Mitsuba 2: A Retargetable Forward and Inverse Renderer

MERLIN NIMIER-DAVID[*], École Polytechnique Fédérale de Lausanne
DELIO VICINI[*], École Polytechnique Fédérale de Lausanne
TIZIAN ZELTNER, École Polytechnique Fédérale de Lausanne
WENZEL JAKOB, École Polytechnique Fédérale de Lausanne

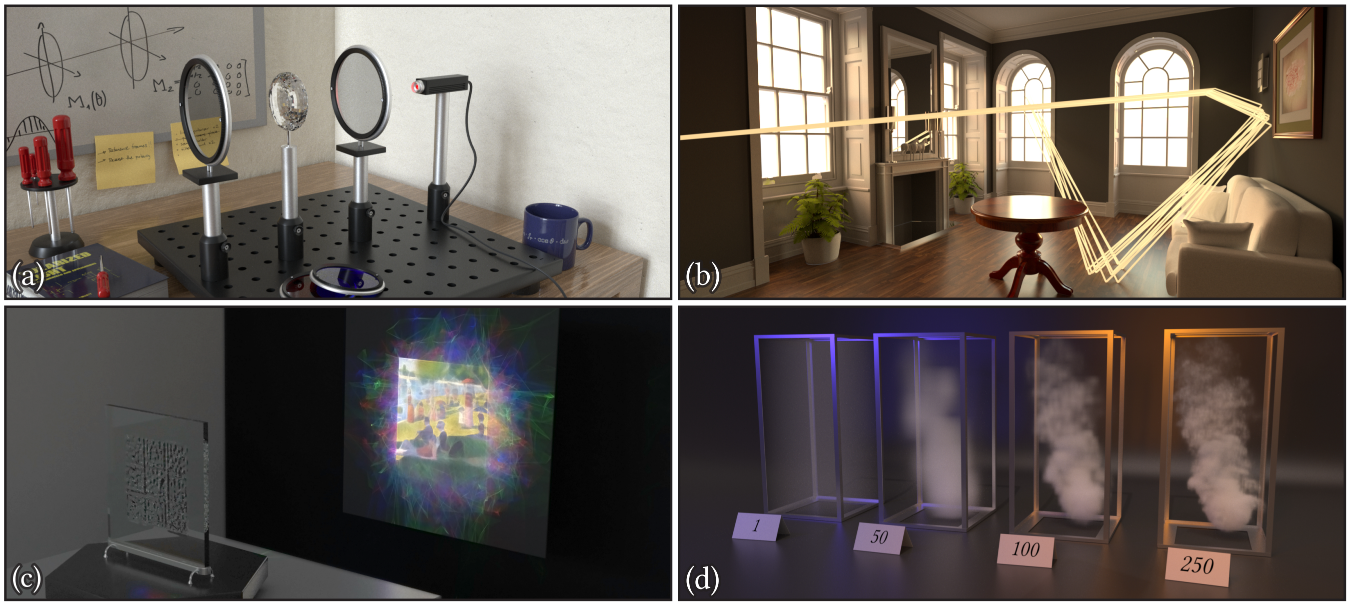


Fig. 1. Four applications enabled using automated transformations of a generic renderer. **(a)** Polarized spectral rendering of an optical experiment that analyzes light with elliptical polarization. **(b)** A coherent MCMC rendering algorithm that explores bundles of nearby light paths to improve convergence at equal render time. **(c)** A refractive slab optimized by inverse rendering to focus light with three primary colors into a rendition of the painting *A Sunday Afternoon on the Island of La Grande Jatte* by Georges Seurat. **(d)** Reconstruction of a smoke volume from reference images; multiple iterations of the optimization are shown. Please refer to the supplemental video for animated visualizations of many results shown in this paper.

Modern rendering systems are confronted with a dauntingly large and growing set of requirements: in their pursuit of realism, physically based techniques must increasingly account for intricate properties of light, such as its spectral composition or polarization. To reduce prohibitive rendering times, vectorized renderers exploit coherence via instruction-level parallelism on CPUs and GPUs. Differentiable rendering algorithms propagate derivatives through a simulation to optimize an objective function, e.g., to reconstruct a scene from reference images. Catering to such diverse use cases is challenging and has led to numerous purpose-built systems—partly, because retrofitting features of this complexity onto an existing renderer involves an error-prone and infeasibly intrusive transformation of elementary data structures, interfaces between components, and their implementations (in other words, everything).

We propose Mitsuba 2, a versatile renderer that is intrinsically retargetable to various applications including the ones listed above. Mitsuba 2 is implemented in modern C++ and leverages template metaprogramming to replace types and instrument the control flow of components such as BSDFs, volumes, emitters, and rendering algorithms. At compile time, it automatically transforms arithmetic, data structures, and function dispatch, turning generic algorithms into a variety of efficient implementations without the tedium of manual redesign. Possible transformations include changing the representation of color, generating a "wide" renderer that operates on bundles of light paths, just-in-time compilation to create computational kernels that run on the GPU, and forward/reverse-mode automatic differentiation. Transformations can be chained, which further enriches the space of algorithms derived from a single generic implementation.

We demonstrate the effectiveness and simplicity of our approach on several applications that would be very challenging to create without assistance: a rendering algorithm based on coherent MCMC exploration, a caustic design method for gradient-index optics, and a technique for reconstructing heterogeneous media in the presence of multiple scattering.

CCS Concepts: • **Computing methodologies → Rendering**.

[*]Joint first authors

Authors' addresses: Merlin Nimier-David, EPFL, merlin.nimier-david@epfl.ch; Delio Vicini, EPFL, delio.vicini@epfl.ch; Tizian Zeltner, EPFL, tizian.zeltner@epfl.ch; Wenzel Jakob, EPFL, wenzel.jakob@epfl.ch.

0730-0301/2019/11-ART203 $15.00
https://doi.org/http://dx.doi.org/10.1145/3355089.3356498

Additional Key Words and Phrases: Ray Tracing, Global Illumination, Differentiable Rendering, SIMD

ACM Reference Format:
Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Trans. Graph.* 38, 6, Article 203 (November 2019), 17 pages. https://doi.org/http://dx.doi.org/10.1145/3355089.3356498

1 INTRODUCTION

Realism has been a major driving force since the inception of the field of computer graphics, and algorithms that generate realistic renderings using physical light transport simulations are now in widespread use. These methods approximate high-dimensional integrals over sets of light paths using *Monte Carlo* (MC) or *Markov Chain Monte Carlo* (MCMC) sampling along with sophisticated appearance models that account for the interaction of light and matter. Because they simulate the intricacies of the visual world, these systems tend to be large and complex: for instance, current versions of PBRT [Pharr et al. 2016] and Mitsuba [Jakob 2010] consist of over 60 and 180 thousand lines of C++ code, respectively. Industrial rendering systems are larger still, with typical sizes on the order of one million lines of code [Hanika 2019].

Despite their size, these systems lack many features of growing importance: for instance, predictive rendering applications require that simulations correctly account for the effects of both spectral transport and polarization. In theory, this is a straightforward extension: one must simply replace RGB radiance and reflectance values by wavelength-dependent Stokes vectors and Mueller matrices and update a few models that are directly affected by these phenomena, e.g., by switching from a spectrally constant refractive index to Cauchy's equation and adopting the complex-valued form of the Fresnel equations. In practice, the modification changes the representation of quantities central to any renderer, requiring a substantial redesign of the entire system.

Vectorized rendering systems, such as MoonRay [Lee et al. 2017] and Iray [Keller et al. 2017], leverage Single Instruction/Multiple Data (SIMD) units on modern CPUs and GPUs to efficiently sample many light paths in parallel, thereby reducing the overall computation time. Efficient vectorization generally involves a mechanical translation into a sequence of *compiler intrinsics* or specialized compiler infrastructure to generate SIMD machine instructions, algorithms with improved coherence[1], and data structures arranged in *structure of arrays* (SoA) form. The latter requires transposing the memory layout of the entire application, which constitutes another example of a highly intrusive change to every system component. Designing vectorized renderers remains a time-consuming endeavor—for instance, the recent MoonRay project marks the result of a concerted four-year development effort of a team of engineers [Lee et al. 2017; Fascione et al. 2017].

Over the last decade, gradient-based optimization techniques have fueled dramatic progress in machine learning, computer vision, and related fields. In computer graphics, differentiable rendering algorithms have opened the door to solving challenging inverse problems including computational material design and methods for scene reconstruction that account for global transport effects such as shadows, interreflection, or even refraction. A differentiable rendering algorithm is able to compute derivatives of the entire simulation with respect to input parameters that could include camera pose, geometry (e.g. vertex positions), BSDFs, textures, and volumes. Recently, Li et al. [2018a] presented the first comprehensive technique for differentiable rendering that accounts for all salient transport effects including discontinuities. Its freely available implementation reveals the inherent challenges of realizing such systems: discontinuities aside, the basic loop of the underlying path tracer (which would likely be realizable using at most 200 lines of C++ code in a "classical" renderer) expands into approximately 3000 lines of hand-written derivative code partitioned over multiple CUDA kernels that communicate through large groups of auxiliary buffers. Adding a new model to the system entails manual differentiation of all relevant expressions with careful consideration of what information must be cached when and where, so that it can be passed from kernel to kernel during subsequent gradient propagation passes. More advanced (but well-understood) bidirectional or volumetric techniques present a formidable challenge in this context.

To address these problems, we propose Mitsuba 2, an open-source architecture for constructing renderers that are intrinsically retargetable to these application domains. Mitsuba 2 takes abstract implementations of a set of standard components (e.g. rendering algorithms, BSDF models, etc.), and *lifts* them onto a concrete set of types, systematically transforming the underlying algorithms to enable a particular feature. Possible transformations include changing the representation of radiance (e.g. to polarized spectra), generating a "wide" renderer that operates on bundles of light paths using AVX512 vector instructions or CUDA kernels, and automatic differentiation of the entire simulation. These transformations can be chained, which further enriches the space of algorithms that can be derived from a single generic implementation. Concretely, our contributions are:

- A versatile framework of composable types that exploit compile-time computation to retarget a complete rendering system from a generic specification to concrete implementations suited to a range of different tasks.
- A lazy *just-in-time* (JIT) compiler that symbolically executes arithmetic and control flow to generate computational kernels for later execution on a GPU.
- An efficient graph-based approach for simultaneous forward- and reverse-mode *automatic differentiation* (AD) that seamlessly composes with other transformations.
- A simplification algorithm that periodically simplifies the graph data structure used by automatic differentiation to reduce the memory usage of differentiable rendering.

In addition to these system contributions, we present several novel applications that are enabled by our system:

- A MCMC rendering technique that explores bundles of nearby light paths to generate coherent work. Our system is able to exploit the similar control flow in each bundle to improve convergence at equal render time.
- A method for creating gradient-index optics that focus incident illumination into caustics that reproduce multiple user-specified

[1] i.e., regular control flow and memory accesses with spatio-temporal locality.

images. Alternatively, height fields with constant index of refraction can be optimized to reproduce color images from an illuminant that provides three primary colors.
- A method for reconstructing the parameters of a heterogeneous participating medium from a set of input images, while accounting for multiple scattering. We showcase reconstruction of 3D densities (e.g. smoke or steam) and scattering-aware texture reproduction inside a dielectric slab.

The open source implementation of Mitsuba 2 and our experimental validations are available at https://mitsuba-renderer.org.

2 BACKGROUND AND RELATED WORK

We now discuss prior work and introduce relevant background material. Since our system touches on a large set of related topics, we restrict our discussion to the most relevant articles.

Coherent and vectorized rendering. Monte Carlo rendering techniques like path tracing sample their domain at random locations, making them ill-suited for modern processor architectures that rely on various forms of coherence to obtain good performance. Coherent rendering techniques are designed to address this flaw, for instance by accumulating a larger amount of work that is then reordered based on the similarity of materials [Áfra et al. 2016], or rays [Pharr et al. 1997]. Disney's Hyperion [Burley et al. 2018] and Dreamwork Animation's MoonRay [Lee et al. 2017] use this approach, the latter also vectorizing arithmetic operations using ISPC [Pharr and Mark 2012]. Weta's Manuka [Fascione et al. 2018] renderer partially evaluates shaders in a coherent shading phase followed by incoherent path tracing. A key issue faced by all of these methods is the difficulty of finding coherent work within an algorithm that is fundamentally incoherent. Our system enables transparent vectorization similar to ISPC that we showcase in a MCMC rendering algorithm that directly generates coherent work.

Automatic differentiation. Many numerical methods require cheap access to derivatives, often to maximize an objective function using a variant of gradient descent. *Automatic differentiation* (AD) provides a powerful tool to automate this process via systematic application of the chain rule [Griewank and Walther 2008].

