Université de Montréal

Neural Probabilistic Path Prediction: Skipping Paths for Acceleration

par
Bowen Peng

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en Informatique, Option Imagerie

Août, 2022

© Bowen Peng, 2022.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Neural Probabilistic Path Prediction: Skipping Paths for Acceleration

présenté par:

Bowen Peng

a été évalué par un jury composé des personnes suivantes:

Mikhail Bessmeltsev,	président-rapporteur
Pierre Poulin,	directeur de recherche
Glenn Berseth,	membre du jury

Mémoire accepté le: 6 octobre, 2022

RÉSUMÉ

La technique de tracé de chemins est la méthode Monte Carlo la plus populaire en infographie pour résoudre le problème de l'illumination globale. Une image produite par tracé de chemins est beaucoup plus photoréaliste que les méthodes standard tel que le rendu par rasterisation et même le lancer de rayons. Mais le tracé de chemins est coûteux et converge lentement, produisant une image bruitée lorsqu'elle n'est pas convergée. De nombreuses méthodes visant à accélérer le tracé de chemins ont été développées, mais chacune présente ses propres défauts et contraintes. Dans les dernières avancées en apprentissage profond, en particulier dans le domaine des modèles génératifs conditionnels, il a été démontré que ces modèles sont capables de bien apprendre, modéliser et tirer des échantillons à partir de distributions complexes. Comme le tracé de chemins dépend également d'un tel processus sur une distribution complexe, nous examinons les similarités entre ces deux problèmes et modélisons le processus de tracé de chemins comme un processus génératif. Ce processus peut ensuite être utilisé pour construire un estimateur efficace avec un réseau neuronal afin d'accélérer le temps de rendu sans trop d'hypothèses sur la scène. Nous montrons que notre estimateur neuronal (NPPP), utilisé avec le tracé de chemins, peut améliorer les temps de rendu d'une manière considérable sans beaucoup compromettre sur la qualité du rendu. Nous montrons également que l'estimateur est très flexible et permet à un utilisateur de contrôler et de prioriser la qualité ou le temps de rendu, sans autre modification ou entraînement du réseau neuronal.

Mots clés: tracé de chemins, illumination globale, apprentissage profond, modèles génératifs conditionnels.

ABSTRACT

Path tracing is one of the most popular Monte Carlo methods used in computer graphics to solve the problem of global illumination. A path traced image is much more photorealistic compared to standard rendering methods such as rasterization and even ray tracing. Unfortunately, path tracing is expensive to compute and slow to converge, resulting in noisy images when unconverged. Many methods aimed to accelerate path tracing have been developed, but each has its own downsides and limitiations. Recent advances in deep learning, especially with conditional generative models, have shown to be very capable at learning, modeling, and sampling from complex distributions. As path tracing is also dependent on sampling from complex distributions, we investigate the similarities between the two problems and model the path tracing process itself as a conditional generative process. It can then be used to build an efficient neural estimator that allows us to accelerate rendering time with as few assumptions as possible. We show that our neural estimator (NPPP) used along with path tracing can improve rendering time by a considerable amount without compromising much in rendering quality. The estimator is also shown to be very flexible and allows a user to control and prioritize quality or rendering time, without any further training or modifications to the neural network.

Keywords: path tracing, global illumination, deep learning, conditional generative models.

CONTENTS

RÉSUMÉ . . . . **iii**

ABSTRACT . . . . **iv**

CONTENTS . . . . **v**

LIST OF ABBREVIATIONS . . . . **viii**

NOTATION . . . . **ix**

ACKNOWLEDGMENTS . . . . **x**

CHAPTER 1: INTRODUCTION **1**

1.1 Overview of Rendering 4

1.2 Global Illumination 7

1.3 Light Transport 8

1.4 Monte Carlo Techniques 10

1.4.1 Path Tracing 12

1.4.2 Photon Mapping 17

CHAPTER 2: ACCELERATION OF PATH TRACING **20**

2.1 Overview 21

2.1.1 Computational Efficiency 22

2.1.2 Convergence, Bias, and Variance Reduction 23

2.2 Sampling Methods 24

2.2.1 Early Stopping 24

2.2.2 Russian Roulette 25

2.2.3 Adaptive Sampling 26

2.2.4 Importance Sampling 26

2.2.5 Next Event Estimation 27

2.2.6 Bidirectional Path Tracing 27

2.2.7 Metropolis Path Tracing 29

2.2.8 Resampling 30

2.3 Information Reuse Methods 30

2.3.1 Irradiance Caching 31

2.3.2 Light Probes 31

2.3.3 Temporal Reprojection 32

2.4 Screen Space Methods 33

2.4.1 Screen Space Ray Tracing 34

2.4.2 Denoising and Super-Resolution 34

CHAPTER 3: NEURAL PROBABILISTIC PATH PREDICTION (NPPP) 36

3.1 Motivation 36

3.2 Theory 37

3.2.1 Statistical Model of Path Tracing 37

3.2.2 Sampling 39

3.2.3 Light Fields 41

3.3 Algorithm 44

3.3.1 Neural Architecture 45

3.3.2 Acceleration Structures 47

3.3.3 Training 47

3.4 Results 49

3.5 Discussion 52

3.5.1 Grid Size 55

3.5.2 Quantization 55
3.6 Implementation Details 56

CHAPTER 4: CONCLUSION 61
4.1 Future Work 61

BIBLIOGRAPHY 63

LIST OF ABBREVIATIONS

BDF	Bidirectional Distribution Function
BRDF	Bidirectional Reflectance Distribution Function
BSDF	Bidirectional Scattering Distribution Function
BTDF	Bidirectional Transmittance Distribution Function
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
GAN	Generative Adversarial Network
GI	Global Illumination
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
KiB/MiB/GiB	Kibibyte/Mebibyte/Gibibyte (2^{10}/2^{20}/2^{30} bytes)
LTE	Light Transport Equation
MIS	Multiple Importance Sampling
MLP	Multilayer Perceptron
PM	Photon Mapping
PT	Path Tracing
VAE	Variational Autoencoder
VR	Virtual Reality

NOTATION

$\int_\Omega$	Definite integral over a solid angle Ω
$f \circ g$	Function composition: $f(g(x)))$
$(x_0 \to x_1 \to \cdots \to x_n)$	Ordered list of vertices $x_0, x_1, \ldots, x_n$
$N(0, 1)$	Normal distribution (Gaussian distribution with $\mu = 0$ and $\sigma = 1$)
$P(X\|Y)$	Conditional probability distribution of X given Y
$x \sim P(X)$	Sample x is distributed according to the distribution $P(X)$
$\theta_{p\to t}$	Learned parameters of an estimator for the distribution $P(X_t\|X_p)$

ACKNOWLEDGMENTS

Many thanks to Prof. Pierre Poulin for the ideas, discussions, and encouragements that led to the investigation and development of NPPP. Also thanks to Arnaud Schoentgen for helping me troubleshoot and setting up the computing environments. Finally, all the thanks to my family, friends, professors, colleagues, and the open source community. Without the help from everyone, I would not be where I am today.

CHAPTER 1

INTRODUCTION

In computer graphics, rendering is the process of converting the representation of a combination of many models (geometry, textures, light sources, materials, shaders, etc.) that form a scene in a computer, into something visible to the human eye, often as a 2D raster image displayed on a screen.

To generate images with ever increasing photorealism, an accurate simulation of the various effects of light is necessary. To reproduce such effects, many global illumination (GI) algorithms have been developed. Some only simulate a subset of all possible effects of light, but most are able to accurately simulate the three most important ones: direct illumination and indirect illumination with and without caustics.

For example, "light tracing" actually simulates the path of photon particles by tracing photons [1] emitted from light sources until they are deposited on a camera's sensor, accounting for the effects of light being reflected/refracted/absorbed by surfaces or scattered/absorbed in volumes (often called "participating media").

One flaw of light tracing (or more specifically, tracing from the light sources) is that it is an extremely inefficient algorithm. In a general situation, infinitely few of all traced photons will actually hit a camera sensor before being absorbed and detected, as a camera's aperture and the path from it to the sensor is very small compared to the scene [2]. Most of the work devoted to paths unconnected to the camera is thus wasted and

1. We are not actually tracing each individual photon, but a group of photons that follow the same path. In the physical world, the intensity of light, or radiance, is proportional to the amount of photons being emitted per unit of time. Each individual photon does not have a notion of intensity. The energy of a photon only depends on its wavelength, which affects the perceived color, but not true intensity. This leads to a long debate about the representation of light transport between physics and computer graphics communities that we will not settle here.

2. Real cameras have apertures that are often measured in millimeters, while a scene can be meters in size, captured on a sensor that is usually a few centimeters wide.

does not contribute to the rendered image.

An obvious modification is to compute the path followed by this process but in reverse order. If we trace a photon's path backward, starting from the camera until it hits a light source, we can compute the equivalent contribution of the light source in the same way as before, with the benefit that all paths are "hitting" a camera's sensor. The caveat here is that a path must reach a light source instead, and intense tiny light sources will be as difficult to hit compared to a camera's sensor through its aperture. Fortunately, most scenes in computer graphics are composed of much larger light sources.

This small modification in the ordering gives us the simplest, most powerful, general, and commonly used Monte Carlo (MC) algorithm for computing global illumination: path tracing (PT).

With path tracing, instead of expecting the photons to hit by chance a very small sensor through an aperture, we want the "inverted" photons to hit light sources with a higher probability. This is less of a problem for interior scenes, as unoccluded light sources are often larger than a camera's aperture. For example, a standard A19 light bulb's diameter is 6cm if we consider that it uses a diffusing glass (whereas if the bulb uses a clear glass, we would need to intersect with the emitting filament size instead). In outdoor scenes, while the Sun is very small as seen from Earth, the whole sky can be considered a light source due to atmospheric scattering, especially on cloudy days.

Path tracing is still computationally expensive, as it is based on a Monte Carlo algorithm that converges slowly. We need to compute millions or even billions of paths to get an image with an acceptable level of noise. Each ray that we trace for a straight segment within a complete path needs an intersection check with the scene's geometry, a texture/material lookup, and a random sampling process to compute the next ray for the same path.

Because of this high computational cost, many methods have been developed to im-

prove performance and convergence. Some methods improve the data structure of the geometry, such that each intersection is evaluated in less time. Some methods try to intelligently compute the next ray, such that it has a higher probability to eventually hit a light source. Other methods try to reduce the final image's variance by introducing some bias.

More recently, learning-based methods have found success in this domain. As the global illumination problem is highly nonlinear and slow to resolve, machine learning (ML) algorithms and deep learning models can be used to learn and predict some processes within an algorithm that is difficult to compute and/or that converges slowly.

One simple example of a machine learning method commonly used to accelerate convergence of path tracing is denoising [2, 6, 36]. We learn a function using a neural network that takes as input an unconverged noisy image from path tracing with a small number of samples [3] and that tries to output a converged image with a very large number of samples. As this algorithm works in screen space, its complexity depends only on the size of the image, while a path tracing algorithm might be slowed down by many other factors related to the 3D scene.

This thesis starts by giving an overview of common acceleration techniques for path tracing. Then we propose a new family of accelerators for path-based Monte Carlo global illumination algorithms. We specifically design a new technique for path tracing called neural probabilistic path prediction (NPPP). It is based on the standard theory of distribution estimation that aims to predict future rays within a path and allows the path tracing algorithm to skip the evaluation of some rays. This makes the construction of a full path from the camera to a light source much faster, as we are able to sample a few times the learned distribution within neural networks instead of intersecting the

3. The number of samples in path tracing usually refers to the number of computed paths per pixel in the rendered image. Path tracing's time complexity for a static scene is proportional to the number of pixels multiplied by the number of samples.

geometry and computing the next ray potentially many times in a scene.

1.1 Overview of Rendering

In order for a computer to synthesize an image given some data, we need a way of representing 2D or 3D scenes, a method to coherently display that scene given a viewport/camera, and in cases where interactivity is important such as in video games, the method has to be fast enough so that the user does not notice any latency. The speed of rendering is often measured in frames per second (FPS), with at least 60 FPS being a good target for video games.

In the following overview, we will mostly focus on rendering 3D objects and scenes, but will not go into details about the different ways to represent such scenes, as the field of geometry in computer graphics is very vast. We will start by giving a quick overview on rendering, and go more in depth when talking about global illumination techniques.

A modern graphics pipeline usually consists of three parts: the application step, the geometry step, and a rasterization step. The first step, which is the application step, usually consists of everything running on the CPU, for example, loading assets into memory, physics simulations, animations, culling of unnecessary geometry, etc. Anything that does not run on the GPU can be considered to be in the application step. After parsing and transforming the scene that we want to display, we usually obtain a large list of triangles represented as three vertices in 3D space, called primitives [4]. Each vertex of the triangles can also have associated data represented in the form of a multidimensional vector, in which it can encode anything from the normal at that location to uv coordinates for texture lookup. A collection of primitives that form an object is called a mesh.

All of the data previously generated can now be copied and loaded in the memory

4. A primitive is usually a triangle, but can sometimes be a quadrilateral, a line, or a point. Other more sophisticated primitives can include bicubic patches, implicit surfaces, signed distance fields, etc. They are often pre-converted into simpler triangles for better performance during rendering.

of a graphics processing unit (GPU). Unlike CPUs, GPUs have a very large number of highly specialized parallel processing units that can work on large amounts of data in parallel.

First, one type of graphics processing programs called shaders can be run on each of the primitives in parallel. Vertex shaders can apply distortions to the vertices depending on external data and can be used for effects such as wind or procedural animations. Geometry shaders can remove or add primitives, and can provide ways to simplify or add details to an object or scene without sending as much data to the GPU.

During the geometry step, the list of vertices in world space is transformed to the camera's local space. Then a projection transformation is applied to all vertices depending on the type of camera. The two main types of projection are central projection for perspective cameras and parallel projection for orthographic cameras.

Finally, during rasterization, each triangle is drawn on the screen according to its depth, and in some cases, a fragment shader can be used to apply effects on the pixels drawn to the screen. For example, fragment shaders can be used to compute lighting, change an object's color, or procedurally generate textures on surfaces.

This rather complex method described above is commonly referred to as the raster-

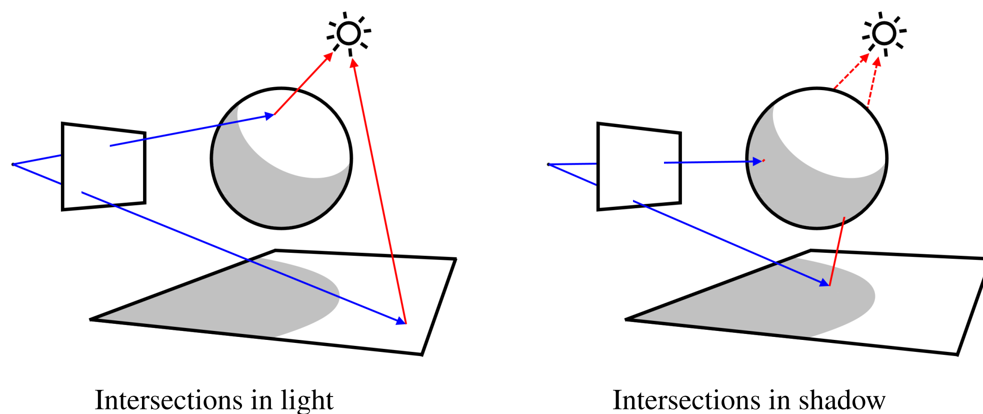


Figure 1.1 – Visualization of the ray tracing algorithm, where a secondary ray (in red) is traced to verify if a point on the surface is being illuminated by the light source or not.

ization pipeline. It is very fast and most GPUs from the last decade are designed and optimized for it. Ray tracing is an alternative to rasterization that is much simpler to understand and to program, but more computationally expensive. Ray tracing has only recently started to become a viable real-time rendering algorithm due to the increasing power of GPUs. The ray tracing algorithm can be described as follows. For each pixel in the viewport, we launch a ray and check for the closest intersection to the geometry, then we color and shade that pixel depending on surface color, textures, lighting, shadows, shading, reflections, etc. A simple visualization of ray tracing is provided in Figure 1.1.

