Neural BRDF Integration and Analysis in Pixar's RenderMan

Dylan Li

Abstract:

Capturing both the detail and accuracy of a complex scene while also ensuring an efficient and cost-effective rendering model has become an aspiring goal for the field of computer graphics. A bi-directional reflection distribution function is the ratio of reflected radiance in direction ω_o to incident irradiance from direction ω_i. (Nicodemus, F. E., Richmond et al., 1977, 3). An attempt at creating a BRDF with neural networks has been attempted in recent years, mainly with high-end hardware and high computational costs. We investigated the mathematics of physically-based shading and why evaluating those models during ray/path tracing is computationally expensive. We identified the specific mathematical operations in a multi-layer BRDF evaluation that create computational bottlenecks during path tracing and used RenderMan to see these lighting terms in action, and analyzed whether a neural network could more efficiently render in an offline path renderer's architecture. We present an alternative application of a NBRDF incorporated into Pixar's RenderMan trained on the MERL dataset for metallic materials. We evaluate the benefits and drawbacks of this neural network and also

propose an extension of the work to include a greater all-in-one neural renderer that is usable by the average consumer.

1. Introduction

With the recent rise in technology comes major innovations in rendering capability. However, as every scene gets more and more complex and renders get more and more detailed, rendering times get slower and slower. A BRDF (also known as a bi-directional reflection distribution function), according to Nicodemus, F. E., Richmond et al. (1977) is the ratio of reflected radiance in direction ωo to incident irradiance from direction ω_i, and characterizes how a surface reflects light as a function of the incident and outgoing directions.(Nicodemus, F. E., Richmond et al., 1977, 3). In this paper, we discuss these difficulties, how neural BRDFs have helped with rendering efficiency, and how neural BRDF evaluation acceleration could affect the landscape of commercial rendering. We have trained a neural network that is able to predict reflectance values on a variety of materials inside Disney's RenderMan.

The foundational reason for why path-tracing is so difficult has to relate to James T. Kajiya's "The Rendering Equation".

2. The Rendering Equation and The Rendering Problem

Equation #1 Modern Form of Kajiya's Rendering Equation

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) cos\Theta_i d\omega_i$$

Where:

L_o : is the radiance/intensity

L_e : is the radiance emitted by the surface poipoint x in the outgoing direction ω_o

f_r : is the BRDF's calculation for how the surface scatters light (varies based on material's nature.

Θ_i : is the angle between the incident light direction ω_i and the surface normal at x

Ω : is the integration domain, the hemisphere of all directions incoming above the rface at x.

This is a modified version of Kajiya's original rendering equation on page 143 of his revolutionary paper, which is written in terms of directions instead of points. The rendering equation states that the light leaving any surface point in any direction is the sum of all the emitted light plus an integral, over the hemisphere of incoming directions with the incoming radiance weighted by the surface's BRDF and a cosine term in Equation #1. We can break this down into parts. The integral looks at all the directions the light comes from. This could be the floor walls, other objects, etc. The integral essentially takes in all the directions of light above a surface and gathers it together into one term. A BRDF tells you what fraction of the light arriving in ω_i gets reflected in the direction ω_o. It is bidirectional because it requires both of these directions at the same time, and it is called a distribution function because it describes how the reflected light is spread across all outgoing directions. In essence, it is a mathematical function that requires 2 directions to produce a result that mimics the texture a material is made of. For example, it can calculate the light scattering all around a tennis ball, or the smooth surface of a plastic sphere that doesn't reflect as

much light. The cosine term in Equation #1 measures the angle at which the light hits an object to determine how direct the light hits to determine the brightness.

Figure 1: Illustration of Symbols and Where They Sit

Figure 2: Visualization of The Cosine Term

Figure 3: Visualization of How a Renderer Estimates The Integral

*note visuals were created with the assistance of generative AI by asking to illustrate these concepts with a visual diagram

Equation #2 Original Form of Kajiya's Rendering Equation

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''\right]$$

Where:

$I(x, x')$: is the related to the intensity of light passing from point x' to point x

$g(x, x')$: is a "geometry" term

$\epsilon(x, x')$: is related to the intensity of emitted light from x' to x

$\rho(x, x', x'')$: is related to the intensity of light scattered from x'' to x by a patch of surface at x'

Essentially, Kaijiya's original formulation is point based, summing over every surface point in the scene and calculating how much light reaches this point from there. The modified directional form was developed by Immel, Cohen, and Greensburg in 1986. Veach, E. (1997) standardized this form, and it is now used in a lot of rendering software, like Disney's RenderMan.

