Efficient Graphics Representation with Differentiable Indirection

Sayantan Datta
sayantan.datta@mail.mcgill.ca
McGill Univeristy
Montreal, Canada
Meta Reality Labs
Redmond, USA

Carl Marshall
Zhao Dong
Zhengqin Li
csmarshall@meta.com
zhaodong@meta.com
zhl@meta.com
Meta Reality Labs
Redmond, USA

Derek Nowrouzezahrai
derek@cim.mcgill.ca
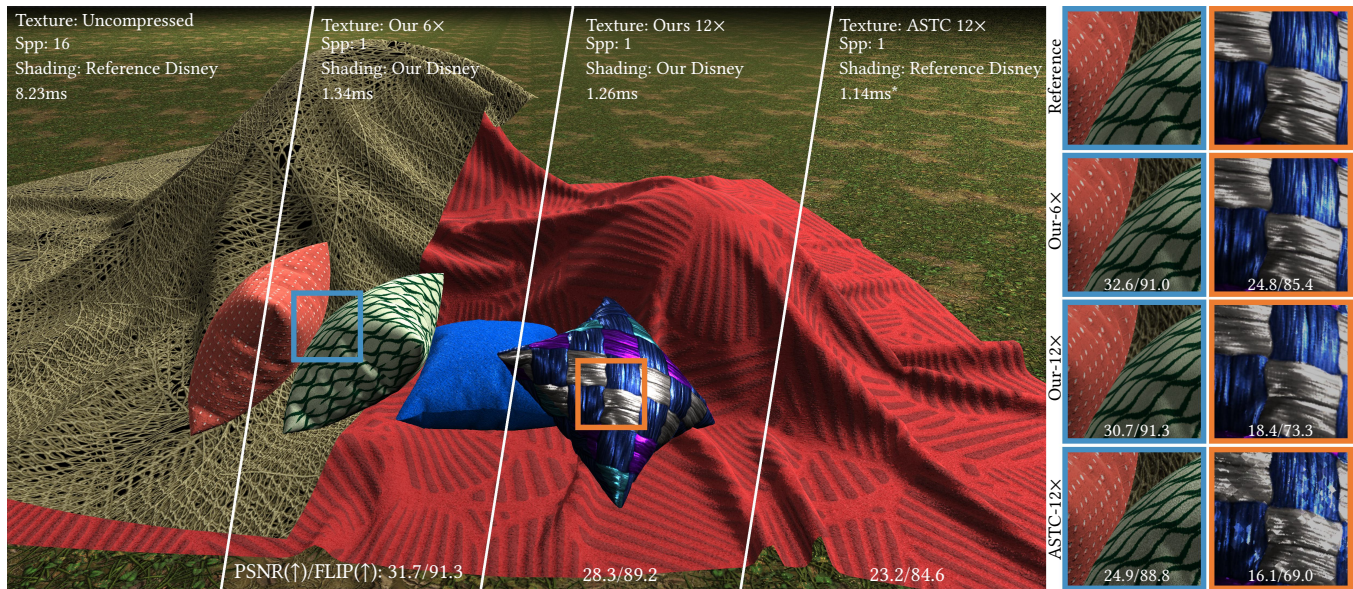McGill University
Montreal, Canada

Figure 1: Figure shows the use of *differentiable indirection* for texture compression/sampling and parametric shading at 4K screen resolution. Our technique relies on a few linearly interpolated indirect memory lookups and applies to a wide range of tasks in the graphics pipeline including distance and radiance field compression.

ABSTRACT

We introduce *differentiable indirection* – a novel learned primitive that employs differentiable multi-scale lookup tables as an effective substitute for traditional compute and data operations across the graphics pipeline. We demonstrate its flexibility on a number of graphics tasks, i.e., geometric and image representation, texture mapping, shading, and radiance field representation. In all cases, *differentiable indirection* seamlessly integrates into existing architectures, trains rapidly, and yields both versatile and efficient results.

CCS CONCEPTS

• **Computing methodologies** → *Rasterization*; *Volumetric models*; *Image compression*.

KEYWORDS

Differentiable LUT, Memory Indirection, Multi-modal Representations, Efficient Neural Alternatives.

1 INTRODUCTION

Neural primitives are the fundamental building block of neural networks and used for a variety of purposes in graphics applications, such as appearance capture [Zhang et al. 2022], shading [Schied

and Kaplanyan 2022], radiance caching [Müller et al. 2021], view-synthesis [Mildenhall et al. 2020], and shadows [Datta et al. 2022]. Having *efficient* neural primitives is vital due to their impact on latency, power, and training speed. Achieving high runtime performance with neural primitives is essential to the adoption of neural networks in real-time and low-power applications, such as AR/VR.

We introduce a simple primitive with excellent runtime characteristics, featuring *low compute FLOPs*, *minimal memory reads per query*, and *a compact parameter size*. Many neural networks rely on *multi-layer perceptrons* (*MLP*) due to their appeal as universal function approximators; however, *MLP* layers are often the most computationally expensive component of a network and scale quadratically (both in FLOPs and bytes transferred) with quality due to large matrix operations [Schmidhuber 2015]. Conversely, combining memory grids with fixed function non-linearities such as *Spherical Harmonics (SH)* [Fridovich-Keil et al. 2022] or *ReLUs* [Karnewar et al. 2022] reduces compute and memory transfer but incurs a large parameter cost. Our novel primitive – *differentiable indirection* – strikes a balance across these criteria and useful for a variety of data compression and compute representation tasks. It is compatible with any differentiable logic, such as *MLPs* or fixed function approaches, but significantly reduces or even eliminates reliance on *MLPs*. Notably, all of our examples are *MLP*-free, thereby eliminating the need for specialized hardware [Nvidia 2019] acceleration in real-time applications. *Differentiable indirection* draws its expressive power solely from memory indirections and linear interpolation. This approach aligns well with the emerging computing paradigm of *compute in memory* [Lin et al. 2022; Wang et al. 2021], which departs from traditional *von Neumann* model that *MLPs* are modelled on. We apply *differentiable indirection* to various tasks in the (neural) graphics pipeline, showcasing its potential as an efficient and flexible primitive for improving runtime efficiency.

2 RELATED WORK

Neural primitives serve as the fundamental building blocks for modern neural techniques. We provide an overview of existing neural primitives and explore their applications in graphics.

MLP architectures provide a compact implicit representation that seamlessly scales up to higher dimensional inputs, such as signed distance field [Park et al. 2019], neural radiance field [Mildenhall et al. 2020], and neural BRDF [Bi et al. 2020; Boss et al. 2021a,b; Zhang et al. 2021]. They trade parameter size for compute, memory bandwidth, and a relatively longer training time as each training example affects all network weights. Even small MLPs (2 layer deep, 64 unit wide) are computationally and memory-intensive, requiring thousands of FLOPs and bytes transferred per query. In contrast, our differentiable indirection relies on memory indirections and interpolations, resulting in reduced computational demands.

Grid-based representations explicitly store trainable parameters on a regular grid [Chabra et al. 2020; Karnewar et al. 2022] or a tree [Sara Fridovich-Keil and Alex Yu et al. 2022; Takikawa et al. 2021] and then retrieve them at run-time using the input coordinate as key. The stored features are processed further using a non-linearity, such as *ReLU* [Karnewar et al. 2022] or *SH* [Fridovich-Keil et al. 2022]. While suitable for fast, localized updates, explicit representations

tend to have large parameter sizes and has difficulty scaling up to higher dimensional inputs.

Recent works combine *MLP* and grid representations to balance between memory and computational cost, achieving complex neural shading [Kuznetsov et al. 2021; Zeltner et al. 2023], neural material texture compression [Vaidyanathan et al. 2023], and efficient and high-quality neural radiance field rendering [Chen et al. 2022; Müller et al. 2022; Sun et al. 2022; Takikawa et al. 2022a]. Particularly, *instant-NGP* combines *multi-resolution hash encoding* [Müller et al. 2022] with pyramid of latent features, demonstrating to be effective for a wide-range of reconstruction and compression applications. Similarly, we show the effectiveness of our primitive in the broader context of neural rendering. However, our technique is also effective as a standalone unit without an *MLP*. This provides unparalleled efficiency advantages, making it particularly suitable for low-power applications like neural shading on mobile devices.

3 OVERVIEW

Differentiable indirection (*DIn*) is a flexible and powerful tool for representing compute/data problems in both the modern and neural graphics pipelines. *DIn* is similar to a pointer indirection - we query a memory location that contains a pointer to a secondary location containing the final output. However, we also make pointer indirection differentiable, hence *differentiable indirection*. Our algorithm learns the pointer values stored as an array using gradient descent. *DIn* is a flexible and simple-to-integrate representation, as demonstrated in our applications to geometric and image representation, texture mapping, shading, and radiance field compression.