Algorithm 1 Basic ray tracing algorithm

```
for every pixel do
    cast a ray from the camera's position through the pixel's position
    find the closest intersection point of the ray with the scene
    while a reflective or refractive surface is intersected do
        ray trace recursively using reflection or refraction laws
        if a closed loop is detected then
            stop recursion
        end if
    end while
    for all light sources do
        cast a ray from the intersection point towards the light source
        if the ray is not blocked then
            accumulate the contribution of the light source for shading
        end if
    end for
    compute the final color at the intersection point
end for
```

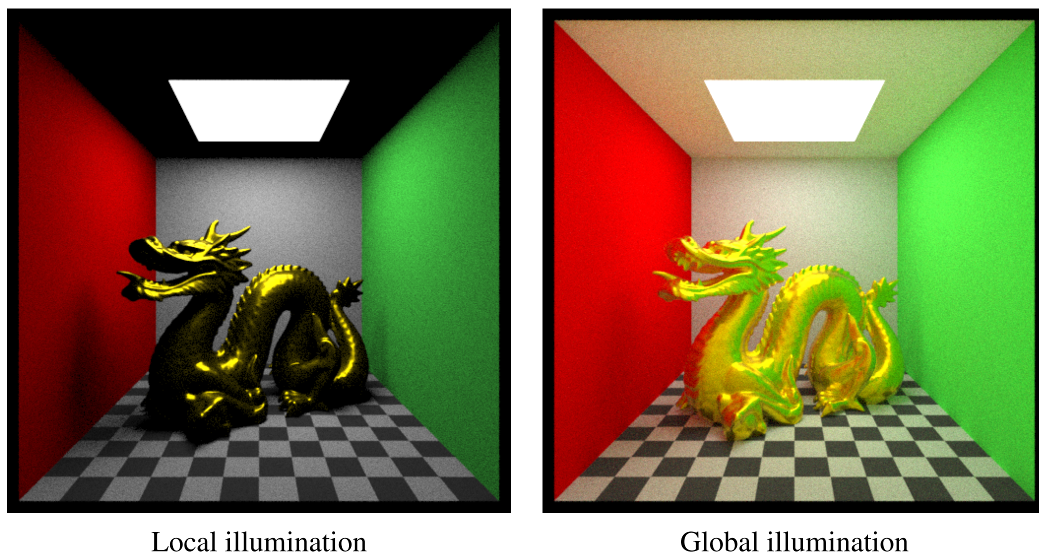
Ray tracing allows for a much more flexible representation of geometry, where in some cases traditionally rasterized primitives such as triangles are not necessarily useful or desirable. For example, a voxel grid can be efficiently ray traced and allows for a much more direct representation, without any need to convert the grid to a list of simpler primitives. The ray tracing problem can also be generalized to not just finding the first intersection, as it is possible to launch a followup ray at the first intersection to check if a

light source is visible from that point. This allows us to directly compute exact shadows, without much change to the algorithm. Mirror and refractive surfaces can also be treated by recursively calling the ray tracing algorithm.

1.2 Global Illumination

In order to generate photorealistic images from a scene, an accurate simulation of the light effects present in the real world is necessary. As light itself is what enables us to see, simulating the interaction of light with the scene can be a good starting point. How light affects illumination in a scene can be decomposed in three main categories: direct illumination, and indirect illumination with and without caustics.

Direct illumination affects all objects that are directly illuminated by a light source. This can easily be computed using the ray tracing algorithm described above as we simply check if the light source is visible from the target location and compute its contribu-



Local illumination Global illumination

Figure 1.2 – Effects of local vs global illumination. Note how the ceiling cannot be directly illuminated by the square Lambertian light source. In the rendering with global illumination, color bleed is also noticeable on the white walls, where it has a red, yellow, and green tinge.

tion to that location using information from the surface's material. Note that all surface points that do not have a free line of sight to a light source will be completely black.

Indirect illumination describes all the light bouncing around a scene from all kinds of materials. For example, when sunlight enters a window, it does not only illuminate the parts directly visible to the sun. Light bounces around on the walls and illuminates the whole room. Common algorithms for indirect illumination include radiosity [7], photon mapping [13], and path tracing.

Caustics describe the complex patterns that occur when light reflects or refracts on curved surfaces. For example, caustics can be seen very prominently at the bottom of a pool when the water surface is disturbed. Caustics may be considered a different category from general indirect illumination due to the fact that they are much harder to accurately compute, as they involve complex processes that concentrate large amounts of light on potentially small areas. The basic path tracing algorithm will have difficulties in simulating caustics as it is tracing rays in reverse. When large amounts of light are focused on a very small area, we need to generate a very large number of backward paths on that small area in order to have an accurate estimate. Many of the backward paths will also not contribute to the caustics as they might not hit a light source. Light tracing, photon mapping, bidirectional path tracing, and Metropolis path tracing are usually more efficient at simulating caustics.

1.3 Light Transport

To get a general picture on how to solve global illumination, we can try to figure out how to transport light in a scene. We can describe the light transport process mathematically in an equation called the light transport equation (LTE), also often called the rendering equation. All previous algorithms listed in the global illumination section aim to partially or fully solve the LTE. A simplified version from the one described by

Kajiya [14] is as follows:

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f_d(p,\omega_o,\omega_i)L_i(p,\omega_i)(n\cdot\omega_i)d\omega_i \ . \tag{1.1}$$

As this equation might seem fairly complex, we can look at it one part at a time. $L_o(p,\omega_o)$ describes the radiance [5] leaving the surface at position p and direction ω_o; it is effectively how bright a surface is when looked at from direction $-\omega_o$. $L_e(p,\omega_o)$ describes the emitted radiance from the surface itself given p and ω_o, for example from black body radiation due to heat. Finally, the large integral represents all the light scattered and reflected from the surface with direction ω_o from all possible incoming light paths over the hemisphere (or sphere) Ω.

Inside the integral, $f_d(p,\omega_o,\omega_i)$ is a bidirectional distribution function (BDF) representing the properties of the surface at point p given outgoing direction ω_o and incoming direction ω_i. $L_i(p,\omega_i)$ is the incoming radiance from direction ω_i. $(n\cdot\omega_i)$ represents the cosine of the angle between the normal vector n and ω_i, assuming all direction and normal vectors are normalized. We are effectively integrating over the contribution of all incoming rays for a specific outgoing direction ω_o.

Common BDFs for surfaces include bidirectional reflectance distribution functions (BRDFs), bidirectional transmittance distribution functions (BTDFs), and their generalization, bidirectional scattering distribution functions (BSDFs). A visualization of three common BRDFs can be found in Figure 1.3, where the length of an outgoing arrow is proportional to the value returned by the function f_d.

The integral part of the LTE is intractable if we try to solve it by brute force, as light rays might scatter around infinitely and might form closed loops where L_i could depend on L_o. Furthermore, if the function f_d is not a dirac delta, we will have an infinite number

5. Radiance is defined as a unit of power per unit of solid angle per unit of projected area. In SI units, it is watt per steradian per square meter, or $\frac{\mathrm{W}}{\mathrm{sr}\cdot\mathrm{m}^2}$. In the less-formal literature, radiance is also sometimes referred to as "intensity".

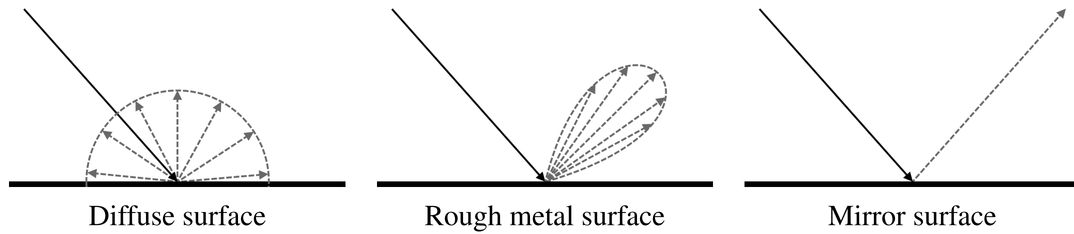


Figure 1.3 – Visualization of common BSDFs.

of rays to launch in order to correctly integrate over all ω_i.

A popular family of algorithms used to solve these types of intractable problems is based on Monte Carlo.

1.4 Monte Carlo Techniques

Instead of computing the LTE by brute force, we can simply decompose the problem as a sampling problem, and solve it using the Monte Carlo method.

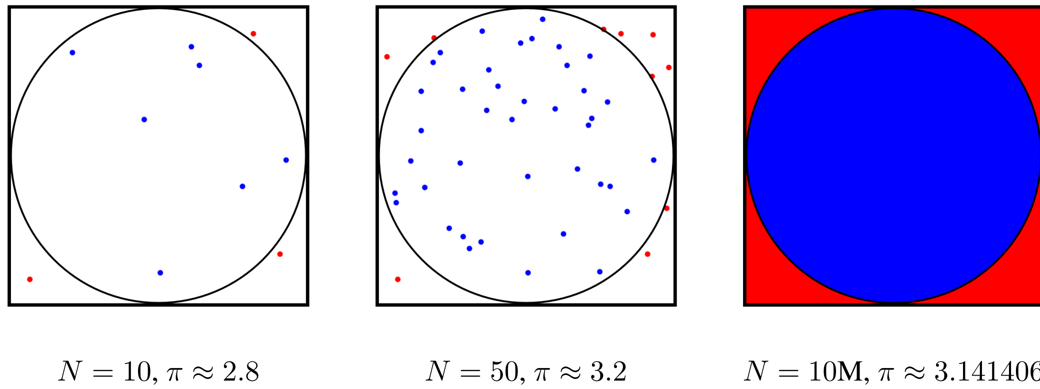
Given a multidimensional definite integral I with volume V of the form:

$$I = \int_{\Omega} f(\mathbf{x}) d\mathbf{x}$$
$$V = \int_{\Omega} d\mathbf{x} \ .$$

If V is known, the integral I can be approximated by uniformly sampling $\mathbf{x}_i$, where $\mathbf{x}_i \in \Omega$:

$$I = \lim_{N \to \infty} \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{x}_i) \ . \tag{1.2}$$

One simple example is to calculate π using a naive Monte Carlo approach. We know that the area of a circle is πr^2, thus for a unit circle with $r = 1$, its area is π. We can simply integrate the area of a unit circle bounded by a square to get π.



$N = 10$, $\pi \approx 2.8$ $N = 50$, $\pi \approx 3.2$ $N = 10\text{M}$, $\pi \approx 3.141406$

Figure 1.4 – Monte Carlo integration of π.

The function to integrate would be:

$$g(\mathbf{x}) = \begin{cases} 1 & \text{if } ||\mathbf{x}|| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

with $\Omega = [-1, 1] \times [-1, 1]$ and $V = 4$ for the area of the square.

$$\pi = \lim_{N\to\infty} \frac{4}{N} \sum_{i=1}^{N} g(\mathbf{x}_i) \ .$$

Similarly, the LTE over a unit hemisphere can be expressed as such:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \lim_{N\to\infty} \frac{2\pi}{N} \sum_{k=1}^{N} f_d(p, \omega_o, \omega_i^k) L_i(p, \omega_i^k)(n \cdot \omega_i^k) \ .$$

An alternative to using a volume V in Monte Carlo estimation is to use the probability distribution $p(\mathbf{x}_i)$.

$$I = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} \ . \tag{1.3}$$

When sampling uniformly, this is equivalent to using the volume, as the probability

distribution of a uniform distribution is simply the reciprocal of its volume. However, this allows us to sample using other distributions that are not uniform.

1.4.1 Path Tracing

As discussed earlier, path tracing consists of launching rays from the camera and tracing paths of virtual photon particle groups randomly until they hit a light source. More formally, it consists of solving the LTE by sampling one path at a time and integrating over many paths using the Monte Carlo algorithm. This is done so that branching is no longer needed.

Given the LTE described earlier in Equation 1.1, we can define a ray tracing function that returns a new position after tracing a ray with position p and direction ω against the scene. The outgoing direction of this new position is simply the negated incoming direction of the previous position.

$$\begin{aligned} p^{j+1} &= r(p^j, \omega_o^{j+1}) \\ \omega_o^{j+1} &= -\omega_i^j \ . \end{aligned}$$

Now we can substitute L_i with L_o by incrementing the position index

$$L_o(p^j, \omega_o^j) = L_e(p^j, \omega_o^j) + \int_\Omega f_d(p^j, \omega_o^j, \omega_i^j) L_o(p^{j+1}, \omega_o^{j+1})(n^j \cdot \omega_i^j) d\omega_i^j \ .$$

We can then simplify the equation as follows:

$$\begin{aligned} L_j &= L_o(p^j, \omega_o^j) \\ E_j &= L_e(p^j, \omega_o^j) \\ K_j \circ L_{j+1} &= \int_\Omega f_d(p^j, \omega_o^j, \omega_i^j) L_o(p^{j+1}, \omega_o^{j+1})(n^j \cdot \omega_i^j) d\omega_i^j \ . \end{aligned} \tag{1.4}$$

We then obtain this infinitely recursive function, starting at $j = 0$:

$$\begin{aligned} L_j &= E_j + K_j \circ L_{j+1} \\ L &= L_0 \ . \end{aligned} \tag{1.5}$$

Now to further understand what is happening in this function, we can try to isolate the contribution of each path length to the total radiance by using a select function that zeroes out the emissivity of surfaces from other path lengths:

$$s(\mathbf{x}, j, m) = \begin{cases} \mathbf{x} & \text{if } j = m \\ 0 & \text{otherwise} \end{cases} \tag{1.6}$$

$$\begin{aligned} L_j^m &= s(E_j, j, m) + K_j \circ L_{j+1}^m \\ L^m &= L_0^m \ . \end{aligned} \tag{1.7}$$

Now for every path length M, we can derive a corresponding equation that does not include the contribution of shorter and longer path lengths. For example, for increasing path lengths starting from $M = 0$, denoted by the superscript m in L^m, we have:

$$\begin{aligned} L^0 &= E_0 \\ L^1 &= K_0 \circ E_1 \\ L^2 &= K_0 \circ K_1 \circ E_2 \\ L^M &= K_0 \circ K_1 \circ \cdots \circ K_{(M-1)} \circ E_M \ . \end{aligned} \tag{1.8}$$

If we expand one L^m term we obtain:

$$
\begin{aligned}
T^m &= \prod_{j=0}^{m-1} f_d(p^j, \omega_o^j, \omega_i^j)(n^j \cdot \omega_i^j) d\omega_i^j \\
L^m &= \underbrace{\int_\Omega \cdots \int_\Omega}_{m} E_m T^m \ .
\end{aligned}
\tag{1.9}
$$

The T^m term is commonly called the throughput. Thus the most important part of path tracing is to be able to compute the throughput for any path length, and multiply it by the emissivity of the surface at the end of the path in order to obtain the contribution for a single path.

