Training Neural Materials For Real-Time Rendering: A Replication with INT8 Quantisation and ACF-Guided Synthesis

Suolin Zhang

MSc Computer Animation and Visual Effects
NCCA, Bournemouth University
September 2025

Table of Contents

1 INTRODUCTION

Over the past decades, real-time rendering has seen significant progress driven by advances in hardware capabilities and supporting software, greatly promoting the development of industries such as gaming and XR virtual production. Nevertheless, achieving production-quality visual accuracy in real-time rendering field still relies on emerging technologies.

Offline-rendering can achieve realistic appearance with much details yet too time cost. Conventional real-time pipelines trade accuracy for speed via simplified BRDFs, baked textures, limited shadowing, and coarse approximations of mesostructure, often losing parallax, self-occlusion, and fine anisotropy. In contrast to typical render methods, neural materials are capable of encoding complex appearance with higher byte efficiency, thereby amortizing the cost of expensive precomputation into the training phase, and deliver rapid inference that better preserves high-frequency detail than hand-crafted approximations.

Most contemporary renderers utilize 4D analytic BRDFs, combined with texture maps such as basecolour/diffuse, metallic/glossiness, and normal/height, to simulate the spatially varying distribution of microstructure and meso-structures in real-world complex materials. However, achieving photorealistic rendering results remains highly challenging for them. To faithfully reproduce the appearance of real-world materials, the 6D BTF—constructed through multi-angle measurements at numerous spatial points on a material sample—offers a more effective and reliable solution, but the data volume acquired from such measurements is often excessively large, typically reaching hundreds of GBs, which poses new demands for data compression techniques.

Recently, neural representations of BTFs have showed advantages in both quality and compression efficiency compared to classic models. However, many systems target only subsets of the problem. A comprehensive model for real-time rendering reference should integrate high-quality result, spatial variation, directional dependence, structure-preserving synthesis, higher dimension effects (parallax & silhouette) and efficient runtime evaluation, sustaining fidelity under changing view/light while controlling memory and latency budgets. It was not until a recent study [Xu et al. 2025] that all these challenges were comprehensively addressed and successfully implemented within the Falcor framework. Informed by the great direction of that, this study aims to replicate the compact neural material pipeline by:

- training a quantised model and exporting a deployable feature bundle,
- analysing ACF-guided synthesis strategy with a two-step dynamic height-field tracing procedure and applied them on trained feature planes in Falcor(Nvdia's real-time rendering framework).

2 PREVIOUS WORK

Some neural materials put efforts to train a general model capable of representing a wide range of materials, while others “use a dedicated network to fit a single material, reducing the network’s learning complexity and enabling a highly lightweight design.”[Xu et al. 2025]

NeuMIP[Kuznetsov et al. 2021] introduced neural texture pyramids and a compact MLP, including a learned offset to mimic parallax. It generalized classic mipmapping to complex appearance and achieved real-time rendering. It then be extended to represent curved geometry by adding curvature as an input and an additional alpha network to model silhouettes[Kuznetsov et al. 2022]. This improves edge fidelity on curved meshes, but evaluating both offset and alpha networks increases shader cost. Reported analyses[Xu et al. 2025] show that two such networks at 1K resolution add ~2.4 ms/px-spp overhead in a typical shader path.

Neural bi-/triple-plane methods[Fan et al. 2023, Xu et al. 2024] decompose the 6D BTF into separable 2D feature planes, concatenated and decoded by a small MLP. This improves quality and compactness over NeuMIP on material slices while keeping inference affordable; however, parallax/silhouette are usually omitted and dynamic synthesis can’t keep satisfying structures among results.

Task-specific decoders[Chen et al. 2024] demonstrate excellent real-time performance and detail in constrained domains, but they typically don’t address dynamic, structure-preserving synthesis plus parallax/silhouettes together.

There are still many related research, most of them, like we mentioned above, use neural textures to record the spatial information in neural features, which are then fed into a multilayer perceptron (MLP) to recover reflectance based on given angular conditions. But all of these previous neural material methods only focus on addressing only one or a few specific aspects of the complete appearance, often neglecting others. In contrast, the method we choose in this study has tackled these essential aspects from a more comprehensive perspective.

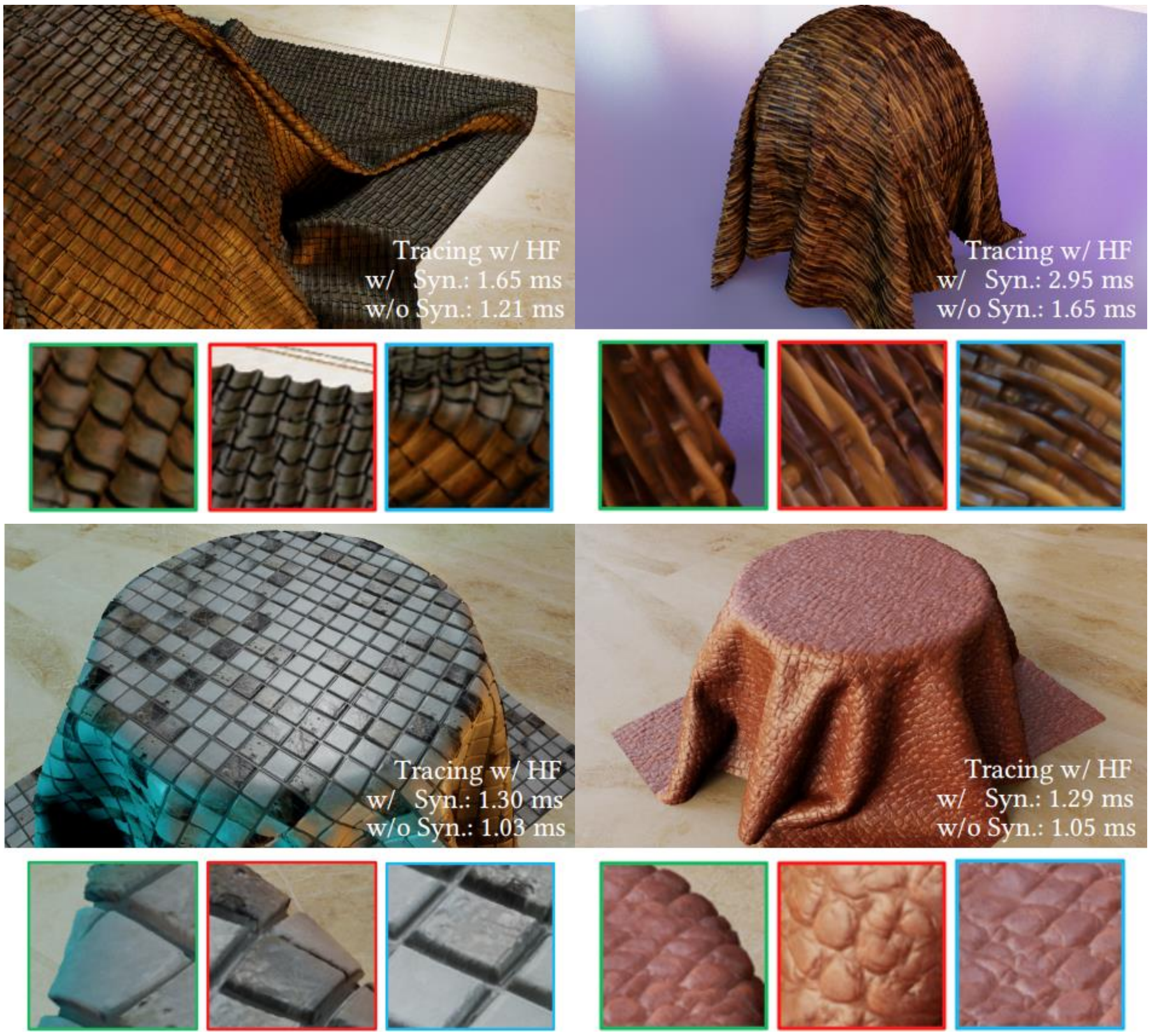

Figure. 1. Render results derived from INT8 quantised triple plane MLP inference and with ACF-guided dynamic synthesis and height filed tracing techniques applied.[Xu et al. 2025]