Our technique in its simplest form requires two arrays – a **primary** array and a **cascaded** array. The article uses the same terminology throughout. Lookup into the primary array returns a pointer into the cascaded array. The corresponding location in the cascaded array contains the output. Figure 2 shows a visual representation of *differentiable indirection* where the primary and the cascaded arrays are highlighted in orange and blue respectively.

Differentiable Arrays. We introduce fully differentiable arrays - the main building block of our technique. A key requirement of our technique is that the arrays are not only differentiable w.r.t. to the content of each cell but also w.r.t. its indices or the uv-coordinates. The latter allows the gradients to backpropagate though the cascaded array to learn the pointer values stored in the primary array. The differentiability is achieved by linearly interpolating the array cells. For an input coordinate $\mathbf{x} \in [0, 1)^d$, the coordinate is scaled by the array resolution (N) and rounding down $\lfloor \mathbf{x} \cdot \mathbf{N} \rfloor$ and up $\lceil \mathbf{x} \cdot \mathbf{N} \rceil$ - forming a d-dimensional voxel encapsulating $\mathbf{x}$. The vertices of the voxel is interpolated according to the distance of the query point from the corners to produce the output (o) given by:

$$\mathbf{o} = \sum_{i=0}^{2^d-1} \alpha_i \mathcal{F}(\mathbf{c}[i]), \qquad (1)$$

where α_i, and $\mathbf{c}[i]$, are the interpolation weights, and the array cell content at the voxel-vertex i respectively. We pass the cell contents through an additional non-linearity $\mathcal{F}$ with some specific characteristics as discussed in the next section. The gradients w.r.t.
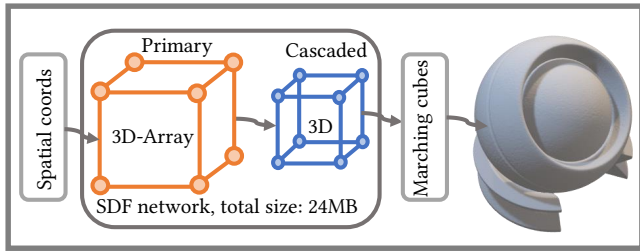
Figure 2: *SDF* representation using *DIn*. Primary and cascaded resolutions are $200^3 \times 3$ and $64^3 \times 1$ respectively.

Figure 3: Visualizing learned primary (2D) and cascaded (2D) arrays, compressing a 2k image by 6×. A 2D-primary and 4D-cascaded produce better results in practice.

the input coordinate **x** and cell content **c**[i] is given by

$$\frac{d\mathbf{o}}{d\mathbf{x}} = \sum_{i=0}^{2^d-1} \frac{d\alpha_i}{d\mathbf{x}} \mathcal{F}(\mathbf{c}[i]), \text{ and } \frac{d\mathbf{o}}{d\mathbf{c}[i]} = \alpha_i \frac{d\mathcal{F}}{d\mathbf{c}[i]} \qquad (2)$$

respectively. The gradients are plugged into *autodiff* framework such as PyTorch [Paszke et al. 2017], enabling backpropagation though arrays. Prior techniques - *Multi Resolution Hash Encoding*, abbreviation *MRHE* [Müller et al. 2022] and *ReLU Fields* [Karnewar et al. 2022] only compute gradient w.r.t. cell contents **c**[i].

Non-linearity. We apply a periodic non-linearity $\mathcal{F}$ to the primary array and an optional periodic/aperiodic non-linearity to the cascaded array. The purpose of the periodic non-linearity is to bound and continuously *wrap-around* the array content for use as an input in the next layer. We test two periodic functions - a sinusoid given by $(1+\sin(nx))/2$ and a non-negative triangle wave with a period 2 with peak output 1 at input 1. We use the triangle wave for all cases except *Disney-BRDF* where we use the sinusoid. Note that the non-linearity $\mathcal{F}$ is applied before interpolation, thus can be removed during inference and baked directly into the array cells for improved efficiency. The output is also bounded to $[0, 1)$ when using a periodic non-linearity, lending the opportunity to quantize the array values to 8/16-bit for inference without significant impact on quality; a property we utilize heavily for all tasks.

Initialization. *Differentiable indirection* is thus a cascade of multi-dimensional differentiable arrays. We initialize the primary array using a linear ramp resembling a standard uv-map in 2D or a similar analog in higher dimension. Initially the primary array is an identity map between input and output. Gradient descent simultaneously distorts the identity map in the primary and updates the values stored in the cascaded. Linear interpolation puts an implicit constraint on the values the primary array may accommodate. If we imagine the uv-map as a fabric, gradient descent is only allowed to locally wrinkle the fabric. The effect is illustrated in figure 3. The choice of initialization for the cascaded array is application specific.

4 APPLICATIONS

We demonstrate the applications of *differentiable indirection* across various stages of the graphics pipeline – starting with geometric representations, followed by examples in deferred shading such as image and texture compression, and parametric shading. Finally, to compress implicit representation such as neural radiance field.
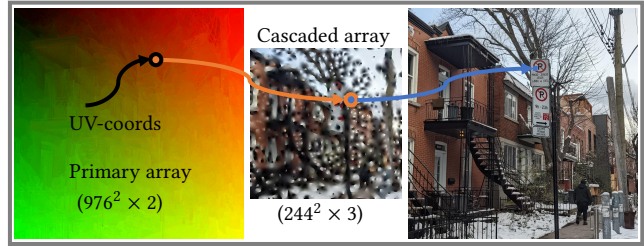
4.1 Compact geometric representation

Our technique is easily adapted to implicit geometry representation task using a *Signed Distance Function* (*SDF*) representation. *SDFs* volumetrically encode the zero-level set of a spatial 3D shape function. *DIn* readily applies to *SDF* representation, compressing the volumetric information in the 3D arrays. Figure 2 shows the primary array queried by a spatial 3D-coordinate that points to the cascaded array containing the signed distances from the zero-level set; we train one network per *SDF* and use marching cubes [Lorensen and Cline 1987] for illustrative surface reconstruction.

Implementation details. We generate training samples pairs – a position and its corresponding signed distance using an *SDF* dataset generator [Takikawa et al. 2022b]. We preferentially sample points closer to the surface zero-level set [Takikawa et al. 2021]; one billion samples from near the surface and 20 million uniformly distributed over the volumetric domain. We use 100 million near-surface samples to compute test time error statistics. While *MRHE* proposes *MAPE* as the loss function, this results in many "floater" artifacts near the surface; we believe this is due to *MAPE* over-emphasizing on-surface sample importance at a cost of distorting the distance field slightly off-surface. We solve this problem by quantizing signed distances to ±1 a.k.a. *Truncated SDF* and applying an *MAE* loss. In a grid based technique like ours, only having two discrete values allows the transition-boundary (representing a surface) to adapt more freely. Figure 2 illustrates a 3D shape generated using our training methodology. We include more results in section 6.0.2 and training details in supplemental section 1.6.

4.2 Compact texture and image representations

We apply *differentiable indirection* to real-time texture and natural image (de-)compression, and filtered texture sampling.

4.2.1 Compact image representation. In this task we query the primary array with 2D uv-coordinates (figure 3). The cascaded array encodes the corresponding color value at the uv location. Figure 3 visualizes the learned arrays' contents – note the primary array favors high frequency details as a pseudo-distorted uv-map. We however recommend alternative network configurations than those in figure 3 which produce improved results, as described next.

Implementation details. We use a 2D 4-channel array as primary and a 4D 3-channels array containing RGB values as the cascaded. For *PBR* textures, we extend the number of channels in the cascaded
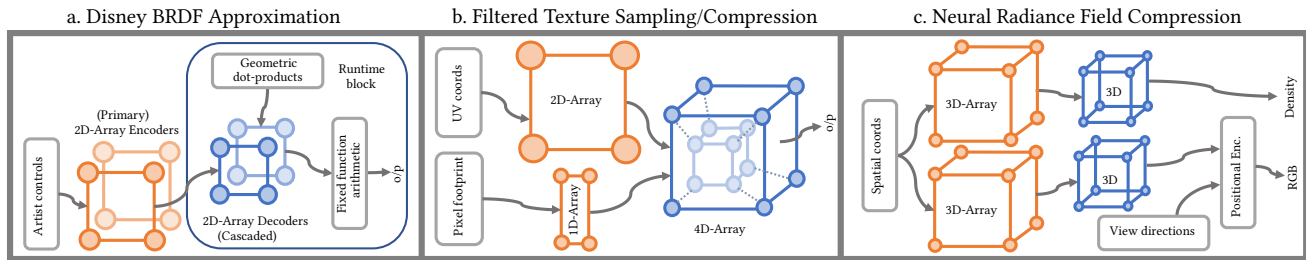
Figure 4: From left to right, parametric shading, compact texture sampling, and radiance field based on *differentiable indirection*.

array to include various shading parameters. During training, the network maps the queried uv-coordinates to the corresponding color/parameter value. A higher dimensional (> 2D) cascaded array results in better compression quality. For a mixture of textures containing natural images and several *PBR* materials from *Adobe Substance*, we noticed a 15% improvement in PSNR moving from 2D to 3D and 3% improvement from 3D to 4D. We generate the training uv-coordinates using stratified random sampling where each strata corresponds to a texel in the base texture. Target color values are obtained with bi-linear sampling of the reference image at the queried uv-coordinates. Details in supplemental section 1.3.

