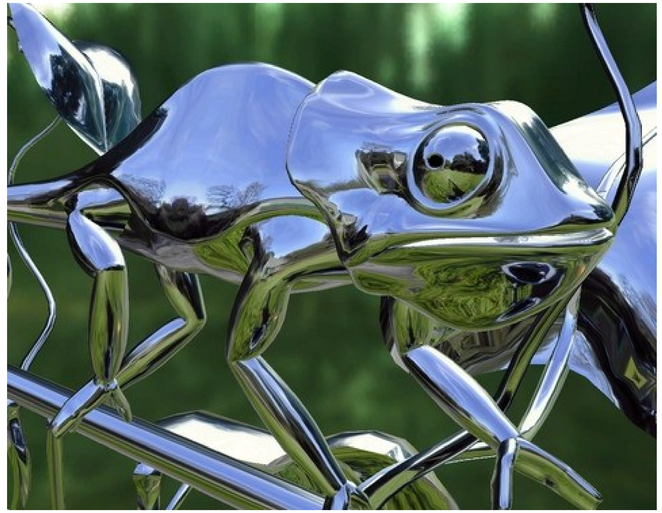
Neural Shading for Real-Time Graphics

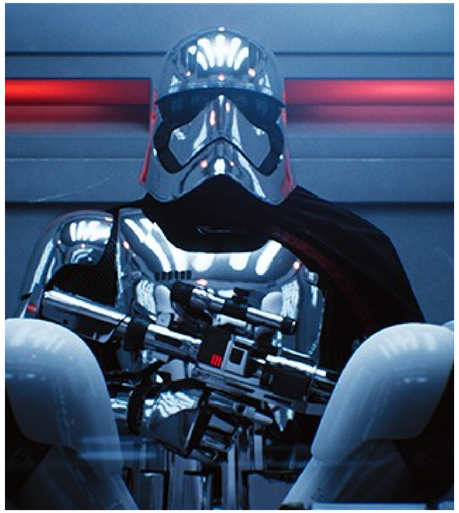
Andrew Allan, Devtech Engineer | Graphics Programming Conference

Introduction

Why Neural Shading?

- Real-time rendering is about approximating reality with the highest fidelity possible within 16 or 33 milliseconds per frame.
- This pursuit has driven over 40 years of increasingly complex graphics pipelines and shader code.



What is Neural Shading?

- Neural Shading integrates machine learning into the real-time rendering pipeline, replacing or augmenting traditional shading functions with learned neural networks.

What is Neural Shading

- Utilises a neural network
- Anything that is trainable

What is Neural Shading

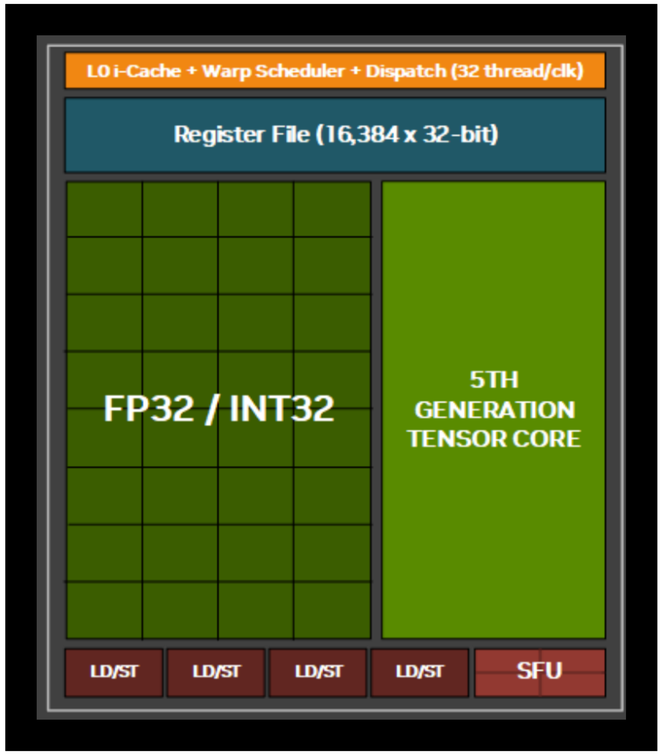
- Utilises a neural network
- Anything that is trainable

Neural Shading

- Runs in the graphics pipeline
- Part of the normal shading code

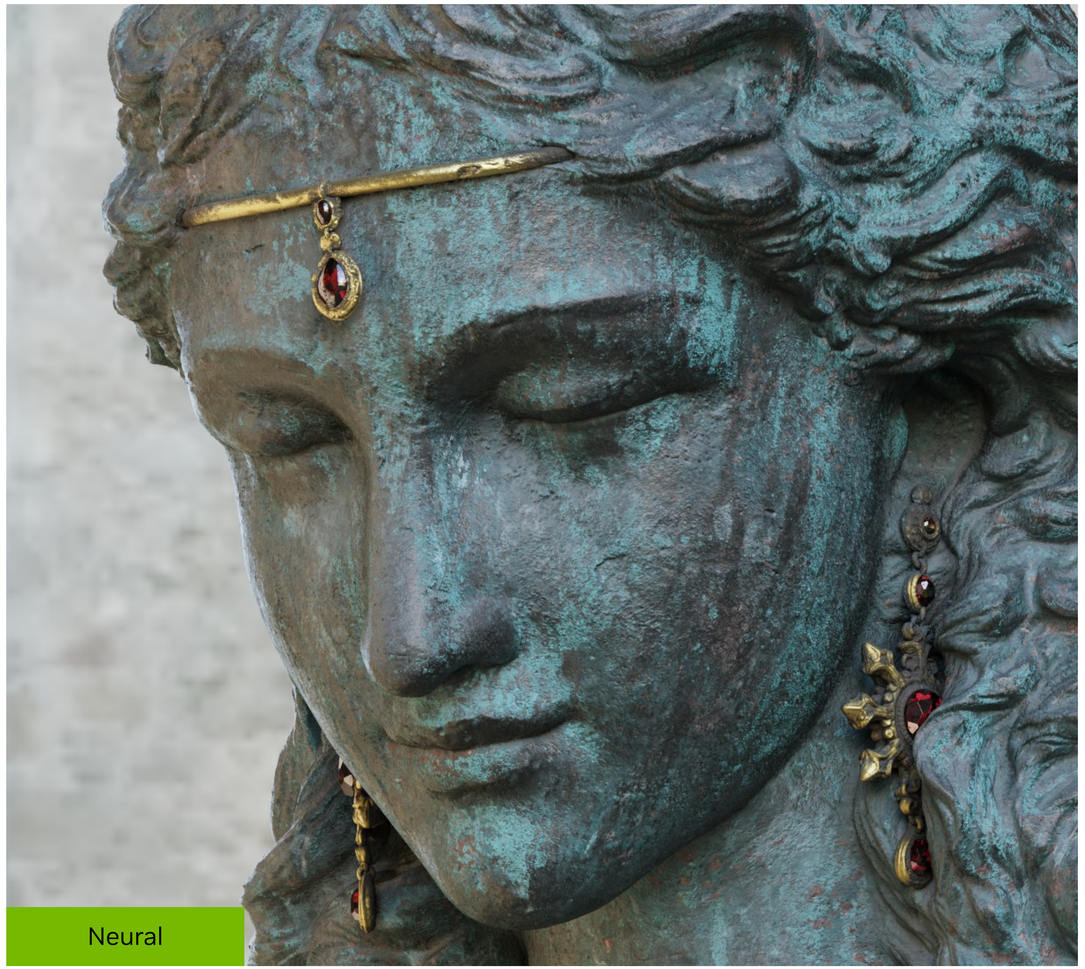
Harnessing Neural Hardware

- Modern consumer GPUs include neural network accelerators that remain idle during traditional rendering.
- Neural Shading allows us to harness these accelerators through Cooperative Vectors, integrating neural computation directly into the graphics pipeline.



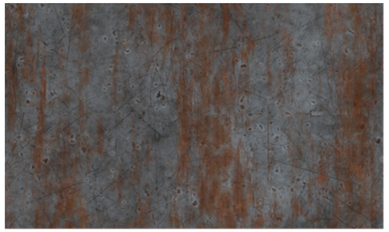
Classical vs. Neural Shading

- Real-time rendering has always relied on approximate mathematical solutions to simulate physically accurate effects.
- Classical engineering methods depend on these analytical models, but many shading problems are too complex or costly to express accurately.
- Neural Shading learns these complex relationships directly from data, bypassing the need for explicit analytical solutions.

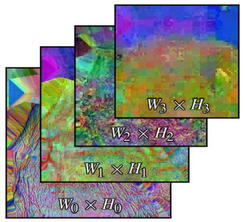


Research

Compression



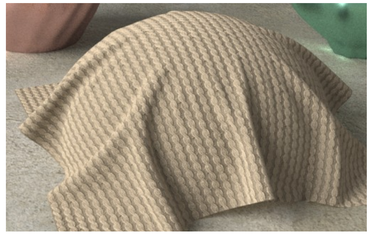
[Fujieda and Harada, 2024]



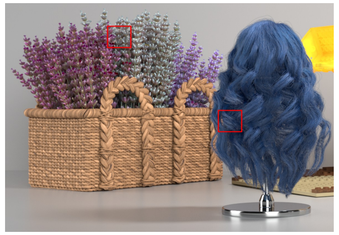
[Belcour and Benyoub, 2025]

[Vaidyanathan et al., 2023]

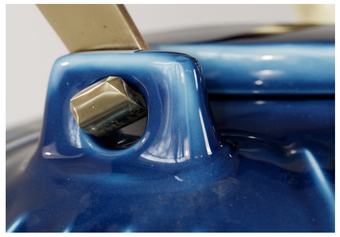
Materials



[Kuznetsov et al., 2021]



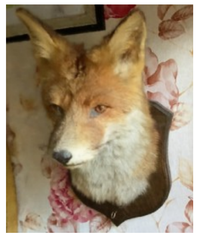
[Mullia et al., 2024]



[Zeltner et al., 2024]

Geometry

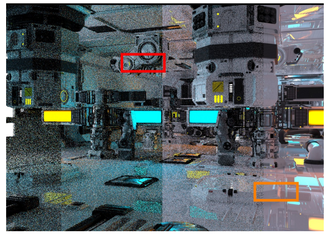
[Mildenhall et al., 2020]



[Müller et al., 2022]

[Kerbl et al., 2023]

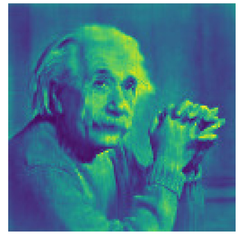
Caching



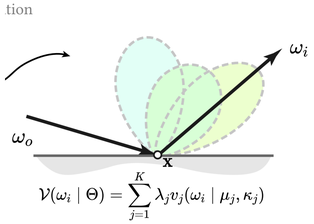
[Müller et al., 2021]

[Dereviannykh et al., 2024]

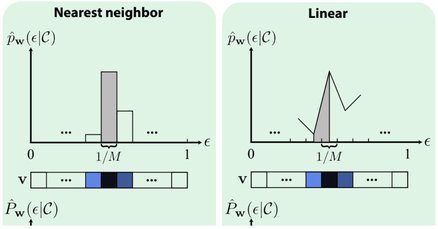
Guiding



[Müller et al., 2019]



[Dong et al., 2023]



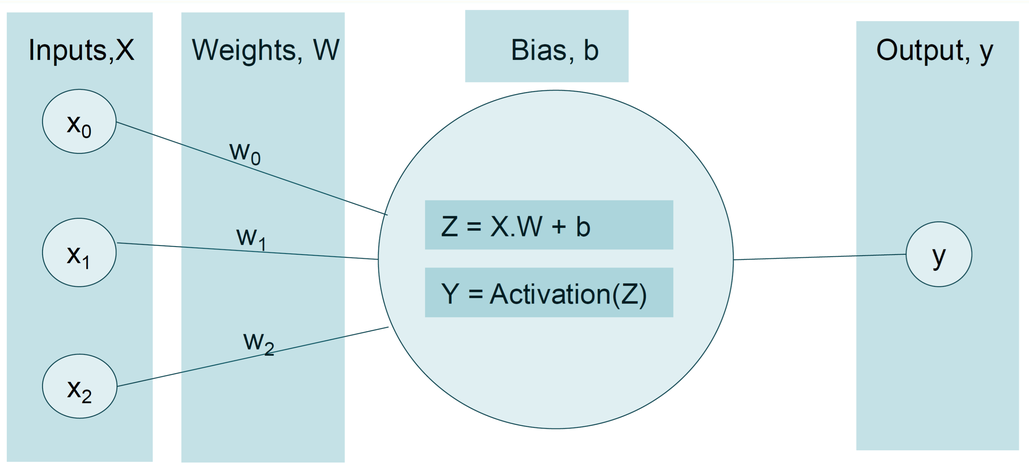
[Figueiredo et al., 2025]

