$.

In practice, when the scene geometry is depicted using polygonal meshes, the boundary curves $\partial^2\Omega(\boldsymbol{x})$ in Eq. (36) are comprised of polygonal face edges. Consider an edge with endpoints $\boldsymbol{p}$ and $\boldsymbol{q}$. Its projection on a unit sphere centered at $\boldsymbol{x}$ is an arc on which every point has the same normal in the tangent space (see Figure 6-a):

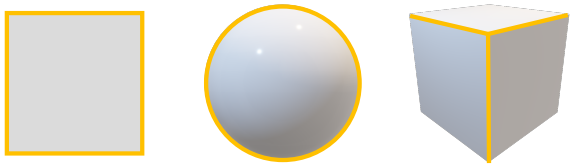
$$\boldsymbol{n}_\perp = \frac{(\boldsymbol{p} - \boldsymbol{x}) \times (\boldsymbol{q} - \boldsymbol{x})}{\|(\boldsymbol{p} - \boldsymbol{x}) \times (\boldsymbol{q} - \boldsymbol{x})\|}. \tag{37}$$

Change rate of $\boldsymbol{\omega}'$. Boundary curves in $\partial^2\Omega(\boldsymbol{x})$ may vary with respect to the scene parameter π—for example, an object may move in 3D along a trajectory parameterized by π. Suppose that a point $\boldsymbol{y} \in \partial^2\Omega(\boldsymbol{x})$ has a change rate $\dot{\boldsymbol{y}}$. Then, the corresponding direction $\boldsymbol{\omega}' = (\boldsymbol{y} \to \boldsymbol{x})$ has the derivative:

$$\dot{\boldsymbol{\omega}}' = \partial_\pi \left(\frac{\boldsymbol{x} - \boldsymbol{y}}{\|\boldsymbol{x} - \boldsymbol{y}\|} \right) = \frac{\dot{\boldsymbol{x}} - \dot{\boldsymbol{y}}}{\|\boldsymbol{x} - \boldsymbol{y}\|} - \boldsymbol{\omega}' \left\langle \boldsymbol{\omega}', \frac{\dot{\boldsymbol{x}} - \dot{\boldsymbol{y}}}{\|\boldsymbol{x} - \boldsymbol{y}\|} \right\rangle. \tag{38}$$

Evaluating ΔL. Recall that, in Eq. (35), $\Delta L(\boldsymbol{x}, \boldsymbol{\omega}')$ indicates the difference in radiance across discontinuity boundaries with respect to $\boldsymbol{\omega}'$. This is given by the difference between two one-sided limits of $\Delta L(\boldsymbol{x}, \hat{\boldsymbol{\omega}})$ when $\hat{\boldsymbol{\omega}}$ approaches the discontinuity boundary $\boldsymbol{\omega}'$ from both sides along the normal direction $\boldsymbol{n}_\perp$. Namely,

$$\Delta L(\boldsymbol{x}, \boldsymbol{\omega}') = \lim_{\epsilon \to 0^-} L(\boldsymbol{x}, \boldsymbol{\omega}' + \epsilon\, \boldsymbol{n}_\perp) - \lim_{\epsilon \to 0^+} L(\boldsymbol{x}, \boldsymbol{\omega}' + \epsilon\, \boldsymbol{n}_\perp). \tag{39}$$



(a) **Boundary** edges (b) **Silhouette** edges (c) **Sharp** edges

Fig. 10. Three types of edges (drawn in yellow) that can cause geometric discontinuities: (a) boundary, (b) silhouette, and (c) sharp.

3.2.5 Differentiation of the interfacial scattering operator. We now move on to the second term in Figure 7, the derivative of $\mathcal{K}_\mathrm{S} L$. Differentiating $(\mathcal{K}_\mathrm{S} L)(\boldsymbol{x}, \boldsymbol{\omega}) = T(\boldsymbol{x}_0, \boldsymbol{\omega})\, L_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega})$ yields

$$\begin{aligned}\left(\partial_\pi \mathcal{K}_\mathrm{S} L\right)(\boldsymbol{x}, \boldsymbol{\omega}) &= \partial_\pi [T(\boldsymbol{x}_0, \boldsymbol{x})\, L_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{x})] \\ &= T(\boldsymbol{x}_0, \boldsymbol{x})\, \dot{L}_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega}) + \dot{T}(\boldsymbol{x}_0, \boldsymbol{x})\, L_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega}) \\ &= T(\boldsymbol{x}_0, \boldsymbol{x}) \left[-\left(\Sigma_\mathrm{t}(\boldsymbol{x}, \boldsymbol{\omega}, D) + \dot{D}\, \sigma_\mathrm{t}(\boldsymbol{x}_0)\right) L_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega}) + \dot{L}_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega}) \right].\end{aligned} \tag{40}$$

The expression of $\dot{T}(\boldsymbol{x}_0, \boldsymbol{x})$ equals:

$$\dot{T}(\boldsymbol{x}_0, \boldsymbol{x}) = -T(\boldsymbol{x}_0, \boldsymbol{x}) \left(\Sigma_\mathrm{t}(\boldsymbol{x}, \boldsymbol{\omega}, D) + \dot{D}\, \sigma_\mathrm{t}(\boldsymbol{x}_0)\right), \tag{41}$$

where Σ_t follows the definition in Eq. (31) but with the distance variable τ replaced by D. The last term $\dot{D}\, \sigma_\mathrm{t}(\boldsymbol{x}_0)$ appears because the total travel distance D in the medium depends on the medium's boundary that might change under π.

The last component needed in Eq. (40) is $\dot{L}_\mathrm{r}(\boldsymbol{x}_0, \boldsymbol{\omega})$, whose specific form is analogous to $\dot{L}^\mathrm{ins}$ from Eq. (35) since both L^ins and L_r involve integrations over $\mathbb{S}^2$ (with the difference that L_r modulates incident radiance with the cosine-weighted BSDF f_s, while L^ins uses the phase function f_p). Thus, $\dot{L}_\mathrm{r}$ takes the form expressed in Eq. (42):

$$\begin{aligned}\dot{L}_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}) = &\int_{\mathbb{S}^2} \partial_\pi \left[f_\mathrm{s}(\boldsymbol{x}, -\boldsymbol{\omega}', \boldsymbol{\omega})\, L(\boldsymbol{x}, \boldsymbol{\omega}') \right] \mathrm{d}\boldsymbol{\omega}' + \\ &\int_{\partial^2\Omega(\boldsymbol{x})} \left\langle \boldsymbol{n}_\perp, \partial_\pi(\boldsymbol{y} \to \boldsymbol{x}) \right\rangle f_\mathrm{s}(\boldsymbol{x}, \boldsymbol{x} \to \boldsymbol{y}, \boldsymbol{\omega}) \Delta L(\boldsymbol{x}, \boldsymbol{y} \to \boldsymbol{x})\, V(\boldsymbol{x}, \boldsymbol{y})\, \tfrac{\sin\theta}{\|\boldsymbol{y}-\boldsymbol{x}\|}\, \mathrm{d}\ell(\boldsymbol{y}),\end{aligned} \tag{42}$$

where ΔL follows Eq. (39). Further, $\partial_\pi f_\mathrm{s}\, L$ in the first term depends on the derivative of the BSDF f_s which can generally be obtained using symbolic/automated differentiation.

3.2.6 Completing $\dot{L}$. The final term in Figure 7 is the derivative of $L^{(0)} = TL_\mathrm{e}$. Since Eq. (40) has already provided $\partial_\pi TL_\mathrm{r}$, $\dot{L}^{(0)}$ can be obtained by respectively replacing L_r and $\dot{L}_\mathrm{r}$ in Eq. (40) with L_e and $\dot{L}_\mathrm{e}$, yielding

$$\dot{L}^{(0)}(\boldsymbol{x}, \boldsymbol{\omega}) = \partial_\pi [T(\boldsymbol{x}_0, \boldsymbol{x})\, L_\mathrm{e}(\boldsymbol{x}_0, \boldsymbol{\omega})] = T(\boldsymbol{x}_0, \boldsymbol{x}) \left[-\left(\Sigma_\mathrm{t}(\boldsymbol{x}, \boldsymbol{\omega}, D) + \dot{D}\, \sigma_\mathrm{t}(\boldsymbol{x}_0)\right) L_\mathrm{e}(\boldsymbol{x}_0, \boldsymbol{\omega}) + \dot{L}_\mathrm{e}(\boldsymbol{x}_0, \boldsymbol{\omega}) \right]. \tag{43}$$

Similar to $\dot{f}_p$ and $\dot{f}_s$, $\dot{L}_\mathrm{e}(\boldsymbol{x}_0, \boldsymbol{\omega})$ can be obtained using symbolic differentiation.