4.2.2 Neural Texture Sampler. For shading applications, we also need to account for texture filtering [Williams 1983] based on the projected pixel footprint onto a geometry surface. With minor modifications to the previous compression-only network, we can additionally treat filtered texture sampling. Figure 4(b) shows our texture sampling/filtering network configuration, requiring two inputs – a uv-coordinate, and a pixel footprint magnitude, both of which are readily available in modern interactive renderers.

Implementation details. We use two primary arrays - a 2D array with 3 channels for input uv-coordinates, and a 1D array with 1 channel for pixel footprint. The output of the two primary arrays is concatenated and used as the input for the cascaded 4-D array. The network approximates a *trilinear texture sampler* similar to those available on modern GPUs but unlike GPUs, we do not store an explicit mip-chain. We soft-emulate a GPU texture sampler to generate our target data. We train our network on random uv-samples

and pixel-footprint values with corresponding target generated with the emulated trilinear sampler. Details in supplemental 1.4.

The cost of evaluating the new network only increases marginally compared to section 4.2.1 while providing higher quality per sample for shading tasks, especially for pixels with large footprint at grazing angles. The effect is illustrated in Figure 5.

4.3 Efficient parametric shading models

So far, we have seen the applications of our technique in data representation. This section introduces parametric shading as compute approximation task. We use our technique on two different *BRDFs* - a simple *isotropic GGX* and a more complex *Disney BRDF*.

4.3.1 Isotropic GGX approximation. *Isotropic GGX* is a popular *BRDF* used for specular shading, expressed analytically as $D(h_z, \alpha_h) = \alpha_h^4 \cdot (1 + (\alpha_h^4 - 1) \cdot h_z^2)^{-2}/\pi$. This application approximates the analytic expression using *DIn*. We use the input parameters h_z, α_h as the input coordinates to the primary array (figure 6) and compare the output of the cascaded array with the corresponding output of the analytic expression. The example serves as a benchmark for comparing various neural primitives in section 6.0.1.

4.3.2 Principled Disney BRDF approximation. We approximate the *Disney BRDF* using *DIn*, retaining all artist controls with negligible impact on final quality (figure 11) while also improving upon the evaluation efficiency (table 1) compared to the reference analytic implementation. We follow the *Principled Disney BRDF* implementation reference [Li 2022] to generate our training data. Excluding
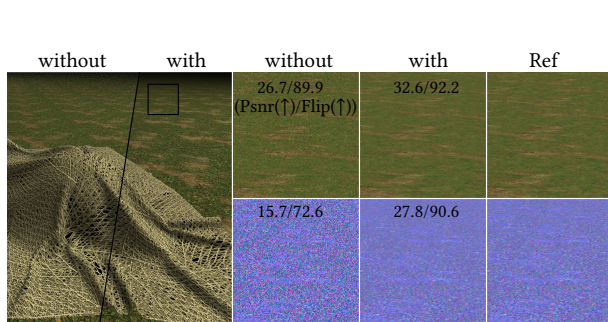

Figure 5: Comparing texture compression with and without pixel footprint. Without the footprint information, the output at grazing angle is noisy as shown in the cutouts.
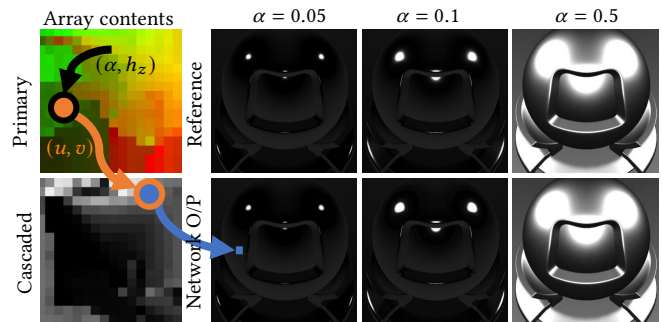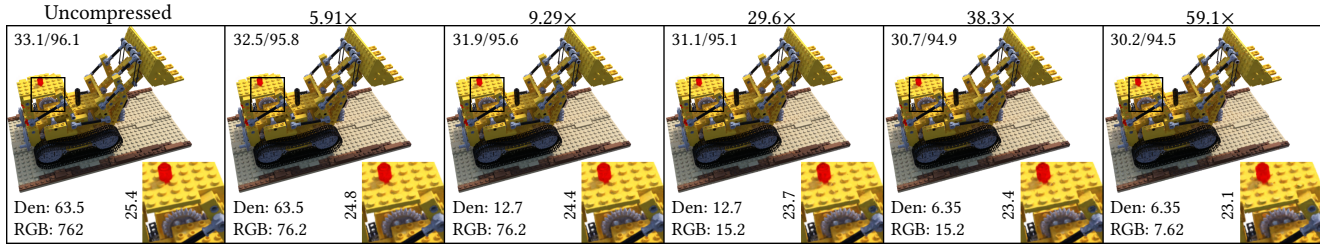

Figure 6: Visualization of the learned primary and cascaded arrays for *isotropic GGX* approximation on left and the resulting rendered output (average 40dB PSNR) on right.

| Uncompressed | 5.91× | 9.29× | 29.6× | 38.3× | 59.1× |

| 33.1/96.1 | 32.5/95.8 | 31.9/95.6 | 31.1/95.1 | 30.7/94.9 | 30.2/94.5 |

| Den: 63.5 RGB: 762 | Den: 63.5 RGB: 76.2 | Den: 12.7 RGB: 76.2 | Den: 12.7 RGB: 15.2 | Den: 6.35 RGB: 15.2 | Den: 6.35 RGB: 7.62 |
| 25.4 | 24.8 | 24.4 | 23.7 | 23.4 | 23.1 |

Figure 7: Radiance field compression with *DIn*. Top left shows average PSNR(↑)/FLIP(↑) w.r.t. all images in test set for varying compression ratios. Bottom left shows the size (in MBs) of the Density and RGB fields. PSNR(↑) for the cutout is provided nearby. Typical dimensions for the density, and RGB grid at 9.3× compression are $160^3 \times 3/64^3 \times 1$, and $280^3 \times 3/64^3 \times 12$ respectively.

glass/transmission term, the reference *BRDF* uses 10 artist controllable parameters and seven geometric dot-products as input. It is challenging to handle a high-dimensional input while also being efficient. While hypothetically possible, 17D arrays would be prohibitively expensive in practice. Other primitives using *MLPs* such as *MRHE* can scale up with higher dimensions. However, such primitive require large *MLPs* to attain desirable results here and do not improve upon the efficiency of reference evaluation.

We use a divide-and-conquer approach to partition the task into several components, leveraging the available domain knowledge in this setting. We start by factoring out the albedo ($\mathcal{A}$) from final *BRDF* output – i.e., as referenced in equation 19 in [Li 2022] and rewrite the reference *BRDF* equation as:

$$\text{Disney}(\mathbf{x}, \mathcal{A}) = \frac{\mathcal{A}}{l} p(\mathbf{x}) + q(\mathbf{x}), \quad (3)$$

where $\mathbf{x} \in [0, 1)^{17}$ are the control parameters, and l is the luminance computed as a weighted sum of albedo-RGB channels according to $(0.2126, 0.7152, 0.0722)$. The p and q terms are single channel positive scalars obtained directly from our factorization. Note that the learnable quantities p, q do not learn any color information, instead we modulate the albedo with the learned parameters. This is crucial for reducing color bleeding in the final output. We further refactor p and q according to

$$\begin{aligned} p(x) &= c_d D_d + c_{m0} D_m + c_{s0} \\ q(x) &= c_c D_c + c_{m1} D_m + c_{s1}, \end{aligned} \quad (4)$$

where D_d, D_m, and D_c are the *Disney-diffuse, Disney-metallic, and Disney-clearcoat* distribution (equation 5, 8, and 12 in [Li 2022]). The remaining c_* terms follow naturally from the factorization. We first approximate each D_* and c_* term independently using *DIn*; the primary aim here is to understand the functional space required to approximate the terms accurately. Once we have the appropriate functional space, we find similar lookups and merge them into fewer indirection pairs. Figure 4(a) illustrates the resulting network architecture. Our network also leverages the fact that the artist control parameters can be encoded into a latent vector (separately and completely offline). As such, the primary arrays are also encoders while the cascaded arrays are runtime decoders. We discuss the efficiency advantages of our approach in section 6.0.6.