3 TECHNICAL BACKGROUND

In this work, the LEATHER11 from UBO2014[Weinmann et al. 2014] was selected as the target for reproduction. The BTF material data's original angles set(θi, ϕi, θo, ϕo) was extracted and re-parameterised via Rusinkiewicz parameterization to half/difference angles (θh, ϕh, θd, ϕd). Spatial coords plus new parameters are then represented by three separate learnable 2D planes U, H and D. The concatenated features are used as input to a four-layer MLP which INT8-quantised via QAT, which predicts the RGB value at the corresponding material location.

The trained model parameters are serialized and packaged for storage, and then unpacked and loaded during the inference stage. At inference, after calculating the specific surface point coordinates through ray tracing and height-field tracing, the corresponding feature values are fetched after histogram transformation and ACF-guided blending. The features are fed to a custom CUDA INT8 MLP kernel to predict reflectance, which is then combined with incident lighting to produce the final RGB contribution.

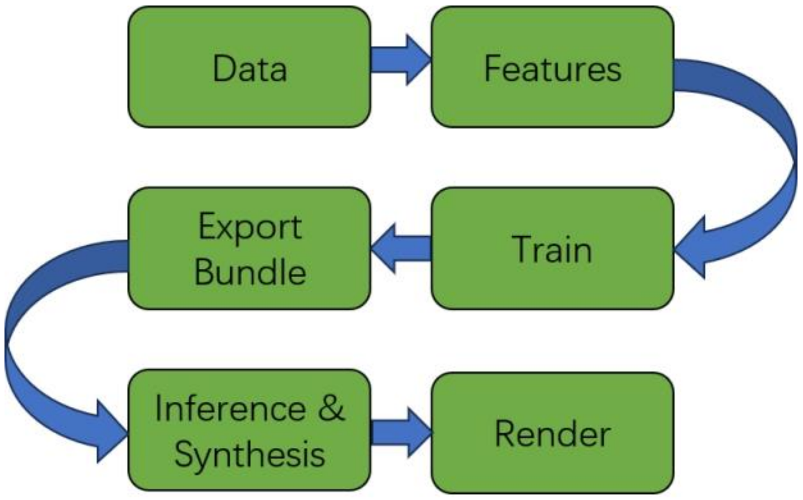

Figure 2 Overview of the neural material pipeline and data pathway.

The specific training and inference correlated methods involved will receive a detailed introduction and analysis in this chapter. Render related techniques won't be delved as they are orthogonal to our work.

3.1 Model Training

3.1.1 Rusinkiewicz Parameterisation

A BTF(Bidirectional Texture Function) is a 6D spatially varying appearance function

$$B: (u, \overrightarrow{\omega_h}, \overrightarrow{\omega_o}) \mapsto R^3, \quad u \in \Omega \subset \mathbb{R}^2, \quad \overrightarrow{\omega_h}, \overrightarrow{\omega_o} \in \mathcal{H}^2.$$

Where:

- **u** is 2D spatial coordinate on the material (UV),
- ωi is incident direction (unit vector) on the local hemisphere above the surface at u; direction light arrives from and
- ωo is outgoing/view direction (unit vector) on the same hemisphere; direction toward the camera/sensor.

It can be represented as β = β(u, v, θi, ϕi, θo, ϕo) initially, where (u,v) are the components of **u**, θi and θo are polar angles of ωi, ωo relative to the surface normal, ϕi, ϕo are azimuth angles of ωi, ωo relative to surface tangent. While with this original representation, the coefficients corresponding to terms that depend on the combined influence of both input and output angles are generally significant. This indicates that the reflective behavior of the material cannot be adequately described using only one angle independently.

The Rusinkiewicz parameterization[Rusinkiewicz 1998] re-parameterizes the BTF in terms of the halfway vector (i.e. the vector halfway between the incoming and reflected rays) and a "difference" vector, which is just the incident ray in a frame of reference in which the halfway vector is at the north pole (see Figure. 3). This change of variables represents BTF as a function of the halfangle and difference angle: β = β(u, v, θh, ϕh, θd, ϕd), where:

$$\vec{h} = \mathrm{sph}(\theta_h, \phi_h) = \frac{\overrightarrow{\omega_i} + \overrightarrow{\omega_o}}{||\overrightarrow{\omega_i} + \overrightarrow{\omega_o}||},$$

$$\vec{d} = \mathrm{sph}(\theta_d, \phi_d) = \mathrm{rot}_{\vec{b}, -\theta_h} \mathrm{rot}_{\vec{n}, -\phi_h} \overrightarrow{\omega_i}.$$

Note (θh, ϕh) are the the spherical coordinates of the halfway vector in the local sphere frame, The two rotations bring the halfangle vector to the north pole, and (θd, ϕd) are the spherical coordinates of the incident ray in this transformed frame as shown in Figure. 3. [Rusinkiewicz 1998]

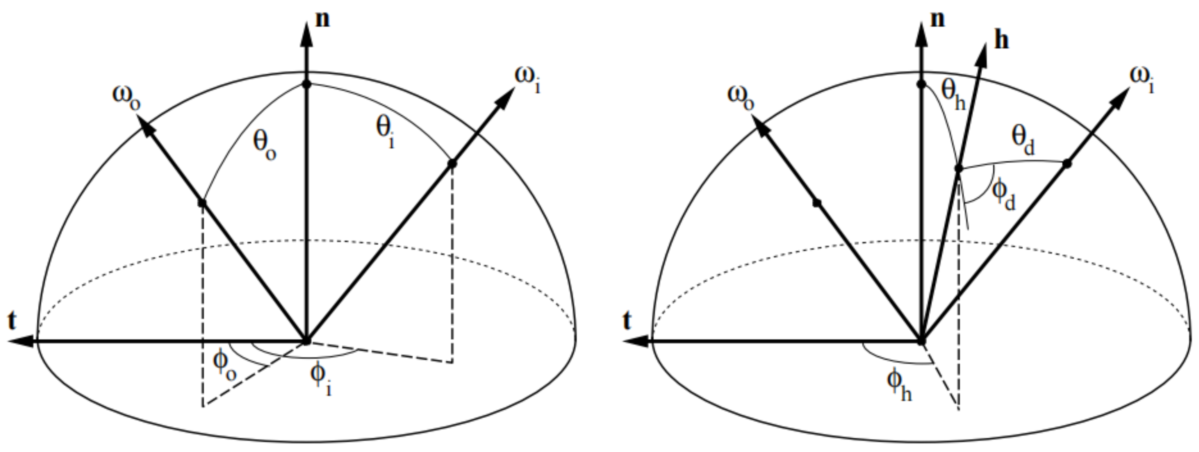

Figure. 3. The changes of axis and variables of Rusinkiewicz reparameterization.[Rusinkiewicz 1998]

After reparameterization, the BTF values exhibit significant variation in response to changes in either the half-vector or difference-vector alone, while rarely being determined by complex, entangled interactions between H and D. This characteristic establishes the foundation for introducing three learnable feature planes for model training.

