Efficient Graphics Representation with Differentiable Indirection - Supplemental

Sayantan Datta
McGill Univeristy
Canada
Meta Reality Labs
USA

Carl Marshall
Zhao Dong
Zhengqin Li
Meta Reality Labs
USA

Derek Nowrouzezahrai
McGill University
Canada

ACM Reference Format:
Sayantan Datta, Carl Marshall, Zhao Dong, Zhengqin Li, and Derek Nowrouzezahrai. 2023. Efficient Graphics Representation with Differentiable Indirection - Supplemental. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ADDITIONAL DETAILS

This section provides additional implementation details, results, and visualizations for each task.

1.1 Isotropic GGX approximation

Isotropic GGX is a popular *BRDF* (*Bidirectional Reflectance Distribution Function*) for modelling glossy reflections in variety of real-time applications. The function itself is very simple, given by:

$$D(h_z, \alpha_h) = \frac{\alpha_h^4}{\pi \cdot (1 + (\alpha_h^4 - 1) \cdot h_z^2)}, \tag{1}$$

where $\alpha_h \in [0, 1)$ controls the glossiness of a surface and $h_z \in [0, 1)$ is the dot-product of the half-vector bisecting the camera and emitter direction with the surface normal. Usually, for a material, an artist sets the glossiness value α_h while the second parameter h_z is obtained directly from the surface normal, emitter and camera direction at the location of shading. When rendering, the equation 1 is evaluated independently for each pixel in a frame. We refer the readers to online tutorials [11] for a lightweight introduction to the subject and other resources [5, 12] for a more rigorous description. A crucial aspect to note here is the difference in how α_h and h_z is obtained in a real-time environment. α_h being an artist control parameter, is associated with the material properties of a 3D mesh as authored by an artist. Such artist control parameters are often stored as textures – uv-mapped to the surface of a mesh. On the other hand, h_z is associated with the geometry and the location of light and camera w.r.t to shade point. As such, h_z is computed on the fly while α_z is fetched from memory. We exploit this distinction in the next section when we approximate the Disney BRDF.

Training and inference: For *Isotropic GGX* training, we use a cascade of two 2D arrays. The primary array is initialized with an undistorted uv-rmap while the cascaded array with a constant 0.5. Since the cascaded array stores values that are beyond $[0, 1)$ range, we do not use any non-linearity for the cascaded array. For *Isotropic GGX*, we set $\rho = 2$. As described in the main document, ρ is the ratio of he ratio of the length of one side of the primary array to the cascaded array. Thus, the primary array is 16x16 and the cascaded array is 8x8 resolution. We obtain training samples from appropriate distributions for the two inputs; h_z sampled from cosine-hemisphere [6] distribution and α_h from an exponential-like distribution with more samples biased towards lower roughness values. Output of the network is compared with the reference analytic output and the network is trained in a local coordinate frame with normal pointing at (0,0,1).

Visualizations and conclusion: Figure 1 visualizes the inference pipeline for *Isotropic GGX* case. The pipeline replaces several FLOPs of compute required to evaluate equation 1 with two memory lookups. Note that our approach produces inherent advantage with increasing resolution as the cost of memory lookups may scale sub-linearly with resolution due to caching as opposed to compute *FLOPs* that scales proportionally with resolution. Thus the case of folding compute *FLOPs* into memory lookups can be interpreted as amortization through memory; compared to super-resolution which amortizes cost by exploiting redundancies in screen-space.

1.2 Disney BRDF approximation

In section 1.1, we saw the specular component of a *BRDF*. Practical *BRDFs*, such as *Disney* is composed of multiple components that allow artists to manipulate and develop plausible materials by controlling a set of parameters. Our implementation, based on the reference [4] document, employs four components: **diffuse**, **metallic**, **clearcoat**, and **sheen**. Controlling these components are the 10 artist control parameters, often stored as uv-mapped textures. These parameters are categorically similar to α_h in section 1.1, except we have many more them. *Disney* also requires 7 geometric dot-products, similar to h_z in section 1.1. Thus *Disney* has total 17 input parameters as opposed to just 2 for *Isotropic GGX*.

Our goal is to fold as many compute operations (for evaluating the reference *BRDF*) into memory lookups while retaining quality and improving runtime performance, as tested on Nvidia Mobile 3070. This is challenging as we must minimize the number of memory lookups as each lookup is 7-10× more expensive compared to a single *MAC* operation, assuming the data resides in closest tier