4.3.3 Optimized shading pipeline. We improve the quality of our final rendered (figure 1, 13) output using an end-to-end optimization of the shading pipeline. Instead of training the texture sampler and

the *Disney BRDF* independently, the goal is to make our sampler aware of the learned *BRDF*, thus improving quality. We do so by training the texture sampler with an additional regularization term that compares the final rendered output through the learned *BRDF*. The extra complexity only affects the training pipeline while the networks and inference pipeline essentially stay the same. More details in supplemental section 1.5.
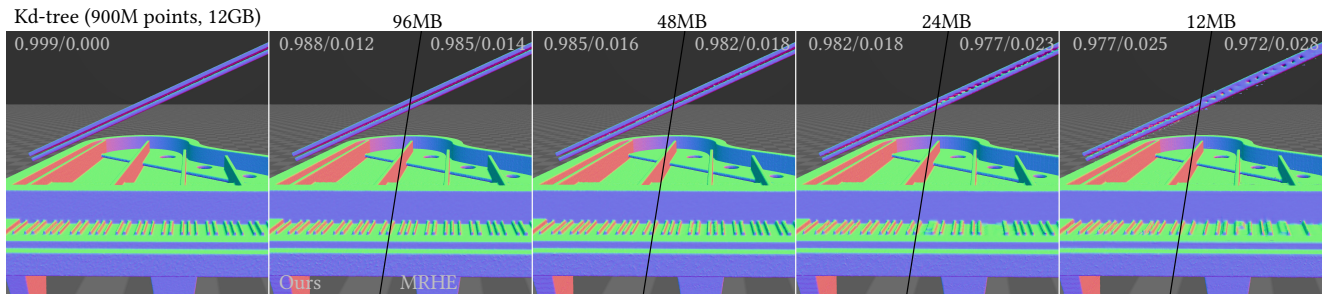
4.4 Compact radiance fields

NeRF volumetrically represents a scene as a 5D spatio-directional function whose outputs are spatial density and view-dependent emitted radiance. While the original *NeRF* [Mildenhall et al. 2020] uses a deep neural network to represent the density and radiance, subsequent versions obtain better quality, and improved training using coordinate networks based on 3D data-structures. They combine regular grid [Karnewar et al. 2022] or a tree [Sara Fridovich-Keil and Alex Yu et al. 2022] with fixed function non-linearity such as *SH* or *ReLU*. Recent state of the art *Direct Voxel* [Sun et al. 2022] achieves high quality representation by adaptive scaling of the voxel grid resolution and fine tuning the representation at each update. However, the resulting voxel grid is enormous, requires hundreds of *megabytes* in parameter space. We follow up on their work and improve the compression of their voxel representation by an order of magnitude without losing significant details (figure 7).

Implementation details. We extract the pre-trained density and view-dependent radiance/RGB grid from the *Direct Voxel* technique and apply *DIn* to compress the volumes. We use a multi-head network, as shown in figure 4(c) for the density and RGB fields. The networks use a spatial 3D coordinate as input to the two primary arrays. The output of the cascaded arrays are trained against trilinearly interpolated values from the corresponding target grids. We provide more details in supplemental section 1.7.

5 GENERAL IMPLEMENTATION DETAILS

The section provides general implementation details across tasks while more accurate task-specific details are provided in the supplemental. We train our networks using vanilla PYTORCH without special optimizations and use 32-bit *full-precision* arithmetic for backpropagation and data generation. We use the *ADAM* optimizer with a learning rate of 0.001 and an *MAE* loss. An important hyperparameter for training is $\rho = N_p/N_c$ – the ratio of the length of one side of the primary array (N_p) to the cascaded array (N_c).

Kd-tree (900M points, 12GB)   96MB   48MB   24MB   12MB

0.999/0.000   0.988/0.012   0.985/0.014   0.985/0.016   0.982/0.018   0.982/0.018   0.977/0.023   0.977/0.025   0.972/0.028

Ours   MRHE

Figure 8: Comparing *DIn* (left) with *MRHE* (right) at equal parameter count for *SDF* representation. *MRHE* requires 6 extra memory lookups and > 512 extra FLOPs to achieve similar results. Metrics IoU(↑)/MAE(↓) are measured on uniform near-surface *SDF* samples. Typical resolution of the primary/cascaded grid at 96MB is $316^3 \times 3/96^3 \times 1$.

We set pseudo-optimal values through a hyperparameter search in each setting. In all cases, the optimal ρ is greater than 1. This often results in the cascaded array being much smaller in size (raw bytes) compared to the primary array. In a real-time environment, the access pattern to the primary array is more coherent compared to the cascaded array, but the cascaded array is also much smaller – a potentially-exploitable property in caching hardware.

Another advantage when ρ is greater than 1 is in quantization: the cascaded array may contain signed or unbounded values, rendering the effective use of quantization cumbersome and additionally ineffective given the smaller array size; however, for the primary arrays, we quantize values to 8-bits after applying the non-linearity $\mathcal{F}$ to array cells. Quantization either lowers memory size or improves resolution for the primary array. A single level grid, on the other hand, may not benefit from such scheme.

To set the array sizes – N_p, N_c, we specify two inputs: desired total representation size (in bytes) and ρ. We also set the length of the sides as the nearest multiple of eight for the primary and a multiple of four for the cascaded. A simple 1-D search suffices to satisfy the constraints whilst also closely matching the desired array sizes. As discussed in section 3, we initialize a d-dim primary array with identity mapping between the input and output.

We implement *Multi Resolution Hash Encoding* in our Pytorch-based framework. When comparing with *MRHE*, we use eight grid levels composed of six levels of multi-resolution *dense-grid* arrays and two levels of *hash-grid* arrays. We found quantizing all levels to 8-bit results in better grid resolutions and improved final quality. Each level stores a 2-channel feature vector, totaling 16 latent input channels to the *MLP*. The *MLP* is four layers deep and 16 wide. For images, we use bi-linearly interpolated 2D grids, and for *NeRF/SDF* we use tri-linearly interpolated 3D grids. At equal parameter count, *MRHE* always requires > 5 additional memory accesses and > 1024 additional FLOPs compared to our method. Figure 15 shows the training characteristics of our technique w.r.t. *MRHE*.

6 RESULTS AND ANALYSIS

6.0.1 Isotropic GGX approximation. We begin with a discussion of *isotropic GGX* approximation as it serves as a simple testbed to compare various neural primitives and analyze their efficiency. Figure 9 compares various neural primitives at equal PSNR (> 40dB) measured across a range of roughness as shown in 6. More importantly,

figure 9 provides an overview of the neural primitives landscape. Notice the differences in resource utilization of the primitives. An *MLP* with 4 hidden layers and 32 units per layer not only requires thousands of FLOPs but also a large memory transfer to fetch the network weights. A single level grid storing the function as texture requires large parameter space but very few FLOPs and memory transfer per pixel. A combination of *MLP* and memory-grid strikes a better balance across the three criterion, but our technique improves further. We only require a modest parameter space and few memory transfers, and FLOPs per pixel. FLOPs required in our technique are for linear interpolation of array cells and often maps directly to hardware texture samplers on GPUs.

6.0.2 Signed Distance Fields. Figure 8 and 14(c) provides a visual and a quantitative analysis of our technique for various parameter sizes. Our technique compares favourably with *MRHE* at equal parameter count but requires fewer lookups and compute per query.