One of two AD variants is typically used depending on the nature of the problem: starting with a set of inputs (e.g. `a`, `b`) and associated derivatives (`da`, `db`) *forward-mode* AD instruments every operation (e.g. `c = a * b`) with an additional step that tracks the evolution of derivatives (`dc = a*db + b*da`). However, a separate propagation pass is needed per input, and the approach thus becomes prohibitively costly when there are many of them. *Reverse-mode* AD, also known as *backpropagation* in the context of neural networks [Rumelhart et al. 1986], is the method of choice when the model has few outputs but many inputs. It traverses the computational graph from outputs to inputs, repeatedly evaluating the chain rule in reverse. A traditional difficulty of reverse-mode AD is that differentiation can only begin once the output has been computed. A record of all intermediate computations must furthermore be kept in memory to enable the backward traversal, which can become prohibitive for long-running computations. Checkpointing strategies [Volin and Ostrovskii 1985] reduce storage by discarding information that can be recovered by repeating parts of the computation, but this introduces considerable additional complexity.

Machine learning frameworks, such as PyTorch [Paszke et al. 2017] and TensorFlow [Abadi et al. 2015] provide a convenient interface to array-based computation on GPUs with built-in reverse-mode differentiation. Both are designed for neural networks, whose computation graphs typically consist of a few hundred arithmetically intensive operations like matrix-vector multiplications or convolutions. In contrast, Mitsuba 2 works with highly unstructured graphs that are potentially extremely large, containing millions of operations with very low arithmetic intensity (e.g. additions).

Domain Specific Languages. A number of domain-specific languages (DSLs) have been proposed to accelerate the development of efficient numerical algorithms in the area of computer graphics. Halide [Ragan-Kelley et al. 2013] facilitates the design of highly optimized image processing pipelines, decoupling the computation from the way it is carried out to expose optimization opportunities. The approach can be combined with reverse-mode AD to create new neural network layers or solve inverse problems [Li et al. 2018b]. Anderson et al. [2017] recently proposed a DSL for Monte Carlo rendering, which symbolically differentiates sampling code to determine associated probability densities that are crucial for many rendering techniques.

Our method is also related to work by Pérard-Gayot et al. [2019], whose system builds on AnyDSL [Leißa et al. 2018] to generate a vectorized implementation for rendering a specific input scene, while applying partial evaluation to combine and specialize the individual components of a renderer. In contrast, our work targets a wider set of transformations including changes to the formulation of light transport (e.g. polarization) and problem statement (e.g. inverse rendering via differentiation) but does not specifically focus on partial evaluation—combining both approaches is likely feasible but beyond the scope of this article.

Differentiable rendering. Several approximate differentiable rendering techniques have been proposed in prior work, which rely on smooth rasterization of meshes or volumes, while ignoring global light transport effects [Kato et al. 2017; Liu et al. 2019; Loper and Black 2014; Petersen et al. 2019; Rhodin et al. 2015]. In contrast, physically based methods that correctly account for interreflection differentiate full transport-level simulations, e.g. to optimize material parameters of surfaces [Azinović et al. 2019; Che et al. 2018] or volumes [Gkioulekas et al. 2013; Khungurn et al. 2015; Zhao et al. 2016]. Special precautions are furthermore required to obtain correct derivatives of non-differentiable visibility changes at silhouette edges [Li et al. 2018a].

Our system is most related to the second category of work, and its main contribution is to provide a straightforward path from a transport algorithm to an efficient differentiable implementation that runs on the GPU. Our system also incorporates a novel approach for dealing with non-differentiable transport effects using a change of variables formulation that is the topic of a separate paper [Loubet et al. 2019] (visibility involves a different set of challenges that are orthogonal to the system-related aspects discussed in this article). Note that none of the applications shown in this article rely on visibility gradients.

Automatic differentiation of a renderer is also helpful in applications other than differentiable rendering, for instance to construct improved MCMC proposals when rendering scenes with challenging light transport [Li et al. 2015].

Template metaprogramming. The term *metaprogramming* refers to a broad range of techniques, in which a program is able to rewrite its own structure (or that of another program) either statically at compile time or dynamically during execution. *Template metaprogramming* (TMP) denotes a static variant of this approach that was "discovered" in the early 1990s when it was found that C++ templates could be used to perform Turing-complete computation during compilation [Meyers 2005]. Initially considered a dangerous feature due to generally fragile support and superlinear complexity of template expansion in early compilers, TMP has seen significant refinements and extensions in later revisions to the standard (C++11, 14, 17) that have elevated its status to that of a top-level language feature.

Mitsuba 2 performs nontrivial transformations of complete programs that would generally require custom compiler infrastructure or tools for source code synthesis (in particular, our requirements far exceed the capabilities of "generics" or "macros" available in languages such as Ada, Rust, and .NET). Even within C++, the specifics of our design have only become possible due to extensions in the recent 2017 standard revision. For this reason, we briefly review relevant features, and how they are used by our system.

Note that TMP usage mainly occurs within internals of our system—rudimentary familiarity with template concepts suffices when developing Mitsuba 2 code.

- *Templates.* Mitsuba 2 components are specified as generic C++ functions and structures parameterized by unknown target-specific types (e.g. the representation of colors or floating point values) and / or constants (e.g. size or depth). For instance, the fragment

```
template <typename Float> struct MicrofacetDistribution {
    using Vector3f = Array<Float, 3>; /* ... */
};
```

  declares a data structure parameterized by an arithmetic type (`typename Float`) that is then used in the declaration of a more complex 3D vector type. In the simplest case, `Float` could be an ordinary floating point value. More advanced usage might involve types that perform arithmetic symbolically.
- *Variadic templates* are templates that accept an arbitrary number of arguments. For example, the generic function

```
template <typename... T> auto f(const T&... args) {
    return g(h(args)...);
}
```

  rewrites function invocations of the form `f(x1, x2, ...)` into `g(h(x1), h(x2), ...)`. We use variadic templates to realize virtual method calls (e.g. BRDF sampling) on the various targets.
- *Compile-time conditionals* facilitate targeted removal of code fragments subject to user-specified conditions. For instance, suppose that the nested block `/* ...*/` in the snippet

```
if constexpr (is_polarized_v<Spectrum>) { /* ... */ }
```

  is only meaningful when dealing with polarized spectra, generating a compilation error in the unpolarized case. To avoid this problem, the **`if constexpr`** statement queries a *type trait* at compile time, excising the nested block in the negative case.
- *Type computation.* It is often difficult or impossible to define types of expressions in a generic program. To address this flaw, modern C++ constructs enable type specifications that are themselves the result of a compile-time calculation. For instance, the snippet

```
using Value = decltype(a[0] + b[0]);
constexpr int Size = A::Size + B::Size;
Array<Value, Size> result = /* ... */;
```

  computes the type resulting from the concatenation of arrays `A a` and `B b`, while applying standard promotion rules (combining `int` and `float` yields `float`, etc.). Here, **`constexpr`** denotes computation to be performed at compile time, and **`decltype`** returns the type of a nested expression without evaluating it.

Expression templates. Widely used numerical libraries such as Eigen [Guennebaud et al. 2010] and Adept [Hogan 2014] rely on a technique known as *expression templates* (ET) [Veldhuizen 1995]. Mathematical expressions in these frameworks return complex types that encode the sequence of operations needed for evaluation rather than triggering evaluation immediately. This enables global optimizations that would be unavailable when considering the operations individually.

We experimented with expression templates during the early stages of this project but ultimately found them not to be a good fit for Mitsuba 2. The approach is ideal for compact statements (e.g. simple matrix updates in the case of Eigen) but does not scale to large expressions that encode complex functionality (e.g. a complete microfacet model with visible normal sampling). Because ET cannot model variable reuse and common subexpressions, the size of the expression templates tends to grow exponentially as a function of the size of the program, which eventually prevents practical usage.

3 SYSTEM DESIGN

The design of Mitsuba 2 was influenced by three guiding principles:

- *No duplication.* Special cases will inevitably arise during certain program transformations—we wish to support these without creating many special variants of an algorithm.
- *Unobtrusiveness.* Several transformations discussed in Section 1 substantially increase the size and complexity of an implementation, obscuring physical and algorithmic concepts. We thus want development to take place at the *input end* of such transformations. The development of generic algorithms should furthermore resemble their "classical" counterparts as much as possible.
- *Modularity.* Physically based rendering systems admit a particularly modular architecture and are often partitioned into a large set of loadable plug-in modules that implement materials, rendering algorithms, and so on. To support the same level of modularity, our approach should be compatible with separate compilation of the various parts of the renderer.

Our system is composed of two principal components: the first, named *Enoki* [Jakob 2019], is a template library responsible for vectorization, JIT compilation, and program transformations. Intended to be as general as possible, it does not contain any rendering-specific code.

The second component builds on top of Enoki and implements a complete rendering system designed for compatibility with the Mitsuba 0.6 [Jakob 2010] scene description language. It replicates Mitsuba's interfaces and plugins with suitable abstractions that admit the discussed program transformations. This component also provides fine-grained bindings and utilities for prototyping forward and inverse rendering pipelines in Python.

We begin with a discussion of the overall architecture, and how arithmetic operations are realized in our system. Following this, we turn to more complex cases that arise in the context of rendering.

Static arrays. The fundamental building block of Enoki is a generic fixed-size container `Array<Value, Size>`, whose default implementation intercepts and carries out arithmetic operations component-wise by forwarding them to its elements (for instance, `a = b + c` will be rewritten into `a[i] = b[i] + c[i]`). Its template parameter `Value` could be an arbitrary data structure (e.g. a string), an arithmetic type, or another Enoki array. In addition to `Array`, our system also provides a set of specializations with slightly different semantics. For instance, matrices or quaternions require a different product operator, while points, vectors, and surface normals behave differently under linear transformations. These containers can be arbitrarily nested to create higher-order tensors, such as a $4 \times 4 \times 8 \times N$ array containing a packet of N spectral Mueller matrices that are each sampled at 8 discrete wavelengths.

All arrays furthermore support implicit *broadcasting*, which suitably expands the size of a tensor so that an operation can be carried out. This is particularly important where vectorized and non-vectorized portions of the system meet. For instance, suppose that the aforementioned higher-order tensor occurs in a product involving another Mueller matrix or a discrete color spectrum. In such a case, Enoki inspects the types of the involved operands at compile time to determine that a broadcast to dimensions (1,2) or 3 of the rank-4 tensor is necessary.

Enoki provides *vertical* and *horizontal* arithmetic operations that are each split into a target-independent frontend portion responsible for broadcasting and type conversion, and a target-specific backend portion. Vertical operations proceed component-wise and produce a tensor of the same shape, while horizontal operations involve a reduction over one or more dimensions, returning a tensor of lower rank. To illustrate these concepts, we show a cross-section through front- and backend parts of a simple vertical product operation:

Frontend. The frontend part takes two arguments of type `T1` and `T2`, of which at least one must be an Enoki array.

```cpp
template <typename T1, typename T2, enable_if_array_t<T1, T2> = 0>
auto operator*(const T1 &a1, const T2 &a2) {
    using E = expr_t<T1, T2>;
    if constexpr (is_same_v<T1, E> && is_same_v<T2, E>)
        return a1.mul_(a2);
    else
        return operator*(E(a1), E(a2));
}
```

The expression `expr_t<T1, T2>` uses TMP to compute the type `E` of an expression involving `a1` and `a2`, while accounting for steps such as type promotion and broadcasting. For instance, multiplying a (scalar) floating point value by an n-d integer array will yield a n-d floating point array. At this point, there are two possibilities: either `T1`, `T2`, and `E` are all identical, in which case the operation is forwarded to the backend method `mul_()`. Otherwise, the operation invokes itself recursively, using the constructor of `E` to convert and broadcast the input arguments into the right format and shape.

CPU Backend. The default backend executes the operation on the desired target platform, relying on a pattern matching mechanism known as *partial specialization*. A large set of `Array<Value, Size>` specializations intercept certain combinations of `Value` and `Size` that are natively supported. For instance, operations involving `Array<float, 16>` map to a single instruction on processors with the AVX512 instruction set, and Enoki thus routes them to a backend that uses compiler intrinsics to generate the desired `vmulps` machine instruction. The function shown below is a method of this backend.

```cpp
Array mul_(const Array &a) const { return _mm512_mul_ps(m, a.m); }
```

The pattern matching mechanism is recursive—arrays with too large or odd sizes that prevent vectorization are partitioned into two sub-arrays, whose larger part is a power of two, and the process repeats anew. This all happens during compilation and hence incurs no runtime cost. In this way, an operation `f = f * f` involving a hypothetical `Array<float, 24>` e.g. compiles to

```
vmulps zmm0, zmm0, zmm0   ; 16-wide multiplication
vmulps ymm1, ymm1, ymm1   ;  8-wide multiplication
```

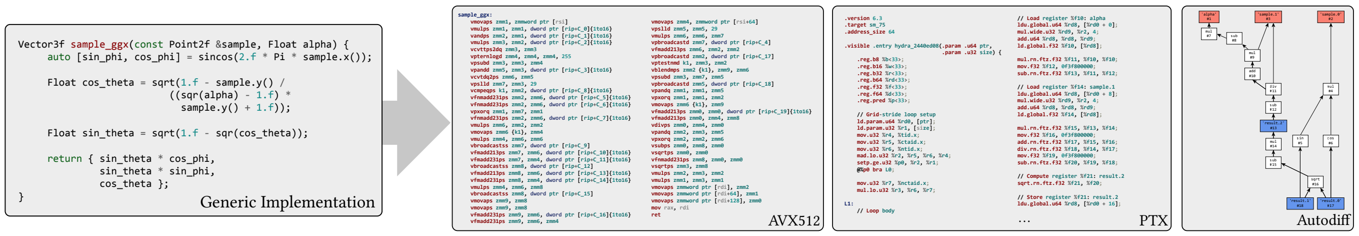
We currently provide backends for SSE4.2, AVX, AVX2, and AVX512 on Intel-compatible processors, NEON on ARM processors, and a scalar fallback mode. Unlike manually vectorized code that relies on compiler intrinsics, the combination of routing and partial specialization makes algorithms developed in Enoki platform-independent (a similar goal is pursued by ISPC [Pharr and Mark 2012]).

