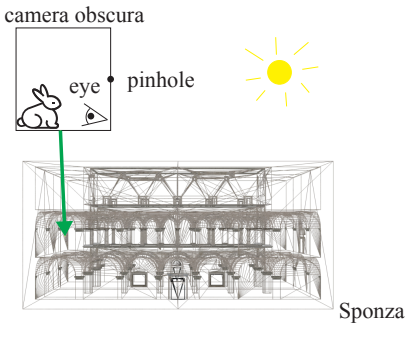
Aether: An Embedded Domain Specific Sampling Language for Monte Carlo Rendering

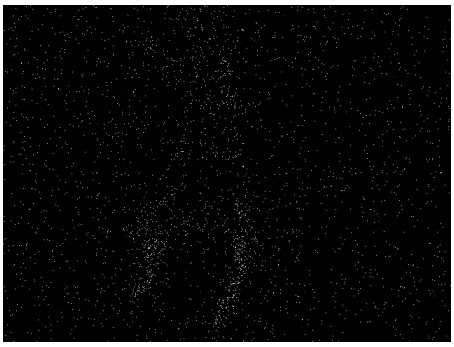
Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, Frédo Durand

Aether: An Embedded Domain Specific Sampling Language for Monte Carlo Rendering

LUKE ANDERSON, MIT CSAIL
TZU-MAO LI, MIT CSAIL
JAAKKO LEHTINEN, Aalto University and NVIDIA
FRÉDO DURAND, MIT CSAIL and Inria, Université Côte d'Azur



Camera Obscura Scene



Bidirectional Path Tracing, 64 spp

Our Tridirectional Path Tracing, 64 spp

Fig. 1. A scene that is challenging to render for traditional Monte Carlo methods: the Sponza Palace atrium projected into a box through a pinhole. The area of the pinhole is only 0.01 percent of the face of the pinhole camera. Using our language we designed a specialized tridirectional path tracing algorithm that generates a light path segment passing through the pinhole. The image shows an equal sample comparison between bidirectional path tracing and our tridirectional path tracing. See Section 7.4 for more details.

Implementing Monte Carlo integration requires significant domain expertise. While simple samplers, such as unidirectional path tracing, are relatively forgiving, more complex algorithms, such as bidirectional path tracing or Metropolis methods, are notoriously difficult to implement correctly. We propose Aether, an embedded domain specific language for Monte Carlo integration, which offers primitives for writing concise and correct-by-construction sampling and probability code. The user is tasked with writing sampling code, while our compiler automatically generates the code necessary for evaluating PDFs as well as the book keeping and combination of multiple sampling strategies. Our language focuses on ease of implementation for rapid exploration, at the cost of run time performance. We demonstrate the effectiveness of the language by implementing several challenging rendering algorithms as well as a new algorithm, which would otherwise be prohibitively difficult.

CCS Concepts: • **Computing methodologies** → **Rendering**; • **Software and its engineering** → **Domain specific languages**;

Additional Key Words and Phrases: Global Illumination

ACM Reference format:
Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. 2017. Aether: An Embedded Domain Specific Sampling Language for Monte Carlo Rendering. *ACM Trans. Graph.* 36, 4, Article 99 (July 2017), 16 pages.
https://doi.org/http://dx.doi.org/10.1145/3072959.3073704

1 INTRODUCTION

Probabilistic integration techniques used in lighting simulation shine by their combination of elegance and efficiency. The pseudocode for an algorithm like bidirectional path tracing with multiple importance sampling fits in a small figure and reveals at once its power and sophistication. Unfortunately, while the implementation of a simple path tracer is relatively straightforward, achieving a correct implementation of more advanced algorithms, such as bidirectional path tracing or Metropolis light transport, is a major undertaking prone to subtle probability bugs that are extremely challenging to chase down. This is obvious from the very small number of available implementations, and even the widely used pbrt [Pharr and Humphreys 2010] did not include full bidirectional path tracing with multiple importance sampling until ten years after the first release, and a publicly available implementation of Metropolis light transport [Veach and Guibas 1997] did not appear until a decade after the original paper [Jakob 2010].

A major implementation difficulty lies with correct handling of probabilities, both in terms of mathematical correctness and bookkeeping, by which we mean the care required for drawing samples and combining estimates from several different, complex samplers, which are, in addition, often defined on different parameterizations (measures). In particular, algorithms such as multiple importance sampling and Metropolis require the computation of not only the probability of a sample with respect to the strategy that generated it, but also with respect to other strategies. This means that simply

keeping track of probabilities as we generate a sample is not sufficient. In addition to the challenge of deriving correct PDF formulas for the sampling of continuous variables, algorithms that assemble sub-paths also need to carefully track and account for the many different combinatorial ways to generate the same path.

For concreteness, the interested reader can compare and contrast the implementations of path tracing and bidirectional path tracing in either the publicly available Mitsuba or pbrt-v3 renderers. It is readily apparent that the move from path tracing to bidirectional path tracing necessitates large changes in software architecture, including the data structures used for holding path data, as well as a significant increase in code complexity stemming from computation of multiple importance sampling weights. Now imagine experimenting with new sampling ideas, such as a tridirectional approach that would trace sub-paths from not only the eye and the light source, but also from a known important opening such as a keyhole or a pinhole. This would require the proper probability computation and book keeping for many different subpath generation strategies, a daunting task with current pen-and-paper approaches.

To address these difficulties, we propose Aether, a new domain specific language that dramatically simplifies the implementation of unbiased Monte Carlo integration. Our central goal is to relieve the programmer of the tasks of deriving and implementing probability density functions, performing explicit measure conversions, and dealing with the book keeping and combinatorics of different sampling strategies. Importantly, we need to facilitate complex samplers that are able to evaluate their PDFs at arbitrary sample points that may have been sampled through other samplers, as required by, for instance, multiple importance sampling or Metropolis. In our language, the programmer writes only sampling code, and the language automatically generates the necessary density code by computing symbolic derivatives and function inverses at compile time. This ensures consistency between a given sample and its density, and eliminates the need for explicit measure conversions. The language also generates the book keeping and sample combination code to deal with multiple samples. Finally, it can compute conditional probabilities, as required by Metropolis-Hastings algorithms. The resulting code is compact and correct by construction, making it easier to focus on higher-level algorithmic and mathematical design. Our focus is on correctness and not speed. While the resulting generated code is currently below hand optimized implementations, the language facilitates the implemention of algorithms that can be difficult to achieve with existings renderers. Our succinct implementations of gradient-domain path tracing and the novel tridirectional path tracing demonstrate the potential of our language.

We plan to make the language publicly available at https://github.com/aekul/aether.

2 PRIOR WORK

Rendering Systems. pbrt [Pharr and Humphreys 2010] and Mitsuba [Jakob 2010] are two well known physically based rendering systems widely used by the research community as testbeds for developing and verifying new algorithms. While they both include sampling functionality, neither supports a simple, robust way to write new code. They both require manual derivations of density functions, and place the burden on the programmer to keep track of the samples, their measures, and how they are to be combined. In particular, both pbrt and Mitsuba keep a record of density values together with their associated samples. This approach is error prone, particularly for inexperienced users, because it does not inform the user of which sampling strategy the density values were obtained from. They both also feature tight coupling of sampling code, density code, and the code for computing the integrand and estimate. This makes it difficult to reuse existing code and extend the systems with implementations of new algorithms.

Probabilistic Programming Languages. Many probabilistic programming languages have been proposed, with varying properties and features: Church [Goodman et al. 2012] (generative models, inference); webppl [Goodman and Stuhlmüller 2014] (inference); Stan [Stan Development Team. 2015] (Bayesian inference), Factorie [McCallum et al. 2009] (factoring graphical models, inference), BLOG [Milch et al. 2007] (inference), and many others. All existing languages that we know of are designed for statistical machine learning tasks and focus on the Bayesian setting: data collection and inference about probabilities. In contrast, we know the sampling process and need to compute the corresponding probabilities. We are solving the forward problem, they are solving the reverse problem.

Symbolic Algebra Systems. There are numerous existing symbolic algebra systems: Mathematica [Wolfram Research, Inc. 2016], SAGE, sympy, and others. While Mathematica and sympy can be incorporated into other programs through compilation or simple importing, none offer a simple way to write compile time expressions in C++. Further, these systems are typically general purpose. In comparison, we only have to handle more restricted operations. This simplifies our task of computing symbolic derivatives and inverses since we only have to handle more domain specific scenarios.

Dimensional Analysis. SafeGI [Ou and Pellacini 2010] is a C++ software library that offers compile time checking of physical dimensions, units, and geometric spaces for rendering systems. This kind of library is orthogonal to our language and could potentially be incorporated into it.

3 MATHEMATICAL BACKGROUND

In physically-based light transport simulation, the intensity of a pixel j is given by the integral

$$I_j = \int_\Omega h_j(\mathbf{x}) f(\mathbf{x}) \mathrm{d}\mu(\mathbf{x}) \tag{1}$$

where Ω is the set of all light paths, h_j is the sensor response function for pixel j, and f is the contribution function that measures the light throughput of a path in a chosen measure μ [Veach 1998]. A path $\mathbf{x} = \{x_1, x_2, \ldots, x_k\}$ is a sequence of vertices (scattering events) in the scene, starting from the light and ending at a virtual sensor.

Modern Monte Carlo techniques, our focus, often sample N random light paths from M distinct distributions and combine them. Each light path $\mathbf{x}_i$ is sampled from a distribution $p_j(\mathbf{x})$, which is one of the M distributions. An estimate of the integral from the paths'

contributions $f(\mathbf{x}_i)$ is weighted according to

$$I \approx \frac{1}{N} \sum_{i}^{N} W_i(\mathbf{x}_i) \frac{f(\mathbf{x}_i)}{p_j(\mathbf{x}_i)}, \tag{2}$$

where the combination weight heuristic $W_i(\mathbf{x}_i)$ is a function of all the probability densities $p_1(\mathbf{x}_i), p_2(\mathbf{x}_i), \ldots, p_M(\mathbf{x}_i)$, not just the density of the sampler that actually drew $\mathbf{x}_i$.

