An Embedded Domain Specific Sampling Language
for Monte Carlo Rendering

by

Luke Anderson

Submitted to the Department of Electrical Engineering and Computer
Science
in partial fulfillment of the requirements for the degree of
Master of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Signature redacted

Author ...........................
Department of Electrical Engineering and Computer Science
April 25, 2017

Signature redacted

Certified by ..........................
Frédo Durand
Professor
Thesis Supervisor

Signature redacted

Accepted by ...........................
Leslie A. Kolodziejski
Chair of the Committee on Graduate Students
DISCLAIMER NOTICE

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

The images contained in this document are of the best quality available.
An Embedded Domain Specific Sampling Language for Monte Carlo Rendering

by

Luke Anderson

Submitted to the Department of Electrical Engineering and Computer Science
on April 25, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract
Implementing Monte Carlo integration requires significant domain expertise. While simple algorithms, 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 a domain specific language for Monte Carlo rendering that offers primitives and data structures for writing concise and correct-by-construction sampling code. The compiler then automatically generates the necessary code for evaluating PDFs and combining multiple samples. Our language focuses on ease of implementation for rapid exploration and research, 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 to implement.

Thesis Supervisor: Frédéric Durand
Title: Professor
Acknowledgments

This thesis is based on work done in collaboration with Tzu-Mao Li, Jaakko Lehtinen, and Frédéric Durand. Without their help and support, it would not have been possible.
Contents

1 Introduction ........................................ 13
  1.1 Prior Work ....................................... 15

2 Mathematical Background ......................... 17
  2.1 Path Samplers and Their Densities ............... 18
  2.2 Case Study: Bidirectional Path Tracing .......... 19

3 Goals and Design .................................. 21
  3.1 Goals ........................................... 21
  3.2 Design Decisions ................................ 22
  3.3 Approach ....................................... 23

4 The Language ....................................... 25
  4.1 Example: Estimating Irradiance at a Point ....... 25
  4.2 Multiple Importance Sampling .................... 27
  4.3 Discrete Random Variables ....................... 28
  4.4 Sampling Strategies ................................ 29
  4.5 Interfacing with Deterministic Code ............... 30
  4.6 Random Sequences ................................ 31
  4.7 Conditional Probability for Metropolis .......... 33

5 Implementation .................................... 35
  5.1 Basic Data Types ................................ 35
    5.1.1 Uniform Random Variables .................. 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.2</td>
<td>Expressions</td>
<td>35</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Vector Expressions</td>
<td>37</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Branching</td>
<td>37</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Samples</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Symbolic PDF Derivation</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Symbolic Jacobian of a Random Variable</td>
<td>38</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Symbolic Inversion of a Random Variable</td>
<td>38</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Sampling a Strategy</td>
<td>39</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Computing the PDF of a Strategy</td>
<td>39</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Computing the PDF of a Random Sequence</td>
<td>41</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Multiple Samples in a Strategy</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>Results</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Path Tracer</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>Bidirectional Path Tracer</td>
<td>45</td>
</tr>
<tr>
<td>6.3</td>
<td>Metropolis Light Transport</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>Tridirectional Path Tracer</td>
<td>49</td>
</tr>
<tr>
<td>6.5</td>
<td>Gradient-Domain Path Tracer</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>Discussion and Limitations</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>57</td>
</tr>
<tr>
<td>A</td>
<td>Code Examples</td>
<td>59</td>
</tr>
<tr>
<td>A.1</td>
<td>BSDF importance sampling</td>
<td>59</td>
</tr>
<tr>
<td>A.1.1</td>
<td>BSDF component sampler</td>
<td>60</td>
</tr>
<tr>
<td>A.2</td>
<td>Emitter sampling</td>
<td>65</td>
</tr>
<tr>
<td>A.3</td>
<td>Sensor sampling</td>
<td>73</td>
</tr>
<tr>
<td>A.4</td>
<td>Bidirectional path tracing with direct light source importance sampling</td>
<td>75</td>
</tr>
<tr>
<td>A.5</td>
<td>Bidirectional mutation</td>
<td>76</td>
</tr>
<tr>
<td>A.6</td>
<td>Caustic perturbation</td>
<td>81</td>
</tr>
<tr>
<td>A.7</td>
<td>Lens perturbation</td>
<td>83</td>
</tr>
</tbody>
</table>
A.8 Multi-chain perturbation ........................................ 85
A.9 Tridirectional path tracing ...................................... 86
A.10 Gradient-domain path tracing ................................. 87

B Solver Patterns .................................................... 93
List of Figures

6-1 A scene modelled after the multiple importance sampling test scene in Veach and Guibas' 1995 paper. Rendered by Mitsuba's path tracer and our path tracer, respectively, with equal sample count. The scene contains sphere shapes and layered BRDFs with different roughnesses, showcasing the ability of our language to handle a variety of different materials and geometry types. ........................... 44

6-2 A scene modelled after the bidirectional path tracing test scene in Veach and Guibas' 1995 paper. Rendered by Mitsuba's bidirectional path tracer and our bidirectional path tracer, respectively, with equal sample count. ........................................ 45

6-3 A scene modelled after the Metropolis light transport test scene in Veach and Guibas' 1997 paper. Rendered by Mitsuba's path space Metropolis light transport and our path space Metropolis light transport, respectively, with equal sample count. .................. 46

6-4 Our path tracer code .................................................. 47

6-5 Our bidirectional path tracer code .................................. 48

6-6 Our bidirectional mutation code ..................................... 49
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.

A comparison of bidirectional and tridirectional path tracing on the Door scene at equal sample counts.

Our tridirectional path tracer code.

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 for clarity.
Chapter 1

Introduction

Monte Carlo integration techniques are widely used in physically based rendering. Implementing simple Monte Carlo algorithms, such as path tracing, is relatively straightforward, but more advanced algorithms, such as bidirectional path tracing or Metropolis light transport, are significantly more challenging and are prone to subtle probability bugs. This is evident from the very small number of available implementations. Even the widely used pbrt did not include full bidirectional path tracing with multiple importance sampling until ten years after its first release. We feel that the difficulty to implement and debug Monte Carlo integration techniques in rendering has seriously hindered the pace of research in this area. As an example, very few articles followed up on Metropolis light transport [17] until more than a decade later when a public implementation finally became available [5].

A major implementation difficulty lies with correct handling of probability densities, both in terms of mathematical correctness and book keeping, by which we mean the care required when generating and combining multiple samples, which are often defined with different parameterizations (measures). In particular, algorithms such as multiple importance sampling and Metropolis require the computation of not only the density 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 densities 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 necessitated large changes in software architecture, including the data structures used for storing path data, and significantly increased the code’s complexity, stemming from the computation of multiple importance sampling weights. Now imagine experimenting with new sampling ideas, such as a tridirectional approach that would trace subpaths from not only the eye and the light source, but also from a known important opening such as a keyhole or pinhole. This would require the proper density 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 a new domain specific language that dramatically simplifies the implementation of unbiased Monte Carlo integration algorithms. 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. Using our language, the programmer writes sampling code. At compile time, the language computes symbolic derivatives and symbolic inverses and automatically generates the sampling code’s corresponding density code. This ensures consistency between a given sample and its density and eliminates the need for explicit measure conversions. The language also generates the code necessary for combining 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. This is particularly advantageous when experimenting with new algorithms. Our focus
is on correctness and not speed, and our language is intended to facilitate research exploration and the generation of reference code.

We plan to make the language publicly available.

1.1 Prior Work

Rendering Systems pbrt [12] and Mitsuba [5] 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 derivation 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 [3] (generative models, inference), webppl [4] (inference), Stan [14] (Bayesian inference), Factorie [9] (factoring graphical models, inference), BLOG [10] (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 [21], 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 [11] is a C++ software library that offers compile time checking of physical dimensions, units, and geometric spaces. This kind of library is orthogonal to our language and could potentially be incorporated into it.
Chapter 2

Mathematical Background

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

$$I_j = \int_{\Omega} h_j(x)f(x)\,d\mu(x)$$

(2.1)

where $\Omega$ 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 $\mu$ [15]. A path $x = \{x_1, x_2, \ldots, x_k\}$ is a sequence of 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 $x_i$ is sampled from a distribution $p_j(x)$, which is one of the $M$ distributions. An estimate of the integral from the paths’ contributions $f(x_i)$ is weighted according to

$$I \approx \frac{1}{N} \sum_{i=1}^{N} W_i(x_i) \frac{f(x_i)}{p_j(x_i)},$$

(2.2)

where the combination weight heuristic $W_i(x_i)$ is a function of all the probability densities $p_1(x_i), p_2(x_i), \ldots, p_M(x_i)$, not just the density of the sampler that actually generated $x_i$.

In a different vein, Markov Chain Monte Carlo methods such as Metropolis Light Transport [17] make use of random walks where the proposed random step from path
x to path y is accepted with probability

$$\min \left\{ 1, \frac{f(y) p_j(x|y)}{f(x) p_j(y|x)} \right\}. \quad (2.3)$$

Here, $p_j(y|x)$ is one of potentially many conditional probability densities used to randomly sample mutations. Implementing mutation samplers and the computation of their conditional probability densities is challenging to the point that few complete implementations of the Metropolis algorithm are known.

### 2.1 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. As each vertex is conditioned only on the previous one, the PDF of the entire path is the product of the individual sampling probabilities:

$$p(x) = p(x_1)p(x_2|x_1)p(x_3|x_2)\ldots \quad (2.4)$$

The standard approach for constructing local importance sampling distributions is to take a 2D uniformly distributed random variable $(u_1, u_2)$ and apply a function to it that warps the 2D unit square onto the (hemi-)sphere of directions, i.e., $\omega = w(u_1, u_2)$. 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, $x = \{x_1, x_2, \ldots\} = S(u_1, u_2, \ldots)$. 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

$$p(\omega) = \sqrt{|\det J^T J|} p(u_1, u_2) \quad (2.5)$$
where \( J = \frac{\partial w}{\partial u_1, u_2} \) is the \( 3 \times 2 \) Jacobian of the mapping from the square to the sphere. Note that evaluating the density at an arbitrary direction \( \omega \) that was not sampled from the same PDF — so that we do not know the \( u_1, u_2 \) that produced \( \omega \) — requires first the inversion

\[
\omega^{-1} = (u_1, u_2).
\]  

(2.6)

Similarly, when computing the density of an entire path \( y \) sampled from another distribution, we must perform the multidimensional inversion \( u_1, u_2, \ldots = S^{-1}(y) \). This reduces to a series of local 2D inversions of the form (2.6).

### 2.2 Case Study: Bidirectional Path Tracing

Concretely, evaluating the estimator (2.2) for a sophisticated path sampler such as bidirectional path tracing [15] requires, in addition to code that produces the actual paths:

- Manually specified density computations for several types of continuous random variables (sensor, light, lens, and BSDF samplers)

- Manually applying surface area measure conversion factors

- Handling special cases when computing full path PDFs (Equation 2.4), such as the primary intersection in the camera subpath and the emitter vertex drawn by a special light sampler from a randomly chosen light source

- Careful enumeration of the discrete choices made in the sampling code to ensure the density code accounts for all the ways a path can actually be generated.
Chapter 3

Goals and Design

3.1 Goals

Correctness By Construction  Estimators written using our language should have the correct expected value. The user may write code that is inefficient and has poor variance, but the expected value should be correct. This means, in particular, that sampling code and the corresponding PDF code should 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. 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 densities 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.