GPU backend. Another array type `GPUArray<Value>` provides *dynamically sized* 1D arrays that are stored on a graphics card. A simple backend for such an array could dispatch each arithmetic operation to a pre-compiled GPU kernel[2], but this leads to poor hardware utilization due to memory traffic (repeated reads and writes of operands) and the large overhead of launching kernels for such a small amount of computation. Consider the following simple program, which counts how many elements of a randomly distributed set of points on $[0, 1]^3$ fall within a sphere of radius one:

```cpp
1  using Float    = GPUArray<float>;
2  using UInt64   = GPUArray<uint64_t>;
3  using Vector3f = Array<Float, 3>;
4  PCG32<UInt64> rng(arange<UInt64>(1000000));
5  Vector3f v(rng.next_float(), rng.next_float(), rng.next_float());
6  size_t inside = count(norm(v) < 1.f);
```

Here, lines 1-3 set up the necessary types, `arange<UInt64>()` in line 4 generates an integer sequence with 1 million entries to select separate streams of a PCG32 random number generator [O'Neill 2014], and line 6 carries out a horizontal counting operation. PCG32 is a linear congruential generator, and operations involving it reduce to a sequence of multiplications and bit-level manipulations (XOR, OR, shifts, etc.).

[2]This is in fact what frameworks such as PyTorch do by default.



```
Vector3f sample_ggx(const Point2f &sample, Float alpha) {
    auto [sin_phi, cos_phi] = sincos(2.f * Pi * sample.x());

    Float cos_theta = sqrt(1.f - sample.y() /
                     ((sqr(alpha) - 1.f) *
                      sample.y() + 1.f));

    Float sin_theta = sqrt(1.f - sqr(cos_theta));

    return { sin_theta * cos_phi,
             sin_theta * sin_phi,
             cos_theta };
}
```

```
sample_ggx:
    vmovaps zmm1, zmmword ptr [rsi]
    vmulps zmm1, zmm1, dword ptr [rip+C_0]{1to16}
    vandps zmm2, zmm1, dword ptr [rip+C_1]{1to16}
    vmulps zmm3, zmm2, dword ptr [rip+C_2]{1to16}
    vcvttps2dq zmm3, zmm3
    vpternlogd zmm4, zmm4, zmm4, 255
    vpsubd zmm3, zmm3, zmm4
    vpandd zmm5, zmm3, dword ptr [rip+C_3]{1to16}
    vcvtdq2ps zmm6, zmm5
    vpslld zmm7, zmm3, 29
    vcmpeqps k1, zmm2, dword ptr [rip+C_8]{1to16}
    vfnmadd231ps zmm2, zmm6, dword ptr [rip+C_5]{1to16}
    vfnmadd231ps zmm2, zmm6, dword ptr [rip+C_6]{1to16}
    vpxorq zmm1, zmm7, zmm1
    vfnmadd231ps zmm2, zmm6, dword ptr [rip+C_7]{1to16}
    vmulps zmm6, zmm2, zmm2
    vmovaps zmm6 {k1}, zmm4
    vmulps zmm4, zmm6, zmm6
    vbroadcastss zmm7, dword ptr [rip+C_9]
    vfmadd213ps zmm7, zmm6, dword ptr [rip+C_10]{1to16}
    vfmadd231ps zmm7, zmm4, dword ptr [rip+C_11]{1to16}
    vbroadcastss zmm8, dword ptr [rip+C_12]
    vfmadd213ps zmm8, zmm6, dword ptr [rip+C_13]{1to16}
    vfmadd231ps zmm8, zmm4, dword ptr [rip+C_14]{1to16}
    vmulps zmm4, zmm6, zmm8
    vbroadcastss zmm8, dword ptr [rip+C_15]
    vmovaps zmm9, zmm8
    vmovaps zmm9, zmm8
    vfmadd231ps zmm9, zmm6, dword ptr [rip+C_16]{1to16}
    vfmadd231ps zmm9, zmm6, zmm4
    vmovaps zmm4, zmmword ptr [rsi+64]
    vpslld zmm5, zmm5, 29
    vmulps zmm6, zmm6, zmm7
    vpbroadcastd zmm7, dword ptr [rip+C_4]
    vfmadd213ps zmm6, zmm2, zmm2
    vpbroadcastd zmm2, dword ptr [rip+C_17]
    vptestnmd k1, zmm3, zmm2
    vblendmps zmm2 {k1}, zmm9, zmm6
    vpsubd zmm3, zmm7, zmm5
    vpbroadcastd zmm5, dword ptr [rip+C_18]
    vpandq zmm1, zmm1, zmm5
    vpxorq zmm1, zmm1, zmm2
    vmovaps zmm6 {k1}, zmm9
    vfmadd213ps zmm0, zmm0, dword ptr [rip+C_19]{1to16}
    vfmadd213ps zmm0, zmm4, zmm8
    vdivps zmm0, zmm4, zmm0
    vpandq zmm2, zmm3, zmm5
    vpxorq zmm2, zmm2, zmm6
    vsubps zmm0, zmm8, zmm0
    vsqrtps zmm0, zmm0
    vfnmadd231ps zmm8, zmm0, zmm0
    vsqrtps zmm3, zmm8
    vmulps zmm2, zmm3, zmm2
    vmulps zmm1, zmm3, zmm1
    vmovaps zmmword ptr [rdi], zmm2
    vmovaps zmmword ptr [rdi+64], zmm1
    vmovaps zmmword ptr [rdi+128], zmm0
    mov rax, rdi
    ret
```

```
.version 6.3
.target sm_75
.address_size 64

.visible .entry hydra_2440ed08(.param .u64 ptr,
                               .param .u32 size) {
    .reg.b8 %b<33>;
    .reg.b16 %w<33>;
    .reg.b32 %r<33>;
    .reg.b64 %rd<33>;
    .reg.f32 %f<33>;
    .reg.f64 %d<33>;
    .reg.pred %p<33>;

    // Grid-stride loop setup
    ld.param.u64 %rd0, [ptr];
    ld.param.u32 %r1, [size];
    mov.u32 %r4, %tid.x;
    mov.u32 %r5, %ctaid.x;
    mov.u32 %r6, %ntid.x;
    mad.lo.u32 %r2, %r5, %r6, %r4;
    setp.ge.u32 %p0, %r2, %r1;
    @%p0 bra L0;

    mov.u32 %r7, %nctaid.x;
    mul.lo.u32 %r3, %r6, %r7;

L1:
    // Loop body

    // Load register %f10: alpha
    ldu.global.u64 %rd8, [%rd0 + 0];
    mul.wide.u32 %rd9, %r2, 4;
    add.u64 %rd8, %rd8, %rd9;
    ld.global.f32 %f10, [%rd8];

    mul.rn.ftz.f32 %f11, %f10, %f10;
    mov.f32 %f12, 0f3f800000;
    sub.rn.ftz.f32 %f13, %f11, %f12;

    // Load register %f14: sample.1
    ldu.global.u64 %rd8, [%rd0 + 8];
    mul.wide.u32 %rd9, %r2, 4;
    add.u64 %rd8, %rd8, %rd9;
    ld.global.f32 %f14, [%rd8];

    mul.rn.ftz.f32 %f15, %f13, %f14;
    mov.f32 %f16, 0f3f800000;
    add.rn.ftz.f32 %f17, %f15, %f16;
    div.rn.ftz.f32 %f18, %f14, %f17;
    mov.f32 %f19, 0f3f800000;
    sub.rn.ftz.f32 %f20, %f19, %f18;

    // Compute register %f21: result.2
    sqrt.rn.ftz.f32 %f21, %f20;

    // Store register %f21: result.2
    ldu.global.u64 %rd8, [%rd0 + 16];

    ...
```

Fig. 2. Starting from an algorithmic template (shown: importance sampling a GGX lobe), our implementation is able to generate high-quality vectorized implementations for CPUs and GPUs. Further instrumentation to track the computation graph enables forward and reverse-mode automatic differentiation.

This example is extremely simple compared to a typical physically based shading model, yet over 180 kernel launches would be needed to execute it using the previously mentioned approach (56 for seeding the random number generators, 32 per sample, and 7 for the final count). While we could create specialized kernels that combine some of these operations (e.g. to generate uniform variates), this approach clearly does not scale to the complexities of an entire renderer.

Our solution to this problem is to perform arithmetic *symbolically*: the backend merely records the desired sequence of operations, postponing evaluation of the kernel for as long as possible. Only once we "peek" inside an array (e.g. in line 6 of the previous example) is it necessary to actually compute its contents. Our GPU backend exploits this using a lazy tracing *just-in-time* (JIT) compilation approach. We use NVIDIA's *Parallel Thread Execution* (PTX) intermediate representation to construct a program in *single static assignment* (SSA) form: for example, the backend operation `GPUArray<float>::mul_()` appends a PTX instruction of the form `mul.f32 $r1, $r2, $r3` that reads its operands and stores the result in a new variable. Thus, a `GPUArray` is only a thin wrapper around an index that references a particular assignment in the SSA intermediate representation.

Each assignment has two reference counters: the first ("external") specifies how many `GPUArray` instances directly point to the associated GPU variable, while the second ("internal") counts how many times it is referenced by other expressions. The assignment is superfluous if both counters reach zero, in which case we remove it from the kernel. To understand the need for two separate counters, consider a function that performs a calculation involving its argument:

```
Point2f square_to_uniform_disk(const Point2f &sample) {
    Float r = sqrt(sample.y());
    auto [s, c] = sincos(2.f * Pi * sample.x());
    return Point2f(s * r, c * r);
}
```

To generate a uniformly distributed point on a disk, this function creates both local variables (`r, s, c`) and temporaries (e.g. the argument to `sincos`). All are used as part of subsequent computations, and their internal reference count is thus positive. When the function returns to the caller, only the return value remains explicitly reachable (for instance, there is no straightforward way of accessing `r` short of recomputing it), which means that the external reference count of all local variables and temporaries is zero. This enables an important optimization: since their contents are no longer directly addressable, these variables don't need to be stored in global memory and can use fast processor registers. Applied to the entire renderer, this optimization significantly increases performance and lowers memory usage.

Note that PTX is only an intermediate representation—a subsequent compiler pass is necessary to generate an executable kernel using the native GPU instruction set SASS. The same pass also performs register allocation and optimizations, such as common subexpression elimination and constant folding. Since this is by far the costliest part of JIT compilation, we cache the resulting kernels and reuse them if the same computation occurs again, which helps when running an iterative algorithm like stochastic gradient descent (compilation typically only occurs once during the first iteration).

Relation to existing frameworks. Our approach is related to tools like TensorFlow [Abadi et al. 2015] and PyTorch [Paszke et al. 2017] but addresses fundamental problems that arise in the context of rendering. Both PyTorch and Tensorflow provide two main operational modes: *eager mode* directly evaluates arithmetic operations on the GPU, which yields excellent performance in conjunction with arithmetically intensive operations like convolutions and large matrix-vector multiplications, both of which are building blocks of neural networks. When evaluating rendering code created from much simpler arithmetic, the resulting memory traffic and scheduling overheads induce severe bottlenecks. An early prototype of Enoki provided a `TorchArray<T>` type that carried out operations using PyTorch's eager mode, and the low performance of this combination eventually motivated us to develop the lazy JIT approach proposed in this article.

The second operational mode requires an up-front specification of the complete computation graph to generate a single optimized GPU kernel (e.g. via XLA in TensorFlow and `jit.trace` in PyTorch). This is feasible for neural networks, whose graph specification is very regular and typically only consists of a few hundred operations. Rendering code, on the other hand, involves much larger graphs, whose structure is *unpredictable*: program execution could jump to almost any part of the system when rendering a complex scene. The full computation graph would simply be the entire codebase (~100K lines of code), which is of course far too big.

Our system was designed to handle the intricacies of physically based rendering and could be interpreted as a middle ground between the two extremes discussed above. Graphs are created on the fly while simulating the process of scattering and transport and tend to be several orders of magnitude larger compared to typical neural networks. They consist mostly of unstructured and comparably simple arithmetic and are lazily fused into optimized CUDA kernels. Since our system works without an up-front specification

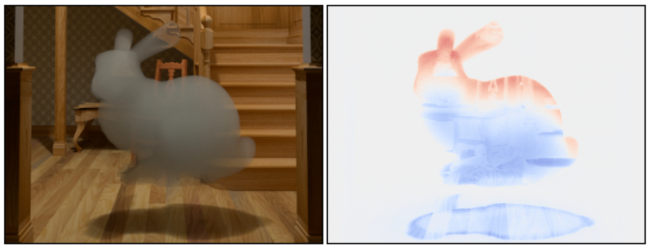


Fig. 3. Visualization of the gradient of an image with respect to the density of a participating medium, computed using forward-mode automatic differentiation (red and blue encode positive and negative values, respectively).

of the full computation graph, it must support dynamic indirection via virtual function calls that can simultaneously branch to multiple different implementations. The details of this are described shortly.