In a different vein, Markov Chain Monte Carlo methods such as Metropolis Light Transport [Veach and Guibas 1997] make use of random walks where the proposed random step from path $\mathbf{x}$ to path $\mathbf{y}$ is accepted with probability

$$\min\left\{1, \frac{f(\mathbf{y})\, p_j(\mathbf{x}|\mathbf{y})}{f(\mathbf{x})\, p_j(\mathbf{y}|\mathbf{x})}\right\}. \tag{3}$$

Here, $p_j(\mathbf{y}|\mathbf{x})$ is one of potentially many conditional probability densities used for randomly sampling mutations. Implementing the mutation samplers and the computation of its conditional probability density are challenging to the point that few complete implementations of the Metropolis algorithm are known.

3.1 Path Samplers and Their Densities

Most current light transport algorithms make use of sequential local sampling, where paths are extended one interaction at a time by sampling directions for extension rays. The process may start from the camera, from the light, or generally anywhere. The PDF of the entire path is the product of the individual sampling probabilities:

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_2, x_1)\ldots \tag{4}$$

The standard approach for constructing local importance sampling distributions is to find a function that warps a 2D uniformly distributed random variable (u_1, u_2) onto the (hemi-)sphere of directions, i.e., $\omega = w(u_1, u_2)$, with the desired density. The next path vertex x_{i+1} is then found by tracing a ray from the current vertex x_i in this direction. Hence, a sequential sampler S is a mapping from a series of 2D uniform random variables to a sequence of vertices, $\mathbf{x} = \{x_1, x_2, \ldots\} = S(u_1, u_2, \ldots)$. While we omit explicit dependence on location, it is understood that the shape of w may depend on, e.g., the incoming direction from the previous vertex, the surface normal, and the reflectance function.

To evaluate the probability density of a sampled local direction, standard probability calculus yields

$$p(\omega) = \sqrt{|\det J^{\mathrm{T}} J|}\; p(u_1, u_2) \tag{5}$$

where $J = \frac{\partial w}{\partial u_1, u_2}$ is the 3×2 Jacobian of the mapping from the square to the sphere. Note that evaluating the density at an arbitrary direction ω that was *not* sampled from the same PDF — so that we do not know the u_1, u_2 that produced ω — requires first the inversion

$$w^{-1}(\omega) = (u_1, u_2). \tag{6}$$

Standard hand-derived density functions implicitly include both the Jacobian and the inversion in the final formula. When computing the density of an entire path $\mathbf{y}$ sampled from another distribution, we must perform the multidimensional inversion $u_1, u_2, \ldots = S^{-1}(\mathbf{y})$. This reduces to a series of local 2D inversions of the form (6).

3.2 Case Study: Path Tracing

Standard implementations of estimators in the form of Equation (2) mix path generation, density computations, and integrand evaluation in a tangled, error-prone manner, making even relatively simple algorithms challenging to get right. Figure 2 features a pseudocode representation of a standard recursive path tracer with next event estimation. The code features functions for sampling BSDFs, picking light sources, and sampling points on light sources (blue); evaluating BSDFs and light emission for arbitrary points and directions (green), as well as code for evaluating densities and MIS weights for all combinations of samples drawn from the light and BSDF samplers. Not shown are code for generating samples from a non-pinhole camera, etc.

While the basic algorithm is easy to describe — append a shadow ray segment and a BSDF sample segment to the current path to create two "virtual" paths, add the contribution of light sources from the two paths with MIS, discard the shadow ray and extend the current path with the BSDF sample, recurse — this structure is barely visible from the code. Even in this simple case, it remains challenging to ensure the density and MIS weight code (red) is consistent with the sampling code (blue), as path generation, evaluation, and density code are all interspersed with each other.

Evaluating the estimator (Equation 2) for a more sophisticated path sampler, such as bidirectional path tracing [Veach 1998] poses even further challenges due to the manual application of surface area measure conversion factors, and, particularly, the fact that all the combinatorial ways of generating a path have to be matched by the manually specified full-path PDF code.

Fundamentally, implementing simple formulations such as Equation (2) is difficult for two reasons. First, the computation of $p_j(x_i)$ must be consistent with the sampling procedure that generates the x_i. Second, while the integrand f and density p_j are mathematically expressed as simple functions, ray-tracing code usually computes them incrementally as bounces are simulated, keeping track of partial values such as marginal probabilities and products of reflectance or radiance. This means that there is no single place where f or p are evaluated and the code for sampling, evaluating the integrand, and computing the density are interleaved, making it hard to create modular strategies.

In contrast, our language focuses on the concept of paths that are built from reusable and composable *strategies*, each of which encapsulates a simple operation such as BSDF sampling. Our language offers automatically derived full-path PDF and MIS weight code, effectively separating these computations from path construction. The interested reader may want to skip ahead and consult Figure 3 to see how this results in more readable and conceptually simpler code. While already helpful with the simple path tracer, the benefits compound when implementing more sophisticated methods.

4 GOALS AND DESIGN

4.1 Goals

Correctness By Construction. Estimators written in our language should have the correct expected value. The user may write code that is inefficient and has poor variance, but the expected value

```
Li = 0; throughput = 1;
Ray ray = spawn ray from camera
its = intersect ray with scene

while (path length is less than maxDepth) {
  break if its is invalid
  bsdf = its.bsdf; P = its.point; N = its.normal;
  wi = −ray.dir; Ld = 0;

  // Next event estimation (shadow ray) with MIS
  lightSrc = discretely pick light source
  lightP = sample point on lightSrc
  if (light is not blocked) {
    lightBsdfPdf = PDF if lightP sampled by bsdf
    lightPdf = PDF if lightP sampled by light
    weight = MISWeight(lightPdf, lightBsdfPdf)
    Le = evaluate lightSrc emission for P, lightP
    f = evaluate bsdf for wi, lightWo
    Ld += weight * Le * f / lightPdf
  }

  // Sample BSDF with MIS
  wo = sample outgoing direction from bsdf
  ray.o = P; ray.dir = wo
  its = intersect ray with scene
  f = evaluate bsdf for wi, wo
  bsdfPdf = PDF if wo sampled by bsdf
  if (hit a light) {
    bsdfLightPdf = PDF if wo sampled by light
    weight = MISWeight(bsdfPdf, bsdfLightPdf)
    Le = evaluate lightSrc emission for P, wo
    Ld += Le * f * weight / bsdfPdf
  }

  Li += throughput * Ld
  throughput *= f / bsdfPdf
}
```

Fig. 2. Pseudocode for a path tracer with MIS. Integrand and estimator code highlighted in green; sampling code highlighted in blue; and PDF and MIS code highlighted in red. We omit some details like geometry terms and conversions between solid angle and area measure.

should be correct. This means, in particular, that sampling code and the corresponding PDF code must be ensured to be consistent.

Conciseness and Expressiveness. Code should be simple, readable, and expressive enough for complex rendering algorithms.

Modularity and Reusability. User code, such as sampling strategies, should be modular and reusable across different algorithms.

Easy Integration with Existing Ray Tracing Kernels. Our language focuses on the probabilistic part of an algorithm, and users should be free to use any existing or novel library to perform computations such as ray casting and radiometric calculations.

Dimensionality of Samples. Rendering algorithms often need to generate samples that have a lower dimensionality than their ambient space, such as the use of 3D coordinates for directions or surface intersection points. We need to properly account for the probabilities on the lower-dimensional manifold.

Flexible Uniform Generation. The user should be free to drive the rendering algorithm using random or quasi-random number generators of their choice.

4.2 Design Decisions

Sampling vs. Density. To make the sampling and PDF code consistent, we chose to require the user to write the sampling code while we derive the corresponding PDF. It is much simpler than the opposite, since the PDF derivation requires a simple derivative, while deducing sampling from a PDF can be arbitrarily hard, especially for multi-dimensional cases.

Embedding in C++. As most renderers are written in C++ or C, we chose to embed our language in C++ for easy composition. The language requires no additional tools beyond a normal compiler and we use template metaprogramming [Veldhuizen 1996] to perform compile time code generation for PDF calculation and other features.

Parameterization and Measure. Renderers often mix path parameterizations, requiring error-prone measure conversion. In our language, the programmer specifies an integrand in a single chosen parameterization (e.g., surface area). They must provide all samples in the same parameterization, via conversion code written in our language. However, it is *not* required to account for measure changes (geometry terms) explicitly because the compiler does it automatically.

Interaction with Deterministic Code. The derivation of probabilities for samples computed according to operations such as ray casting requires the symbolic inversion and differentiation of, e.g., the intersection point. Writing a full ray casting in our language would be prohibitive and violate our goal of interportability. To achieve both mathematical correctness and modularity, we require values coming from external code to be constant in the neighborhood of a sample. This means that, for example, ray casting will pass the vertices of the intersected triangle, and that the intersection coordinates need to be re-computed in our language.

Incremental Paths. In order to make the symbolic expression of path sampling functions tractable, we assume that all paths are sampled sequentially and at each step only a single vertex is sampled. For some special cases, such as tridirectional path tracing (Section 7.4), we provide a construct to sample two or more vertices at the same time. Samplers with global dependencies, such as the Manifold perturbation of Jakob and Marschner [2012], remain future work.

4.3 Approach

Our language focuses on sampling and integration in probabilistic ray tracing. It is embedded in C++ via template metaprogramming and can be used with existing libraries for ray casting and shading. Users are responsible for sampling and path assembly, while the computation of densities (both at the local ray level and global path level, for the current strategy as well as other strategies) and the combination of samples for MIS are handled automatically. For this, users write sampling code and append vertices to a path data structure in our language, but use existing C++ libraries for ray casting and geometric acceleration, as well as for the computation of shading and path throughput. What needs to be (re)written in our language is the importance sampling portion of a shader and the calculation of the intersection location within a visible primitive.

```
// Create camera subpath
RandomSequence<Vertex> camPath;
camPath.Append(sampleCamera)  // Append always called with a
     strategy
camPath.Append(samplePrimaryHit)
while (camPath length is less than maxDepth) {
  camPath.Append(sampleBSDF)
}

Li = 0;
for (pathLen = 2...camPath.Size()) {
  bsdfPath = camPath[1...pathLen + 1]   // first pathLen+1 vertices
  directPath = camPath[1...pathLen]     // first pathLen vertices
  directPath.Append(sampleLight)        // add shadow ray target

  // MIS weights
  combined = combine(bsdfPath, directPath);
  foreach (path in combined) {
    Li += path.weight * integrand of path / path.Pdf();
  }
}
return Li;
```

Fig. 3. Pseudocode for a path tracer with MIS in our language. Integrand and estimator code highlighted in green; sampling code highlighted in blue; and PDF and MIS code highlighted in red.