3.2 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 code. It is much simpler than the opposite, since the PDF derivation requires a
simple derivative, while deducing sampling code from a PDF can be arbitrarily hard,
especially for multidimensional 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 addi-
tional tools beyond a normal compiler and we use template metaprogramming [19] to
perform compile time code generation for PDF calculation and other features.

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

Interaction with Deterministic Code  The derivation of densities 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 engine
using our language would be prohibitively difficult and violate our goal of compos-
ability. 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 recomputed using our language.
**Incremental Paths**  We assume that all paths are sampled sequentially so that a vertex depends on at most one other vertex, so that their densities follow Equation 2.4. Samplers with global dependencies, such as Manifold perturbation [6], remain future work.

### 3.3 Approach

Using our language the programmer writes code that produces multidimensional path samples. To produce each element of the path, the programmer writes code that transforms a 2D uniform random variable into a 3D vertex.

The programmer can optionally combine multiple samples using MIS. This process is simplified by the fact that every sample generated by our language records how it was sampled. The programmer can combine samples without manually accounting for how each sample was generated.

From this user provided sampling code, our system automatically deduces the code necessary for computing PDFs and MIS weights of combined samples.

Using the assumption that all paths are sampled sequentially, the full density (Equation 2.4) of a path requires only the inversion and Jacobian of each local 2D/3D transformation. We use symbolic differentiation to compute the Jacobians needed to generate local PDF code (Equation 2.5). We use symbolic inversion to support the computation of PDFs of arbitrary samples (Equation 2.6).
Chapter 4
The Language

We first illustrate the important constructs of our language with a simple example of Monte Carlo integration with importance sampling.

4.1 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

\[ E = \int_{\Omega} L_i(\omega) \cos \theta d\omega. \]

Sampling the Hemisphere

We want to sample the hemisphere using importance sampling according to \( \cos \theta \) [12]:

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

auto r = sqrt(u1);
auto phi = 2_1 * pi * u2; //2_1 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 all of the
expressions are symbolic to enable the derivation of the inverse and the PDF. It
starts with symbolic uniform random variables \( u_1 \) and \( u_2 \) and defines the symbolic
expression to transform them into a random variable \( \cosHemisphere \) representing
directions on the hemisphere. The \texttt{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 — \( u_1 \) and \( u_2 \) — and the random vector
has 3 outputs, because directions are 2 dimensional but encoded with 3 coordinates.

The compiler automatically generates a \texttt{Sample()} method, which evaluates this
function. More importantly, the compiler also computes the symbolic Jacobian de-
terminant and symbolic inverse, both of which are used in the other automatically
generated method, \texttt{Pdf()}. This function can compute not only the PDF at a point
sampled with this random variable, but also for any given direction. This becomes
particularly valuable when combining samples.

**Compute an Estimate** Using the language provided \texttt{Pdf()} method, computing an
estimate is simple:

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

Spectrum total(0);
int N = 10000;

auto myIntegrand = []\((\text{const auto} \& \text{sample})\) \rightarrow \text{Spectrum} \{
  // user provided regular C++ code to compute L_i \times |\cos(\theta)|
}\)

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

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

  // Compute the pdf
  auto p = cosHemisphere.Pdf(x);

  // Add f(x) / p(x) 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. The estimator is correct by construction because our language ensures that the sampler and PDF code are consistent.

For convenience, our language also 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.

## 4.2 Multiple Importance Sampling

Multiple importance sampling requires evaluating several probability densities for each generated sample (Equation 2.2). 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 densities is easy. Denoting the cosine weighted sample by \( x_{\text{Cos}} \) and the uniform sample by \( x_{\text{Uniform}} \), we simply compute

\[
\begin{align*}
\text{auto } &\text{pCos} = x_{\text{Cos}}.Pdf(x_{\text{Cos}}); \\
\text{auto } &\text{pUniformCos} = x_{\text{Uniform}}.Pdf(x_{\text{Cos}}); \\
\text{auto } &\text{pUniform} = x_{\text{Uniform}}.Pdf(x_{\text{Uniform}}); \\
\text{auto } &\text{pCosUniform} = x_{\text{Cos}}.Pdf(x_{\text{Uniform}});
\end{align*}
\]

These four densities can then be combined using an MIS heuristic of our choice. As the samples \( x_{\text{Cos}} \) and \( x_{\text{Uniform}} \) also store the random variable from which they were
sampled, we can use their \( \text{Pdf}() \) 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 \( \text{Pdf}() \) 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 \( \text{Pdf}() \) simply returns 0. 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 \( \text{combine}() \) primitive, which accepts a user defined combining heuristic and any number of samples. The above example can be written more concisely as

\[
\text{Estimator< Spectrum > myEstimator(myIntegrand);} \\
\text{// User provided code to compute the MIS weight} \\
\text{auto myWeightFn = [](float pdfA, float pdfB) {} \\
\text{ // e.g. the power heuristic:} \\
\text{ return (pdfA * pdfA) / (pdfA * pdfA + pdfB * pdfB);} \\
\};} \\
\text{// Combine samples with power heuristic} \\
\text{auto combined = combine<PowerHeuristic>(xCos, xUniform);} \\
\text{// Accumulate weighted integrand values} \\
\text{total += myEstimator(combined);} \\
\]

### 4.3 Discrete Random Variables

In addition to continuous random variables, the language 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 multilayered BSDF.

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

Consider the example of discretely sampling a light source:

\[
\text{std::vector<Emitter*> emitters = scene.getEmitters();} \\
\text{// Create a uniform discrete distribution} \\
\]
auto emitterDiscrete = discrete(emitters);

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

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

The language also supports piecewise constant and piecewise linear distributions, which are useful for environment map sampling.

4.4 Sampling Strategies

So far, we have seen simple continuous or discrete random variables. We now generalize them to strategies, which are random variables that can include both discrete and continuous choices.

For example, consider sampling a point in space by first sampling a light source (discrete), then sampling a point on that light source (continuous). As we will see in Section 5, the PDF derivation is more challenging because we need to consider the inversion of which discrete choice was made.

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

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

        auto emitter = context.Sample(emitterDiscrete, context.Uniform1D(rng));

        auto triPt = emitter.randomPoint();

        return triPt.Sample(context.Uniform1D(rng), context.Uniform1D(rng));
    }
};
```
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()` is used to generate 1D uniforms (`context.Uniform2D()` can be used to generate 2D uniforms). Apart from these details, the `context` is an internal component needed for keeping track of discrete choices (as detailed in Section 5) and does not concern the user.

### 4.5 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 the intersection point coordinates be computed using 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

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

```cpp
// Compute ray-triangle intersection using 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 using our language to obtain the final intersection point. The Jacobian determinant of this step is equal to the standard geometry term needed for measure conversions, but the programmer does not need to manually account for this. 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.

### 4.6 Random Sequences

The random vector introduced above has a fixed size and is designed for situations when its coordinates are sampled at once. The language 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. We use this type to represent light 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:

```cpp
// 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 2.4). 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 \texttt{slice}, \texttt{concat}, and \texttt{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 hemisphere sampling would be as simple as defining a new strategy based on \texttt{uniformHemisphere} and appending that instead. The rendering algorithms described in Section 6 make much use of this freedom.

### 4.7 Conditional Probability for Metropolis

Our language also provides constructs for Metropolis-Hastings based sampling algorithms, which require conditional probabilities of mutators that alter existing paths in random ways (Equation 2.3). 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 \texttt{ConditionalPdf()} function for computing the conditional PDF of mutating one sample into another:

```c++
// 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 6 and Appendix A for a full implementation, including several different mutations.

Our language provides further convenience constructs to simplify MCMC implementations: a \texttt{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.
Chapter 5

Implementation

5.1 Basic Data Types

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

5.1.1 Uniform Random Variables

The fundamental operation of the language 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:

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

Each uniform declared in this manner instantiates a new type.

5.1.2 Expressions

Basic expressions of the language 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, \texttt{2.1} is the compile time literal representation of the floating point number 2.0. All literal numbers in the language are written in this form. Other floats, with values not known at compile time, must be introduced with \texttt{constant()}, but they won't then be simplified.

**Expressions are Types** All expressions in the language are represented by composing types into symbolic tree structures of expressions and subexpressions called \textit{expression templates} [18]. 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 \texttt{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 \texttt{Expr<Type>}). The exact type of the expressions is not important to the user so \texttt{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
5.1.3 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.

5.1.4 Branching

The language also provides condition expressions (e.g. \( 2 \_1 \cdot u_1 > 1 \_1 \)) 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:

```cpp
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.

5.1.5 Samples

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. Note that 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).

5.2 Symbolic PDF Derivation

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

5.2.1 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:

\[
\frac{d(a + b, x)}{dx} = \frac{d(a, x)}{dx} + \frac{d(b, x)}{dx}
\]

\[
\frac{d(a \times b, x)}{dx} = \frac{d(a, x)}{dx} \times b + a \times \frac{d(b, x)}{dx}
\]

\[
\frac{d(c, x)}{dx} = 0 \quad // \quad c \text{ is a constant}
\]

\[
\frac{d(x, x)}{dx} = 1
\]

etc.

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

5.2.2 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 \( x = \cos(2 \times \pi \times u) \), we first decompose to \( x = \cos(y_1), y_1 = 2 \times \pi \times u \), then solve each equation individually, and recompose 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. \( \sin(u) = x \Rightarrow u = \arcsin(x) \)) to obtain the final result.

A more complete list of the patterns used by the solver is available in Appendix B.

### 5.2.3 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.

### 5.2.4 Computing the PDF 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 4: 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 PDF 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 2 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 2 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 don't 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 [4] 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 maintaining a record of the discrete choices made during this process. It also prevents the user provided random number generator from drawing uniforms; we don't 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 PDF (`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 PDF. This is the value of `samplePointOnLight.Pdf(p)` had `triangleLight` been discretely sampled.

The `PdfContext` consults its record of discrete 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 PDF (emitterDiscretePdf(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 PDF. This is the value of samplePointOnLight.Pdf($p$) had sphericalLight been discretely sampled.

The PdfContext consults its record of discrete 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.

5.2.5 Computing the PDF of a Random Sequence

Every random sequence maintains the strategies from which it was sampled. 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 its corresponding strategy and multiplying them together.

5.2.6 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 in our tridirectional path tracer where 2 vertices are dependent on one another and are sampled together. We currently treat this as a special case. Handling general pairs of samples and higher numbers of samples is challenging because the size of the expression trees quickly become very large, which has a negative impact
on compile time. Our solver is not currently equipped to deal with this. Extending the language to handle this is left as future work.
Chapter 6

Results

Using our language we implemented several Monte Carlo rendering algorithms. Specifically, we implemented a path tracer [7], a bidirectional path tracer [16], a path space Metropolis light transport algorithm [17], a novel tridirectional path tracer, and a gradient-domain path tracer [8]. The algorithms are incorporated into two renderers: embree’s example renderer [20] and Mitsuba [5]. 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, rough-conductor, 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 [13] 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 Appendix A.

43
Mitsuba and Ours

Figure 6-1: A scene modelled after the multiple importance sampling test scene in Veach and Guibas' 1995 paper. Rendered by Mitsuba's path tracer and our path tracer, respectively, with equal sample count. The scene contains sphere shapes and layered BRDFs with different roughnesses, showcasing the ability of our language to handle a variety of different materials and geometry types.

We assume the following common inputs to all algorithms:

