Mitsuba 2: A Retargetable Forward and Inverse Renderer – Supplemental Material

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

TABLE OF CONTENTS

1 OPTIMIZATION

Unbiased gradient estimation. In all of our optimization examples we minimize an average a pixel-wise loss over a set of parameters θ

$$\min_\theta L[\theta] = \min_\theta \frac{1}{N} \sum_i^N l\left(I^i, \hat{I}^i_\theta\right) \tag{1}$$

where i enumerates all pixels, l is the loss function, I^i the reference pixel value and $\hat{I}^i_\theta$ a Monte Carlo estimate of the value of pixel i.

If we naively apply automatic differentiation to the above loss function, the resulting estimate of the gradient $\nabla_\theta L(\theta)$ is biased due to the variance in the Monte Carlo estimate of the pixel value and the use of a non-linear loss function [Azinović et al. 2019]. For example in the case of evaluating an L_2 loss of a Monte Carlo estimate $\hat{I}^i_\theta$ compared to a reference I^i, we obtain

$$\nabla_\theta E[(\hat{I}^i_\theta - I^i)^2] = 2E[\nabla_\theta \hat{I}^i_\theta](E[\hat{I}^i_\theta] - I^i) + 2\mathrm{Cov}\left[\hat{I}^i_\theta, \nabla_\theta \hat{I}^i_\theta\right] \tag{2}$$

$$\neq E[\nabla_\theta \hat{I}^i_\theta](E[\hat{I}^i_\theta] - I^i) = \nabla_\theta (E[\hat{I}^i_\theta] - I^i)^2. \tag{3}$$

This means that the gradient estimated from the noisy Monte Carlo pixel color estimate (LHS) is off by a covariance term from the true gradient (RHS). A direct application of automatic differentiation would evaluate both color and gradient information simultaneously using the same random numbers. When optimizing an L_1 or L_2 loss, or their respective relative variants, the bias fortunately can be eliminated by evaluating pixel color and gradient separately (thus setting the above covariance term to zero). We provide a command `reattach`, which attaches gradient information from an independent computation graph to an output variable. Within an optimization script written in Python this can be used as follows

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.

```
image = render_scene(scene, spp=64, seed=seed1)
gradient_image = render_scene(scene, spp=16, seed=seed2)
reattach(image, gradient_image)
l = loss(image, reference_image)
# ...
```

After reattaching the gradient information, the image variable can be used to compute the loss function, without any further code modifications. Note that our gradient estimate is unbiased for the loss functions we use in our examples, but not for arbitrary objective functions (e.g. not for an L_3 loss). Similar to Azinović et al. [2019], we render the color image with a higher sample count than the gradient. This provides a better estimate of the loss value without the expense of maintaining a memory-heavy AD graph involving many samples.

Multi-pass rendering. Since storing a big computation graph requires a considerable amount of GPU memory, we split the gradient computation into multiple passes at a lower number of samples per pixel. After each pass, we accumulate the current gradient and discard the AD graph before starting the computation of the next pass. Using the unbiased gradient estimate, this multi-pass approach does not introduce any additional error into the optimization.

Optimization techniques. We minimize the objective function using standard optimizers such as stochastic gradient descent with momentum or Adam [Kingma and Ba 2014]. When optimizing for the values of 2D or 3D textures, we improve convergence using a multiresolution approach, starting at a low resolution and gradually increasing it as the optimization progresses. To constrain physical parameters (e.g. index of refraction) to a valid range, we map them through a scaled sigmoid.

Statistics. We show various statistics on our optimizations in Table 1. All timings have been measured running Mitsuba 2 on a Nvidia RTX Titan GPU (24 GB of memory).

	Number of Parameters	Image resolution	Views	SPP	∇-SPP	Passes	Iter. time	Iterations
RGB caustics	2048x2048	128x128x3	1	256	64	10	5 s	1500
Gradient-index	160x160x160	256x256	2	40	10	4	3 s	500
Smoke	128x128x128	128x128	9	64	16	8	3 min	250
Translucent slab	256x256x64x3	128x128x3	1	16	4	6	1.5 min	170

Table 1. Various statistics on our optimization applications. The number of samples per pixel (SPP) and gradient samples (∇-SPP) are the number of samples per render pass.