Our language is centered on five main types of constructs. We provide *sampling primitives* (discrete, continuous, and strategies, which subsume and combine the two) where the programmer writes code that produces multidimensional path samples by transforming a series of 2D uniform random variables into sequences of vertices. Multiple importance sampling is handled through a *combine* primitive that takes care of weight computation. We provide a *path data structure* to which vertices can be appended and which takes care of density computation. It can also handle complex cases, such as in bidirectional path tracing, where vertices are inserted both from the light and from the eye. We clearly separate the definition of an *integrand* from the sampling code for modularity. Our code interacts with deterministic external code via a *locally-constant* construct where values are assumed constant in the neighborhood of a sample. These constructs are then extended to handle Markov chain Monte Carlo approaches such as Metropolis.

The pseudocode in Figure 3 shows the implementation of a path tracer as it would be written in our language. It highlights several of the main primitives of our language. It also shows how our approach differs from a regular path tracer implementation in several ways.

Sampling and PDF. We provide five sampling primitives:

- Random variables generate one dimensional samples (Section 5.1)
- Random vectors generate multi-dimensional samples (Section 5.1)
- Discrete random variables for making discrete choices (Section 5.2)
- Strategies are a compound sampling primitive. They use random variables and random vectors as building blocks (Section 5.3) and can also include discrete distributions. They are used to sample the next vertex in a path.
- A generic data structure (`RandomSequence`) for sampling light paths (Section 5.6). They are constructed from a list of strategies.

All PDF computations are handled by our language. In standard renderers, the programmer must manually account for PDFs, which is error prone, and further complicated by the fact some importance sampling code features a mixture of continuous (e.g. BSDF sampling) and discrete sampling (e.g. choosing a light source).

Whenever generating a sample in our language, the sample is drawn from one of our language primitives. The programmer does not need to write PDF code: all primitives have an automatically derived `Pdf()` method. We use symbolic differentiation to compute the Jacobians needed to generate PDF code (Equation 5).

Combine. In a path tracer with MIS, even with only 2 different sampling distributions, there are 4 necessary PDFs to compute. In standard renderers, this is error prone; it is the responsibility of the programmer to manually account for how each sample is actually sampled. This becomes increasingly complex as the number of combinations increases.

In our language, every sample maintains a record of how it was sampled and has an automatically derived `Pdf()` method. This method can be called on arbitrary samples, which is enabled by symbolic inversion (Equation 6). MIS then becomes simple (Section 5.4); our language offers a convenience `combine()` primitive for this purpose.

Path Data Structure. Our language introduces the `RandomSequence` (Section 5.6) data structure, which we use to store path data. It is the fundamental building block of all algorithm implementations in our language. Random sequences can be extended with new samples, offer the automatically derived `Pdf()` method for computing the PDF of the path, and have methods for slicing, concatenating, and reversing sequences, while maintaining the same `Pdf()` interface on the new sequence. These methods are particularly useful in algorithms like bidirectional path tracing where many paths are sliced and concatenated for MIS.

Constructing a path involves implementing strategies. `sampleCamera`, `samplePrimaryHit`, and `sampleBSDF` are strategies for sampling the camera vertex, obtaining the primary hit vertex, and sampling the previous BSDF and raycasting to obtain the next vertex. One notable advantage of using strategies to construct paths is that they are then reusable in other algorithms.

Random sequences are sampled sequentially so the full density functions (Equation 4) can be built from the primitive operations of 2D/3D function inversion and taking Jacobians. In contrast to typical path tracer implementations, we explicitly sample full paths before combining them or computing the integrand. We feel this is a clearer way of constructing paths.

Separate Integrand and Sampling. Typical path tracer implementations generally mix integrand, sampling, PDF, and MIS code together inside a for loop. Combining all of these elements reduces the clarity of the algorithm and makes it difficult to reuse any of them in other algorithms. In our language we handle these aspects of the algorithm separately. This allows sampling strategies and integrand evaluation code to be reused.

Our language is implemented using C++ template metaprogramming, allowing all necessary inversion and Jacobian computation to be resolved at compile time. That is, the template mechanism is used as a programming language where templates act as functions and types act as values. Under the hood, each symbolic expression is represented as a complex type, and template instantiation is used to transform these symbolic expressions. Template metaprogramming can constrain the possible syntax (e.g. we define real number literals as `1_l`, `2_l`, etc.) but it enables powerful compile time symbolic manipulation, including symbolic inversion and symbolic differentiation, before code generation.

4.4 Scope and Limitations

Our language can handle Monte Carlo rendering methods that can be expressed in a form similar to Equation (2), which proceed by weighted summation of point estimates of an analytic integrand f at sample locations generated probabilistically. Sampling can include discrete and continuous random variables, based on the multidimensional analytic mapping of uniform random variables. This includes most forms of importance sampling, path tracing, bidirectional path tracing, and Markov-chain methods such as Metropolis. We support cases where samples are correlated, in particular when subpaths are reused for different estimates, which allows us to express virtual point light source methods such as instant radiosity [Keller 1997] (see Section 7.6), where light subpaths are reused. We can also support methods such as photon mapping [Jensen 1996] (see Section 7.7), which can be expressed with an additional convolution of the integrand with a density estimation kernel. We can handle adaptive sampling techniques where the number of samples or the importance function depends on previous samples.

Adaptive Approximation of the Integrand. Techniques such as irradiance caching [Ward et al. 1988], lightcuts [Walter et al. 2005], or multi-dimensional adaptive sampling and reconstruction [Hachisuka et al. 2008a] which can be seen as performing adaptive approximations of the integrand, are not supported by our approach, and in particular we cannot offer correctness guarantees. However, parallel versions of irradiance caching (e.g. [Jones and Reinhart 2016; Wang et al. 2009]) that populate the cache ahead of time could probably be expressed with a combination of correlated samples and convolution of the integrand similar to virtual point lights and photon mapping, although we have not implemented it.

Non-Analytic PDFs. We focus on transform sampling techniques for analytic PDFs. We currently cannot handle numerical techniques such as manifold walk sampling [Jakob and Marschner 2012] or woodcock tracking [Woodcock et al. 1965] used in volumetric rendering.

Dirac Delta Functions. Our language currently does not handle Dirac functions, such as point light sources and mirrors. The exception to this is the pinhole camera, which we support for sampling eye subpaths. We assume the ratio between position density on the film plane and the integrand is 1, but do not use an explicit construct to characterize it as a Dirac. These functions could potentially be handled symbolically, by storing each Dirac as a symbolic function and relying on our symbolic simplication to cancel out the same Diracs in the numerator and denominator of a given expression, but this is not implemented at present. We approximate perfect specularity with extremely shiny BSDFs, as detailed later.

Zero PDFs. Our code has the correct expected value only if the user sampling code can generate samples almost everywhere where the integrand is non-zero.

Inversion. Symbolic inversion can be in theory intractable, although our language has so far managed the samplers we have implemented. The system assumes all provided sampling functions are bijective. If a non-bijective function is provided, either the inversion will fail and result in a compile error, or the inversion will succeed but only provide one of the possibly many inverse values.

Volumes. We currently do not support volumetric interactions.

Performance. Both compile time and run time performance of samplers written in our language are below hand optimized implementations.

5 THE DOMAIN SPECIFIC LANGUAGE

We now introduce our embedded domain specific language, Aether. We first illustrate the important constructs of our language with a simple example of Monte Carlo integration with importance sampling, before moving to more advanced features.

5.1 Example: Estimating Irradiance at a Point

Suppose we want to compute the irradiance at a fixed point by using Monte Carlo integration to evaluate the hemispherical integral

$$E = \int_\Omega L_i(\omega) \cos\theta \mathrm{d}\omega.$$

Sampling the Hemisphere. We want to sample the hemisphere using importance sampling according to $\cos\theta$ [Pharr and Humphreys 2010]:

```
// Declare symbolic uniform random variables
variable<1> u1;
variable<2> u2;

auto r = sqrt(u1);
auto phi = 2_l * pi * u2; //2_l is a literal for 2.0
auto cosHemisphere = random_vector<2>(
  r * cos(phi),
  r * sin(phi),
  sqrt(1 - u1)
);
```

The above code looks similar to regular numeric sampling code, but under the hood all of the expressions are symbolic to enable the derivation of the inverse and the PDF. It starts with symbolic uniform random variable `u1` and `u2` and defines the symbolic expression to transform them into a *random variable* (`cosHemisphere`) representing directions on the hemisphere. The `random_vector<N>()` function constructs a vector of random variables, where `2` is the number of uniform random variables on which it depends. In this example, there are 2 uniforms — `u1` and `u2` — and the random vector has 3 outputs, because directions are two-dimensional but encoded with 3 coordinates.

The compiler automatically generates a `Sample()` method, which evaluates this function. More importantly, the compiler also computes the symbolic Jacobian determinant and symbolic inverse, both of which are used in the other automatically generated method, `Pdf()`. This function can not only compute the PDF at a point sampled with the strategy, but also for any given direction. It is fairly straightforward in this simple example, but becomes more valuable for sophisticated samplers.

Compute an Estimate. Using the language provided `Pdf()` method, computing an estimate is simple:

```
// User provided uniform sampler
MyRng rng;

Spectrum total(0);
int N = 10000;

auto myIntegrand = [](const auto& sample) -> Spectrum {
  // user provided regular C++ code to compute Li * |cos(theta)|
}

for (int i = 0; i < N; i++) {
  // Draw a sample
  auto xCos = cosHemisphere.Sample(rng(), rng());

  // Compute the integrand
  auto f = myIntegrand(xCos);

  // Compute the PDF
  auto p = cosHemisphere.Pdf(xCos);

  // Add f(xCos) / p(xCos) to running total
  total += f / p;
}

// Compute the final estimate
auto estimate = total / N;
```

To sample `cosHemisphere` the user provides 2 numbers between 0 and 1 to the `Sample()` method. These uniforms can be obtained from any random number generator, or they can come from a quasi-Monte Carlo sequence. This estimator is correct by construction because Aether uses symbolic differentiation to ensure that the sampling and PDF code are consistent.