```cpp
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
```

### 6.1 Path Tracer

Figure 6-4 shows the main logic of our path tracer. Although unidirectional path tracers are usually considered simple 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 [1] can be handled in the sampBSDF strategy using a discrete random variable (see Section 4.3), and the probability of termination is automatically accounted for inside path.Pdf() and for
Figure 6-2: A scene modelled after the bidirectional path tracing test scene in Veach and Guibas’ 1995 paper. Rendered by Mitsuba’s bidirectional path tracer and our bidirectional path tracer, respectively, with equal sample count.

the MIS weight computations. 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 6-1 shows a comparison. The scene also showcases the ability of our language to handle different types of geometry and layered BRDFs.

6.2 Bidirectional Path Tracer

The complexity of bidirectional path tracing was a major motivation for our development of the language. The main logic of our implementation (Figure 6-5) 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 imple-
Figure 6-3: A scene modelled after the Metropolis light transport test scene in Veach and Guibas’ 1997 paper. Rendered by Mitsuba’s path space Metropolis light transport and our path space Metropolis light transport, respectively, with equal sample count.

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. Using our language, the same is achieved by a simple change in constructing the subpaths (see Appendix A 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 6-2 shows a comparison. Further code is available in Appendix A.

6.3 Metropolis Light Transport

The original Metropolis light transport algorithm [17] 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 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
Figure 6-4: Our path tracer code

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 6-6). Interested readers are referred to Appendix A 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 4.3. 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. 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. We approximate perfect specularity by extremely shiny BSDFs, for which the procedure yields essentially the same result as Veach’s perturbation. The code for these mutation strategies can be found in Appendix A.

We compare our Metropolis light transport implementation to Mitsuba’s implementation. Figure 6-3 shows a comparison. The Mitsuba rendering uses a perfectly

```cpp
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(o, 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;
        auto &weight = combined.weight;
        // Compute w*f/p and splats contribution
        film->Record(project(path),
            weight * (integrand(path) / path.Pdf()));
```
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) {
      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));
}

Figure 6-6: Our bidirectional mutation code

specular glass material.

6.4 Tridirectional Path Tracer

To demonstrate the flexibility of our approach, we introduce an extension to bidirectional path tracing, which we call tridirectional path tracing. As motivation, consider a scene where the camera and the emitter are placed in separate rooms, and there is only a relatively small aperture connecting the two. If we apply bidirectional path tracing to such a scene, only those paths where by chance a connection edge (or the sensor 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
Figure 6-7: 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.

Figure 6-7: 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.

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 6-7) 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 6-8 illustrates the reduced variance at equal sample counts.

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 $\omega$; we intersect 2 rays $(x, \omega)$ 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 6-9.
6.5 Gradient-Domain Path Tracer

Gradient-domain path tracing [8] 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 $x$ and the shifted path as $x'$,

$$\frac{\partial x'}{\partial x} = \frac{p(x|x')}{p(x'|x)} \cdot (6.1)$$

We implement this compactly using the conditionalPdf function provided by our language. When the shift is not invertible, namely either $p(x|x')$ or $p(x'|x)$ 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 in the original author's implementation. Another challenge in gradient-domain path tracing is the multiple importance sampling between base paths and shifted paths. Our PDF code handles this automatically. Code for gradient-domain path tracing can be found in Appendix A. Figure 6-10 shows a rendering with both diffuse and specular
Figure 6-9: Our tridirectional path tracer code materials.
Figure 6-10: 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 for clarity.
Chapter 7

Discussion and Limitations

In addition to demonstrating that our language is capable of succinctly expressing a wide range of rendering algorithms, such as Metropolis light transport and gradient-domain path tracing, 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 Some modern algorithms, particularly biased methods based on density estimation, cannot currently be directly implemented using our language. These include methods in the photon mapping family, e.g., [2]. In addition, the global, non-linear manifold perturbation [6] does not fit our assumptions. We are interested to explore both directions in the future. Also, we currently do not support volumetric interactions.

Performance The performance, at both compile time and run time, of samplers written using our language remains below that of 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, our 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 slow 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.

Regardless, we feel that providing validation targets for careful implementations of algorithms discovered by the rapid exploration enabled by our language 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.
Chapter 8

Conclusion

Our language 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, 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.
Appendix A

Code Examples

We provide in this section the code snippets used by the rendering algorithms we implemented using our domain specific language.

A.1 BSDF importance sampling

The following strategy is responsible for BSDF importance sampling:

```cpp
struct sample_bsdf_t {
    template <typename T>
    auto operator()(Context<T>& context,
                    const RandomSequence<Vertex>& path,
                    UniDist& uniDist,
                    const Raycast& raycaster,
                    const uint32_t componentMask) const
    {
        const Intersection &its = path.Back().Get(intersection_);
        // Transform the incoming direction to local coordinate
        auto shadingFrame = make_random_frame(its);
        auto wi = constant((path[path.Size() - 2].Value() -
                             path[path.Size() - 1].Value()).normalized());
        auto wiLocal = to_local(shadingFrame, wi);

        // getBSDF() returns nullptr if the intersection is not valid
        const BSDF *bsdf = its.getBSDF();
        // Produce a bsdf sampler
        auto bsdfSampler = bsdf == nullptr ? bsdf_component_sampler() :
            bsdf->makeSampler(wiLocal.Value(), its, componentMask);

        // Do bsdf importance sampling and transform the outgoing direction
        auto woLocal = bsdfSampler.Sample(wiLocal, context, uniDist);
        auto wo = to_world(shadingFrame, woLocal);

        bool sampleNext = path.Back().Valid() && its.isValid();
        if (sampleNext && rrDepth != -1 && path.Size() > rrDepth) {
            // Russian roulette
            const Real rrProbability =
                std::min((Real) bsdf->getAlbedo(its).max(),(Real) 0.95);
auto sampleNextRV = discrete_dynamic(std::vector<int>{true, false},
std::vector<Real>{rrProbability, 1.f - rrProbability});
sampleNext = context.Sample(sampleNextRV, context.UniformID(uniDist));

// Trace ray and intersect with scene
auto p = constant(path.Back().Value());
const Ray ray(to_point(p.Value()), to_vector(wo.Value()), its.time);
const Intersection hit = context.ConstantCall(raycaster, ray, sampleNext);
// Do symbolic intersection
auto pt = intersect(hit, p, wo);

// If sampleNext == false, invalidates the sample
return optional_sample(
    sampleNext
    , pt
    , intersection_ = hit
);

// Depth where we begin to do russian roulette
// If set to -1, RR is disabled
const int rrDepth;

A.1.1 BSDF component sampler

The bsdfSampler used in sample_bsdf is of type bsdf_component_sampler (where the
type bsdf_composite_t is an aggregate of all BSDF types we support):

```cpp
struct bsdf_component_sampler {
  void AddComponent(const bsdf_composite_t &component, const Real weight) {
    components.push_back(component);
    weights.push_back(weight);
  }

  template <typename Wi, typename ContextT>
  auto Sample(const Wi& wi, Context<ContextT>& context, UniDist &uniDist) const {
    if (components.size() == 0) {
      auto component = bsdf_composite_t{cosine_hemisphere_sampling{}};
      return component.Sample(wi, context, uniDist);
    }
    auto componentSampler = discrete_dynamic(components, weights);
    auto component = context.Sample(componentSampler, context.UniformID(uniDist));
    return component.Sample(wi, context, uniDist);
  }

  std::vector<bsdf_composite_t> components;
  std::vector<Real> weights;
};
```
An example of makeSampler for Mitsuba’s roughplastic material, which is a two-layer BRDF, weighted by a transmittance function:

class RoughPlastic : public BSDF {
    // ...
    // makeSampler is a virtual function of the BSDF class
    // componentMask can be used to determine the type of
    // BSDF we want to sample (e.g. reflect/refract only)
    bsdf_component_sampler makeSampler(
        const Vector &wiLocal, const Intersection &its,
        const uint32_t componentMask) const {
        // roughness value
        Real alpha = m_alpha->eval(its).average();
        // Evaluate transmittance function
        Real probSpecular = 1 -
            m_externalRoughTransmittance->eval(Frame::cosTheta(wiLocal), alpha);
        probSpecular = (probSpecular*m_specularSamplingWeight) /
            (probSpecular*m_specularSamplingWeight +
            (1-probSpecular) * (1-m_specularSamplingWeight));
        // Build component sampler
        bsdf_component_sampler sampler;
        if (probSpecular < 1) {
            sampler.AddComponent(cosine_hemisphere_sampling{}, Real(1 - probSpecular)) ;
        }
        if (probSpecular > 0) {
            sampler.AddComponent(beckmann_sampling{
                alpha, 1.0, beckmann_sampling::REFLECT}, Real(probSpecular));
        }
        return sampler;
    }
    // ...
    ref<Texture> m_alpha; // roughness
    ref<RoughTransmittance> m_externalRoughTransmittance;
    Float m_specularSamplingWeight;
    // ...
}
The `cosine_hemisphere_sampler_t` functor importance samples a cosine hemisphere:

```cpp
struct cosine_hemisphere_sampler {
  template <typename Wi, typename ContextType>
  auto Sample(const Wi&, Context<ContextType>& context, UniDist &uniDist) const
  {
    // Importance sample cosine hemisphere
    auto r = sqrt(one - u1);
    auto phi = two * pi * u2;
    auto x = r * cos(phi);
    auto y = r * sin(phi);
    auto z = sqrt(u1);
    auto wo = make_random_vector(make_random_var(x), make_random_var(y),
                                  make_random_var(z));
    auto uv = context.Uniform2D(uniDist);
    return wo.Sample(uv[0], uv[1]);
  }

  inline bool operator==(const cosine_hemisphere_sampler &)
  { return true; }
};
```
The `beckmann_sampler_t` functor importance samples a micronormal based on the Beckmann distribution, then reflects/refracts the incoming direction:

```cpp
struct beckmann_sampler {
    enum Mode {
        REFLECT,
        REFRACT
    };

    auto SampleMicroNormal() const {
        // Importance sampling the Beckmann distribution
        // http://www.graphics.cornell.edu/~bjw/microfacetbsdf.pdf
        // Equation 28 and 29
        auto phi = two * pi * u2;
        auto tanThetaSqr = constant(- alpha * alpha) * make_random_var(log(one - u1) );
        auto cosThetaSqr = make_random_var(one) /
                           sqrt(make_random_var(one) + tanThetaSqr);
        auto sinTheta = sqrt(make_random_var(one) - sq(cosTheta));
        return make_random_vector(
            sinTheta * cos(phi),
            sinTheta * sin(phi),
            cosTheta
        );
    }

    template <typename Wi, typename ContextT>
    auto Sample(const Wi& wi, Context<ContextT>& context, UniDist &uniDist) const
    {
        auto uv = context.Uniform2D(uniDist);
        auto m = SampleMicroNormal().Sample(uv[0], uv[1]);

        // Branching construct in our language: if this is a reflection sampler then
        // reflect the micronormal, otherwise do refraction
        auto dir = pattern(
            when(constant(mode) == constant(REFLECT), reflect(wi, m)),
            otherwise(refract(eta, wi, m))
        );
        return sample(dir);
    }

    inline bool operator==(const beckmann_sampler &other) const {
        // Necessary operator for bsdf composite
        return alpha == other.alpha && eta == other.eta && mode == other.mode;
    }

    Real alpha;
    Real eta;
    Mode mode;
};
```
The `reflect` and `refract` functions used in the sampling procedure:

```cpp
template<typename Wi, typename M>
auto reflect(const Wi &wi, const M &m) {
    return two * dot(wi, m) * m - wi;
}

template<typename Wi, typename M>
auto refract(const Real eta, const Wi &wi, const M &m) {
    // http://www.graphics.cornell.edu/~bjw/microfacetbsdf.pdf
    // Eq 40, note that there is a typo and the eta inside the square root should be squared
    auto c = dot(wi, m);
    /* eta = eta_interior / eta_exterior
       ceta = eta_incoming / eta_transmittance
       if wi[2] > 0, we are at exterior, so ceta = 1/eta, otherwise ceta = eta */
    auto ceta = constant(at<2>(wi).Value() > 0 ? (Real(1) / eta) : eta);
    auto dsq = makerandom_var(one) + sq(ceta) * (sq(c) - make_random_var(one));
    auto d = sqrt(dsq);
    auto e = pattern(when(at<2>(wi) > make_random_var(zero), d), otherwise(-d));
    auto ret = (ceta * c - e) * m - ceta * wi;
    return ret;
}
```
A.2 Emitter sampling