Putting together. We now combine Eqs. (30, 40, 43) to get $\dot{L}$, the partial scene derivative of the interior radiance L.

Since $\dot{L} = \dot{L}_\mathrm{r} + \dot{L}_\mathrm{e}$, adding Eq. (40) and (43) effectively yields the derivative of the source term Q:

$$\dot{Q}(\boldsymbol{x}, \boldsymbol{\omega}) = \left(\partial_\pi \mathcal{K}_\mathrm{S} L + \dot{L}^{(0)}\right)(\boldsymbol{x}, \boldsymbol{\omega}) = T(\boldsymbol{x}_0, \boldsymbol{x}) \left[-\left(\Sigma_\mathrm{t}(\boldsymbol{x}, \boldsymbol{\omega}, D) + \dot{D}\, \sigma_\mathrm{t}(\boldsymbol{x}_0)\right) L(\boldsymbol{x}_0, \boldsymbol{\omega}) + \dot{L}(\boldsymbol{x}_0, \boldsymbol{\omega}) \right]. \tag{44}$$

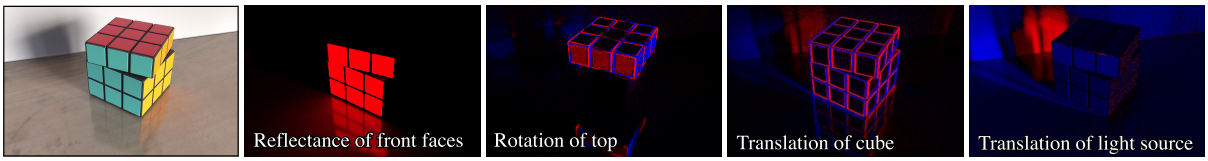
Adding Eq. (30) and (44) completes the derivation of $\dot{L}$, as shown in Eq. (??) below.

$$\begin{aligned}[L(\boldsymbol{x},\boldsymbol{\omega})]^{\bullet} = &\int_0^D T(\boldsymbol{x}',\boldsymbol{x})\big[\sigma_{\mathrm{s}}(\boldsymbol{x}')\,\dot{L}^{\mathrm{ins}}(\boldsymbol{x}',\boldsymbol{\omega}) + (\dot{\sigma}_{\mathrm{s}}(\boldsymbol{x}') - \Sigma_{\mathrm{t}}(\boldsymbol{x},\boldsymbol{\omega},\tau)\,\sigma_{\mathrm{s}}(\boldsymbol{x}'))\,L^{\mathrm{ins}}(\boldsymbol{x}',\boldsymbol{\omega})\big]\,\mathrm{d}\tau \\ &+ T(\boldsymbol{x}_0,\boldsymbol{x})\big[-\big(\Sigma_{\mathrm{t}}(\boldsymbol{x},\boldsymbol{\omega},D) + \dot{D}\,\sigma_{\mathrm{t}}(\boldsymbol{x}_0)\big)\,L(\boldsymbol{x}_0,\boldsymbol{\omega}) + \dot{L}(\boldsymbol{x}_0,\boldsymbol{\omega}) + \dot{D}\,\sigma_{\mathrm{s}}(\boldsymbol{x}_0)\,L^{\mathrm{ins}}(\boldsymbol{x}_0,\boldsymbol{\omega})\big].\end{aligned} \tag{45}$$

3.3 Radiative Backpropagation

The preceding two sections discussed various challenges that arise during the differentiation of light transport integrals containing discontinuities. We now turn to another important aspect that distinguishes differentiable from "ordinary" rendering: the parameter space $\mathcal{X}$ tends to be high-dimensional ($10^6 - 10^9$ D), and the same is also true for the rendered image $\mathbf{y} \in \mathcal{Y}$. This implies that the Jacobian matrix $\mathbf{J}_f = \partial f / \partial \mathbf{x}$ of partial derivatives of f is extremely large—e.g. ~3.7 terabytes of memory in single precision if $\dim \mathcal{X} = \dim \mathcal{Y} = 10^6$, making its explicit storage or computation impractical. The Jacobian also lacks the sparseness or low-rank structure that are common requirements of algorithms that can efficiently operate on large matrices.

Although working with $\mathbf{J}_f$ directly is hopeless, matrix-vector products involving $\mathbf{J}_f$ can be computed very efficiently, as we will see in this section. Two different types of matrix-vector products are used by current differentiable renderers: *forward-mode* methods evaluate $\boldsymbol{\delta}_{\mathbf{y}} = \mathbf{J}_f\,\boldsymbol{\delta}_{\mathbf{x}}$ to determine how an infinitesimal perturbation $\boldsymbol{\delta}_{\mathbf{x}}$ of the scene parameters changes the output image. In the simplest case, $\boldsymbol{\delta}_{\mathbf{x}}$ is a "one-hot" vector, which will extract a single column of $\mathbf{J}_f$ and thus visualize how the rendered image will change when one of the scene parameters is perturbed, as shown below for a Rubik's cube.



For example, the second image shows that changing the color of the front faces of the Rubik's cube will lead to a corresponding change in the rendered image (where red indicates a positive change and blue indicates negative changes). Also, observe how the indirect nature of light (inter-reflection) leads to changes in the reflection on the shiny table. Forward-mode differentiation can be handy to visualize and debug differentiable rendering algorithms, but it is a poor fit for gradient-based optimization of scenes with many parameters, since its runtime cost grows linearly with the number of scene parameters for which derivatives are sought.

In contrast, *reverse-mode* approaches rely on a matrix-vector product involving the *transpose* of the Jacobian, i.e., $\boldsymbol{\delta}_{\mathbf{x}} = \mathbf{J}_f^T\,\boldsymbol{\delta}_{\mathbf{y}}$, which captures how the scene parameters should be updated to realize a desired perturbation $\boldsymbol{\delta}_{\mathbf{y}}$ of the output image. This is an ideal match for gradient-based optimization involving the composition a rendering algorithm and an objective function. Together with the chain rule, it enables splitting the differentiation of $f(g(\mathbf{x}))$ into three conceptual steps:

(1) Running an ordinary rendering algorithm to obtain $\mathbf{y} = f(\mathbf{x})$.

(2) Differentiating the objective function in reverse mode to obtain $\boldsymbol{\delta}_{\mathbf{y}} = \mathbf{J}_g^T(\mathbf{y}) \cdot 1$. The parentheses indicate that the derivative of the objective is evaluated at $\mathbf{y}$, and the right hand side of the matrix-vector product is trivial as g is scalar-valued. This operation produces a vector with one element per pixel of the output image, indicating how the image should be changed to first order.

    Variables containing derivatives with respect to the optimization objective are denoted *adjoint variables* in the context of reverse-mode differentiation, and for this reason, we will call $\boldsymbol{\delta}_{\mathbf{y}} \in \mathbb{R}^n$ the *adjoint rendering*. Note the term *adjoint* has a variety of other meanings, e.g. to characterize equations and algorithms that build on

the reciprocal nature of light transport and scattering. As we will see shortly, these two interpretations are closely related when derivatives are at play. .

(3) Finally, the last step entails differentiating the rendering algorithm in reverse mode to obtain $\boldsymbol{\delta}_{\mathbf{x}} = \mathbf{J}_f^T(\mathbf{x}) \cdot \boldsymbol{\delta}_{\mathbf{y}}$. In the following, we will only focus on this part, since it requires new methods specific to differentiable rendering and thus represents the main source of difficulty.

Differentiation of numerical algorithms is commonly performed out using tools for *automatic differentiation* (AD), which Section 5.2 will discuss in more detail. In particular, forward and reverse mode are standard choices within a large design space supported by modern AD tools. However, simply applying reverse-mode AD to an entire rendering algorithm is unlikely to yield satisfactory results due to the vast amount of arithmetic with incoherent control flow performed by physics-based rendering techniques.

Reverse-mode AD hinges on the ability to replay the execution of a program in reverse, which requires maintenance of some kind of data structure (computation graph, queue, etc.) enabling this reversal. For non-trivial rendering tasks involving global inter-reflection, this data structure becomes staggeringly large, exceeding the available system memory by many orders of magnitude. Simple scenes rendered at low resolutions and sample counts can fit, though the memory traffic to maintain this data structure is often a severe performance bottleneck even in such cases.