3.1.2 Quantisation

Quantisation maps values from a large set to a smaller, discrete set. In neural networks this typically means reducing 32-bit floating-point weights/activations to low-precision integers (e.g., 8-bit). The motivation is practical: INT8 arithmetic and narrower tensors reduce compute and memory traffic at inference. On common GPU/CPU back ends (e.g., DP4A/TensorCore paths), INT8 matrix multiply can deliver up to an order-of-magnitude higher peak MAC throughput and ~4× lower bandwidth than FP32, while preserving accuracy when trained properly [Wu et al. 2020]. A large body of work shows that with quantisation-aware training (QAT) the quality loss from low-bit quantisation is often minimal relative to FP32 across many tasks [Gholami et al. 2022; Jacob et al. 2018; Nagel et al. 2021; Wu et al. 2020].

3.1.2.1 Uniform Symmetric Quantisation

To quantise a NN's activations and weights to a finite set of values, a function should be defined in the first place, it always shown as follows:

$$\mathrm{Q(r)} = \mathrm{Int}\left(\frac{\mathrm{r}}{\mathrm{s}}\right) - \mathrm{Z}, \qquad \hat{\mathrm{r}} = \mathrm{s(Q + Z)} \tag{1}$$

Where Q is the quantisation operator, Int() is usually a round function maps a floating number to integer, r is a real valued input which is usually the maximum of activations or weights of one layer, s is a real valued scaling, Z is an integer zero point and $\hat{\mathrm{r}}$ is the dequantised value. This method of quantisation is also known as Uniform Quantisation as the target values are uniformly spaced.

The scaling factor s scaling factor essentially divides a given range of real values *r* into a number of partitions, it can be calculated as:

$$\mathrm{s} = \frac{\beta - \alpha}{2^{\mathrm{b}} - 1}. \tag{2}$$

Here $[\alpha, \beta]$ is the bounded range used to clip r before quantisation, and *b* is the target quantisation bit width. It is popular to use a symmetric quantisation scheme which keep $\alpha = -\beta = \max(|r_{max}|,|r_{min}|)$. Using symmetric quantisation will simplifies the quantisation equation(2) to

$$Q(r) = Int\left(\frac{\mathrm{r}}{\mathrm{s}}\right) \qquad \hat{r} = \mathrm{sQ} \tag{3}$$

by replacing zero point Z to zero.

There are two choices to calculate scaling factor s for a INT-8 quantisation, to map a full range [-128, 127], s is chosen as $2max(|r|)/(2^n - 1)$, while in "restricted range" [-127, 127], s is chosen as $max(|r|)/(2^{n-1} - 1)$.

3.1.2.2 Quantisation-Aware Trainning

Implement quantisation to a trained model may introduce a perturbation to original parameters, which will push the model away from the its initial convergence point. Quantization-Aware Training (QAT) facilitates the convergence of quantized models with reduced performance degradation. During QAT, both the forward and backward passes are conducted in full floating-point precision. However, after each gradient update, the model parameters are quantized to simulate the effects of quantization during inference. In essence, QAT trains the network in the presence of quantisation effects so it learns weights/activations that are robust to rounding and clipping at inference.

An important subtlety in backpropagation is how the non-differentiable quantization operator (Eq. 3) is treated.[Gholami et al. 2022] Since the gradient of the operator(Eq. 3) with respect to *r* is zero almost everywhere, it becomes impossible to directly compute gradients relative to *r* during the backward pass of QAT. A widely adopted solution to this problem is the Straight-Through Estimator (STE), which bypasses the rounding operation during gradient propagation. Specifically, during backpropagation, the derivative of the quantized output *Q* with respect to the full-precision input *r* is approximated as:

$$\frac{\partial Q}{\partial r} \approx 1.$$

Which allows the gradient to flow through the non-differentiable quantization operator as if it were the identity function. Despite the coarse approximation of STE, it often works well in practice, except for ultra low-precision quantization such as binary quantization.[Bai et al. 2018]

3.1.3 MLP

A Multilayer Perceptron (MLP), also known as a deep feedforward network, is a fundamental type of artificial neural network. It serves as a powerful universal function approximator, capable of learning complex, non-linear mappings between high-dimensional inputs and outputs.

$$f_{\Theta}\colon R^{d_{\text{in}}} \to R^{d_{\text{out}}}$$

At its core, an MLP is composed of multiple layers of interconnected nodes (neurons), including an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to every neuron in the subsequent layer, forming a "fully-connected" architecture.

With L hidden layers, widths $\{d1,\ldots,dL\}$, parameters $\theta = \{W_l, b_l\}_{l=0}^{L}$, and activation σ(·), the computation is

$$h_0 = x \in R^{d_{\text{in}}},$$
$$h_{\mathrm{l}+1} = \sigma(W_{\mathrm{l}} h_{\mathrm{l}} + b_{\mathrm{l}}), \quad \mathrm{l} = 0, \ldots, L-1,$$
$$y = W_L h_L + b_L \in R^{d_{\text{out}}}.$$

3.2 Height-Field Tracing

A height field describes a surface as a scalar elevation h(u) over a 2D parameter domain u=(u,v). Given a camera/view ray, the displaced surface point is the location where the ray intersects the graph of this height field. Conceptually, hit finding reduces to locating the root of a one-dimensional function along the ray's footprint in texture space.

As the ray passes across the texture domain, it induces a parametric curve u(t) and an associated ray "height" z(t) (the ray's elevation within a thin shell that bounds the height field). Define

$$g(t) = z(t) - h(u(t)).$$

A surface hit occurs when $g(t^*) = 0$. Because h is sampled from an image/field, we do not solve this analytically; instead we search for t^* by probing g(t).

Modern approaches adopt a two-stage procedure that balances robustness, accuracy, and cost: coarse search (ray marching) and local refinement(contact refinement).

March along u(t) with relatively large steps and evaluate g. The goal is not precision; it is to bracket the crossing by finding two consecutive samples where g changes sign (last point "above" the surface and first point "below"). This quickly narrows the interval that contains. Once a sign change is detected, restrict attention to that small interval and refine the solution with a few inexpensive steps. Typical choices include:

- Linear interpolation between the bracketing samples,
- Bisection (halve the interval repeatedly),
- Secant-style update (use local slope from two samples).