There are a few key differences. For one, instead of measuring light from one point to another, you measure the light coming one direction to a point. The "p" function becomes the BRDF "f_r". The g function and the cosine factors inside the p function get turned into $\cos\Theta_i$ and the solid angle $d\omega_i$. We want to integrate over a solid angle, a three-dimensional analog of an angle instead of the surface area so we don't have to calculate the geometry term (a surface that's farther away will cover a smaller solid angle). We also run the integral over the hemisphere Ω instead of a point, where you would have to integrate over every surface S in the scene, which would decrease the renderer's efficiency. We also change from "intensity" to "radiance" since that is what measured data like the MERL dataset is calibrated against.

The issue with this rendering equation is the integral over all possible light paths in a scene. Because incoming radiance depends recursively on the rest of the scene (look at

the x terms in Equation #2), you are essentially trying to evaluate an integral that cannot be evaluated. Every time a light ray bounces off another surface, you have to consider the radiance again. Thus path tracing is forced to be an approximation of an intractable integral.

Now that we know the rendering equation cannot be solved analytically, we estimate with Monte Carlo sampling. Monte Carlo Sampling is essentially a broad class of computational algorithms that at its core consists of repeated random sampling to produce an approximation of a deterministic problem, in our case, a problem that would be too complex for mathematical analysis. For example, a simple use case of a Monte Carlo approximation would be using random points to generate an approximation of the area of a random shape.

Figure 4: Visualization of Area Example With Monte Carlo Sampling

We throw a bunch of random points into a defined section of a 2D plane. We can then measure how many of those points are contained in the shape and how many landed outside of it. Because points are randomly distributed over a specified area, it is reasonable to assume that the area of the shape is proportional to the number of hits over the total number of thrown points. In other words, the ratio of hits over total points is a decent approximation for the area of the shape to the specified area we chose. This allows us to approximate any seemingly complex shape that would be extremely difficult to evaluate

analytically into a simple algebraic equation. A computational example could be the following:

Figure 5: Computational Area Example With Monte Carlo Sampling

Applying this into rendering, we can, for instance, cast many random light paths for each pixel, and then average the radiance. This would give us a decent approximation for what the actual radiance would be if we were to theoretically solve it analytically. Because of these incredibly useful traits, Monte Carlo approximation can be used to evaluate the integral term in our rendering equation. This is what is called Monte Carlo integration, and is what modern renderers use to calculate the radiance of anything.

An extension of this “random sampling” is importance sampling. This is where mathematicians use randomness to estimate something but incorporate probability to improve the approximation. By using a probability density function, where there is more emphasis put on the areas that are more important, the approximation becomes more accurate with fewer samples.

However, it is important to note that just increasing sample counts doesn’t mean you will get perfect results. AMD GPUOpen (2024) showed an example where visible noise persisted even at 32,768 samples per pixel. This, again, further illustrates how there is no guarantee that increasing the number of samples will result in a cleaner look because of the random nature of the method.

Figure 6: Path tracing with 32,768 samples per pixel

Evermotion Archinteriors scene rendered by the AMD Capsaicin Framework. Reproduced from Kazmierczyk et al., "Neural Supersampling and Denoising for Real-Time Path Tracing," AMD GPUOpen, 2024. https://gpuopen.com/learn/neural_supersampling_and_denoising_for_real-time_path_tracing/

3. Methodology

Previous work has shown that it is possible for a small neural network to represent a measured BRDF in a compact manner and that importance sampling the BRDF is also possible in isolation (Sztrajman et al., 2021). This means that we are able to test the network's ability to reproduce a material's characteristics without all the tools a path tracer offers. Zeltner et al. (2024) has shown that the NBRDF can run inside a real-time GPU pipeline (the process of converting our 3D geometrical data into a 2D image that the screen can display).

Building off of Sztrajman's NBRDF implementation, we propose a new extension and application of NBRDFs, a modified training pipeline, and implementation into an offline path

tracer's internal plugin architecture. When RenderMan looks at a scene, it constantly runs a Monte Carlo approximation over and over again, constantly evaluating the direction of light, tracing the ray, and weighing the result.

Our neural based plugin needs to be able to do this process constantly inside RenderMan. We created both GenerateSample() and EvaluateSample() that have to agree on their estimation. GenerateSample() essentially uses a proxy distribution, a distribution sampled based on where the BRDF is likely to be bright.