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

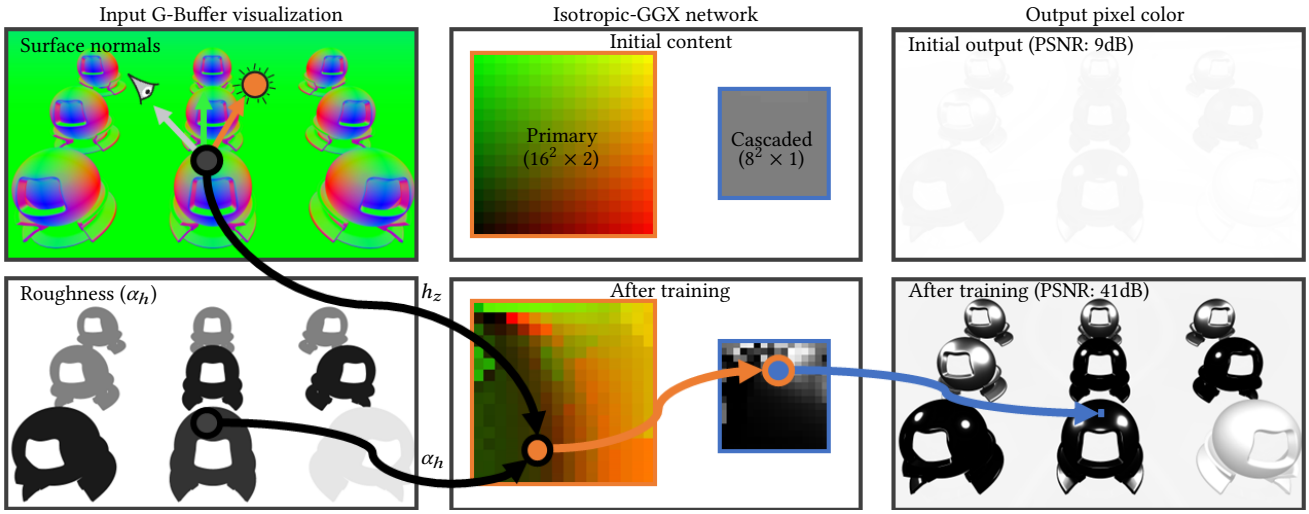

Figure 1: Figure visualizes the *Isotropic GGX* inference pipeline. Network input is computed from various G-Buffer components as visualized on the left. Content of primary and cascaded array is visualized in the centre while the network output (pixel color) is shown on the right for before and after training.

cache or *SRAM*. Here we show one of the many possible uses of *differentiable indirection* to approximate *Disney BRDF* while adhering to the aforementioned constraints. While the exact recipe is not important, it may be useful to highlight some practical motivation that we take into consideration.

Following from the main paper, we refactor p and q according to

$$\begin{aligned} p(x) &= c_d D_d + c_{m0} D_m + c_{s0} \\ q(x) &= c_c D_c + c_{m1} D_m + c_{s1}, \end{aligned} \tag{2}$$

where D_d, D_m, and D_c are the *Disney-diffuse*, *Disney-metallic*, and *Disney-clearcoat* distribution (equation 5, 8, and 12 in reference document [4]). Such factorization minimizes color bleeding by separating out the albedo from final *BRDF* expression. We first detail the D_d, D_m, and D_c term followed by a discussion of rest of the C_* terms. The D_* terms constitutes the distinctive makeup of the *BRDF*, hence, we approximate them with best possible quality.

Disney-metallic (D_m): The *Disney-metallic* term is modelled using the *Anisotropic GGX* function as described by equation 8 in the reference [4] document. *Anisotropic GGX* requires four input – the roughness α_x, α_y along the two tangents and the dot-products (h_x, h_y) of half-vector (similar to *Isotropic GGX* case) with the two surface tangents. While it is possible to model the function directly using *differentiable indirection* composed of a 4D primary and a 4D cascaded array, doing so is inefficient due to several reasons. First, the raw size of 4D arrays are too big to accommodate in lowest tier caches or *SRAM*. Second, two lookups adds to latency and 4D interpolations requires many *FLOPs*, which the technique originally intended to replace. Also, the process does not exploit the fact that some computations can be baked into the textures as an offline process. Instead we, rewrite equation 8 as follows:

$$D_m(\alpha_x, \alpha_y, h_x, h_y) = \frac{1}{d_0(\alpha_x, \alpha_y) \cdot h_x^2 + d_1(\alpha_x, \alpha_y) \cdot h_y^2 + d_2(\alpha_x, \alpha_y)},$$

where we learn the coefficients $d_*(\alpha_x, \alpha_y)$ using *differentiable indirection*. Note that, we now have the advantage that the primary array of the *differentiable indirection* is large (2k×2k) and works completely offline. The output of the primary array is stored as uv-mapped textures instead of α_x, α_y. The cascaded array is small 2D array that can easily fit in lower tier caches or *SRAM*.

Disney-diffuse (D_d): We express the *Disney-diffuse* term as a weighted sum of artist parameter ($\alpha_{\mathbf{d}} \in [0,1)^2$) functions f_i and geometry dot-product ($\mathbf{h_d} \in [0,1)^3$) functions g_i as shown below:

$$D_d(\alpha_{\mathbf{d}}, \mathbf{h_d}) \approx \sum_{i=0}^{N} f_i(\alpha_{\mathbf{d}}) g_i(\mathbf{h_d}).$$

The accuracy is improved with more terms but empirical observations show $N = 3$ is sufficient to attain accuracy over 35dB. Due to reasons similar to *Disney-metallic*, the artist parameter controlled functions f_i are modelled using *differentiable indirection* while the g_i terms are directly computed.

Disney-clearcoat (D_c): The *Disney-clearcoat* term is modelled similar to the *metallic* term except it is simpler. We again rewrite equation 12 in the reference document as:

$$D_c(\alpha_c, h_z) = \frac{1}{c_0(\alpha_c) \cdot h_z^2 + c_1(\alpha_c)},$$

where we model the functions c_i using *differentiable indirection*.

Coefficient terms: The coefficient terms c_* in equation 2 are trained together with the D_* terms. That is we have independent loss functions for D_* terms but we train $c_* D_*$ together. Hence, the individual c_* terms are less accurate but it is acceptable as long as they complement the D_* terms in their product. Many of the c_* and D_* terms share similar lookups, further amortizing cost.