Core Concepts

Learned Function

Multilayer Perceptron

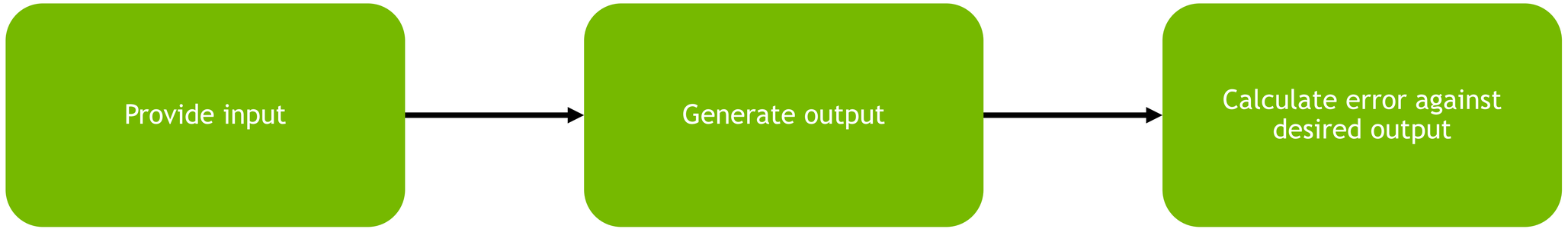
- We model the learned shading functions using small Multilayer Perceptron (MLP) networks.
  - An MLP is composed of many interconnected neurons, each performing a weighted sum of its inputs followed by a nonlinear activation.
  - A network is structured with an input layer, one or more hidden layers and an output layer.
- Each network is trained during prior to rendering to approximate reference shading data.
- During rendering, shaders execute MLP inference on the GPU to evaluate shading results in real time.



Training

Forward Phase

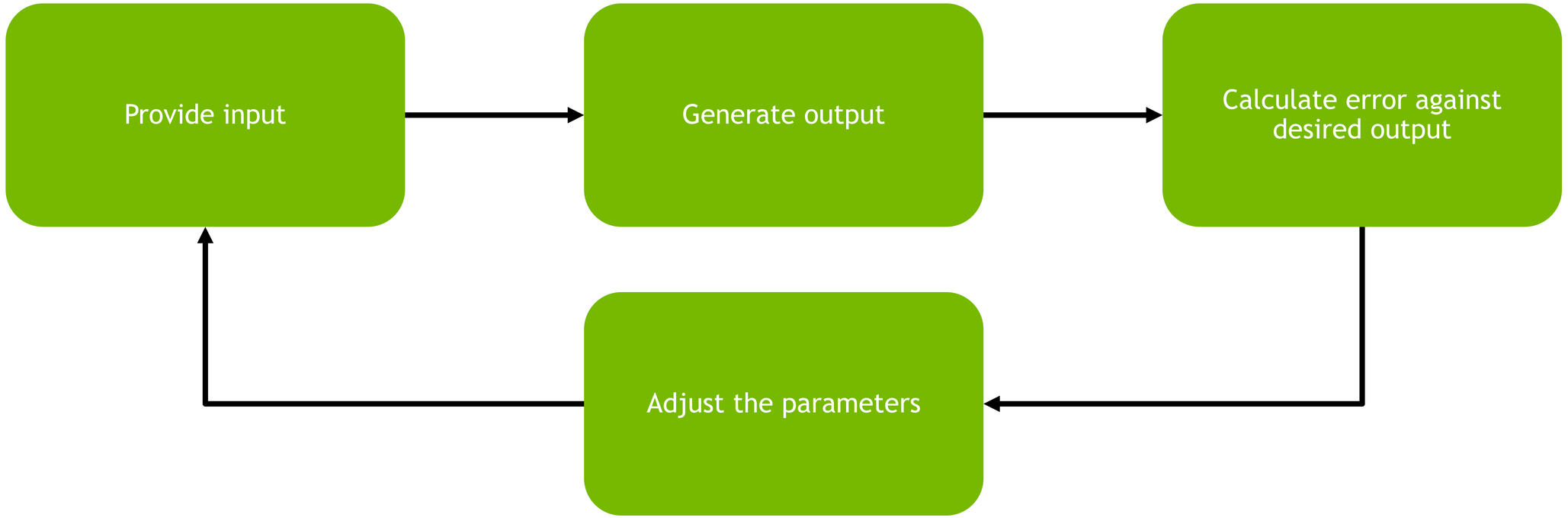
- During forward phase of training, the network takes inputs and produces a predicted output. This result is compared to the desired output to calculate an error, providing a measure of how closely the model reproduces the target shading or visual appearance.



Training

Backwards Phase

- The calculated error is backpropagated through the network to compute gradients, which update the network's parameters to better match the target shading.



Optimizer

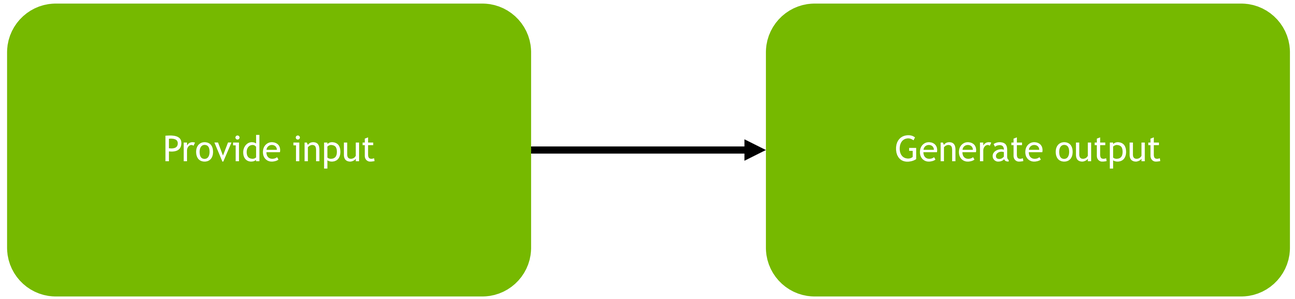
Backwards Phase

- Once gradients are calculated, an optimizer uses them to adjust the network's weights and biases in order to minimize the loss.
- The simplest form is Stochastic Gradient Descent (SGD), which updates each weight by subtracting the gradient scaled by a learning rate.
- Adam improves on SGD by adapting the learning rate for each weight using momentum and gradient history, leading to faster and more stable convergence.

Inference

Forward Phase

- During inference, the trained network takes the inputs and produces a final shading output directly, using the parameters learned during training.



First Neural Shader

Tools

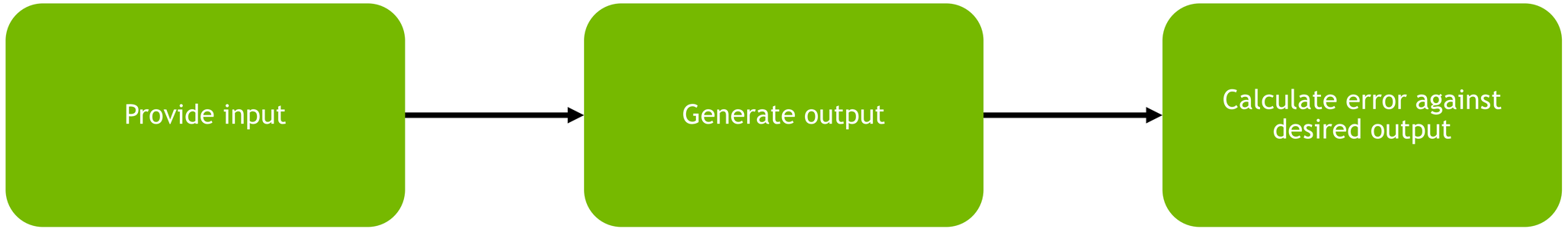
- Powerful / flexible shading language
- Write once / run anywhere
  - Compiles to SPIR-V and DXIL
- Generics
- Supports ‘auto-diff’ (does calculus for us)
  - Very useful for developing

- Python/C++ interface to Slang
- Full featured graphics api
- Cross platform
- Functional api to directly call Slang functions from Python

MLPs in Shaders

Forward Pass

- Let’s use our simplified model to train a network that generates the pixels of a texture




```
float2 uvCoordinates;
```

```
float3 predictedColor;
```

```
float3 error = predictedColor
  - actualColor;
```

MLPs in Shaders

Shader Code

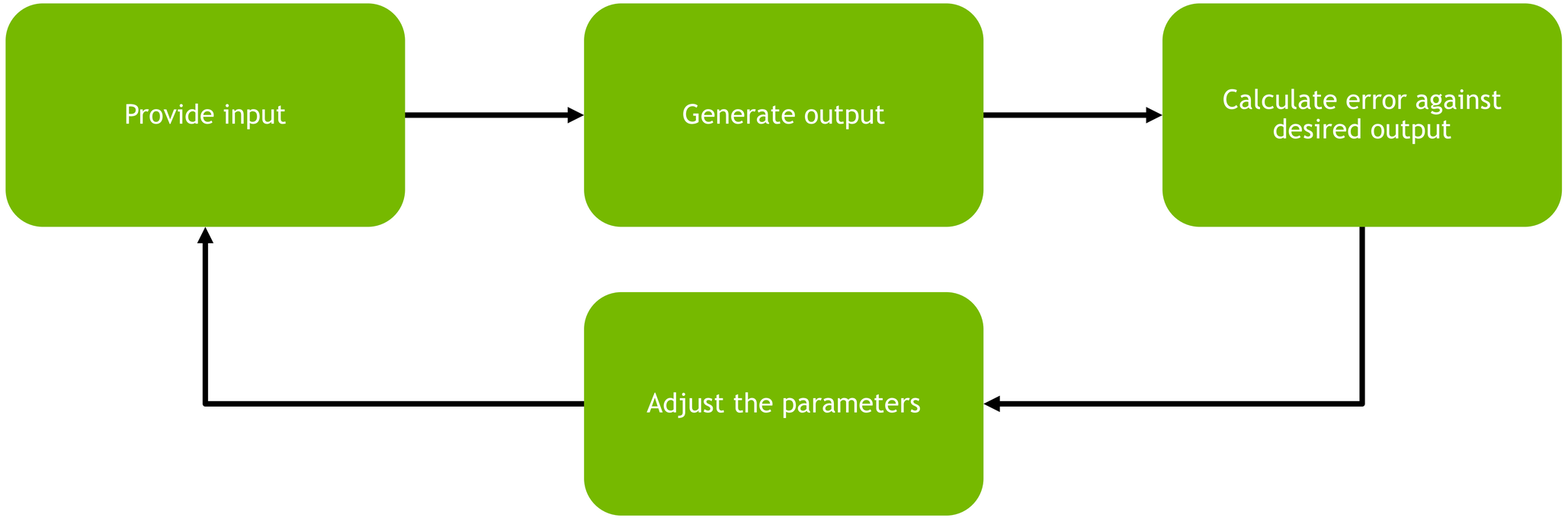
- In shader code, MLPs are implemented directly within the regular shading stage:

```
float3 loss(uint2 uvCoordinates, float3 actualColor)
{
	// Generate output
	float3 predictedColor = forwardPass(uvCoordinates);

	// Calculate error
	float3 error = predictedColor - actualColor;
	return error * error; // squared error
}
```

MLPs in Shaders

Backwards Pass

- Now we must close the training loop by backpropagating the error through the network to generate gradients and adjust the network parameters accordingly.