This section presents the high-level ideas underlying *radiative backpropagation* (RBP) [31], which is an alternative approach to differentiable rendering that addresses this limitation. Its main insight is that reverse-mode differentiation of a rendering algorithm has a physical interpretation where "derivative radiation" is "emitted" by sensors, "scattered" by the scene, and eventually "received" by objects with differentiable parameters. In this section, we will derive this analogy in a simplified setting.

Radiative backpropagation solves this modified light transport problem to fully decouple steps 1 and 3 of the previous list, removing the need for an intermediate data structure for program reversal. This enables differentiable simulation of complex scenes (e.g. long-running simulations involving many parameters) with greatly improved performance: speedups of up to a factor of 1000× were observed by Nimier-David et al. in a comparison to reverse-mode AD.

3.4 Primal equations

To keep the discussion short, we will focus on the core idea without worrying about discontinuities, moving cameras, and participating media—the resulting method will thus compute shading and illumination-related derivatives only. For completeness, we briefly recapitulate the three equations that fully define the original (non-differentiated) light transport problem: the first is the measurement equation, which expresses the value of pixel measurements I_k via a ray-space integral involving the pixel's importance function W_k and the incident radiance L_i [41]:

$$I_k = \int_{\mathcal{A}} \int_{S^2} W_k(\mathbf{p}, \boldsymbol{\omega})\, L_i(\mathbf{p}, \boldsymbol{\omega})\, \mathrm{d}\boldsymbol{\omega}^{\perp}\, \mathrm{d}\mathbf{p}.$$

Such ray-space product integrals can also be interpreted as a type of inner product, simplifying the notation:

$$=: \langle W_k, L_i \rangle. \tag{46}$$

Next, the transport equation relates incident and outgoing radiance using the *ray tracing* function $\mathbf{r}(\mathbf{p}, \boldsymbol{\omega})$, which returns the nearest surface visible along the ray $(\mathbf{p}, \boldsymbol{\omega})$:

$$L_i(\mathbf{p}, \boldsymbol{\omega}) = L_o(\mathbf{r}(\mathbf{p}, \boldsymbol{\omega}), -\boldsymbol{\omega}). \tag{47}$$

Finally, the rendering equation goes in the opposite direction and relates outgoing to incident radiance via the surface's emission and BSDF:

$$L_o(\mathbf{p}, \boldsymbol{\omega}) = L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_{S^2} L_i(\mathbf{p}, \boldsymbol{\omega}')\, f_s(\mathbf{p}, \boldsymbol{\omega}, \boldsymbol{\omega}')\, \mathrm{d}\boldsymbol{\omega}'^{\perp}. \tag{48}$$

Scattering and transport both can be shown to be linear operators (i.e. an infinite-dimensional generalization of a matrix) [3, 41], hence the above two equations are often abbreviated as

$$L_i = \mathcal{K}_\mathrm{T} L_o, \quad L_o = L_e + \mathcal{K}_\mathrm{S} L_i,$$

where $\mathcal{K}_\mathrm{T}$ and $\mathcal{K}_\mathrm{S}$ and encode (47) and (48), respectively. Substituting both operators and solving for L_o yields the *solution operator* $\mathcal{K}_\mathrm{R}$, which can be expressed in terms of a Neumann series expansion.

$$\begin{aligned} L_o &= L_e + \mathcal{K}_\mathrm{S}\mathcal{K}_\mathrm{T} L_o \\ &= \underbrace{(I - \mathcal{K}_\mathrm{S}\mathcal{K}_\mathrm{T})^{-1}}_{=:\mathcal{K}_\mathrm{R}} L_e = \sum_{i=0}^{\infty} (\mathcal{K}_\mathrm{S}\mathcal{K}_\mathrm{T})^i L_e, \end{aligned}$$

where I is the identity. Substituting these relations into the measurement equation (46) transforms it into

$$\begin{aligned} I_k &= \langle W_k, L_i \rangle \\ &= \langle W_k, \mathcal{K}_\mathrm{T} L_o \rangle \\ &= \langle W_k, \mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R} L_e \rangle. \end{aligned}$$

Veach also showed that $\mathcal{K}_\mathrm{S}$, $\mathcal{K}_\mathrm{T}$, and $\mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R}$ are all *self-adjoint* linear operators (in the sense of functional analysis) when the scene satisfies elementary physical constraints—in particular, energy-conserving and reciprocal BSDFs. Self-adjoint operators $\mathcal{K}$ have the property that $\langle \mathcal{K} v_1, v_2 \rangle = \langle v_1, \mathcal{K} v_2 \rangle$, which implies that

$$I_k = \langle W_k, \mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R} L_e \rangle. = \langle \mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R} W_k, L_e \rangle.$$

This equation is the foundation of algorithms such as bidirectional path tracing: it states that instead of transporting radiance to the sensor, we can also go the other way and transport importance W_k to the light sources and perform the ray-space product integral there. The main appeal of this equivalence is that it exposes alternative sampling strategies that often achieve a lower amount of variance compared to standard path tracing.

3.5 Differential equations

We now turn again to differentiable rendering: by differentiating the left and right-hand side of the measurement, transport, and rendering equations, we obtain a simple differential theory similar to what was previously discussed in Section 3.1 and 3.2. In contrast to these sections, our focus will be on obtaining a formulation enabling reverse-mode differentiation.

To start, a differential measurement of pixel k involves a ray-space inner product involving *differential incident radiance* $\partial_\mathbf{x} L_i$ and importance W_k (where we have assumed a static sensor and the short-hand notation $\partial_\mathbf{x}$ refers to a gradient with respect to all scene parameters).

$$\partial_\mathbf{x} I_k = \int_{\mathcal{A}} \int_{S^2} W_k(\mathbf{p}, \boldsymbol{\omega})\, \partial_\mathbf{x} L_i(\mathbf{p}, \boldsymbol{\omega})\, \mathrm{d}\boldsymbol{\omega}^{\perp}\, \mathrm{d}\mathbf{p}. \tag{49}$$

We shall think of differential radiance as simply another type of radiance that can be emitted, scattered, and received by sensors. Differentiating the transport equation yields no surprises: differential radiance is transported in the same manner as normal radiance:

$$\partial_\mathbf{x} L_i(\mathbf{p}, \boldsymbol{\omega}) = \partial_\mathbf{x} L_o(\mathbf{r}(\mathbf{p}, \boldsymbol{\omega}), -\boldsymbol{\omega}). \tag{50}$$

The derivative of the rendering equation is interesting:

$$\partial_{\mathbf{x}} L_o(\mathbf{p},\boldsymbol{\omega}) = \underbrace{\partial_{\mathbf{x}} L_e(\mathbf{p},\boldsymbol{\omega})}_{\textbf{Term 1}} + \int_{S^2} \Big[\underbrace{\partial_{\mathbf{x}} L_i(\mathbf{p},\boldsymbol{\omega}')\, f_s(\mathbf{p},\boldsymbol{\omega},\boldsymbol{\omega}')}_{\textbf{Term 2}} + \underbrace{L_i(\mathbf{p},\boldsymbol{\omega}')\, \partial_{\mathbf{x}} f_s(\mathbf{p},\boldsymbol{\omega},\boldsymbol{\omega}')}_{\textbf{Term 3}} \Big] \, \mathrm{d}\boldsymbol{\omega}'^{\perp} \tag{51}$$

The above can be interpreted as another kind of energy balance equation. In particular,

- **Term 1.** Differential radiance is "emitted" by light sources with differentiable parameters.
- **Term 2.** Differential radiance incident on a surface "scatters" just like normal radiance (according to the BSDF).
- **Term 3.** Surfaces with differentiable parameters convert ordinary radiance into differentiable radiance. Terms 1 and 3 together constitute the two causes of "differential emission".