Finally, we simply sum all L^m to compute all the paths:

$$
L = \sum_{m=0}^{\infty} L^m \ . \tag{1.10}
$$

With this, we can see how a path tracing algorithm is rather simple to understand and

Algorithm 2 Naive path tracing algorithm

for every pixel **do**
  cast a ray from the camera's position through the pixel's position
  $p^0 \leftarrow$ find the closest intersection point of the ray with the scene
  $L \leftarrow 0$
  for number of samples $n = 0$ to N **do**
    for path of length $m = 0$ to M **do**
      randomly and uniformly sample a path of length m starting from p^0
      compute the throughput T^m of that path
      find the emissivity E_m of the surface at the end of the path
      $L \leftarrow L + E_m \cdot T^m$
    end for
  end for
  set the pixel color as L
end for

implement. The pseudocode for a naive path tracing algorithm based on Equation 1.10 is shown in Algorithm 2.

We will not go into details about every step of the path tracing algorithm here, as this is just to illustrate the general idea and should facilitate the understanding of the next chapters if the reader is unfamiliar with path tracing in general.

One observation we can make is that we are wasting the computation from the previous path of length m when computing a new path of length $m + 1$. To accelerate the algorithm we can simply re-use the throughput of the previous path and sample from the end of the previous path (see Equation 1.9), where the total throughput is simply a product of the previous path's throughput $p^0 \ldots p^{j-1}$ with the throughput of the current vertex at position p^j. The pseudocode of this improved algorithm is described in Algorithm 3.

Algorithm 3 Basic path tracing algorithm

```
for every pixel do
    cast a ray from the camera's position through the pixel's position
    p^0 ← closest intersection point of the ray with the scene
    L ← 0
    for number of samples n = 0 to N do
        T ← 1.0
        p ← p^0
        for path of length m = 0 to M do
            starting from p, randomly and uniformly sample a ray
            q ← closest intersection point of the ray
            T_q ← the throughput of q → p
            find the emissivity E_m of the surface at q
            p ← q
            T ← T · T_q
            L ← L + E_m · T
        end for
    end for
    set the pixel color as L
end for
```

An interesting property of this algorithm is that N and M control the bias and variance of the estimator. More specifically, when N is increased, variance decreases, but

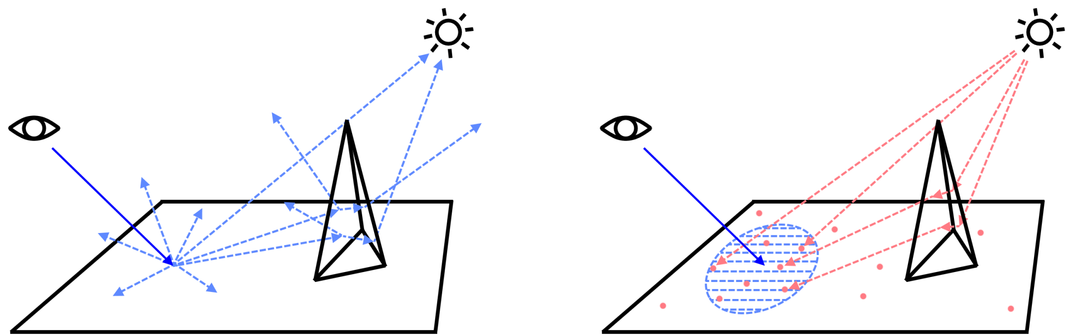


Figure 1.5 – Visualization of the path tracing (left) and photon mapping (right) algorithms.

the algorithm runs for a longer time. When M is increased, bias is reduced, variance is increased, and the algorithm runs for a longer time. We will discuss bias and variance more in depth in Chapter 2, but for now, a simple explanation of bias and variance in path tracing is that higher variance means that the output is more noisy, and higher bias means that averaging an infinity of noisy outputs does not converge to the correct value. We can see that for this simple algorithm, it is always biased when $M \neq \infty$, as we skip the evaluation of some paths of light. We will discuss in later chapters ways to make this algorithm unbiased without having to always sample up to infinity.

One of the biggest problems with path tracing is that it handles caustics [6] very poorly. Caustics that manifest on surfaces as strong and visually obvious shapes are formed when a large amount of light from a very small [7] light source is concentrated into a much small surface area. This is a major problem for the path tracing algorithm as it cannot reliably find very small light sources viewed from the surface where caustics are

6. Caustics are an optical phenomenon where an envelope of light rays is refracted or reflected, and then focused by a curved object in a way that forms concentrated dots, lines and patterns of light on a surface. They are commonly seen at the bottom of pools or when sunlight passes through a wine glass.

7. It is not necessary for the light source itself to be small. This requirement is also met for light sources hidden behind objects with a small gap or behind a small hole, where the receiving surface is very small, or a directional light source infinitely far away.

cast. Most of the caustics will be missed as the path tracing algorithm has a very low probability of finding the light. However, some paths, by chance, could reach a light. These connecting paths will add a very large contribution to the radiance, which makes the resulting few pixels very bright. This adds a lot of noise to the final image and slows down convergence considerably (see Figure 1.6). These artifacts are commonly called "fireflies" in the rendering community.

1.4.2 Photon Mapping

The photon mapping algorithm introduced by Jensen [13] aims to solve this issue by applying the LTE directly to "photons", or photon particles. We thus deposit photons on diffuse surfaces using surface absorption rules (e.g., BRDFs). These photons are used to encode the surface radiance at their locations. This creates a photon map that we can use to compute the surface radiance when we launch camera rays. Instead of recursively trying to reach the light sources like in path tracing, we use the photons in the photon map as a density estimation to compute surface radiance.

If we were able to launch an infinity of photons, all positions of surfaces in the scene that should be lit would have photons that describe the radiance at that position. We can simply sum those photons to get an exact radiance of that surface. Note that the brightness/contribution of each photon is scaled by the number of photons that we launched. This is to normalize the brightness of a scene so that it does not vary in function of the number of photons. This is expressed by the following equation:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} f_d(p, \omega_o, \theta_k) \delta(p, p_k) \phi_k \ ,$$

where $\delta(p, p_k)$ is the Kronecker delta function, which returns 1 when $p = p_k$ and 0 otherwise. θ is the photon's incident direction and ϕ the photon contribution.

However, if the number of photons is finite, the Kronecker delta will always be equal

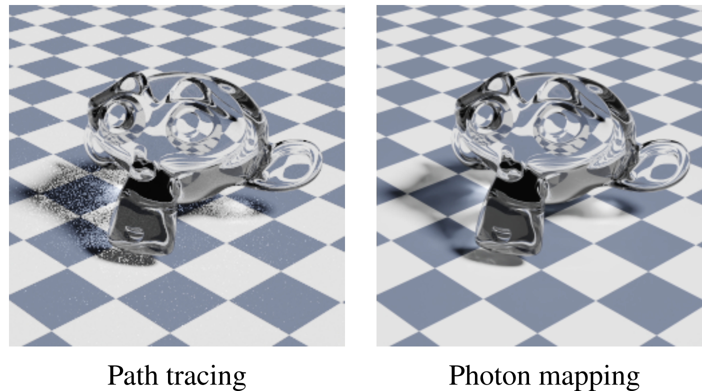


Figure 1.6 – A glass Suzanne monkey rendered using two different algorithms. Note that while both are rendered using very large numbers of samples, the path tracing algorithm fails to converge in that time while the blur present in photon mapping becomes negligible when using a small kernel. The overall visual effects of photon mapping is more perceptually pleasing compared to path tracing for rendering caustics.

to 0, as the probability of a photon to be exactly at a position p is 0. To solve this problem when the number of photons is finite, we can use kernel density estimation (KDE) to average the photons within an area instead, expressed as:

$$L_o(p, \omega_o) \approx L_e(p, \omega_o) + \frac{1}{Nh} \sum_{k=1}^{N} f_d(p, \omega_o, \theta_k) K\left(\frac{p - p_k}{h}\right) \phi_k$$

where K is a kernel function and h the bandwidth (or "width"). Increasing h will widen

Algorithm 4 Basic photon mapping algorithm

while all photons are not yet cast **do**
  choose a random light source in the scene according to its emission
  cast a photon in a random direction using the light source's emissive distribution
  trace that photon's path and save "radiance particles" along it on diffuse surfaces
end while
for every pixel **do**
  cast a ray from the camera's position through the pixel's position
  $p \leftarrow$ closest intersection point of the ray with the scene
  recursively apply path tracing if the surface at p is not diffuse
  accumulate radiance particles around p using a kernel and compute L
  set the pixel color as L
end for

the area within which photons are summed. This decreases variance but increases bias. A schematic illustration of the photon mapping algorithm can be found in Figure 1.5 (right) and a resulting image in Figure 1.6 (right).

One of the main drawbacks of photon mapping is that it is biased by design. If a bandwidth h is bigger than zero, averaging the output of this algorithm many times will not converge to a correct solution, as the result will be blurry from averaging photons over an area. It also has to store all the photons in memory, which can be expensive when using large numbers of photons. We will not discuss further improvements that can be brought to photon mapping, but many of them exist. Notably, Progressive Photon Mapping (PPM) [11] and Stochastic Progressive Photon Mapping (SPPM) [10], as they relieve photon mapping from its biased nature by using multiple passes.

CHAPTER 2

ACCELERATION OF PATH TRACING

Monte Carlo algorithms such as the one used in path tracing are computationally very expensive, and often do not produce an acceptable image given constraints on time, computational power, or monetary budget. As path tracing is very simple to understand and implement, and produces unbiased and photorealistic images, substantial effort and research have been invested to improve its performance.

In this chapter, we give an overview of the main families of acceleration methods for path tracing. This gives context and helps understand the motivation for creating our novel acceleration method described in Chapter 3. This overview also provides a point of reference to compare the strengths and weaknesses of existing algorithms. As making an extensive literature survey on the acceleration of path tracing is very difficult, we condense both previous work, related work, and background into this chapter while providing information about the techniques as much as possible with mostly intuitive explanations, in order to avoid overwhelming the reader. However, we must warn that this list is far from exhaustive and does not come even near to describing all existing acceleration methods, not to mention the many other methods for global illumination that can be used with path tracing simultaneously for the purpose of acceleration. The field of rendering is simply too vast to be able to condense each algorithm's contributions into a few pages. Rather, the reader can expect to see a few simple algorithms that hold the core ideas which can be further developed into the methods used today.

2.1 Overview

This chapter is subdivided into four sections: an introduction to the terminology, sampling methods, information reuse methods, and screen space methods. The first section describes the terminology behind the acceleration of algorithms, such as computational efficiency, consistency, bias, and variance reduction.

The second section describes the largest family of acceleration methods, sampling methods, that modify the sampling process at the heart of path generation. These methods add major modifications to the path tracing algorithm itself, and therefore combining them with other algorithms from the same family may not be a trivial task. Many of the methods in this family are mathematically proven to be unbiased and are also used as a base method to be combined with algorithms from the next two families if needed.

The third section introduces information reuse methods. These methods can usually be described as adding a fast cache for difficult-to-compute methods within the path tracing algorithm. If an expensive function often returns similar values, we can store that function's output in a cache. Caching methods often take advantage of strong priors in the scene, lighting conditions, and temporal/spatial coherency for applying corrections. For example, if a light switch is toggled on/off or an object appears in the scene, we might want to clear the cache or update the cache intelligently as its information might be outdated. Due to the nature of caching, many methods in this family are biased, with moderate visual side effects. However, these methods can bring important performance improvements to the path tracing algorithm, often many orders of magnitude compared to sampling methods. They are used extensively in real-time applications such as video games or interactive software, when speed is of the utmost importance.

The fourth and final section describes the most controversial family of methods, screen space methods. These methods completely ignore the 3D nature of the scene, and use very strong priors from the 2D image itself, that is, "use the image to estimate

the image". As they work with pixels from an image, their speed only depends on the resolution and not on the complexity of the scene, which means that they can be trivially parallelized on the GPU to offer unprecedented acceleration. However, in many cases, these biased methods have severe visual side effects that can be hard to rectify without adding constraints to the scene.

2.1.1 Computational Efficiency

Computational efficiency refers to the general property of an algorithm that evaluates how many resources it needs to complete a task. Even if the algorithm itself cannot be modified, any subtask within the algorithm can still be improved, which in turn improves the overall performance of the algorithm. For example, one obvious improvement to a subtask within the path tracing algorithm would be to speed up ray-scene intersections. A naive intersection algorithm would need to check each object against the ray, and find the closest intersecting object. This is very slow if there are many objects, as each ray needs to loop over a long list of objects. Checking one million ray intersections against one thousand objects would require one billion comparisons for a single ray. Using a better data structure such as a bounded volume hierarchy (BVH) can significantly reduce the number of comparisons. In the general case where primitives are well distributed in a scene, the BVH intersection algorithm has a time complexity of $O(\log(n))$, thus each ray in our example (with 1M rays and 1000 objects) only needs to be intersected $\log_2(1000) \approx 10$ times instead of 1000 times, which amounts to a total of ten million comparisons, two orders of magnitude fewer than the naive method.

2.1.2 Convergence, Bias, and Variance Reduction

In statistics, an estimator is said to be consistent if it converges [1] to the true value as the number of samples (sample size) increases. Both path tracing and photon mapping are consistent. In path tracing, increasing the number of samples towards infinity gives the correct radiance. In photon mapping, if we reduce the size of the kernel towards zero while increasing the number of photons towards infinity, we also obtain the correct value for radiance. This is the basis for Progressive Photon Mapping (PPM) [11].

An unbiased estimator has the property that the mean (average) of its output distribution is the same as the correct value, no matter what sample size we choose. Path tracing is unbiased while photon mapping is biased. In path tracing, if we take a finite number of samples in each pass, but repeat this process infinitely and average the results, we obtain the correct value. However, in photon mapping, if we choose a finite number of photons with a non-zero bandwidth for the kernel, averaging the results for an infinite number of iterations will not produce the correct value, as averaging many blurry outputs from a non-zero bandwidth does not produce a more detailed output.

The variance of an estimator describes how spread apart is its output distribution. Given a random variable X, its variance corresponds to $\mathbb{E}[(X - \mathbb{E}[X])^2]$. A high variance implies that each time we use the estimator, we can obtain a very different output. Variance often manifests itself as image noise in the output of a global illumination algorithm. Reducing the variance of an estimator is essential to improve its performance, as a low variance implies that we will have a higher probability to obtain a good value when sampling the estimator only a few times. Some variance reduction methods are unbiased, while others introduce bias by using a strong prior or regularization. Often biased variance reduction methods are much more effective and flexible in practice than unbiased ones, as they can benefit from additional parameters that let the user choose the

1. Convergence here refers to the convergence of random variables in probability theory.

strength of variance reduction against bias. In classical statistics, this is often referred to as the "bias-variance tradeoff".

2.2 Sampling Methods

Sampling methods aim to improve the sampling process of paths. In the basic path tracing algorithm, as we create and randomly sample paths uniformly, we might generate a lot of paths that add a negligible or even a nil contribution to the final result. Generating less useful paths wastes a lot of computations for little benefit, especially if they are common within a complex scene. These methods often try to sample the most useful paths first in order to improve convergence and performance.