Contact-refinement[Riccardi 2019] schemes achieve accuracy comparable to many fine uniform steps, but with far fewer total evaluations: one short coarse pass to find contact, followed by a tiny fine pass to lock onto t^*.

3.3 Autocovariance Preserving Synthesis

Autocovariance function measures how similar two locations are in a texture and it can be used as a guide(an importance sampler) during synthesis.

If an offset u has high ACF, regions separated by u in the exemplar(input sample) tend to share the same motif (ridges, bump, weave repeats). Sampling patch shifts that score high on ACF therefore preserves structure.

For a scalar guide signal ***f*(s)** over texture space ***S*** (e.g., a single "U-plane" feature), the mean and ACF are

$$\mu = \frac{1}{|S|}\int_S f(\mathbf{s})\, d\mathbf{s}, \quad \mathrm{ACF}(\mathbf{u}) = \frac{1}{|S|}\int_S (f(\mathbf{s}) - \mu)\, (f(\mathbf{s}+\mathbf{u}) - \mu)\, d\mathbf{s}.$$

By construction, ACF(0) is maximal (the variance). As ‖u‖ grows, lower ACF indicates the regions are less alike.

Synthesis repeatedly choose where to copy from. A **probability density** provides a clean, repeatable rule for these choices. Turn ACF into a probability density so it can sample offsets:

$$p(\mathbf{u}) \propto (\max\{0, \mathrm{ACF}(\mathbf{u})\}).$$

This simply says: choose offsets with probability proportional to how strongly the exemplar correlates at that offset. Negative values are clipped to keep a valid pdf. Moreover, for a PDF p(u) to be valid, it must obey the normalization constraint: $\sum p$(u) = 1, where the summation is over all possible offset values u.

Highly structured textures sometimes need a stronger bias. A simple "curve" TTT can reshape ACF before normalisation, e.g.

$$p(\mathbf{u}) \propto (\max\{0, \mathrm{ACF}(\mathbf{u})\})^{\gamma}.$$

With $\gamma > 1$ which amplifies peaks (more structure) or, with 0 < γ< 1, softens them (more variety). This "curved ACF" only changes how often each offset is picked; it does not alter the exemplar itself.

4 IMPLEMENTATION

This chapter describes the end-to-end system—from dataset processing to on-GPU inference—and explains how synthesis and dynamic displacement jointly determine the rendered appearance.

4.1 Training an INT8-Quantized Triple-Plane MLP Model

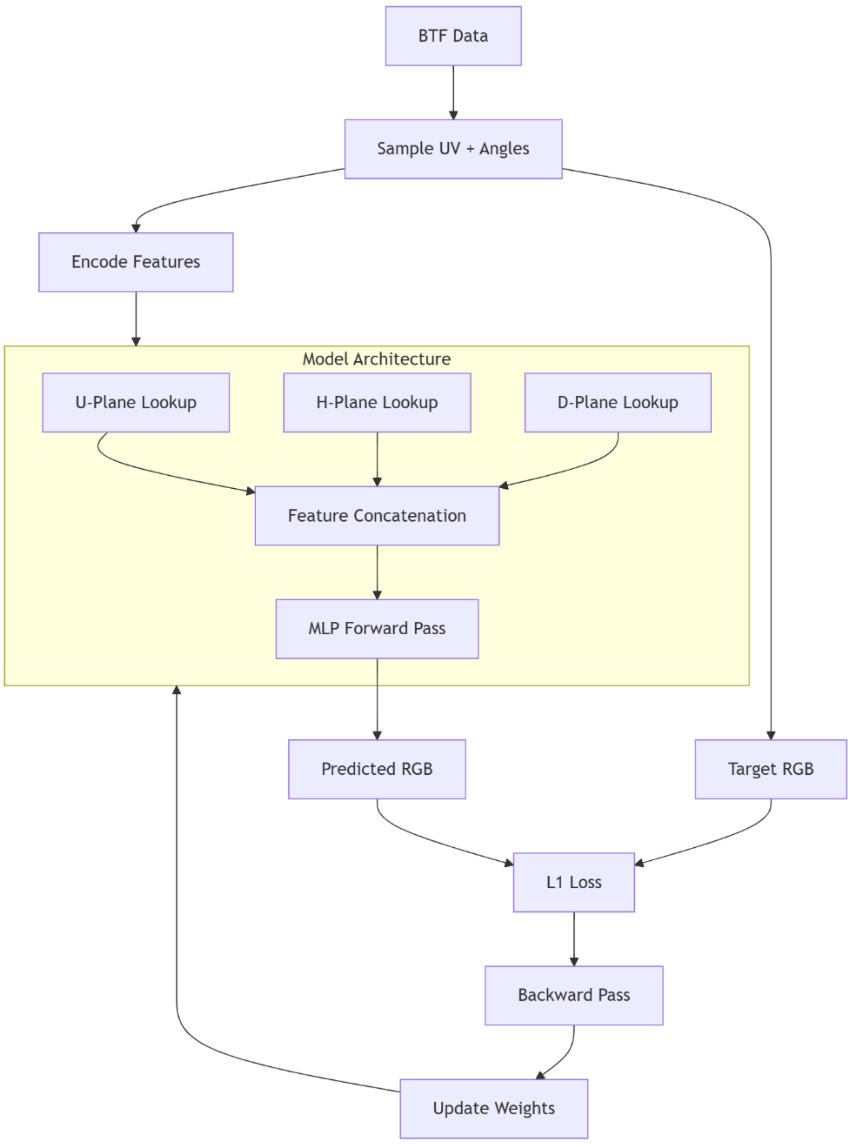

Figure. 4. Proposed training process diagram

4.1.1 Data Pipeline

We generate training pairs on the fly directly from the measured BTF. Each step randomly picks one angular configuration (θi,ϕi,θo,ϕo) from the BTF's angle list, opens (or fetches from a small per-worker cache) the corresponding image, then samples a random (u,v) in $(0,1)^2$. The pixel at (u,v) in that image (converted once to linear RGB) is the supervision y. In parallel, we convert the chosen angles to Rusinkiewicz half/difference parameters (θh,ϕh,θd,ϕd), and build the input vector x=[u, v, θh, ϕh , θd, ϕd]. This on-the-fly scheme avoids precomputing a massive dataset, naturally covers the full spatial–angular domain, and keeps I/O manageable via a tiny LRU frame cache per worker.

Validation uses a fixed list of (u, v, angles) tuples for reproducibility. The dataset does not touch the neural planes; those are sampled inside the model during the forward pass.

Algorithm 1:

```
for each batch:
  choose (θi,φi,θo,φo);frame ← cache.get_or_load()
  sample (u,v) ∈ [0,1)²; y ← bilinear(frame, u, v)
  (θh,φh,θd,φd) ← rusinkiewicz(θi,φi,θo,φo)
  x ← [u,v, θh,φh, θd,φd]
feed (x,y) to the model
```

4.1.2 Triple-Plane and Compact MLP

We factor the 6D BTF into three learnable 2D feature planes and a small decoder MLP. The planes store localized embeddings that are cheap to fetch and cache-friendly:

- U-plane U(u,v): spatial variation over the material surface.
- H-plane H(θh,ϕh): half-angle dependence (microfacet alignment).
- D-plane D(θd,ϕd): difference-angle dependence (view–light coupling).