Based on this observation, we separate terms 1 and 3 into a new differential emission term

$$\mathbf{Q}(\mathbf{x},\omega) = \partial_{\mathbf{x}} L_e(\mathbf{p},\boldsymbol{\omega}) + \int_{S^2} L_i(\mathbf{p},\boldsymbol{\omega}')\, \partial_{\mathbf{x}} f_s(\mathbf{p},\boldsymbol{\omega},\boldsymbol{\omega}')\, \mathrm{d}\boldsymbol{\omega}'^{\perp},$$

enabling (51) to be stated as a standard rendering equation involving $\mathbf{Q}$:

$$\partial_{\mathbf{x}} L_o = \mathbf{Q} + \mathcal{K}_{\mathrm{S}} \partial_{\mathbf{x}} L_i.$$

Note the boldface term $\mathbf{Q}$: this is a vector-valued equation specifying a separate light transport problem per differentiable parameter. Another important difference of $\mathbf{Q}$ compared to an ordinary emission term is that it can take on negative values. Other than that, this interpretation involves the same equations and operators, making it possible to directly write down the solution in terms of the standard solution operator $\mathcal{K}_{\mathrm{R}}$.

$$\partial_{\mathbf{x}} L_o = \mathbf{Q} + \mathcal{K}_{\mathrm{S}} \mathcal{K}_{\mathrm{T}} \partial_{\mathbf{x}} L_o = \mathcal{K}_{\mathrm{R}} \mathbf{Q}. \tag{52}$$

This process generalizes to more complex settings (differentiable participating media and geometry), in which case $\mathbf{Q}$ contains additional terms. Let's now see how all of this is useful.

3.6 Putting things together

Recall our goal of evaluating $\mathbf{J}_f^T \boldsymbol{\delta}_y$ efficiently as part of step (3) on page 20. We also saw in the visualizations on page 19 how each column of the Jacobian records a "gradient image" with respect to a scene parameter x_i. Similarly, each row (or column of the transpose) holds derivatives of a single pixel measurement $I_0, \ldots, I_n$ with respect to all scene parameters $\mathbf{x}$:

$$\mathbf{J}_f^T = \left[\partial_{\mathbf{x}} I_0, \ldots, \partial_{\mathbf{x}} I_n \right]. \tag{53}$$

The matrix-vector multiplication $\mathbf{J}_f^T \boldsymbol{\delta}_y$ then yields

$$\mathbf{J}_f^T \boldsymbol{\delta}_{\mathrm{y}} = \sum_{k=1}^{n} \boldsymbol{\delta}_{\mathrm{y},k}\, \partial_{\mathbf{x}} I_k$$

where $\boldsymbol{\delta}_{\mathrm{y},k}$ is the k-th pixel of $\boldsymbol{\delta}_{\mathrm{y}}$. Next, we expand $\partial_{\mathbf{x}} I_k$ using the differential measurement equation (49) and regroup:

$$= \int_{\mathcal{A}} \int_{S^2} \underbrace{\left[\sum_{k=1}^{n} \boldsymbol{\delta}_{\mathrm{y},k} W_k(\mathbf{p},\boldsymbol{\omega}) \right]}_{=: A_e(\mathbf{p},\boldsymbol{\omega})} \partial_{\mathbf{x}} L_i(\mathbf{p},\boldsymbol{\omega})\, \mathrm{d}\boldsymbol{\omega}^{\perp}\, \mathrm{d}\mathbf{p},$$

where we have defined the *emitted adjoint radiance* $A_e(\mathbf{p},\boldsymbol{\omega})$. The reason for this naming convention is that the original rendering records radiance values, and A_e encodes how this radiance should change to improve the objective function.

Just like importance, we also shall think of adjoint radiance A_e as an emitted quantity—for instance, in the case of a pinhole camera, it can be interpreted as a textured "spot light" that projects the adjoint rendering into the scene. With these simplifications, the desired gradient turns into simply another inner product on ray space:

$$\mathbf{J}_f^T \boldsymbol{\delta}_\mathbf{y} = \langle A_e, \partial_\mathbf{x} L_i \rangle$$

which can be expressed directly in terms of the emission $\mathbf{Q}$ via the solution operator (52)

$$= \langle A_e, \mathcal{K}_\mathrm{T} \mathcal{K}_\mathrm{R} \mathbf{Q} \rangle.$$

At this point, we refer back to Section 3.4, whose end highlights the self-adjointness of $\mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R}$. Once more, we can exploit the reciprocal nature of the physics of light to propagate differential radiation "from the other end", i.e., starting with adjoint radiance at the sensor, and scattering and transporting it towards scene positions $\mathbf{p}$ with differentiable parameters (i.e. nonzero $\mathbf{Q}(\mathbf{p}, \cdot)$):

$$\mathbf{J}_f^T \boldsymbol{\delta}_\mathbf{y} = \langle \mathcal{K}_\mathrm{T} \mathcal{K}_\mathrm{R} A_e, \mathbf{Q} \rangle. \tag{54}$$

The impact of this change is profound and goes far beyond simple variance improvements as was the case for bidirectional path tracing. This is due to the dimension of the involved quantities: A_e associates an intensity value with rays passing through the sensor's aperture, and solving the associated transport problem (i.e. $\mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R} A_e$) is just like rendering an ordinary scene (i.e. "easy").

In contrast, $\mathbf{Q}$ is a vector-valued function with millions to billions of entries—one for each differentiable scene parameter. Evaluating $\mathcal{K}_\mathrm{T}\mathcal{K}_\mathrm{R}\mathbf{Q}$ would entail the solution of that many separate transport problems, which is clearly impractical.

4 PSEUDOCODE AND DISCUSSION

A simple path tracing-style implementation based on these insights has the following structure:

```
def rbp(p, omega, adjoint_radiance):
    # 1. Find intersection with the closest surface
    p' = ray_intersect(p, omega)

    # 2. Accumulate scene parameter gradients
    grad += Q(p, -omega) * adjoint_radiance

    # 3. Sample the surface's BSDF
    omega', weight = sample_bsdf(p', -omega)

    # 4. Recurse
    rbp(p', omega', adjoint_radiance * weight)
```

This algorithm would be invoked with directions (p, ω) sampled from the sensor's importance function (e.g. uniform in image space for orthographic or pinhole camera models), and the third argument is initialized with the corresponding value from the adjoint rendering $\boldsymbol{\delta}_y$. Alternatively, one could also directly importance sample the pixels of $\boldsymbol{\delta}_y$ proportional to their value as an additional variance reduction strategy.

Step 2 is a critical component, which evaluates the differential emission $\mathbf{Q}$, weights the result by the adjoint radiance propagated along the current sub-path, and accumulates the product into a global gradient vector `grad`. A property of $\mathbf{Q}$ with significant performance implications is its sparsity: at a given surface location $\mathbf{p}$, it will normally only depend on a few parameters including shading model parameters or nearby texels influencing the appearance at $\mathbf{p}$. In practice, this step will therefore perform highly localized updates to components instead of modifying

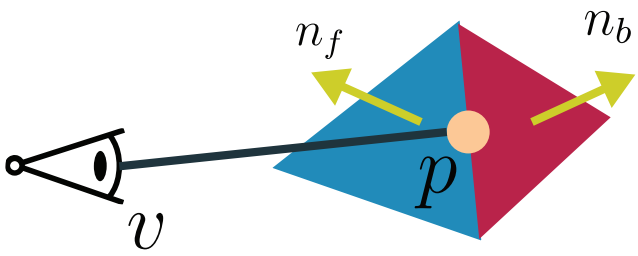


Fig. 11. Silhouette edges are the main cause of the discontinuities in rendering. Given a viewpoint v and an edge associated with two faces, the edge is a silhouette if for any point p on it, the vector $p - v$ is facing towards different directions with respect to the two normals, that is, sign $\left(\langle p - v, n_f \rangle\right) \neq$ sign $(\langle p - v, n_b \rangle)$.

the high-dimensional vector `grad` directly. We note that this section omitted several other performance-critical aspects of radiative backpropagation to keep the discussion self-contained and refer to the original paper [31] for further details.

The function $\mathbf{Q}$ contains an integral that requires a Monte Carlo sampling strategy— in practice, the sampled direction from step 3 can be re-used here, and direct illumination strategies (optionally with Multiple Importance Sampling) are also of use to further reduce variance. $\mathbf{Q}$, including this sampling step, can be derived by hand, though this tedious when the scene contains advanced shading models and emitters. AD systems based on static or dynamic (JIT-based) program transformation are likely preferable in practice. Please see Section 5.2 for a thorough overview of AD techniques.

Although the high-level structure of the radiative backpropagation superficially resembles path tracing, a major difference becomes apparent at the implementation level: physics-based rendering algorithms are typically dominated by *gather* operations that perform read-only memory accesses to scene data, while a comparably small number of write operations targets pixels in the output image. In radiative backpropagation, everything is reversed: the method reads from an image and is dominated by *scatter* operations that update gradients associated with different parts of the scene.