2.2.1 Early Stopping

As discussed in the previous sections, the path tracing algorithm requires evaluating paths of unbounded lengths in order for the result to be correct and unbiased. However, in practice, computing a path of infinite length is impossible. An arbitrary limit has to be chosen in order to terminate the path tracing algorithm. Fortunately, shorter paths

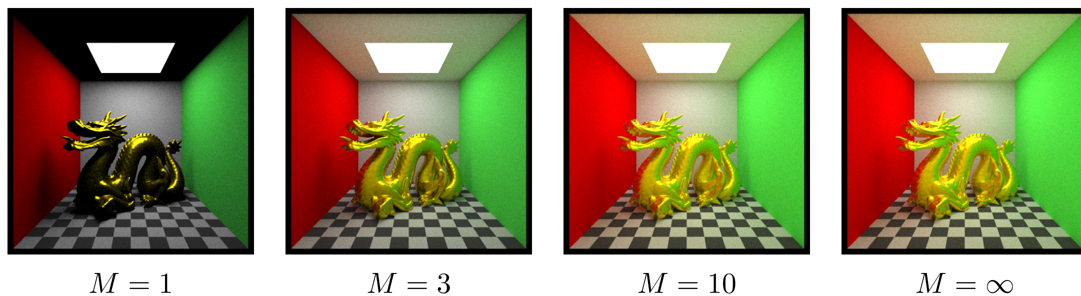


Figure 2.1 – A golden Stanford dragon rendered using path tracing with a maximum path length of M and 1024 samples per pixel. As M increases, we observe new paths (e.g., the ceiling at $M = 3$), and a brighter image overall. $M = \infty$ was computed using Russian roulette (discussed in Section 2.2.2).

are often more important than longer paths as they have higher throughput on average [2]. However, as some light sources may only be reached after a large number of bounces, this can introduce a considerable amount of bias in the output, while at least it guarantees that the algorithm will stop. We can also add additional conditions such as stopping the algorithm when the throughput could become negligible, meaning that evaluating further bounces would be unnecessary. Early stopping causes a loss of energy in the estimation that makes the rendered image darker, with the magnitude of the effect depending on the importance of the truncated paths. An example of early stopping (when $M \neq \infty$) is given in Figure 2.1.

2.2.2 Russian Roulette

As early stopping is biased, Russian roulette can be used as an unbiased alternative to aggressive early stopping. This algorithm consists of randomly deciding to terminate the path tracing algorithm with a probability q each time that we may increase the path length. Unlike early stopping, we can make this algorithm unbiased by simply dividing the throughput by $(1-q)$ each time that we increase the path length to get the new throughput. This division corrects for the lost energy caused by randomly terminating. The probability q can be chosen arbitrarily or be proportional to $1-T$, where T is the throughput. In this case, the probability of termination will be high when the throughput is close to 0. Note that even though this method helps in terminating less useful paths while staying unbiased, it also increases variance as the contribution is set to 0 when a path is terminated while the contribution is increased instead when the path is not terminated. Russian roulette is the simplest of all the sampling methods and is always implemented in even the most basic path tracers. However, it does not guarantee termi-

2. This is true on average as the only way for this to fail is to use Dirac deltas for all BRDFs (e.g., a room with perfect mirrors on all sides). As long as one of the BRDFs in the scene is not a Dirac delta, throughtput will decrease on average as path length increases, because the product of n values between 0 and 1 never increases when n increases.

nation unless combined with early stopping, usually chosen to be a very large maximum path length in order to have as little bias as possible.

2.2.3 Adaptive Sampling

One way to reduce variance when given a limited budget of samples is to estimate variance during the sampling process using previous samples and favor sampling areas with high variance. In path tracing, this usually consists of computing the variance of each pixel and allocate more samples to pixels that have higher variance. This way, areas that converge very quickly do not need to be evaluated many times, saving computational power. Adaptive sampling is consistent but biased, as the average of weighted averages is not equal to the true average. Methods to alleviate this bias have been proposed [18].

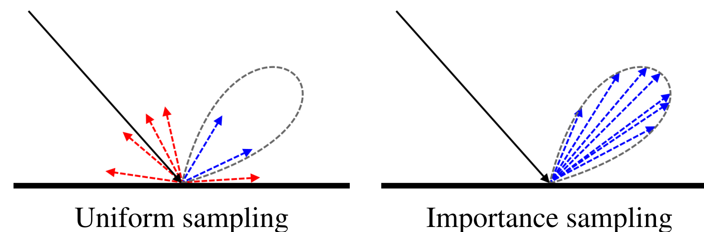


Figure 2.2 – Visualization of random samples using uniform vs importance sampling. The gray lobe in a dashed line represents the throughput distribution of that surface and the red arrows indicate rays with throughput of 0. The importance sampling algorithm samples low throughput rays less often, thus improving efficiency.

2.2.4 Importance Sampling

In importance sampling, instead of sampling uniformly, we sample a probability distribution that results in a higher probability of larger throughput for the path. With this sampling method, we favor sampling paths that have a higher throughput and pay less attention to insignificant paths. A common and simple distribution to sample from is the cosine hemisphere distribution, which is proportional to the $n \cdot \omega$ term used to

compute the throughput of a path. This distribution generates more direction vectors that are close to the normal than the uniform hemisphere distribution, which makes the generated paths more likely to have a higher throughput due to the cosine term in the LTE. In order to make the result unbiased, we divide the throughput with the probability density function (PDF) $p(x)$ of the distribution used in sampling instead of dividing by the volume like before when sampling uniformly. This is exactly as described in Equation 1.3.

2.2.5 Next Event Estimation

Trying to find a light source by sampling random directions is hard. It would be more efficient to sample the light source too at the end of each path length. We are more likely to find emissive surfaces when explicitly sampling light sources and checking if the ray is obstructed by an object in between the light source and the surface (see Figure 2.3). However, deciding on which light source to sample when there are multiple light sources is difficult, and the naive implementation of Next Event Estimation (NEE) would decide either to sample a random light or all the lights. Better NEE implementations can be built using resampling methods, explained in the following sections.

2.2.6 Bidirectional Path Tracing

As stated in the previous chapter, one of the major weaknesses of path tracing is its inability to efficiently estimate caustics and scenes where there are small areas with very bright light sources. Bidirectional path tracing aims at correcting these weaknesses by launching rays from the camera and from light sources, then trying to connect them to form a full path from the camera to the light source. This helps finding paths from the camera to the light when the light source is very small or is very hard to find from the position of the camera, as shown in Figure 2.4.

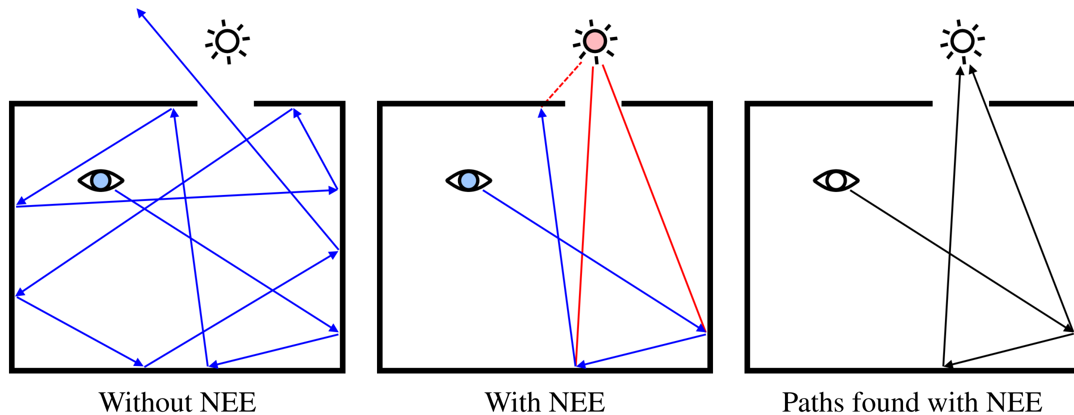


Figure 2.3 – Visualization of the advantages of using Next Event Estimation (NEE). Red segments are explicitly computed by checking for intersections against the light source. This algorithm lowers the average path length required to find a light source.

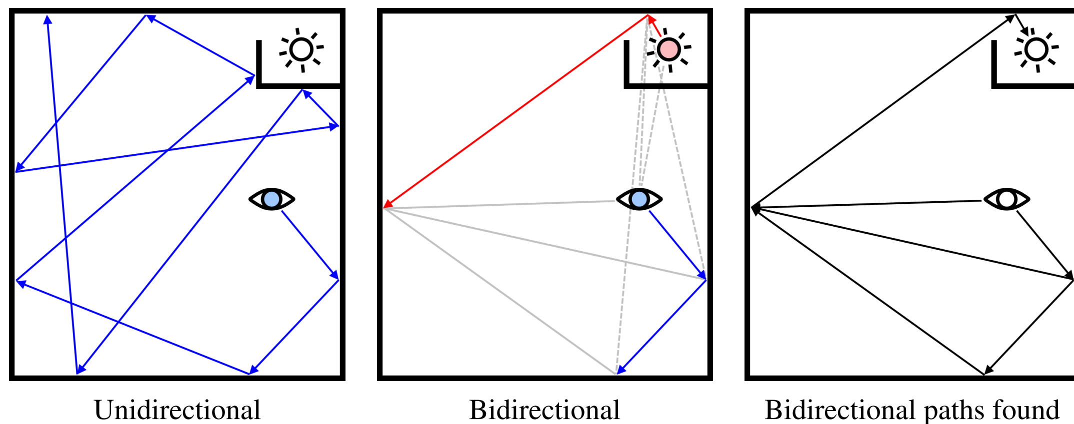


Figure 2.4 – Visualization of the advantages of bidirectional path tracing. For scenes where the light is very small or hidden away, unidirectional path tracing might have difficulties in finding the light. Bidirectional path tracing generates paths from light sources (in red), which are then properly connected with camera paths (in blue).

Each path in this algorithm is more expensive to generate compared to traditional path tracing, but as paths can be reused, and connections can be found intelligently and then weighted using multiple-importance sampling (MIS) [31], it converges in fewer iterations in more complex scenes. For more detailed information, bidirectional path tracing is described extensively in the work of Veach and Guibas [32].

2.2.7 Metropolis Path Tracing

Metropolis light transport [33] takes the bidirectional path tracing algorithm even further. The basic idea is similar to bidirectional path tracing, but we also mutate good paths that contribute well to the prediction in order to find new paths. A simple visualization is shown in Figure 2.5. This general idea is also similar to genetic algorithms. However, instead of solving an optimization problem where we try to find a global minimum, we solve a sampling problem where we want to sample as much as possible paths that are diverse and important. For example, outlier paths that have a strong contribution to the radiance are very important for caustics.

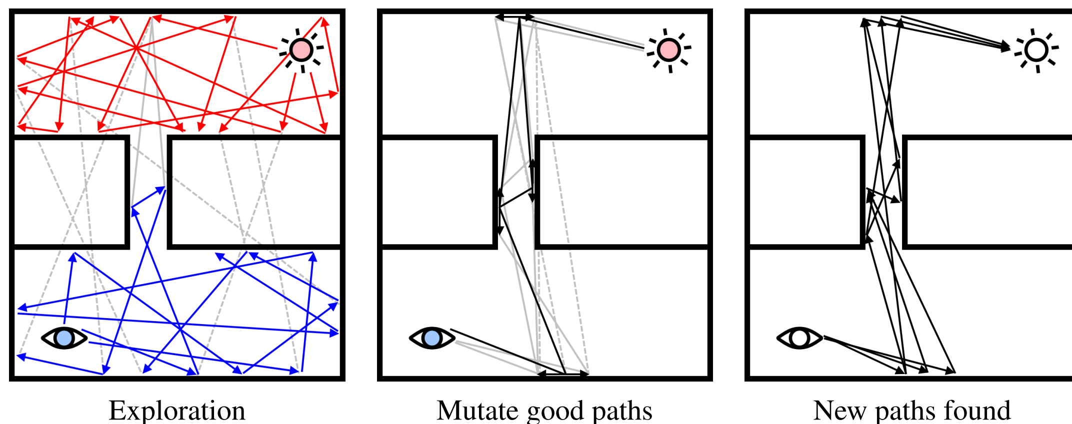


Figure 2.5 – Illustration of the Metropolis algorithm. Mutating good paths can be an excellent strategy to find similar good paths to a light source.

While this algorithm is very good at rendering caustics and indirect lighting, it is more expensive to evaluate per sample compared to other simpler methods. Often Metropolis path tracing is used alongside regular path tracing for rendering complex scenes, where the algorithm can be selected depending on sampling conditions in order to speed up simpler light paths.

2.2.8 Resampling

Resampling methods such as RIS [28], GRIS [20], ReSTIR [3], and ReSTIR GI [25] are all based on similar principles to importance sampling. These algorithms use a reservoir of previous samples to weigh and change how future samples are generated. For example, in the case of ReSTIR, it improves the NEE algorithm by saving previous light sources that were successfully sampled in a reservoir and weighing future samples using that reservoir by reducing the probability to sample occluded light sources. ReSTIR improves convergence significantly for scenes that have many small light sources as traditional path tracing with NEE is ineffective at finding light sources in these types of scenes. ReSTIR GI further improves the algorithm to include past indirect light samples in the reservoir. We not only sample light sources directly, we also sample previous paths and sub-paths saved in the reservoir that are connected to an unoccluded light source, similarly to how bidirectional path tracing connects two paths together. The ReSTIR family of algorithms also accounts for changes in the scene during rendering by using temporal reprojection, which allows these algorithms to be used for real-time rendering of interactive media.

2.3 Information Reuse Methods

Information reuse methods are methods that accelerate rendering by re-using previous estimations stored in a fast cache. Often assumptions are made such as a static scene or static light sources. Furthermore, some of these methods can only accelerate one part of a global illumination algorithm and are biased. As such, they are usually limited in scope and constrained in the type of scenes they can render, but they offer excellent acceleration for the specific purpose they were designed for. Using information reuse methods alongside with a sampling method usually offers the acceleration needed for

interactive applications, such as 3D video games.

2.3.1 Irradiance Caching

The original irradiance caching algorithm was proposed in 1988 by Ward et al. [34]. Its basic idea is to cache the estimated irradiance at positions of Lambertian surfaces in the scene that were previously sampled. If a full path from the camera to the light source is found, it is possible to compute the radiance at each vertex along that path, and to accumulate its radiance in an irradiance cache for use in future samples. An irradiance cache is more memory efficient compared to other methods that store the entire path of the light, like Metropolis path tracing, but can suffer from problems such as blurriness if the irradiance cache is not precise enough or has a too low resolution. Similarily to photon mapping, if the previously sampled positions are stored as particles, a kernel density estimator can be used. If the cache is represented instead by a regular grid or an adaptive grid (e.g., octree) where we accumulate the irradiance in each cell, simply querying the grid is sufficient. However, as the discontinuities of the grid can be visible in the rendering, interpolation or smoothing is commonly used. The main disadvantage of irradiance caching is that it assumes that the scene and lighting conditions are static. If there are significant changes in the scene, the cache might need to be cleared or corrected using temporal reprojection. For a static scene, pre-generating an irradiance cache and saving it alongside the scene itself for future use is very beneficial. More recent algorithms that encode the cache within a neural network [23] have shown to be faster and suffer less from discontinuities and loss of details.

2.3.2 Light Probes

The light probes or irradiance probes methods are very similar to irradiance caching, but with a key difference. Instead of an irradiance cache present in the entire scene,

we can approximate that cache using discrete light probes that effectively act like pre-baked light sources that simulates indirect lighting [3]. We first generate the probes by solving the LTE in advance, and while we usually evaluate the full LTE for each sample, we can query the light probes as light sources during sampling instead of the actual light sources. This reduces the number of bounces that we need to evaluate, as light probes scattered around in the scene act like simulated indirect illumination. With most implementations of light probes only requiring one bounce, we can use ray tracing to evaluate direct illumination against the probes instead of the much slower path tracing algorithm. Traditional light probes cannot encode adequately light effects from specular and mirror surfaces, and require additional bounces to handle them. More recently, light probes have been augmented with light fields [22], which allows more complex effects to be taken in account by the probes themselves.