2 COMPARISON TO REDNER

Redner, the publicly available implementation of the technique by Li et al. [Li et al. 2018], is a GPU-based differentiable renderer that supports global illumination in combination with several material models and emitter types. It accounts for derivatives due to visibility changes at silhouette edges using *edge sampling*, while Mitsuba 2 relies on a reparameterization approach that is discussed in a separate article [Loubet et al. 2019].

In the following, we compare the performance of our system compared to *Redner* as of commit 4582928. We disable special handling of non-differentiable visibility changes in both systems to measure the core performance of the underlying approach for automatic differentiation and vectorized execution in the GPU. For a performance comparison that also includes visibility-related effects, refer to the paper by Loubet et al. [Loubet et al. 2019].

The script used to drive optimizations in *Redner* was adapted from the 5th tutorial distributed with the framework. In Table 2, we report time per iteration for forward and backward passes, but exclude the scene construction time (performed at each iteration in *Redner*) since it could most likely be avoided in cases where geometry is not updated. For *Redner*, we report timings both for the default configuration options (double precision) and by replacing the scalar type to use single precision. In these examples, we observe 10-75% faster iterations against the default configuration of Redner when using Mitsuba 2.

	Number of Parameters	Image resolution	SPP	Redner (float)	Redner (default)	Mitsuba 2
Diffuse Cornell box	5×3	256×256	16	0.2336 s/it	0.3553 s/it	0.3140 s/it
Textured monkey	$512 \times 512 \times 3$	256×256	16	0.1785 s/it	0.2280 s/it	0.1501 s/it
Textured monkey	$1024 \times 1024 \times 3$	256×256	16	0.1792 s/it	0.2293 s/it	0.1503 s/it
Textured sphere	$512 \times 512 \times 3$	256×256	16	0.1538 s/it	0.1991 s/it	0.1132 s/it
Textured sphere	$1024 \times 1024 \times 3$	256×256	16	0.1548 s/it	0.1998 s/it	0.1133 s/it

Table 2. Timing comparisons against *Redner*.

Measurements were made on a desktop computer with an NVIDIA RTX 2080 Ti (driver version: 435.21) with 11GB of GPU RAM. The times reported are averaged over 100 iterations of gradient descent (timings are stable over iterations). The three scenes are shown in Figure 1. The Cornell Box is optimized for 5 diffuse RGB coefficients, while the other two scenes are optimized for a diffuse RGB texture at 512×512 or 1024×1024 resolution. Renders were produced with a unidirectional path tracer at maximum path depth 5. The "monkey" mesh has 31658 vertices and 62976 faces, while the "sphere" mesh has 482 vertices and 960 faces.

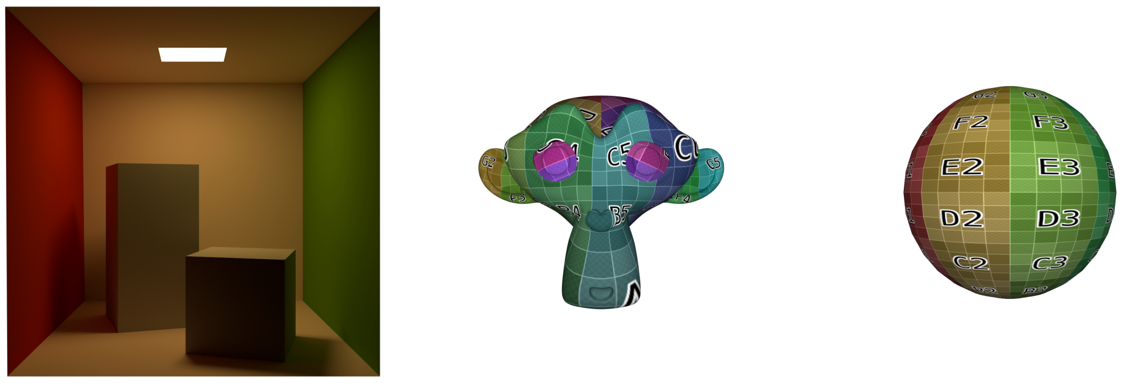

Fig. 1. Scenes used for the performance measurements against *Redner*.