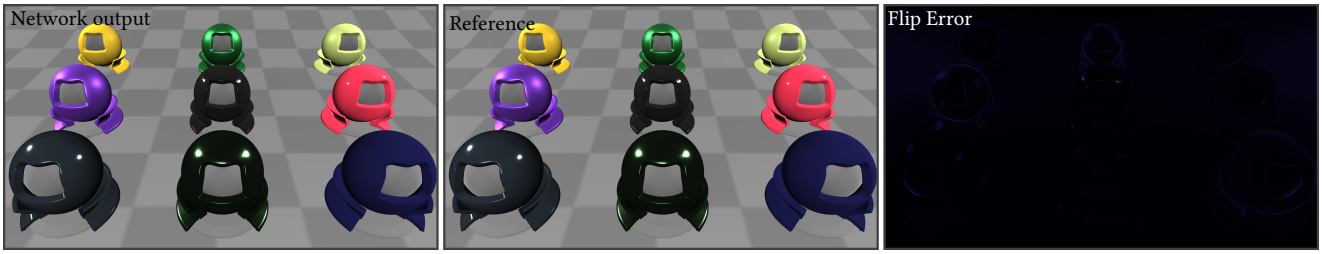

Figure 2: Figure visualizes the approximate *Disney BRDF* for a variety of material configuration.

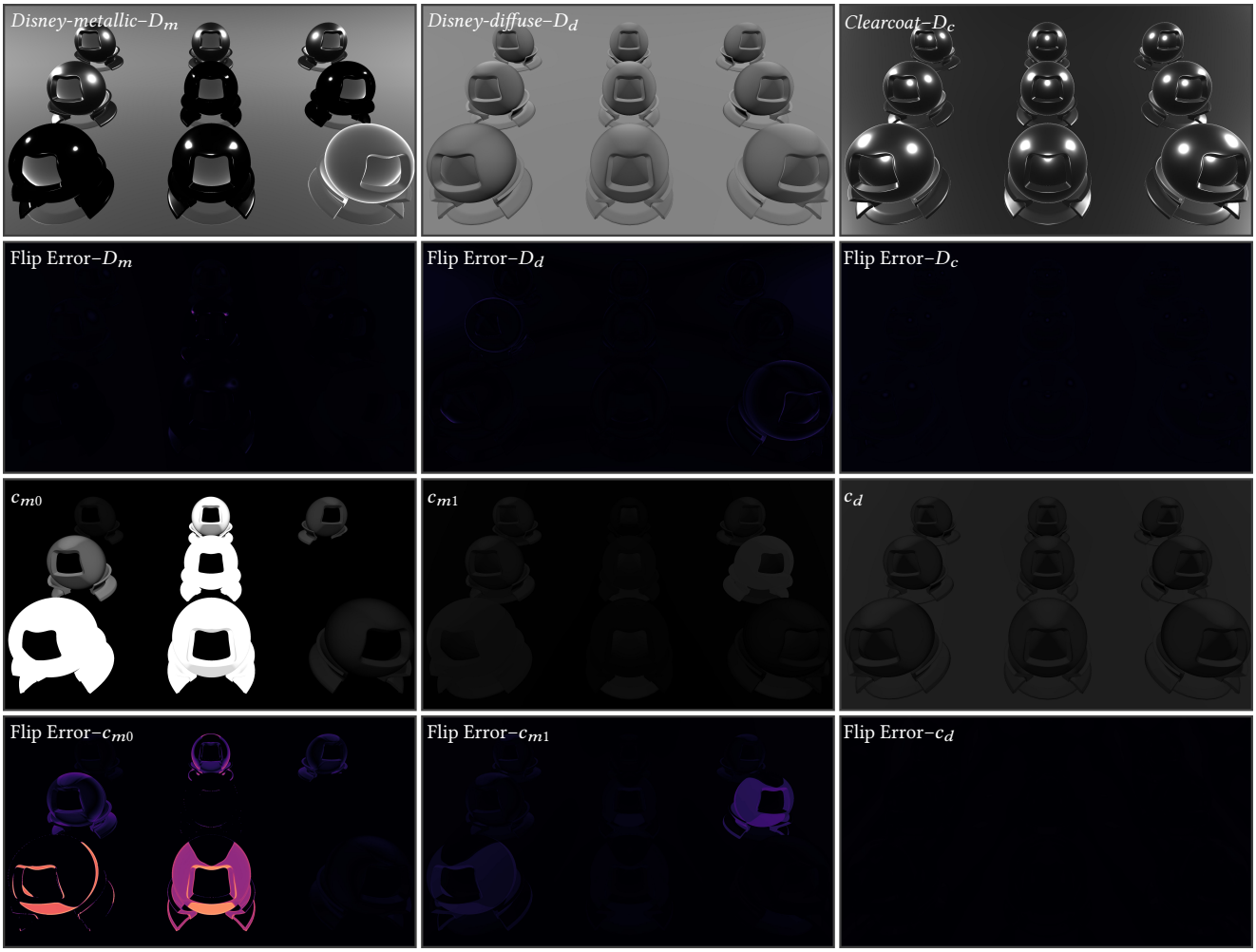

Figure 3: Figure visualizes various intermediate outputs for the approximate *Disney BRDF* under a variety of material configuration. Errors in the term c_m0 are masked by the D_m term as the product $c_{m0}D_m$ is jointly optimized while D_m is also optimized separately.

Training and inference: Training pipeline is similar to section 1.1, except we have many more parameters that we sample from the appropriate distributions. We sample the roughness parameters for *metallic* and *clearcoat* from an exponential like distribution while rest of the artist parameters and albedo from a uniform distribution. We sample the emitter and camera direction from a hemispherical distribution with normal pointing at (0,0,1). One important note is that we directly learn the denominator in the expressions for D_m, D_c, thus partially avoiding the unbounded non-linear behavior of the two functions. During inference, only the cascaded arrays, as

Table 1: Brief description of various lookups associated with Disney BRDF approximation.

Dimension/ Interpolation	O/P Channels	Associated Output
16x16 / Bilinear	3	Anisotropic GGX
16x16 / Bilinear	1	Smith Masking
16x16 / Bilinear	4	Disney Diffuse, Clear coat masking
16x16 / Bilinear	4	Metallic, Clear coat, Sheen
16 / Linear	2	Clear coat gloss

outlined in table 1, are required as the output of the primary arrays are encoded as textures.