Another related framework is ArrayFire [Yalamanchili et al. 2015], which provides a JIT compiler that lazily fuses instructions similar to our `GPUArray<T>` type. ArrayFire targets a higher-level language (C), appears to be limited to fairly small kernels (100 operations by default), and does not support a mechanism for automatic differentiation. In contrast, Mitsuba 2 emits an intermediate representation (PTX) and fuses instructions into comparatively larger kernels that often exceed 100K instructions.

Autodiff backend. Enoki's last array type, `DiffArray<Value>`, enables transparent forward and reverse-mode differentiation. Similar to `autograd` in PyTorch [Paszke et al. 2017] or Stan [Carpenter et al. 2015], gradient evaluation requires a declaration of relevant inputs followed by a statement that triggers the graph traversal:

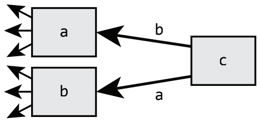
```
// Forward-mode AD:
using Float = DiffArray<float>;
Float in = 1.0f;
set_requires_gradient(in);

auto [out1, out2] = f(in);
forward(in);
float grad1 = gradient(out1),
      grad2 = gradient(out2);
```

```
// Reverse-mode AD:
using Float = DiffArray<float>;
Float in1 = 1.f, in2 = 2.f;
set_requires_gradient(in1);
set_requires_gradient(in2);
Float out = f(in1, in2);
backward(out);
float grad1 = gradient(in1),
      grad2 = gradient(in2);
```

Both forward and reverse-mode AD have useful applications in the context of rendering: the former to visualize gradients for a scene parameter (Figure 3), and the latter to optimize a scene with respect to an objective function involving a rendered image (Figure 1 (c, d)).

A `DiffArray` consists of two parts: a value that is used during the forward pass, and an index that refers to a node in a separately maintained directed acyclic graph capturing the structure of the computation. By default, the index is set to an invalid state indicating that the variable does not participate in AD. Returning to our previous example of the multiplication `c = a * b`, the backend creates a new node `c` (if applicable) referencing the operands:



Edges are *weighted* and store partial derivatives of the operation with respect to its inputs—here, this is simply the product rule. Reverse or forward mode traversal entails a sequence of multiply-accumulate operations to apply the chain rule. A reverse-mode traversal of the above graph triggers two updates: `da += b * dc` and `db += a * dc`.

Automatic differentiation on the GPU. In practice, we generally combine the previous two array types by nesting them, making `DiffArray<GPUArray<float>>` the basic numeric type of our differentiable renderer. The combination of lazy JIT compiler and AD has interesting consequences: computation related to derivatives is queued up along with primal arithmetic and can thus be compiled to into a joint GPU kernel[3], leveraging subexpression elimination and constant folding to further improve efficiency. The performance of our JIT-compiled kernels for differentiable rendering is competitive with hand-written derivative code: in particular, we find that Mitsuba 2 is typically 10-15% faster than *Redner*, the open source implementation of the method by Li et al. [2018a] (Table 1).

	Number of Parameters	Redner	Mitsuba 2
Cornell box	5×3	0.4672 s/it	0.4275 s/it
Textured monkey	$512 \times 512 \times 3$	0.2297 s/it	0.2017 s/it
Textured sphere	$1024 \times 1024 \times 3$	0.1981 s/it	0.1749 s/it

Table 1. Timing comparison against *Redner* when optimizing diffuse appearance model parameters. Special handling of discontinuities was disabled in both renderers to only benchmark the differentiable parts of the problem. Additional information on this comparison can be found in the supplemental material. For benchmarks that include the effects of visibility, please refer to the separate paper by Loubet et al. [2019].

Graph simplification. Rendering algorithms vectorized using Mitsuba 2 use a wavefront tracing approach that tends to be storage intensive even without differentiation—for instance, the basic intersection data structure of Mitsuba 2 stores shape pointers and primitive indices, position, ray distance, wavelengths (if applicable), time, differential geometry, and texture differentials. At 172 bytes per intersection, this yields 1376 MiB for a single wavefront of a low-resolution 512×512 image at 32 samples per pixel. Other data structures for querying BSDFs or light sources are similarly affected.

In a differentiable renderer, these are generally all functions of differentiable scene parameters that influence the rendered image and hence have to be kept in memory to enable the final reverse-mode graph traversal. This makes direct applications of AD to rendering very costly: in scenes with interreflection, it is necessary to record many large data structures in memory, including the computation graphs that were used to create them.

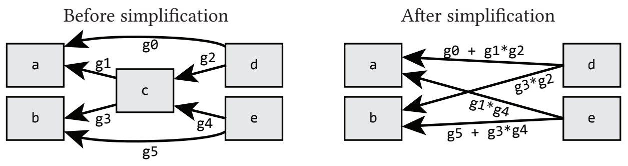
To avoid this problem, Enoki periodically simplifies the computation graph by eagerly evaluating the chain rule at interior nodes to reduce storage requirements. Consequently, our system does not follow a strict reverse- or forward-mode graph traversal, making it an instance of *mixed-mode*, or *hybrid* AD [Griewank and Walther 2008]. When working with differentiable GPU arrays, simplification occurs before each JIT compilation pass. The fundamental operation of the simplification process is known as *vertex elimination* [Griewank and Reese 1991; Yoshida 1987] and collapses an interior node with

[3]For example, if a forward computation evaluates the expression `sin(x)`, the weight of the associated backward edge in the computation graph is given by `cos(x)`. The computation of both of these quantities is automatically merged into a single joint kernel.

	No JIT	JIT
No graph simplification	2.92 s 2.54 GiB	0.19 s 2.43 GiB
Graph simplification	4.45 s 1.86 GiB	0.20 s 1.36 GiB

Table 2. We show the effects of graph simplification and JIT compilation on runtime and memory consumption of Mitsuba 2 when differentiating a rendering of a Cornell Box scene (256 × 256 pixels, 16 samples per pixel) with respect to the complex-valued index of refraction of a rough conductive material assigned to the interior.

d_i in-edges and d_o out-edges, creating $d_i \cdot d_o$ new edges, whose weights are products of the original edge weights. These are then merged with existing edges, if applicable:



Although this operation may increase the density of the graph connectivity if $d_i, d_o > 1$, collapsing such nodes is often worthwhile since it enables later simplifications that can reduce an entire subgraph to a single edge. Compared to direct traversal of the original graph, simplification increases the required amount of arithmetic in exchange for lower memory usage.

In conjunction with the GPU backend, this optimization is particularly effective: removals often target nodes whose primal computation has *not yet taken place.* Since edge weights of collapsed nodes are no longer directly reachable, they can be promoted to cheap register storage.

We found that the order of collapse operations has a significant effect on the efficiency and size of the resulting kernels. Unfortunately, propagating derivatives in a way that results in a minimal number of operations is known to be NP-hard [Naumann 2007]. We use a greedy scheme that organizes nodes in a priority queue ordered by the number of edges $d_i \cdot d_o$ that would be created by a hypothetical collapse operation, issuing collapses from cheapest to most expensive until the cost exceeds an arbitrary threshold that we set to 10 edges. Note that removal of a node changes the cost of adjacent nodes due to the creation of new edges, hence their position in the priority queue must be updated. Table 2 shows the effect of both JIT compilation and graph simplification on runtime and memory consumption.

Custom data structures. Modern renderers use auxiliary data structures to facilitate communication between different system components. This includes surface and medium interactions, data structures for direct illumination and BSDF sampling, and so on. Mitsuba 2 uses the same overall approach, except that these are now specified in generic form to permit retargeting. A definition usually begins with a series of statements that compute the necessary types, followed by the declaration of matching fields. For example, the renderer's surface intersection data structure roughly looks as follows:

```
template <typename Point3f> struct SurfaceInteraction {
    using Float    = value_t<Point3f>;
    using Vector3f = Vector<Float, 3>;
    using Frame3f  = Frame<Vector3f>;
    using UInt32   = uint32_array_t<Float>;
    using Shape3f  = replace_scalar_t<Float, const Shape<Float> *>;

    Float t;          // ray distance
    Point3f p;        // position
    Vector3f wi;      // incident direction
    Frame3f sh_frame; // shading coordinate frame
    UInt32 prim_id;   // intersected primitive (e.g. triangle ID)
    Shape3f shape;    // pointer to Shape<..> instance
    /// ...
};
ENOKI_STRUCT(SurfaceInteraction, t, p, wi, sh_frame, prim_id, shape)
```

Here, `value_t<T>` extracts the value underlying an array `T`, and `replace_scalar_t<T, X>` returns an array of the same structure as `T`, but using a representation based on the (scalar) type `X`. For instance, `uint32_array_t<T>` is an alias for `replace_scalar_t<T, uint32_t>` and returns an unsigned integer version of the argument. The macro on the last line allows certain Enoki operations to be applied to the data structure itself, causing them to recursively propagate through all fields.

This type of recursive lifting greatly facilitates tasks such as switching data structures to a *Structure of Arrays* (SoA) representation. For this, we can simply substitute a vectorized floating point type at the root level (e.g. `Array<float, 16>` or `GPUArray<float>`), letting the type system do the remaining work.

Masks. Vectorized algorithms process multiple elements at once, hence standard language features like **if** statements are unsuitable for modeling their control flow[4]. Comparisons and other logical operation involving Enoki arrays thus produce *masks* (arrays of boolean values), which support arbitrary nesting, broadcasting, and are realized using bit-efficient hardware registers whenever possible. Masks are often used to select from one of two expressions using the ternary conditional operator `select(mask, expr_true, expr_false)`.

Consider the following example usage of masks to create a two-sided material that samples one of two BRDFs depending on whether the incident direction `wi` lies on the top or bottom side of a surface:

```
using Mask = mask_t<Float>;
Mask active_top = wi.z() > 0.f;
BSDFSample3f bs;
bs[ active_top] = brdf_top->sample(/* ... */);
bs[!active_top] = brdf_bot->sample(/* ... */);
```

Here, `mask_t<T>` returns the mask type associated with an array `T`, and the last two lines are conditional assignments that automatically propagate through the structure's fields. Although correct, this example is unsatisfactory in two ways: first, the `sample()` calls do not inform the callee if entries are handled elsewhere and could have been skipped, which may generate unnecessary memory traffic (e.g. texture lookups). Secondly, one of the function calls could be elided if all directions lie on the same side. Detecting this case requires a horizontal reduction.

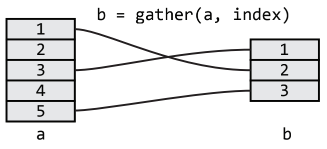
[4] A compiler frontend like ISPC [Pharr and Mark 2012] has an advantage here, since it can automate the conversion of conditional statements to masks.

Horizontal operations. Horizontal operations involve a reduction over one or more dimensions, returning a tensor of lower rank. Examples are logical reductions that can be applied to masks (`all()`, `any()`, `none()`, `count()`, etc.) and arithmetic reductions for standard arrays (horizontal sums, products, maxima, etc.). Their realization depends on the target: in scalar mode, horizontal operations simply return their argument as no reduction is needed. On CPU vector targets, they are implemented using a sequence of vertical operations and are hence slightly more expensive than normal arithmetic.

When working with GPU arrays, horizontal reductions are best avoided whenever possible. Vertical operations are scheduled asynchronously and execute concurrently on the entire chip, which is key to their efficiency. In contrast, horizontal operations create synchronization barriers that require all queued computation to finish before the reduction can take place. They are also often unnecessary: for example, `any(mask)` is almost certainly `true` if `mask` is a large array and the underlying condition is satisfied with a nonzero probability. For this reason, Enoki provides logical reductions with a default choice (e.g. `all_or<true>`) that take precedences when targeting the GPU. Integrating these improvements into the previous example addresses the discussed flaws. Note that a mask `active` is generally also provided by the calling function when masks are consistently maintained.

```
Mask active_top = wi.z() > 0.f && active,
     active_bot = wi.z() < 0.f && active;
if (any_or<true>(active_top))
    bs[active_top] = brdf_top->sample(/* ... */, active_top);
if (any_or<true>(active_bot))
    bs[active_bot] = brdf_bot->sample(/* ... */, active_bot);
```

Scatters and gathers. Enoki arrays provide natural scatter, gather, and atomic scatter-add primitives, which are essential for texture and volume lookups or splats into an image buffer. The main challenge of these operations is they constitute a special case during automatic differentiation. Consider the following operation, which selects a subset of an input array:



Here, reverse propagation of a derivative `db` into `da` requires a suitable `scatter_add()` operation. Analogously, scatters turn into gathers during reverse-mode AD. Our differentiable array backend recognizes these operations and inserts a special type of edge into the graph to enable the necessary transformations. In the context of rendering, these edges e.g. encode the sparsity pattern of a large Jacobian matrix that relates pixels in the rendered image to texels or voxels in the scene definition. One current limitation of Enoki is that such special edges cannot be merged into ordinary edges during graph simplification. Handling this case could further reduce memory usage and is an interesting topic for future work.