3 EXAMPLE SOURCE CODE

This section provides longer example of typical Mitsuba 2 source code, specifically an annotated path tracer with multiple importance sampling.

```
/**
 * This class contains the source code of a simple path tracer with multiple
 * importance sampling.
 *
 * The integrator plugin is parameterized by two types: the first is the
 * underlying floating point type 'Float', which could be an ordinary C++
 * 'float', an Enoki packet type e.g. for vectorizing on AVX512, a GPU array,
 * or a differentiable GPU array.
 *
 * The 'Spectrum' type is used for color-related computations. When building a
 * monochromatic version of this plugin, it is an 1-element spectrum
 * 'mitsuba::Spectrum<Float, 1>', RGB mode uses Color3f<Float>, spectral mode
 * uses mitsuba::Spectrum<Float, MTS_WAVELENGTH_SAMPLES> with a larger number
 * of spectral samples, and polarized mode furthermore wraps everything
 * in a MuellerMatrix<T>.
 */

template <typename Float, typename Spectrum>
class PathIntegrator : public MonteCarloIntegrator<Float, Spectrum> {
public:
    /* Import derived types based on the template parameters 'Float' and 'Spectrum'.
     * For instance: UInt32, Vector3f, SurfaceInteraction3f, etc.. */
    MTS_IMPORT_TYPES()

    /// Initialize the integrators with parameters from the XML scene description
    PathIntegrator(const Properties &props) : MonteCarloIntegrator(props) { }

    /// Evaluates radiance along the specified path
    Spectrum eval(const RayDifferential3f &ray_,
                  RadianceSample &rs, Mask active) const {
        RayDifferential3f ray(ray_);
        const Scene *scene = rs.scene;

        // Tracks radiance scaling due to index of refraction changes
        Float eta = 1.f;

        // MIS weight for intersected emitters (set by previous iteration)
        Float emission_weight = 1.f;
```

```
/// Product of sampling weights, radiance accumulated so far
Spectrum throughput(1.f), result(0.f);

// ---------------------- First intersection ----------------------

// Generate ray intersections using the active backend (Mitsuba / Embree / OptiX)
SurfaceInteraction3f si = rs.ray_intersect(ray, active);

// Compute the opacity associated with the ray(s)
rs.alpha = select(si.is_valid(), Float(1.f), Float(0.f));

/* Did we intersect an emitter (or a set of emitters)?
   If so, obtain a pointer to them. */
EmitterPtr emitter = si.emitter(scene, active);

for (int depth = 1;; ++depth) {
    // ---------------- Intersection with emitters ----------------

    /* If we intersected an emitter, accumulate the associated radiance. When
       'Float' is a GPU array, 'any_or<true>' returns the template argument, causing
       the branch to always be taken to avoid a costly horizontal reduction.
       Note the use of masks in the assignment and function call to ensure correctness. */
    if (any_or<true>(neq(emitter, nullptr)))
        result[active] += emission_weight * throughput * emitter->eval(si, active);

    // Mark wavefront elements as inactive if the ray did not intersect anything
    active &= si.is_valid();

    /* Russian roulette: try to keep path weights equal to one,
       while accounting for the solid angle compression at refractive
       index boundaries. Stop with at least some probability to avoid
       getting stuck (e.g. due to total internal reflection) */
    if (depth > m_rr_depth) {
        /* Compute the termination probability. 'unpolarized' returns
           the (1, 1) element of a Mueller matrix and is the identity
           given other types. Note that the 'hmax' horizontal operation
           used below is actually a vertical operation on GPU/CPU SIMD
           targets since it adds up spectral channels that are
           themselves arrays. */
        Float q = min(hmax(unpolarized(throughput)) * sqr(eta), .95f);
        active &= rs.next_1d(active) < q;
        throughput *= rcp(q);
    }

    /* Stop when we've exceeded the max path depth, or when there is no more work
       On the GPU, the following lines cause a kernel launch that evaluates
       previously queued computation. */
    if (none(active) || (uint32_t) depth >= (uint32_t) m_max_depth)
        break;

    /* --------------------- Emitter sampling --------------------- */

    BSDFContext ctx;

    /// Obtain a pointer(s) to the BSDF(s) associated with the intersection(s)
    BSDFPtr bsdf = si.bsdf(ray, active);

    /* Only sample the emitter when the BSDF isn't a Dirac delta function. When
       'Float' is a GPU array, 'any_or<true>' returns the template argument, causing
       the branch to always be taken to avoid a costly horizontal reduction. */
    Mask active_e = active && neq(bsdf->flags() & BSDF::ESmooth, 0u);
    if (any_or<true>(active_e)) {