The following strategies are responsible for emitter sampling. `direct_sample_emitter_t` is used for direct light importance sampling, `sample_position_on_emitter_t` is used for sampling a position on a light source, and `sample_direction_from_emitter_t` samples a direction from a light source after the position has been sampled.

```
struct direct_sample_emitter_t {
  template <typename T>
  auto operator()(Context<T>& context,
                  const RandomSequence<Vertex>& path,
                  UniDist& uniDist,
                  const refvector<Emitter> &emitters) const {
    auto emitters_rv = discrete_dynamic(emitters);
    // Uniformly choosing an emitter
    auto emitter = context.Sample(emitters_rv, context.Uniform1D(uniDist));
    auto sampler = emitter->makeSampler();
    // Importance sample a point on the emitter
    auto pt = sampler.Sample();
    emitter_sampling_tag::Direct{}, path.Back(), context, uniDist);
    return sample(
      pt,
      emitter_ = emitter.get()
    );
  }
};
```

```
struct sample_position_on_emitter_t {
  template <typename T>
  auto operator()(Context<T>& context,
                  const RandomSequence<Vertex>& path,
                  UniDist& uniDist,
                  const refvector<Emitter> &emitters,
                  const Float &time) const {
    auto emitters_rv = discrete_dynamic(emitters);
    auto emitter = context.Sample(emitters_rv, context.Uniform1D(uniDist));
    auto sampler = emitter->makeSampler();
    // Sample a position from the emitter
    auto pt = sampler.Sample();
    emitter_sampling_tag::Position{}, context, uniDist, time);
    return sample(
      pt,
      emitter_ = emitter.get()
    );
  }
};
```
struct sample_direction_from_emitter_t {
  template <typename T>
  auto operator()(Context<T>& context,
  const RandomSequence<Vertex>& path,
  UniDist& uniDist,
  const Raycast& raycaster) const {
      // Obtain the sampled emitter
      auto emitter = path.Back().Get(emitter_);
      auto sampler = emitter != nullptr ? emitter->makeSampler() :
      emitter_composite_t{};
      // Sample a direction from emitter
      auto wo = sampler.Sample(
        emitter_sampling_tag::Direction{}, path.Back(), context, uniDist);
      // Trace ray and intersect with scene
      auto p = constant(path.Back().Value());
      const Intersection &its = path.Back().Get(intersection_);
      const Ray ray(to_point(p.Value()), to_vector(wo.Value()), its.time);
      const Intersection hit = context.ConstantCall(raycaster, ray);
      // Symbolic intersection
      auto pt = intersect(hit, p, wo);
      return sample(
        pt
        , intersection_ = hit
      );
  }
};

Declarations for the triangle mesh emitter, environment map emitter, and sphere emitter:

namespace emitter_sampling_tag {
  struct Direct {};
  struct Position {};
  struct Direction {};
}

struct trimesh_emitter {
  trimesh_emitter(): trianglePtrs(nullptr), triangleWeights(nullptr), positions (nullptr) {}

  trimesh_emitter(const std::vector<const Triangle*> *trianglePtrs,
  const std::vector<mclib::Real> *triangleWeights,
  const Point *positions,
  const Normal *normals) :
  trianglePtrs(trianglePtrs), triangleWeights(triangleWeights), positions( positions), normals(normals) {}
}

template <typename ContextType>
auto SampleDirect(const Vertex &, mclib::Context<ContextType>& context,
mitsuba::UniDist &uniDist) const;

template <typename ContextType>
auto SamplePosition(mclib::Context<ContextType>& context, mitsuba::UniDist & uniDist, const Float &time) const;

template <typename ContextType>
auto SampleDirection(const Vertex &, mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist) const;

const std::vector<const Triangle*> *trianglePtrs;
const std::vector<Real> *triangleWeights;
const Point *positions;
const Normal *normals;

struct envmap_emitter {
  envmap_emitter() : emitter(nullptr), distribution2d(nullptr), boundSphereRadius(0.f) {}  
  envmap_emitter(const Emitter *emitter, const Distribution2D *distribution2d, const mitsuba::Point &boundSphereOrigin, const Real boundSphereRadius) : 
    emitter(emitter), distribution2d(distribution2d), boundSphereOrigin(boundSphereOrigin), boundSphereRadius(boundSphereRadius) {}  
}

template <typename ContextType>
auto SampleDirect(const Vertex &, mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist) const;

template <typename ContextType>
auto SamplePosition(mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist, const Float &time) const;

template <typename ContextType>
auto SampleDirection(const Vertex &, mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist) const;

const Emitter *emitter;
const Distribution2D *distribution2d;
mitsuba::Point boundSphereOrigin;
mclib::Real boundSphereRadius;
};

struct sphere_emitter {
  sphere_emitter() {}  
  sphere_emitter(const mitsuba::Point &origin, const mclib::Real radius) : origin(origin), radius(radius) {} 
}

template <typename ContextType>
auto SampleDirect(const Vertex &, mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist) const;

template <typename ContextType>
auto SamplePosition(mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist, const Float &time) const;

template <typename ContextType>
auto SampleDirection(const Vertex &, mclib::Context<ContextType>& context, mitsuba::UniDist &uniDist) const;

mitsuba::Point origin;
mclib::Real radius;
Definitions for the triangle mesh emitter, environment map emitter, and sphere emitter:

```cpp
struct uniform_triangle_t {
    auto operator()(const Vector3& tri_v0, const Vector3& tri_v1, const Vector3& tri_v2) const {
        auto v0 = constant(tri_v0);
        auto v1 = constant(tri_v1);
        auto v2 = constant(tri_v2);

        constexpr auto a = sqrt(u1);
        constexpr auto b1 = one - a;
        constexpr auto b2 = u2 * a;

        return b1 * v0 + b2 * v1 + (one - b1 - b2) * v2;
    }

    constexpr auto operator()() const {
        constexpr auto a = sqrt(u1);
        return make_random_vector(make_random_var(one - a), make_random_var(u2 * a)) ;
    }
};

constexpr uniform_triangle_t uniform_triangle{};
```

```cpp
using emitter_composite_t = CompositeRandomVar<trimesh_emitter, envmap_emitter, sphere_emitter>;

// Defer emitter_composite_t::Sample to the corresponding operators
template <>
struct SampleCall<trimesh_emitter> {
    template <int I, typename Cond, typename ContextType>
    auto operator()(const trimesh_emitter& emitter, _int<I>, Cond, mitsuba::emitter_sampling_tag::Direct,
        const Vertex &previousVertex, Context<ContextType>& context, mitsuba::UniDist &uniDist) const {
        return emitter.SampleDirect(previousVertex, context, uniDist);
    }

    template <int I, typename Cond, typename ContextType>
    auto operator()(const trimesh_emitter& emitter, _int<I>, Cond, mitsuba::emitter_sampling_tag::Position,
        mclib::Context<ContextType>& context, UniDist &uniDist, const mitsuba::Float &time) const {
        return emitter.SamplePosition(context, uniDist, time);
    }

    template <int I, typename Cond, typename ContextType>
    auto operator()(const trimesh_emitter& emitter, _int<I>, Cond, mitsuba::emitter_sampling_tag::Direction,
        const Vertex &previousVertex, Context<ContextType>& context, mitsuba::UniDist &uniDist) const {
        return emitter.SampleDirection(previousVertex, context, uniDist);
    }
};

// .. omit corresponding definitions for other emitters
```
auto trimesh_emitter::SampleDirect(const Vertex &prevVtx, Context<ContextType>& context, UniDist &uniDist) const {
    // For triangle mesh these two sampling procedures are the same
    return SamplePosition(context, uniDist, prevVtx.Get(intersection_).time);
}

template<typename ContextType>
auto trimesh_emitter::SamplePosition(Context<ContextType>& context, UniDist &uniDist, const Float &time) const {
    // Limitation: does not deal with moving triangles
    // Build the discrete array for sampling triangles
    auto rv = discrete_dynamic(*trianglePtrs, *triangleWeights);
    auto trianglePtr = context.Sample(rv, context.Uniform1D(uniDist));
    // Transform from Mitsuba's Point type to our internal Vector type
    const mclib::Vector3 vO = to_vector3(positions[trianglePtr->idx[0]]);
    const mclib::Vector3 vi = to_vector3(positions[trianglePtr->idx[1]]);
    const mclib::Vector3 v2 = to_vector3(positions[trianglePtr->idx[2]]);
    auto uv = context.Uniform2D(uniDist);
    // Uniformly sample on a triangle
    auto pt = uniform_triangle(vO, vi, v2).Sample(uv[0], uv[1]);
    // Fill intersection information for integrand evaluation and upcoming sampling
    Intersection its;
    // Important to show that this intersection is valid
    its.t = 0.f;
    its.p = to_point(pt.Value());
    its.geoFrame = Frame(to_normal((v1 - v0).cross(v2 - v0).normalized()));
    if (normals == nullptr) {
        its.shFrame = its.geoFrame;
    } else {
        // Do phong normal interpolation
        const Normal n0 = normals[trianglePtr->idx[0]];
        const Normal n1 = normals[trianglePtr->idx[1]];
        const Normal n2 = normals[trianglePtr->idx[2]];
        const Float a = sqrt(uv[0]);
        const Float b0 = 1.f - a;
        const Float b1 = uv[1] * a;
        its.shFrame = Frame(b0 * n0 + b1 * n1 + (1.f - b0 - b1) * n2);
    }
    its.time = time;
    its.isTriangle = true;
    return sample(pt,
                   vO_, v0,
                   v1_, v1,
                   v2_, v2,
                   is_triangle_ = Real(true),
                   intersection_ = its);
}