6.0.3 Neural Radiance Fields. Figure 7 shows an increasing overall compression due to the underlying compressed density and RGB grids. The RGB grid is more compressible than the density grid and we refer to plot 14(b) to choose a combination of density and RGB compression that retains maximum reconstruction quality. While the density grid is compressible upto 10×, the RGB grid
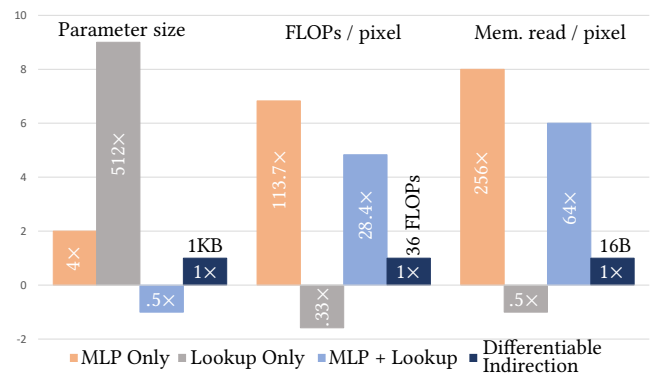
Figure 9: Comparing relative resource utilization of various primitives on a log scale at equal quality (PSNR 40dB) approximation of *isotropic GGX*.
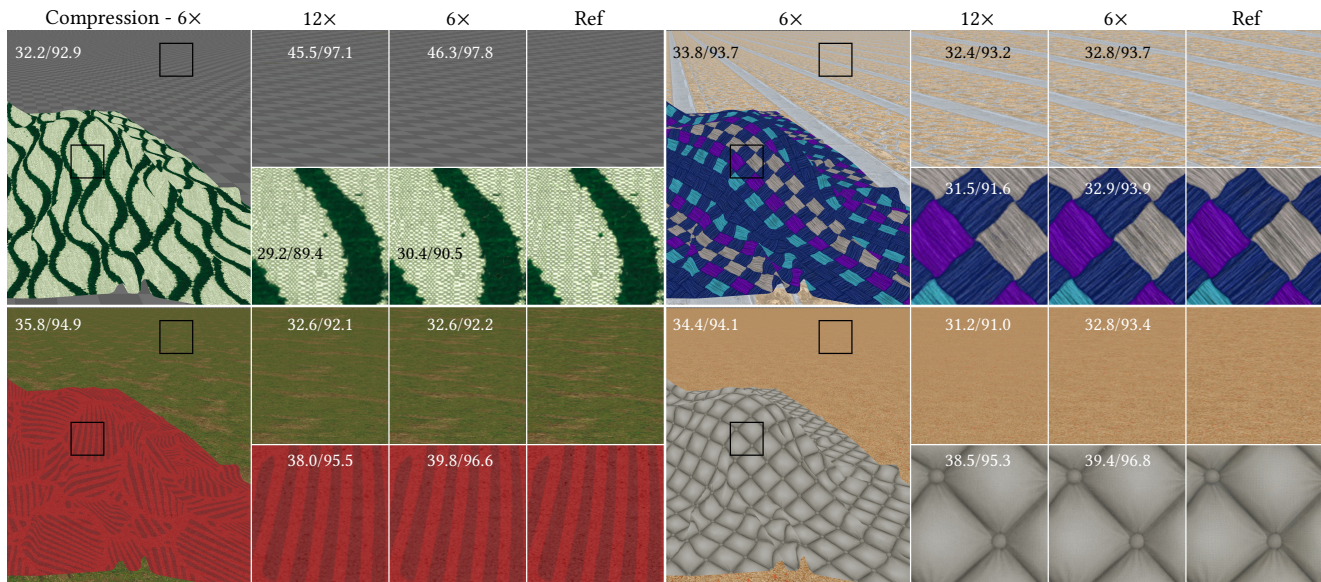
Figure 10: Compressed texture sampling (and filtering) using *DIn*. Top and bottom cutouts show the sampler output for large and small footprints respectively. PSNR(↑)/FLIP(↑) w.r.t. uncompressed 16spp anisotropic filtered 1K base textures.

is compressible upto 100×. Both reference grids have the same resolution but the RGB grid has higher number of latent channels. Thus, there are more redundancies to be exploited in the RGB grid which may explain the differences in their compression.

6.0.4 Compact Image Representation. Our texture compression technique is resource efficient while also being competitive w.r.t. state of the art *block compression* [Delp and Mitchell 1979] techniques such as *ASTC* (figure 14). We require few bytes (≤ 12) of memory read per pixel and a few linear interpolations (≤ 20) to decode a texture. Other neural techniques [Vaidyanathan et al. 2023] may use per material (or a group of materials) *MLP* decoder. This necessitates thousands of additional FLOPs and *kilobytes* of memory transfer to fetch the *MLP* weights per pixel. This is potentially problematic when several materials are present on screen, especially on low end hardware. *DIn* has a constant resource utilization independent of the number of materials present on screen.

We compare our technique with *ASTC* [Nystad et al. 2012], *ETC2* [Ström and Akenine-Möller 2005] and *MRHE* as shown in plot 14(a). *ETC2* has a fixed compression of 6× while *ASTC* has a variable compression up to 24×. Ours and *MRHE*, being learned techniques, achieve unbounded variable compression. In figure 14(a), we downsample the image for compression beyond their respective maximum for *ETC2* and *ASTC*. Our technique is generic, yet comparable in quality with specialized *ASTC*.

6.0.5 Texture sampling. Figure 10 shows the output of our texture sampler at 6×, and 12× compression. Figure 12, compares a single evaluation of our network (1-spp) with nearest neighbor and anisotropic sampled *ASTC* for Albedo, Normal, and AO textures. Note the cost of evaluation of our network is comparable to nearest neighbor *ASTC* while retaining quality much superior to 1-spp *ASTC*. Thus by amortizing texture compression and filtering in one

network, we can extract higher quality per sample than we could with isolated compression and sampling.

6.0.6 Disney BRDF approximation. Here, the primary and cascaded arrays are split into an offline-encoder and a runtime-decoder (figure 4(a)). The encoder transforms the artist control parameters or *PBR* textures into latent encoded textures of the same resolution. Similar to *PBR* textures, these encoded textures are uv-sampled at runtime and used as an input to the decoder along with other geometric dot products. This lends the opportunity to make the encoder arrays much larger - 2k×2k in resolution, as they are completely offline. The cascaded arrays (decoder) are decidedly small - 16×16 in resolution, so they fit in lowest tier caches/*SRAM* and accessed with minimal latency at runtime.

A sweep across all artist parameters is shown in figure 11. The runtime part of the network requires four 2D-bilinear lookups and a single 1D lookup. We require an additional 41 FLOPs to combine the output of the lookups into final result. The reference analytic implementation requires 240+ FLOPs in total. Our runtime arrays are 16x16 resolution and ≤ 4 channels deep, making it easier to test on a commodity GPU. Table 1 shows the runtime performance

Table 1: HLSL runtime performance of approximate and reference Disney BRDF at 4K resolution on Mobile 3070Ti.

Technique	Point emitter count (time in ms)			
	1	2	3	4
Differentiable Indirection	0.654	0.710	0.798	0.923
Reference Disney	0.690	0.838	1.11	1.41
Baseline Diffuse (shader overhead)	0.642	0.648	0.656	0.668

of our technique running at 4K resolution for up to 4 point light sources. A performance advantage is obtained when using the hardware texture sampler for bilinear interpolations.

6.0.7 Shading pipeline optimization. Figure 1 and 13 shows the final results of our end-to-end optimized shading pipeline. We obtain an additional 8%, 5% better PSNR in the first, and second figure respectively using this approach. As shown figure 1, the runtime performance at 4K resolution is < 1.5ms for our learned texture sampling and shading on a Nvidia 3090 GPU. In absence of *ASTC* hardware, we report runtime with *BC7* in figure 1.

7 CONCLUSION AND FUTURE WORK

We show *differentiable indirection* as a powerful primitive that efficiently represents *data* and *compute* across neural graphics pipeline with applications potentially beyond graphics. While our technique is *bandwidth*, *compute*, and *space* efficient, the technique is challenging to apply at higher dimensions. Section 4.3.2 shows a recipe to overcome this using parameter space factorization and exploitation of the problem structure. More generally, we suspect factorization techniques such as spectral or tensor decomposition may prove useful. The effectiveness our technique is also improved by scaling the grid resolution, feature count, or by augmenting the latent representation with fixed function logic such as *SH, PE* or other parameter-free embeddings. While our technique is aimed at improving runtime efficiency, the effectiveness of our technique in the context of direct reconstruction or inverse-rendering tasks is yet to be explored. Finally, we look forward to interesting arrangements of differentiable arrays as regular layers in neural/array networks.

ACKNOWLEDGMENTS

REFERENCES

Adobe. 2023. Substance-3D. https://www.adobe.com/products/substance3d/3d-assets.html

Sai Bi, Zexiang Xu, Pratul Srinivasan, Ben Mildenhall, Kalyan Sunkavalli, Miloš Hašan, Yannick Hold-Geoffroy, David Kriegman, and Ravi Ramamoorthi. 2020. Neural reflectance fields for appearance acquisition. *arXiv preprint arXiv:2008.03824* (2020).