Table. a. Size of feature planes of training model

	Height	Width	Channels
U Plane	400	400	8
H Plane	50	50	8
D Plane	50	50	8

At training time each plane is a float tensor sampled with bilinear interpolation at its native resolution. For a sample [u,v,θh,ϕh,θd,ϕd], we read U,H,D, concatenate features z=[U‖H‖D], then decode with a compact 4-layer MLP (bias-free, ReLU between linear layers). The last layer outputs linear RGB in [0,1] (clamped). Quantization hooks (QAT) wrap the linear layers but do not change the forward structure.

This design keeps most capacity in the planes (good locality, few parameters per texel) and uses the MLP only to fuse angular/spatial cues, which is fast to evaluate and easy to quantize.

Algorithm 2:

```
# Given x = [u,v, th,ph, td,pd]
Ufeat ← bilerp(U_plane,  u, v)
Hfeat ← bilerp(H_plane, th, ph)
Dfeat ← bilerp(D_plane, td, pd)
z ← concat(Ufeat, Hfeat, Dfeat)
# 4× Linear + ReLU (bias-free; with QAT wrappers enabled)
h1 ← ReLU(QLinear1(z))
h2 ← ReLU(QLinear2(h1))
h3 ← ReLU(QLinear3(h2))
rgb ← clamp01( QLinear4(h3) )   # linear RGB
return rgb
```

4.1.3 INT8 Quantization-Aware Training (QAT)

We train the decoder MLP to behave like an INT8 network during learning. Each Linear layer is wrapped with a symmetric, per-tensor fake-quant block for weights and activations:

- Symmetric INT8: zero-point Z=0; scale s estimated from running maxima (EMA of max|x|).
- Quant–dequant in train: $\hat{x} = \mathrm{dequant}(\mathrm{round}(x/s)) = s \cdot \mathrm{round}(x/s)$, passed to the next op; gradients flow through via straight-through estimator.
- Phases: start in observe (collect ranges only), then switch to quantize (apply quant–dequant) after warm-up.
- Granularity: one scale per tensor (per activation tensor, per weight matrix), matching the runtime CUDA path.
- Bias-free MLP: avoids extra rescaling terms and simplifies dequant at inference.

Training otherwise follows the standard loop (L1 loss, AdamW). Validation toggles the model to quantize to measure real INT8 behavior. At export, we freeze the collected scales and write INT8 weights plus the dequant scales.

4.1.4 Loss, Optimization, and Schedule

We train in linear RGB with an L1 loss, which is robust to outliers and aligns well with perceived error on BTF slices. The optimizer is AdamW, with two parameter groups: (i) the three feature planes (U,H,D), and (ii) the decoder MLP. The learning rate follows a cosine schedule with warm restarts, which lets the model explore early and refine near the end of each cycle.

The training uses gradient accumulation to reach an effective large batch without exceeding GPU memory, and (optionally) mixed precision for speed. Validation runs each epoch on a fixed set of (u, v, angles) tuples; when QAT is enabled, validation flips the model to quantized mode to report true INT8 behavior. All runs are seeded for reproducibility.

4.1.5 Export: Falcor-ready INT8 package

After training converges (with QAT scales stabilized), the model is serialized into a self-contained bundle that Falcor can load without Python. The export contains only data artifacts—no code.

Content includes:

- INT8 weight stream (Weight_int8_*.bin): Packed row-major matrices for the 4 MLP layers. Values are int8, arranged in 4-byte groups (4×int8 per 32-bit word) for DP4A/TensorCore paths.
- Dequantization scales (Scales_*.bin): A small float array with per-tensor symmetric scales for each layer's activations and weights, in the exact order the runtime expects (e.g. [sa0,sw0,sa1,sw1,…]). These recover real values from INT32 accumulators.
- Feature planes:
  - UPlane_*.bin: spatial plane
  - HPlane_*.bin: half-angle plane
  - DPlane_*.bin: difference-angle plane
- Plane metadata (PlaneMeta_*.bin or JSON): Dimensions for each plane (width, height, layers/channels per layer), packing stride, and axis conventions (so runtime sampling matches training).
- Material info (ModelInfo_*.bin or JSON): Hidden widths, layer counts, activation type, expected input ordering [u,v,θh,ϕh,θd,ϕd], colour space (linear RGB), and an internal format/version tag.

4.2 Inference & Rendering

This chapter explains what happens per frame—at a high level—when the trained neural material is used inside Falcor. As the core functions of code are from InstantNeuralMaterial[Xu et al. 2025], the code won't be analysed in this chapter but only the workflow will be illustrated.

4.2.1 Falcor Render Stages

Falcor uses render graph and render passes to generate a film. In this work, render pass runs in three thin stages each frame. Think of them as **trace → infer → compose**.

Each frame begins in **tracingPass**, where Falcor fires primary rays and (if enabled) refines intersections with the two-step height-field tracer. For every hit on geometry designated as "neural," the shader computes the local configuration—UVs, texture derivatives, and the incident/view directions—then maps directions to Rusinkiewicz half/difference angles. These values are compactly packed into a per-pixel buffer, a valid-pixel mask is set, and a lightweight lighting term (e.g., environment sample and visibility) is recorded for later composition.

Control then moves to **cudaInferPass**, which launches a CUDA kernel over the valid pixels. The kernel fetches features from the U/H/D planes at the packed coordinates; when synthesis is enabled it uses the curved-ACF pdf to pick and blend nearby patch offsets to preserve large-scale structure before concatenation. The concatenated feature vector is run through the INT8 MLP using packed weights and per-layer dequant/re-quant scales (dp4a dot products), producing a linear-RGB reflectance value per pixel.

Finally, **displayPass** combines this predicted reflectance with the lighting captured during tracing to form the shaded color, applies any downstream tone mapping or frame-graph post, and writes the result to the render target.

4.2.2 Height-field Tracing

Height-field tracing refines the geometric hit so parallax and silhouettes are recovered from a 2D displacement signal. At ray time, the mesh triangle is treated as a thin shell extruded along its vertex normals between a minimum and maximum height. The primary ray first intersects this shell volume to find a valid segment inside which the true displaced surface must lie. We then march coarsely along this segment in texture space using a max-filtered LoD of the height field; this acts like a lightweight, dynamic acceleration hierarchy that quickly skips empty space and brackets where the ray crosses the surface (i.e., where the ray height transitions from above to below the synthesized height)

Once a coarse cell containing the crossing is identified, tracing switches to a fine contact-refinement phase. Starting from the beginning of that coarse cell, the algorithm takes a small, fixed number of fine steps (e.g., up to 32), sampling the height each step and detecting the sign change between consecutive samples. A short linear interpolation between the last two samples estimates the precise intersection t along the ray and the corresponding height. This yields a refined hit position in world space and, crucially, an updated set of UV coordinates and local frame for shading. Because the height field can depend on view derivatives, the method uses the pixel's ddx/ddy to pick the appropriate LoD during both coarse and fine passes, reducing aliasing and stabilizing motion.