Method dispatch. Indirect branches are a common feature of rendering code, for example to sample the BSDF of an intersected shape:

```
SurfaceInteraction3f si = /* ... */;
BSDFSample3f bs = si.bsdf->sample(si, sample);
```

In a scalar program, this operation represents an ordinary virtual function call that requires no special handling. Vectorization, however, turns `si.bsdf` into an array of pointers, that potentially refer to many different BSDF instances. Enoki intercepts such function calls by overloading the "`->`" operator and dispatches them using one method call per unique pointer. The details of this step vary depending on the target: when vectorizing for CPUs (e.g. Intel AVX512), we repeatedly extract a nonzero pointer from the array (using the `vpcompressq` instruction, if available) and zero-fill all entries of the same value. We then perform an indirect branch, providing the callee with a mask of the active entries.

To handle indirect branches involving GPU arrays, we issue a horizontal operation to extract a list of unique pointers along with a list of indices per pointer specifying what elements of the array refer to it. This is realized using a parallel radix sort and run length encoding via NVIDIA's `cub` library [Merrill 2015]. Following this, we gather the argument values associated with a particular instance, perform the function call, and then scatter the results back into the output array, which looks roughly as follows:

```
// foreach (bsdf, indices) in partition(si.bsdf):
BSDFSample3f temp = bsdf->sample(gather(si,     indices),
                                 gather(sample, indices));
scatter(bs, temp, indices);
```

All three steps merely enqueue computation to be executed at a later point. Enoki uses TMP to automatically rewrite the earlier virtual function call into this form, hence no target-specific code is necessary. For differentiable arrays, Enoki automatically propagates derivative information through virtual function calls.

Note that method calls on the GPU typically produce computation involving arrays of different sizes—for instance, n_1 rays of a large wavefront may have interacted with one type of BRDF, n_2 with another, and so on. Sampling these BRDFs consequently leads to arithmetic operations involving arrays of size n_1, n_2, etc., which we track in separate queues of the JIT compiler. The operations in each queue can execute in parallel to work in other queues, and we therefore flush queues into independent CUDA streams. Cached kernels can be reused even if they were generated for a computation involving a different array size (in other words, size is a runtime kernel parameter). In the context of rendering, this means that the number of wavefront elements following a particular path of execution can change over time without requiring re-compilation.

Mathematical support library. Enoki includes an extensive mathematical support library with types that are relevant for physically based rendering, such as complex numbers, matrices, quaternions, and related operations (determinants, matrix, inversion, etc.). A set of transcendental and special functions supports real, complex, and quaternion-valued arguments in single and double-precision using polynomial or rational polynomial approximations, generally with an average error lower than 1/2 ULP (unit in the last place) on their full domain. These include exponentials, logarithms, and trigonometric and hyperbolic functions, as well as their inverses. Our system also provides real-valued versions of error function variants, Bessel functions, and elliptical integrals.

Importantly, all of this functionality is realized using the abstractions of Enoki, which means that it transparently composes with vectorization, the lazy JIT compiler, automatic differentiation, etc.

Language bindings. Mitsuba 2's focus on types has another less obvious benefit: they provide a rich description of structure and memory layout that enables high-quality language bindings. We extended `pybind11` [Jakob et al. 2017]—itself based on metaprogramming—to create bindings from one-line declarations of the form

```
module.def("func", &func);
```

A metaprogram then analyzes the function's type to synthesize code that automatically converts function arguments and return values. We used this approach to create fine-grained Python language bindings of all major rendering system components for CPU (scalar and vectorized) and GPU (vectorized differentiable) targets, enabling prototyping of complete rendering algorithms, e.g., using interactive Jupyter notebook sessions. Enoki arrays also support implicit bidirectional conversion to other array libraries, such as NumPy and PyTorch. The latter allows the renderer to be used as a differentiable layer in a larger computation graph realized using PyTorch.

Development and challenges. We used the abstractions of Enoki to implement a complete rendering system that currently includes a path tracer, volumetric path tracer, and adjoint light tracer (all with multiple importance sampling). Our system supports standard light sources (point / area / directional lights and environment maps) and both specular and rough microfacet BRDFs with visible normal sampling [Heitz and d'Eon 2014] for conductors, dielectrics, and plastic-like materials. Each component of the renderer is compiled as a plugin (i.e. a shared library) that contains multiple instantiation of an abstract implementation. Please refer to the supplemental material to see a longer example of typical Mitsuba 2 source code, specifically an annotated path tracer with multiple importance sampling.

The renderer is currently spectral by default and uses Monte Carlo sampling to integrate over continuous wavelengths spanning the 360 to 830nm range using 4 sampled wavelengths per ray. A monochromatic mode is also available, which is mainly used in automated tests, and for debugging. The transformation from RGB values (e.g. in texture maps) to reflectance spectra relies on the vectorized spectral upsampling model of Jakob and Hanika [2019].

Three backends are available for ray tracing: scalar and packet tracing on the CPU either use a builtin kd-tree or Embree [Wald et al. 2014], and ray tracing on the GPU relies on OptiX [Parker et al. 2010]. The builtin kd-tree is useful for debugging, e.g. to render an image in double precision which neither Embree nor OptiX support. All results shown this article were created using Embree and OptiX.

During development, we encountered two standard constructions that require special precautions. First, sampling code often relies on Newton or Newton-Bisection iterations to numerically invert cumulative distribution functions (CDFs), whose inverse does not have a closed-form expression. The iteration's stopping criterion `if (all(converged))` poorly interacts with the GPU backend, since this is a horizontal operation that would serialize the computation at every step. In such cases, we determined[5] suitable fixed upper bounds for the iteration count that we use instead. A related example are discrete CDFs inverted using a binary search, e.g., to pick rows and columns of an environment map. Here, a tight bound is given by $\lceil \log_2(N-2) \rceil + 1$, where N is the number of entries. Following this change, all Newton iterations or binary search steps are unrolled into the current kernel.

For gradient-based optimization, we had to ensure that certain operations that normally run as a pre-process step before rendering begins are recorded in the computation graph. An example is the computation of smooth shading normals from vertex positions. One is a function of the other, hence it is important to accurately capture their relationship during optimization. Our system provides reconstruction filters (e.g. a Gaussian or Mitchell-Netravali filter), whose contribution to the image is differentiable with respect to the position of a sample. This is necessary e.g. to optimize the shape of caustics due to chains of purely specular transport.

Our optimization examples were all developed in Python scripts that begin by loading an XML scene that specifies the starting point of the optimization. The scene can be queried for differentiable parameters, some of which are subsequently connected to an optimizer (SGD, Adam, etc.) along with custom loss functions.

4 APPLICATIONS

We demonstrate the effectiveness and simplicity of our approach on several challenging applications. Please refer to the supplemental material for details on timing and optimization parameters.

4.1 Polarized light transport

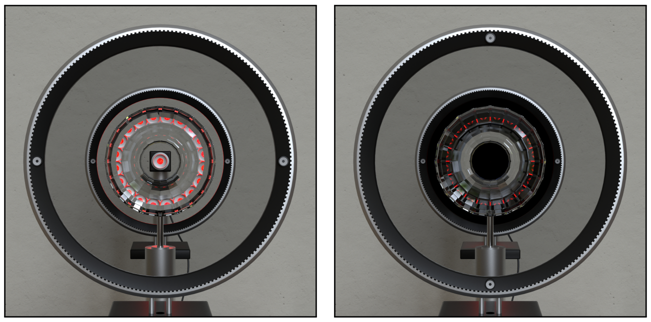
The polarization state of a beam of light is normally described using a 4-dimensional quantity known as the *Stokes vector*, which parameterizes the elliptical shape of the associated transverse oscillation. When the beam interacts with a surface, this polarization state changes, and the details of this change can be encoded in a 4×4 *Mueller matrix* [Collett 1993; Wilkie and Weidlich 2012].

The distinction between Stokes vectors and Mueller matrices unfortunately introduces an asymmetry that has the potential of significantly complicating the structure of a rendering system like ours [Jarabo and Arellano 2018; Mojzík et al. 2016]. For instance, spectra are normally used to represent emission, reflectance, and importance in non-polarized renderers, but polarization requires us to change them to Stokes vectors in the former case, and Mueller matrices in the latter two cases. Each matrix is only valid with respect to specific incident and outgoing reference coordinate frames that need to be correctly aligned when sampling or evaluating materials.

In Mitsuba 2, we made the decision to represent all of the above quantities using Mueller matrices, which leads to some unnecessary arithmetic but allows for a simpler API (in particular emitters and sensors are symmetric). We furthermore compute the necessary reference frames on demand, rather than storing them.

Using these simplifications, polarized algorithms are almost equivalent to their classical counterparts, the main difference being some additional care regarding the alignment of coordinate frames when evaluating reflectance models. Assuming that this detail is considered during development, type transformation can easily and automatically instantiate polarized and non-polarized forms of rendering algorithms, reflectance models, etc. Since the signature of essentially any function or data structure quantifying light or reflectance requires modifications, such a change would be tedious in a system that provides no assistance (e.g. Mitsuba 0.6).

[5]There are "only" 4 billion single precision floating point values, and it is normally possible to test all of them in a few minutes.



(a) Front polarizer at 0° (b) Front polarizer at 90°

Fig. 4. Close-up rendering through the optical setup from the scene in Figure 1 (a): a dielectric object is placed between two linear polarization filters. In a cross-polarized configuration, the filters block transmission, while refraction yields elliptically polarized light that remains visible.

We also modified various reflectance models in Mitsuba 2 to correctly account for polarization during scattering. In particular, conductors and dielectrics use the complex-valued form of the Fresnel equations when polarization is active. We also support the pBRDF model by Baek et al. [2018] that includes both a polarized specular component based on microfacet theory and a polarized diffuse component. Finally, our system provides BSDFs of standard optical elements such as linear polarizers or phase retarders like quarter-wave plates that are useful for prototyping optical experiments.

Figure 4 shows renderings of a dielectric object mounted between two linear polarizers generated with Mitsuba 2's path tracer. The two orthogonal polarizers block transmitted light completely, while refraction through the glass introduces elliptical polarization that is able to pass through the second filter.

Polarization could in principle be composed with differentiation to solve inverse problems. This is an interesting area for future research, for instance to improve geometric reconstruction from photographs taken with filters, while accounting for polarized interreflection.

4.2 Coherent MCMC sampling

Vectorization using Mitsuba 2 enables simultaneous generation of multiple light paths using modern SIMD instruction sets. Unfortunately, a naively vectorized path tracer only obtains marginal performance improvements on most scenes, due to the algorithm's fundamental lack of coherence (Figure 5a). Metropolis-type rendering techniques [Kelemen et al. 2002; Veach 1997] explore nearby light paths, improving coherence somewhat, but their sequential nature makes them challenging to vectorize. In this section, we discuss two MCMC schemes that are designed to produce coherent workloads, better leveraging the vector capabilities of our system.

Coherent Pseudo-Marginal MLT. We propose a novel MLT method that improves coherence by replacing point evaluations with coherent bundles of light paths. A key assumption of our approach is that nearby points in primary sample space [Kelemen et al. 2002] correspond to paths that undergo similar control flow, and which reference nearby regions of memory. The core idea of our method is to sample a modified target function π that is the result of convolving the path contribution function f with a Gaussian kernel G:

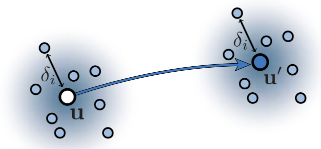
$$\pi(\mathbf{u}) = \int_{\mathcal{U}} G(\boldsymbol{\delta}) \cdot f(\mathbf{u} - \boldsymbol{\delta})\, d\boldsymbol{\delta}, \tag{1}$$

where $\mathcal{U}$ is primary sample space. Intuitively, blurring the integrand should facilitate exploration, even in the presence of discontinuities e.g. due to visibility changes.

Target functions used in MCMC are normally deterministic: for example, given a particular primary sample space position $\mathbf{u}$, evaluating the path contribution $f(\mathbf{u})$ will always produce the same result. A key observation of pseudo-marginal MCMC [Andrieu and Roberts 2009] is that it is possible to retain the fundamental properties of MCMC when π is replaced by an unbiased estimator. We use this insight to replace evaluations of the target function $\pi(\boldsymbol{u})$ with a Monte Carlo estimate of the convolution:

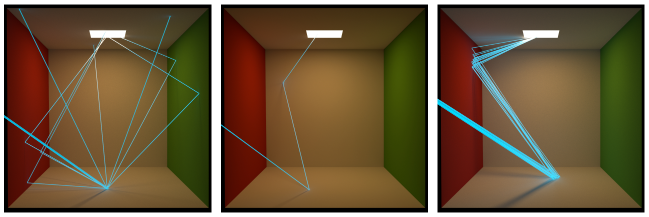
$$\pi(\mathbf{u}) = \mathbb{E}_{G(\boldsymbol{\delta})}\left[f(\mathbf{u} - \boldsymbol{\delta})\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{u} - \boldsymbol{\delta}_i). \tag{2}$$

where the offsets $\boldsymbol{\delta}_i$ are drawn from a multivariate normal distribution, as shown in the following visualization:



We use a standard Metropolis iteration to sample the center position $\boldsymbol{u}$ and apply vectorization to evaluate all path contributions $f(\mathbf{u} - \boldsymbol{\delta}_i)$ at once. We thus name our method *Coherent Pseudo-Marginal MLT* (CPMMLT). Setting N to the maximum vector instruction width (e.g. $N = 16$ for AVX512), sampling and evaluation of these paths is easily vectorized in Mitsuba 2. A qualitative comparison against paths evaluated by a vectorized path tracer or PSSMLT is shown in Figure 5. Weighting the particles by $f(\boldsymbol{u} - \boldsymbol{\delta}_i)/\pi(\boldsymbol{u})$ yields unbiased samples of the original distribution that can be used to reconstruct the non-blurred image.