FLOPs calculation and performance The reference Disney BRDF uses 92 additions, 150 multiplications, 17 divisions, 5 square-roots, and 1 logarithm. Our implementation, in addition to the 5 lookups as detailed below in table 1, requires 11 additions, 26 multiplications, and 2 divisions. The extra arithmetic is required to combine the output of the LUTs to the final *BRDF* output. To compare compute as a single number, we assume additions and multiplications require 1 *FLOP* each, divisions 2 *FLOPs*, square root and logarithms 4 *FLOPs* to compute. These are likely conservative estimates on modern hardware and low power hardware may require more *FLOP*s for complex instructions. Plugging these values, we estimate the required number of *FLOP*s for reference implementation is about 300 *FLOP*s. We use a factor of 0.8 to account for inefficiency in our implementation and arrive at a 240 *FLOP*s estimate for reference implementation. Similarly, our implementation requires 5 lookups and 41 extra *FLOP*s.

We note that our implementation is not unique but one of several possible implementations using *differentiable indirection*. However, our factorization yields good results close to reference with performance advantage even on commodity GPU hardware. Note the total array size is 6KB (half-precision) and 12KB (full-precision) and easily stored in on-chip memory such as *SRAM* or *L1*-cache. To obtain a performance advantage over reference implementation on a commdity GPU hardware such as Nvidia 3000 series, it is recommended to use hardware texture sampler with hardware bi-linear interpolations to fetch the results from the LUTs.

Figure 2 shows the final output of our technique for a variety of materials. Figure 3 shows the various intermediate output of our technique. Notice how errors in c_{m0} complements D_m.

1.3 Compact Image Representation

Images are multi-channel 2D arrays storing a variety of presumably locally coherent data. Images are often natural – such as photographs taken by a smartphone camera or synthetic – such as video-game textures. They are also used as containers to store other forms of data such as *normal* [10, 13] maps used to alter shading computations or *displacement* [3, 9] maps to alter the underlying geometry. Usually photographs or textures have 3 channels for storing red, green, and blue colors for each pixel. With *PBR* shading, textures may have more than 3 channels. Extra channels store information like *roughness* maps, *normal* maps, *ambient-occlusion*

Table 2: Empirically obtained optimal resolution ratios ($\rho = N_p/N_c$) for varying compression of 4K RGB-textures.

	Compression Ratio			
	3x	6x	12x	24x
RGB	128	128	80	72

maps or other spatially varying parameters required to reproduce the behavior of a material. A compact representation of images is thus of utmost importance. In a real-time context, lowering the number of compute *FLOP*s, memory bandwidth required to decode a pixel and the overall representation size is crucial.

Block compression [2, 7] techniques exploits spatial locality of images by regressing a line through a block of pixels and only storing an index identifying a point on the line for each pixel. Our technique may be considered an generalized extension of block compression where we learn the indices (in the primary array) and the regressed line (in the cascaded array) by using gradient decent and *differentiable indirection*. There are several possible arrangements for primary and cascaded array in our case. Figure 4 and 5 shows a hyper-parameter sweep across various primary/cascaded arrangements such as 2D/2D, 2D/3D, and 2D/4D.

Setup: We use a 2D array with 4 channels as the primary and a 4D array with k-channels as the cascaded array, where k is the number channels in a texture or image. We initialize the primary array with uv-ramp repeated twice for 4 channels. We initialize the cascaded array with grey, white, light-blue (0.5,0.5,1), grey for albedo, ao, normal, and roughness channels respectively. Other channels are set zero. Let us assume the length of one side of the primary array is N_p and the cascaded array is N_c. The ratio of the two sides indicated by $\rho = N_p/N_c$. Some pseudo-optimal values of ρ are provided in table 2. The values are obtained by varying ρ for a given resolution and compression and selecting the one with best output quality. Our use cases have $\rho \in [40, 128]$. A safe default value for 4K textures is $\rho = 96$ and for 1K or 2K textures is $\rho = 64$ at 12x compression ratios.

Generally, the optimal ρ is proportional to the redundancies in a texture. Usually, natural images and high resolution images have more redundant pixels and prefers a higher ρ. The optimal ρ also varies inversely with the required compression ratios. Using the defaults values under various tested circumstances produce results within 5% from optimal.

To set up the resolution of the primary and cascaded array, we use three information - uncompressed size of the texture in bytes denoted by B, expected compression denoted by e, and ρ. The size of our representation in bytes is given by:

$$\begin{aligned} B_{comp} &= 4N_p^2 + kN_c^4 \\ &= 4\rho^2 N_c^2 + kN_c^4, \text{ using } \rho = \frac{N_p}{N_c}. \end{aligned} \tag{3}$$

Note that $e = B/B_{comp}$; thus using equation 3, we can solve for N_c. However, we add some additional constraints. We require N_p, and N_c are integer multiples of 8, and 4 respectively to avoid memory alignment issues. Thus instead of directly solving the variable, we do a linear search in N_c that minimizes the difference