The output of height-field tracing replaces the naïve barycentric hit: UVs, normal frame, and depth now reflect the displaced geometry. Those refined UVs are what the next stage (ACF-guided feature synthesis) uses to fetch features, so the apparent micro-relief and silhouette continuity propagate directly into the neural material's reflectance

4.2.3 ACF-guided Synthesis

ACF-guided synthesis preserves large-scale structure when fetching neural features. After height-field tracing refines the hit, the kernel samples the U/H/D planes not just at the raw UV, but through a patch-shift selection driven by the material's autocovariance function (ACF). Intuitively, the ACF tells how well a motif matches itself at an offset; high values mean "same pattern if shifted by this vector." We convert the normalized ACF to a sampling pdf (optionally shaped by a simple curve, e.g., a Bézier "curved ACF") so offsets with stronger self-similarity are chosen more often.

At runtime, for each valid pixel we draw one or two offsets from this pdf and read features at the corresponding shifted coordinates (e.g., $u_a=u+\Delta_a$, $u_b=u+\Delta_b$). A dual-tiling blend then mixes the two samples, which reduces tiling repetition, avoids hard seams, and keeps repetitive structure (weaves, ridges) aligned across tiles. The same offsets are applied consistently to U (spatial), while H/D (angular) sample at the original half/diff angles so view/light dependence remains intact. The result is a synthesized feature vector that preserves the exemplar's long-range layout yet remains temporally stable, and this vector is what proceeds into the INT8 MLP for reflectance prediction.

4.2.4 INT8 Cuda Inference

INT8 CUDA inference evaluates the compact MLP per pixel using integer math for speed and bandwidth. After height-field tracing and ACF synthesis, each valid pixel has a small input record (UV/angles already reduced to feature vectors). The CUDA kernel launches one thread per pixel (or a small group), reads the synthesized U/H/D features, and concatenates them into a single vector in registers.

Network weights are pre-packed: four int8 values are stored in one 32-bit word so the kernel can use the GPU's dp4a instruction (int8×int8→int32) for fast dot products. At block start, the kernel copies the current layer's weight tiles from global memory into shared memory to amortize latency, while the per-layer dequantization scales (symmetric, per-tensor) live in constant memory or registers.

The forward pass proceeds layer by layer. Before each matmul, the activation vector is quantized on the fly to int8 using the stored scale; the matmul accumulates into int32 (one accumulator per output channel). After the dot product, results are dequantized back to float (or fp16), the ReLU is applied, and—except for the final layer—the activations are re-quantized to int8 for the next layer. This quantize → dp4a → dequant pattern keeps traffic and compute low while matching the behavior learned during QAT. The final layer dequantizes to floating-point RGB in [0,1][0,1][0,1], which is written to the reflectance buffer.

A few practical details enable throughput: coalesced reads of packed weights, fixed output widths to avoid divergent control flow, and small, consistent per-thread workloads so occupancy stays high. Numerically, clamping and scale guards prevent overflow, ensuring the INT8 path closely tracks the QAT-trained model. The kernel returns one linear-RGB reflectance per valid pixel; the render pass then multiplies it by the stored lighting to form the final shaded color.

5 RESULT

This chapter reports training and runtime outcomes for the UBO2014 – LEATHER11 material.

5.1 Training Phase

In this work, the training was conducted on a single NVDIA RTX 4080 GPU with 64 GB of CPU RAM. The batch size for each iteration was 2,560,000, and the training time per epoch was approximately 15 minutes. The total training time for 300 epochs was about 75 hours. Without QAT quantization, with both weights and activation values maintained in float32 precision, the L1 loss after 300 epochs was approximately 0.02. With QAT quantization applied, where both weights and activation values were quantized to INT8, the L1 loss after 300 epochs was approximately 0.023, resulting in an increase in error of about 15%.

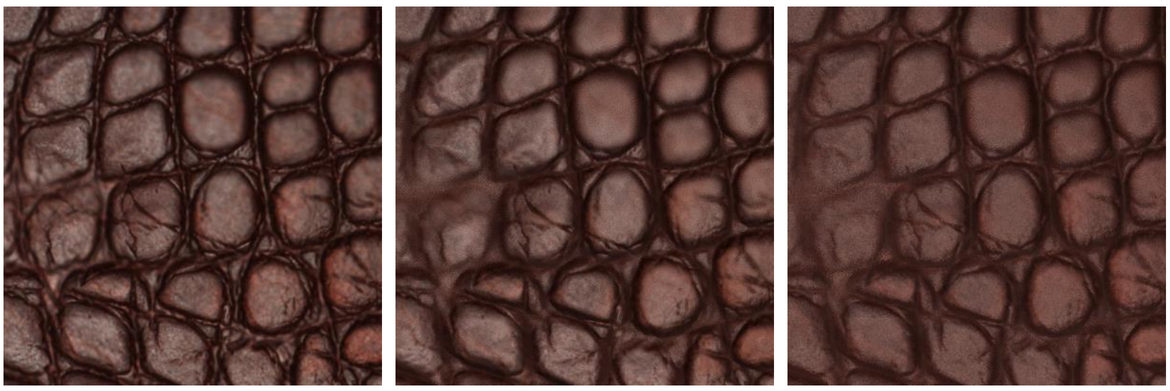

Figure. 5. From left to right are the ground truth, the training result with float32, and the training result with int8.

This work also conducted separate QAT int8 training on the resampled LEATHER11 sample from UBO2014. The L1 loss of this sample reached a remarkably high value of approximately 0.38 after 300 epochs. The exported results of this sample were subsequently used for inference in Falcor.