Multiple-Try Metropolis. Our approach is related to work by Segovia et al. [2007a; 2007b], who integrate Multiple-Try Metropolis (MTM) [Liu et al. 2000] into a vectorized MLT renderer. Unlike standard Metropolis-Hastings random walks, MTM generates a set of N proposals to choose from at each iteration ("trial set"). After drawing



(a) Vector Path Tracer (b) PSSMLT (c) CPMMLT (Ours)

Fig. 5. Qualitative comparison of ray distributions generated using three different methods: **(a)** paths in a packet path tracer decohere at the first bounce, thus losing the benefits of vectorization. **(b)** PSSMLT's exploration is more coherent, but evaluates only one ray at a time. **(c)** our method evaluates coherent bundles of e.g. 16 rays at each iteration.

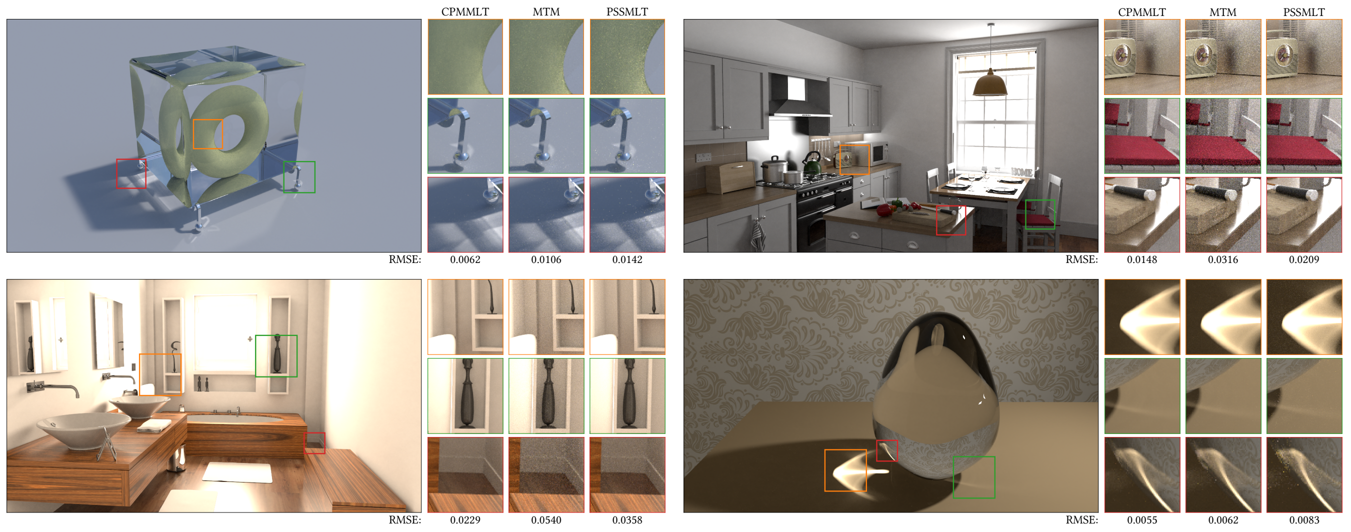
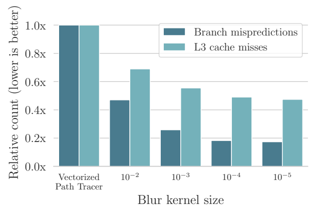

CPMMLT
MTM
PSSMLT
RMSE: 0.0062 0.0106 0.0142
RMSE: 0.0148 0.0316 0.0209
RMSE: 0.0229 0.0540 0.0358
RMSE: 0.0055 0.0062 0.0083

Fig. 6. Equal-time comparisons on the TORUS, KITCHEN, SALLE DE BAIN and GLASS EGG scenes. Convergence of our method is superior to MTM and PSSMLT. The Root Mean Squared Error (RMSE) numbers shown are computed on the entire image for each method and averaged over three runs.



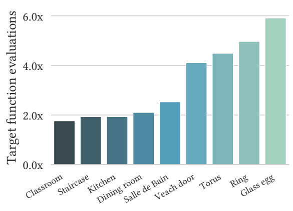
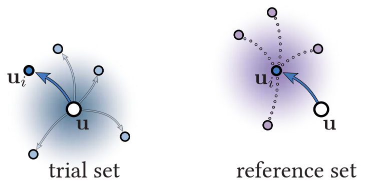


Fig. 7. Left: At equal run time on the STAIRCASE scene, the average number of cache misses and branch mispredictions decrease when working with coherent bundles of rays. The kernel's standard deviation (horizontal axis) provides a trade-off between exploration and coherence. Textures were scaled by a factor of 10 to simulate a production use case where scene data does not fit in cache. Right: vectorized evaluation of coherent ray bundles allow our method to compute 2 to 6 times more rays per second than a PSSMLT sampler. Scenes with lower geometric complexity benefit the most from vectorization.

one proposal $\mathbf{u}_i$ from the trial set, a "reference set" with N states $\mathbf{u}'_i$ is sampled around $\mathbf{u}_i$. The new state's $\mathbf{u}_i$ acceptance probability is computed using a generalized Metropolis-Hastings ratio.



In our evaluations, we compare to an improved variant of MTM with *waste recycling* [Murray 2007, Section 3.2.3] that enables the use of intermediate samples in the reconstruction of the final image.

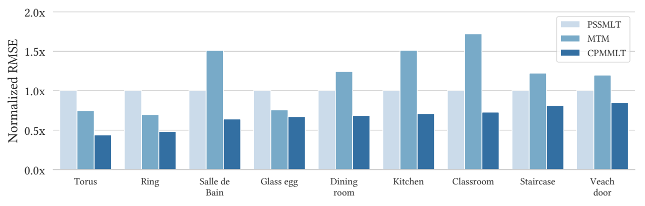
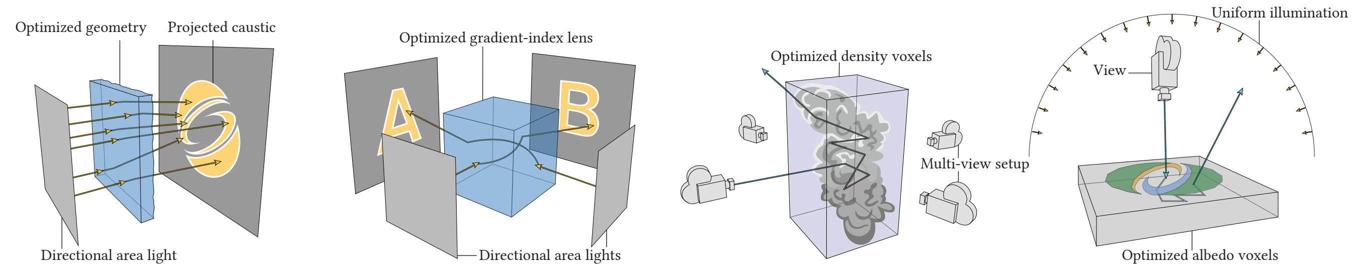


Fig. 8. We render a variety of scenes from Bitterli's repository [2016] at equal time with our sampler and baselines. To simplify comparison across scenes, we report Root Mean Squared Error relative to the error obtained by PSSMLT on each scene (lower is better).

Results. We benchmark our method against (scalar) PSSMLT and vectorized waste-recycled MTM on a variety of scenes [Bitterli 2016]. All methods were implemented in Mitsuba 2. We select the parameters for each MLT method using a grid search. Since they are not very scene-dependent, we use the same parameters for all experiments.

Our experiments show that CPMMLT's improved coherence results in faster convergence at equal time, as shown in Figures 6 and 8. Experiments were run on 28-core machines supporting 16-wide SIMD (Intel Xeon Gold 6132, 2.60GHz). Error numbers were averaged over 3 equal-time runs.

Assuming that the Gaussian kernel's standard deviation is not too large, our method evaluates positions in primary sample space which map to light paths with matching control flow, virtual function calls and memory access patterns. Figure 7 (left) plots branch mispredictions and L3 cache misses as a function of G's standard deviation, confirming this claim: coherence increases considerably compared to a naive path tracer when using our sampling method. We found that CPMMLT evaluates 2 to 6 times more rays per second than PSSMLT, depending on the scene and its geometric complexity (Figure 7, right).



(a) Caustic design: surface displacement **(b)** Caustic design: gradient-index optics **(c)** Volume density reconstruction **(d)** Textured translucent slab

Fig. 9. We showcase four different material design and reconstruction applications that optimize **(a)** a refractive height field focusing collimated illumination into a desired image on a target surface. **(b)** a cube with spatially varying index of refraction that propagates light along curved rays, encoding two separate images for illumination arriving from perpendicular directions. **(c)** a heterogeneous medium with multiple scattering that approximates reference imagery from multiple camera positions. **(d)** heterogeneous parameters of a dielectric slab with subsurface scattering, whose appearance approximates a reference image.

4.3 Caustic design

Mitsuba 2's differentiable rendering capabilities greatly facilitate material design applications. In this section, we present two methods for computational caustics that optimize either the geometry of a glass slab or the index of refraction of a gradient-index lens so that they project a desired image onto a target surface. The corresponding experimental setup is shown in Figure 9 (a) and (b).

Surface displacements. This first problem has been studied by Papas et al. [2011], who used a decomposition of Gaussian kernels and Yue et al. [2014], who solve a sequence of Poisson problems to construct a smooth height field. Schwartzburg et al. [2014] used a tailored optimization formulation based on optimal transport.

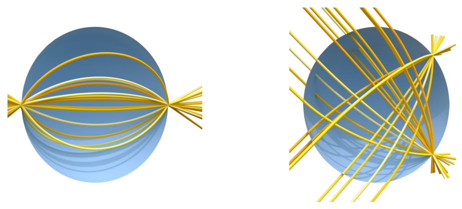
While our current results do not match the quality of a purpose-built system in terms of contrast and precision, its ease of use is appealing: the only requirement is a suitable forward simulation, which can then be optimized using a variant of gradient descent. The method's generality makes it immediately applicable to broader settings. For instance, in Figure 10 (c), the optimization generates geometry that blends primary colors in the right proportions to create a color image.

We render caustics using a standard light tracer. For optimization, differentiable image reconstruction filters are crucial to capture the relationship between the geometry and brightness and position of the various parts of a caustic. Figure 10 (a-c) shows three results and intermediate optimization states for displacement-based caustics.

Gradient-index optics. In materials with a varying index of refraction, light travels along curved rays according to the Eikonal equation. Expressed as a second-order ODE [Sharma et al. 1982], it relates the change in position to the gradient of the refractive index:

$$\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2} = n(\boldsymbol{x})\,\nabla n(\boldsymbol{x}). \qquad (3)$$

Gradient-index optics exploit this effect to create lenses that lack the typical aberrations of spherical lens elements. For instance, the (mostly theoretical) Maxwell fish-eye lens with a radially symmetric index of refraction ($\eta(r) = {}^{1}/_{1+r^2}$) or the Luneburg lens ($\eta(r) = \sqrt{2-r^2}$) image a point onto an antipodal point or collimate it, respectively:



Fabrication of materials with a varying index of refraction is an active area of research [Nguyen et al. 2017] that could one day reduce the cost of current aspherical optics. Here, our differentiable renderer already provides a helpful tool for optimizing the properties of such a material based on a user-specified objective function. Figure 10 (d–e) shows two caustics that are simultaneously projected by a gradient-index cube illuminated from perpendicular directions. Note that the discrete appearance of intermediate optimization steps is due to the trilinear interpolation of refraction values, which causes piecewise constant gradients in the ODE in Equation 3.

Integrating a differentiable ODE solving step in our system is easy—we reproduce the central solver loop used to create this result:

```
1  Point3f p_out; Vector3f v_out;
2  Mask active = true;
3  for (size_t i = 0;; ++i) {
4      auto [ior, ior_grad] = evaluate_ior(p, active);
5      Vector3f v_half = fmadd(.5f * step_size * ior, ior_grad, v);
6      Point3f p_next = fmadd(step_size, v_half, p);
7      Mask escaped = active && !is_inside(p_next);
8      active &= !escaped;
9      p_out[escaped] = p; v_out[escaped] = v;
10     if (i >= 2.f / step_size && none(active))
11         break;
12     p = p_next;
13     v = fmadd(half_step * ior, ior_grad, v_half);
14 }
```

Here, lines 5, 6, and 13 are the Leapfrog discretization of the Eikonal equation (3), where `fmadd` denotes a fused multiply-addition operation. Lines 7–9 keep track of which lanes have exited, saving the final state when rays leave the material. Line 10 is the stopping condition, written is such a way that the potentially costly horizontal reduction `none(active)` is skipped as long as it is likely that at least one ray remains inside.