2.3.3 Temporal Reprojection

For interactive applications, the scene and lighting conditions might change often and unpredictably. If we are reusing previous estimations, we need to correct the position of that estimation when an object moves. For example, the ReSTIR [3] algorithm uses reprojection when reusing samples that are on moving objects and light sources to improve acceptance rates. If samples are not reprojected when an object moves, that sample might not be at an adequate location anymore or might even be inside an object.

Reprojection in 3D scenes is rather simple as we just need to track the movement and velocity of all vertices or compute the position difference by subtracting the previous positions with the new positions. For screen space reprojection methods where past

3. Light probes can be placed regularly, using an adaptive algorithm, or manually in the scene. As the light probes are capable of emitting different amounts of light in different directions, only a few of them are needed for simpler scenes. This is commonly achieved using spherical harmonics. However, a related algorithm, Instant Radiosity [16], uses simpler but many more virtual point lights (VPLs) instead of a few probes. In this case, each VPL is acting as a point light source that emits the same amount of light in all directions.

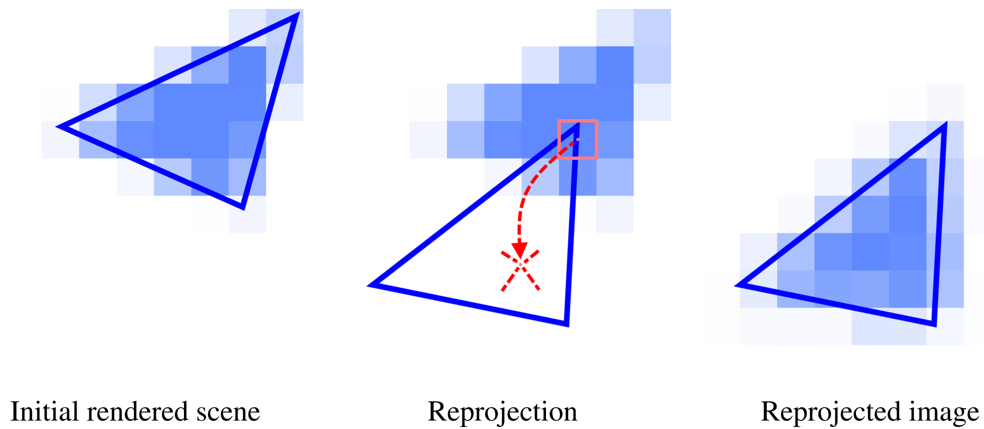


Figure 2.6 – An illustration of the 2D reprojection algorithm, where the reprojection of a single pixel is shown in the middle figure. The final reprojected image is entirely created from the initial rendering by reprojecting the pixels using the motion vectors of the triangle.

information is kept in a 2D buffer and is attached to the camera position, we can use optical flow or project the 3D velocity of vertices from the scene to the 2D screen using the camera projection matrix. Special care has to be made for occlusions and self-occlusions when working in 2D, as a previously hidden object can become visible without any associated reprojected data. This might cause a conflict with the overridden data at the target position. An example of 2D reprojection is shown in Figure 2.6.

2.4 Screen Space Methods

Screen space methods work on the final 2D rendering or on intermediate buffers such as depth, normal, or albedo buffers. As they ignore the 3D nature of a scene, these methods do not scale with the complexity of the scene, but rather only with the screen resolution. If a smaller resolution is used, these methods can be extremely fast compared to other methods. The only downside is that hidden objects cannot affect the result of these algorithms. Currently, these methods are often used to provide global illumination effects for devices with less compute power, or for highly interactive applications that

require very high framerates, such as VR games, especially if the accuracy of the illumination is not very important. Screen space methods generate results that are inaccurate but often look right at a first glance, as they generally remove obvious and distracting artifacts present in the final image.

2.4.1 Screen Space Ray Tracing

Screen space ray tracing [21] is a family of algorithms that ignore the true 3D geometry of the scene and use 2D proxies such as depth and normal buffers to compute illumination. Intersecting rays with a scene can be expensive if the geometry is very complex, and might not bring obvious benefits to the image if most of the scene is not visible on screen. Intersecting rays against a 2D buffer is very quick and scales in complexity only with the resolution of the image. For scenes where effects of on-screen objects are the most obvious and hidden objects do not contribute much to the image, for example indirect illumination on small objects or a puddle of water in the lower part of the screen reflecting the upper half, screen space ray tracing can be a fast way to approximate the effects of global illumination. This method will fail in cases where objects outside of the viewport contribute a lot to the illumination of the image, such as a very bright light behind the viewport that is shining in the forward direction. In such cases, the rendering will fail to include any effect from objects not visible on screen.

2.4.2 Denoising and Super-Resolution

Denoising [2, 6, 36] and super-resolution [35] methods consist of first quickly generating a low quality image, then using an algorithm to remove noise, increase resolution, remove some artifacts, or improve the image in other ways. These methods have seen a resurgence in recent times due to the popularity of deep learning and their ease of use after training a end-to-end deep neural network, where we can simply input everything

(e.g., noisy image, depth buffer, normal buffer, etc.) into the network to obtain the final rendering as output. As they work on the image itself, they are very easily adapted for any global illumination algorithm as they do not modify the algorithm itself. In the simplest case, the denoising or super-resolution algorithms can take the output from the path tracer, as is without any additional information, and process the image as it sees fit. Deep learning denoising and super-resolution methods are especially effective at improving unconverged images from path tracing, as the neural network can be trained using ground truths obtained from the same path tracing algorithm. Common issues in machine learning such as domain generalization and overfitting problems are much less pronounced here as a high quality dataset can be created for training using the same global illumination algorithm used for rendering, whereas other domains using machine learning do not always have correct and high quality ground truths.

CHAPTER 3

NEURAL PROBABILISTIC PATH PREDICTION (NPPP)

As seen in the previous chapter, many sub-problems within path tracing can be accelerated by modifying how paths are sampled and by reusing past information. While the field of deep generative models [8, 17, 19, 26] has started to mature recently and was found to be very useful in conditional image generation [15, 27, 29], language modelling [4, 30], and audio synthesis [24], very few attempts (that we are aware of) have been made to investigate the potential applications of modelling the path tracing process of light transport itself as a conditional generative process. In this chapter we will show similarities between conditional generative processes and the sampling process used in path tracing, and then lay down the foundations for constructing a fast conditional generative model that allows the skipping of ray evaluations during sampling given as few assumptions as possible on the 3D scene.

3.1 Motivation

A common method to accelerate the path tracing algorithm for real-time use is to truncate the maximum number of light bounces to a very small number (e.g., less than four bounces). This limits the number of ray-scene intersections and reduces divergence on the GPU, but introduces a large amount of bias from discarding the energy from longer paths, which darkens the scene displayed in the image. Fewer bounces also mean that light travels for a shorter distance before stopping, which can look unrealistic in some scenarios, for example a long corridor with reflective walls and a light source at the other end. Finding a way to allow us to evaluate longer paths without taking a performance hit from more computations and GPU thread divergence will be our main

goal. This is what our method tries to solve.

We can describe a single light path from a camera into a 3D scene as an ordered list of vertices $(x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n)$, where each vertex encodes the position, total throughput, and accumulated radiance of a ray interacting with the scene. Note that while the sampling process requires us to find each vertex, one at a time, a probabilistic generative process does not. To compute the final radiance of a single path, we only need the information from vertex x_n and the emissivity of the surface at x_n, as the final vertex already contains the total throughput and accumulated radiance.

Thus, the general idea for our method is to use a deep generative model to output the final vertex x_n given x_0 as input, as it does not need to reproduce the sampling process $x_0 \ldots x_{n-1}$ that led to the sampling of x_n. If this generative model is fast enough, it can replace the sampling process within the path tracing algorithm for longer paths. As the reader might guess, generating light paths with a statistical model is not that simple and the many problems and pitfalls to this approach will be discussed in the following sections.

3.2 Theory

To successfully create a conditional generative model for path tracing, we first need to define the conditional probabilities of each vertex with respect to previous vertices, sampled in the past. From now on, it will be more useful to describe paths as probability distributions instead of individual paths.

3.2.1 Statistical Model of Path Tracing

As stated before, we can use an ordered list of vertices $(x_0 \rightarrow \cdots \rightarrow x_n)$ to describe a full path of light of length n where each vertex x_i encodes the position, outgoing direction, total throughput, and accumulated radiance.

We can define a joint distribution of all possible paths of length n as follows:

$$P(X_n, ..., X_0) \tag{3.1}$$

where X_i is the random variable that describes all the possible outcomes for x_i.

As the position of the camera x_0 in a path tracer is pre-defined and given as input, we can describe the previous distribution as a joint distribution of vertices conditioned on the initial camera ray described by vertex x_0. The path tracing algorithm can be described as updating a pixel's value by sampling and integrating over the radiance given by this conditional distribution

$$P(X_n, ..., X_1|X_0) \ .$$

As directly computing this joint distribution is intractable, we can expand the joint conditional distribution using the chain rule and sample the light path incrementally. This gives us the probabilistic formulation for the path tracing algorithm.

$$\begin{aligned} P(X_n, \ldots, X_1|X_0) &= \frac{P(X_n, \ldots, X_0)}{P(X_0)} \\ &= \frac{P(X_n|X_{n-1}, \ldots, X_0)P(X_{n-1}, \ldots, X_0)}{P(X_0)} \\ &= \frac{P(X_n|X_{<n})P(X_{n-1}|X_{<n-1}) \ldots P(X_1|X_0)P(X_0)}{P(X_0)} \\ &= P(X_n|X_{<n})P(X_{n-1}|X_{<n-1}) \ldots P(X_2|X_1, X_0)P(X_1|X_0) \ . \end{aligned} \tag{3.2}$$

For the basic path tracing algorithm, as the probability of sampling the next vertex only depends on the previous vertex and not on all previous vertices, we can further simplify

the conditional probability equation:

$$\begin{aligned} P(X_n|X_{<n})\dots P(X_1|X_0) &= P(X_n|X_{n-1})\dots P(X_1|X_0) \\ &= \prod_{i=1}^{n} P(X_i|X_{i-1}) \; . \end{aligned} \tag{3.3}$$

This final conditional probability equation represents the statistical model of the basic path tracing algorithm. It is very similar to the generative processes seen in recurrent models and autoregressive models. In fact, this equation is a special case of the equation describing an autoregressive model, where the model has a context size [1] of one, which can also be considered to be a Markov process.

3.2.2 Sampling

Sampling x_n from the distribution described in Equation 3.3 is slow, as we need to sample each vertex incrementally from x_0 to x_{n-1} before being able to sample x_n. In order to obtain x_n quickly, what about sampling from the following conditional distribution?

$$P(X_n|X_0) \; .$$

If this distribution is known, sampling from it would be extremely efficient as we are skipping many ray-scene intersections, especially if n is large. However, path tracing does not allow us to directly sample x_n from x_0 without computing the full path. Fortunately, we can find an estimator f using deep learning. A neural network with parameters θ can learn the following function either explicitly or implicitly:

$$f(X_0, \theta) \sim P(X_n|X_0) \; .$$

1. The context size here refers to how many previous values in a sequence are taken into account by an autoregressive model to predict the next value's probabilities.

One requirement for this estimator f is that it needs to be quicker to sample from, when compared to evaluating the full path of light, in order to be useful at accelerating path tracing.

The next major obstacle for creating this estimator is that a neural network's parameters θ are fixed and cannot be easily modified without retraining the whole neural network. Training is especially expensive and best avoided for real-time applications. It would not be acceptable to train an estimator for the whole scene as it would break as soon as any object moves.

The solution to this problem is to train an estimator that learns the probability function of a sub-path for only a portion of the scene.

$$f(X_p, \theta_{p \to t}) \sim P(X_t | X_p) \; . \tag{3.4}$$

Instead of sampling the full path, we can skip the vertices between t and p,
where $n > t > p > 0$.

$$P(X_n | X_{n-1}) \ldots P(X_t | X_p) \ldots P(X_1 | X_0) \; . \tag{3.5}$$

Now, static portions of a scene can be approximated by this estimator, and dynamic objects can use the original path tracing algorithm. Sampling from this new distribution with an estimator in the middle can be much more efficient than sampling from the original distribution if $(t - p)$ is big and the estimator is very quick. A simple diagram of an estimator used with path tracing is provided in Figure 3.1.

Furthermore, the number of estimators is not limited to one in a scene; there can be multiple estimators in a single light path, which can further accelerate the path tracing process. In fact, each object in a scene that has a static mesh can be considered static and have a corresponding estimator to accelerate the path tracing process. Euclidean

transformations [2] on objects can be applied to the estimator by transforming the vertices x_t and x_p using the forward and inverse transformations respectively. For non-static meshes (e.g., animated characters), additional parameters can be added to the estimator to learn the effect of non-Euclidean transformations.

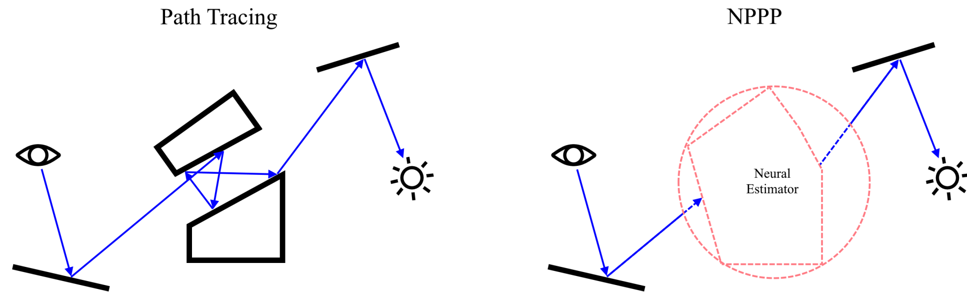


Figure 3.1 – The general idea of NPPP. Instead of computing many bounces and ray-scene intersections, a neural network is used for encapsulating light interactions within a set of potentially complex objects.

3.2.3 Light Fields

In order to parameterize our estimator, we take inspiration from light fields. A light field is a vector function that returns the amount of light flowing through every point in space. The 5D function that describes all possible light rays in 3D space is called the plenoptic function:

$$L(x, y, z, \theta, \phi) \ , \tag{3.6}$$

where x, y, z are the 3D coordinates, and the two additional variables θ, ϕ encode the direction of the ray. For any point and direction in 3D space, this function can return a value representing some data for that ray (e.g., radiance, visibility, etc.).

2. Euclidean transformations preserve lengths and angle measures (i.e., rotation, translation, and reflection).

Similarly to a light field, we can define a "latent path field" that encodes the probability distribution of the final vertex of a sub-path given an arbitrary vertex and direction in space.

$$O(X_p, \theta, \phi) = P(X_t | X_p) \; . \tag{3.7}$$

If the region of interest of this arbitrary field (e.g., radiance, latent, visibility, etc.) is always outside of a convex hull [3], we know that any ray outside would not encounter any obstacle that is not our target object within the convex hull, thus the values along any ray would be constant before hitting the convex hull. For example, if the field encodes radiance, the radiance values along a ray will always be constant before hitting the convex hull, where we would not know if it was changed unless we look inside of the convex hull. With this property, we can compress this field to only encode information from within the convex hull on its surface and not waste space to represent all of the constant rays outside of the convex hull.