template<typename ContextType>
auto trimesh_emitter::SampleDirection(const Vertex &prevVtx, Context<ContextType>& context, UniDist &uniDist) const {
    auto uv = context.Uniform2D(uniDist);
    // Sample from cosine hemisphere and transform to world coordinate
    auto localDir = CosHemisphereRV().Sample(uv[0], uv[1]);
    auto frame = make_random_frame(prevVtx.Get(intersection_));
    auto worldDir = to_world(frame, localDir);
    return sample(worldDir);
constexpr auto SphericalRV() {
    constexpr auto phi = two * pi * u1;
    constexpr auto sinPhi = sin(phi);
    constexpr auto cosPhi = cos(phi);
    constexpr auto theta = pi * u2;
    constexpr auto sinTheta = sin(theta);
    constexpr auto cosTheta = cos(theta);
    return make_random_vector(
        make_random_var(sinPhi * sinTheta),
        make_random_var(cosTheta),
        make_random_var(- cosPhi * sinTheta)
    );
}

template <typename ContextType>
auto envmap_emitter::SampleDirect(const Vertex &prevVtx, ContextType& context, UniDist &uniDist) const {
    // Special construct for sampling from a piecewise distribution
    // representing the importance distribution of the environment map
    auto uv = context.PiecewiseLinear2D(uniDist, distribution2d);
    // Sample from a sphere and transform the direction to world coordinate
    const Float time = prevVtx.Get(intersection_).time;
    const Transform &trafo =
        emitter->getWorldTransform()->eval(time);
    auto frame = make_random_frame(trafo);
    auto dir = to_world(frame, (SphericalRV().Sample(uv[0], uv[1])));
    auto p = constant(prevVtx.Value());
    auto bo = named_constant(sph_center_, to_vector3(boundSphereOrigin));
    auto br = named_constant(sph_radius_, boundSphereRadius);
    // Symbolic intersect with scene bounding sphere
    auto pt = intersect_sphere(bo, br, p, dir);

    // Fill intersection information
    Intersection its;
    its.isTriangle = false;
    its.sph_center = boundSphereOrigin;
    its.sph_radius = boundSphereRadius;
    its.time = time;
    return sample(
        pt,
        sph_center_ = to_vector3(boundSphereOrigin),
        sph_radius_ = boundSphereRadius,
        is_triangle_ = Real(false),
        intersection_ = its
    );
}

template <typename ContextType>
auto envmap_emitter::SamplePosition(mclib::Context<ContextType>& context,
    mitsuba::UniDist &uniDist, const Float &time) const {
    // Sample a position on the scene bounding sphere
    auto uv = context.Uniform2D(uniDist);
    auto d = SphericalRV().Sample(uv);
    auto bo = named_constant(sph_center_, to_vector3(boundSphereOrigin));
    auto br = named_constant(sph_radius_, boundSphereRadius);
    auto pt = bo + d * br;

    // Fill intersection information
    Intersection its;
its.isTriangle = false;
its.sph_center = boundSphereOrigin;
its.sph_radius = boundSphereRadius;
its.time = time;
return sample(
    pt,
    sph_center = to_vector3(boundSphereOrigin)
    , sph_radius = boundSphereRadius
    , is_triangle = Real(false)
    , intersection = its
);

template<typename ContextType>
auto envmap_emitter::SampleDirection(const Vertex &prevVtx, Context<ContextType>&
    context, UniDist &uniDist) const {
    // Special construct for sampling from a piecewise distribution
    // representing the importance distribution of the environment map
    auto uv = context.PiecewiseLinear2D(uniDist, distribution2d);
    const Transform &trafo =
        emitter->getWorldTransform()->eval(prevVtx.Get(intersection).time);
    auto frame = make_random_frame(trafo);
    auto dir = to_world(frame, -(SphericalRV().Sample(uv[0], uv[1])));
    return sample(dir);
}

template<typename CosCutoffMinusOne>
auto ConeRV(CosCutoffMinusOne &cosCutOffMinusOne) {
    // Sampling from a cone
    auto u1r = make_random_var(u1);
    auto cosTheta = make_random_var(one) + u1r * cosCutOffMinusOne;
    auto sinTheta = sqrt(one - sq(cosTheta));
    auto phi = two * pi * u2;
    auto sinPhi = sin(phi);
    auto cosPhi = cos(phi);
    return make_random_vector(
        cosPhi * sinTheta,
        sinPhi * sinTheta,
        cosTheta
    );
}

template<typename ContextType>
auto sphere_emitter::SampleDirect(const Vertex &prevVtx,
    Context<ContextType>& context, UniDist &uniDist) const {
    // Cone importance sampling
    auto p = constant(prevVtx.Value());
    auto so = named_constant(sph_center_, to_vector3(origin));
    auto sr = named_constant(sph_radius_, radius);
    // Determine the angle of the cone
    const Vector3 toCenter = to_vector3(origin) - prevVtx.Value();
    const Real distance = toCenter.norm();
    const Real sinAlpha = radius / distance;
    const Real cosAlpha = sqrt(1.f - sinAlpha * sinAlpha);
    const Vector3 toCenterN = toCenter / distance;
    // Generate a coordinate system based on the direction toward sphere's center
    auto frame = make_random_frame(to_vector(toCenterN));
    auto cosCutoffMinusOne = constant(cosAlpha - 1.0);
    auto uv = context.Uniform2D(uniDist);
    // Sample from the cone and transform to world coordinate
    auto dir = to_world(frame, (ConeRV(cosCutoffMinusOne).Sample(uv[0], uv[1])));
// Intersect with the sphere emitter
auto pt = intersect_sphere(so, sr, p, dir);
// Fill intersection information
Intersection its;
its.isTriangle = false;
its.sph_center = origin;
its.sph_radius = radius;
its.geoFrame = Frame(to_normal((pt.Value() - to_vector3(origin)).normalized()));
its.shFrame = its.geoFrame;
its.t = (Float)length(pt.Value() - prevVtx.Value());
its.time = prevVtx.Get(intersection_.time);
return sample(
    pt,
    sph_center_ = to_vector3(origin)
    sph_radius_ = radius
    is_triangle_ = Real(false)
    intersection_ = its
};

template <typename ContextType>
auto sphere_emitter::SamplePosition(Context<ContextType>& context, UniDist & uniDist, const Float & time) const
{
    // Uniformly choose a position on the sphere
    auto uv = context.Uniform2D(uniDist);
    auto d = SphericalRV().Sample(uv);
    auto bo = named_constant(sph_center_, to_vector3(origin));
    auto br = named_constant(sph_radius_, radius);
    auto pt = bo + br * d;
    // Fill intersection information
    Intersection its;
    its.isTriangle = false;
    its.sph_center = origin;
    its.sph_radius = radius;
    its.geoFrame = Frame(to_normal(d.Value()));
    its.shFrame = its.geoFrame;
    // Important to show that this intersection is not invalid
    its.t = Float(0);
    its.time = time;
    return sample(
        pt,
        sph_center_ = to_vector3(origin)
        sph_radius_ = radius
        is_triangle_ = Real(false)
        intersection_ = its
    );
}

template <typename ContextType>
auto sphere_emitter::SampleDirection(const Vertex & prevVtx, Context<ContextType>& context, UniDist & uniDist) const {
    auto uv = context.Uniform2D(uniDist);
    auto localDir = CosHemisphereRV().Sample(uv[0], uv[1]);
    auto frame = make_random_frame(prevVtx.Get(intersection_));
    auto worldDir = to_world(frame, localDir);
    return sample(worldDir);
}
A.3 Sensor sampling

The sensor sampling procedures are similar to the emitter sampling procedures, except that there is no specialized direct importance sampling. Our current implementation only supports a perspective pinhole camera.

```cpp
struct sample_position_on_sensor_t {
    template <typename T>
    auto operator()(Context<T>& context, const RandomSequence<Vertex>& path,
                    UniDist& uniDist, const Sensor* sensor, const Float& time) const {
        auto sampler = sensor->makeSampler();
        auto pt = sampler.Sample(sensor_sampling_tag::Position{}, context, uniDist, time);
        return sample(
            pt,
            sensor_ = sensor
        );
    }
};

struct sample_direction_from_sensor_t {
    template <typename T>
    auto operator()(Context<T>& context, const RandomSequence<Vertex>& path,
                    UniDist& uniDist, const Raycast& raycaster, const int x, const int y) const {
        auto sensor = path.Back().Get(sensor_);
        auto sampler = sensor->makeSampler();
        auto p = constant(path.Back().Value());
        // Sample direction from the sensor, intersect with the scene
        auto dir = sampler.Sample(sensor_sampling_tag::Direction{}, path.Back(),
                                   context, uniDist, x, y);
        const Ray ray(to_point(p.Value()), to_vector(dir.Value()), path.Back().Get(intersection_.time));
        const Intersection its = context.ConstantCall(raycaster, ray);
        auto pt = intersect(its, p, dir);
        return sample(
            pt,
            intersection_ = its
        );
    }
};
```
A.4 Bidirectional path tracing with direct light source importance sampling

We show the bidirectional path tracer with direct light source sampling here. The only difference from the code shown in the body of the thesis is in the path slicing and concatenation part.

```cpp
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);
}

// 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
        RandomSequence<Vertex> path;
        auto camSlice = camPath.Slice(0, camSize);
        if (emtSize == 1) {
            // Employ specialized importance sampling
            camSlice.Append(sampEmtDirect, uniDist, emitters);
            camSlice.Sample();
        } else {
            auto emtSlice = emtPath.Slice(0, emtSize);
            path = camSlice.Concat(reverse(emtSlice));
        }
        paths.push_back(path);
    }
    // Combine bsdf path and direct path
    // Returns a list of paths with their MIS weights
    auto combinedList = combine<PowerHeuristic>(paths);
    // Deposit the contributions
    for (const auto &combined : combinedList) {
        const auto &path = combined.sequence;
        // Splat the contribution to corresponding screen position
        film->Record(project(path),
                     combined.weight * (integrand(path) / path.Pdf()));
    }
}
```
A.5 Bidirectional mutation

The full bidirectional mutation is below. It includes several speedup improvements not mentioned in the body of the thesis.

```cpp
// We implement Mitsuba's variant of bidirectional mutation, which samples a // desired length first, then samples the length to delete, finally it samples // the length of the eye subpath and light subpath to insert.

// Some notation convention: // path_length is the number of edges of the path, // including two stub vertices before the sensor vertex and after the emitter vertex
// For example, a LSDDE path has path_length = 6 // delete_begin and delete_end represents the index of the edge to be deleted, // the index begins from 0 at the camera edge.
// For example, for a LSDDE path, if we choose delete_begin = 1 and delete_end = 3, // we are left with a LSDE path after deletion (2 edges between DD and DE are deleted)

struct bidir_mutation_t {
    // Mutation wrapper containing vector buffers, caches for random variables, // scene, and parameters // The buffers and caches are for speeding up the process by reducing unnecessary memory allocation
    BidirectionalMutation *mutation;

    template <typename T>
    auto operator()(Context<T>& context, const RandomSequence<Vertex>& path) {
        // Set up common variables
        const Scene *scene = mutation->scene;
        UniDist &uni_dist = mutation->uni_dist;
        const Raycast &raycaster = mutation->raycaster;
        const int min_length = mutation->min_length;
        const int max_length = mutation->max_length;

        // Sample the discrete random variables that determine the portion of the path to delete // and the portion of the path to insert
        const int path_length = path.Size() + 1;
        const auto& desired_length_rv = get_desired_length_rv(path_length);
        const int new_path_length = context.Sample(desired_length_rv, context.
            Uniform1D(uni_dist));
        const int new_path_size = new_path_length - 1;
        if (new_path_length < min_length || new_path_length > max_length) {
            // For some reason we generate a path that is not in the acceptabled range , reject
            return RandomSequence<Vertex>();
        }

        // Speed up by early exit // This function always returns true when the context is SamplingContext // When the context is PdfContext, it takes the size of the target RandomSequence and return whether it's the same // If user passed a wrong 'new_path_size', the correctness of the ConditionalPdf is no longer gauranteed.
        if (!context.MatchTargetSize(new_path_size)) {
            return RandomSequence<Vertex>();
        }

    }

};
```