Guillaume Boissé. 2023. GFX: A minimalist and easy to use graphics API built on top of Direct3D12/HLSL for rapid prototyping. https://github.com/gboisse/gfx

Mark Boss, Raphael Braun, Varun Jampani, Jonathan T Barron, Ce Liu, and Hendrik Lensch. 2021a. Nerd: Neural reflectance decomposition from image collections. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 12684–12694.

Mark Boss, Varun Jampani, Raphael Braun, Ce Liu, Jonathan Barron, and Hendrik Lensch. 2021b. Neural-pil: Neural pre-integrated lighting for reflectance decomposition. *Advances in Neural Information Processing Systems* 34 (2021), 10691–10704.

Moshe Caine. 2016. 3D Bronze horse model. https://skfb.ly/Lz7L

Rohan Chabra, Jan Eric Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard A. Newcombe. 2020. Deep Local Shapes: Learning Local SDF Priors for Detailed 3D Reconstruction. *CoRR* abs/2003.10983 (2020). arXiv:2003.10983 https://arxiv.org/abs/2003.10983

Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. 2022. Tensorf: Tensorial radiance fields. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXII*. Springer, 333–350.

Sayantan Datta, Derek Nowrouzezahrai, Christoph Schied, and Zhao Dong. 2022. Neural Shadow Mapping. In *ACM SIGGRAPH 2022 Conference Proceedings* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 9 pages. https://doi.org/10.1145/3528233.3530700

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

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

Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5501–5510.

Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy Mitra. 2022. ReLU Fields: The Little Non-Linearity That Could. In *ACM SIGGRAPH 2022 Conference Proceedings* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 27, 9 pages. https://doi.org/10.1145/3528233.3530707

Alexandr Kuznetsov, Krishna Mullia, Zexiang Xu, Miloš Hašan, and Ravi Ramamoorthi. 2021. NeuMIP: Multi-Resolution Neural Materials. *Transactions on Graphics (Proceedings of SIGGRAPH)* 40, 4, Article 175 (July 2021), 13 pages.

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

Zhiting Lin, Zhongzhen Tong, Jin Zhang, Fangming Wang, Tian Xu, Yue Zhao, Xiulong Wu, Chunyu Peng, Wenjuan Lu, Qiang Zhao, and Junning Chen. 2022. A review on SRAM-based computing in-memory: Circuits, functions, and applications. *Journal of Semiconductors* 43, 3 (mar 2022), 031401. https://doi.org/10.1088/1674-4926/43/3/031401

William E. Lorensen and Harvey E. Cline. 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. Association for Computing Machinery, New York, NY, USA, 163–169. https://doi.org/10.1145/37401.37422

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

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

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. *CoRR* abs/2003.08934 (2020). arXiv:2003.08934

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Trans. Graph.* 41, 4, Article 102 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530127

Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. 2021. Real-Time Neural Radiance Caching for Path Tracing. *ACM Trans. Graph.* 40, 4, Article 36 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459812

Nvidia. 2019. Nvidia cooperative matrix. https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_NV_cooperative_matrix.html

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.

Jeong Joon Park, Peter R. Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. 2019. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation. *CoRR* abs/1901.05103 (2019). arXiv:1901.05103 http://arxiv.org/abs/1901.05103

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance Fields without Neural Networks. In *CVPR*.

Christoph Schied and Anton Kaplanyan. 2022. Systems and methods for graphics rendering based on machine learning. https://patents.google.com/patent/US11436793B1/en US Patent No. 11436793B1, Filed February 12, 2021, Issued September 6th., 2022.

Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (jan 2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

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

Jacob Ström and Tomas Akenine-Möller. 2005. IPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (Los Angeles, California) (HWWS '05). Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/1071866.1071877

Cheng Sun, Min Sun, and Hwann-Tzong Chen. 2022. Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction. In CVPR.

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

Towaki Takikawa, Alex Evans, Jonathan Tremblay, Thomas Müller, Morgan McGuire, Alec Jacobson, and Sanja Fidler. 2022a. Variable Bitrate Neural Fields. In ACM SIGGRAPH 2022 Conference Proceedings (Vancouver, BC, Canada) (SIGGRAPH '22). Association for Computing Machinery, New York, NY, USA, Article 41, 9 pages. https://doi.org/10.1145/3528233.3530727

Towaki Takikawa, Andrew Glassner, and Morgan McGuire. 2022b. A Dataset and Explorer for 3D Signed Distance Functions. Journal of Computer Graphics Techniques (JCGT) 11, 2 (27 April 2022), 1–29. http://jcgt.org/published/0011/02/01/

Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes. (2021).

Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn. 2023. Random-Access Neural Compression of Material Textures. In Proceedings of SIGGRAPH.

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

Joey De Vries. 2023b. PBR Shading. https://learnopengl.com/PBR/Theory

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.

Yin Wang, Hongwei Tang, Yufeng Xie, Xinyu Chen, Shunli Ma, Zhengzong Sun, Qingqing Sun, Lin Chen, Hao Zhu, Jing Wan, Zihan Xu, David Wei Zhang, Peng Zhou, and Wenzhong Bao. 2021. An in-memory computing architecture based on two-dimensional semiconductors for multiply-accumulate operations. Nature Communications 12, 1 (07 Jun 2021), 3347. https://doi.org/10.1038/s41467-021-23719-3

Lance Williams. 1983. Pyramidal Parametrics. In Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques (Detroit, Michigan, USA) (SIGGRAPH '83). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/800059.801126

Tizian Zeltner, Fabrice Rousselle, Andrea Weidlich, Petrik Clarberg, Jan Novák, Benedikt Bitterli, Alex Evans, Tomáš Davidovič, Simon Kallweit, and Aaron Lefohn. 2023. Real-Time Neural Appearance Models. arXiv:2305.02678 [cs.GR]

Kai Zhang, Fujun Luan, Zhengqi Li, and Noah Snavely. 2022. IRON: Inverse Rendering by Optimizing Neural SDFs and Materials from Photometric Images. In IEEE Conf. Comput. Vis. Pattern Recog.

Xiuming Zhang, Pratul P Srinivasan, Boyang Deng, Paul Debevec, William T Freeman, and Jonathan T Barron. 2021. Nerfactor: Neural factorization of shape and reflectance under an unknown illumination. ACM Transactions on Graphics (TOG) 40, 6 (2021), 1–18.

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
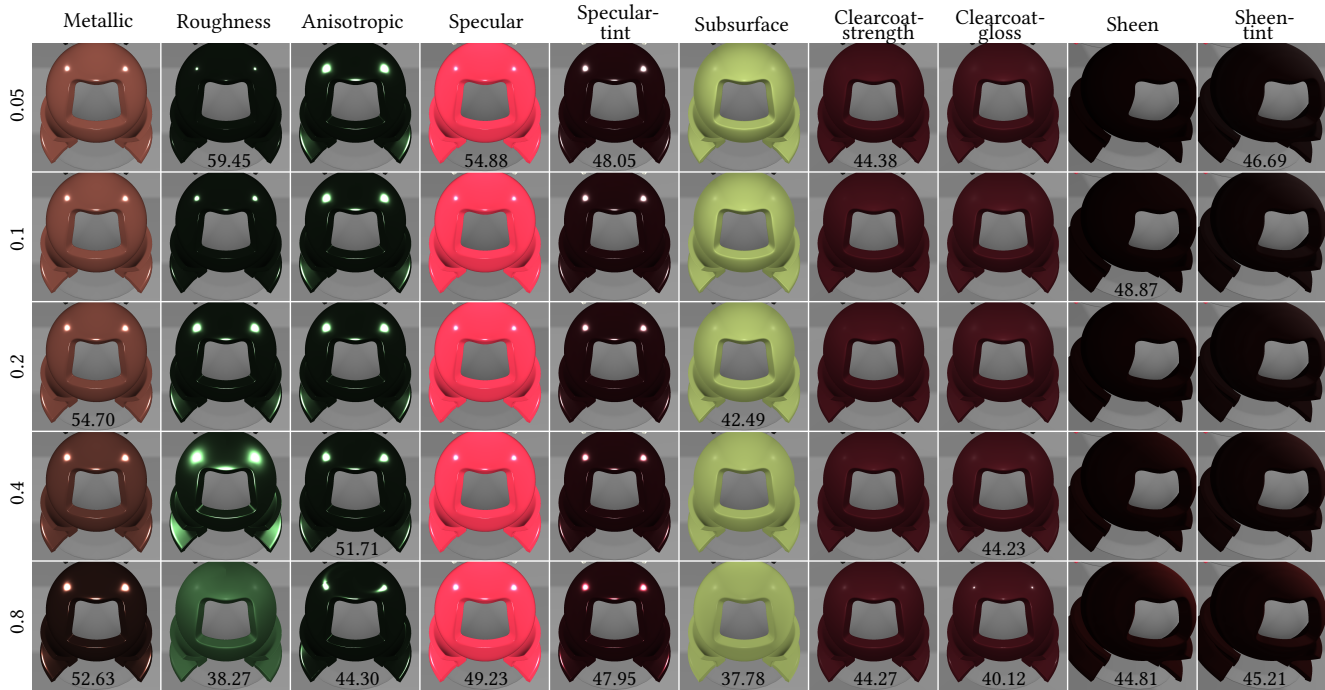
Figure 11: Figure demonstrating the retention of all artist-control parameters using our efficient *Disney BRDF* approximation using *DIn*. The minimum and maximum PSNR when compared with analytic evaluation is provided for each row of varying parameter value along the vertical axis.