For convenience, Aether provides an `Estimator` object construct, which stores the integrand and handles the computation of $\frac{f(x)}{p(x)}$ (and also handles boundary cases, e.g. when $p(x) = 0$). It is especially useful when combining multiple samples, where it also handles computing the necessary weighting values.

5.2 Discrete Random Variables

In addition to continuous random variables, Aether also supports discrete random variables, which are frequently used in rendering algorithms, e.g. when discretely sampling a light source or discretely sampling a component of a multi-layered BSDF.

They are constructed from standard C++ containers, and support the same `Sample()` and `Pdf()` methods as the continuous random variables.

Consider the example of discretely sampling a light source:

```
std::vector<Emitter*> emitters = scene.getEmitters();
// Create a uniform discrete distribution
auto emitterDiscrete = discrete(emitters);

// Sample an emitter
auto em = emitterDiscrete.Sample(rng());

// Compute its probability
auto p = emitterDiscrete.Pdf(em);
```

Aether also supports piecewise constant and piecewise linear distributions, which are useful for enviroment map sampling.

5.3 Sampling Strategies

So far, we have seen simple continuous and discrete random variables. We now introduce *strategies*, the most general sampling construct in Aether, that encapsulate operations such as BSDF importance sampling, light sampling, lens sampling, etc. Strategies combine together continuous random variables and vectors as well as discrete random variables — for instance, for choosing light sources or BSDF layers — while still providing automatic PDF derivation. Strategies are designed to be supplied to random sequences (Section 5.6) to create incrementally sampled paths.

To implement a strategy, the user writes sampling code as usual, but it is encapsulated within a function object with a particular form:

```
struct SamplePointOnLightStrategy {
  template <typename T>
  auto operator()(Context<T>& context,
                  MyRng& rng,
                  const std::vector<Emitter*>& emitters) const {
    // Create a uniform discrete distribution
    auto emitterDiscrete = discrete(emitters);

    // Randomly pick an emitting triangle
    auto emitter = context.Sample(emitterDiscrete, context.
         Uniform1D(rng));

    // random variable for uniform point on emitter
    // (for implementation, see supplemental material)
    auto triPt = emitter.randomPoint();

    // Sample the point
    auto uv = context.Uniform2D(rng);
    return triPt.Sample(uv[0], uv[1]);
  }
};
```

The body of the strategy looks similar to the continuous and discrete sampling code we have already seen, with three exceptions: first, the sampler is wrapped in a function call `operator()`; second, discrete random variables (`emitterDiscrete`) are not sampled directly, but are passed as an argument to `context.Sample()`; and third, `context.Uniform1D()` and `context.Uniform2D()` are used to generate uniforms. The `Context` is an internal component needed for keeping track of discrete choices, as detailed in Section 6, and does not concern the user apart from the above.

5.4 Multiple Importance Sampling

Multiple importance sampling requires evaluating several probability densities for each sample drawn (Equation 2). Suppose we wish to use MIS to combine the cosine hemisphere samples with samples from an analogous uniform hemisphere random variable `uniformHemisphere`. A key language feature that enables this is that the `Pdf()` method of random variables accepts as argument not just samples generated by its own `Sample()` method, but *any* sample. As

a result, evaluating all four densities is easy. Denoting the cosine-weighted sample by `xCos` and the uniform sample by `xUniform`, we simply compute

```
// PDF of xCos if sampled by cosHemisphere
auto pCos = xCos.Pdf(xCos);
// PDF of xCos if sampled by uniformHemisphere
auto pUniformCos = xUniform.Pdf(xCos);
// PDF of xUniform if sampled by uniformHemisphere
auto pUniform = xUniform.Pdf(xUniform);
// PDF of xUniform if sampled by cosHemisphere
auto pCosUniform = xCos.Pdf(xUniform);
```

These two strategies (four densities) can then be combined using a MIS heuristic of our choice. As the samples `xCos` and `xUniform` also store the random variable from which they were sampled, we can use their `Pdf()` methods directly to compute the PDFs. However, no extra storage is needed at run time because all such dependencies are resolved statically.

If the sample provided to `Pdf()` is not of the correct dimension or domain (e.g. querying a light sampler for a direction outside the light), the inverse will return invalid uniforms (outside $[0, 1]$) and `Pdf()` simply returns 0. The MIS heuristic will then ensure correct operation. The programmer does not need to manually check for these cases.

To simplify the process of combining potentially arbitrary numbers of samples, our language provides the `combine()` primitive, which accepts a user defined combining heuristic and any number of samples. The above example can be written more concisely as

```
Estimator<Spectrum> myEstimator(myIntegrand);

// User provided code to compute the MIS weight
auto myWeightFn = [](float pdfA, float pdfB) {
  // e.g. the power heuristic:
  return (pdfA * pdfA) / (pdfA * pdfA + pdfB * pdfB);
};

// Combine samples with power heuristic
auto combined = combine<PowerHeuristic>(xCos, xUniform);

// Accumulate weighted integrand values
total += myEstimator(combined);
```

5.5 Interfacing with Deterministic Code

We want our language to be usable with external code such as ray casting engines. This creates the need to compute densities that depend on data computed outside our language, for which symbolic descriptions are unavailable. To balance the need for such an interface with the need for symbolic derivation, we require that the data coming from the deterministic external code be constant in the neighborhood of a sample. This means, for example, that ray casting cannot directly return an intersection point, but should instead return the triangle's vertices, which are constant in a neighborhood of the intersection, and that the intersection point coordinates must be computed in our language. This ensures that proper derivatives, inverses and densities can be derived symbolically.

Our language provides the `ConstantCall` mechanism for calling external deterministic code. A triangle intersection is implemented as

```
// Intersect ray (p, dir) with the scene and expect a constant as a
      result
Intersection its = context.ConstantCall(raycaster, Ray(p.Value(),
     dir.Value()));

// Compute ray-triangle intersection in our language
auto v0 = constant(its.v0);
auto v1 = constant(its.v1);
auto v2 = constant(its.v2);
auto e1 = v0 - v2;
auto e2 = v1 - v2;
auto N = cross(e1, e2);
auto t = dot(v0 - p, N) / dot(dir, N);
return  p + t * dir;
```

The `Intersection` object returned by the ray casting engine includes the vertices (and other relevant information) of the intersected object. We first cast each vertex to a constant (`constant(its.v0)`, etc.) and implement a standard ray-triangle intersection in our language to obtain the final intersection point. (As it turns out, the Jacobian determinant of this step equals the standard geometry term needed for measure conversions, but the programmer never needs to write it out.) Different importance samplers may require other information, in which case constant per-vertex normals or material properties may need to be included with `Intersection` as well.

5.6 Random Sequences

The random vector introduced above (Section 5.1) has a fixed size and is designed for situations when all its coordinates are sampled at once. Aether offers another important data structure, `RandomSequence`, which supports the creation of incremental sequences of generic random variables, where each element is assumed to depend only on the previous element. Each element of a random sequence is sampled from a strategy. We use this type to represent transport paths. The type of data stored in the sequence is user provided, e.g., a Vertex type representing a point on a surface.

Consider sampling a 2-vertex path segment starting from an emitter. This is implemented by a random sequence of two strategies, `SamplePointOnLightStrategy` (defined above), and `SampleHemiAndIntersectStrategy`, a strategy that picks a cosine-weighted direction, traces a ray starting at the previous path vertex, and computes the intersection:

```
// Strategy for sampling a cosine-weighted direction
// and intersecting it with the scene
struct SampleHemiAndIntersectStrategy {
  template <typename T>
  auto operator()(Context<T>& context,
                  const RandomSequence<Vertex>& path,
                  MyRng& rng,
                  Raycaster& raycaster) const {

    // Sample uniforms between 0 and 1
    auto uv = context.Uniform2D(rng);

    // Sample the outgoing direction
    // cosHemisphere is defined as above
    auto dir = cosHemisphere.Sample(uv[0], uv[1]);

    // Get the previous Vertex from RandomSequence
    auto p = path.Back();

    // Compute intersection for ray (p, dir) as above
    // ...
  }
};

// Create an initially empty path
RandomSequence<Vertex> path;
```

```
// Append a point on a light
SamplePointOnLightStrategy samplePointOnLight;
path.Append(samplePointOnLight, rng, emitters);

// Append an intersection point
SampleHemiAndIntersectStrategy sampleHemiAndIntersect;
path.Append(sampleHemiAndIntersect, rng, raycaster);

// Actually sample the strategies
path.Sample();
```

The fundamental operation of a random sequence is to `Append` new elements to the sequence. When `path.Append(samplePointOnLight, ...)` is called, the `samplePointOnLight` strategy is stored, without being sampled. Random sequences are evaluated lazily. When `Sample()` is called, each strategy is then sampled in sequence with `path` as an argument (along with any other provided arguments). The result is a new `Vertex` sample, which is stored along with the strategy from which it was sampled.

Like the other random variables in the language, random sequences offer a `Pdf()` method. Since each element of the random sequence stores a strategy, which itself has a `Pdf()` method, the PDF of the random sequence is computed by sequentially computing the PDF of each sample against its corresponding strategy (Equation 4). Like all other random variables, the random sequence `Pdf()` method can be used to compute the density of any sequence, not just itself, and hence random sequences can be combined with MIS just as easily.

Our language also provides the functions `slice`, `concat`, and `reverse` for extracting subsequences, concatenating sequences, and reversing their order.

The separation of strategies and random sequences has the additional benefit that strategies can be built to be orthogonal and easily reused: they are not tied to a specific random sequence nor to a particular algorithm. For instance, in the above example, changing the cosine-weighted hemisphere sampling to uniform sampling would be as simple as defining a new strategy based on `uniformHemisphere` and appending that instead. The rendering algorithms described in Section 7 make much use of this freedom.

5.7 Conditional Probability for Metropolis Sampling

Aether also provides constructs for Metropolis-Hastings based sampling algorithms, which require conditional probabilities of mutators that alter existing paths in random ways (Equation 3). Mutations are implemented as functions that take in a random sequence and return a new sequence, in the same manner as strategies. This allows primitive strategies such as BSDF, lens, and emitter samplers to be reused when implementing mutations.