This poses a constraint on our new estimator. We might get an incorrect result if we start the sampling process inside of the convex hull that we are estimating or have some external object intersecting the space within the convex hull. However, this constraint can be respected in most scenes without too much difficulty. If somehow we must have two objects intersecting each other's convex hulls or allow the camera to go inside of an object's convex hull, we can simply detect when that happens and revert back to the standard path tracing algorithm without using our estimator for that specific object.

Furthermore, instead of learning a complicated distribution of rays on a surface described by an arbitrary convex hull, we can simplify our estimator $f(x_p, \theta_{p \to t})$ by estimating the distribution of rays on a spherical projection that bounds the convex hull. The bounding sphere parameterization simplifies our estimator significantly, as we do not need to know about the geometry, textures, or other properties of objects inside of

3. A convex hull of an object is the smallest convex shape that the object can fit in.

the bounding sphere, and as we are only interested in the distribution of rays exiting this sphere conditioned on a ray entering this sphere. Additionally, intersection checks against a sphere are much faster than most other arbitrary geometry representations.

If a ray hits the bounding sphere, a "mask field" can be used just like our path field in order to check if the ray has hit the object itself (or a set of objects). This mask field (which can use the same parameterization as the latent field) simply returns 0 or 1 depending on whether an incoming ray hits the object itself after entering the bounding sphere. For better performance, if the mask field does not confirm a hit, the estimator is not used. This allows the correct handling of cases where an external object intersects with the bounding sphere, we just need to move the predicted rays on the bounding sphere slightly back along the direction of the ray, otherwise the predicted output rays might start from within another object. In Figure 3.1, the output ray from the estimator starts at the surface of the bounding sphere, but might need to be moved back a bit (the distance of the dashed blue line) if another object is slightly inside of the bounding sphere. However, for accurate results, external objects still cannot reside (partially or fully) within the convex hull itself. An additional estimator can be trained to learn the distance from the sphere to the convex hull so that we can know exactly how far back to move the ray. However, this is not needed if only a few objects slightly intersect the volume of the object's bounding sphere. In practice, moving back for a predetermined distance or to the plane that intersects the object's midpoint with the same normal as the direction works well. The latter method will always keep the ray inside of the bounding sphere.

For a better mask field that does not suffer from "staircase" artifacts (see Figure 3.8) from the limited resolution of a discrete regular grid (shown in Figure 3.3), which stores the values for the field, a potential candidate for future research is to use signed distance fields [9] instead of just encoding a binary value. Signed distance fields can allow smooth

representation of curved surfaces when the underlying grid is very coarse.

Finally, the convex hull parameterization can also be trivially modified to allow the encoding of something similar to an environment map or hemispherical skylight, where the camera is always inside of the inverted inner convex hull. For example, we can accelerate the volumetric path tracing of faraway clouds using NPPP. The animation of those clouds can also be encoded in an additional parameter and learned in advance. NPPP is more versatile than a hemispherical skylight as it is an encoding placed in 3D space that allows portions of the scene to be "outside" while a hemispherical skylight is only 2D and is placed at infinity.

3.3 Algorithm

In path tracing, efficiently sampling from the probability distribution is more important than finding its density function. Therefore, our estimator $f(x_p, \theta_{p\to t})$ should be a generative model. Training a density estimation model would not be beneficial if we cannot sample efficiently from it.

Pre-existing generative models such as GANs, VAEs, and autoregressive models are either too slow or too hard to train for our purpose of accelerating path tracing. We will design a simple and fast generative model specifically for our task of sampling outgoing rays X_t depending on incoming rays X_p.

For any specific incoming ray x_p in a scene, sampling the outgoing rays multiple times will give us a 12-dimensional point cloud (two sets of x, y, z values for the position and normalized direction vector, and two sets of r, g, b values for total throughput and accumulated radiance), which is a good approximation of X_t if this point cloud is very large. The core idea of our method is to train a deep neural network that allows efficient sampling of X_t by learning a mapping from a random point cloud generated with a normal distribution $N(0, 1)$ to the outgoing rays' point cloud using Chamfer loss in the

output (which is order invariant). This nonlinear mapping of a high-dimensional space will allow us to sample an approximation of X_t by first sampling $N(0,1)$, which is very easy and quick, and passing it through the neural network.

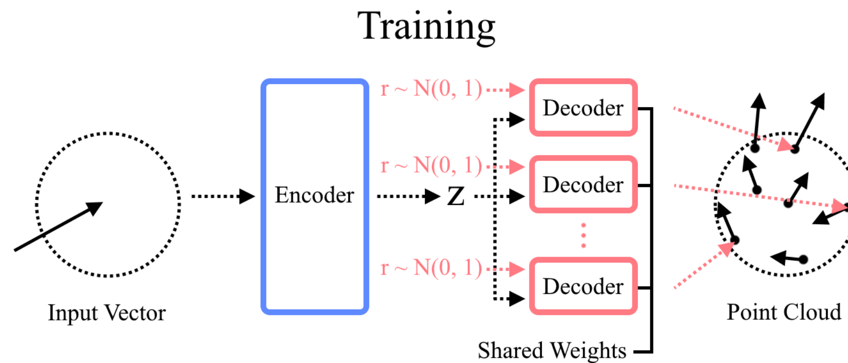


Figure 3.2 – Illustration of the neural architecture during training. The decoders have shared weights but do not share the random vector r, which is re-sampled at each training iteration for each decoder. In our implementation, each object or group of objects uses a completely different model. Sharing the decoder is possible, but it complicates the training procedure.

3.3.1 Neural Architecture

To be able to sample from the distribution $P(X_t|X_p)$, our neural network has to take into account the value of x_p. Directly inputting x_p to the neural network is not feasible as the network would need to be very large, and consequently very slow when trying to encode the distribution $P(X_t|X_p)$ itself in the weights of the neural network.

To solve this problem, we can divide our network into two parts, an encoder $f(x_p) = z$ that will map x_p to a latent vector z, and a decoder $g(z) = P(X_t|z) \approx P(X_t|X_p)$ that will take the latent z and a random vector r from $N(0,1)$ to produce one sample of X_t. Both the encoder and decoder are implemented using a basic multilayer perceptron (MLP) architecture.

After training, we can sample the encoder from all needed positions and directions,

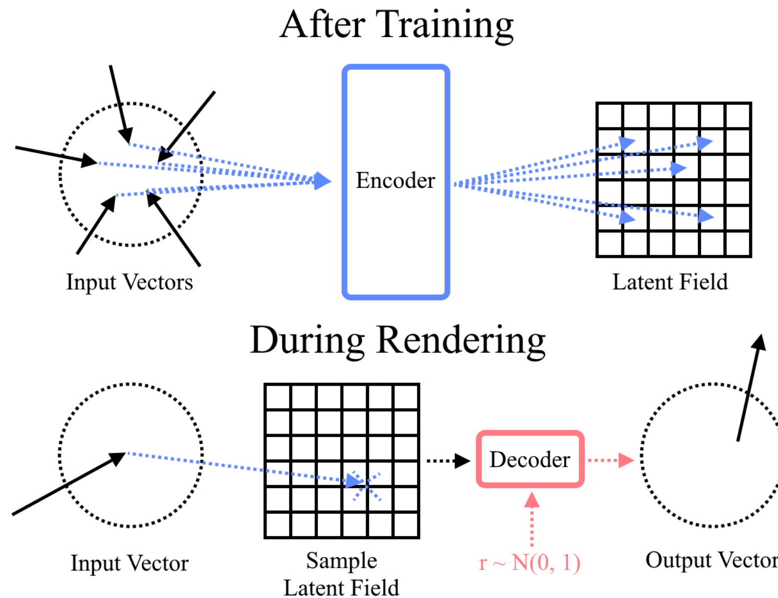


Figure 3.3 – Illustration of the latent field method used to bypass the encoder during rendering. This allows the estimator to be extremely fast when the decoder is lightweight.

and store each of the latents z corresponding to each input vector in a regular grid parameterized by $O(X_p, \theta, \phi)$ or other parameterizations such as the bounded sphere described earlier. Any parameterization will work as long as it does not take too much memory and preserves needed details. For any new incoming ray x_p, we can linearly interpolate the regular grid to obtain its corresponding latent z, which can then be used to sample $P(X_t|z) \approx P(X_t|X_p)$ during rendering.

As we will not be using the encoder to sample from X_t during rendering, we can make it as large as we want. Larger encoders allow faster convergence during training and result in a better encoding of the distribution within the latent z. The decoder can be made very small and lightweight for fast sampling on the GPU.

3.3.2 Acceleration Structures

For our basic implementation of NPPP, we use a two-level acceleration structure, also called top-level/bottom-level acceleration structures (TLAS/BLAS), where each individual object's mesh is represented in a bottom-level bounding volume hierarchy (BVH), and the placements of those objects in the scene are encoded in a top-level BVH. This allows us to attach an estimator to the leaf nodes of the top-level BVH and reuse the mesh and estimator if an object is duplicated in the scene with an Euclidean or uniform scaling transformation. Also, depending on whether we want greater speed or more accuracy, we can choose to use the estimator without evaluating the bottom-level BVH or go with the standard path tracing algorithm. Because our estimator is a neural network with a fixed number of parameters, sampling a path using the estimator reduces divergence [4] on a GPU compared to sampling using the BVH, where a BVH is often traversed at different depths depending on the location of the intersection. Reducing divergence further accelerates the path tracing algorithm on a GPU.

If multiple objects are static in relation to each other and form together a convex hull that no other object or the camera will ever enter, for example a boat tied to the roof of a car, an extension to this algorithm can be made by allowing it to dynamically train and attach estimators to non-leaf nodes of the top-level BVH during the rendering process.

3.3.3 Training

To train an estimator for an object or a group of objects, the target objects are first placed in an empty scene. Then, we select a random initial position on the bounding sphere and a random direction which forms the incoming ray; they will serve as input

4. Due to how modern GPUs are designed, parallel code is most efficient when all threads run the same operations at the same time. Divergence occurs when some threads run different code or run for longer than others due to branching, which results in slowdowns as all threads executed in parallel must wait for the diverging threads to finish before the others can continue running.

to the neural network. Using the same path tracing algorithm that is used for rendering, we randomly trace the path of that incoming ray until it leaves the object's convex hull and save the outgoing position, direction, total throughput, and accumulated radiance as a 12D vector k times in order to obtain a 12D point cloud of size k. If the object is not intersected by the incoming ray, a flag is set for later when creating the "mask field". A larger point cloud lets us train a better and more accurate estimator, but significantly increases the size of the dataset on disk or in memory. For reference, the uncompressed dataset size for our Stanford Dragon model was 10 GiB. One example pair of an input and a target in the dataset consists of an incoming ray paired with its corresponding point cloud. Finally, we generate as many examples as possible for our dataset in order to maximally cover the object with incoming rays. If some input rays are more important than others (e.g., when modeling a pipe that carries light inside, where both ends of the pipe have a more complex output ray distribution than the outer surface), another ray parameterization can be used and/or a non-uniform distribution of input rays around the object can be generated in the dataset to optimize for quality and memory usage in specific use cases.

During training, the decoder is duplicated until there are k decoders with shared weights. There should be as many decoders as points in the point cloud. All random inputs r to each decoder should be a new and different value sampled from $N(0, 1)$. The input to the encoder is the incoming ray as a vector. The output of the many duplicated decoders is the point cloud itself, represented as a unordered list of vectors. The full network is trained end-to-end using Chamfer loss to account for output order invariance and using standard Adam loss for quick convergence.

3.4 Results

The results shown in this section are from our implementation of a standard path tracer in GLSL shaders, using importance sampling, Russian roulette, the Disney principled shader [5], and a two-level BVH acceleration structure. NPPP is incorporated by replacing parts of our scene with a neural network estimator when conditions are met. For fairness during benchmarking, the NPPP code inside of the shader is removed when the algorithm is disabled to avoid branching. The machine used to run tests has the following specs: AMD Vega 64 GPU, AMD Ryzen 1700 CPU, 32 GiB of RAM, and Google Chrome as the browser for benchmarks.

The first scene tested, shown in Figure 3.4, is a simple box with colored walls and a golden Stanford Dragon in the middle, similar to the Cornell box. The target quality for this rendering is to use a maximum of five bounces for light $(M = 5)$ and 1024 samples per pixel on a 512×512 canvas. The time needed to render this scene using path tracing only is $38.3s$.

Reducing the maximum number of bounces for the path tracing algorithm alone accelerates the rendering but reduces the overall brightness of the scene due to a loss of energy carried by longer light paths. NPPP allows us to have an additional parameter that controls when to use the estimator. Instead of stopping the path of light, we can opt to use the much faster but less precise neural network estimator after P bounces. This allows us to effectively accelerate rendering without reducing the maximum number of bounces.

NPPP, when used conservatively with $P = 2$, can offer some acceleration without sacrificing visual quality. Compared to lowering the hard limit M on the number of bounces that the path tracing algorithm can generate, it is more efficient with respect to perceived quality to use NPPP instead for longer paths. In this scene, PT + NPPP can produce an almost identical image compared to the original rendering in $0.64\times$ the

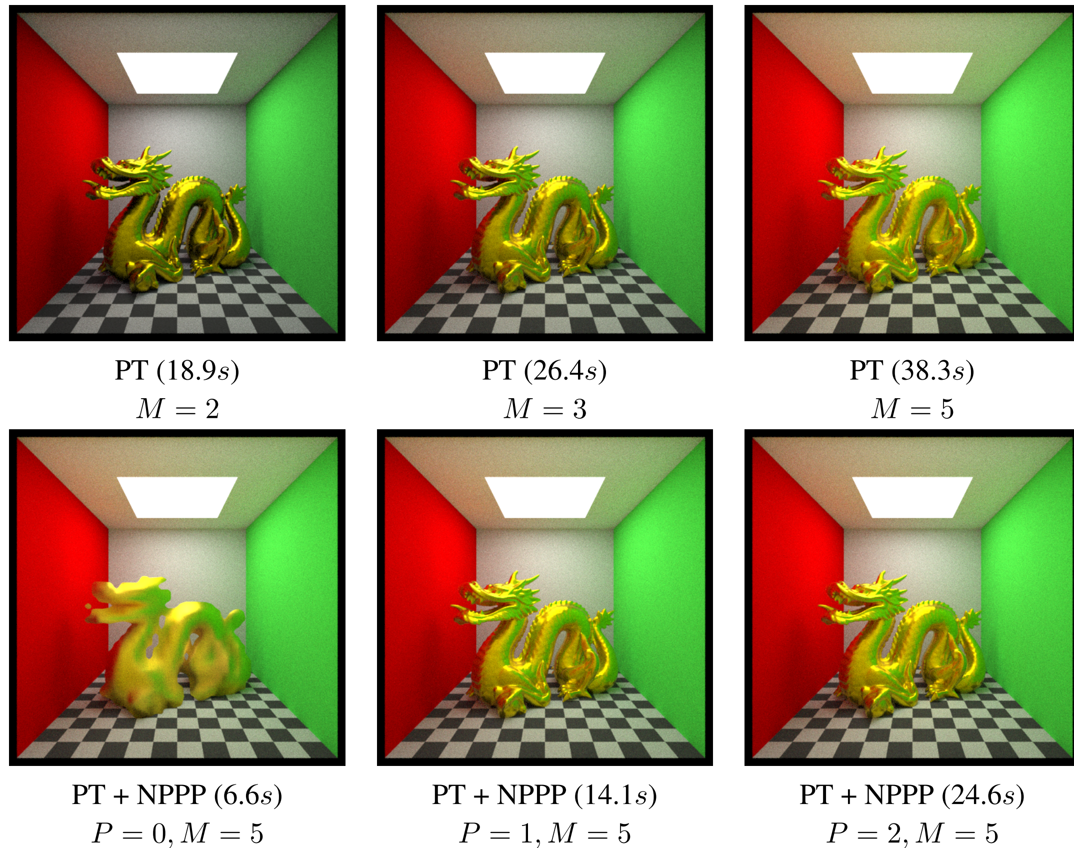


Figure 3.4 – A benchmark of a golden Stanford Dragon with 300k triangles rendered at 1024 samples per pixel (spp) using various parameter values. M is the maximum number of bounces. P is the minimum number of bounces before NPPP is used. NPPP is enabled only for the Dragon model and not for the walls/ceiling/floor. The latent vector size is set to 16. Both latent and mask grids used are of size 32^4 and are quantized to 8 bits, occupying 17 MiB of memory.