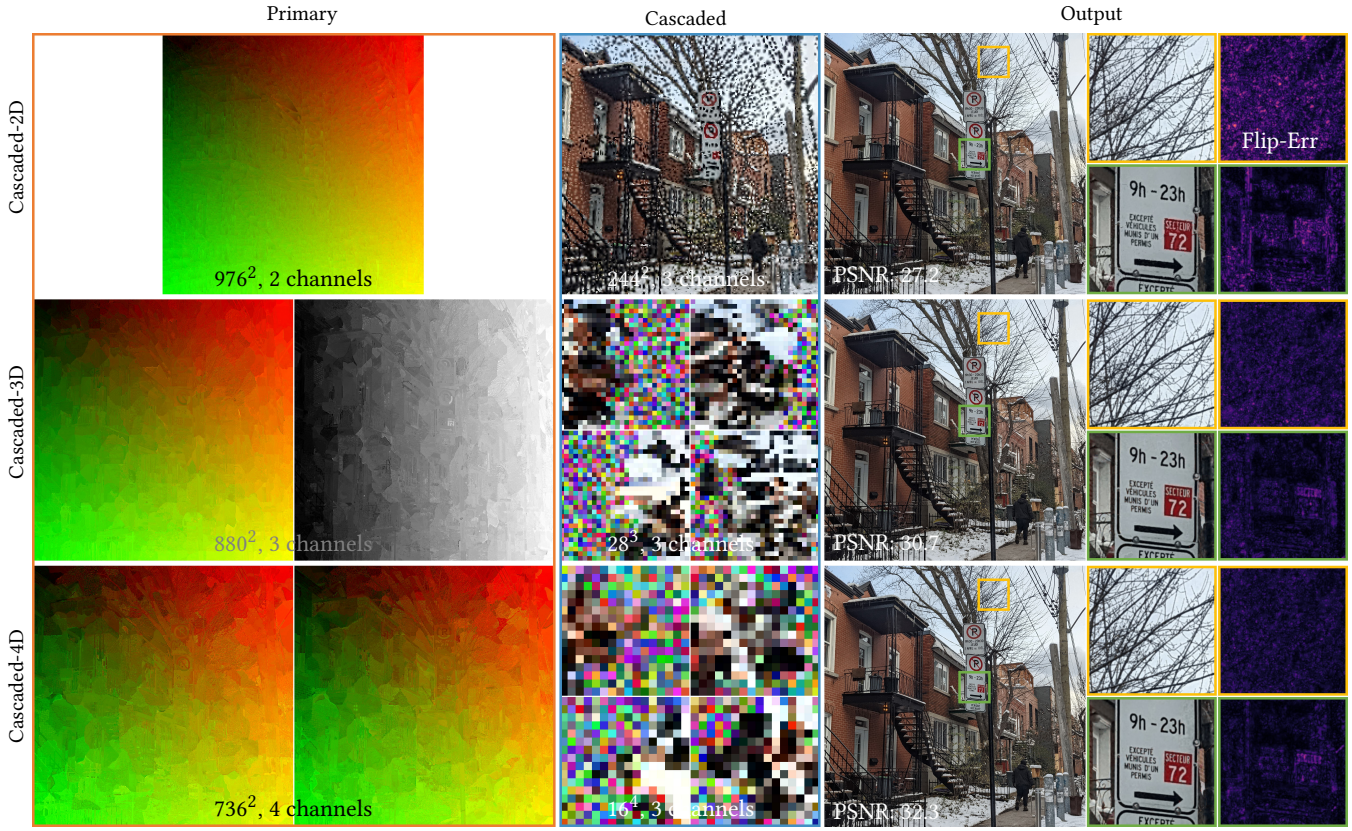

Figure 4: Figure visualizes the contents of the primary and cascaded arrays for varying (vertically) network configurations. All network compresses a 2k image by 6× using differentiable indirection. In case of 3D and 4D cascaded arrays, 2D slices of the multi-dimensional volume is visualized in the 2^{nd} and 3^{rd} row.

$e - B/B_{comp}$ while also satisfying all constraints.

Training and inference: We train the network using uv-color pairs and use stratified random sampling to generate the training uv coordinates. Each strata corresponds to a texel in the base texture. We obtain the target color values using a bi-linear sampler. We tested with VGG-19 [8] and SSIM losses which require 2D patches of texels to perform convolutions. However, the convolutional losses did not significantly impact quality but slows the training. We use MAE as loss and ADAM optimizer with 0.001 learning rate. For inference, we quantize both primary and the cascaded array to 8-bit. An interesting effect is that the cascaded array is often ≤ 1MB for 4K textures, which may fit in a lower tier cache.

Results and visualizations: Figure 4 shows the content of primary and cascaded arrays for 2D, 3D, and 4D network configurations. Figure 5 shows the output of various network configurations for a range of compression between 6× to 48×.

1.4 Neural texture sampling

Using uv-mapped textures in 3D scenes requires appropriate *texture filtering* to avoid aliasing. Aliasing occurs as the pixels on screen do not align one-to-one with the texels on a texture. The mapping is either one-to-many (minification-filtering) or many-to-one (magnification-filtering), depending on the size of the projected pixel footprint in the texture-space. In modern GPUs, filtering is performed inside hardware using a chain of mip-maps obtained from the base texture. At runtime, the appropriate mip-levels are selected based on the pixel-footprint and *tri-linearly* interpolated between adjacent mip-levels. More advanced filtering involves *anisotropic* filtering which not only takes into account the size of the pixel footprint but also its orientation in texture-space. Our goal is to approximate a *tri-linear* texture sampler using *differentiable indirection* without storing an explicit mip-chain.