We automatically derive the `ConditionalPdf()` function for computing the conditional PDF of mutating one sample into another:

```
// p(curPath | proposalPath)
auto pCurGivenProposal = ConditionalPdf(myMutation, curPath,
     proposalPath);
// p(proposalPath | curPath)
auto pProposalGivenCur = ConditionalPdf(myMutation, proposalPath,
      curPath);
```

Internally, the conditioned sample is treated as a constant input, and the PDF of the mutation strategy is then evaluated like the non-conditional PDF of a strategy. Please consult Section 7 and the supplemental material for a full implementation, including several different mutations.

Aether provides further convenience constructs to simplify MCMC implementations: a `MarkovChainState` object, which stores a sample, its target density, and its contribution; an `acceptProbability` function, which computes the necessary conditional PDFs and acceptance ratio; and a `Mutation` wrapper that applies a given mutation to the current state to produce a proposal, and uses `acceptProbability` to compute the acceptance ratio for the pair of states.

6 IMPLEMENTATION

6.1 Basic Data Types

The basic data types of Aether are floating point numbers, uniforms, random variables, random vectors, and random sequences.

Uniform Random Variables. The fundamental operation of Aether is to transform uniform random variables (*uniforms*) into more general random variables. *Uniforms* are the only programming variables in the language. They are declared with a unique ID (unique within an expression):

```
variable<1> u1; variable<2> u2;
```

Each uniform declared in this manner instantiates a new type.

Expressions. Basic expressions of Aether are composed of literals, uniforms, and constant parameters. The language includes standard mathematical functions (`sqrt`, `sin`, `cos`, `tan`, etc.) and operators (`*`, `+`, `-`, `\`) for transforming uniform random variables into more complex expressions. Our syntax mimics the standard mathematical format of regular C++. This is achieved by templated operator overloading.

As C++ lacks support for compile-time templatized floats, floating-point literals have to be declared with a special syntax . For example, `2_l` is the compile time literal representation of the floating point number 2.0. All literal numbers in Aether are written in this form. Other floats, with values not known at compile time, must be introduced with `constant()`, but they will not then be simplified.

Expressions are Types. All expressions in Aether are represented by composing types into symbolic tree structures of expressions and subexpressions called *expression templates* [Veldhuizen 1995]. Internally, each expression is represented as a single empty templated type. We use template metaprogramming extensively to manipulate these expressions. We implemented a standard library of compile time containers and algorithms that operate on types, designed specifically for working with large, nested types.

The use of the `auto` keyword is not helpful just for brevity. The expressions are not evaluated immediately, they are represented as templatized symbolic algebraic expression trees, and as such, have complex types (of the form `Expr<Type>`). The exact type of the expressions is not important to the user so `auto` is preferred.

Simplification. We use template metaprogramming to automatically simplify all expressions to a canonical form. Upon creation at compile time, we recursively sort the subtrees of each expression by variable ID, variable degree, size of tree, operator precedence, and lexicographic order. During this process, expressions are simplified where possible using template pattern matching and various simplification rules. This helps reduce the size of the expression tree, and

hence, the size of its type. This improves compile time so we always simplify expressions immediately instead of lazily. When dealing with large type expressions, the language will sometimes avoid simplification rules (e.g. expanding power expressions or large matrix vector products) that would greatly increase the size of the type, which would likely have a negative impact on compile time. The language uses the heuristic of expression size to determine whether a simplification rule should be applied. In these cases, the resulting expression may not be fully simplified.

Vector Expressions. Simple expressions can be combined into vector expressions, which are represented symbolically as typelists of expressions. These provide indexed access and can be transformed with standard vector operations (`dot`, `cross`, `normalize`, `length`, etc.). They are simplified in the same manner as basic expressions. Vector expressions can also be combined into matrix expressions, which are represented symbolically as typelists of vector expressions.

Branching. Aether also provides condition expressions (e.g. `2_l * u1 > 1_l`) and logic operators (`&&`, `||`, `!`). These are used in the language's `pattern` construct for creating symbolic branch expressions (like if/else, but with a mandatory default case), which is a list of (condition expression, value expression) pairs. For example:

```
auto a = // vector expr
auto b = // another vector expr
auto value = pattern(
  // First condition; return a
  when(dot(a, a) – dot(b, b) > 0_l, a)
  , otherwise(b) // default case; return b
);
```

Like simpler expressions, these too are simplified automatically at compile time. We use this construct frequently, e.g. for orienting tangent vectors when constructing a coordinate basis at a surface point or branching over different possible BSDF samplers (the language provides a `CompositeRandomVariable` type to simplify the construction of branches over multiple possible samplers).

Sample. The result of calling `Sample()` on a random variable is a sample, which is itself a random variable: it stores the same expression tree as the sampled random variable, as well as the uniforms that were passed to `Sample()` and the result of evaluating the random variable's expression tree. Since the expression trees are simply type names, there is almost no overhead in copying or storing them (except the storage required for any parameters of the random variable).

6.2 Symbolic PDF Derivation

In order to compute the PDF of a random variable, we require the symbolic Jacobian and the symbolic inverse.

Symbolic Jacobian of a Random Variable. Computing the Jacobian requires partial derivatives. To obtain them, we recursively apply fundamental rules of differentiation to the symbolic expression trees that represent random variables:

```
d(a + b, x) = d(a, x) + d(b, x)
d(a * b, x) = d(a, x) * b + a * d(b, x)
d(c, x) = 0 // c is a constant
d(x, x) = 1
etc.
```

Like all expressions in Aether, each partial derivative is simplified automatically. The partial derivatives are then collected in the symbolic Jacobian and the determinant is computed.

Symbolic Inversion of a Random Variable. To invert an expression, we first recursively decompose any subexpression containing a variable into a new equation. For example, when solving for u in $x = \cos(2 * \pi * u)$, we first decompose to $x = \cos(y_1), y_1 = 2 * \pi * u$, then solve each equation individually, and re-compose the results. This step greatly reduces the size of the expression tree for each equation that needs to be solved, which is beneficial for compile times.

Our solver is pattern matching and rule based. It iteratively attempts to match the equation against a set of patterns until the variable of interest is isolated. If a match in one iteration is successful, the corresponding rule is applied, and the process restarted. This is not guaranteed to terminate, but it is able to successfully solve all the necessary equations for our implemented samplers. Examples of specific patterns include equations with barycentric coordinates, linear systems, and scaled vectors.

If the equation does not match one of these specific patterns, we apply more generic rules. The goal of this heuristic stage is to transform the equation at each step into a form to which we can apply primitive inverse operations. This mainly involves attempting to simplify the equation until there is only a single occurrence of the variable of interest. Examples of these generic patterns include separating constants from variables, expanding variables inside parentheses, and factoring variables. Once there is only a single occurrence of the variable of interest, the solver applies primitive inverse operations (e.g. `sin(u)= x => u = arcsin(x)`) to obtain the final result.

A more complete list of the patterns used by the solver is available in the supplemental material.

Sampling a Strategy. When a strategy is sampled, at run time a new `SamplingContext` object is created and passed to the strategy function. The function is then evaluated like regular sampling code. When calls like `context.Sample(emitterDiscrete, context.Uniform1D(rng))` are encountered, the `SamplingContext` simply calls `rng()` and passes the result to `emitterDiscrete.Sample()`. This style of writing indirect method calls has no impact when sampling, but is essential for computing the PDF.

Density of a Strategy. In order to compute the PDF of a strategy, we need to be able to handle both discrete and continuous random variables together.

Consider the case of the `samplePointOnLight` strategy from Section 5: we discretely sample a light source, then sample a point on it. To compute the PDF of a sample, we need to evaluate both the discrete probability and the continuous PDF. This is difficult, because `samplePointOnLight.Pdf()` needs to first determine which light source was sampled by the discrete random variable before evaluating the corresponding continuous random variable.

Suppose there are two light sources, a `triangleLight` and a `sphericalLight`, and a point in space `p`, sampled from some other source. We want to evaluate `samplePointOnLight.Pdf(p)`.

There are two possibilities: either `p` was sampled on `triangleLight` by its random variable (a uniform triangle) or it was sampled on `sphericalLight` by its random variable (a uniform sphere). But given only `p`, we do not know which light it was sampled on. Instead, we need to try both possibilities and sum the results. That is, we enumerate all possible discrete choices, compute the resulting PDF of each of them and sum the results. This approach of enumeration is similarly used in [Goodman and Stuhlmüller 2014] to compute marginal distributions of a computation.

When `samplePointOnLight.Pdf(p)` is called, a new `PdfContext` object is created to help compute the PDF. The main purpose of `PdfContext` is recording the discrete choices made during this process. It also prevents the user-provided random number generator from drawing uniforms; we do not need uniforms since we are not sampling anything, just computing the PDF. This is why in strategies uniforms are generated with `context.Uniform1D(rng)` instead of simply `rng()`.

The strategy is then run. When attempting to make a discrete choice, e.g. sampling a light source with `context.Sample(emitterDiscrete, context.Uniform1D(rng));`, `emitterDiscrete.Sample()` is not actually called. Instead, the `PdfContext` records that a discrete choice has been reached: it returns the 1st result (`triangleLight`) from the discrete random variable and records its discrete probability (`emitterDiscrete.Pdf(triangleLight)`). The function continues with `triangleLight` as the sampled light source and returns its uniform triangle random variable. We compute the PDF of `p` according to this random variable and multiply it by the previously recorded discrete probability. This is the value of `samplePointOnLight.Pdf(p)` if the `triangleLight` had been discretely sampled.

The `PdfContext` checks its recorded choices and recognizes that only the 1st of 2 possible choices has been evaluated. So the function is run again with the same `PdfContext` object. This time, `context.Sample(emitterDiscrete, context.Uniform1D(rng));` returns the 2nd possible result (`sphericalLight`) for the discrete choice and records its discrete probability (`emitterDiscrete.Pdf(sphericalLight)`). The function continues with `sphericalLight` as the sampled light source and returns its uniform sphere random variable. We compute the PDF of `p` according to this random variable and multiply it by the previously recorded discrete probability. This is the value of `samplePointOnLight.Pdf(p)` if the `sphericalLight` had been discretely sampled.

The `PdfContext` consults its recorded choices again. Both choices have been evaluated; there are no more. The process ends and the final result is the sum of the two PDFs.