76
const auto& delete_length_rv = get_delete_length_rv(path_length, new_path_length);
const int delete_length = context.Sample(delete_length_rv, context.UniformID(unidist));
const int insert_length = new_path_length - path_length + delete_length;
const auto& delete_begin_rv = get_delete_begin_rv(path, delete_length, insert_length);
if (delete_begin_rv.Size() == 0) {
    // Can't find proper segment to delete, reject the mutation
    return RandomSequence<Vertex>{};
}
const int delete_begin = context.Sample(delete_begin_rv, context.UniformID(unidist));
const int delete_end = delete_begin + delete_length;

// insert_length represents the number of edges we want to add
// If insert_length == 1, we only need to do connection
const int insert_size = insert_length - 1;
int eye_subpath_insert_size = 0;
if (insert_length > 0) {
    const auto& eye_subpath_insert_size_rv = get_eye_subpath_insert_size_rv(insert_size);
    eye_subpath_insert_size = context.Sample(eye_subpath_insert_size_rv, context.UniformID(unidist));
}
const int light_subpath_insert_size = insert_size - eye_subpath_insert_size;

// Set up buffers to re-use the memory chunk
RandomSequence<Vertex> &eye_subpath = mutation->eye_subpath;
RandomSequence<Vertex> &light_subpath = mutation->light_subpath;
RandomSequence<Vertex> &reversed_light_subpath = mutation->reversed_light_subpath;
RandomSequence<Vertex> &proposal_path = mutation->proposal_path;

// Slice the path and stores the two segments in subpaths
slice(path, 0, delete_begin + 1, eye_subpath);
if (delete_end < path.Size()) {
    slice(path, delete_end, path.Size() - delete_end, light_subpath);
    reverse(light_subpath, reversed_light_subpath);
} else {
    // delete_end == path.Size();
    clear(light_subpath);
    clear(reversed_light_subpath);
}

const Float time = path[1].Get(intersection_).time;
// Append subpaths
for (int i = 0; i < eye_subpath_insert_size; ++i) {
    if (eye_subpath.Size() == 0) {
        eye_subpath.Append(samplePositionOnSensor, uni_dist, scene->getSensor(time);
    } else if (eye_subpath.Size() == 1) {
        AppendDirectionFromSensor(eye_subpath, uni_dist, raycaster);
    } else {
        AppendBSDF(eye_subpath, uni_dist, raycaster);
    }
}

if (light_subpath_insert_size == 1 && reversed_light_subpath.Size() == 0) {
    // Special case: do direct importance sampling
}
```cpp
eye_subpath.Append(directSampleEmitter, uni_dist, scene->getEmitters());
}

else {
    for (int i = 0; i < light_subpath_insert_size; ++i) {
        if (reversed_light_subpath.Size() == 0) {
            AppendPositionOnEmitter(reversed_light_subpath, uni_dist, scene->getEmitters(), time);
        } else if (reversed_light_subpath.Size() == 1) {
            AppendDirectionFromEmitter(reversed_light_subpath, uni_dist, raycaster);
        } else {
            AppendBSDF(reversed_light_subpath, uni_dist, raycaster);
        }
    }
}

reverse(reversed_light_subpath, light_subpath);
concat(eye_subpath, light_subpath, proposal_path);
return proposal_path;

void construct_desired_length(const int path_length,
    std::vector<int> &desired_length_buffer, std::vector<Real> &desired_length_weight) {
    Reuse the vector buffers to avoid unnecessary allocation
    desired_length_buffer.clear();
    desired_length_weight.clear();
    for (int length = mutation->min_length; length <= mutation->max_length; length++) {
        Geometric distribution centered at path_length
        const int offset = std::abs(length - path_length);
        desired_length_buffer.push_back(length);
        desired_length_weight.push_back(pow(2.0, -offset));
    }
}

const discrete_random_var_dynamic<int>& get_desired_length(const int path_length) {
    Search the r.v. in cache given path_length, if not found construct the array
    std::vector<int> &desired_length_buffer = mutation->desired_length_buffer;
    std::vector<Real> &desired_length_weight = mutation->desired_length_weight;
    auto &rv_cache = mutation->desired_length rv;
    auto it = rv_cache.find(path_length);
    if (it == rv_cache.end()) {
        construct_desired_length(path_length, desired_length_buffer,
            desired_length_weight);
        return rv_cache.insert({path_length,
            discrete_dynamic(desired_length_buffer, desired_length_weight)}).first->
            second;
    }
    return it->second;
}

void construct_delete_length(const int path_length, const int desired_length,
    std::vector<int> &delete_length_buffer, std::vector<Real> &
    delete_length_weight) {
    Reuse the vector buffers to avoid unnecessary allocation
    delete_length_buffer.clear();
    delete_length_weight.clear();
    int min_deletion = std::max((path_length == desired_length) ? 2 : 1,
```
for (int length = min_deletion; length <= path_length; length++) {
  // Geometric distribution centered at 2.
  const int offset = std::abs(length - 2);
  delete_length_buffer.push_back(length);
  delete_length_weight.push_back(pow(2.0, -offset));
}

const discrete_random_var_dynamic<int>& get_delete_length(rv(const int path_length, const int desired_length) {
  // Search the r.v. in cache given path_length, if not found construct the array
  std::vector<int> &delete_length_buffer = mutation->delete_length;
  std::vector<Real> &delete_length_weight = mutation->delete_length_weight;
  auto &rv_cache = mutation->delete_length_rv;
  auto rv_key = std::make_pair(path_length, desired_length);
  auto it = rv_cache.find(rv_key);
  if (it == rv_cache.end()) {
    construct_delete_length(path_length, desired_length, delete_length_buffer, delete_length_weight);
    return rv_cache.insert({rv_key,
      discrete_dynamic(delete_length_buffer, delete_length_weight)}).first->second;
  }
  return it->second;
}

discrete_random_var_dynamic<int> get_delete_begin(rv(const RandomSequence<Vertex> &path, const int delete_length, const int insert_length) {
  std::vector<int> &delete_begin_buffer = mutation->delete_begin;
  delete_begin_buffer.clear();
  const Real specular_roughness_threshold = mutation->specular_roughness_threshold;
  const int path_length = path.Size() + 1;
  int min_index = -1;
  int max_index = path_length - delete_length - 1;
  if (delete_length == 1 || insert_length == 1) {
    // Never only delete the edge between the "stub" vertex and the camera/light source
    min_index++;
    max_index--;
  }
  for (int index = min_index; index <= max_index; index++) {
    // Never delete an edge where the left vertex is specular
    const bool left_spec = index > 0 && index < path.Size() - 1 && is_specular(path[index], specular_roughness_threshold);
    const int right_index = index + delete_length;
    const bool right_spec = right_index > 0 && right_index < path.Size() - 1 &&
      is_specular(path[right_index], specular_roughness_threshold);
    if (!left_spec && !right_spec) {
      delete_begin_buffer.push_back(index);
    }
  }
  return discrete_dynamic(delete_begin_buffer);
}

discrete_random_var_dynamic<int> get_eye_subpath_insert_size(rv(const int
insert_size) {
    // Search the r.v. in cache given path_length, if not found construct the array
    auto &rv_cache = mutation->eye_subpath_insert_size_rv;
    auto it = rv_cache.find(insert_size);
    if (it == rv_cache.end()) {
        // discrete_dynamic(insert_size + 1) builds a discrete uniform r.v. with values from 0 to insert_size
        return rv_cache.insert({insert_size, discrete_dynamic(insert_size + 1)}).first->second;
    }
    return it->second;
};
A.6 Caustic perturbation

The main logic of the caustic perturbation:

```cpp
struct caustic_perturbation_t {
    // Mutation wrapper containing scene, raycaster, and parameters
    CausticPerturbation *mutation;

    template <typename T>
    auto operator()(Context<T>& context, const RandomSequence<Vertex>& path) const
    {
        UniDist &unidist = mutation->unidist;
        const Raycast &raycaster = mutation->raycaster;
        auto sampleBSDF = mutation->sampleBSDF;
        if (path.Size() <= 3) {
            return RandomSequence<Vertex>{};
        }
        // Find the second non-specular vertex
        int split_pos = 0;
        for (int i = 2; i < path.Size(); i++) {
            if (i == path.Size() - 1 ||
                is_diffuse(path[i], mutation->specular_roughness_threshold)) {
                split_pos = i;
                break;
            }
        }
        // Adjust the perturbation angle so that the changes on the image position
        // is similar to lens perturbation [Veach p.354]
        const Real lengthE = (path[0].Value() - path[1].Value()).norm();
        const Real lengthL(0);
        for (int i = 1; i < split_pos; i++) {
            lengthL += (path[i].Value() - path[i + 1].Value()).norm();
        }
        const Real factor = lengthE / lengthL;
        const Real theta_1 = mutation->theta_1 * factor;
        const Real theta_2 = mutation->theta_2 * factor;
        auto light_subpath = reverse(path.Slice(split_pos, path.Size() - split_pos));
        auto perturb_dir = perturb_direction(theta_1, theta_2, unidist, raycaster);
        const Vector3 wo = (path[split_pos - 1].Value() - path[split_pos].Value()).normalized();
        // Perturb the direction by sampling an exponential distribution
        light_subpath.Append(perturb_dir(context, context, path[split_pos].Value(), wo, path[split_pos].Get(intersection_.time));
        for (int i = split_pos + 1; light_subpath.Size() < path.Size() - 1; i++) {
            const bool reflect = is.reflect(path[i], path[i - 1], path[i + 1]);
            light_subpath.Append(sampleBSDF, unidist, raycaster, reflect ?
                BSDF::EReflection : BSDF::ETransmission);
        }
        return path.Slice(0, 1).Concat(reverse_(light_subpath));
    }
};
```
The `perturb_direction` functor used in the mutation:

```cpp
struct perturb_direction {
    const Real theta_1;
    const Real theta_2;
    UniDist &uni_dist;
    const Raycast &raycaster;

    template <typename T>
    auto operator()(Context<T>& context, const Vector3 &prevPos, const Vector3 &prevDir, const Float time) {
        // Sampling the perturb direction
        auto theta = constant(theta_2) * exp(-log(constant(theta_2 / theta_1)) * u1);
        auto phi = two * pi * u2;
        auto sin_theta = sin(theta);
        auto cos_theta = cos(theta);
        auto sin_phi = sin(phi);
        auto cos_phi = cos(phi);
        auto offset = make_random_vector(
            sin_theta * cos_phi,
            sin_theta * sin_phi,
            cos_theta
        );
        // The perturb direction is centered around previous direction
        auto frame = make_random_frame(to_vector(prevDir));
        auto uv = context.Uniform2D(uni_dist);
        auto dir = to_world(frame, offset).Sample(uv[0], uv[1]);

        auto p = constant(previousPosition);
        const Ray ray(to_point(p.Value()), to_vector(dir.Value()), time);
        const Intersection its = context.ConstantCall(raycaster, ray);
        // Symbolic intersection
        auto pt = intersect(its, p, dir);

        return sample(
            pt
            , intersection_ = its
        );
    }
};
```
A.7 Lens perturbation