EvaluateSample() takes the direction of light that RenderMan gives it, converts it to different directions usable by the Rendering Equation with another function StandardToHalfDiff(), and then evaluates the same formula that GenerateSample does, which is:

Equation 3: EvaluateSample() Calculation

$$p(\omega_i) \;=\; 0.5 \cdot pcos(\omega_i) \;+\; 0.5 \cdot pGGX(\omega_i)$$

We started with MERL training data, which contains the measured reflectance values of 100 materials. Each material is a binary file holding a 90×90×180 grid over the half-angle parameterization (theta-h, theta-d, phi-d) and about 33 MB per material. We trained two networks on two different materials: alum-bronze, saved as nbrdf_weights_v2.pt, and chrome, saved as nbrdf_weights_v2_chrome.pt . Each material under this methodology thus has its own neural network. We chose these 3 materials for a few reasons. Firstly, we

want to be able to ensure that our measurements with alum-bronze were not isolated only to that material. Second, we want to be able to differentiate metal from nonmetal and see whether the network can still handle materials with less radiance. Rusinkiewicz 6D encoding is a specialized strategy of parameterization to estimate or track an object's 3D translation and rotation. The details about how it works isn't important, just how we use it. We utilized Rusinkiewicz 6D encoding to help our neural network learn how to evaluate three things: what is the direction of the light, where is the view direction, and what is the resulting radiance? For a shiny surface, we need to know where the light, camera, and midpoint are. We use the half vector h, the direction at the midpoint between these two directions. When a surface reflects light, it only shows to the viewer when its normal is along the h vector; essentially telling you how close the surface was to being directly in the direction of the camera. It is called 6D because the three angles (θh, θd, φd) need to be converted to unit vectors in Cartesian coordinates:

Equation 4: h-vector and d-vector Definitions

$$h = (sin\,\theta h,\ 0,\ cos\,\theta h)$$

$$d = (sin\,\theta d\ cos\,\varphi d,\ sin\,\theta d\ sin\,\varphi d,\ cos\,\theta d)$$

All of the MERL materials are isotropic, meaning they look the same no matter how you spin it.

We then used a MLP, a small kind of neural network that consists of a small stack of layers doing arithmetic. Our MLP uses 3 layers, with 6 numbers going in from our

Rusinkiewicz encoding, which then go into 2 layers each containing 64 neurons that added a weighted sum of the 6 numbers, an adjustable constant, and then passes that result through ReLU (if a number is negative, make it zero). The output layer contains 3 neurons, representing the RGB reflectance values. We then use a function exp(x), an exponential output that ensures final answers are positive and that there are no big differences in brightness. Since exp(x) can approach 0 but can never reach it, we subtract values by 0.001 to help with directions where the surface reflects nothing at all. We finally use a flat binary export, a folder containing all the network's numbers in a fixed order. The file (19212 bytes) is laid out in order of: layer 1 weights, layer 1 biases, layer 2 weights, layer 2 biases, the output weights (64x3), and the output biases (the RGB values).

The plugin opens the file with LoadFromFile in binary mode, makes a *fread* call for each block and fills the array in. We do this so we don't have to link the plugin to a PyTorch reader, decreasing the loading time.

To measure the accuracy of the neural network during training , we use log mean squared error. This essentially measures the squared difference of the logs. Unlike typical MSE, log-MSE allows us and the neural network to better understand how off the network's prediction is from the actual value in the database.

Equation 5: Brightness-Weighted Sampling in Training Code

```
brightness = train_table[:, 3:].mean(axis=1)
weights = brightness + brightness.mean() * 0.1
```

```
probs = weights / weights.sum()
```

Talking a little more about the plugin itself, there are a few intentional choices we made. Firstly, It is reflection only, because MERL doesn't have transmission data for these materials. Outputs are also clamped to be greater than or equal to 0. If the weights weren't able to load, we just automatically set them to the constant ambient term of 0.5 and print an error to stderr. The network evaluation is a plain scalar loop, with around 4700 multiply-adds every evaluation. Also, any of the samples that landed below the hemisphere we are evaluating don't assign a weight and PDF.

Going into performance, we measured performance under three criteria: correctness, efficiency, and speed. Correctness essentially goes through some internal checks like energy conservation and analytical match to ensure that the model produced by the neural network is reasonable in its results. This is a pass/fail test rather than a numerical measurement. Second is the sampling efficiency. We measure how good the importance sampling strategy is in the BRDF model.