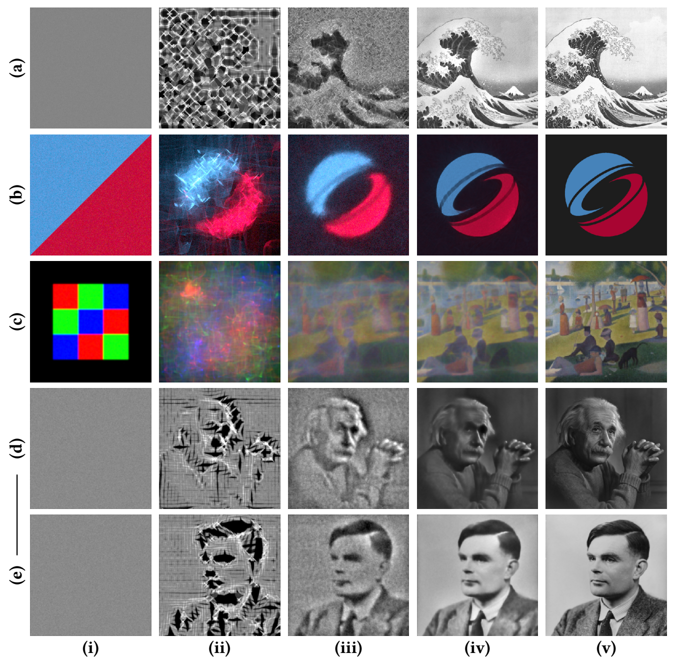


Fig. 10. We optimize **(a–c)** refractive height fields to focus collimated light into a desired image (Figure 9a), and **(d–e)** the spatially-varying index of refraction of a *single* gradient-index lens to project different images when illuminated from two incident directions (Figure 9b). **(i)** starting from a uniform solution that simply refracts light through, **(ii–iii)** the optimization quickly approximates the main features of the target image **(v)**. We render the final state **(iv)** after adjusting the emitters' intensity, which was not part of the optimization.

We optimize both types of caustics using gradient descent with momentum and a multiresolution approach. For details on the optimization procedure, please see our supplemental text document.

4.4 Heterogeneous Participating Media

Finally, we apply our system to the solution of several challenging inverse volume rendering problems. Volumetric light transport in computer graphics generally relies on the *radiative transfer equation* (RTE), which models the material as a suspension of unresolved scattering and absorbing particles [Chandrasekhar 1960]. The integral form of RTE is given by

$$L(\mathbf{x},\boldsymbol{\omega}) = \int_0^\infty \mathrm{Tr}(\mathbf{x},\mathbf{x}_t) \int_{S^2} \sigma_s\, L(\mathbf{x},\boldsymbol{\omega}')\, f_p(\boldsymbol{\omega},\boldsymbol{\omega}')\,\mathrm{d}\boldsymbol{\omega}'\,\mathrm{d}t, \tag{4}$$

where $L(\mathbf{x},\boldsymbol{\omega})$ is the radiance in direction $\boldsymbol{\omega}$ at position $\mathbf{x}$. We define $\mathbf{x}_t = \mathbf{x} + t \cdot \boldsymbol{\omega}$ as the position at distance t on the ray $(\mathbf{x},\boldsymbol{\omega})$ and the *transmittance* as $\mathrm{Tr}(\mathbf{x},\mathbf{x}_t) = \exp(-\int_0^t \sigma_t(\mathbf{x}_t)\,\mathrm{d}t)$. The extinction coefficient σ_t is the sum of the absorption σ_a and scattering coefficient σ_s and quantifies the loss in radiance along a ray due to absorption and out-scattering. For simplicity, we assume that σ_t does not vary spectrally. The *phase function* $f_p(\boldsymbol{\omega}',\boldsymbol{\omega})$ models the angular distribution of scattered directions. The above integral can be estimated using *volumetric path tracing*, which involves alternating steps of sampling the distance to the next scattering event proportional to the transmittance, and choosing a scattered direction according to the phase function.

Differentiable volumetric path tracing. In the following applications, we wish to minimize the pixel-wise difference between a reference image and an image rendered using a differentiable volumetric path tracer. The main challenge here is to sample the free-flight distance in a differentiable way. We initially experimented with delta tracking [Woodcock et al. 1965], which uses a form of rejection sampling to sample either virtual or real medium interactions. While this approach initially appears non-differentiable, it is possible to use the formulation of Galtier et al. [2013] to move the gradient into the integral before introducing the discrete rejection sampling step involving virtual particles [Tregan et al. 2019].

However, the potentially unbounded number of iterations make delta tracking a poor fit for our GPU-based differentiable renderer, resulting in incoherent execution paths and low GPU utilization. Therefore, we resort to *ray marching*, which steps through the medium in regular steps and results in more coherent code, at the cost of introducing a small amount of bias into the rendered image. Even when using ray marching, we must be careful to ensure differentiability. The distance sampling step could sample a position outside of the volume's bounding box, or an interaction with a surface inside of the medium. While this is no problem when rendering normal images, issues arise during a direct application of AD to the estimator. Importance sampling a distance according to a desired density $p(x)$ can be interpreted as a change of variables in primary sample space involving a transformation T with Jacobian determinant $1/p(x)$. Furthermore, all terms depend on scene parameters θ.

$$\int_\Omega f_\theta(x)\,\mathrm{d}x = \int_{\mathcal{U}} \frac{f_\theta(T_\theta(u))}{p_\theta(T_\theta(u))}\,\mathrm{d}u \tag{5}$$

When the free-flight distance e.g. leaves the bounding box, the functions T and p become discontinuous, and it is no longer legal to differentiate under the integral sign. The solution is to differentiate *before* transforming to an integral over primary sample space:

$$\nabla_\theta \int_\Omega f_\theta(x)\,\mathrm{d}x = \int_\Omega \nabla_\theta f_\theta(x)\,\mathrm{d}x = \int_{\mathcal{U}} \frac{\nabla_\theta f_\theta(T_\theta(u))}{p_\theta(T_\theta(u))}\,\mathrm{d}u \tag{6}$$

This approach was also used by Khungurn et al. [2015]. Enoki provides a function `detach()` that removes a node from the computation graph, which we use to disable gradient computation for the functions T and p while sampling free-flight distances. In the remainder of this section, we use our differentiable volumetric path tracer to solve several different inverse problems.

Volume density reconstruction. A differentiable path tracer can be used to reconstruct medium densities from captured images with multiple scattering. We demonstrate the feasibility of this idea using a synthetic smoke plume rendered in front of a black background (constructing a calibrated system for real-world volume acquisition is beyond the scope of this paper).

We use a multiresolution approach as in Section 4.3 to robustly optimize the high-resolution density grid. Several intermediate steps of the optimization are shown in Figure 11. The density is constrained

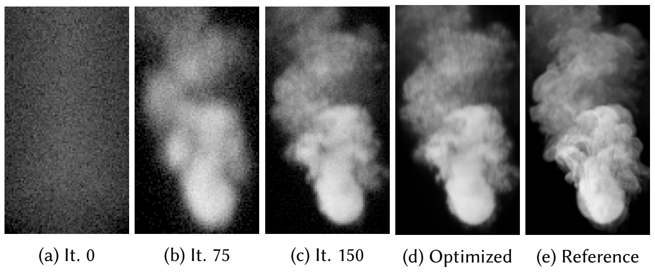


Fig. 11. Reconstruction of a heterogeneous smoke volume. The images show intermediate optimization results, as well as a higher-quality rendering of the final optimized smoke and ground-truth density.

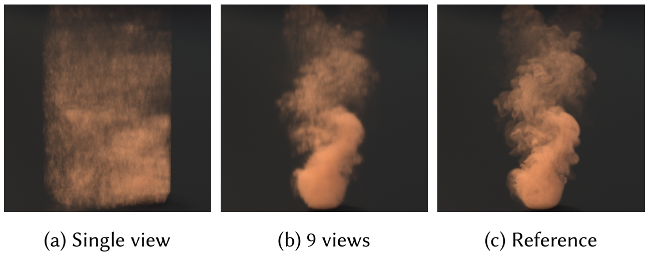


Fig. 12. Comparison of a single-view volume reconstruction and a multi-view approach. The single-view reconstruction suffers from artifacts when the medium is observed from a novel viewpoint.

to lie between zero and one, and we use a L_1 loss function. A single-view reconstruction results in artifacts (Figure 12a), and multiple views are thus needed to sufficiently constrain the medium density. We optimize the medium to match nine different reference views, which addresses this problem (Figure 12b).

Textured translucent slab. With recent 3D printers capable of printing colored ink mixtures, there has been increased interest in optimizing the appearance of 3D printed objects. Since the printed material is translucent, accurately reproducing a desired surface texture is non-trivial due to the effect of multiple scattering. While several optimization methods exist [Elek et al. 2017; Sumin et al. 2019], they build on complex and highly problem-specific optimization routines. Within our framework, we can solve similar problems without the need for specialized solvers. We demonstrate this in a slightly simplified setting, in which we assume the medium's extinction coefficient to be homogeneous and monochromatic. We optimize the heterogeneous albedo values of a medium contained within a dielectric boundary, illuminated by a uniform environment emitter (Figure 9 (d)). The real fabrication setting is more complex, but we believe that this example nevertheless shows the flexibility of our system to solve a variety of inverse problems.

In Figure 13 we show how our optimization reduces the differences between the scattering volume and a diffuse surface with an image texture. We compare against a naive solution that also constitutes the starting guess of our optimization. To initialize it, we first render a semi-infinite medium using a range of single scattering albedo (σ_s/σ_t) values, tabulating the resulting overall albedo of the material. This table can then be inverted to map colors from the target image to medium coefficients, which we extrude along the

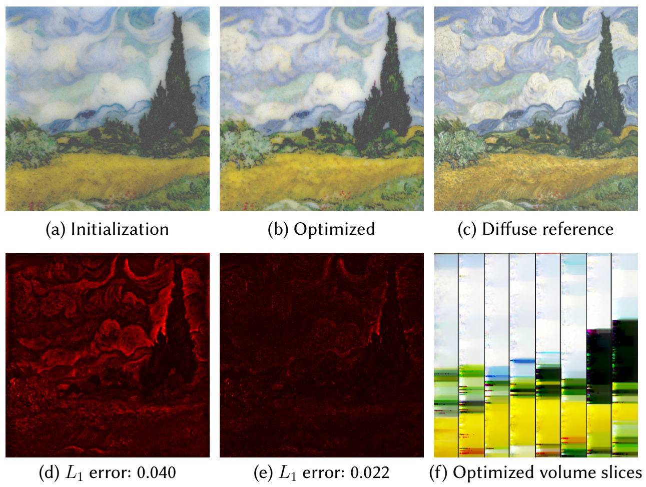


Fig. 13. Comparison of a naive voxel color assignment **(a)** to the result after optimization **(b)**, with L_1 error maps **(d), (e)** compared to the reference **(c)**. In **(f)** we show different vertical slices through the optimized slab.

depth of the slab (only coloring the top layer of voxels would lead to significant color reproduction errors). We then render differentiable 128^2 images with 64 scattering events within a voxel grid of size $256 \times 256 \times 64$ and use them to optimize the medium parameters. Compared to the naive solution, which has relatively low contrast (Figure 13), our method is able to account for the influence of material variation in the neighborhood of a voxel, resulting in better agreement to the reference.

5 CONCLUSION

Physically based rendering is the result of a complex interplay involving countless different system components. Similar to how a photon can interact with distant parts of a large and detailed scene, program execution in a renderer tends to take twisty paths through immense codebases, whose size is measured in multiple hundred thousand lines of code. But simply rendering an image is often not enough—depending on the application, the entire process needs to be very accurate, very fast, or differentiable (or worse, several of the above). Such requirements imply painstaking global transformations into highly specialized implementations that are challenging to understand and maintain.

These challenges motivate the design of our system: the combination of generic algorithms and composable compile-time transformations of types enable development at a high level of abstraction. Without code duplication, our system is then able to generate high-quality scalar, vector and GPU implementations with competitive performance. Another type of transformation changes the representation of radiance, making light transport effects like polarization considerably easier to support. Finally, Mitsuba 2's lazy JIT compiler and automatic differentiation unlock a path to straightforward conversion of any rendering algorithm or appearance model into an optimization technique for solving associated inverse problems.

A number of limitations and open questions remain: due to our reliance on deeply nested templates, error messages provided by the compiler can be cryptic. We address such problems by performing a scalar-only build of the renderer—once this succeeds, the other variants should follow suit, assuming that the transformations themselves are correct. C++20 introduces a feature named *concepts* that will likely address this problem more elegantly. Mitsuba 2 code also requires a conversion of conditional statements into masks, which can be tedious when a model requires intricate conditional logic.