Radiative Backpropagation resembles the *adjoint sensitivity method* [36] from the area of optimal control. Here, an ordinary differential equation is integrated up to a certain time in the primal phase. The adjoint phase then solves the same equation once more *backwards in time* to propagate sensitivities with respect to an optimization objective to parameters of the model. The methods are similar mainly in spirit, while the details vary considerably: the adjoint sensitivity method is an approach for ordinary differential equations with a reversible time dimension, while rendering typically involves steady-state simulations (i.e. lacking a natural time dimension) involving integral- and integro-differential equations.

4.1 Path-Space Differentiable Rendering

TBD.

5 PHYSICS-BASED DIFFERENTIABLE RENDERING IMPLEMENTATION

5.1 Monte Carlo Estimators

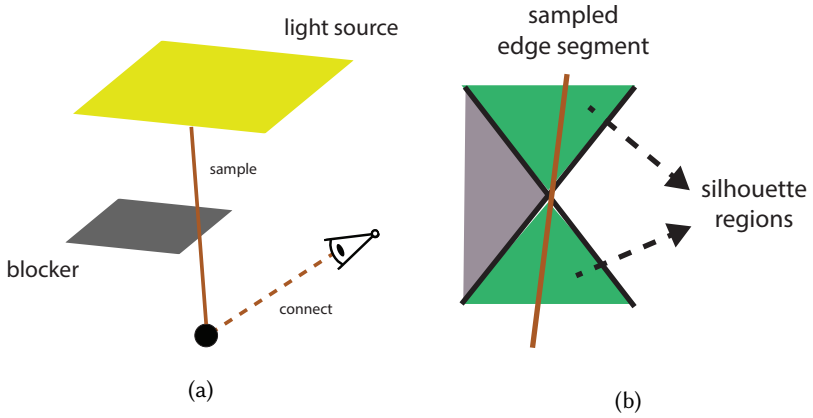


Fig. 12. Zhang et al. [44] proposed an alternative edge sampling algorithm by sampling a *path segment* from an edge directly. Given an edge, such as an edge associated with the blocker in (a), they sample a segment by randomly selecting a point and a direction, and connect the segments to light sources and cameras. To efficiently select a direction from the selected edge, we can use the fact that only a few set of directions will see the edge as a silhouette edge (the green region in b).

5.1.1 Differentiable Path Tracing with Edge Sampling. As mentioned in the previous sections, we can discretize the boundary integral by sampling points on triangle edges and evaluate the rendering contribution at the two sides of the edge. A naïve approach is to collect all the edges on the meshes, uniformly select one, and then uniformly select a point on the edge. This works for simple scenes such as the one in Figure 2 but generates large variance for more complex scenes. We want to select the edges and points in a smarter way. We will discuss boundaries integral for the surfaces directly visible to the camera first, then we will discuss boundaries for shadow and global illumination.

One of the key observation to speedup the sampling of edges is to notice that, if our shading is smooth over the mesh, the only edges that can be the boundaries are the *silhouette* edges (Fig. 11). For the edges associated with two faces, we classify them as non-silhouette if the normals of the two faces face towards the same sides, when comparing to the vector connecting between an arbitrary point on the edge p and the viewpoint v. That is, if the two associated faces have normals n_f and n_b, the edge is a silhouette edge if and only if

$$\text{sign}\left(\langle p - v, n_f \rangle\right) \neq \text{sign}\left(\langle p - v, n_b \rangle\right). \tag{55}$$

Edges that associate with only one face is always a silhouette edge, and edges associate with three or more faces can be converted into multiple edges with two faces. The majority of the edges of a mesh with respect to a viewpoint are not silhouettes, as the number of silhouettes is roughly at the complexity of $\sqrt{N}$ with N being the number of edges [27].

For directly visible surfaces, given a set of edges of a mesh and the camera position, we can classify them into silhouette and non-silhouette edges, and then project all the silhouette edges to the screen. The classification and projection can be done at a precomputation pass before rendering. During rendering, we then randomly pick one edge with probability proportional to their length in the screen space, and uniformly pick one point on the edge to evaluate the contribution at the two sides (Eq. 16). This is fast and does not require any sophisticated data structures. To further improve the edge selection for scenes with high depth complexity, we can potentially incorporate more advanced occlusion culling algorithms to skip the edges [11, 12]. Finally, this can also be modified to work with non-pinhole projective cameras: we can compute a bounding box for the aperture of the

camera, and conservatively classify the edges as silhouette as long as one viewpoint in the bounding box v can view the edge as a silhouette.

For the secondary visibility such as shadow, mirror reflection, and global illumination, determining silhouettes for our shading points becomes more involved. In contrast to the directly visible surfaces case, now the viewpoint v is our shading point and can be anywhere in the scene. Sampling an edge in this case becomes a problem similar to the *many-lights* sampling problem [34, 43]. In many-lights sampling, given a shading point and a set of light sources, we want to sample the light sources to compute the lighting at the shading point. In contrast, now given a shading point and a set of geometric edges, we want to sample the edges to compute the boundary integral. In the many-lights sampling literature, a common way (but not the only way) to sample the light sources is to build a tree data structure to select important light sources while traversing the tree: for each node in the tree, we can collect some summary statistics of the light sources, such as their total emission power, spatial bounding box, ranges of their orientations, etc, and use that to determine the importance of the tree node. We can use the same strategy for sampling edges as well, while rejecting nodes containing only non-silhouettes given a shading point. Li et al. [23] proposed to detect the silhouette in the Hough space [33] and we refer to Li's thesis for the details regarding this approach.

Li's edge sampling approach [23, 24], while being reasonably efficient for directly visible surfaces, is not extremely efficient for secondary visibility when the scenes complexity is high. An alternative proposed by Zhang et al. [44] is to sample a path *segment* from an edge, and connect that path segment to cameras and light sources (Fig. 12). When selecting the directions for the path segment, we can make sure the edge is always a silhouette edge by sampling the directions where all points on that directions will recognize the edge as a silhouette (Fig. 12b). Zhang et al. further improve this by selecting the edge using a hash-grid where the importance of each grid is estimated using photon mapping [19]. This approach simplifies the edge selection process and thus makes it significantly more efficient, since it does not need to select an edge for each shading point. See their paper for more details.

Selecting an edge efficiently for solving the boundary integral is still an unsovled issue. Zhang et al. [44]'s approach, while being shown to be efficient for a variety of scenes, still left some cases unhandled. For example, it is difficult to connect the edge segment to the light sources and cameras when there are highly-specular surfaces, such as mirror or glass, between the segment's endpoints and the target. This connection problem resembles the notorious Specular-Diffuse-Specular light path sampling problem in forward rendering [41]. More future research is required to solve the full differentiable rendering equation efficiently.

5.1.2 *Reparameterized Differentiable Path Tracing.*

One potential concern regarding edge sampling strategies for differentiable rendering is the complexity of finding contributing silhouette edges in an arbitrary integrand. Currently used strategies tend to find high-contribution edge segments with an insufficient density, which leads to gradients with high variance. This issue grows in severity as the geometric complexity of objects in the scene increases.

Interestingly, explicit sampling of discontinuities can be avoided by applying carefully chosen changes of variables that remove the dependence of discontinuities on scene parameters. The resulting integrands of course still contain discontinuities, but they are *static* in the re-parameterized integral and hence no longer prevent differentiation under the integral sign. **This section discusses an alternative way of differentiating edges by performing a suitable change of variables in each integral that is simple to compute using ordinary ray tracing operations.**

5.1.3 *Differentiable Volume Path Tracing.*

5.2 Automatic Differentiation for Rendering

A substantial amount of the course has been focused on the discontinuities of the rendering integrand, and how to correctly take them into account. However, most differentiable rendering systems will also need to compute the interior integrand. For the interior integrands, the discretization commute with the differentiation, so we can apply existing technique of automatic differentiation [15] to the rendering programs to compute the derivatives.

Widely used deep learning frameworks, such as PyTorch [35] and TensorFlow [1] provide a convenient interface to array-based computation on GPUs with built-in automatic differentiation. These frameworks are designed for linear algebra or neural networks, whose computation graphs typically consist of a few hundred arithmetically intensive operations like matrix-vector multiplications or convolutions. On the flipside, computation graphs produced by rendering algorithms tend to be highly unstructured and extremely large, containing millions of operations with very low arithmetic intensity (e.g. additions). Implementing a rendering pipeline inside a deep learning framework will be extremely inefficient since data are repeated stored to and read back from the memory buffers. Therefore, we need new automatic differentiation compilers for efficient differentiable rendering.