We have three variants of sampling strategy the plugin: cosine only, GGX half-vector sampling, and a combination of both. Cosine-weighted hemisphere sampling is where we draw directions with a probability that is proportional to $\cos(\theta_i)$ (the cosine term in the rendering equation). This avoids wasting samples in the edges of the hemisphere where the cosine term would make these directions negligible anyways. Walter et al. (2007) talks about GGX sampling to derive microfacet BRDFs, which statistically model the scattering of

light from a large collection of microfacets (small bumps in a rough surface). Rather than sampling the outgoing direction, we sample a microfacet normal (which is the half vector h) from the GGX distribution and reflect the view direction around to get the incident direction. It essentially sets its probability mass in the areas where the h-vector comes out close to the normal of the surface. What this does is the areas with low “roughness” reflect right back almost all of the light it receives, whereas areas with high “roughness” will be not nearly as bright, since the light scatters all over the place.

The plugin runs a 50/50 mixture:

- With probability 0.5: sample from cosine-weighted hemisphere
- With probability 0.5: sample from GGX distribution around mirror direction

The mixture sampling method combines both strategies and weights when no single method is known to match the integrand all the time. Christensen et al. (2018) in their RenderMan’s architecture paper talk about how calls like GenerateSample and EvaluateSample drive a Bxdf plugin, and how they used multiple importance sampling (MIS) to reduce variance in sampling strategies. When we combine GGX and cosine sampling we can cover the grazing angles with cosine sampling and the specular highlights in the materials like chrome.

Finally, we have speed, which is measured through per-evaluation cost and total render time, and a sampling analysis. We measure per-evaluation cost by doing bench_eval.cpp times EvalReflectance. We ran the same benchmark 5 separate times and took the

median, while keeping computer hardware and usage constant. We measure the whole render time with time_render.py. We use the ratio of time between our model and the sample models to see relatively how much slower/faster our model is compared to the sample models.

To test performance, we used a laptop powered with an Intel(R) Core Ultra 9, 288 V, 3300 Mhz, 8 Cores with 32 GB or RAM in Windows 11 with RenderMan Version 27.3. We kept resolution (768×512), lighting, compiler and flags (MSVC cl /02), build configuration (release, not debug) and the thread count passed to prman constant.

For a more detailed, step-by-step methodology, look at the Appendix section titled “Additional Information On Methodology".

4. Results

Below is the data collected for alum.bronze and chrome during network training:

Figure 7: Training Curve for Chrome Material

Figure 8: Comparison Slice Between Raw Data and NBRDF for Chrome Material

Figure 9: Training Curve for alum.bronze Material

Figure 10: Raw MERL vs MBRDF Comparison Slice

\

Table 1 - Energy Conservation Measurements

Material	Max Channel (GT)	Max Channel (NBRDF)	Passes Test?
alum-bronze	0.1981	0.1956	Yes
chrome	0.2463	0.1857	Yes

Table 2 - White-Furnace Test: Rendered vs. Reference for alum-bronze

theta_o	px R	px G	px B	ref R	ref G	ref B
5.6	0.4226	0.2915	0.1998	0.1304	0.0969	0.0688
16.9	0.1021	0.0802	0.0577	0.1265	0.0941	0.0661
28.1	0.0488	0.0380	0.0276	0.1232	0.0924	0.0643
39.4	0.0339	0.0256	0.0178	0.1262	0.0961	0.0666
50.6	0.0258	0.0194	0.0131	0.1280	0.0996	0.0714
61.9	0.0224	0.0172	0.0112	0.1302	0.1036	0.0812
73.1	0.0221	0.0170	0.0109	0.1594	0.1269	0.1129
84.4	0.0516	0.0461	0.0403	0.1341	0.1036	0.0897

RMSE per channel: R=0.13593 G=0.09698 B=0.07254   max|diff|=0.29229D

We also found that GGX-only sampling was around 18-38 times noisier than the mixture sampling across the entire range. Cosine only was around 7-11 times the noise, but collapses at grazing angles like 73 degrees and 84.4 degrees, where the noise increases to 475x and 838x relative to the mixture sampling (computed analytically) when shipped at $\alpha=0.05$ (the roughness parameter for the GGX distribution).

Figure 11: Comparison Between GT and NBRDF Models While Rendering a Sphere

This result was a render through python and demonstrates that our neural networks have sufficient accuracy when trying to replicate alum-bronze and chrome. At least in terms of accuracy, we can state with some degree of confidence that our NBRDFs are accurate on their own.