When using PT alone, reducing the parameter M increases speed but noticeably darkens the scene. With PT + NPPP, decreasing P yields similar acceleration without noticeable degradation, unless using the extreme case where $P = 0$ (no real geometry bounce is performed)

running time $(24.6s)$.

If we can tolerate a reduction of quality in the shadows and small details in lighting, setting $P = 1$ makes the rendering faster than limiting the maximum number of bounces to two ($M = 2$) and runs in $0.37\times$ as long as the original rendering $(14.1s)$. Empirically,

we find that on average, setting $P = n$ is faster than setting $M = n + c$ because the overhead of using NPPP is very low and the neural network runs in constant $O(1)$ time with respect to scene complexity. How small c can be depends on the complexity of the scene.

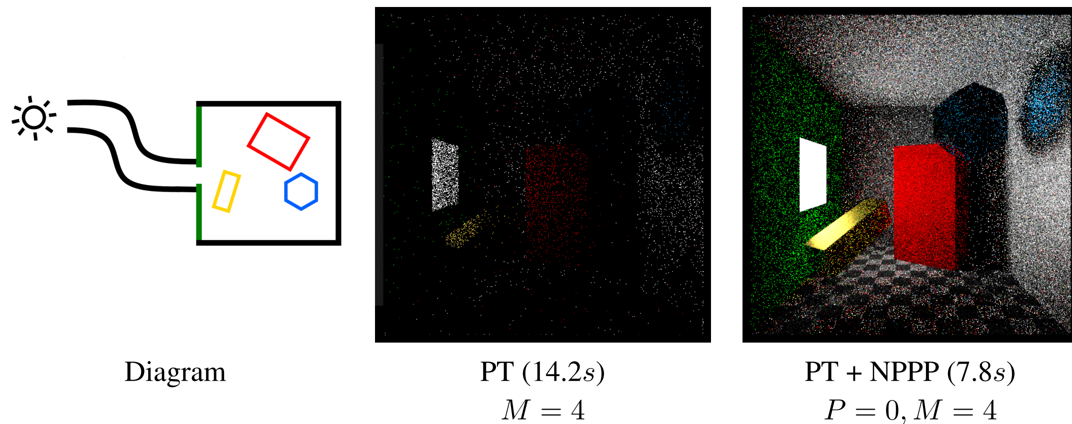


Figure 3.5 – A benchmark of a scene simulating difficult lighting conditions rendered at 1024spp. NPPP is enabled only for the curved corridor and not for the walls and objects in the room. No modifications to the bounding sphere parameterization is used here, and better parameterizations of the NPPP estimator specifically for a curved tunnel should yield even better results (such as allocating the entire latent field to the entrance only).

The second test aims to see if NPPP can allow us to estimate light effects from large numbers of bounces ($M > c$) when given a limited number of bounces ($M \leq c$). In the scene shown in Figure 3.5, the path from the camera to the light source is more complex and requires a larger number of bounces due to the curved tube leading to the light. The curved tube's material is a diffuse white surface with an albedo of 0.9.

In this scene, using only a maximum of four bounces ($M = 4$) does not allow path tracing alone to find the light source effectively. However, by attaching an estimator to the curved tube, a ray entering the estimator will have a larger probability of exiting from the same end and a smaller probability of exiting from the opposite end. Even if the probability of exiting from the illuminating end of the tube is small, it is still much large than the probability without NPPP. For standard path tracing, a ray that bounced

for three times or more in the room will never find the light source due to the curved tunnel, but for PT + NPPP, all rays that enter the tunnel have a chance of exiting towards the light source.

Rendering with NPPP improves convergence (less noise) and reduces rendering time due to not having to evaluate any bounce within the tunnel.

The third scene is used as a stress test for our algorithm and shows that the neural estimator can be instanced without using any additional memory when an object is duplicated and moved in the scene. This scene is very similar to the first one, but we instead lay out in a very large room fourty thousand Stanford Dragons in a (200×200) grid, each having random rotations along their Y axis. In this scene, we observe that NPPP can be used to great effect, where it accelerates the path tracing algorithm almost twofold. When NPPP is used with the settings $P = 1$ and $M = 6$, the rendering takes only 5 minutes and 2 seconds, while path tracing alone with $M = 6$ takes 9 minutes and 45 seconds to produce a slightly higher quality image. The decrease in visual quality from NPPP is almost imperceptible unless it is being compared directly side by side to a ground truth image (see Figure 3.6). We can see that NPPP produces the desired global illumination effects while only needing a single bounce against the dragon geometry ($P = 1$), with higher order bounces being estimated using NPPP. With $P = 2$, the result is almost identical while being faster to path tracing alone. The NPPP overhead in this scene for five NPPP bounces ($P = 1$, $M = 6$) is 1 minute and 52 seconds, as path tracing with a single bounce ($M = 1$) takes 3 minutes and 10 seconds.

3.5 Discussion

One important property to note about NPPP is that it is biased for most scenes due to inaccuracies in the estimator. However, this bias may be more acceptable if the alternative is to limit the number of bounces, which is also biased. One can think of NPPP as

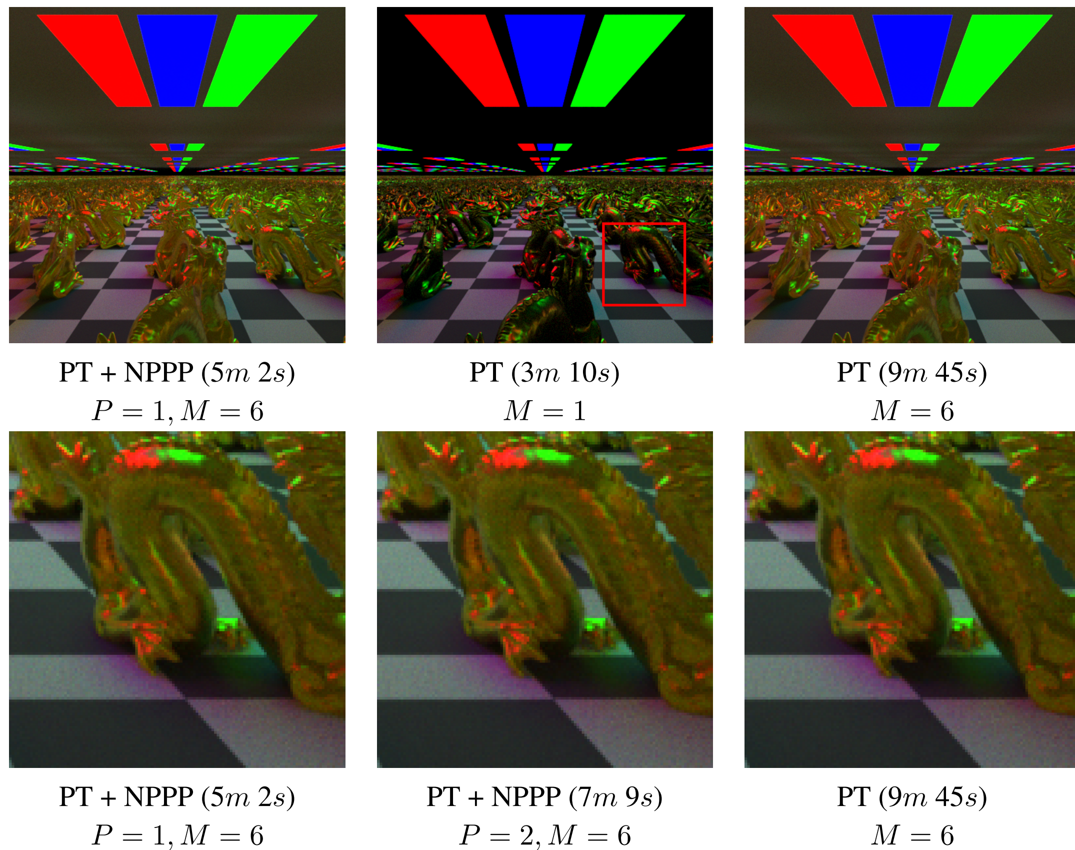


Figure 3.6 – Stress test scene with 40k (200×200) instanced Stanford Dragons placed in a large room for a total of 12 billion triangles rendered at 2048spp. Unless otherwise stated, all other settings are the same as the first scene (see Figure 3.4).

a better approximation of the correct result compared to truncating path lengths. NPPP does not guarantee the accuracy of light transport when it is used, as the neural network is simply an estimation of the properties of the object, and is subject to bias. Further research in this direction might be needed in order to quantify the bias and variance reduction of this algorithm.

Many factors and parameters affect the quality of the estimation given by NPPP. As seen earlier, changing P, which is the minimum number of bounces before NPPP is used, greatly affects the quality of the final image. A lower number for P increases rendering speed but decreases quality, and vice versa. The grid sizes used for the latent and mask

fields also affect the estimator's quality. However, they only change how much memory is used and do not affect performance. If the object is only viewed from a certain angle, we might want to construct a specific parameterization that is more memory efficient or discard parts of the latent field for that specific use case to have a better rendering quality for the same memory requirement. For example, for objects far away, we might only need to store a 2D latent field that only encodes position instead of the full 4D latent field, as small changes in ray direction will not be noticeable.

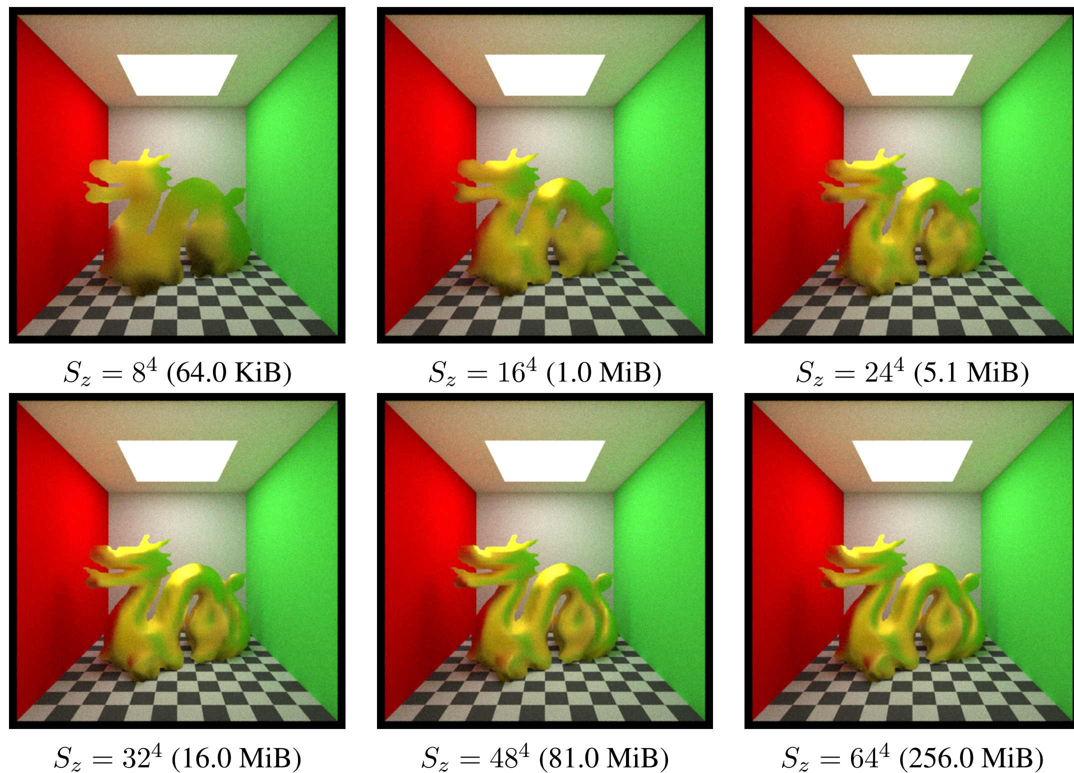


Figure 3.7 – Visual comparison and memory requirements for different S_z (size of latent grid). The mask grid size is set to 64^4. The settings $P = 0$ and $M = 5$ are used for all instances. The latent vector size is set to 16. Both the mask and latent grids are quantized to 8 bits.

3.5.1 Grid Size

Another parameter determined in advance is the size of the grid for both the latent field and the mask field. As our spherical parameterization of the field is 4D, a large field will occupy much more memory. If stored as 32-bit floating point numbers, the latent field is $\frac{32}{8} \cdot d \cdot q^4$ bytes in size, where d is the size of the latent vector and q is the size of one side of the regular grid. For example, a 32-bit regular grid of size $q = 32^4$ with a latent vector of size $d = 16$ occupies 128MiB of memory.

A smaller grid saves memory but results in a much lower quality of images. During our tests, a grid size smaller than 16^4 is barely usable. The only downside of using a larger grid is memory size, as the grid size does not affect performance. Therefore, if memory is plentiful, larger grid sizes may and should be used for a higher quality result. For a 32-bit mask field, its size is simply $\frac{32}{8} \cdot q^4$ bytes. The sizes of the mask and latent fields are independent and can be changed depending on the desired quality and complexity of the object or scene.