In this section, we will review how automatic differentiation works, so that the readers can potentially build their own compilers for differentiating their programs, or manually apply the automatic differentiation rules to transform their programs. Most of the contents below in the subsection are adapted from Chapter 2 of Li's thesis [23]. Griewank [15] provide a good introductory text to automatic differentiation. Villard and Monagan's work on Maple's automatic differentiation compiler [42] is a well written example for how to implement a compiler for automatic differentiating imperative programs.

Given a computer program containing control flow, loops, and/or recursion, with some real number inputs and outputs, our goal is to compute the derivatives between the outputs and the inputs. In particular, in differentiable rendering we are especially interested in the gradients, that is, the derivative of a scalar output with respect to the inputs.

5.2.1 Finite differences and symbolic derivatives. Before discussing automatic differentiation algorithms, it is useful to review other ways of generating derivatives, and compare them to automatic differentiation.

A common approximation for derivatives are finite differences, sometimes also called numerical derivatives. Given a function $f(x)$ and an input x, we approximate the derivative by perturbing x by a small amount h:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} \quad \text{or} \tag{56}$$

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{57}$$

The problem with this approximation is two-fold. First, the optimal choice of the step size h in a computer system is problem dependent. If the step size is too small, the rounding error of the floating point representation becomes too large. On the other hand, if the step size is too large, the result becomes a poor approximation to the true derivative. Second, the method is inefficient for multivariate functions. For a function with 100 variables and a scalar output, computing the full gradient vector requires at least 101 evaluations of the original function.

Another alternative is to treat the content of the function f as a sequence of mathematical operations, and symbolically differentiate the function. Indeed, most of the rules for differentiation are mechanical, and we can apply the rules to generate $f'(x)$. However, in our case, $f(x)$ is usually an *algorithm*, and symbolic differentiation does not scale well with the number of symbols. Consider the following code:

Using the symbolic differentiation tools from mathematical software such as Mathematica [17] would result in the following expression:

$$\frac{df(x)}{dx} = e^{x+e^{e^{e^{e^{e^{e^x}}}}}} + e^{e^{e^{e^{e^{e^x}}}}} + e^{e^{e^{e^{e^x}}}} + e^{e^{e^{e^x}}} + e^{e^{e^x}} + e^{e^x} + e^x. \tag{58}$$

```
function f(x):
    result = x
    for i = 1 to 8:
        result = exp(result)
    return result
```

Fig. 13. A code example that iteratively computes a nested exponential for demonstrating the difference between symbolic differentiation and automatic differentiation.

The size of derivative expression will become intractable when the size of the loop grows much larger. Using forward-mode automatic differentiation, which will be introduced later, we can generate the following code for computing derivatives:

```
function d_f(x):
    result = x
    d_result = 1
    for i = 1 to 8:
        result = exp(result)
        d_result = d_result * result
    return d_result
```

The code above outputs the exact same values as the symbolic derivative (Equation 58), but is significantly more efficient (8 v.s. 37 exponentials). This is due to automatic differentiation's better use of the intermediate values and the careful factorization of common subexpressions.

5.2.2 Algorithms for generating derivatives. We start from programs with only function calls and elementary operations such as addition and multiplication. In particular, we do not allow recursive or circular function calls. Later, we generalize the idea to handle control flow such as loops and branches, and handle recursion. Throughout the chapter, we assume all function calls are side-effect free. To the author's knowledge, there are no known automatic differentiation algorithms for transforming arbitrary functions with side effects.

The key to automatic differentiation is the chain rule. Consider the following code with input `x` and output `z`:

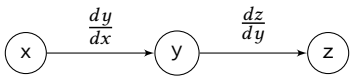
```
y = f(x)
z = g(y)
```

Assume we already know the derivative functions $\frac{df(x)}{dx}$ and $\frac{dg(y)}{dy}$, and we are interested in the derivative of the output `z` with respect to input `x`. We can compute the derivative by applying the chain rule:

```
dydx = dfdx(x)
dzdy = dgdy(y)
dzdx = dzdy * dydx
```

We can recursively apply the rule to generate derivative functions, until the function is an elementary function for which we know the analytical derivatives, such as addition, multiplication, `sin()`, or `exp()`.

A useful mental model for automatic differentiation is the *computational graph*. It can be used for representing dependencies between variables. The nodes of the graph are the variables and the edges are the derivatives between the adjacent vertices. In the case above the graph is linear:



Computing derivatives from a computational graph involves traversal of the graph, and gathering of different paths that connect inputs and outputs.

In practice, most functions are multivariate, and often times we want to have multiple derivatives such as for the gradient vector. In this case, different derivatives may have common paths in the computational graph that can be factored out, which can greatly impact efficiency. Consider the following code example and its computational graph:

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)
```

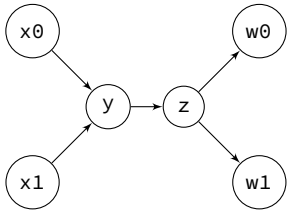
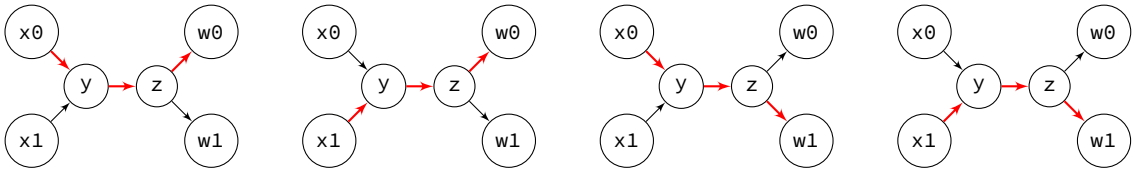


Fig. 14. Code example and computational graph with two inputs `x0, x1` and two outputs `w0, w1`

There are four derivatives between the two outputs and two inputs. We can obtain them by traversing the four corresponding paths in the computational graph:



(a) $\frac{\partial w_0}{\partial x_0}$ (b) $\frac{\partial w_0}{\partial x_1}$ (c) $\frac{\partial w_1}{\partial x_0}$ (d) $\frac{\partial w_1}{\partial x_1}$

For example, in (a), the derivative of `w0` with respect to `x0` is the product of the three red edges:

$$\frac{\partial w_0}{\partial x_0} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x_0}, \tag{59}$$

and in (b), the derivative of `w0` with respect to `x1` is

$$\frac{\partial w_0}{\partial x_1} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x_1}. \tag{60}$$

We can observe that some of the derivatives share common subpaths in the computational graph. For example the two derivatives above $\frac{\partial w_0}{\partial x_0}$ and $\frac{\partial w_0}{\partial x_1}$ share the same subpath `y, z, w0`. We can therefore factor this subpath out and premultiply $\frac{\partial w_0}{\partial y} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}$ for the two derivatives. In a larger computational graph, this factorization can have enormous impact on the performance of the derivative code, even affecting the time complexity in terms of the number of inputs or outputs.

Different automatic differentiation algorithms find common factors in the computational graph in different ways. In the most general case, finding a factorization that results in minimal operations is NP-hard [30]. Fortunately, in many common cases, such as factorization for the gradient vector, there are efficient solutions.

If the input is a scalar variable, no matter how many variables there are in the output, *forward-mode* automatic differentiation generates derivative code that has the same time complexity as the original algorithm. On the other hand, if the output is a scalar variable, no matter how many input variables there are, *reverse-mode* automatic

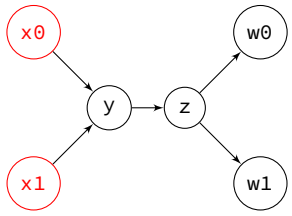
differentiation generates derivative code that has the same time complexity as the original algorithm. The latter case is particularly interesting, since it means that we can compute the gradient with the same time complexity (the "cheap gradient principle"), which can be useful for various optimization and sampling algorithms.

5.2.3 Forward-mode differentiation. We start with the simplest algorithm, usually called forward-mode automatic differentiation, and sometimes also called dual number. Forward-mode traverses the computational graph from the inputs to outputs, computing derivatives of the intermediate nodes with respect to all input variables along the way. Forward-mode is efficient when the input dimension is low and the output dimension is high, since for each node in the computational graph, we need to compute the derivatives with respect to every single input variable.

We will describe forward-mode using the previous example in Figure 14. Starting from the inputs, the goal is to propagate the derivatives with respect to the inputs using the chain rule. To handle function calls, for every function `f(x)` referenced by the output variables, we generate a derivative function `df(x, dx)`, where `dx` is the derivative of `x` with respect to the input variables.

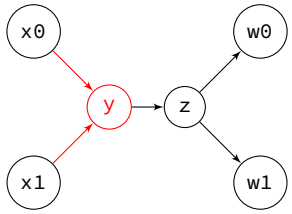
We start from the inputs `x0, x1` and generate $\frac{\partial x_0}{\partial x_0} = 1$ and $\frac{\partial x_1}{\partial x_1} = 1$. We use a 2D vector `dx0dx` to represent the derivatives of `x0` with respect to `x0` and `x1`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)
```



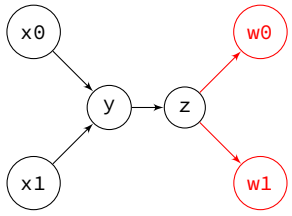
We then obtain the derivatives for `y` with respect to the inputs. We assume we already applied forward-mode automatic differentiation for `f`, so we have a derivative function `df(x0, dx0dx, x1, dx1dx)`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx,
          x1, dx1dx)
z = g(y)
w0, w1 = h(z)
```



Finally, we propagate the derivatives from `z` to the outputs `w0, w1`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx,
          x1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = h(z)
dw0dx, dw1dx = dh(z, dzdx)
```



The time complexity of the code generated by forward-mode automatic differentiation is $O(d)$ times the time complexity of the original algorithm, where d is the number of input variables. It is efficient for functions with few input variables.