Training MLPs in Shaders

Differentiation

Inference

- We have our loss function:

```
float3 loss(uint2 uvCoordinates, float3 actualColor)
{
    // Generate output
    float3 predictedColor = forwardPass(uvCoordinates);

    // Calculate error
    float3 error = predictedColor - actualColor;
    return error * error; // squared error
}
```

Backwards

- How do we differentiate the loss function?
- In HLSL we will need to manually derive it.
- But with Slang, we can let the compiler derive it!

```
bwd_diff(loss)( /* ... */ );
```

- This saves a great deal of time and effort during the experimentation stage of training a neural network

Training MLPs in Shaders

Gradients

- We can now derive the gradients

```
void calculateGradients(uint2 uvCoordinates)
{
	// Generate output
	float3 predictedColor = forwardPass(uvCoordinates);

	// Get desired output
	float3 actualColor = inputTexture[uvCoordinates].rgb;

	// Calculate error
	bwd_diff(loss)(uvCoordinates, actualColor);
}
```

Training MLPs in Shaders

Optimizer

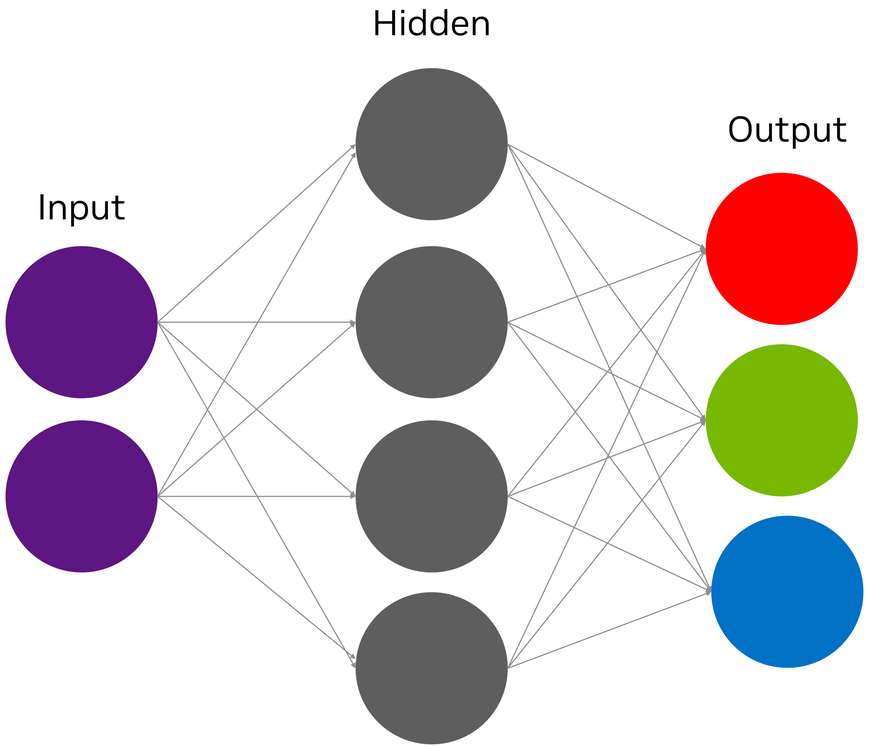
- With the gradients we now iterate through each of the weight and bias adjust them accordingly

```
float optimizerStep(float weightBias, float gradient, float learningRate)
{
	float updatedWeight = weightBias – learningRate * gradient;
	return updatedWeight;
}
```

MLPs in Shaders

First Attempt

- Let try this simple network

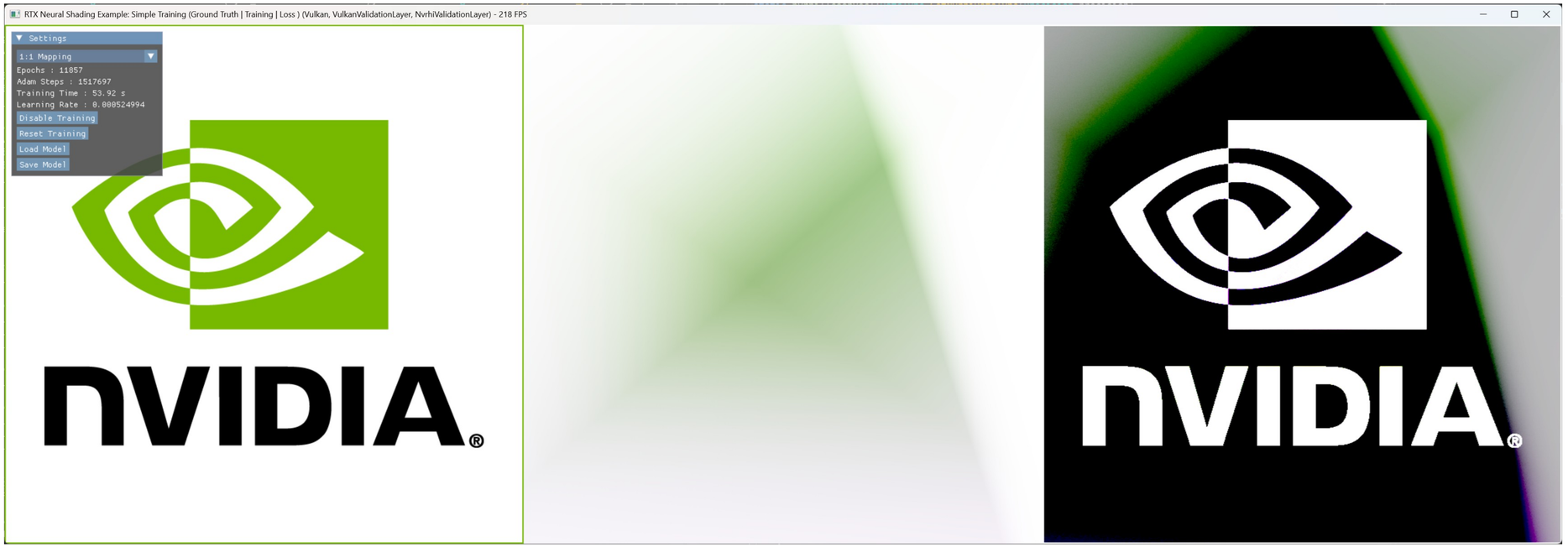


Results

- With this first attempt, did it get close?

Results

- With this first attempt, did it get close?
- Well, no. This leads to the key part of training a neural network



Reference | Prediction | Error

Iteration

SlangPy to the Rescue!

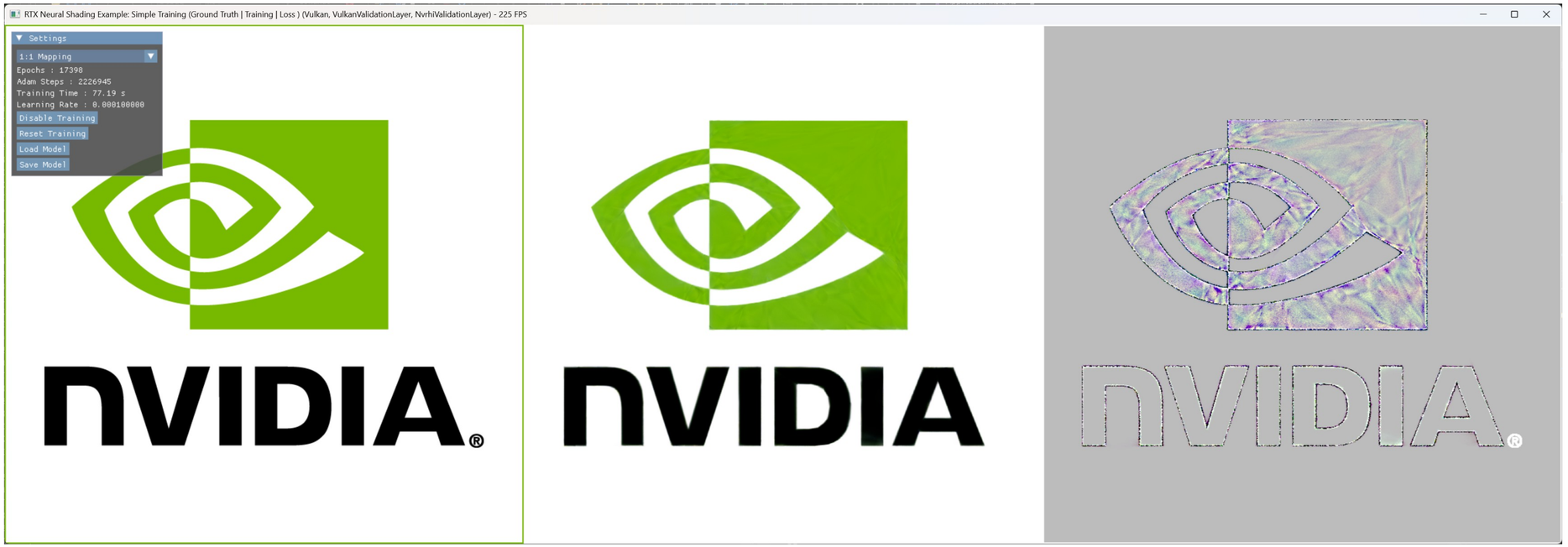
- When training a neural network for a new task, the initial results are rarely perfect. Iteration is essential and the ability to iterate quickly is even more important.
- Efficient GPU-based training pipelines are essential for rapid experimentation and refinement, which is why SlangPy was developed.
- SlangPy provides both Python and C++ interfaces to Slang, enabling fast prototyping of shading and neural rendering techniques.
- Once the model performs as expected, it can be deployed in C++, reusing the same Slang code for seamless integration into production code.

```
mlp = TrainableMLP(device, spy.DataType.float16,
                   num_hidden_layers=4,
                   input_width=2,
                   hidden_width=64,
                   output_width=3,
                   hidden_act=LeakyReLUAct(),
                   output_act=SigmoidAct())
```

Iteration

SlangPy to the Rescue!

- So, with SlangPy we can quickly experiment with all configurations of the neural network
  - Including but not limited to; networks size and depth, activation functions, input encoding and different optimizers



Reference | Prediction | Error

Cooperative Vector

Cooperative Vector

API

- Cooperative vector operations allow multiple threads within a warp to jointly execute small matrix and vector computations on Tensor Cores, providing efficient acceleration for MLP inference and training
  - They are a long vector type that extends traditional vector ranges up to 128 elements.
- Cooperative vector functionality is vendor neutral on DirectX 12 and available on Vulkan through an NVIDIA extension.
- DirectX 12
  - DirectX Agility SDK 1.717.0-preview* with Shader Model 6.9 preview
- Vulkan
  - VK_NV_cooperative_vector
  - Available from Vulkan SDK 1.4

*Don’t ship with the preview SDK

Cooperative Vector

Shader Code

- Cooperative vector provides the key functions we need to accelerate inference and training within shaders
- Inference
  - Matrix Multiply (Add): coopVecMatMul(Add)
    - Input Vector * Matrix (+ Bias)
- Training
  - Outer product Accumulate: coopVecOuterProductAccumulate
    - Compute the outer product of two vectors and accumulate the results into memory.
  - Reduction Accumulate: coopVecReduceSumAccumulate
    - Accumulate element of the input vector into memory.