Table 3 - Per-Evaluation Cost (Median Of 5 Total Runs)

EvalReflectance	StandardToHalfDiff (ns)	EvalNetwork only
1292.96	77.38	1211.24

Table 4 - Whole Render Time (median ± MAD, 64 spp, 8 threads)

Bxdf	Time (s)	Speed Relative to PxrDiffuse
PxrDiffuse	1.594 ± 0.011	1.00
PxrSurface	2.129 ± 0.009	1.34
NBRDF	8.287 ± 0.015	5.20

4. Evaluation and Conclusion

Looking at the performance data, there are a few conclusions we can draw and evaluations we can make.

The neural network is able to represent the material accurately. The training converges cleanly in Figures 3 and 5 and we see that at around 1000 steps, error decreases at a slower rate as you increase steps. The learned slices are also accurately able to reproduce the MERL data and structure. The analytic sphere renders have an extremely low MAE and match the actual data really well. Both networks follow laws of energy conservation.

The conclusion with RenderMan is not as clean, however. Although the plugin does integrate into RenderMan, the white furnace test we did to test whether it integrated correctly is not as is expected. Near the normal incidence the sphere renders 0.42 vs the expected 0.13 and across 17 degrees to 73 degrees it renders 3-6 times too dark. This is an issue with our integration layer, and the most likely cause is that the light from the hemisphere doesn't reach the material through rays the material samples.

One evaluation takes around 1.29 μs and almost 95% of that is concentrated in the MLP. A whole render usually takes 5.2 times as long as PxrDiffuse and 3.9x and as long as PxrSurface. This demonstrates to us that at least for more simplistic renders, RenderMan's Bxdf models are faster, but not by many orders of magnitude. We also uncovered that combining sampling techniques beat the other techniques many 1-3 orders of magnitude, although this does come from analytic variance.

Comparing the model to MERL data, Each trained network is 19,212 bytes (4,803 float32 parameters), about 1,800 times smaller than the 35 megabyte MERL data it trains from, while matching it with an analytic MAE of 0.0006 (alum-bronze) and 0.0029 (chrome). This could be a big deal when expanding this method to industrial applications, where datasets could take up much more storage. In addition, the network is able to turn the discrete data in MERL into a continuous function that could compute a value for any input. There are some things to note with this however. The network approximates the table, as we know, so this data would not be exact (as seen in the MAE). Also, MERL stores doubles and our weights are in float32, but even then, the network would still be around 900 times smaller than the MERL dataset.

Note that the "compression ratio" is a conclusion found by Sztrajman et al. 's (2021) method, which we reimplemented, rather than a contribution of this work. What we contributed is showing that a network this compact can be loaded and evaluated as a native RenderMan plugin.

We have discovered that integrating a trained NBRDF into RenderMan's native architecture is feasible. We have proved that the plugin can load, render, and evaluate a network that is physically accurate and energy conserving. We found that the file is many orders of magnitude smaller than the raw MERL data it was trained on. However, it costs around 4 to 5 times an analytical material for smaller scale tests and correct light transport within a lighted scene has not been established. Importance sampling is most effective when a mixture of GGX and cosine methods are combined together, rather than just using one strategy over the other.

In the future, it is very feasible that we would be able to fix the light transport issue and correctly integrate it into RenderMan and possibly even integrate it into a rendering software like Blender.

Appendix

Related Works

In chapter 8 of Eric Veach's 1997 Stanford PhD thesis, Robust Monte Carlo Methods for Light Transport Simulation, he takes a recursive rendering equation and expands it algebraically into a sum of terms ordered by path length (such as one-bounce/two bounce terms), and extends that infinitely. Kajiya also discusses this infinite dimensional integral. He writes a formal solution to the rendering equation, where each successive term corresponds to the kth time the light has bounced before reaching the camera, each bounce living in a higher dimensional integral. The exact image is for every k from 0 to infinity. Since we obviously cannot evaluate infinitely many integrals, we have to resort to an approximation.

Burley (2012) establishes Disney principled BRDF's parameterization and the ideal design philosophy of a model. A production shading model should primarily focus on being usable for artists, not necessarily on being as physically accurate as possible, although that can be helpful. This gives us the standard for comparison for our neural network at the end, and also a standard for design: we want to make sure that an accurate measured data model is still intuitive and controllable for artists.

Matusik et al. (2003), MERL BRDF Database is the paper that explained the 100 isotropic material database, measured across incident/outgoing angle pairs, and is the standard in BRDF work like in Sztrajman et al. (2021). This is the paper and database we used to train our neural networks

Sztrajman et al. (2021) was able to encode a trained network into a more compact embedding. They found that the computational cost of the neural BRDF was not much higher than normal BRDFs.