Setup: Our texture sampler takes two input - a uv coordinate and an estimate of the pixel footprint. In real-time systems, the latter is computed using shader derivatives. Shader derivatives are *numerical finite difference* derivatives of a quantity w.r.t x (horizontal, called *ddx*) and y (vertical, called *ddy Direct 3D*) axis in screen space. These derivatives are generally hardware accelerated and computed in *pixel/fragment* shaders. For inference purposes, we collect the shader derivatives of the uv-coordinates as part of the *GBuffer* generation among many other parameters required for shading. We compute the pixel footprint as the magnitude of

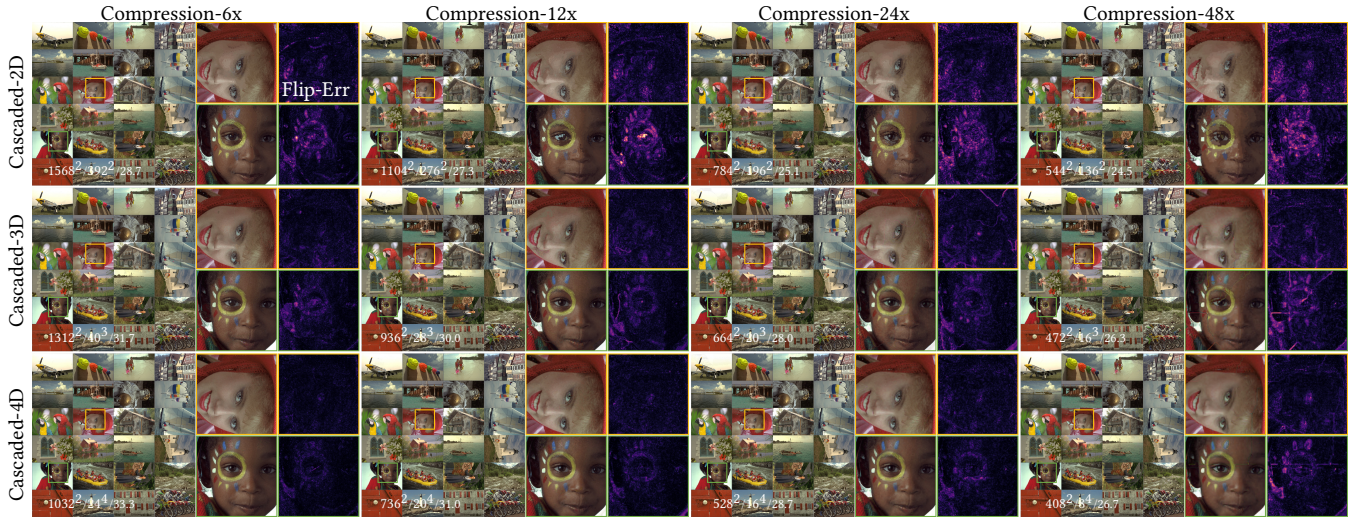

Figure 5: Figure shows a collection of natural images (as an atlas) under varying compression (horizontally) using different network architectures (vertically). All primary arrays are 2D arrays while the cascaded arrays are 2D, 3D, and 4D for the first, second, and third rows respectively. The atlas has a base resolution of 3K and the output of our 6×, 12×, 24×, and 48× compressed representations is shown along the horizontal axis. Resolution of the primary, cascaded array and the achieved PSNR is provided for all outputs. Images from ©Kodak [1] image suite are rearranged as an atlas.

the cross product between the of the two derivatives of the uv coordinates w.r.t horizontal and vertical axis in screen-space.

Our network consist of two primary arrays- one corresponding to the uv-input and the other corresponding to the pixel footprint input. A single cascaded array is used. The shape of the arrays are provided in the table 3.

For the cascaded array, N_c is the sides attached to *Primary-0* and N_{lod} is the side attached to *Primary-1*. We first estimate N_{lod} - setting an initial value $\approx log_2(N_{base})$, where N_{base} is the base texture resolution. We fine tune N_{lod} as hyper parameter search. The pseudo-optimal values are N_{lod} = 8, 12, 12 for 1K, 2k, 4k textures respectively. We set $N_{p1} = 4N_{lod}$. The total bytes required for our representation is thus

$$\begin{aligned} B_{comp} &= 3N_{p0}^2 + N_{p1} + kN_c^3 N_{lod} \\ &= 3\rho^2 N_c^2 + N_{p1} + kN_c^3 N_{lod}, \text{ using } \rho = \frac{N_{p0}}{N_c}. \end{aligned} \tag{4}$$

Rest of the values - N_{p0}, and N_c are estimated similar to section 1.3 with pseudo-optimal values values of ρ provided in table 4.

Table 3: Network configuration details for texure sampler.

Table Name	Shape/Resolution	Input	o/p channels
Primary-0	2D, $N_{p0} \times N_{p0}$	uv-coords	3
Primary-1	1D, N_{p1}	Pixel-footprint	1
Cascaded	4D, $N_{lod} \times N_c^3$	Primary-1 Primary-0	k

Training: To generate the target data for training, we recreate a proxy *tri-linear* texture sampler that mimics a *tri-linear* texture sampler in real-time 3D APIs such as *OpenGL* or *D3D*. The proxy sampler for data generation works by generating a mip-chain of the base texture and *tri-linearly* interpolate between the mip-levels according to pixel footprint. For training, we randomly sample the uv-coordinates using stratified-random sampling, similar to section 1.3. We also randomly sample the pixel footprint values $\in [0, 1)$, where 0 corresponds to most detailed and 1 to least detailed level-of-detail in the mip-chain. We pass the uv and pixel footprint values to our network and the proxy texture sampler and compare their output to train our network.

We ensure half of the random samples belong to LOD-0 by generating uniform random samples and raising it to the power n, where $n = -log(N_{base})/log(p)$. In our case, $p = 0.5$ - corresponding to 50% samples in LOD-0. We provide a proof in the next section.

Sampling LODs for training data generation: Our goal is to generate more training samples from more detailed and less samples from less detailed LODs. One way achieve this is to sample the

Table 4: Empirically obtained optimal resolution ratios (N_{p0}/N_c) for varying compression of 4K RGB and material texture for combined compression and sampling.