Cooperative Vector

Inference in the Graphics Pipeline

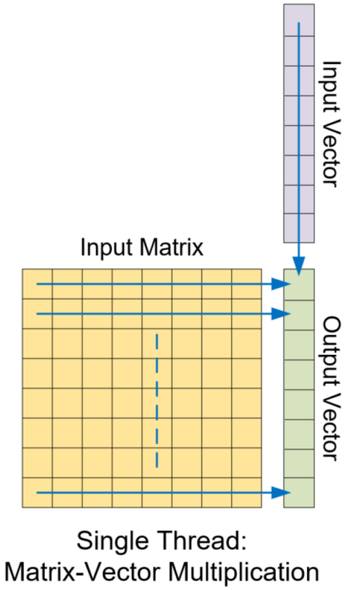
- In shader code, MLPs are implemented directly within the regular shading stage:

```
float3 forwardPass(uint2 uvCoordinates)
{
        // Provide input
        CoopVec<half, 2> inputParams = uvCoordinates;

        //Generate output
        CoopVec<half, 4> hiddenParams;
        hiddenParams = coopVecMatMulAdd<half, 4>(inputParams, matrixBiasBuffer, matrixOffset[0],...)
        hiddenParams = activation(hiddenParams)

        CoopVec<half, 3> outputParams;
        outputParams = coopVecMatMulAdd<half, 3>(hiddenParams, matrixBiasBuffer, matrixOffset[1],...)
        return float3(finalActivation(outputParams).xyz);
}
```

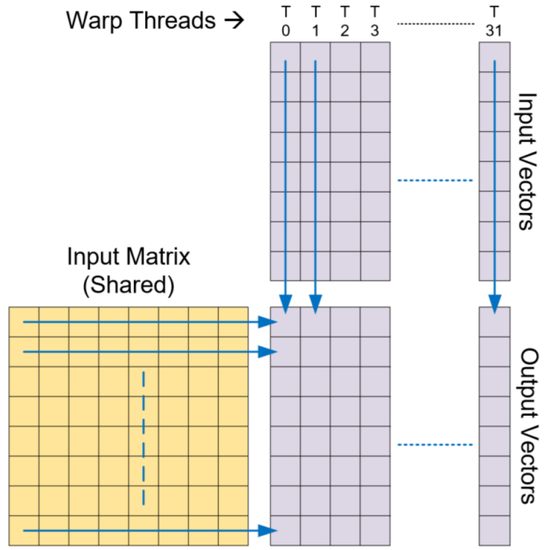
Cooperative Vector

Mapping to Hardware

- Hardware Tensor Cores
  - Provides Matrix-Matrix multiplication using entire wave/warp
  - Low precision (FP16, FP8, INT8)
- Cooperative Vector API
  - Provide Matrix-Vector multiplication in each thread



Single Thread:
Matrix-Vector Multiplication

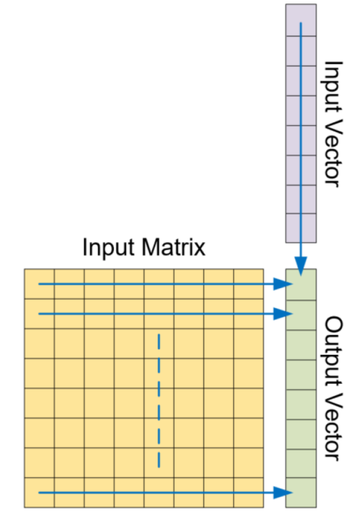


Many Threads:
Matrix-Matrix Multiplication
Maps to Tensor Cores

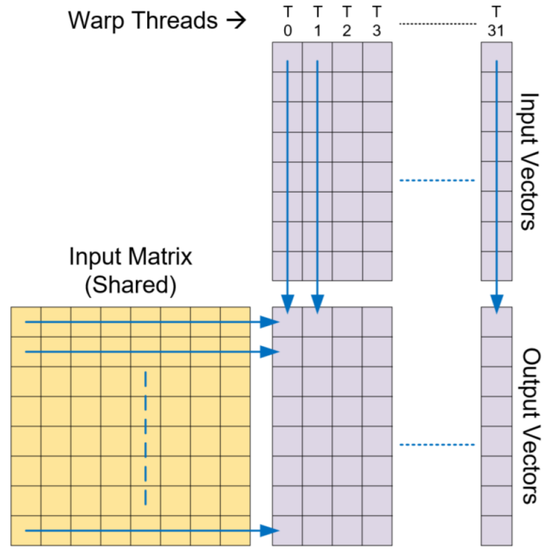
Cooperative Vectors

Mapping to Hardware

- Cooperative Vector can combine Matrix-Vector multiplications from all threads in a wave / warp into a single matrix.
- This can be evaluated in a single Matrix Multiply Accumulate (MMA) across the entire wave / warp on the Tensor Cores
- However, the shading language allows matrix inputs to be different per thread.
  - If this is the case the driver will transparently serialize the divergent matrix operation.
- For optimal performance, matrix inputs should be consistent across all threads within a wave.



Single Thread:
Matrix-Vector Multiplication

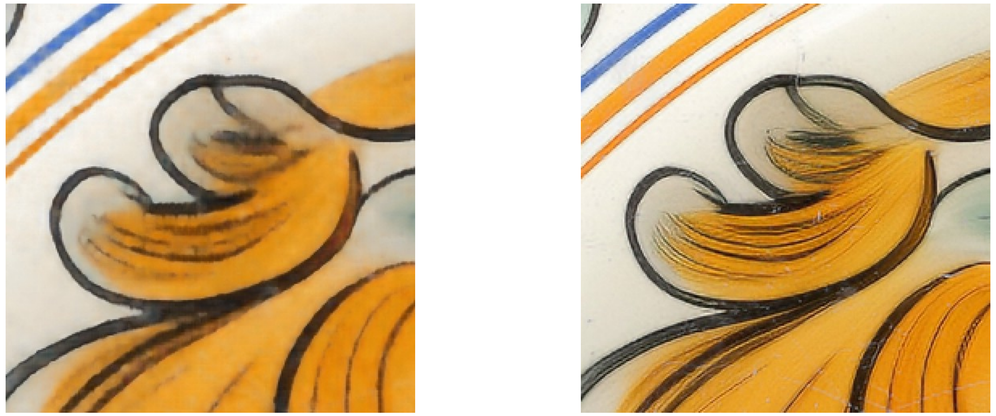


Many Threads:
Matrix-Matrix Multiplication
Maps to Tensor Cores

Applications: Neural Texture Compression

What is NTC?

- Neural Texture Compression (NTC) is a machine learning–based method for texture storage and reconstruction.
- It encodes textures into compact latent features instead of storing full-resolution texels.
- At runtime, a small neural network reconstructs texture values from the latent features on the GPU.
- NTC is deterministic, not generative.



Crops from an NTC compressed texture at 0.5 and 20.0 bpp

Why NTC?

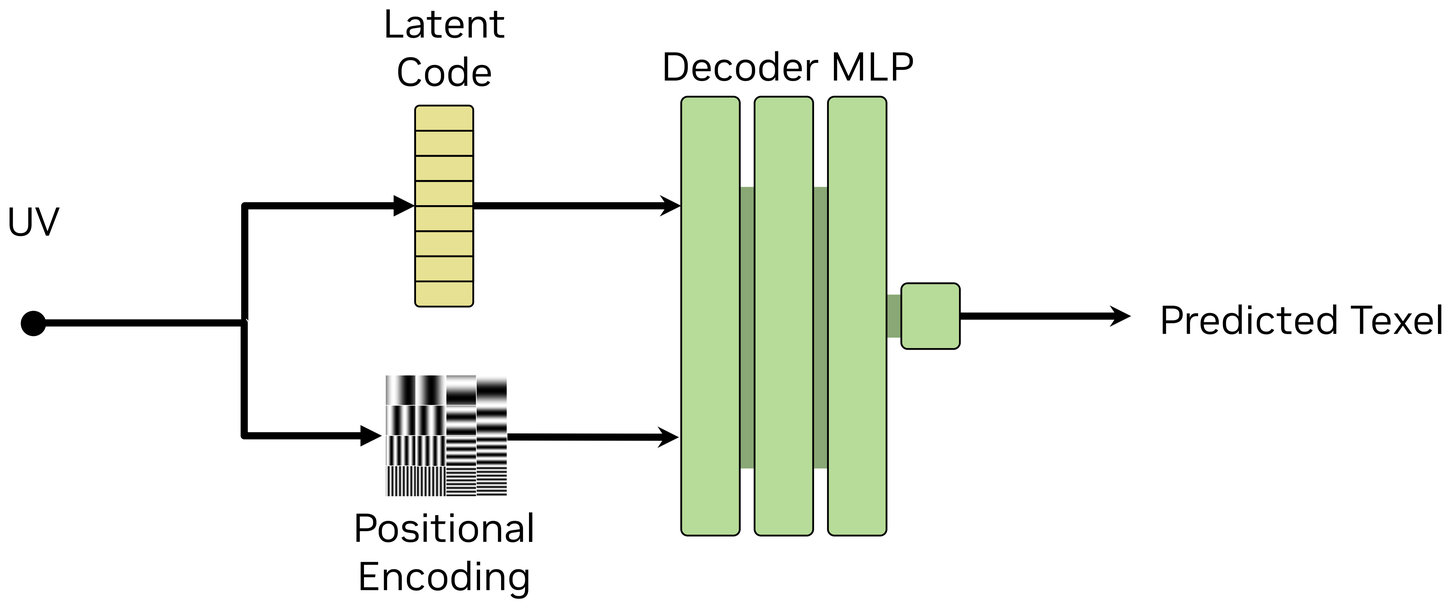
- NTC achieves higher compression ratios than formats like BCn.
- It supports high channel count materials, efficiently compressing multi-channel data
- All while reducing disk footprint and download size via more compact texture storage.

Latent Textures

- Textures are encoded into latent feature maps, stored as multi-channel neural data rather than traditional texels.
- Each latent texel stores a learned feature vector, capturing material information instead of final color values.
- A neural decoder reconstructs full-resolution textures from these latent features at runtime.
- Latent textures achieve high compression ratios by reducing redundancy and learning shared texture patterns.

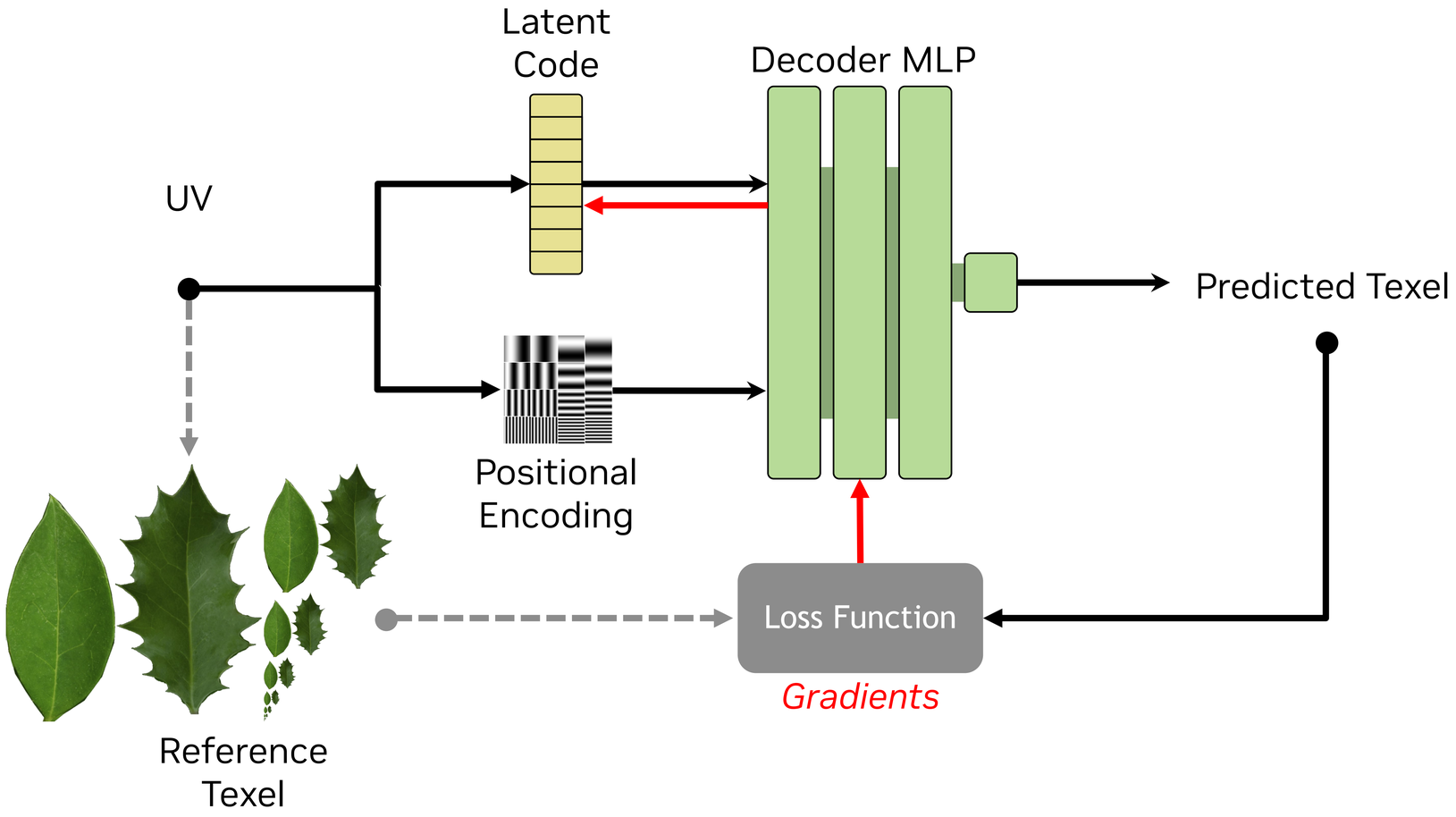
Learned Reconstruction

Network



Learned Reconstruction

Training the Network



Examples

Tuscan Villa Scene with BCn textures – 6.5 GB VRAM

Examples

Tuscan Villa Scene with NTC textures – 970 MB VRAM

Examples

Downscaled BCn Textures – 970 MB VRAM

Examples

Full Resolution NTC Textures – 970 MB VRAM

Examples

Quality Comparison

Neural Texture Compression

Benefits

Practical

- Reduces disk footprint, lowering install and patch sizes
- Lowers download bandwidth requirements, enabling faster content delivery.
- In some use case, decreases VRAM usage by storing textures as compact latent data.
- Can be used now
  - SDK available: github.com / NVIDIA-RTX / RTXNTC