3.5.2 Quantization

Encoder-decoder architectures in machine learning are somewhat naturally resistant to degradation from quantization because of the way the encoder compresses similar outputs as similar latent vectors. We can greatly compress the latent field using quantization without much loss in quality. In our tests, the latent field can be quantized to 4 bits and the mask field to 2 bits before noticeable artifacts appear. A 4-bit latent field is $8\times$ smaller than a full-precision 32-bit latent field. As we did not have access to specialized hardware during our tests that can take advantage of quantization, we did not test for performance improvements from quantization. Newer GPUs with specialized ray tracing hardware and machine learning compute units (e.g., NVIDIA RTX 3080, AMD RX 6800 XT) can take advantage of the memory speed and latency improvements from

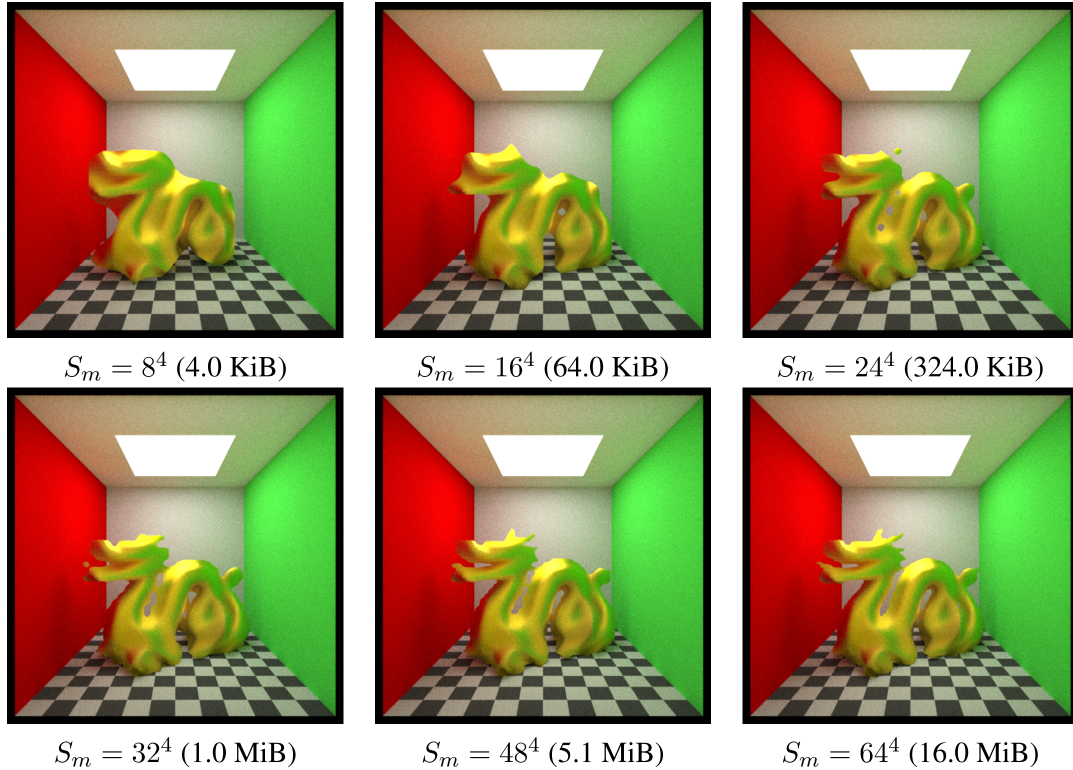


Figure 3.8 – Visual comparison and memory requirements for different S_m (size of mask grid). The latent grid size is set to 64^4. The settings $P = 0$ and $M = 5$ are used for all instances. Both the mask and latent grids are quantized to 8 bits.

using quantized data to improve the speed of NPPP even further.

Recently, methods for training neural networks that take into account quantization [12] have been developed and can be used to further improve the robustness of the estimator to quantization. Future research directions in this domain could be about the compression of the latent vector representation using quantization or some other methods for reducing the memory footprint of the latent and mask grids.

3.6 Implementation Details

In order for the results to be more easily reproducible, we give in this section details about all the steps we took in order to obtain the results discussed above. The basic tools

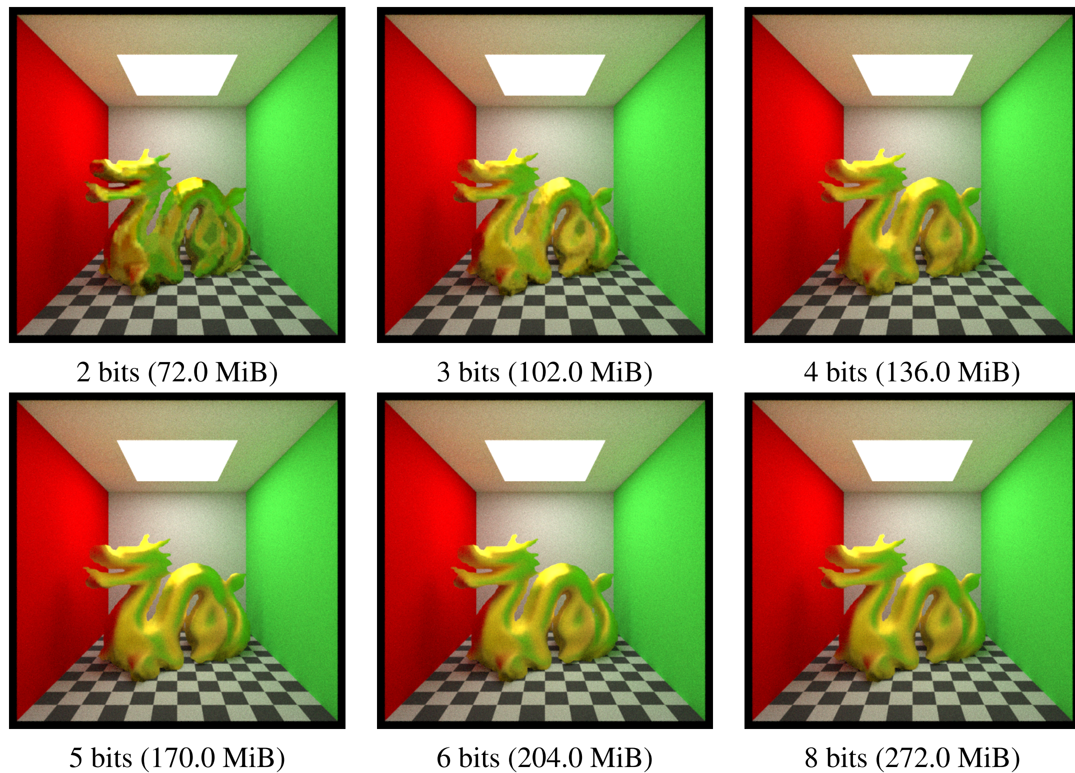


Figure 3.9 – Visual comparison and memory requirements for quantization levels of both the latent and mask grids. The latent and mask grid sizes are set to 64^4. The settings $P = 0$ and $M = 5$ are used for all instances.

required are an open-source rendering engine (PBRT, Mitsuba, etc.) where NPPP can be implemented and a machine learning library (TensorFlow, PyTorch, scikit-learn, etc.) that will ease the training process of the neural network. We opted to use a mix of online open-source code and custom code in order to write a specialized interactive path tracer that runs on web browsers for portability of the demos. We used TensorFlow [1] to train our network but any other machine learning library that supports Chamfer loss will work out of the box.

To generate the dataset, we need to modify the rendering engine to save pairs of input ray and output ray cloud instead of an image. Modifications can be made inside of the path tracing routine to save the position, direction, throughput, and accumulated

radiance of rays instead of accumulating radiance values. After the modifications, we place in an empty scene the object or collection of objects that we want the estimator to learn. To generate the input-output pairs, we choose a random ray that intersects the bounding sphere of our object as input, and run the path tracing algorithm many times in order to obtain many output rays with their corresponding data for throughput and accumulated radiance. To save storage, we can opt to not save output rays that do not intersect our object but only save a flag that will be used later to determine the mask. We repeat these steps by choosing another random input ray until enough input rays cover our object. How many input-output pairs and the size of the output point cloud needed to have an accurate estimator depend on the complexity of the object, but we found that 200k pairs with a point cloud size of 512 are a good starting point for fairly complex objects.

After obtaining the full dataset, the training step is pretty straightforward. Using the neural architecture shown in Figure 3.2, we train two networks. The first network predicts the mask field, where the input is the input ray and the output is a binary value representing whether the ray intersects the object. The random vector r in the decoder is removed and only one decoder is used for learning the mask field. The second network predicts the rays themselves. It is trained as-is with Chamfer loss using the input rays (that intersect our object) and the point clouds. Experimentally, we found that using a 16-dimensional vector for both r and z is good for most objects. We trained the networks for 10 epochs using Adam with parameters $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$.

To obtain the mask and latent fields, we sample the encoder in both of the two networks using a predetermined parameterization, and save the outputs of the mask network and the encoder network for the latent field in a regular grid. A common 4D parameterization for our use case is a pair of points on a sphere described by a pair of two angles (longitude and latitude). Both regular grids can be quantized to 8 bits in order to save

space without any noticeable degradation in quality.

In order to implement NPPP in our path tracer, we need to attach the estimator to the BVH node that bounds the entirety of our object or collection of objects. When a ray intersects a node with an estimator attached, we want to be able to skip all intersections with deeper nodes by using the mask field. We can simply check for intersections against the mask field to determine if the ray has intersected with the object of interest or passed through empty space. If the intersection is confirmed, we can use the estimator to predict a new outgoing ray, skipping most of the code in the path tracing algorithm. If the object was moved using Euclidian transformations, we can simply apply the reverse and forward transformations to the input and output rays respectively from the estimator. Note that we must not use the estimator if the current bounce is smaller than the parameter P, which controls the rendering quality, or if the incoming ray is coming from within the object itself or from a bounce within the object's convex hull. If the convex hulls of objects with estimators are not allowed to intersect with other objects, the last check can be skipped, improving performance.

The estimator is implemented in two parts. First we need to implement the sampling routine for the mask and latent fields. When an input vector is given, we can find the mask value and the latent vector by interpolating values in the mask/latent grids. We used a simple quadrilinear interpolation for our 4-dimensional regular grids. More complex interpolation methods such as cubic or spline interpolation will produce higher quality renderings but will be significantly slower due to the high dimensionality of our grids. After sampling from the grids, we can generate a random vector r from a normal distribution and use it with the sampled latent vector z for the decoder to produce an output ray. As the decoder network is a simple MLP, only a few hundred multiply-add operations are needed, combined with the ReLU function. For the weights of the network, we simply generated our shaders dynamically at runtime to include the weights,

but loading them as uniforms or textures will work too.

Finally, the output ray from the estimator (after performing necessary Euclidean transformations) can be used as-is for the next bounce against the scene in the path tracer, as it already contains values for throughput and radiance. We multiply the estimated throughput with the total throughput and add the estimated radiance with the total accumulated radiance of that path.

CHAPTER 4

CONCLUSION

We explored deep conditional generative models for light transport in this thesis. We showed that an estimator can be built for path tracing to allow the skipping of ray evaluations in arbitrary scenes. This new method is very flexible and allows a higher degree of control over the quality and speed tradeoff it offers to a path tracing algorithm. Furthermore, unlike neural caching methods, the neural network is trained only for each individual object or a group of many objects. It does not need re-training when there are some changes in the scene or lighting conditions. One downside of this method is its memory consumption, where each complex object might need up to tens of megabytes for a high quality rendering. Future research in this direction would be needed to reduce the memory footprint and allow very complex scenes to be represented accurately.

4.1 Future Work

The method presented in this thesis is not only applicable to path tracing, but also to many other light transport algorithms such as photon mapping, light tracing, bidirectional path tracing, and volumetric path tracing. The estimator network would need to be tweaked in order to take into account the additional information required for these methods. For instance, for photon mapping, we can modify the estimator so that it returns the position of a deposited photon in addition to the output ray. For bidirectional path tracing, we would need an additional estimator to handle path connections. For volumetric path tracing, the algorithm works as-is if the volume in question is far away (e.g., clouds). It can bring significant performance improvements as volumetric path tracing is very expensive due to ray marching and multiple scattering. If the camera is within the

fluid such as inside of a smoke filled room, we would need to use a 5D parameterization for the mask and latent fields. We also need to consider the case when an object intersects with the fluid. If the object is static, it can be included within the estimator, but if the object can move anywhere it wants, another solution must be found.

Furthermore, improvements to the algorithm itself are also promising, such as finding good parameterizations for each type of objects and scene, finding memory efficient representations of the latent field, and designing higher quality/faster neural architectures.

Finally, we can avoid the need to create a pre-trained neural estimator if we implement the training step within the rendering engine. This would allow the flexibility of immediately using new objects without the need to train the neural networks manually using an external library.

BIBLIOGRAPHY

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Steve Bako, Thijs Vogels, Brian Mcwilliams, Mark Meyer, Jan NováK, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Trans. Graph.*, 36(4), jul 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073708. URL `https://doi.org/10.1145/3072959.3073708`.

[3] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Trans. Graph. (Proceedings of SIGGRAPH)*, 39(4), July 2020. doi: 10/gg8xc7.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[5] Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7, 2012.

[6] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[7] Michael F Cohen, John R Wallace, and Pat Hanrahan. *Radiosity and realistic image synthesis*. Morgan Kaufmann, 1993.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems*, 27, 2014.

[9] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. 2007.

[10] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):1–8, dec 2009. ISSN 0730-0301. doi: 10.1145/1618452.1618487. URL https://doi.org/10.1145/1618452.1618487.

[11] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5), dec 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409083. URL https://doi.org/10.1145/1409060.1409083.

[12] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and train-

ing of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[13] Henrik Wann Jensen. Global illumination using photon maps. In *Eurographics workshop on Rendering techniques*, pages 21–30. Springer, 1996.

[14] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4): 143–150, aug 1986. ISSN 0097-8930. doi: 10.1145/15886.15902. URL `https://doi.org/10.1145/15886.15902`.

[15] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[16] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 49–56, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0897918967. doi: 10.1145/258734.258769. URL `https://doi.org/10.1145/258734.258769`.

[17] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[18] David Kirk and James Arvo. Unbiased sampling techniques for image synthesis. *ACM SIGGRAPH Computer Graphics*, 25(4):153–156, 1991.

[19] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the fourteenth International Conference on Artificial Intelligence and Statistics*, pages 29–37. JMLR Workshop and Conference Proceedings, 2011.

[20] Daqi Lin, Markus Kettunen, Benedikt Bitterli, Jacopo Pantaleoni, Cem Yuksel, and Chris Wyman. Generalized resampled importance sampling: Foundations of ReSTIR. *ACM Trans. Graph.*, 41(4), jul 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530158. URL https://doi.org/10.1145/3528223.3530158.

[21] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, 2014.

[22] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–11, 2017.

[23] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372*, 2021.

[24] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[25] Yaobin Ouyang, Shiqiu Liu, Markus Kettunen, Matt Pharr, and Jacopo Pantaleoni. ReSTIR GI: Path resampling for real-time path tracing. In *Computer Graphics Forum*, volume 40, pages 17–29. Wiley Online Library, 2021.

[26] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1530–1538. JMLR.org, 2015.

[27] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S Sara Mahdavi, Rapha Gontijo Lopes, et al. Photorealistic text-to-image diffusion models with deep language understanding. *arXiv preprint arXiv:2205.11487*, 2022.

[28] Justin F. Talbot, David Cline, and Parris Egbert. Importance resampling for global illumination. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*, EGSR '05, page 139–146, Goslar, DEU, 2005. Eurographics Association. ISBN 3905673231.

[29] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International conference on machine learning*, pages 1747–1756. PMLR, 2016.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[31] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[32] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *EGWR '94*, pages 147–162, 1994.

[33] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 65–76, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0897918967. doi: 10.1145/258734.258775. URL https://doi.org/10.1145/258734.258775.

[34] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph.*, 22(4):85–92, jun 1988. ISSN 0097-8930. doi: 10.1145/378456.378490. URL https://doi.org/10.1145/378456.378490.

[35] Lei Xiao, Salah Nouri, Matt Chapman, Alexander Fix, Douglas Lanman, and Anton Kaplanyan. Neural supersampling for real-time rendering. *ACM Trans. Graph.*, 39(4), jul 2020. ISSN 0730-0301. doi: 10.1145/3386569.3392376. URL https://doi.org/10.1145/3386569.3392376.

[36] Bing Xu, Junfei Zhang, Rui Wang, Kun Xu, Yong-Liang Yang, Chuan Li, and Rui Tang. Adversarial monte carlo denoising with conditioned auxiliary feature modulation. *ACM Trans. Graph.*, 38(6), nov 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356547. URL https://doi.org/10.1145/3355089.3356547.