However, for many applications of derivatives, including differentiable rendering, we need to differentiate functions with thousands or even millions of input variables. Using forward-mode for this would be infeasible, as

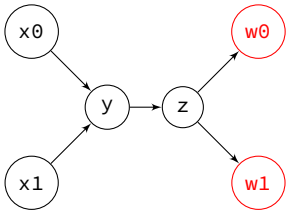
we need to compute the derivatives with respect to *all* input variables for every output in the computational graph. Fortunately, there is another algorithm called reverse-mode automatic differentiation that can generate derivative code that has the same time complexity as the original algorithm when there is only a single output, regardless of the number of input variables.

5.2.4 Reverse-mode differentiation. Reverse-mode propagates the derivatives from outputs to inputs, unlike forward-mode, which propagates the derivatives from inputs to outputs. For each node in the computational graph, we compute the derivatives of all outputs with respect to the variable at that node. Therefore reverse-mode is much more efficient when the input dimension is large and the output dimension is low. Remarkably, **reverse-mode can compute gradient in the same time complexity as the original algorithm!** However, reverse-mode is also more complicated to implement since it needs to run the original algorithm backward to propagate the derivatives.

We again use the same previous example in Figure 14 to illustrate how reverse-mode works. Similar to forward-mode, we need to handle function calls. For every function `y = f(x)` referenced by the output variables, we generate a derivative function `df(x, dy)`, where dy is a vector of derivatives of the final output with respect to the function's output `y` (in contrast, in forward-mode, the derivative functions take the input derivatives as arguments). Handling control flow and recursion in reverse-mode is more complicated. We discuss them later.

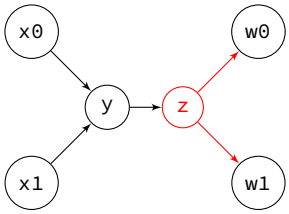
We start from the outputs `w0, w1` using $\frac{\partial w_0}{\partial w_0} = 1$ and $\frac{\partial w_1}{\partial w_1} = 1$. We use a 2D vector `dwdw0` to represent the derivatives of `w0, w1` with respect to `w0`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
```



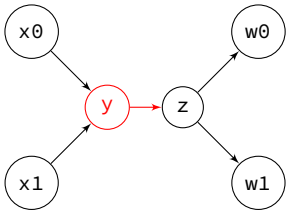
Next, we propagate the derivatives to variable `z` on which the two outputs depend. We assume we already applied reverse-mode to the function `h` and have `dh(z, dwdw0, dwdw1)`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
```



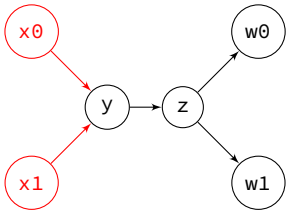
Similarly, we propagate to `y` from `z`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
dwdy = dg(y, dwdz)
```



Finally we obtain the derivatives of the outputs `w` with respect to the two inputs.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
dwdy = dg(y, dwdz)
dwx0, dwx1 = df(x0, x1, dwdy)
```



A major difference between reverse-mode and forward-mode that makes the implementation of reverse-mode much more complicated, is that we can only start the differentiation after the final output is computed. This makes it impossible to interleave the derivative code with the original code like in forward-mode. This issue has the most impact when differentiating programs with control flows or recursions.

5.2.5 Beyond forward and reverse modes. As we have discussed, forward-mode generates less operations when the number of inputs is small, while reverse-mode generates less operations when the number of outputs is small.

In general, we can think of derivative computation as a pathfinding problem on the computational graph: We want to find all the paths that connect between inputs and outputs. Many of the paths share common subpaths and it is more computationally efficient to factor out the common subpaths. Forward-mode and reverse-mode are two different greedy approaches that factor out the common subpaths either from the input node or output node.

Sometimes we do not just want to achieve the least number of operations. On modern hardware architectures, reading from and writing to the memory outside of caches is significantly more time-consuming than the arithmetic computation. Sometimes we wish to compute more to trade-offs with less memory traffic. Therefore, it is sometimes worth exploring approaches between the spectrum of forward- and reverse- modes. Mitsuba 2 [32] took this approach and applied a greedy vertex elimination algorithm [14].

5.2.6 Automatic differentiation as program transformation. Typically the implementation of automatic differentiation systems can be categorized as a point in a spectrum, depending on how much is done at compile-time. At one end of the spectrum, the *tracing* approach, or sometimes called the *taping* approach, re-compiles the derivatives whenever we evaluate the function. At the other end of the spectrum, the *source transformation* approach does as much at compile-time as possible by compiling the derivative code only once. The tracing approach has the benefit of simpler implementation, and is easier to incorporate into existing code, while the source transformation approach has better performance, but usually can only handle a subset of a general-purpose language and is much more difficult to implement.

A common misconception is that the tracing approach (PyTorch being a notable example) results in more user-friendly frontend, while the source transformation, or static compilation approach (TensorFlow 1.0 being a popular example) results in a messy meta-programming interface. In fact, the frontend of the compiler does *not* dictate how a program is compiled. A language can have the exact same syntax/frontend as PyTorch, and statically compile to highly-optimized code (this is roughly what TorchScript does). On the other hand, a language can also have the exact same syntax/frontend as TensorFlow, and being interpreted directly in a Read–Eval–Print Loop (REPL) environment (the eager-mode of TensorFlow 1.0 does this).

5.2.7 Control flows and recursions. Handling control flow and recursion in forward-mode is trivial. We do not need to modify the flow at all. Since forward-mode propagates from the inputs, for each statement, we can compute its derivative immediately.

In reverse-mode, however, control flow and recursion introduce challenges, since we need to *revert* the flow. Consider the iterative exponential example from Figure 13. To apply reverse-mode, we need to revert the for

loop. We observe an issue here: we need the intermediate `exp(result)` values for the derivatives. To resolve this, we will need to record the intermediate values during the first pass of the loop:

```
function d_f(x):
    result = x
    results = []
    for i = 1 to 8:
        results.push(result)
        result = exp(result)

    d_result = 1
    for i = 8 to 1:
        // one-based indexing
        d_result = d_result * exp(results[i])
    return d_result
```

The general strategy for transforming loops in reverse-mode is to push intermediate variables into a stack for each loop [42], then pop the items during the reverse loop. Nested loops can be handled in the same way. For efficient code generation, dependency analysis is often required to push only variables that will be used later to the stack (e.g. [40]).

The same strategy of storing intermediate variables in a stack also works for loop continuations, early exits, and conditioned while loops. We can use the size of the stack as the termination criteria. For example, we modify the previous example to a while loop and highlight the derivative code in red:

```
function d_f(x):
    result = x
    results = []
    while result > 0.1 and result < 10:
        results.push(result)
        result = exp(result)

    d_result = 1
    for i = len(results) to 1:
        d_result = d_result * exp(results[i])
    return d_result
```

In general, given a loop, we store all the intermediate variables generated in the loop inside a stack. We then *replay* the stack from the end to the beginning, and traverse the computational graph backward for the statements inside the loops. For if conditions we can do the same transformation. We will store which condition the forward pass satisfied in the stack, along with the intermediate variables of that particular branch.