Conceptual

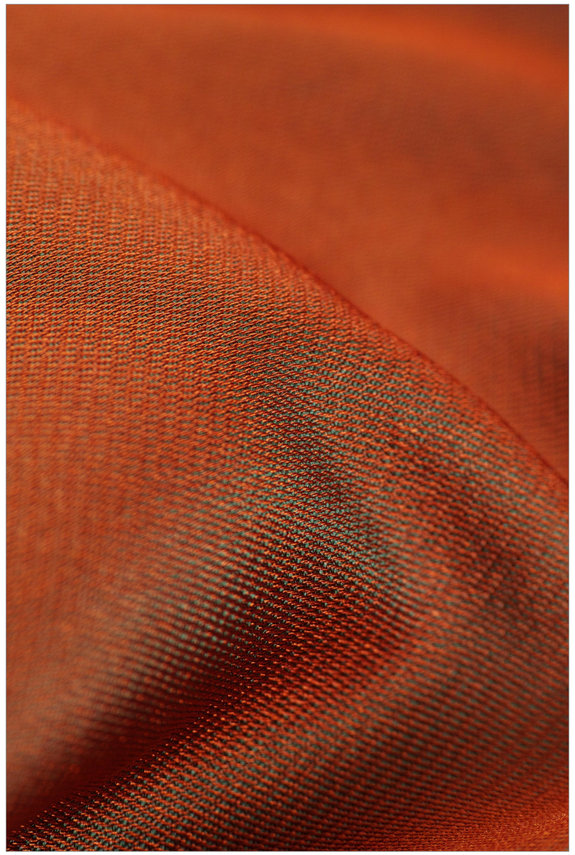
- Enables higher detail materials within the same memory budget.
- Can be extended with perceptual loss functions for higher compression ratios with better visual detail