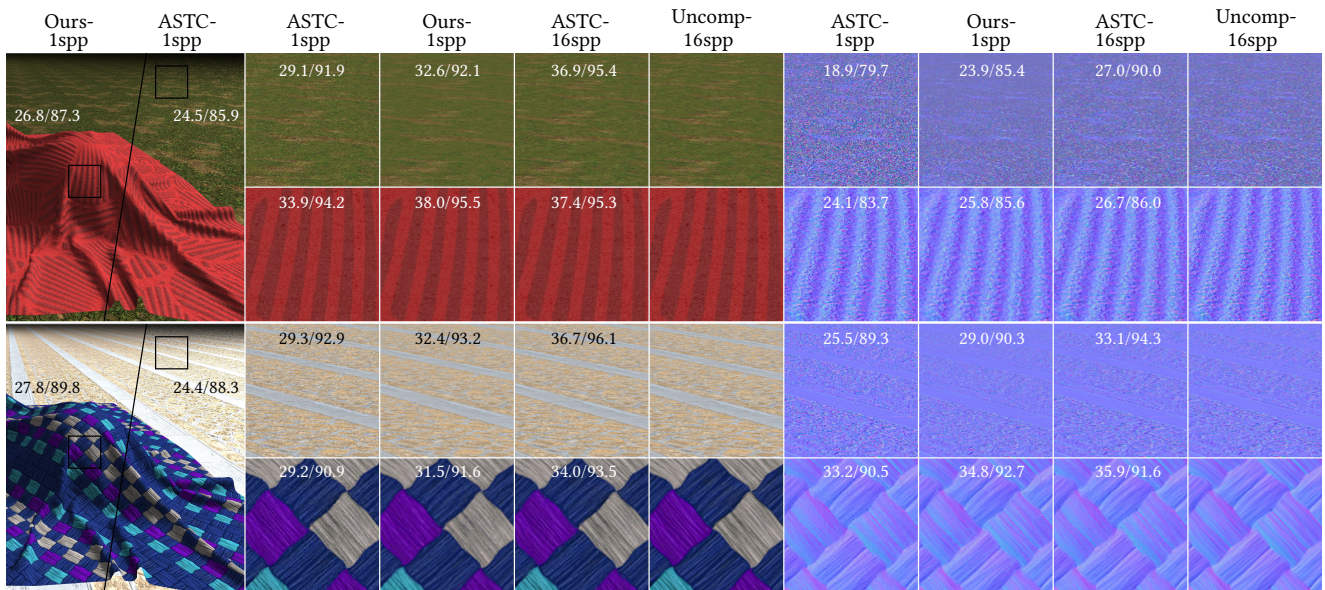**Figure 12: Figure comparing combined effect of texture compression (12x) and filtering using *differentiable indirection* w.r.t. *ASTC* at varying pixel footprints and filtering sample count. Our technique is computationally comparable to *ASTC*-1spp, yet yields better filtered results compared to the same. PSNR(↑)/FLIP(↑) w.r.t. uncompressed 16spp reference is provided. Uncompressed textures are have a base resolution of 1K.**
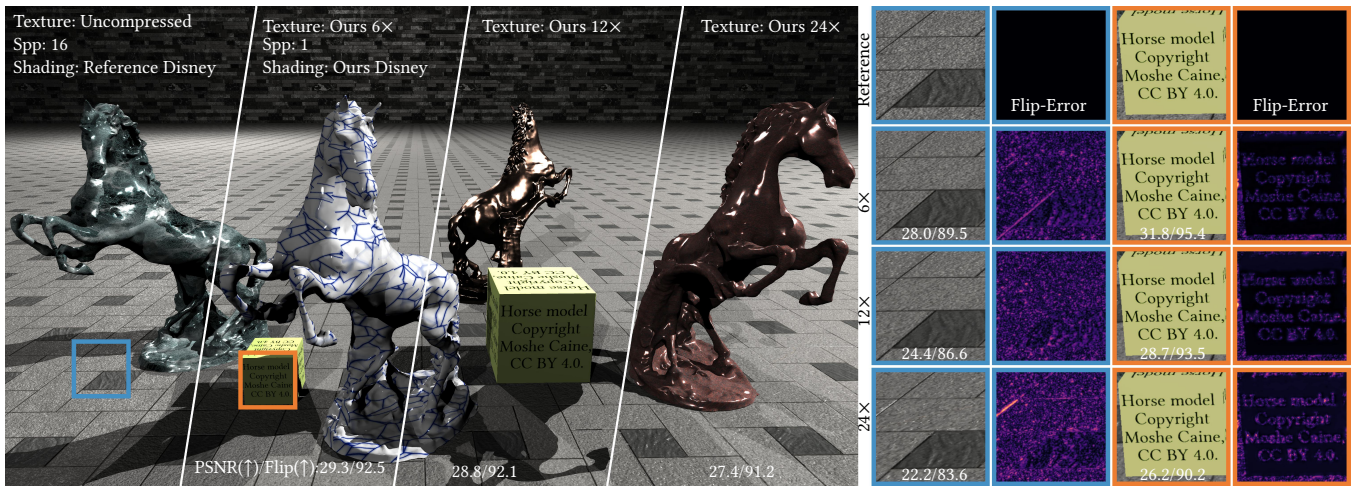
Figure 13: Figure shows the use of our technique on end-to-end *PBR* shading. We jointly optimize our compressed latent texture representation with our differentiable *Disney BRDF* for a 5% improvement in PSNR over independent texuring and shading. PSNR(↑)/FLIP(↑) w.r.t. uncompressed 16spp reference is provided. Uncompressed textures are have a base resolution of 1K. Horse model © Moshe Caine [Caine 2016], CC-BY-4.0.
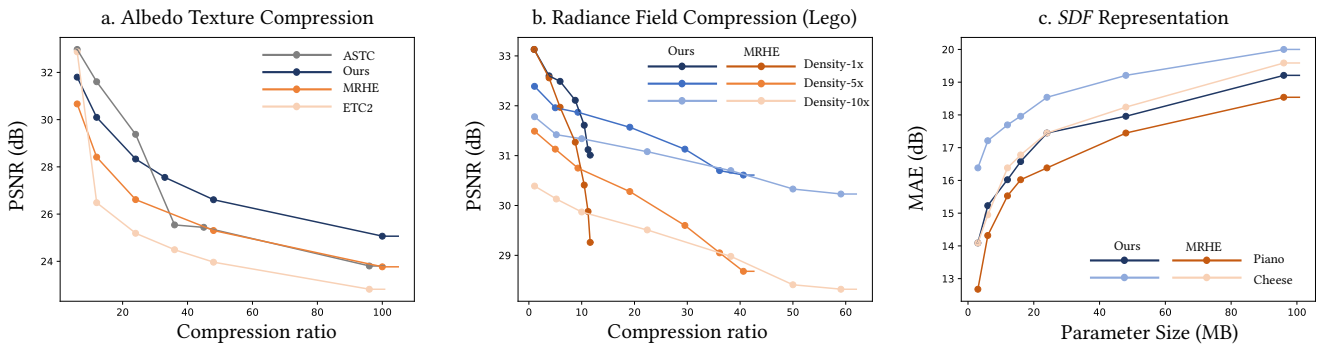
Figure 14: a. Figure compares texture compression behavior for various techniques for 1k textures. *ASTC*, and *ETC2* textures are downsampled beyond their maximum compression ratio of 24×, and 6× respectively. b. Figure shows various iso-lines corresponding to a fixed density-grid compression with varying RGB-grid compression. Total uncompressed grid size is 826MB. c. Figure shows variation in *MAE* (dB) with parameter size for *SDF* representation. *MAE* is computed only on perturbed near-surface samples.