Zeltner et al. (2024) puts a trained neural network model inside a rendering pipeline to test real time, testing different constraints like latency GPU-resident interference. This research is focused on a production offline path tracer not using top-of-the-line hardware.

Christensen et al.. (2018) describes RenderMan's architecture and how plugins and the internal structure operates.

Walter et al. (2007) describes how non-neural BRDFS work. They explain a lot of the theory behind BRDFs and are interesting to compare to NBRDF and their performances.

The main gap in these sources is that none have integrated an trained NBRDF into a production offline path tracer's architecture.

Additional Information On Rendering

Monte Carlo approximation raises another important issue. Cutting the noise, essentially how much error comes with the approximation, made from this calculation in half requires a 4x increase of the sample number. Monte Carlo estimation converges as $1/\sqrt{N}$, meaning that error decreases proportional to the square root of the sample number. In other words, trying to make a scene less noisy is not a linear solution. Veach's thesis (1997) thus

concludes this as the dominant reason as to why rendering costs are so expensive and slow. Trying to make a scene look less and less noisy requires exponentially greater sample numbers.

Modern research has pivoted towards better sampling rather than just increasing the sample count. GSCache authors say that slow rendering performance and high pixel variance (variability between pixel values) is caused by our approximations with Monte Carlo integration. There have been attempts like radiance caching, a computational technique that stores radiance values at points to efficiently estimate the lighting of a scene, that try to make sampling easier and drastically decrease the level of noise. Even then, guiding the sampling cannot decrease the noise level for more complex scenes that modern productions require. It is difficult to find just one technique that can fix all the issues with the rendering equation.

With the recent developments in neural networks, machine learning models that stack "neurons" in layers to learn pattern-recognition and biases from data to map inputs to outputs, it is theoretically possible to capture the nature of complex materials like bronze or silver while avoiding the issues with computational time and cost discussed earlier. This is, in a sense, extending beyond the idea of Monte Carlo and allowing us to create a model that can estimate the lighting of a scene based on the learning the neural network does from a dataset without having to do every physical calculation.

A neural network based BRDF (called an NBRDF) could reduce the costs and memory of each material, and can learn how to render complex materials and evaluate it in one compact forward pass, where the input data can be directly transferred into an output. Zeltner et al. (2024) demonstrate how neural representations that have encoded a material's nature are evaluated dramatically faster than hand-built multilayer renderers that do not use neural networks. Networks could also learn where it is necessary to sample, giving the network a way to generate directions proportional to the material's nature to increase the number of useful samples. This would then solve the problem of choosing better sample data so we do not have to rely on solely increasing the sample count, which could lead to more inconsistent results.

Figure 12: Log-MSE Formula Ran In Training

A typical shiny object has tiny radiance values everywhere except for highlights. Under MSE, the highlights small misses like 0.1% off would count millions of times more than the dim region being double the brightness. Log-MSE, instead, measures the relative error of the network, allowing us to ignore the errors that were relatively small and focus on the errors that actually negatively affect our accuracy.

Since our materials' most important attributes for their character are the highlights, a few percent off in the brightest regions matters a lot more than an image a few percent off in the dim areas. Thus we use a training loop that prioritizes brightness-weighted sampling,

where bright measurements are drawn more often in each batch, while adding a constant that acts like an ambient term for the dimmer regions.

Additional Information On Methodology

Figure 13: Full Path From Measured Data to Rendered Sample.

*note visuals were created with the assistance of generative AI by asking to illustrate with a visual diagram

The MERL database (Matusik et al., 2003) ships 100 isotropic materials. Each is a binary file holding a 90 × 90 × 180 grid over the Rusinkiewicz half-angle parameterization — 34,992,012 bytes, about 33 MiB. We trained one network per material on alum-bronze and chrome.

The model is a multilayer perceptron: 6 inputs, two hidden layers of 64 neurons, 3 outputs. Each hidden neuron takes a weighted sum of its inputs, adds a learned bias, and passes the result through ReLU, which replaces negatives with zero. The three output neurons are the RGB reflectance. That is 4,803 parameters in total.

The output passes through $exp(x) - 0.001$. It is exponential, because reflectance spans orders of magnitude between a dark grazing direction and a specular highlight, and because exp is positive everywhere, which enforces a non-negative BRDF without a clamp doing the work. But exp approaches zero without reaching it, so subtracting 0.001 gives the network a way to express directions where the material reflects nothing at all; the result is then clamped at zero. Training is meant to minimize log-MSE. Plain MSE would be dominated almost entirely by the few very bright specular bins, and would treat the entire dark region of the material as noise worth ignoring. Taking logs first makes a factor-of-two error cost the same whether it happens in a bright bin or a dark one, which is the right accounting when the target spans this much dynamic range.