Applications: Neural Materials

Real Materials

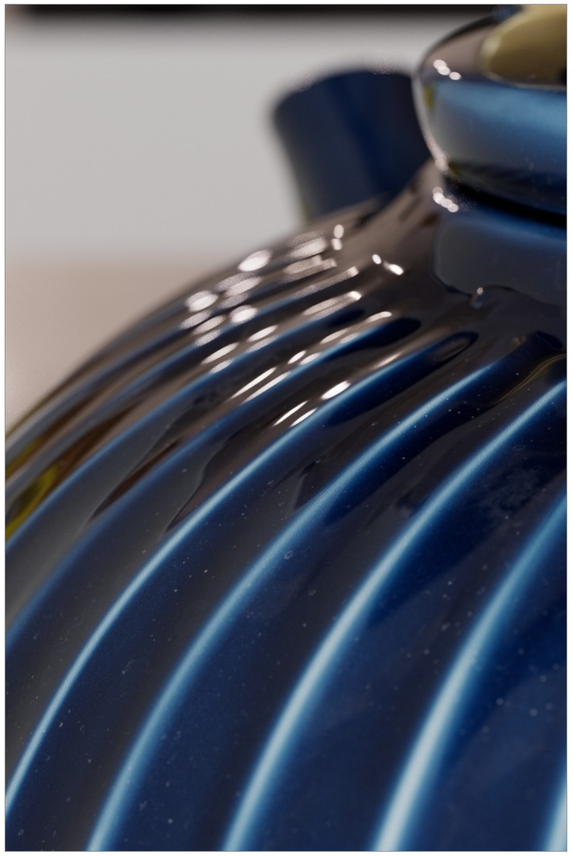
Inspiration



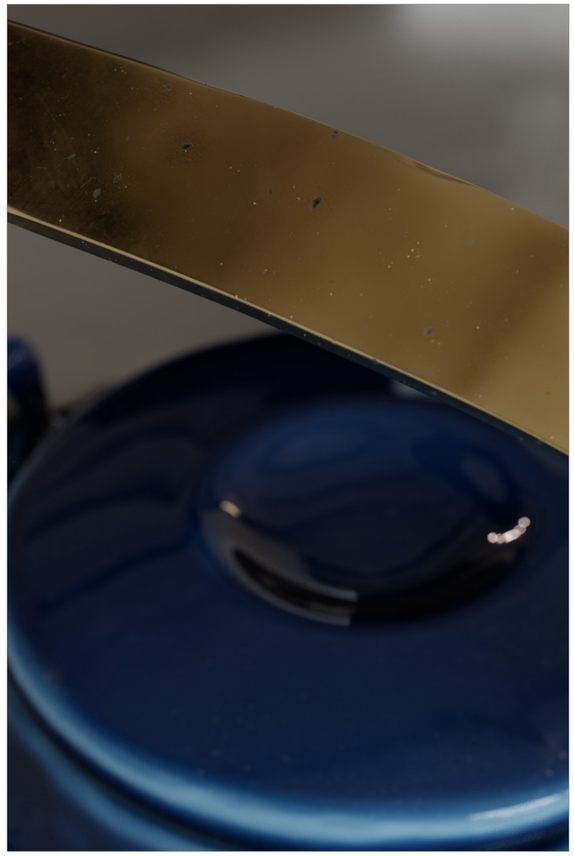



Materials

We can render such complex materials BUT not in real time

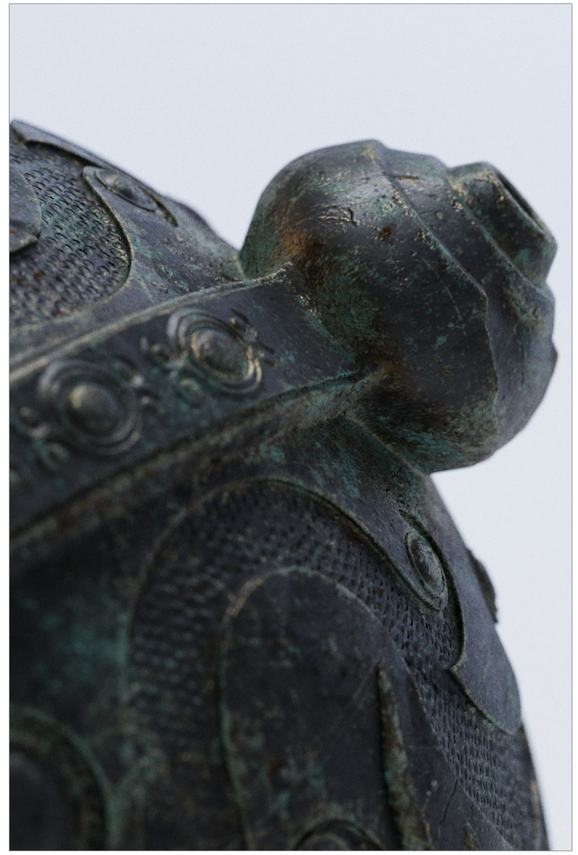


Blue Teapot Ceramic



Metal Teapot Handle

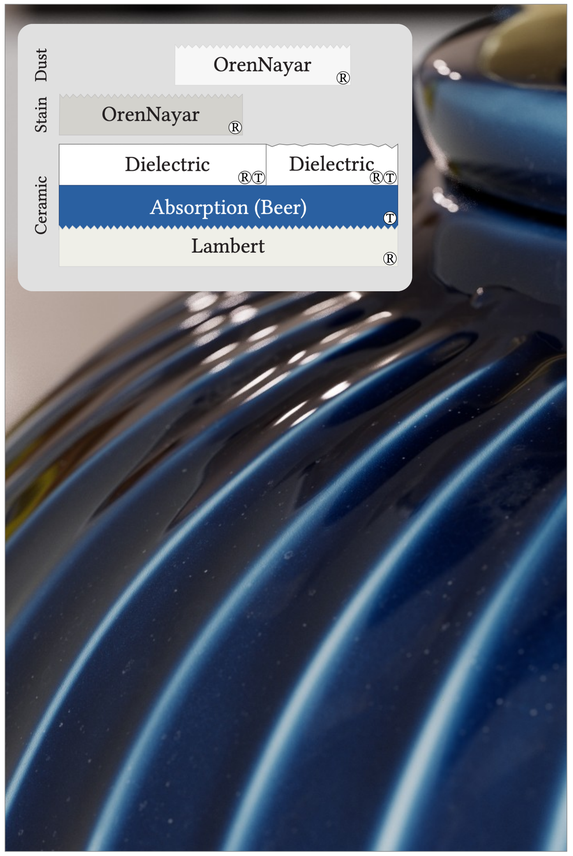
Metal Slicer Blade



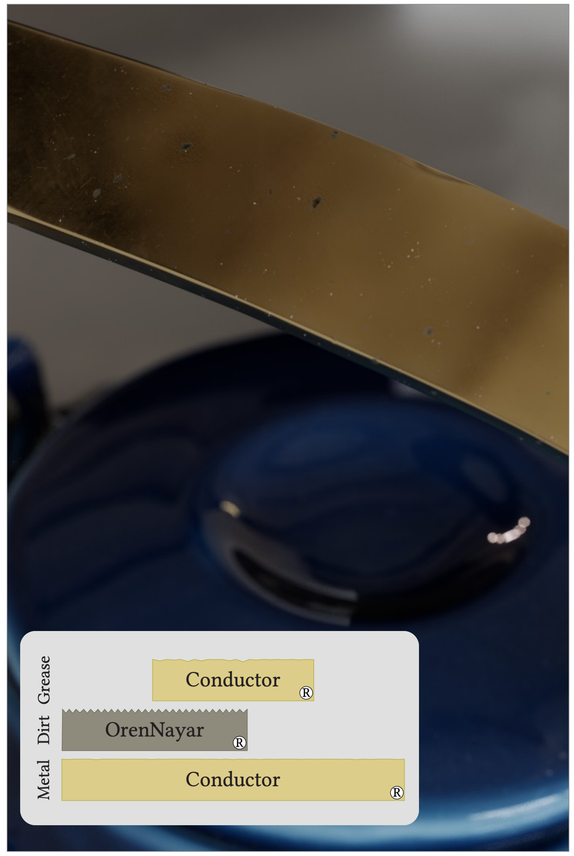
Aged Metal Inkwell

Materials

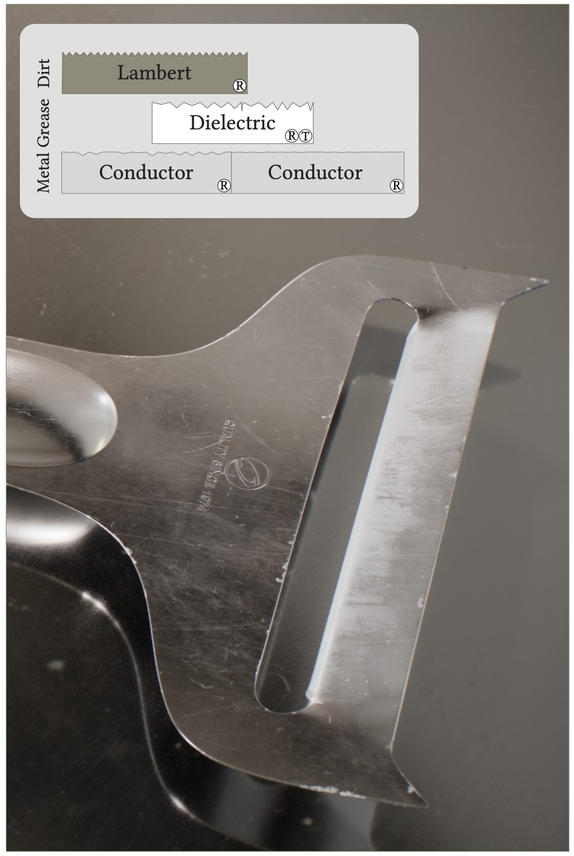
These are complex materials graphs



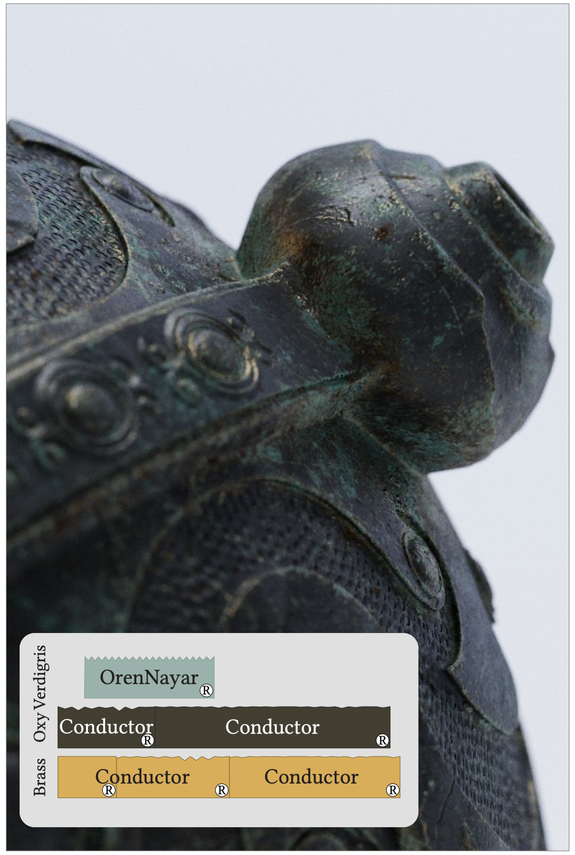
Blue Teapot Ceramic



Metal Teapot Handle



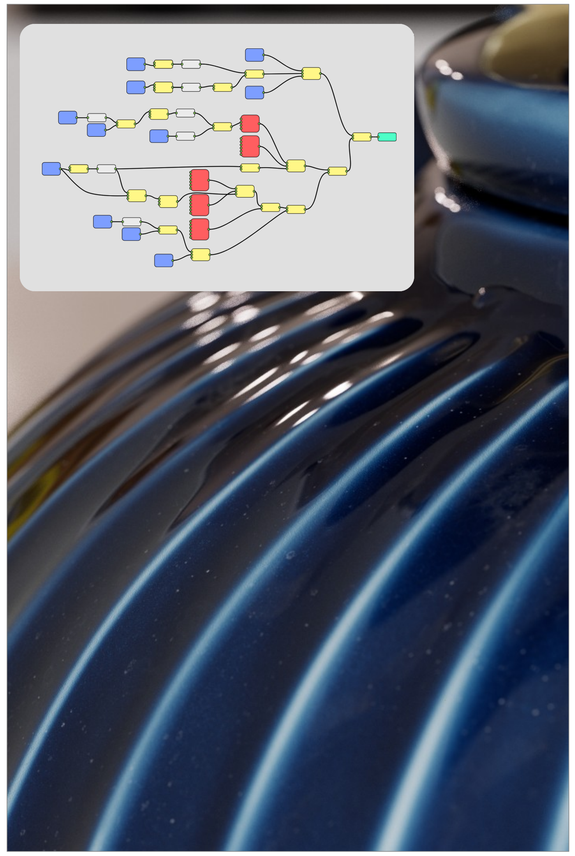
Metal Slicer Blade



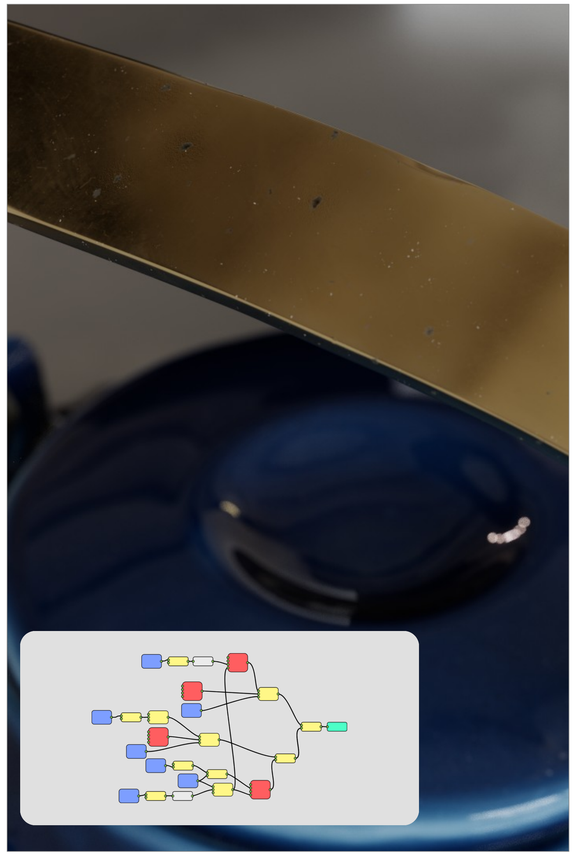
Aged Metal Inkwell

Materials

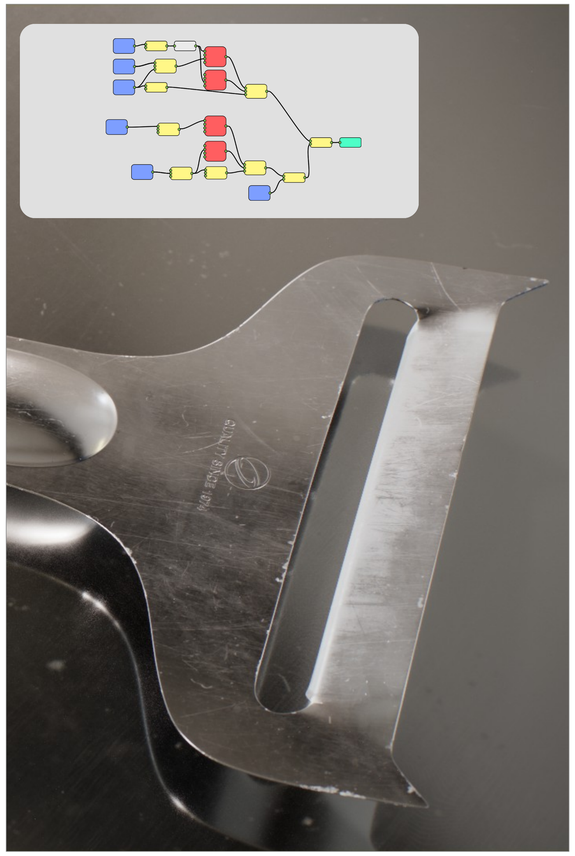
Which we don’t know how to simplify



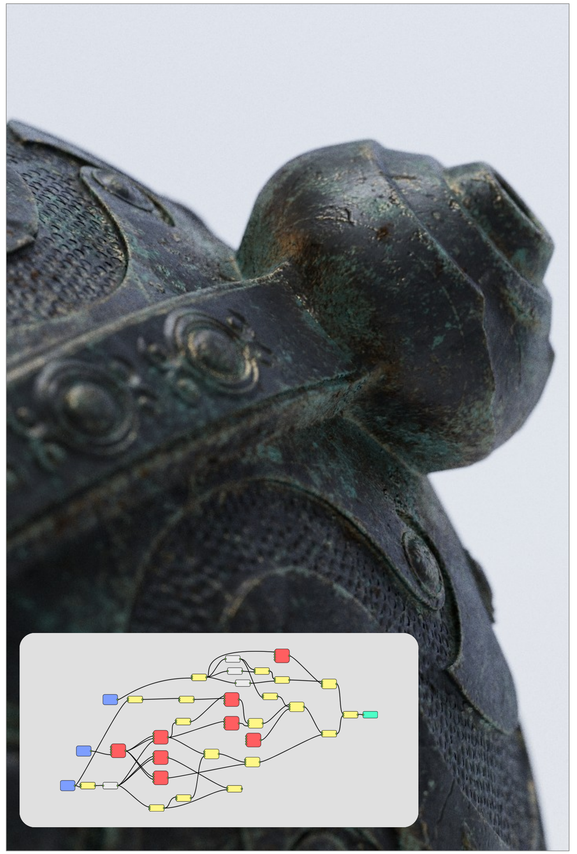
Blue Teapot Ceramic



Metal Teapot Handle



Metal Slicer Blade



Aged Metal Inkwell

What are Neural Materials?

- Neural Materials represent material appearance using learned neural features instead of hand-authored parameters.
- They compress many material channels into a compact latent representation for efficient storage and streaming.
- This enable richer, more detailed materials within the same memory and bandwidth budget.

What Makes a Material Realistic?

- Let's look at this material in more detail
- Artists have long understood that achieving realism in CG materials means combining multiple material layers, each capturing a different light-reflection behaviour.