The `PdfContext` maintains its record of discrete choices as a stack so this process works even for multiple discrete random variables in the one strategy.

To ensure correctness we require that the strategy is a pure function i.e. the function has no side effects; the result is always the same given the same arguments. Our language assumes this to be the case for all strategies. C++ does not support pure functions so it is the programmer's responsibility to ensure this assumption is true. In particular, care should be taken to avoid any global variables that may introduce side effects.

Computing the PDF of a Random Sequence. Every random sequence maintains the strategies with which it was created. Computing the PDF of a provided random sequence according to this list of strategies involves iteratively computing the PDF of each vertex with respect to the corresponding strategy and multiplying them together.

Multiple Samples in a Strategy. It is possible to return 2 samples from a strategy (instead of just 1), using `sample_tuple()`. This is used for our tridirectional path tracer where 2 vertices are dependent on one another and are sampled together. Handling higher numbers of samples is challenging because the size of the expression trees will quickly become very large, which our solver is not currently equipped to deal with and which has a negative impact on compile time. Extending the language to handle this is left as future work.

7 RESULTS

Using Aether we implemented several Monte Carlo rendering algorithms. Specifically, we implemented a path tracer [Kajiya 1986], a bidirectional path tracer [Veach and Guibas 1994], a path-space Metropolis light transport algorithm [Veach and Guibas 1997], a novel tridirectional path tracer, a gradient-domain path tracer [Kettunen et al. 2015], instant radiosity [Keller 1997], and a probabilistic progressive photon mapper [Knaus and Zwicker 2011]. The algorithms are incorporated into two renderers: embree's example renderer [Wald et al. 2014] and Mitsuba [Jakob 2010]. We reuse the raycasting and integrand evaluation code (e.g. BRDF evaluation) inside the renderers, and write our own code for generating samples. We never call the original PDF evaluation functions in the renderers. The results shown here are generated from our Mitsuba-based renderer. We implemented importance sampling functions for the *diffuse*, *roughplastic*, *roughconductor*, and *roughdielectric* materials in Mitsuba with the Beckmann microfacet distribution. We also implemented light source importance sampling for triangular-mesh-based area lights, spherical area lights [Shirley et al. 1996], and environment lights. We verified the importance sampling PDFs generated by our code against manually derived PDFs. Thanks to the modular nature of the sampling strategies, it is relatively easy to add more material types and light types.

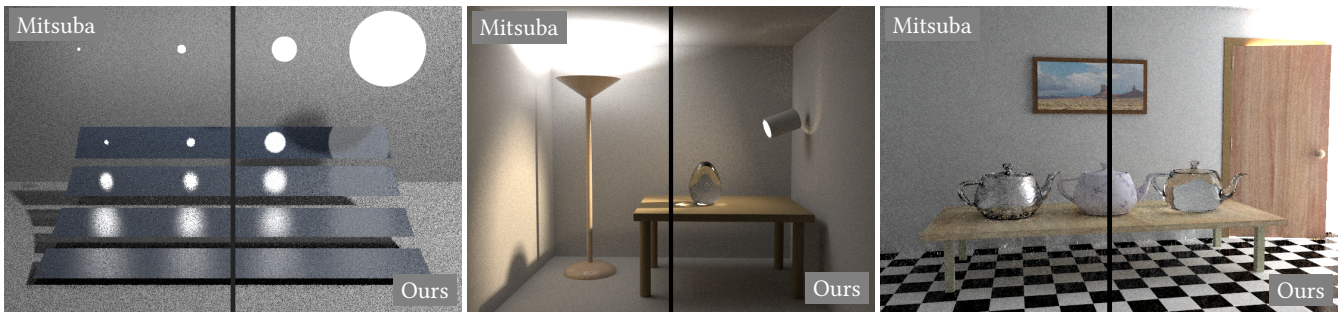
The rest of this section consists of code of our rendering algorithms and our implementation experiences. We verify our implementation by rendering multiple classical test scenes and comparing them to Mitsuba's implementation. The details of the sampling strategies such as `sampCamPos`, `sampBSDF`, etc., can be found in the supplementary material.

We assume the following common inputs to all algorithms:

```
vector<Emitter*> emitters; // List of light sources
UniDist uniDist; // Draws from U(0, 1)
Raycaster raycaster; // Ray–scene intersection
Integrand integrand; // Evaluates f(x)
int maxDepth; // Maximum depth
int x, y; // Pixel coordinate
Film* film; // For splatting contribution
```

7.1 Path Tracer

Figure 5 shows the main logic of our path tracer written in Aether. Although unidirectional path tracers are usually considered simple



(a) path tracing (b) bidirectional path tracing (c) path-space Metropolis light transport

Fig. 4. Three scenes modelled after the test scenes in the original papers of multiple importance sampling, bidirectional path tracing, and Metropolis light transport papers. The scenes contain a variety of different materials, geometry types, and lighting conditions. The images are rendered by the respective light transport algorithm in Mitsuba and our implementation.

```
// Create camera subpath
RandomSequence<Vertex> camPath;
// Append the camera position
camPath.Append(sampCamPos, uniDist);
// Append the primary intersection point for pixel (x, y)
camPath.Append(sampCamDir, uniDist, raycaster, x, y);
// Extend by successive BSDF sampling until max depth
for(; camPath.Size() <= maxDepth;) {
  camPath.Append(sampBSDF, uniDist, raycaster);
}
camPath.Sample();

Spectrum Li(0);
for (int length = 2; length < camPath.Size(); length++) {
  // BSDF sampled path
  auto bsdfPath = camPath.Slice(0, length + 1);
  // BSDF sampled path + direct light sampling
  auto directPath = camPath.Slice(0, length);
  // Direct sampling the light
  directPath.Append(sampEmtDirect, uniDist, emitters);
  directPath.Sample();
  // Combine bsdf path and direct path
  // Returns a list of paths with their MIS weights
  auto combinedList =
    combine<PowerHeuristic>(bsdfPath, directPath);
  // Sum up the contributions
  for (const auto &combined : combinedList) {
    const auto &path = combined.sequence;
    Li += combined.weight *
      (integrand(path) / path.Pdf());
  }
}
return Li;
```

Fig. 5. Our path tracer code

to implement, multiple importance sampling already introduces a certain degree of complexity. In order to compute the MIS weights, it is necessary to compute all 4 combination densities between the BSDF and light source samplers (`bsdfPath`, `directPath`). The `combine` call automatically handles this complication. Note that there is no need for the user to maintain a throughput value during the BSDF sampling loop. Russian roulette [Arvo and Kirk 1990] can be done in the `sampBSDF` strategy using a discrete random variable (Section 5.2), and the probability of termination is automatically handled inside `path.Pdf()` and MIS weight computation. As a side note, most modern path tracers, including Mitsuba and pbrt, ignore the probability of path termination when computing MIS weights. (Note that this does not break correctness as the weights still sum to one.)

We verify our implementation by comparing to the reference implementation in Mitsuba. Figure 4a shows a comparison. The scene also showcases the ability of our language to handle different types of geometry and layered BRDFs.

7.2 Bidirectional Path Tracer

The complexity of bidirectional path tracing is a major driving force of our development of the language. The main logic of our Aether implementation (Figure 6) is only ten lines longer (after removing all the empty lines and comments) than the previous path tracer. The major additions are the sampling of the emitter subpaths and the extra path slicing and concatenation. In contrast, Mitsuba and pbrt's implementations for bidirectional path tracing are significantly longer than their unidirectional path tracers.

Some variants of bidirectional path tracing perform additional direct light source sampling when concatenating the camera and emitter subpaths. Existing implementation techniques lead to the additional complexity spilling out of the relevant samplers, decreasing readability and maintainability. For example, in Mitsuba's implementation, the `sampleDirect` flag has to be checked several times during sampling, PDF computation, integrand evaluation, and MIS computation. In our language, the same is achieved by a simple change in constructing the subpaths (see supplementary material for the code). The modification is self-contained due to automatic handling of PDF computation and the decoupling of sampling and integrand code.

We verify our implementation by comparing to the reference implementation in Mitsuba. Figure 4b shows a comparison. Further code is available in the supplemental material.

7.3 Metropolis Light Transport

The original Metropolis light transport algorithm proposed by Veach and Guibas [1997] is notoriously difficult to implement. To our knowledge, after its introduction in 1997, there was no publicly available implementation until Mitsuba 0.4 was released in 2012. Our language provides constructs that address both main sources of implementation difficulty: the asymmetric Metropolis-Hastings

```
RandomSequence<Vertex> camPath;
// ... sample camera subpath as in the path tracer

// Create emitter subpath
RandomSequence<Vertex> emtPath;
// Randomly sample a light and a position on the light
emtPath.Append(sampEmtPos, uniDist, emitters);
// Sample direction from emitter and intersect with scene
emtPath.Append(sampEmtDir, uniDist, raycaster);
for(; emtPath.Size() <= maxDepth;) {
  emtPath.Append(sampBSDF, uniDist, raycaster);
}
emtPath.Sample();

// Combine subpaths
for (int length = 2; length <= maxDepth + 1; length++) {
  // Collect paths with specified length
  std::vector<RandomSequence<Vertex>> paths;
  for (int camSize = 0; camSize < length; camSize++) {
    const int emtSize = length - camSize;
    // Slice the subpaths and connect them together
    auto camSlice = camPath.Slice(0, camSize);
    auto emtSlice = emtPath.Slice(0, emtSize);
    paths.push_back(camSlice.Concat(reverse(emtSlice)));
  }
  // Combine bsdf path and direct path
  // Returns a list of paths with their MIS weights
  auto combinedList = combine<PowerHeuristic>(paths);
  for (const auto &combined : combinedList) {
    const auto &path = combined.sequence;
    // Compute w*f/p and splats contribution
    film->Record(project(path),
      combined.weight * (integrand(path) / path.Pdf()));
  }
}
```

Fig. 6. Our bidirectional path tracer code

```
auto operator()(Context<T>& context,
    const RandomSequence<Vertex>& path) {
  // ...sample the discrete random variables that
  // determine the no. of vertices to delete/insert.
  // The results are stored in delBegin, delEnd,
  // camInsertLen, and emtInsertLen

  auto camPath = slice(path, 0, delBegin);
  auto emtPath = slice(path, delEnd,
                       path.Size() - delEnd);
  auto rEmtPath = reverse(emtPath);
  // Append vertices to the eye subpath
  for (int i = 0; i < camInsertLen; i++) {
    if (camPath.Size() == 0) {
      camPath.Append(sampCamPos, uniDist);
    } else if (camPath.Size() == 1) {
      camPath.Append(sampCamDir, uniDist, raycaster);
    } else {
      camPath.Append(sampBSDF, uniDist, raycaster);
    }
  }
  // Append vertices to the light subpath
  if (emtInsertLen == 1 && rEmtPath.Size() == 0) {
    // Specialized direct importance sampling
    camPath.Append(sampEmtDirect, uniDist, emitters);
  } else {
    for (int i = 0; i < emtInsertLen; i++) {
      if (rEmtPath.Size() == 0) {
        rEmtPath.Append(sampEmtPos, uniDist, emitters);
      } else if (rEmtPath.Size() == 1) {
        rEmtPath.Append(sampEmtDir, uniDist, raycaster);
      } else {
        rEmtPath.Append(sampBSDF, uniDist, raycaster);
      }
    }
  }
  return camPath.Concat(reverse(rEmtPath));
}
```

Fig. 7. Our bidirectional mutation code

acceptance probabilities, and maintaining the light path data structure. Indeed, our language automatically generates the required conditional PDF code, and provides constructs such as `Slice` and `Concat` for editing the path data structures.