	Compression ratio			
	3x	6x	12x	24x
RGB (3-channels)	80	64	64	48
RGB, Normal, AO (7-channels)	128	96	96	72

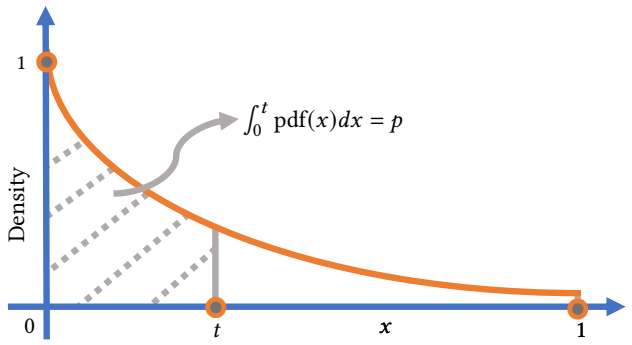

Figure 6: Figure illustrating the fraction of samples p below a threshold t as being equivalent to evaluating the *CDF* at t.

pixel footprint values from an exponential distribution, however, exponential distribution tends to put too few samples for less detailed LODs. The samples however, are better distributed by simply raising the samples collected from a uniform distribution to the power of n. In this section, we calculate the optimal value of n. We generate the pixel footprint samples (x_f) as

$$x_f = u^n, \; u \sim U(0,1), \tag{5}$$

where U indicates a uniform distribution. Let us assume we generate fraction $p(\neq \rho)$ of the total samples from pixel footprint value t or below. The *cumulative distribution function* of U is given as

$$\text{CDF}_U(u) = P(U \leq u) = u. \tag{6}$$

We calculate the *cumulative distribution function* of X_f as

$$\begin{aligned} \text{CDF}_{X_f}(x_f) &= P(X_f \leq x_f), \text{ using eq. 5} \\ &= P(U^n \leq x_f) \\ &= P(U \leq x_f^{\frac{1}{n}}), \text{ using eq. 6} \\ &= x_f^{\frac{1}{n}}. \end{aligned} \tag{7}$$

From figure 6, note that fraction of samples p below a threshold t as being equivalent to evaluating the *CDF* at t. Therefore,

$$CDF_{X_f}(t) = t^{\frac{1}{n}} = p \text{ or } n = log_p(t). \tag{8}$$

In our application, $t = 1/N_{base}$, and $p = 0.5$.

Optional – Exponential sampling: We perform a similar analysis as last section for selecting the correct parameter (λ_e) for sampling the footprints from an exponential distribution given by $exp(-\lambda_e x)$.

$$x_f = -\frac{1}{\lambda_e} ln(u) \tag{9}$$

$$\text{CDF}_{X_f}(x_f) = 1 - e^{-\lambda \cdot x_f} \tag{10}$$

$$\lambda_e = -\frac{ln(1-p)}{t} \tag{11}$$

Even with appropriate parameters, the low resolution LODs receive too few samples, hence not used for our application.

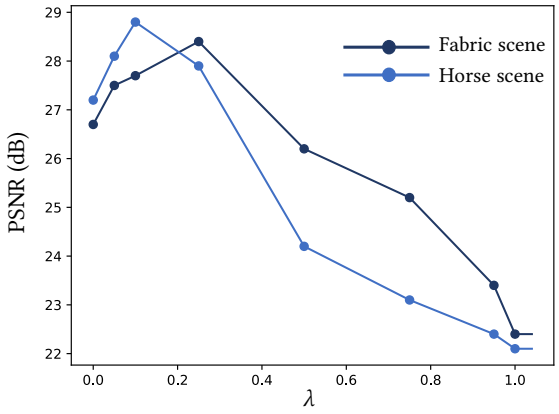

Figure 7: Plot illustrating variation in quality with increasing λ for two scenes. λ indicates the fraction of shading loss used for training as discussed in 1.5

1.5 Optimized shading pipeline

The section aims at improving the quality of the final render by making the neural texture sampler aware of the approximate *BRDF*. We do so by enforcing two loss functions. First loss compares the output of our neural texture sampler with the output of the reference proxy texture sampler similar to section 1.4.

The second loss compares the output of neural texture sampler through our approximate *Disney BRDF* with the corresponding target reference. To generate the reference, we collect 16 uniformly distributed samples across the pixel footprint, and aggregate the samples post evaluation through the reference Disney Brdf; process referred as *appearance filtering* in the literature. However, to minimize aliasing at large pixel footprints without increasing sample count, we also pre-filter each individual sample with tri-linear filtering. We assume axis-aligned pixel footprint. For backpropagation, we use our pre-trained neural Disney BRDF as a differentiable fixed function layer - i.e. we freeze the contents of the pre-trained decoder arrays in our neural BRDF while allowing gradients to back-propagate through the decoder arrays.

We blend in the two losses using a hyper-parameter $\lambda \in [0, 1]$, where $\lambda = 0$ indicates purely the first loss while 1 indicates pure second loss. We notice some interesting details.

Optimizing our neural sampler directly using the shaded samples (i.e. $\lambda = 1$) results in training instability. Our test with a simple diffuse BRDF also indicates similar issues. We believe there are two main issues. First, the optimization is underconstrained i.e. different combinations of albedo, normal, AO may yield the same shaded result. Second, the unbounded non-linearities in the Disney BRDF may also cause the training to diverge. We solve the first issue by adding a regularization term, in our case we do so by setting $\lambda < 1$. Setting λ strictly less than 1 essentially uses the first loss as regularization. We fix the second issue by clipping the gradients backpropagating through the non-linear metallic component of the BRDF in ±0.1 range. A learning rate scheduler may also improve convergence in this case. A variation in quality due to increasing λ is shown in plot 7. For the *Horse scene*, and the *Fabric scene* in

Table 5: Empirically obtained optimal resolution ratios (N_p/N_c) for varying parameter size in the SDF task.