Recursion is equally or even more troublesome compared to conditions loops for reverse-mode. Consider the following tail recursion that represents the same function:

```
function f(x):
    if x <= 0.1 or x >= 10:
        return x
    result = f(exp(x))
    return result
```

It is tempting to use the reverse-mode rules we developed previously to differentiate the function like the following:

```
function d_f(x, d_result):
    if x <= 0.1 or x >= 10:
        return 1
    result = f(exp(x))
    return d_f(result, d_result) * exp(x)
```

However, a close inspection reveals that the generated derivative function `d_f` has higher time complexity compared to the original function ($O(N^2)$ v.s. $O(N)$), since every time we call `d_f` we will recompute `f(exp(x))`, resulting in redundant computation.

A solution to this, similar to the case of loops, is to use the technique of memoization. We can cache the result of recursive function calls in a stack, and traverse the recursion tree in reverse by traversing the stack:

```
function d_f(x, d_result):
    if x <= 0.1 or x >= 10:
        return 1
    results = []
    result = f(exp(x), results)

    d_result = 1
    for i = len(results) to 1:
        d_result = d_result * exp(results[i])
    return d_result
```

This also works in the case where `f` recursively calls itself several times. A possible implementation is to use a tree instead of a stack to store the intermediate results.

The transformations above reveal an issue with the reverse-mode approach. While for scalar output, reverse-mode is efficient in time complexity, it is not efficient in memory complexity, since the memory usage depends on the number of instructions, or the length of the loops. A classical optimization to reduce memory usage is called "checkpointing". The key idea is to only push to, or to *checkpoint*, the intermediate variable stack sporadically, and recompute the loop from the closest checkpoint every time. Griewank [13] showed that by checkpointing only $O(\log(N))$ times for a loop with length $O(N)$, we can achieve memory complexity of $O(\log(N))$, and time complexity of $O(N \log(N))$ for reverse-mode.

5.2.8 Differentiating SIMD-based rendering programs. Differentiating a general parallel program, while preserving the parallelism is not trivial due to the potential race condition. Fortunately, in rendering, most of the parallelism are independent. Most rendering programs can be written inside a parallel for loop, launching threads that do not communicate with each other. In this case, the reverse-mode differentiation can be done with the same strategy above by first executing the parallel for loop and storing the intermediate variables in a stack of arrays, then launch the same parallel for loop again with the statements inside transformed with reverse-mode automatic differentiation. Importantly, any read from external arrays in the original parallel for loop needs to be transformed to an atomic scatter write to avoid race conditions.

Enoki [18], the core of Mitsuba 2, is a just-in-time compiler that specializes at compiling and differentiating parallel programs with independent threads. Enoki takes a C++ or Python frontend of a SIMD program, and automatically generate vectorized CPU or GPU code and the corresponding reverse-mode transformed code.

6 PHYSICS-BASED DIFFERENTIABLE RENDERING APPLICATIONS

TBD.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y.Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

[2] James Arvo. 1994. The Irradiance Jacobian for partially occluded polyhedral sources. In *SIGGRAPH '94*. 343–350.

[3] James Arvo. 1995. *Analytic methods for simulated light transport*. Ph.D. Dissertation. Yale University.

[4] Edwin Catmull. 1978. A Hidden-Surface Algorithm with Anti-Aliasing. *Comput. Graph. (Proc. SIGGRAPH)* 12, 3 (1978), 6–11.

[5] Subrahmanyan Chandrasekhar. 1960. *Radiative Transfer*. Courier Corporation.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed Ray Tracing. *Comput. Graph. (Proc. SIGGRAPH)* 18, 3 (1984), 137–145.

[7] Franklin C Crow. 1977. The aliasing problem in computer-generated shaded images. *Commun. ACM* 20, 11 (1977), 799–805.

[8] Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 2012. 3D rasterization: a bridge between rasterization and ray casting. In *Graphics Interface*. 201–208.

[9] Martin de La Gorce, David J Fleet, and Nikos Paragios. 2011. Model-based 3D hand pose estimation from monocular video. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 9 (2011), 1793–1805.

[10] Harley Flanders. 1973. Differentiation under the integral sign. *The American Mathematical Monthly* 80, 6 (1973), 615–627.

[11] Ned Greene and Michael Kass. 1994. Error-bounded antialiased rendering of complex environments. In *SIGGRAPH*. 59–66.

[12] Ned Greene, Michael Kass, and Gavin Miller. 1993. Hierarchical Z-buffer visibility. In *SIGGRAPH*. ACM, 231–238.

[13] Andreas Griewank. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. 1, 1 (1992), 35–54.

[14] Andreas Griewank and Shawn Reese. 1991. On the Calculation of Jacobian Matrices by the Markowitz Rule. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, 126–135.

[15] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Vol. 105. SIAM.

[16] Warren Hunt, Michael Mara, and Alex Nankervis. 2018. Hierarchical Visibility for Virtual Reality. *ACM Comput. Graph. Interact. Tech. (Proc. I3D)* 1, 1 (2018), 1–18.

[17] Wolfram Research, Inc. [n.d.]. Mathematica, Version 11.3. Champaign, IL, 2018.

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

[19] Henrik Wann Jensen. 1995. Importance driven path tracing using the photon map. In *Rendering Techniques (Proc. EGWR)*. Springer, 326–335.

[20] James T. Kajiya. 1986. The Rendering Equation. In *SIGGRAPH '86*. 143–150.

[21] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2018. Neural 3D mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3907–3916.

[22] L. Gary Leal. 2007. *Advanced Transport Phenomena: fluid mechanics and convective transport processes*. Vol. 7. Cambridge University Press.

[23] Tzu-Mao Li. 2019. *Differentiable visual computing*. MIT PhD Dissertation.

[24] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo ray tracing through edge sampling. *ACM Trans. Graph.* 37, 6 (2018), 222:1–222:11.

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

[26] Matthew M Loper and Michael J Black. 2014. OpenDR: An approximate differentiable renderer. In *European Conference on Computer Vision*. Springer, 154–169.

[27] Morgan McGuire. 2004. Observations on silhouette sizes. *J. Graph. Tools* 9, 1 (2004), 1–12.

[28] Morgan McGuire and Louis Bavoil. 2013. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques* 2, 4 (2013).

[29] Don P Mitchell and Arun N Netravali. 1988. Reconstruction filters in computer-graphics. *Comput. Graph. (Proc. SIGGRAPH)* 22, 4 (1988), 221–228.

[30] Uwe Naumann. 2008. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming* 112, 2 (2008), 427–441.

[31] Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. 2020. Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)* 39, 4 (July 2020). https://doi.org/10.1145/3386569.3392406

[32] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 1–17.
[33] Matt Olson and Hao Zhang. 2006. Silhouette extraction in Hough space. *Comput. Graph. Forum (Proc. Eurographics)* 25, 3 (2006), 273–282.
[34] Eric Paquette, Pierre Poulin, and George Drettakis. 1998. A Light Hierarchy for Fast Rendering of Scenes with Many Lights. *Comput. Graph. Forum (Proc. Eurographics)* (1998), 63–74.
[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[36] Lev Semenovich Pontryagin. 1962. *Mathematical theory of optimal processes.* CRC Press.
[37] Thomas Porter and Tom Duff. 1984. Compositing digital images. (1984), 253–259.
[38] Ravi Ramamoorthi, Dhruv Mahajan, and Peter Belhumeur. 2007. A First-order Analysis of Lighting, Shading, and Shadows. *ACM Trans. Graph.* 26, 1 (2007), 2:1–2:21.
[39] Pedro V. Sander, Hugues Hoppe, John Snyder, and Steven J. Gortler. 2001. Discontinuity edge overdraw. In *Symposium on Interactive 3D Graphics and Games*. ACM, 167–174.
[40] Michelle Mills Strout and Paul Hovland. 2006. Linearity analysis for automatic differentiation. In *International Conference on Computational Science*. Springer, 574–581.
[41] Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph.D. Dissertation. Stanford University. Advisor(s) Guibas, Leonidas J.
[42] Dominique Villard and Michael B Monagan. 1999. ADrien: an implementation of automatic differentiation in Maple. In *International Symposium on Symbolic and Algebraic Computation*. 221–228.
[43] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. 2005. Lightcuts: A Scalable Approach to Illumination. *ACM Trans. Graph. (Proc. SIGGRAPH)* 24, 3 (2005), 1098–1107.
[44] Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-Space Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 6 (2020), 143:1–143:19.
[45] Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shaung Zhao. 2019. A Differential Theory of Radiative Transfer. *ACM Trans. Graph.* 38, 6 (2019), 227:1–227:16.