We implement the four mutation strategies proposed by Veach and Guibas: bidirectional mutation, lens perturbation, caustic perturbation, and multi-chain perturbation. For illustration, we show code for the main part of the bidirectional mutation here (Figure 7). Interested readers are referred to the supplementary material for the entire code.

Bidirectional Mutation. The bidirectional mutation strategy is responsible for producing large changes to the path in Metropolis light transport. It first selects a range of the light path to delete. This breaks the light path into a camera subpath and an emitter subpath. It then selects the number of vertices to be inserted for the camera subpath and emitter subpath respectively. We implement the discrete selection process using the discrete random variable construct introduced in Section 5.2. The insertion is done in a bidirectional-path-tracing-like fashion. We then simply `Slice`, `Append`, and `Concat` the paths to form the proposal path.

Lens, Caustics, and Multi-Chain Perturbations. These perturbation strategies attempt to make small changes to the path, then propagate the changes through a chain of specular surfaces. We set a threshold on the roughness of the BSDF, below which we consider the surface to be specular, and follow the specular chain by importance sampling the BSDF and `Append` the vertices. We only sample the transmissive components of the BSDF when the original path is transmissive, and vice versa. After termination, the perturbed path is reconnected to the original using `Slice` and `Concat`. We approximate perfect specularity by extremely shiny BSDFs, for which the procedure yields essentially the same result as Veach's perturbation. The code of these mutation strategies can be found in the supplementary material.

We compare our Metropolis light transport implementation to Mitsuba's implementation. Figure 4c shows a comparison. The Mitsuba rendering uses a perfectly specular glass material.

7.4 Tridirectional Path Tracer

To demonstrate the flexibility of Aether, we introduce an extension to bidirectional path tracing, which we call tridirectional path tracing. As motivation, consider a scene where the camera and the light are placed in separate rooms, and there is only a relatively small aperture connecting the two (e.g. Figure 1). If we apply bidirectional path tracing to such a scene, only those paths where by chance a connection edge (or the camera or emitter subpath) passes through the small aperture will contribute to the image, leading to high variance. Indeed, this challenge was one of the original motivators for Metropolis light transport.

To increase the likelihood of obtaining paths through the small aperture, we extend bidirectional path tracing by sampling a 2-vertex "portal segment" that passes through the small aperture (Figure 8) by construction. We connect each camera prefix segment to each emitter suffix segment as usual, but also connect each camera prefix and emitter suffix segment to the portal segment. Figure 1 and Figure 9 illustrates the reduced variance at equal sample counts.

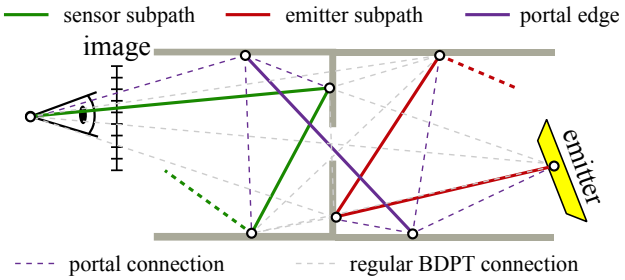
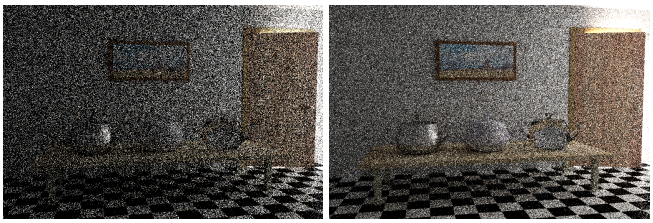


Fig. 8. Tridirectional path sampling. In addition to the standard sensor and emitter subpaths sampled by a bidirectional path tracer (green and red, respectively), we sample two-vertex "portal edge" segments (purple) starting at random locations on user-specified portals. In addition to the standard sensor-emitter connections (gray, dashed), we connect one end of the portal edge to all vertices of the sensor subpath and the other end to all vertices of the emitter subpath.



Bidirectional, 4spp Tridirectional, 4spp

Fig. 9. A comparison of bidirectional and tridirectional path tracing on the Door scene at equal sample counts. The mean square error are 1.032 and 0.434 respectively.

We assume that the geometry of the small aperture is known in advance. Sampling the 2-vertex segment involves first sampling a position x on the surface of the small aperture, then sampling an outgoing direction ω; we intersect 2 rays (x, ω) and $(x, -\omega)$ with the scene to obtain the 2 vertices, one on each side of the small aperture.

The code for slicing and concatenating the paths is shown in Figure 10.

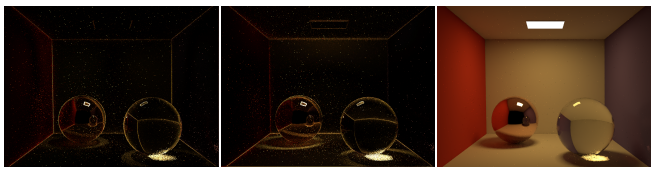
7.5 Gradient-Domain Path Tracing

Gradient-domain path tracing [Kettunen et al. 2015] samples image gradients using pairs of correlated paths, and reconstructs the final image by solving a screened Poisson problem. The path pairs are generated by shifting paths generated by a standard path tracer by one pixel using a deterministic shift mapping, and accumulating differences in throughput modulated by the shift's Jacobian determinant, which requires considerable care to derive. We observe that the shift mapping can be implemented using the lens perturbation, and that the required determinant is the ratio of its conditional densities. Denoting the original path as $\mathbf{x}$ and the shifted path as $\mathbf{x}'$,

$$\left|\frac{\partial \mathbf{x}'}{\partial \mathbf{x}}\right| = \left|\frac{p(\mathbf{x}|\mathbf{x}')}{p(\mathbf{x}'|\mathbf{x})}\right|. \tag{7}$$

```
// segment contains two vertices of the 'portal edge'
// We assume segment never hits the sensor, but it
// could hit the emitter
RandomSequence<Vertex> segment;
// ...
std::vector<RandomSequence<Vertex>> paths;
for (int camSize = 1; camSize < length; camSize++) {
  const int emtSize = length - camSize;
  // Tri-directional subpath
  if (camSize > 1 && emtSize >= 1) {
    // Shorten the sensor and emitter subpaths by 1
    auto camSlc = camPath.Slice(0, camSize - 1);
    auto emtSlc = emtPath.Slice(0, emtSize - 1);
    // Replace with segment
    paths.push_back(
      camSlc.Concat(segment).Concat(reverse(emtSlc)));
  }
  // Shorten the sensor subpaths by 2
  if (sensorSubpathSize > 2) {
    auto camSlc = camPath.Slice(0, camSize - 2);
    auto emtSlc = emtPath.Slice(0, emtSize);
    paths.push_back(
      camSlc.Concat(segment).Concat(reverse(emtSlc)));
  }
  // Shorten the emitter subpaths by 2
  if (emitterSubpathSize >= 2) {
    auto camSlc = camPath.Slice(0, camSize);
    auto emtSlc = emtPath.Slice(0, emtSize - 2);
    paths.push_back(
      camSlc.Concat(segment).Concat(reverse(emtSlc)));
  }
  // Slice and concat without the segment
  auto camSlc = camPath.Slice(0, camSize);
  auto emtSlc = emtPath.Slice(0, emtSize);
  paths.push_back(camSlc.Concat(reverse(emtSlc)));
}
// ...combine the paths as in BDPT
auto combinedList = combine<PowerHeuristic>(paths);
// ...
```

Fig. 10. Our tridirectional path tracer code



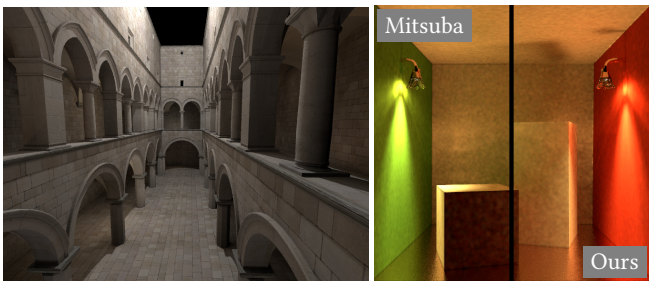
Hori. grad. Vert. grad. Result

Fig. 11. Our implementation of gradient-domain path tracing at 16 samples per pixel, implemented through the conditional probability density of the shift map. The intensity of the gradients are adjusted to have a clearer view.

We implement this compactly using the `ConditionalPdf` function provided by our language. When the shift is not invertible, namely either $p(\mathbf{x}|\mathbf{x}')$ or $p(\mathbf{x}'|\mathbf{x})$ is zero, we set the Jacobian determinant to zero, so that the shifted path has zero contribution. These non-invertibility checks were explicitly handled inside the shift mapping by the author's implementation. Another challenge in gradient-domain path tracing is the multiple importance sampling between base paths and shifted paths. Aether handles this automatically. Code for gradient-domain path tracing can be found in the supplementary material. Figure 11 shows a rendering with both diffuse and specular materials.