The main logic of the lens perturbation:

```c++
struct lens_perturbation_t {
    // Mutation wrapper containing scene, raycaster, and parameters
    LensPerturbation *mutation;

    template <typename T>
    auto operator()(Context<T>& context,
        const RandomSequence<Vertex>& path) const {
        const Real min_jump = mutation->min_jump;
        const Real covered_area = mutation->covered_area;
        const Real specular_roughness_threshold =
            mutation->specular_roughness_threshold;
        auto sampleBSDF = mutation->sampleBSDF;

        auto eye_subpath = path.Slice(0, 1);
        // Perturb the screen position
        eye_subpath.Append(perturbLens, path,
            min_jump, covered_area, uniDist, raycaster);
        // Follow specular chain
        // (is_specular return false if path[i] is the light)
        for (int i = 1; i < specular(path[i], specular_roughness_threshold); i++) {
            const bool reflect =
                isReflect(path[i], path[i - 1], path[i + 1]);
            eye_subpath.Append(sampleBSDF, uniDist, raycaster,
                reflect ? EReflection : ETransmission);
        }
        auto suffix = path.Slice(eye_subpath.Size(), path.Size() - eye_subpath.Size());
        return eye_subpath.Concat(suffix);
    }
};
```
The `perturb_lens` functor used in the mutation:

```cpp
class perturb_lens {
    const Scene *scene;
    const Real min_jump;
    const Real covered_area;
    UniDist &uni_dist;
    const Raycast &raycaster;

template <typename T>
auto operator()(Context<T>& context, const RandomSequence<Vertex> &path) const
{
    const Sensor *sensor = scene->getSensor();
    const Vector2 res = sensor->getResolution();
    const Vector2 invRes = sensor->getInvResolution();
    // Sample the screen space offset
    auto r1 = constant(min_jump);
    auto r2 = constant(Real(std::sqrt(covered_area_value * Real(res[0]) * Real(res[1]) / M_PI)));
    auto r = r2 * exp(-log(r2 / r1) * u1);
    auto phi = two * pi * u2;
    auto offset =
        make_random_vector(r * cos(phi) * constant(Real(invRes.x)),
                          r * sin(phi) * constant(Real(invRes.y)));

    // Project the path onto screen position
    Point2 prevPos;
    bool insideScreen = project(path, prevPos);
    prevPos[0] = Real(invRes.x);
    prevPos[1] = Real(invRes.y);
    // Perturb screen position
    auto uv = context.Uniform2D(uni_dist);
    auto newPos = (constant(to_vector2(prevPos)) + offset).Sample(uv[0], uv[1]);
    insideScreen = insideScreen &&
        at<0>(newPos).Value() >= 0 && at<0>(newPos).Value() < 1 &&
        at<1>(newPos).Value() >= 0 && at<1>(newPos).Value() < 1);
    // Sample new direction from the sensor, intersect with scene
    auto sensorSampler = sensor->makeSampler();
    auto p = constant(path[0].Value());
    auto dir = sensorSampler.Sample(sensor_sampling_tag::Direction{},
                                    path[0], context, uni_dist, newPos);
    const Ray ray(to_point(p.Value()), to_vector(dir.Value()), path[1].Get(intersection_.time));
    const Intersection its = context.ConstantCall(raycaster, ray);
    auto pt = intersect(its, p, dir);

    return optional_sample(
        insideScreen
        , pt
        , intersection_ = its
    );
}
};
```
A.8 Multi-chain perturbation

The main logic of the multi-chain perturbation:

```cpp
struct multichain_perturbation_t {
    // Mutation wrapper containing scene, raycaster, and parameters
    MultiChainPerturbation *mutation;

    template <typename T>
    auto operator()(Context<T>& context, const RandomSequence<Vertex>& path) const
    {
        // Set up inputs
        const Scene *scene = mutation->scene;
        const Real min_jump = mutation->min_jump;
        const Real covered_area = mutation->covered_area;
        const Real theta_1 = mutation->theta_1;
        const Real theta_2 = mutation->theta_2;
        UniDist &uni_dist = mutation->unidist;
        const Raycast &raycaster = mutation->raycaster;
        auto sampleBSDF = mutation->sampleBSDF;

        auto eye_subpath = path.Slice(0, 1);
        // Perturb the screen position
        auto perturb = perturb_lens{scene, min_jump, covered_area, uni_dist, raycaster};
        eye_subpath.Append(perturb(context, path));
        int index = 1
        // Propagate through possibly multiple specular chain
        for (; index < path.Size();)
        {
            // Propagate through specular chain
            for (; is_specular(path[index], mutation->specular_roughness_threshold);
            {
                const bool reflect =
                    is_reflect(path[1], path[1 - 1], path[1 + 1]);
                eye_subpath.Append(sampleBSDF, uni_dist, raycaster,
                    reflect ? BSDF::EReflection : BSDF::ETransmission);
            }
            // If next vertex is light source or diffuse, we're done
            if (index == path.Size() - 1 ||
                !is_specular(path[index + 1],
                    mutation->specular_roughness_threshold))
            {
                break;
            }
            // Perturb the direction
            auto perturb_dir = perturb_direction{theta_1, theta_2, uni_dist, raycaster};
            const mclib::Vector3 wo =
                (path[index + 1].Value() - path[index].Value()).normalized();
            eye_subpath.Append(perturb_dir(context,
                path[index].Value(), wo, path[index].Get(intersection_.time));
            index++;
        }
        auto suffix = path.Slice(eye_subpath.Size(), path.Size() - eye_subpath.Size());
        return eye_subpath.Concat(suffix);
    }
};
```
A.9 Tridirectional path tracing

The sampling procedure for generating the portal segment:

```cpp
struct sample_portal_t {
    template <typename T>
    auto operator()(Context<T>& context,
        const RandomSequence<Vertex>& seq,
        UniDist& uniDist,
        const Raycast& raycaster,
        const std::vector<std::array<mclib::Vector3, 3>>& tris) const {
        // Discretely sample a triangle on the portal
        auto trisRV = discrete_dynamic(tris);
        auto tri = context.Sample(trisRV, context.Uniform1D(uniDist));

        auto v0 = constant(tri[0]);
        auto v1 = constant(tri[1]);
        auto v2 = constant(tri[2]);

        auto uv = context.Uniform2D(uniDist);
        // Sample point on portal triangle
        auto portalPt = uniform_triangle(v0, v1, v2).Sample(uv[0], uv[1]);

        auto pi = make_random_var(literal<245850922, 78256779>{})�
        auto r = sqrt(mclib::one - sq(u3));
        auto phi = two * pi * u4;
        auto x = r * cos(phi);
        auto y = r * sin(phi);
        auto z = u3;

        auto ab = context.Uniform2D(uniDist);
        // Sample uniform hemisphere for outgoing direction
        auto wi = make_random_vector(x, y, make_random_var(z)).Sample(ab[0], ab[1]);

        auto e1 = v0 - v2;
        auto e2 = v1 - v2;
        auto N = normalize(cross(e1, e2));
        auto basis = coordinate_basis(N);
        // Convert outgoing direction to world coordinates
        auto dir = basis * wi;

        // First intersection point
        auto pt1 = intersect(its1, portalPt, dir);

        // Second intersection point, in opposite direction
        auto pt2 = intersect(its2, portalPt, -dir);

        auto sample1 = optional_sample(
            its1.isValid()
        , pt1
        , intersection_ = its1
        );

        auto sample2 = optional_sample(
            its2.isValid()
        , pt2
        , intersection_ = its2
        );

        return sample_tuple(sample1, sample2);
    }
}
```
A.10 Gradient-domain path tracing

The main logic of the gradient-domain path tracer:

```cpp
const Point2i offsets[] = {
    Point2i(-1, 0),
    Point2i( 1, 0),
    Point2i( 0, -1),
    Point2i( 0, 1)
};
constexpr int numOffsets = sizeof(offsets) / sizeof(Point2i);
// These offset paths are never actually traced, they merely store the
// strategies for
offsetSubpaths[numOffsets];
// Set up the gradient shifts strategies
Strategy<gradient-perturbation-t> perturbGradient[numOffsets] = {
    Strategy<gradient-perturbation-t>(scene, uniDist, raycaster,
        specularRoughnessThreshold, offsets[0]),
    Strategy<gradient-perturbation-t>(scene, uniDist, raycaster,
        specularRoughnessThreshold, offsets[1]),
    Strategy<gradient-perturbation-t>(scene, uniDist, raycaster,
        specularRoughnessThreshold, offsets[2]),
    Strategy<gradient-perturbation-t>(scene, uniDist, raycaster,
        specularRoughnessThreshold, offsets[3])
};
// Sample camera subpath
sensorSubpath.Append(samplePositionOnSensor, uniDist, scene->getSensor(), time);
for (int i = 0; i < numOffsets; i++) {
    offsetSubpaths[i].Append(samplePositionOnSensor, uniDist, scene->getSensor(), time);
    offsetSubpaths[i].Append(sampleDirectionFromSensor, uniDist, raycaster, x + offsets[i].x, y + offsets[i].y);
}
// When we call .Sample(), the path is actually sampled and the rays are traced
// We don't call .Sample() for the offset subpaths because they are only used
// for MIS computation
sensorSubpath.Sample();
```