Substrate Reference

1 Base Ceramic

Substrate Reference

2 Gold Vapor

Substrate Reference

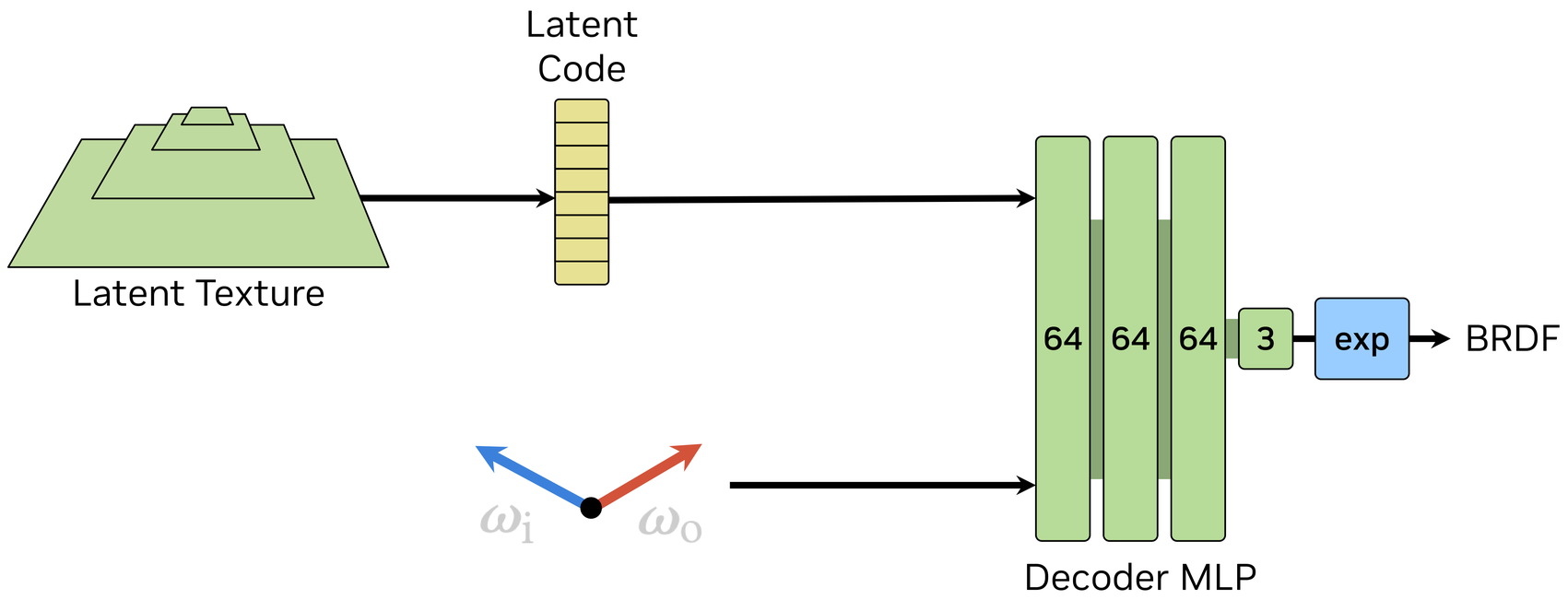
3 Glazing


Substrate Reference
19 Texture Channels
4 Dust

Neural Materials

Training

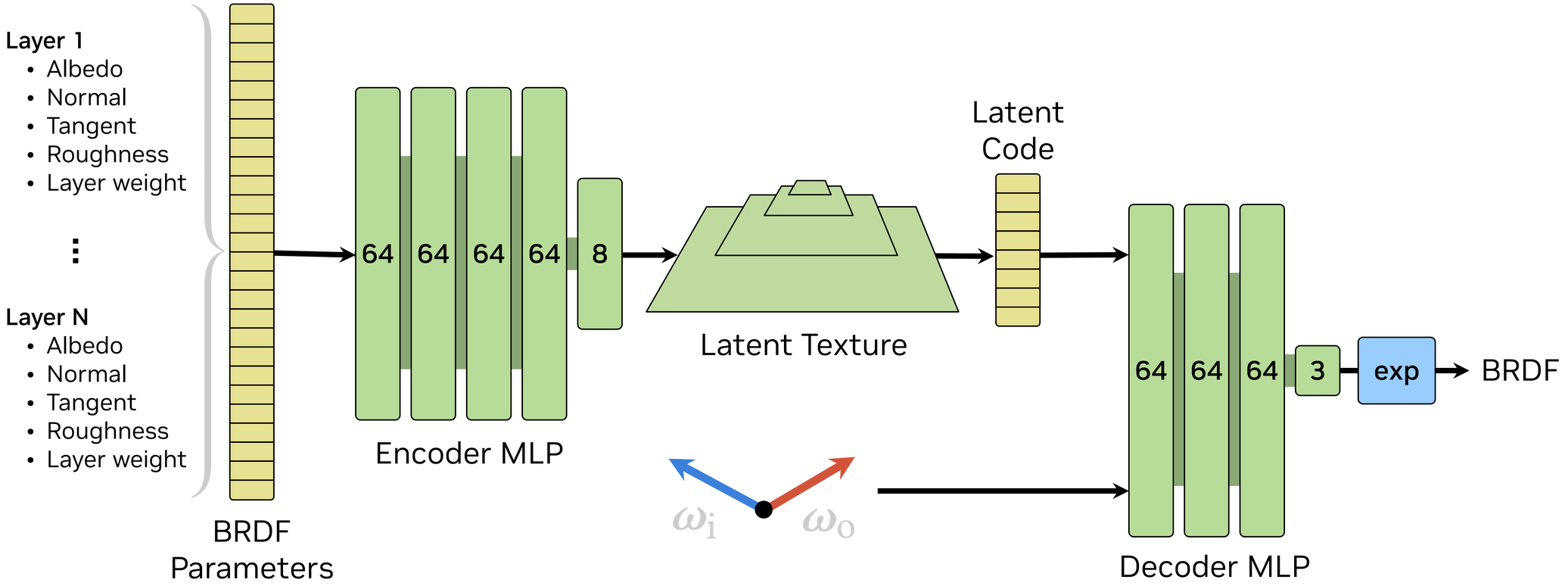
- What if we used a neural network to represent a material, how would we train it?



Neural Materials

Improving Training

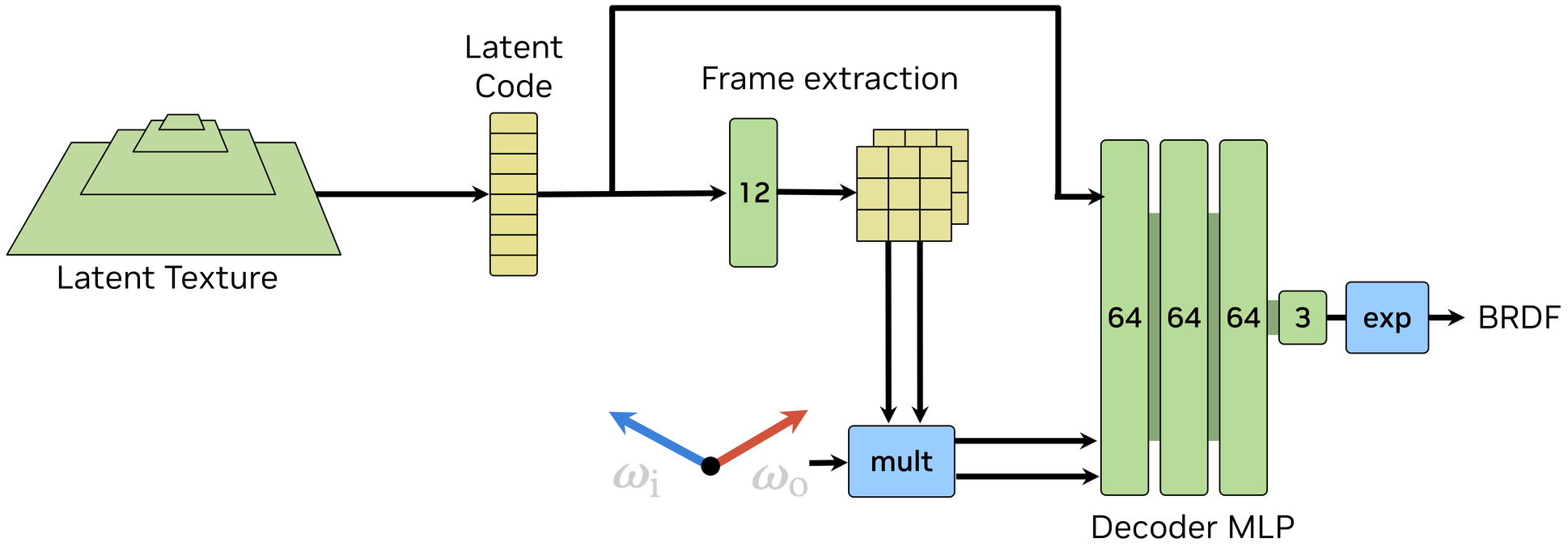
- We can extend this model to better represent the input texture



Neural Materials

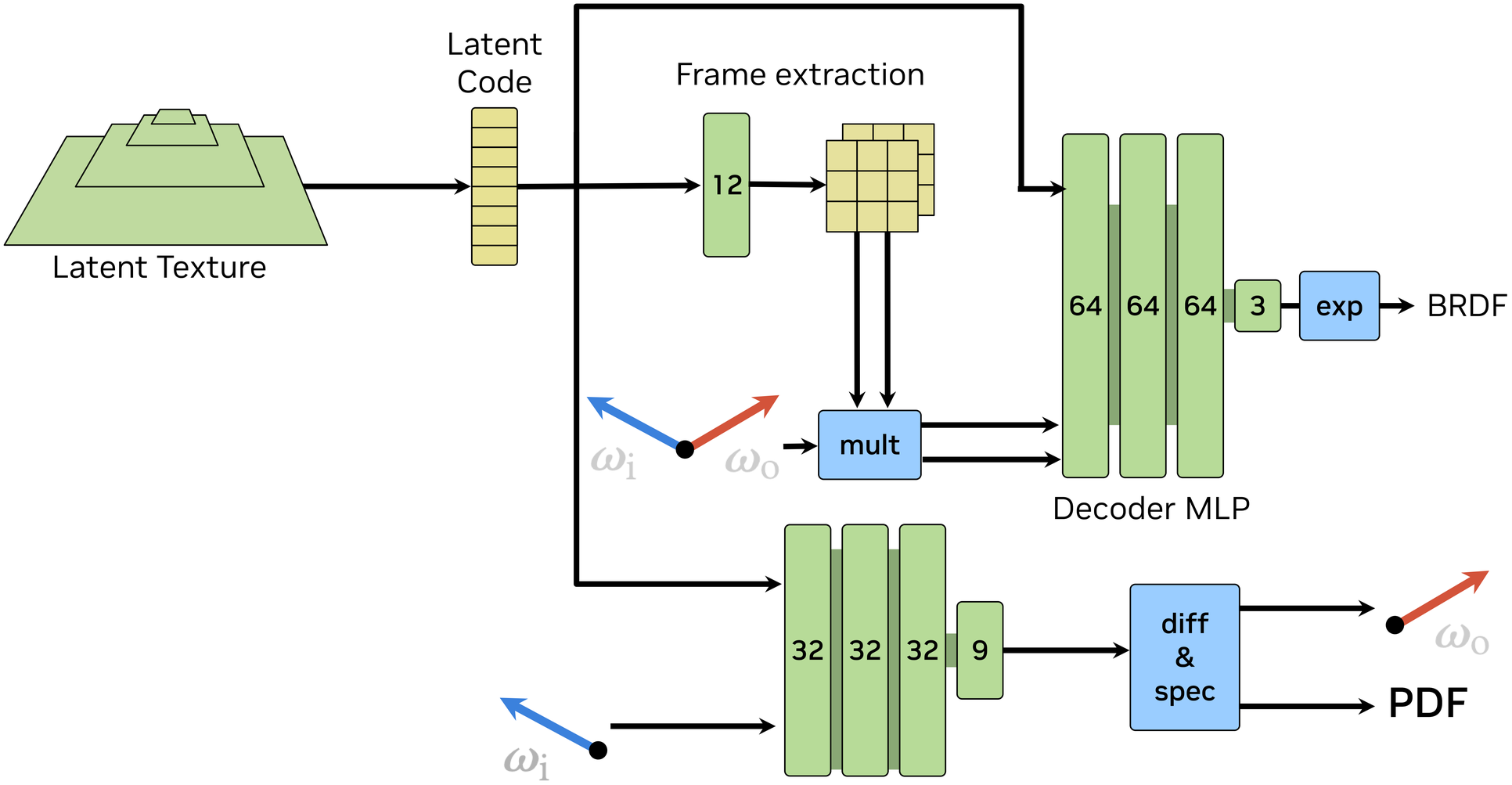
Improving the Network: Normal Maps

- Complete BRDF prediction is a little more involved than just a simple MLP. Let's improve the network