        /* Importance sample a direction towards an emitter. Returns
           'ds' of type DirectionSample3f and a Spectrum. On vectorized
           targets, this will potentially importance multiple different
           emitters. */
        auto [ds, emitter_val] = scene->sample_emitter_direction(
            si, rs.next_2d(active_e), /* test_visibility = */ true, active_e);

        // Disable elements where sampling failed, or which are occluded
        active_e &= any(neq(unpolarized(emitter_val), 0.f));

        // Transform the direction to the light source into the local BSDF frame
        Vector3f wo = si.to_local(ds.d);

        /* Query the BSDF. Note that on vectorized targets, 'bsdf'
           might be an array containing pointers to many different BSDFs.
           In this case, Enoki's method call handler will perform a
           sequence of function calls corresponding to the unique array
           elements. On the GPU backend, the computation of all the
           different function calls is queued up, to be flushed and
```

```
               executed in parallel at the next horizontal reduction or
               method call. */
            Spectrum bsdf_val = bsdf->eval(ctx, si, wo, active_e);

            /* In polarized mode, 'bsdf_val' is a MuellerMatrix expressed
               with respect to a local coordinate system that must now be
               transformed to world coordinates. This expands to a no-op
               in non-polarized mode. */
            bsdf_val = si.to_world_mueller(bsdf_val, -wo, si.wi);

            /* Determine probability of having sampled that same
               direction using BSDF sampling. */
            Float bsdf_pdf = bsdf->pdf(ctx, si, wo, active_e);

            // Apply MIS weight and accumulate contribution
            Float mis = select(ds.delta, 1.f, mis_weight(ds.pdf, bsdf_pdf));

            // Note: the order of multiplication operations is important for polarization
            result[active_e] += throughput * bsdf_val * emitter_val * mis;
        }

        // ----------------------- BSDF sampling ----------------------

        /* Importance sample the BSDF * cos(theta).
           Returns 'bs' of type BSDFSample3f and a Spectrum. */
        auto [bs, bsdf_val] = bsdf->sample(ctx, si, rs.next_1d(active),
                                           rs.next_2d(active), active);

        /* In polarized mode, 'bsdf_val' is a MuellerMatrix expressed
           with respect to a local coordinate system that must now be
           transformed to world coordinates. This expands to a no-op
           in non-polarized mode. */
        bsdf_val = si.to_world_mueller(bsdf_val, -bs.wo, si.wi);

        // Accumulate path throughput
        throughput *= bsdf_val;

        // Potential early-out on the CPU in case the throughput is zero
        active &= any(neq(unpolarized(throughput), 0.f));
        if (none_or<false>(active))
            break;
            /// Keep track of IOR changes
            eta *= bs.eta;

            // Intersect the sampled ray against the scene geometry
            ray = si.spawn_ray(si.to_world(bs.wo));
            SurfaceInteraction3f si_bsdf = scene->ray_intersect(ray, active);

            /* Determine probability of having sampled that same
               direction using emitter sampling. */
            emitter = si_bsdf.emitter(scene, active);
            DirectionSample3f ds(si_bsdf, si);
            ds.object = emitter;

            if (any_or<true>(neq(emitter, nullptr))) {
                /* If the sampled BSDF component is a Dirac delta function,
                 * that connection strategy is unavailable. */
                Float emitter_pdf =
                    select(neq(bs.sampled_type & BSDF::EDelta, 0u), 0.f,
                           scene->pdf_emitter_direction(si, ds, active));

                // Set emitter MIS weight consumed in next iteration
                emission_weight = mis_weight(bs.pdf, emitter_pdf);
            }

            si = std::move(si_bsdf);
        }

        return result;
    }

    /// Declare run-time type information for Mitsuba's object model
    MTS_DECLARE_CLASS()

protected:
    // MIS weighting scheme: the power heuristic
    Float mis_weight(Float pdf_a, Float pdf_b) const {
        pdf_a *= pdf_a;
        pdf_b *= pdf_b;
        return select(pdf_a > 0.f, pdf_a / (pdf_a + pdf_b), 0.f);
    };
```

```
};

MTS_IMPLEMENT_CLASS(PathIntegrator, MonteCarloIntegrator);
MTS_EXPORT_PLUGIN(PathIntegrator, "Path Tracer integrator");
```

REFERENCES

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*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

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

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