	Parameter size					
	96MB	48MB	24MB	16MB	12MB	6MB
Piano	2.56	2.71	2.08	2.81	2.56	2

figure 1, 13 of the main paper shows an improvement in quality with a $\lambda = 0.1, 0.25$ respectively.

1.6 Signed Distance Fields

In the SDF case, we use an Adam optimizer paired with a learning rate scheduler. Also, as described in the main paper, our training samples mostly consist of near surface samples and use an MAE loss. We quantize the distances to ±1 and learn a truncated SDF representation. For faster convergence, we also initialize the cascaded 3D-grid with an approximate SDF representation. At each voxel of the cascaded array, we use a Kd-Tree to lookup the nearest sample in the training set and set the voxel's content as the corresponding truncated distance. We notice, pre-initializing the cascaded array minimizes artifacts. The primary 3D-array is initialized as uniform ramp as usual. We set up the array resolutions similar to section 1.3 and use $\rho \in [1.5, 2.5]$ as shown in table 5. During inference we quantize both the primary and cascaded arrays to 8-bit.

1.7 Radiance field compression

Similar to section 1.6, we initialize the cascaded 3D-array with the a scaled version of the target grid extracted from the *Direct Voxel* technique. Such initialization is optional but improves training convergence. However, unlike section 1.6, we only quantize the primary array to 8-bit during inference. ρ values are provided in table 6. We use standard MAE loss and ADAM optimizer. We use the *Direct Voxel* code base to train the initial grid-representation with some minor modification – we remove the *MLP* used at the output of the RGB field and replace it with *positional encoding* (*PE*). The output of the *PE* is summed and passed through a *sigmoid* function to generate the final output. We note the *Direct Voxel* code base already uses *PE*, we thus only replace the *MLP* with a summation. Default uses 12 channel output from *RGB* grid which is mixed in with *PE* view-directions.

Extracting the grid representation and replacing with our primary/cascaded network is challenging as the gird representation is deeply embedded and requires several modifications in the inference code. One needs to carefully match the output of the grid representation for a given input; as such it is important to use the same sampling recipe used in the *Direct Voxel* technique to generate the target data with. We also carefully align the data to initialize the cascaded array such that the output is as close to the reference as possible. A misaligned initialization may cause slower training.

Table 6: Empirically obtained optimal resolution ratios (N_p/N_c) for varying compression ratios in NeRF task. Uncompressed density and RGB grids have a resolution of $256 \times 256 \times 256$ with 1 and 12 channels respectively.

	Compression Ratio					
	5x	10x	25x	50x	75x	100x
Density Grid (1 channel)	3.57	3.8	2.69	1.97	2.82	2.51
RGB Grid (12-channels)	5.05	3.42	4.14	4.53	3.58	5.74

1.8 Guidelines for training stability

While *differentiable indirection* mostly works out of the box, we present some additional issues to lookout for. One common issue is the magnitude of the gradient during training; very large gradients may cause training instabilities. When training with unbounded non-linear output, such as glossy *BRDF*s, it may be useful to clip the gradients backpropagating to the array cells. Note that gradient backpropagating to the primary array depends on the contents of the cascaded array. Thus pre-initializing the cascaded array with approximately correct values reduces the magnitude of gradients backpropagating to the primary array. Also note the gradients backpropagating to primary array is proportional to the resolution of the cascaded array. Thus in case of large arrays, it may be useful to monitor the gradient amplification through array layers, crucially when working with multi-level or deep indirection.

REFERENCES

[1] Kodak corporation. 1999. Kodak Lossless True Color Image Suite. https://r0k.us/graphics/kodak/

[2] E. Delp and O. Mitchell. 1979. Image Compression Using Block Truncation Coding. *IEEE Transactions on Communications* 27, 9 (1979), 1335–1342. https://doi.org/10.1109/TCOM.1979.1094560

[3] William Donnelly. 2005. Per-Pixel Displacement Mapping with Distance Functions. https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-8-pixel-displacement-mapping-distance-functions

[4] Tzu-Mao Li. 2022. UCSD CSE 272 Assignment 1: Disney Principled BSDF. https://sayan1an.github.io/disneyLi.html

[5] Wenzel Jakob Matt Pharr and Greg Humphreys. 2018. Physically Based Rendering:From Theory To Implementation, 3rd Edition. https://pbr-book.org/3ed-2018/Reflection_Models

[6] Wenzel Jakob Matt Pharr and Greg Humphreys. 2018. Physically Based Rendering:From Theory To Implementation, 3rd Edition. https://pbr-book.org/3ed-2018/Monte_Carlo_Integration/2D_Sampling_with_Multidimensional_Transformations#CosineSampleHemisphere

[7] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. 2012. Adaptive Scalable Texture Compression. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 105–114.

[8] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[9] László Szirmay-Kalos and Tamás Umenhoffer. 2008. Displacement Mapping on the GPU — State of the Art. *Computer Graphics Forum* 27, 6 (2008), 1567–1592. https://doi.org/10.1111/j.1467-8659.2007.01108.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01108.x

[10] Joey De Vries. 2023. Normal mapping. https://learnopengl.com/Advanced-Lighting/Normal-Mapping

[11] Joey De Vries. 2023. PBR Shading. https://learnopengl.com/PBR/Theory

[12] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. 2007. Microfacet Models for Refraction through Rough Surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Grenoble, France) *(EGSR'07)*. Eurographics Association, Goslar, DEU, 195–206.

[13] Junqiu Zhu, Sizhe Zhao, Yanning Xu, Xiangxu Meng, Lu Wang, and Ling-Qi Yan. 2022. Recent advances in glinty appearance rendering. *Computational Visual Media* 8, 4 (01 Dec 2022), 535–552. https://doi.org/10.1007/s41095-022-0280-x