The trained weights are written as a flat float32 blob in a fixed order: W1 (6×64), b1 (64), W2 (64×64), b2 (64), W3 (64×3), b3 (3). 4,803 floats × 4 bytes = 19,212 bytes. The plugin opens it with *fopen* in binary mode and issues one *fread* per block straight into the destination array. Because the order is fixed and the file carries no metadata, nothing has to link against a PyTorch reader, and the renderer picks up no Python dependency. If the file is missing or the wrong size, the plugin writes to stderr and falls back to a constant 0.5, so the frame completes visibly wrong rather than taking the renderer down with it.

Size Comparison:

- The MERL Table was around 35 MB

- The Trained Network was around 20 KB
- The trained network is around 1800 times smaller, about three orders of magnitude

RenderMan estimates the rendering equation by Monte Carlo: choose a direction, trace it, weight the result, repeat, millions of times per frame. A Bxdf plugin takes part through two entry points, and the correctness of the whole estimator depends on them agreeing with each other.

Figure 14: Visualization Entry Point Pipelines

*note visuals were created with the assistance of generative AI by asking to illustrate with a visual diagram

GenerateSample is the BSDF-sampling side. The plugin picks ωi itself, drawing from a proxy distribution placed where the BRDF is likely to be bright, and reports both the reflectance and the density it used. EvaluateSample is the light-sampling side: the

integrator has already chosen ωi by sampling a light source and simply asks what the material does for that direction. StandardToHalfDiff converts the pair (ω_i, ω_o) from the local shading frame into the same (θ_h, θ_d, φ_d) encoding used during training, so the network sees inputs in exactly the form it was fitted on.

Both report the same mixture density:

$$p(\omega_i) = 0.5 \cdot p_cos(\omega_i) + 0.5 \cdot p_GGX(\omega_i, \omega_o)$$

This matters more than it looks. Multiple importance sampling (Veach & Guibas, 1995) combines the two strategies by weighting each contribution according to the density the other strategy would have given that same direction. Those weights only sum correctly if both sides agree about what the density is.

Cosine-weighted hemisphere sampling draws directions with probability proportional to cos θi, matching the cosine factor already present in the rendering equation. It stops the sampler wasting effort near the horizon, where that factor would suppress the contribution anyway. GGX half-vector sampling (Walter et al., 2007) treats the surface as a field of microfacets (small bumps), each a perfect mirror. Rather than sampling the outgoing direction directly, it samples a microfacet normal, which is the half vector h, from the GGX distribution and reflects ωo about it to get ωi. The roughness parameter α sets how wide that cone is: a small α places almost every sample inside a tight lobe around the mirror direction, a large α spreads them broadly. The plugin uses α = 0.05.

Figure 15: Visualization of Importance Sampling Strategies

*note visuals were created with the assistance of generative AI by asking to illustrate with a visual diagram

Neither strategy alone suits alum-bronze. The trained network shows this material is darkest near the mirror direction and brightest toward grazing angles, the opposite of what a narrow GGX proxy assumes. Sampling GGX-only therefore places nearly every sample in the material's darkest region, which is what produced a near-black render in early tests. Cosine-only has the complementary failure: its density $\cos \theta i / \pi$ falls to zero as ωi grazes the surface, which is precisely where this material's energy is, so the ratio f/p explodes there.

The mixture is not a hedge between two mediocre options. Because its density is a sum, it is bounded below by whichever component is larger at any direction, so it never inherits either one's blow-up

Performance was measured against three criteria in a deliberate order (correctness, sampling efficiency, speed) because a render-time number means nothing if the integrated render is wrong. Correctness is a gate: it is pass/fail rather than a measurement, and it decides whether the later numbers can be interpreted at all.

Figure 16: 3 Pipelines For Measuring Performance

*note visuals were created with the assistance of generative AI by asking to illustrate with a visual diagram

Directional albedo stays below 1 for every channel and angle on both measured materials, so the trained network conserves energy. The white-furnace render does not match: against a reference computed with no renderer involved, max|diff| is 0.292 where the bar is 0.02, with the sphere rendering about 3× too bright near normal incidence and 3–6× too dark across the middle of the range. open The convergence-slope test was not run, since the sampler is already known to be biased rather than merely noisy.