Despite the optimizations pursued in this article, reverse-mode automatic differentiation significantly increases the amount of application state, and parameters like resolution, sample count, and number of passes, require careful adjustment to avoid out-of-memory errors. In contrast to neural networks, gradient-based optimization of renderings is prone to low-quality local minima and hence requires careful initialization. Our GPU backend renders images using a sequence of separate kernel launches that exchange information through global memory, which causes large communication-related overheads. Systems like OptiX that compile an entire renderer into a single "megakernel" avoid this type of overhead, although their increased register usage tends to impede the latency-hiding ability of modern GPUs [Laine et al. 2013]. We currently unroll loops and recursive algorithms, which is likely not always ideal. It would be interesting to study these various trade-offs, particularly in the context of differentiable rendering algorithms.

Other relevant areas of future work include more aggressive graph simplifications that reduce memory requirements, the design of specialized sampling strategies that improve variance specifically for gradients, and the development of parameterized sampling strategies embedded within the renderer that are trained end-to-end in conjunction with a denoising step.

We believe that Mitsuba 2 will be a helpful tool for researchers in computer graphics, computer vision, and many other areas (e.g. design or architecture) that optimize geometry or materials to achieve a goal that can be specified as a differentiable algorithm.

ACKNOWLEDGMENTS

We would like to thank Sébastien Speierer for early contributions to Mitsuba 2, Hannes Hergeth for his experiments involving polarized light transport, Matt Pharr for feedback on an early paper draft, Salma El Alaoui Talibi for providing voiceover for the video, and Olesya Jakob for feedback on the scenes shown in Figure 1.

Additional scenes and meshes for the evaluation are from Benedikt Bitterli's repository [2016] and Blend Swap user *nukkio*. They were rendered with environment maps from HDRI Haven.

This research was supported by the Swiss National Science Foundation (SNSF) as part of grant 200021_184629.

REFERENCES

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.

Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. 2016. Local Shading Coherence Extraction for SIMD-efficient Path Tracing on CPUs. In *Proceedings of High Performance Graphics (HPG '16)*. Eurographics Association.

Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. 2017. Aether: An embedded domain specific sampling language for Monte Carlo rendering. *ACM Transactions on Graphics* 36, 4 (2017).

Christophe Andrieu and Gareth O Roberts. 2009. The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics* (2009).

Dejan Azinović, Tzu-Mao Li, Anton Kaplanyan, and Matthias Nießner. 2019. Inverse Path Tracing for Joint Material and Lighting Estimation. In *Proceedings of Computer Vision and Pattern Recognition (CVPR), IEEE.*

Seung-Hwan Baek, Daniel S. Jeon, Xin Tong, and Min H. Kim. 2018. Simultaneous Acquisition of Polarimetric SVBRDF and Normals. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2018)* 36, 6 (2018).

Benedikt Bitterli. 2016. Rendering resources. https://benedikt-bitterli.me/resources/.

Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Transactions on Graphics* 37, 3, Article 33 (July 2018).

Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *CoRR* abs/1509.07164 (2015). arXiv:1509.07164 http://arxiv.org/abs/1509.07164

Subrahmanyan Chandrasekhar. 1960. *Radiative transfer.* Dover publications, New York.

Chengqian Che, Fujun Luan, Shuang Zhao, Kavita Bala, and Ioannis Gkioulekas. 2018. Inverse Transport Networks. *arXiv preprint arXiv:1809.10820* (2018).

Edward Collett. 1993. *Polarized light : fundamentals and application.* Marcel Dekker New York.

Oskar Elek, Denis Sumin, Ran Zhang, Tim Weyrich, Karol Myszkowski, Bernd Bickel, Alexander Wilkie, and Jaroslav Křivánek. 2017. Scattering-aware Texture Reproduction for 3D Printing. *ACM Transactions on Graphics* 36, 6 (Nov. 2017).

Luca Fascione, Johannes Hanika, Marcos Fajardo, Per Christensen, Brent Burley, and Brian Green. 2017. Path Tracing in Production - Part 1: Production Renderers. In *ACM SIGGRAPH 2017 Courses (SIGGRAPH '17)*.

Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics* 37, 3, Article 31 (Aug. 2018).

Mathieu Galtier, Stéphane Blanco, Cyril Caliot, Christophe Coustet, Jérémi Dauchet, Mouna El Hafi, Vincent Eymet, Richard Fournier, Jacques Gautrais, Anaïs Khuong, et al. 2013. Integral formulation of null-collision Monte Carlo algorithms. *Journal of Quantitative Spectroscopy and Radiative Transfer* 125 (2013).

Ioannis Gkioulekas, Shuang Zhao, Kavita Bala, Todd Zickler, and Anat Levin. 2013. Inverse Volume Rendering with Material Dictionaries. *ACM Transactions on Graphics* 32, 6, Article 162 (Nov. 2013).

A Griewank and S Reese. 1991. *On the calculation of Jacobian matrices by the Markowitz rule.* Technical Report. Argonne National Lab., IL (United States).

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

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

Johannes Hanika. 2019. Personal communication.

Eric Heitz and Eugene d'Eon. 2014. Importance Sampling Microfacet-Based BSDFs using the Distribution of Visible Normals. In *Computer Graphics Forum*, Vol. 33.

Robin J. Hogan. 2014. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Software* 40, 4, Article 26 (July 2014).

Wenzel Jakob. 2010. Mitsuba renderer. https://www.mitsuba-renderer.org. (Date accessed: 2019-08-25).

Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki. (Date accessed: 2019-08-25).

Wenzel Jakob and Johannes Hanika. 2019. A Low-Dimensional Function Space for Efficient Spectral Upsampling. *Computer Graphics Forum (Proceedings of Eurographics)* 38, 2 (March 2019).

Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11—Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

Adrian Jarabo and Victor Arellano. 2018. Bidirectional Rendering of Vector Light Transport. *Computer Graphics Forum* 37, 6 (2018).

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2017. Neural 3D Mesh Renderer. *CoRR* abs/1711.07566 (2017). arXiv:1711.07566 http://arxiv.org/abs/1711.07566

Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. 2002. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum*, Vol. 21. Wiley Online Library.

Alexander Keller, Carsten Wächter, Matthias Raab, Daniel Seibert, Dietger van Antwerpen, Johann Korndörfer, and Lutz Kettner. 2017. The Iray Light Transport Simulation and Rendering System. In *ACM SIGGRAPH 2017 Talks (SIGGRAPH '17)*. ACM, New York, NY, USA.

Pramook Khungurn, Daniel Schroeder, Shuang Zhao, Kavita Bala, and Steve Marschner. 2015. Matching Real Fabrics with Micro-Appearance Models. *ACM Transactions on Graphics* 35, 1 (2015).

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. ACM.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized production path tracing. In *Proceedings of High Performance Graphics*. ACM.

Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA (Nov. 2018).

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018a. Differentiable Monte Carlo Ray Tracing Through Edge Sampling. *ACM Transactions on Graphics* 37, 6, Article 222 (Dec. 2018).

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018b. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 37, 4 (2018).

Tzu-Mao Li, Jaakko Lehtinen, Ravi Ramamoorthi, Wenzel Jakob, and Frédo Durand. 2015. Anisotropic gaussian mutations for metropolis light transport through hessian-hamiltonian dynamics. *ACM Transactions on Graphics* 34, 6 (2015).

Jun S Liu, Faming Liang, and Wing Hung Wong. 2000. The multiple-try method and local optimization in Metropolis sampling. *J. Amer. Statist. Assoc.* 95, 449 (2000).

Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. 2019. Soft Rasterizer: Differentiable Rendering for Unsupervised Single-View Mesh Reconstruction. *CoRR* abs/1901.05567 (2019). arXiv:1901.05567 http://arxiv.org/abs/1901.05567

Matthew M Loper and Michael J Black. 2014. OpenDR: An approximate differentiable renderer. In *European Conference on Computer Vision*. Springer.

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing Discontinuous Integrands for Differentiable Rendering. *ACM Transactions on Graphics* (Dec. 2019).

Duane Merrill. 2015. CUB library. http://nvlabs.github.io/cub. (Date accessed: 2019-08-25).

Scott Meyers. 2005. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education.

Michal Mojzík, Tomáš Skřivan, Alexander Wilkie, and Jaroslav Křivánek. 2016. Bi-Directional Polarised Light Transport. In *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*, Elmar Eisemann and Eugene Fiume (Eds.). The Eurographics Association.

Iain Murray. 2007. *Advances in Markov chain Monte Carlo methods*. University of London.

Uwe Naumann. 2007. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming* 112 (2007).

Du T. Nguyen, Cameron Meyers, Timothy D. Yee, Nikola A. Dudukovic, Joel F. Destino, Cheng Zhu, Eric B. Duoss, Theodore F. Baumann, Tayyab Suratwala, James E. Smay, and Rebecca Dylla-Spears. 2017. 3D-Printed Transparent Glass. *Advanced Materials* 29, 26 (2017).

Melissa E. O'Neill. 2014. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905. Harvey Mudd College, Claremont, CA.

Marios Papas, Wojciech Jarosz, Wenzel Jakob, Szymon Rusinkiewicz, Wojciech Matusik, and Tim Weyrich. 2011. Goal-Based Caustics. *Computer Graphics Forum (Proceedings of Eurographics)* 30, 2 (June 2011).

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics* 29, 4, Article 66 (July 2010).

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).

Felix Petersen, Amit H. Bermano, Oliver Deussen, and Daniel Cohen-Or. 2019. Pix2Vex: Image-to-Geometry Reconstruction using a Smooth Differentiable Renderer. *CoRR* abs/1903.11149 (2019). arXiv:1903.11149 http://arxiv.org/abs/1903.11149

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation* (third ed.). Morgan Kaufmann.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering Complex Scenes with Memory-coherent Ray Tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co.

Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE.

Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: Generating Renderers without Writing a Generator. *ACM Transactions on Graphics* 38, 4 (7 2019).

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Notices* 48, 6 (June 2013).

Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. 2015. A Versatile Scene Model with Differentiable Visibility Applied to Generative Pose Estimation. In *Proceedings of ICCV 2015*.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. MIT Press, Cambridge, MA, USA, Chapter Learning Internal Representations by Error Propagation.

Yuliy Schwartzburg, Romain Testuz, Andrea Tagliasacchi, and Mark Pauly. 2014. High-contrast Computational Caustic Design. *ACM Transactions on Graphics* 33, 4, Article 74 (July 2014). Proc. SIGGRAPH 2014.

Benjamin Segovia, Jean-Claude Iehl, and Bernard Péroche. 2007a. Coherent metropolis light transport with multiple-try mutations.

Benjamin Segovia, Jean Claude Iehl, and Bernard Péroche. 2007b. Metropolis instant radiosity. In *Computer Graphics Forum*, Vol. 26. Wiley Online Library.

Anurag Sharma, D Vizia Kumar, and Ajoy K Ghatak. 1982. Tracing rays through graded-index media: a new method. *Applied Optics* 21, 6 (1982).

Denis Sumin, Tobias Rittig, Vahid Babaei, Thomas Nindel, Alexander Wilkie, Piotr Didyk, Bernd Bickel, Jaroslav Křivánek, Karol Myszkowski, and Tim Weyrich. 2019. Geometry-Aware Scattering Compensation for 3D Printing. *ACM Transactions on Graphics* (2019).

JM Tregan, S Blanco, J Dauchet, M Hafi, R Fournier, L Ibarrart, P Lapeyre, and N Villefranque. 2019. Convergence issues in derivatives of Monte Carlo null-collision integral formulations: a solution. *arXiv preprint arXiv:1903.06508* (2019).

Eric Veach. 1997. *Robust monte carlo methods for light transport simulation*. Number 1610. Stanford University PhD thesis.

Todd Veldhuizen. 1995. Expression Templates. *C++ Report* 7 (1995).

Yu M Volin and GM Ostrovskii. 1985. Automatic computation of derivatives with the use of the multilevel differentiating technique—1. Algorithmic basis. *Computers & mathematics with applications* 11, 11 (1985).

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33, 4, Article 143 (July 2014).

Alexander Wilkie and Andrea Weidlich. 2012. Polarised Light in Computer Graphics. In *SIGGRAPH Asia 2012 Courses (SA '12)*. ACM, New York, NY, USA, Article 8.

E Woodcock, T Murphy, P Hemmings, and S Longworth. 1965. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings of the Conference on Applications of Computing Methods to Reactor Problems*, Vol. 557.

Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

Toshinobu Yoshida. 1987. Derivation of a computational process for partial derivatives of functions using transformations of a graph. *Transactions of Information Processing Society of Japan* 11, 19 (1987).

Yonghao Yue, Kei Iwasaki, Bing-Yu Chen, Yoshinori Dobashi, and Tomoyuki Nishita. 2014. Poisson-Based Continuous Surface Generation for Goal-Based Caustics. *ACM Transactions on Graphics* 33, 3, Article 31 (June 2014).

Shaung Zhao, Lifan Wu, Frédo Durand, and Ravi Ramamoorthi. 2016. Downsampling Scattering Parameters for Rendering Anisotropic Media. *ACM Transactions on Graphics* 35, 6 (2016).