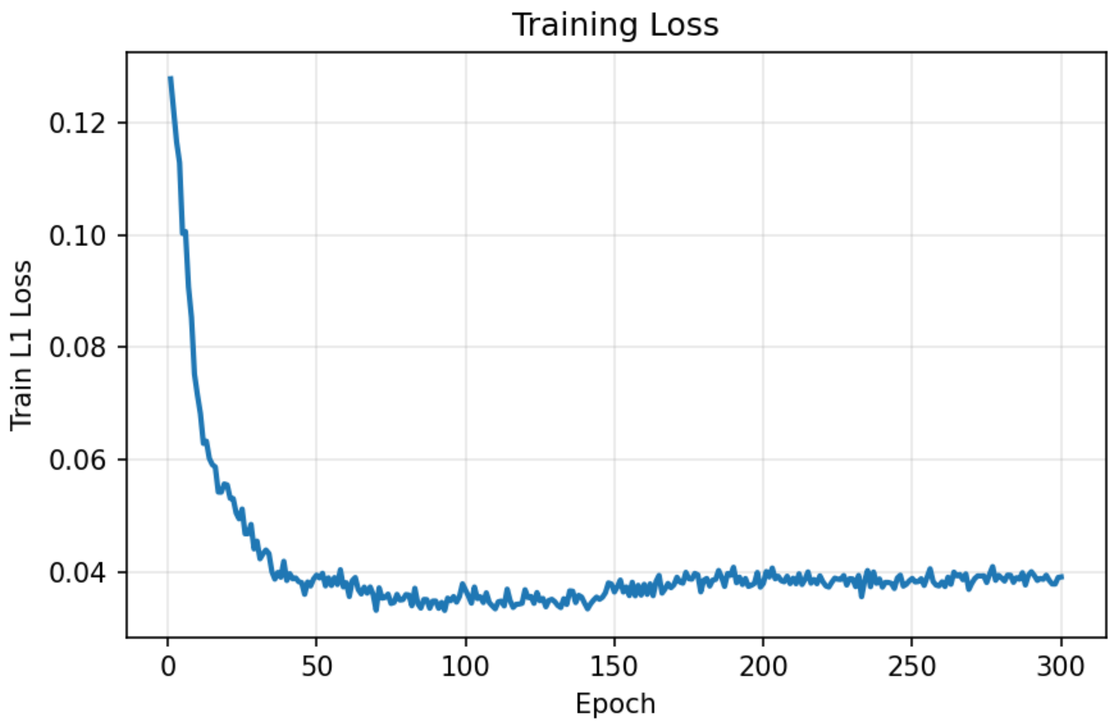

Figure. 6. The training loss of resampled UBO2014 LEATHER11

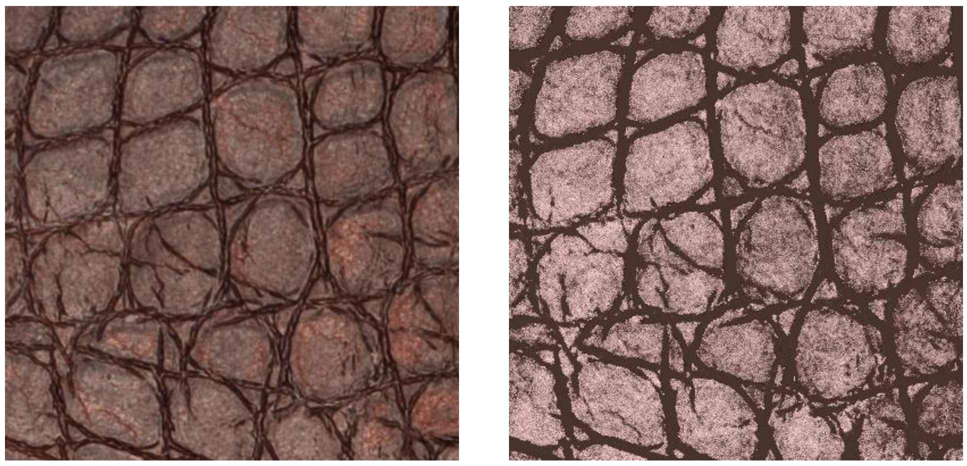

Figure. 7. From left to right are ground truth and training result of resampled UBO2014 LEATHER11

This work also visualised the feature maps of the U, H, and D planes exported from the training of the resampled Leather11 sample. From the visualisation results, the U-plane exhibits a high degree of alignment with the material's spatial structure, demonstrating the effectiveness of the triple plane configuration.

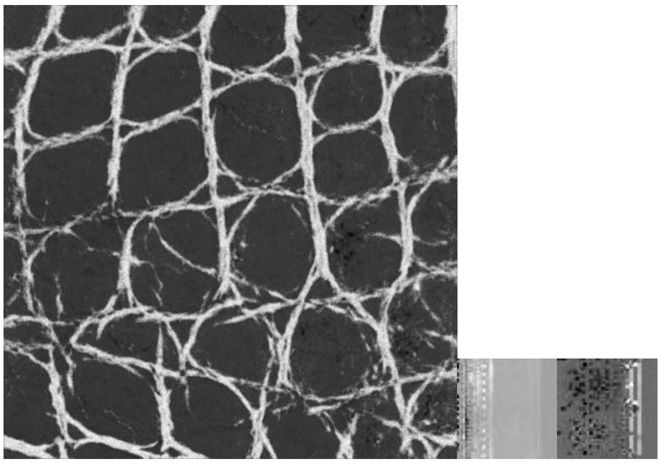

Figure. 8. Visualisation of U, H and D planes exported from UBO2014 LEATHER11 INT8 quantised training

5.2 Running Phase

The inference and rendering code in Falcor was successfully adapted and executed based on the working in progress version code uploaded to GitHub by the researchers of the referenced work[Xu et al. 2025]. Although the customized neural material-related code could be successfully built and executed after modifications, and our U-plane has demonstrated its ability to learn the spatial distribution characteristics of the material, unfortunately, the dynamic synthesis, while enabled, was not practically applied. The ACF values remained zero across all texture spaces. As a result, although the exported data was loaded and rendered, the rendered material did not exhibit any spatial structure resembling Leather11.

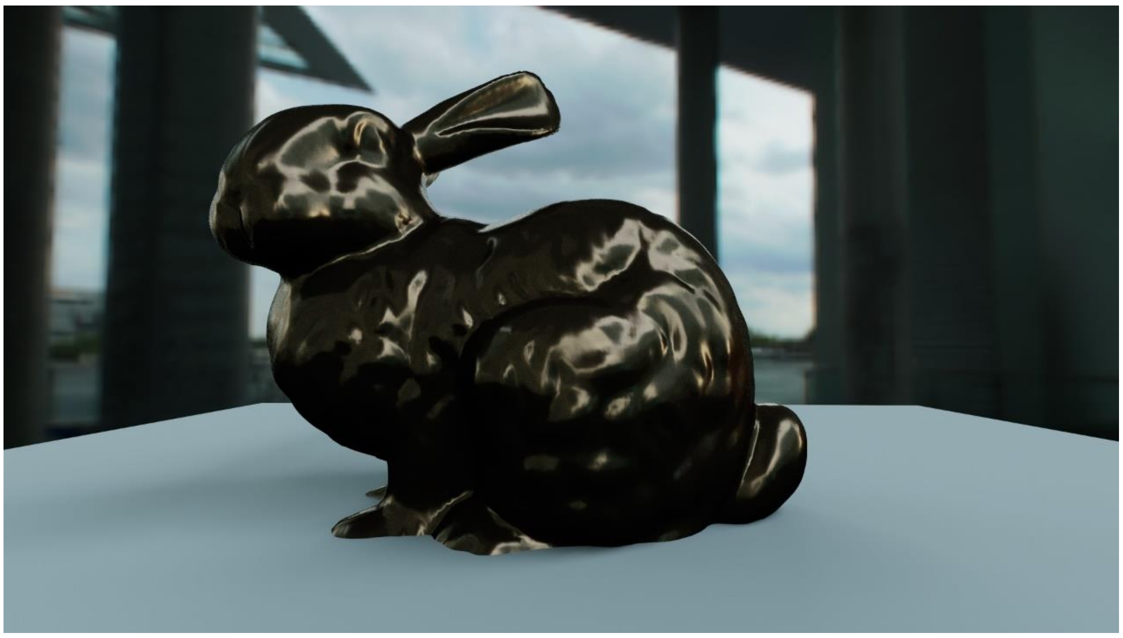

Figure. 9. The neural material render result from Falcor shows the dynamic synthesis failed

In addition, the current build of Falcor occasionally crashes when Mogwai (Falcor's rendering client) attempts to load neural materials. The inference speed is also unstable, fluctuating between 4 and 8 milliseconds during testing on a single RTX 4080 GPU. Due to time constraints, these issues remain unresolved and will need to be addressed in the future.