7.6 Instant Radiosity

Instant radiosity [Keller 1997] reuses a set of emitter subpaths and connects them to all pixels, at each vertex on the emitter subpath,



(a) instant radiosity (b) progressive photon mapping

Fig. 12. (a) Sponza scene we used for testing our instant radiosity implementation. The number of virtual point lights is 827. (b) A glossy Cornell box used for testing photon mapping. Rendered by Mitsuba's stochastic progressive photon mapping and our probabilistic photon mapping, respectively, with the same number of photon paths and iterations.

the algorithm creates a *virtual point light* to light the scene. It is possible to reuse light paths in our language by storing all emitter subpaths in the form of `RandomSequence`. Gathering the contribution from virtual point lights by slicing and concatenating light paths is then similar to our (bidirectional) path tracing implementation. As in typical implementations of instant radiosity, we also clamp the geometry term at connection to a fixed value by rewriting the integrand evaluation function. In our implementation we use ray casting to check the visibility between the shading point and virtual point light, but it is possible to switch to shadow mapping also by modifying the integrand evaluation function.

Code for instant radiosity can be found in the supplementary material. Figure 12a shows a rendering of the atrium Sponza scene using 827 virtual point lights.

7.7 Progressive Photon Mapping

Photon mapping [Jensen 1996], similar to instant radiosity, generates a global set of emitter subpaths and reuses them across all pixels. For each pair of emitter subpath and camera subpath, the contribution is defined by a kernel function based on the distance between the endpoints of the two subpaths. The kernel is usually parametrized by a radius so that when the distance is larger than the radius, the path pair has zero contribution. Progressive photon mapping is a variant of photon mapping that makes the estimator consistent to the path integral by reducing the kernel radius over multiple iterations (e.g. [Hachisuka et al. 2008b; Knaus and Zwicker 2011]). We implement the probabilistic version of progressive photon mapping [Knaus and Zwicker 2011] in Aether.

We interpret photon mapping as a convolution over a kernel function in path space (e.g. [Georgiev et al. 2012; Hachisuka et al. 2012]), and we construct an unbiased estimator of the convoluted path integral. The photon paths are stored using `RandomSequence` in a first pass. In the second pass the camera subpaths are traced until they hit a diffuse surface, and then concatenated (using `Concat`) with the photon paths if the distance between their endpoints is within the kernel radius. We then compute the contribution of the concatenated path. The integrand function needs to be rewritten to take care of the photon kernel. We also reuse Mitsuba's kd-tree for querying nearby photons within the kernel radius. It is the user's responsibility to ensure the correctness of the kd-tree. After photon gathering, a new iteration starts, and the kernel radius is shrunken as in [Knaus and Zwicker 2011].

Extending our photon mapping implementation to combine with bidirectional path tracing using multiple importance sampling [Georgiev et al. 2012; Hachisuka et al. 2012] requires special care from the users to correctly handle the differences on the dimensionality.

Code for photon mapping can be found in the supplementary material. Figure 12b shows a rendering of a glossy Cornell box. The scene is challenging for unbiased methods like bidirectional path tracing because of the glossy floor and the light sources enclosed in glass.

7.8 Discussion and Limitations

In addition to demonstrating that Aether is capable of succinctly expressing a wide range of rendering algorithms, such as Metropolis light transport, gradient-domain path tracing and progressive photon mapping, we found that it enables easy experimentation with different sampling schemes. Examples include direct light source importance sampling in the bidirectional path tracer and the tridirectional path tracer. From a software engineering perspective, the consistency between the sampling code and the generated PDF code, and the separation of the sampling and integrand evaluation, means the coupling between the code modules is drastically reduced. The user can easily adjust their sampling algorithms without the need to modify several code modules.

Algorithmic Limitations. As discussed in Section 4.4, some modern algorithms, for example the global, non-linear manifold perturbation of Jakob and Marschner [2012] does not fit our assumptions. Also, we currently do not support volumetric interactions. We are interested to explore both directions in the future.

Performance. The performance, both during compilation and run time, of samplers written in our language remains below hand-optimized implementations. A full-rebuild of our Mitsuba-based renderer takes around 7 minutes on a 4-core machine, while the original Mitsuba renderer takes around 2.5 minutes. In our test scenes, out path tracer is around 18 times slower than Mitsuba, and our bidirectional path tracer is around 13.5 times slower than Mitsuba. Our bidirectional path tracer takes 3.5 times more computation time than our path tracer. Our Metropolis light transport implementation is around 19 times slower than Mitsuba. We found that much of the overhead is due to the re-execution of the sampling strategies using `PdfContext` and memory allocation/deallocation during PDF computation.

We feel, regardless, that providing validation targets for careful implementations of algorithms discovered by the rapid exploration enabled by Aether is a bridge to practical applicability. Still, studying the reasons behind and potentially mitigating the performance issues, e.g. through static analysis and compiler optimization techniques, is an important avenue for future work.

8 CONCLUSION

Our language, Aether, dramatically reduces the time required for correct implementation of modern unbiased light transport algorithms, as demonstrated by the concise implementations of path tracing, bidirectional path tracing, path space Metropolis light transport, gradient-domain path tracing, instant radiosity, progressive photon mapping, and the novel tridirectional path tracing. Our language makes the probabilistic code correct by construction, letting the programmer focus on new algorithmic ideas. Key to this ease of use is the automatic derivation of PDF and sample combination code, which results in concise and modular implementations. We hope this efficiency will boost the research community's ability to prototype, test, and validate novel light transport techniques.

ACKNOWLEDGMENTS

Jonathan Ragan-Kelley provided valuable feedback and helped scope the project from its inception. Matt Pharr and Wenzel Jakob gave us useful insights and encouragement. The door scene was modeled by Miika Aittala, Samuli Laine, and Jaakko Lehtinen. The Sponza scene was modeled by Marko Dabrovic. The Cornell box scene with glass lamps was modeled by Toshiya Hachisuka. This work was partially funded by DARPA REVEAL and Toyota.

REFERENCES

James Arvo and David Kirk. 1990. Particle Transport and Image Synthesis. *Comput. Graph. (Proceedings of SIGGRAPH 1990)* 24, 4 (Sept. 1990), 63–66.

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light Transport Simulation with Vertex Connection and Merging. *ACM Trans. Graph.* 31, 6, Article 192 (2012), 10 pages.

Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Daniel Tarlow. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014). Accessed: 2015-12-17.

Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. 2008a. Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2008)* 27, 3, Article 33 (Aug. 2008), 10 pages.

Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. 2008b. Progressive Photon Mapping. *ACM Trans. Graph. (Proceedings of SIGGRAPH Asia 2008)* 27, 5, Article 130 (Dec. 2008), 8 pages.

Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik Wann Jensen. 2012. A Path Space Extension for Robust Light Transport Simulation. *ACM Trans. Graph. (Proceedings of SIGGRAPH Asia 2012)* 31, 6, Article 191 (Nov. 2012), 10 pages.

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

Wenzel Jakob and Steve Marschner. 2012. Manifold Exploration: A Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Trans. Graph.* 31, 4 (2012), 58:1–58:13.

Henrik Wann Jensen. 1996. Global Illumination Using Photon Maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*. 21–30.

Nathaniel L. Jones and Christoph F. Reinhart. 2016. Parallel Multiple-bounce Irradiance Caching. In *Proceedings of the Eurographics Symposium on Rendering (EGSR '16)*. 57–66.

James T. Kajiya. 1986. The rendering equation. In *SIGGRAPH 1986*. 143–150.

Alexander Keller. 1997. Instant Radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. 49–56.

Markus Kettunen, Marco Manzi, Miika Aittala, Jaakko Lehtinen, Frédo Durand, and Matthias Zwicker. 2015. Gradient-Domain Path Tracing. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2015)* 34, 4 (2015).

Claude Knaus and Matthias Zwicker. 2011. Progressive Photon Mapping: A Probabilistic Approach. *ACM Trans. Graph.* 30, 3, Article 25 (May 2011), 13 pages.

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*. 1249–1257.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic Models with Unknown Objects. *Statistical relational learning* (2007), 373.

Jiawei Ou and Fabio Pellacini. 2010. SafeGI: Type Checking to Improve Correctness in Rendering System Implementation. *Computer Graphics Forum (Proceedings of EGSR 2010)* 29, 4 (2010), 1269–1277.

Matt Pharr and Greg Humphreys. 2010. *Physically Based Rendering, Second Edition: From Theory To Implementation* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Peter Shirley, Changyaw Wang, and Kurt Zimmerman. 1996. Monte Carlo Techniques for Direct Lighting Calculations. *ACM Trans. Graph.* 15, 1 (Jan. 1996), 1–36.

Stan Development Team. 2015. *Stan Modeling Language Users Guide and Reference Manual, Version 2.9.0.* http://mc-stan.org/

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph.D. Dissertation. Stanford University.

Eric Veach and Leonidas Guibas. 1994. Bidirectional Estimators for Light Transport.

Eric Veach and Leonidas J. Guibas. 1997. Metropolis Light Transport. In *SIGGRAPH 1997*. 65–76.

Todd Veldhuizen. 1995. Expression templates. *C++ Report* 7, 5 (1995), 26–31.

Todd Veldhuizen. 1996. Using C++ template metaprograms. In *C++ gems*. SIGS Publications, Inc., 459–473.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2014)* 33, 4, Article 143 (July 2014), 8 pages.

Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. 2005. Lightcuts: A Scalable Approach to Illumination. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2005)* 24, 3 (July 2005), 1098–1107.

Rui Wang, Rui Wang, Kun Zhou, Minghao Pan, and Hujun Bao. 2009. An Efficient GPU-based Approach for Interactive Global Illumination. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2009)* 28, 3, Article 91 (July 2009), 8 pages.

Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. 1988. A Ray Tracing Solution for Diffuse Interreflection. *Comput. Graph. (Proceedings of SIGGRAPH 1988)* 22, 4 (June 1988), 85–92.

Wolfram Research, Inc. 2016. *Mathematica.*

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 *Proc. Conf. Applications of Computing Methods to Reactor Problems*, Vol. 557.

Received January 2017; revised April 2017; final version April 2017; accepted April 2017