// Loop until reaching specified maximum depth
// e.g. if maxDepth == 2, we don't need sensorSubpath.Size() > 3
for(;sensorSubpath.Size() <= m_maxDepth;)
    sensorSubpath.Append(sampleBSDF, uniDist, raycaster);
```
const ref_vector<Emitter> &emitters = scene->getEmitters();
// A convenience data structure that cache all the occlusion queries
```
OcclusionCache occlusionCache(scene, time);
// Initialize the contributions
Spectrum primary(0.f);
Spectrum gradients[numOffsets];
for (int i = 0; i < numOffsets; i++) {
    gradients[i] = Spectrum(0.f);
}
for (int pathLength = 2; pathLength <= m_maxDepth; pathLength++) {
    // Sample emitter subpath
    auto emitterSamplingPath = sensorSubpath.Slice(0, pathLength);
    if (!emitterSamplingPath.Back().Get(intersection_.isValid()) { continue; }
    emitterSamplingPath.Append(directSampleEmitter, uniDist, emitters);
    emitterSamplingPath.Sample();
    auto bsdfSamplingPath = sensorSubpath.Slice(0, pathLength + 1);
    // We don't use the combine construct here, because combine construct
    // assumes all paths that passed in are actually sampled.
    // Evaluate emitter contribution
    TSpectrum<double, SPECTRUM_SAMPLES> emitterEval(0.f);
    const double pdfEmitter = emitterSamplingPath.Pdf(emitterSamplingPath);
    if (isValid(pdfEmitter) && !occlusionCache.Query(
        std::make_pair(emitterSamplingPath[emitterSamplingPath.Size() - 2].Value (),
        emitterSamplingPath[emitterSamplingPath.Size() - 1].Value ())) {
        emitterEval = evalBidir(scene, emitterSamplingPath);
        if (isValid(pdfEmitter)) {
            const double pdfBSDF = bsdfSamplingPath.Pdf(emitterSamplingPath);
            const double weight = misWeight(pdfEmitter, pdfBSDF);
            Spectrum emitterPrimary = weight * Spectrum(emitterEval / pdfEmitter);
            primary += emitterPrimary;
        }
    }
    const double pdfBSDF = bsdfSamplingPath.Pdf(bsdfSamplingPath);
    TSpectrum<double, SPECTRUM_SAMPLES> bsdfEval(0.f);
    if (isValid(pdfBSDF)) {
        bsdfEval = evalBidir(scene, bsdfSamplingPath);
        if (isValid(pdfBSDF)) {
            const double pdfBSDF = bsdfSamplingPath.Pdf(bsdfSamplingPath);
            const double weight = misWeight(pdfBSDF, pdfEmitter);
            Spectrum bsdfPrimary = weight * Spectrum(bsdfEval / pdfBSDF);
            primary += bsdfPrimary;
        }
    }
    if (isValid(pdfEmitter) && isValid(pdfBSDF)) { continue; }
}
for (int i = 0; i < numOffsets; i++) {
    auto emitterSamplingPathShift = emitterSamplingPath.Mutate(perturbGradient[i]);
    auto bsdfSamplingPathShift = bsdfSamplingPath.Mutate(perturbGradient[i]);
    // For MIS computation
    auto offsetEmitterSamplingPath = offsetSubpaths[i].Slice(0, pathLength);
    AppendDirectSampleEmitter(offsetEmitterSamplingPath, uniDist, emitters);
auto offsetBSDFSamplingPath = offsetSubpaths[i].Slice(0, pathLength + 1);

// Compute emitter sampling contribution
TSpectrum<double, SPECTRUMSAMPLES> shiftEmitterEval(o.);
Float emitterJacobian = jacobian(perturbGradient[i], emitterSamplingPath, emitterSamplingPathShift, buffer);
if (emitterSamplingPathShift.AllValid() && !occluded(
    emitterSamplingPathShift, occlusionCache)) {
    shiftEmitterEval = emitterJacobian * evalBidir(scene, emitterSamplingPathShift);
} else {
    // Not invertible, set jacobian to zero
    emitterJacobian = 0.0;
}

// Compute the contribution for emitter sampled gradient
// 4 possible strategies here: two from this path, two from offset path
if (isValid(pdfEmitter)) {
    const double pdfBSDF = bsdfSamplingPath.Pdf(emitterSamplingPath);
    const double pdfOffsetEmitter = offsetEmitterSamplingPath.Pdf(
        emitterSamplingPath) * emitterJacobian;
    const double pdfOffsetBSDF = offsetBSDFSamplingPath.Pdf(
        emitterSamplingPath) * emitterJacobian;
    const double weight = misWeight(0, {pdfEmitter, pdfBSDF, pdfOffsetEmitter, pdfOffsetBSDF});
    Spectrum emitterGradient = weight * Spectrum((shiftEmitterEval - emitterEval) / pdfEmitter);
    gradients[i] += emitterGradient;
}

// Compute bsdf sampling contribution
TSpectrum<double, SPECTRUMSAMPLES> shiftBSDFEval(O.);
Float bsdfJacobian = jacobian(perturbGradient[i], bsdfSamplingPath, bsdfSamplingPathShift, buffer);
if (bsdfSamplingPathShift.AllValid() && !occluded(bsdfSamplingPathShift, occlusionCache)) {
    shiftBSDFEval = bsdfJacobian * evalBidir(scene, bsdfSamplingPathShift);
} else {
    // Not invertible, set jacobian to zero
    bsdfJacobian = 0.0;
}

// Compute the contribution for BSDF sampled gradient
// 4 possible strategies here: two from this path, two from offset path
if (isValid(pdfBSDF)) {
    const double pdfEmitter = emitterSamplingPath.Pdf(bsdfSamplingPath);
    const double pdfOffsetEmitter = offsetEmitterSamplingPath.Pdf(
        bsdfSamplingPath) * bsdfJacobian;
    const double pdfOffsetBSDF = offsetBSDFSamplingPath.Pdf(bsdfSamplingPath) * bsdfJacobian;
    const double weight = misWeight(1, {pdfEmitter, pdfBSDF, pdfOffsetEmitter, pdfOffsetBSDF});
    Spectrum bsdfGradient = weight * Spectrum((shiftBSDFEval - bsdfEval) / pdfBSDF);
    gradients[i] += bsdfGradient;
}
Deposit the contribution

```
std::lock_guard<std::mutex> lock(mutex);

Point2 screenPosition;
bool inside = project(scene, sensorSubpath, screenPosition);
if (!inside) {
  return;
}

const Float invSampleCount = Float(1) / sampler->getSampleCount();
primary *= invSampleCount;
for (int i = 0; i < numOffsets; i++) {
  gradients[i] *= invSampleCount;
}
gradients[0] *= Float(-1);
gradients[2] *= Float(-1);
// Splatting
primalImage->put(screenPosition, &primary[0]);
gradientXImage->put(screenPosition - Vector2(1, 0), &gradients[0][0]);
gradientXImage->put(screenPosition, &gradients[1][0]);
gradientYImage->put(screenPosition - Vector2(0, 1), &gradients[2][0]);
gradientYImage->put(screenPosition, &gradients[3][0]);
```

if (!m_hideEmitters) {
  auto path = sensorSubpath.Slice(e, 2);
  if (path.AllValid()) {
    Spectrum contribution = estimateBidir(scene, make_view(path));
    visibleEmitterImage->put(screenPosition, &contribution[0]);
  }
}

The gradient_perturbation_t functor used in the gradient-domain path tracer:

```
struct OffsetSampler {
  Real Uniform1D() {
    return Real(0);
  }

  std::array<Real, 2> Uniform2D() {
    return {{previousScreenPosition[0] + (offset[0] * Real(invResolution.x)),
             previousScreenPosition[1] + (offset[1] * Real(invResolution.y))}};
  }

  const Point2 &previousScreenPosition;
  const Point2i &offset;
  const Vector2 &invResolution;
};
```

// Gradient perturbation resembles the multi-chain perturbation
// Except that the screen offset is fixed (given by OffsetSampler),
// and that we only perturb a very small distance for perturb_direction

```
struct gradient_perturbation_t {
  template <typename T>
  auto perturb(Context<T>& context, const RandomSequence<Vertex>& path) const {
    const Sensor *sensor = scene->getSensor();
    const Vector2 invRes = sensor->getInvResolution();

    // Shift coordinates
    auto uniformScreenSpace = make_random_vector(make_random_var(u1),
                                               make_random_var(u2));

    // Project the path onto screen position
```
Point2 prevPos;
bool insideScreen = project(path, prevPos);
prevPos.x *= Real(invRes.x);
prevPos.y *= Real(invRes.y);

_OffsetSampler offsetSampler{prevPos, offset, invRes};
auto uv = context.Uniform2D(offsetSampler);
auto newPos = uniformScreenSpace.Sample(uv);

insideScreen = insideScreen && (
    at<0>(newPos).Value() >= 0 && at<0>(newPos).Value() < 1 &&
    at<1>(newPos).Value() >= 0 && at<1>(newPos).Value() < 1);

// Sample new direction and intersect
auto sensorSampler = sensor->makeSampler();
auto p = constant(path[0].Value());
auto dir = sensorSampler.Sample(sensor_sampling_tag::Direction{},
    path[0], context, uniDist, newPos);
const Ray ray(to_point(p.Value()), to_vector(dir.Value()), path[1].Get(
    intersection_.time));
const Intersection its = context.constant_call(raycaster, ray);
// Symbolic intersection
auto pt = intersect(its, p, dir)

return optional_sample{
    insideScreen
    , pt
    , intersection_ = its
};

}template <typename T>
auto operator<(Context<T>& context, const RandomSequence<Vertex>& path) const
{
    slice(path, 0, 1, camPath);
    // Shift the screen position
    camPath.Append(perturb(context, path));
    int index = 1;
    for(;index < path.Size();)
    {
        for (; is_specular(path[index], specular_roughness_threshold); index++)
        {
            if (index >= path.Size() - 1) {
                break;
            }
            const bool reflect = is_reflect(path[i], path[i - 1], path[i + 1]);
            camPath.Append(sampleBSDF, uniDist, raycaster,
                reflect ? BSDF::EReflection : BSDF::ETransmission);
        }
        if (index == path.Size() - 1) {
            break;
        }
        if (!is_specular(path[index + 1], specular_roughness_threshold))
        {
            break;
        }
        // Perturb a very small direction
        auto perturbDir = perturb_direction(degToRad(0.001f), degToRad(0.001f),
            uniDist, raycaster);
        const Vector3 wo =
            (path[index + 1].Value() - path[index].Value()).normalized();

    }
camPath.Append(perturb_dir(context,
    path[index].Value(), wo, path[index].Get(intersection_.time));
    index++;
} slice(path, camPath.Size(), path.Size() - camPath.Size(), pathSuffix);
concat(camPath, pathSuffix, proposal_path);
return proposalPath;
}

const Scene *scene;
UniDist &uniDist;
const Raycast &raycaster;
const Real specular_roughness_threshold;
const Point2i offset;
Strategy<sample_bsdf_t> sampleBSDF;
// Buffers for reducing redundant memory allocation
mutable RandomSequence<Vertex> camPath;
mutable RandomSequence<Vertex> pathSuffix;
mutable RandomSequence<Vertex> proposalPath;
};
Appendix B

Solver Patterns

The specific patterns used by the solver include:

- Barycentric coordinates: equations of the form \(a \cdot x_1 + b \cdot x_2 + c \cdot x_3 = Y\), where \(a + b + c = 1\)

- Linear systems: equations of the form \(M \cdot x = Y\)

- Scaled vectors: equations where a unit vector has been scaled by a scalar, e.g., \(c \cdot x = Y\), where \(c\) may contain variables. This also includes the case where the vector \(x\) has one constant coordinate, which occurs with camera space projections

- Reflected/refracted vectors: equations of the form \(2 \cdot \text{dot}(N, W) \cdot N - W = X\) and \((c \cdot \text{dot}(N, W) + e) \cdot N - c \cdot W = X\), where \(N\) is the variable vector, \(W\) is a constant vector, \(c\) is a constant scalar, and \(e\) is a variable scalar. We use this when sampling from the Beckmann microfacet distribution.

- Equations containing both \(\sin\) and \(\cos\), e.g. when solving \(\sin(u) = X\) and \(\cos(u) = Y\) we combine the solutions \((u = \arcsin(X)\) and \(u = \arccos(Y))\) into the 2 argument \(u = \arctan2(X, Y)\) function, which ensures the solution is in the correct quadrant.

- Two endpoints of a segment passing through a sampled vertex; this is used in the tridirectional path tracer.
The generic rules used by the solver include:

- Multiply both sides by the denominator (to remove fractions)
- Move constants to the left hand side / right hand side (to collect constants)
- Move variables to the left hand side / right hand side (to collect variables)
- Square both sides to remove square roots or raise to exponent to remove other fractional powers
- Expand expressions with exponents
- Factor variables
Bibliography