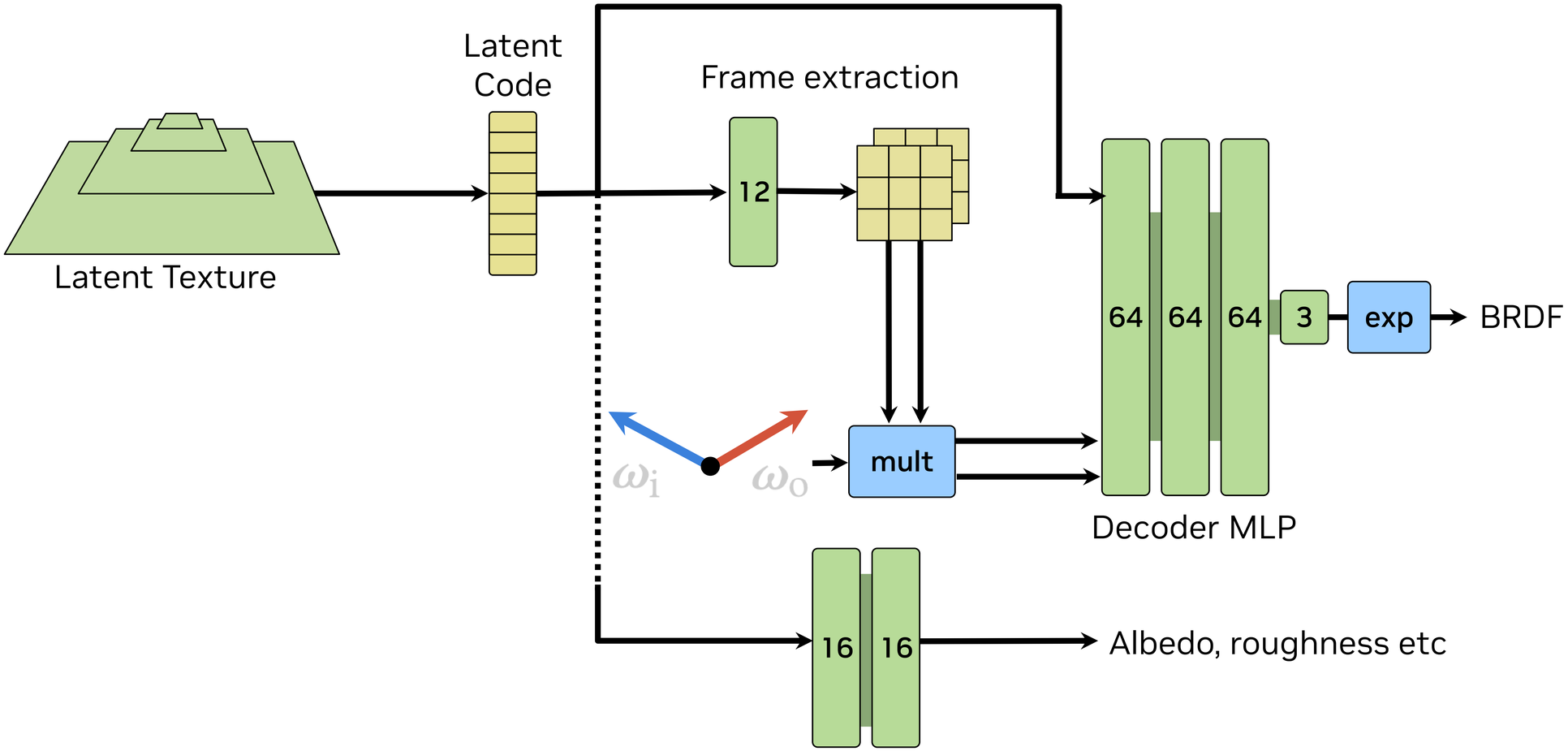
Neural Materials

Auxiliary Networks: Importance Sampling



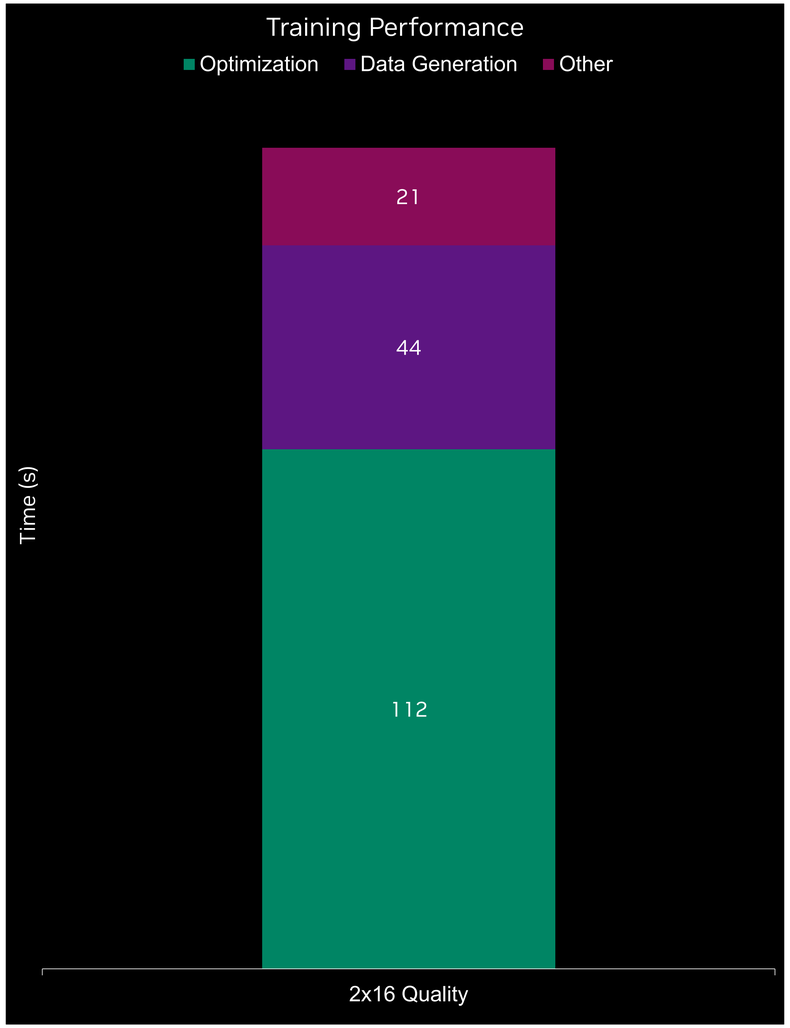
Neural Materials

Auxiliary Networks: Denoiser Inputs




Neural Material
Real-Time Path Tracing + DLSS-RR
Reference: 19 texture channels
Neural: 8 texture channels

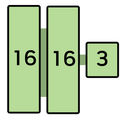
Training Performance



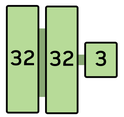
Inference Performance & Quality Comparison

RTX 5090, 2k, fully path-traced

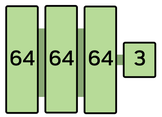
Neural
2x16 wide layers



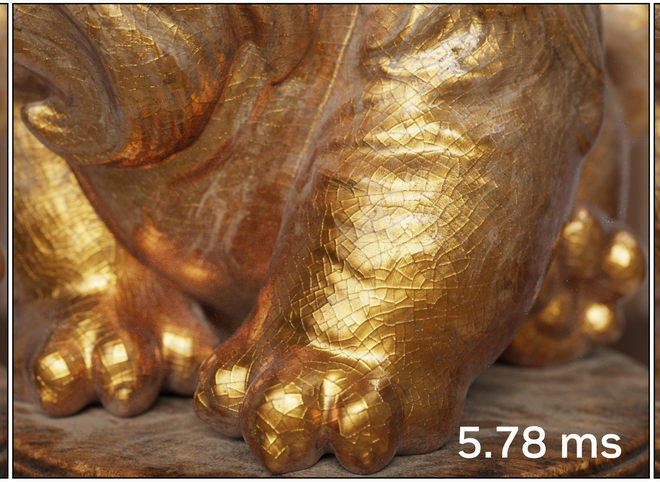
Neural
2x32 wide layers

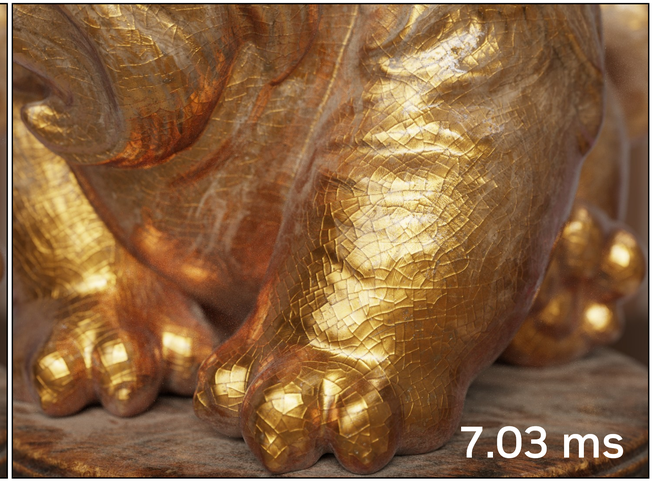


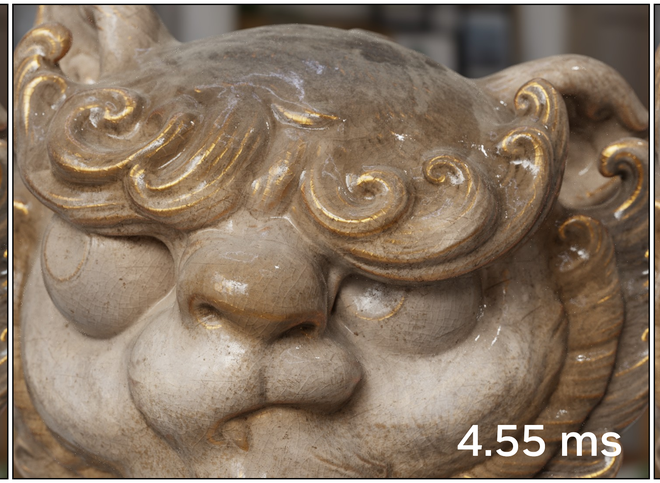
Neural
3x64 wide layers

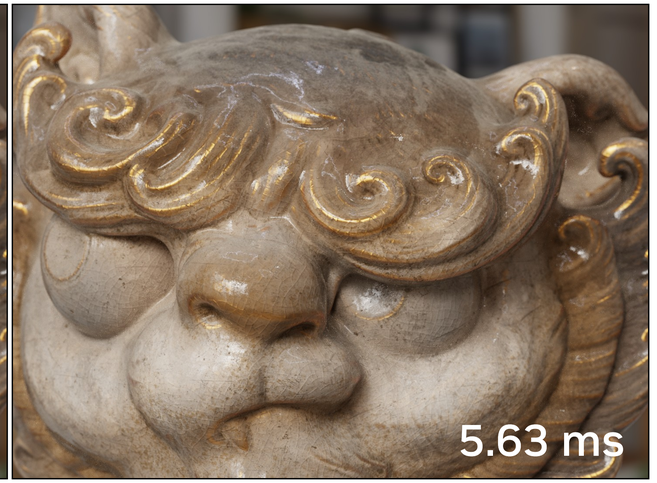


Reference





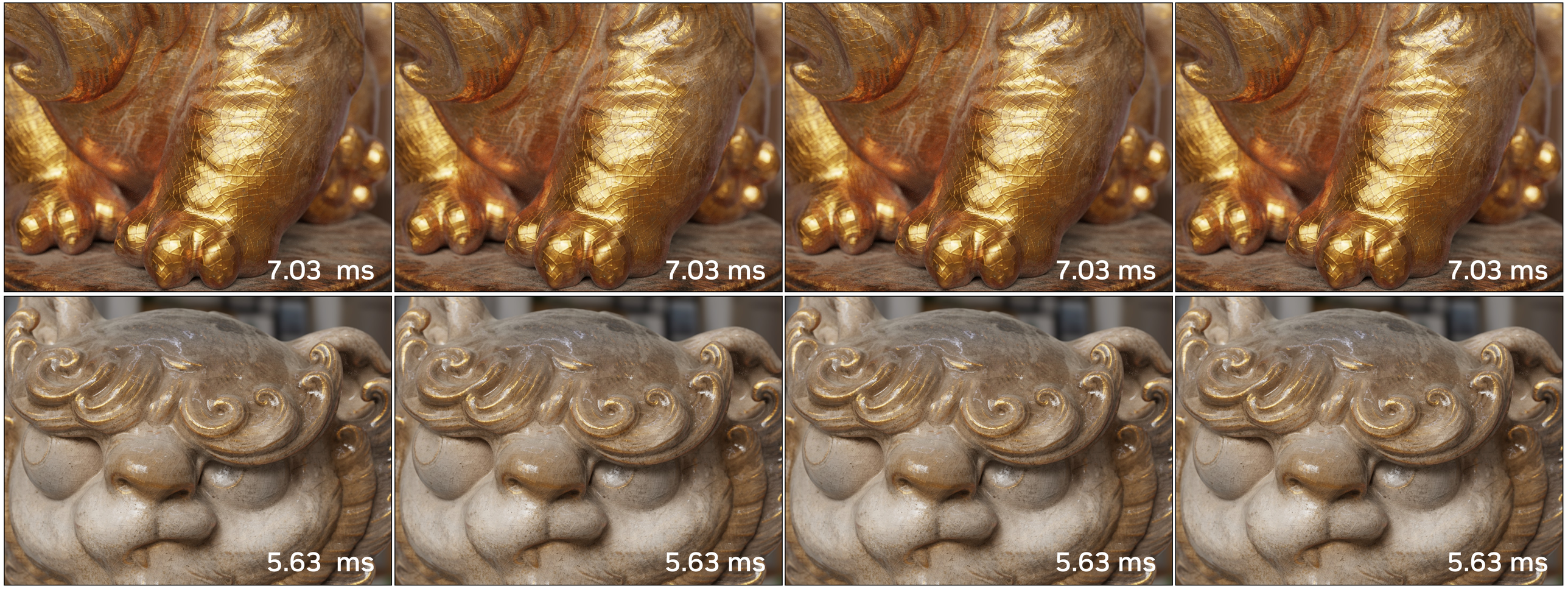




Reference Quality & Performance Comparison

RTX 5090, 2k, fully path-traced

Reference | Reference | Reference | Reference



Performance

Why Neural Materials Are Faster?

Neural Materials replaces heavy BRDF math and multiple texture reads with a lightweight neural decoder.

- They avoid complex analytic BRDF evaluations
- They reduce memory traffic by collapsing several multi-channel textures with a compact multi-channel latent texture
- It computes all material layers in a single, efficient pass.

Neural Materials

Benefits

Neural Materials brings high quality complex materials to real time rendering.

- Encodes complex material properties into compact neural representations.
- Reduces texture size and bandwidth while maintaining visual fidelity.
- Enables real-time rendering of high-quality materials learned from data.

Call to Action

Call to Action

- Neural shading is not difficult it is just new!
- We are in the exploration phase of the technology.
- If you would like to learn more pull the Neural Shading SDK and SIGGRAPH course.
- Try experimenting for yourself!

Getting to know Slang

16:00 Secondary Room

Resources

- **RTX Neural Shading**
  - https://github.com/NVIDIA-RTX/RTXNS

- **Neural Shading Course SIGGRAPH 2025**
  - https://research.nvidia.com/labs/rtr/publication/duca2025neural/