The intended rendered ablation compares three plugin variants at equal sample counts. It could not be measured: a controlled ablation showed mixture, cosine-only and GGX-only producing bit-identical output (identical bin means and identical per-pixel noise) which means GenerateSample's contribution is not reaching the image under a dome light at all. That independently corroborates a defect already on record in this project. The efficiency question was answered analytically instead, by integrating the

single-sample estimator variance over the hemisphere by quadrature.

Measured as per-evaluation cost and whole-render time. bench_eval.cpp times EvalReflectance directly, with no renderer involved; the benchmark was run five times with hardware and load held constant, and the median reported. time_render.py times whole renders, and the ratio against stock Bxdfs gives the relative cost.

Results From Performance Testing:

- 1,292.96 ns per evaluation
- 77.28 ns per angle conversion
- Whole render is 5.2 time slower than PxrDiffuse
- Whole Render is 3.89 times slower than PxrSurface

The two speed figures should always be quoted together. Per evaluation the network is far more expensive than an analytic model, but a whole frame is only 5.2× slower than the cheapest stock Bxdf, because ray traversal and shading-point setup are shared overhead that the material does not pay twice. The inner loop is also scalar and not vectorized — W2[i][j] is a strided column read of a row-major matrix, so 5.2× is an upper bound with an obvious optimization still available, not a floor.

Constants: Intel Core Ultra 9 288V, 3.3 GHz, 8 cores · 32 GB RAM · Windows 11 · RenderMan 27.3 · MSVC cl /O2, release build. Resolution, lighting, pixel filter, compiler flags and build configuration were held fixed across every compared render. Thread count was pinned at -t:8 for all timing runs; the per-evaluation benchmark is

single-threaded by construction.

References

AMD GPUOpen / Kazmierczyk, K., et al. (2024). Neural supersampling and denoising for real-time path tracing. AMD GPUOpen.

https://gpuopen.com/learn/neural_supersampling_and_denoising_for_real-time_path_tracing/

Bauer, D., Wu, Q., Gadirov, H., & Ma, K.-L. (2025). GSCache: Real-time radiance caching for volume path tracing using 3D Gaussian splatting. arXiv:2507.19718.

Burley, B. (2012). Physically based shading at Disney. In ACM SIGGRAPH 2012 Course Notes.

Christensen, P. H., Fong, J., Shade, J., Wooten, W. L., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., Rayner, B., Brouillat, J., & Liani, M. (2018). RenderMan: An advanced path-tracing architecture for movie rendering. ACM Transactions on Graphics, 37(3), Article 30. https://doi.org/10.1145/3182162

Immel, D. S., Cohen, M. F., & Greenberg, D. P. (1986). A radiosity method for non-diffuse environments. ACM SIGGRAPH Computer Graphics, 20(4), 133–142. https://doi.org/10.1145/15922.15901

Kajiya, J. T. (1986). The rendering equation. In Computer Graphics (Proceedings of SIGGRAPH ‘86), 20(4), 143–150. https://doi.org/10.1145/15886.15902

Matusik, W., Pfister, H., Brand, M., & McMillan, L. (2003). A data-driven reflectance model. ACM Transactions on Graphics, 22(3), 759–769. https://doi.org/10.1145/882262.882343

Nicodemus, F. E., Richmond, J. C., Hsia, J. J., Ginsberg, I. W., & Limperis, T. (1977). Geometrical considerations and nomenclature for reflectance (NBS Monograph 160). National Bureau of Standards.

Sztrajman, A., Rainer, G., Ritschel, T., & Weyrich, T. (2021). Neural BRDF representation and importance sampling. Computer Graphics Forum, 40(6), 332–346. https://doi.org/10.1111/cgf.14335

Veach, E. (1997). Robust Monte Carlo methods for light transport simulation [Doctoral dissertation, Stanford University]. http://graphics.stanford.edu/papers/veach_thesis/

Walter, B., Marschner, S. R., Li, H., & Torrance, K. E. (2007). Microfacet models for refraction through rough surfaces. In Proceedings of the 18th Eurographics Conference on Rendering Techniques (pp. 195–206).

Zeltner, T., Rousselle, F., Weidlich, A., Clarberg, P., Novák, J., Bitterli, B., Evans, A., Davidovič, T., Kallweit, S., & Lefohn, A. (2024). Real-time neural appearance

models. ACM Transactions on Graphics, 43(5), Article 118.

https://doi.org/10.1145/3659577