5.3 Evaluation

Due to the short project timeline (two months) and the extensive time required for each training session (75 hours for 300 epochs), this work did not allow for sufficient optimization to improve the training process. The reference study completed 300 epochs in just 18 hours on a single RTX 4090, using only about a quarter of the time required in our case. Without access to a 4090 for comparative experiments, it remains unclear whether this significant discrepancy stems from hardware performance differences or insufficient model optimization. According to the figures in their appendix, the L1 loss after 300 epochs was approximately 0.017. However, since the specific model details were not provided, it is difficult to evaluate the convergence performance on the LEATHER11 material.

At this stage, the value of our training phase lies in establishing a new baseline for INT8 QAT MLP training on the RTX 4080 for specific materials, which offers certain reference significance.

For the inference phase, the reference work generally achieved inference speeds of only 1–2 ms on an RTX 4090. Similarly, due to hardware disparities, a quantitative comparison with our results is not feasible. Nevertheless, it is evident that the inference and rendering component of our work is not yet complete and requires further refinement of the related code.

6 CONCLUSION

This project set out to train a compact, INT8-quantized triple-plane neural material and to deploy it in Falcor with structure-preserving synthesis and dynamic height-field tracing. The work achieved a complete training pipeline (on-the-fly BTF sampling, Rusinkiewicz encoding, U/H/D feature planes, bias-free 4-layer MLP, symmetric per-tensor QAT) and produced a Falcor-ready bundle (INT8 weights, planes, scales). On the primary LEATHER11 run, training converged to L1 ≈ 0.020 (FP32) and ≈ 0.023 (INT8, QAT)—a modest ~15% delta that is in line with expectations for per-tensor symmetric INT8. Feature-plane visualization confirmed that the U-plane learns spatial structure, supporting the chosen factorization.

On the deployment side, the Falcor pass graph was adapted and the CUDA inference path executed with packed weights and dequant scales. However, two shortcomings limited end-to-end outcome:

- Structure-preserving synthesis did not take effect at runtime—the ACF evaluated to zero everywhere—so renders lacked the intended spatial motif and
- stability/performance issues remained (occasional Mogwai crashes; 4–8 ms/frame variability on RTX 4080).

Overall, the project partially met its objectives: it delivered a reproducible INT8 training recipe and a deployable bundle, but fell short of demonstrating the full ACF-guided synthesis + dynamic tracing pipeline in Falcor. Still, the work establishes a baseline on RTX 4080 for INT8 QAT with triple planes, and documents the critical interfaces between training export and CUDA inference.

Future development should prioritize the following directions:

1) Debug and Activate ACF Synthesis: The highest priority is to diagnose why the analytically computed ACF (anisotropic correlation function) remains zero at runtime. This likely involves verifying the correct binding

and sampling of the exported feature planes within the Falcor shader, ensuring the ACF computation logic is correctly implemented, and validating the data flow between the CPU-side resource management and the GPU-side inference kernels.

2) Performance Profiling and Stabilization: A systematic profiling of the Falcor application is necessary to identify the root causes of instability (crashes) and inference time variability. This includes checking for memory access violations, resource contention, and potential bottlenecks in the custom CUDA kernels
3) Expanded Validation: Future work should include training and deployment on a broader set of materials not only from the UBO2014 dataset but also synthesised materials which contains multiple structures. This would help validate the generalizability of the proposed pipeline and identify any material-specific challenges in the quantization or deployment process.
4) Exploration for Real-time Rendering Engines: A crucial next step is to explore the integration and performance of this neural material pipeline within mainstream real-time rendering engines (e.g., Unity or Unreal Engine). This exploration should assess the practicality of deploying the INT8 quantized model, its overhead within a complex game/application environment, and the necessary adaptations to fit into standard engine rendering subroutines.

In summary, the project delivers a working INT8 neural-material training/export stack and a partially integrated real-time renderer. While the final visual demonstration lacks the intended structure due to synthesis issues, the path to closure is clear and actionable, and the artifacts produced (training scripts, bundle format, Falcor pass integration) provide a solid foundation for completing and extending the system.

7 REFERENCES

Zilin Xu, Xiang Chen, Chen Liu, Beibei Wang, Lu Wang, Zahra Montazeri, and Ling-Qi Yan. 2025. Towards Comprehensive Neural Materials: Dynamic Structure-Preserving Synthesis with Accurate Silhouette at Instant Inference Speed. In Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers (SIGGRAPH Conference Papers '25). Association for Computing Machinery, New York, NY, USA, Article 161, 1–11. https://doi.org/10.1145/3721238.3730626

Alexandr Kuznetsov, Krishna Mullia, Zexiang Xu, Miloš Hašan, and Ravi Ramamoorthi. 2021. NeuMIP: Multi-Resolution Neural Materials. ACM Trans. Graph. 40, 4, Article 175 (jul 2021), 13 pages. https://doi.org/10.1145/3450626.3459795

Alexandr Kuznetsov, Xuezheng Wang, Krishna Mullia, Fujun Luan, Zexiang Xu, Milos Hasan, and Ravi Ramamoorthi. 2022. Rendering Neural Materials on Curved Surfaces. In ACM SIGGRAPH 2022 Conference Proceedings (Vancouver, BC, Canada) (SIGGRAPH'22). Association for Computing Machinery, New York, NY, USA, Article 9, 9 pages. https://doi.org/10.1145/3528233.3530721

Jiahui Fan, Beibei Wang, Miloš Hašan, Jian Yang, and Ling-Qi Yan. 2023. Neural Biplane Representation for BTF Rendering and Acquisition. In Proceedings of SIGGRAPH 2023.

Zilin Xu, Zahra Montazeri, Beibei Wang, and Ling-Qi Yan. 2024. A Dynamic By-example BTF Synthesis Scheme. In SIGGRAPH Asia 2024 Conference Papers (SA '24). Association for Computing Machinery, N ew York, NY, USA, Article 132, 10 pages. https://doi.org/10.1145/3680528.3687578

Xiang Chen, Lu Wang, and Beibei Wang. 2024. Real-time Neural Woven Fabric Rendering. In Proceedings of SIGGRAPH 2024.

Michael Weinmann, Juergen Gall, and Reinhard Klein. 2014. Material Classification Based on Training Data Synthesized Using a BTF Database. In Computer Vision – ECCV 2014, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, 156–171.

Szymon M. Rusinkiewicz. 1998. A New Change of Variables for Efficient BRDF Representation. In Rendering Techniques '98, George Drettakis and Nelson Max (Eds.).Springer Vienna, Vienna, 11–22.

Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602* (2020).

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, 291–326.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

Yu Bai, Yu-Xiang Wang, and Edo Liberty. Prox-quant: Quantized neural networks via proximal operators. *arXiv preprint arXiv:1810.00861*, 2018.

Andrea Riccardi. 2019. A new approach for Parallax Mapping: presenting the Contact Refinement Parallax Mapping Technique. https: /www.artstation.com/blogs/andreariccardi/3VPo/a-new-approach-for-parallax mapping-presenting-the-contact-refinement-parallax-mapping-technique