Figure 15: Figure shows the convergence characteristics of *DIn* and *MRHE* for various tasks in the same framework with similar training overheads.

Efficient Graphics Representation with Differentiable Indirection - Supplemental

1 ADDITIONAL DETAILS

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

1.1 Isotropic GGX approximation

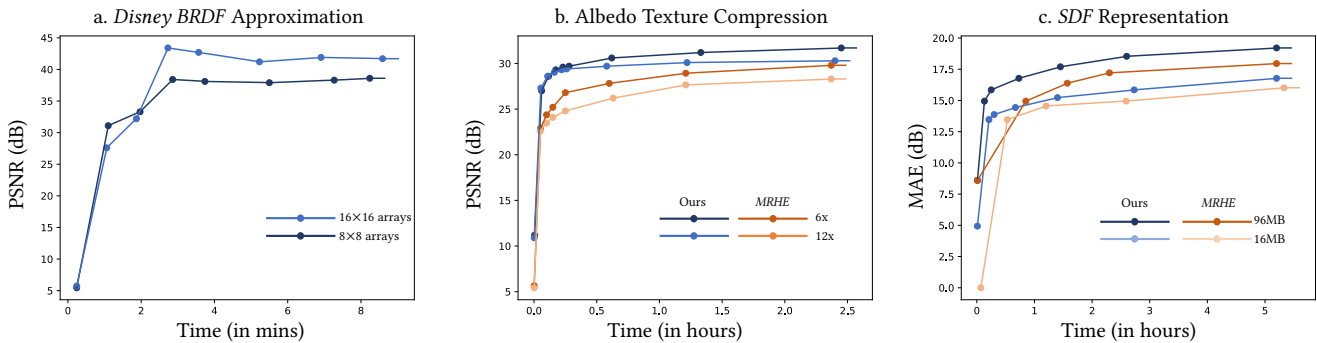*Isotropic GGX* is a popular *BRDF* (*Bidirectional Reflectance Distribution Function*) for modelling glossy reflections in variety of real-time applications. The function itself is very simple, given by:

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

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. An artist decides the glossiness value α_h; the second parameter h_z is obtained from the surface normal, emitter and camera direction at the location of shading, as shown in figure 1. When rendering, equation 1 is evaluated independently for each pixel on screen. We refer the readers to online tutorials [Vries 2023b] for a lightweight introduction to *shading* and other resources [Matt Pharr and Humphreys 2018a; Walter et al. 2007] 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. the shade point. As such, h_z is computed on the fly while α_h 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-map 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 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 is sampled from a cosine-hemisphere [Matt Pharr and Humphreys 2018b] 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 (equation 1) and the loss is back-propagated to train the arrays. The training is performed in a local coordinate frame with normal vector pointing at (0,0,1).

Visualizations and conclusion. Figure 1 visualizes the inference pipeline for *isotropic GGX* case. The pipeline replaces many compute FLOPs required to evaluate equation 1 with two memory lookups. Our approach may produces inherent advantage with increasing resolution as the cost of memory lookups may scale sublinearly with resolution due to caching, while compute FLOPs may scale linearly with resolution. Compared to super-resolution which amortizes cost by exploiting spatial (and/or temporal) locality in screen-space, the case of folding compute FLOPs as lookups may be interpreted as amortization through memory.

1.2 Disney BRDF approximation

Section 1.1 discussed a simplified *BRDF*. Practical *BRDFs*, such as *Disney*, is composed of multiple components which enable artists to manipulate and develop plausible materials by controlling a set of parameters. Our implementation, based on the reference [Li 2022] 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* in section 1.1.

Our goal is to fold as many compute operations (for evaluating the reference *BRDF*) into memory lookups while retaining quality and also improving runtime performance. This is challenging as we must minimize the number of memory lookups. Each lookup is potentially 7-10× more expensive compared to a single *MAC/FMA* operation, even assuming the data resides in closest tier cache or *SRAM*. Here we show one of the many possible instantiations of *differentiable indirection* to approximate *Disney BRDF* while adhering to the aforementioned constraints. While the exact recipe is not crucial, it is useful to show some of the practical constraints that we consider in our implementation.

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

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

where D_d, D_m, and D_c are the *Disney-diffuse*, *Disney-metallic*, and *Disney-clearcoat* distributions (equation 5, 8, and 12 in reference document [Li 2022]). 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 [Li 2022] document. *anisotropic GGX* requires four input – the roughness α_x, α_y along the surface tangents, and the dot-products h_x, h_y of the half-vector (similar to *isotropic GGX* case) with the surface tangents. Hypothetically it is possible to model *anisotropic GGX* using *differentiable indirection* consisting of a 4D primary – queried using the coordinates ($\alpha_x, \alpha_y, h_z, h_y$), and a corresponding 4D cascaded – queried using the pointers stored in the primary. However, 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 our technique is intended to replace. Also, such process would not exploit the fact that some computations can be baked into the textures as an offline
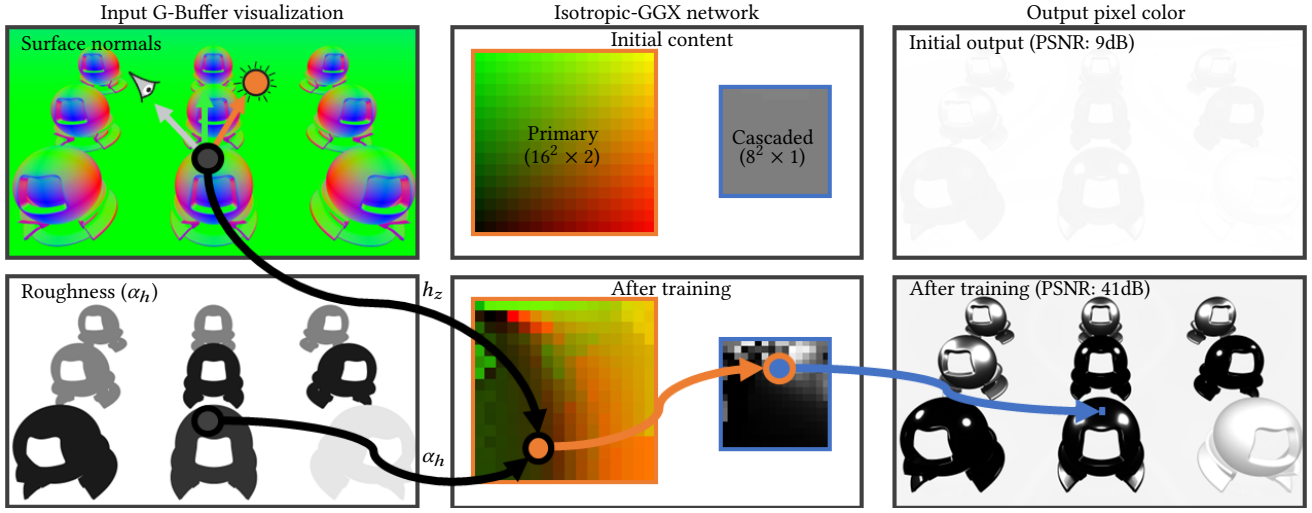
Figure 1: Figure visualizes the *isotropic GGX* inference pipeline. The primary array is queried with the parameters (α_h, h_z) computed from various G-Buffer components as visualized on the left. Content of primary and cascaded array is visualized in the centre while the network output (pixel color) on a variety glossiness (α_h) is shown on the right, before and after training.

process. Instead, we rewrite equation 8 as follows:

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

and learn the coefficients $d_*(\alpha_x, \alpha_y)$ using *differentiable indirection*. Thus the primary is a 2D array queried using the coordinates (α_x, α_y). The cascaded is also a 2D-array (with 3 channels). The channels in the cascaded corresponds to the triplet – d_0, d_1, d_2 in the above equation. Notice we only use *DIn* to process the artist control parameters α_x, α_y. The geometric dot products h_x, h_y are mixed in with the output of *DIn* to produce the final output. Such factorization has the advantage that the primary array can be large (2k×2k) and works completely offline. The primary array acts as an encoder that takes in α_x, α_y and outputs an encoded information or pointer. As discussed earlier, artist control parameter α_x, α_y are stored as uv-mapped textures. In our case, we use the primary array to process each texel in the texture and store the encoded texture instead. The encoded texture is accessed similar to a standard texture fetch and processed using the cascaded array. The cascaded array is kept small (16×16) so that it fits in lower tier cache or *SRAM*. Notice only the cascaded array is required at runtime, minimizing the number of lookups and FLOPs.

Disney-diffuse (D_d). We express the *Disney-diffuse* term as a weighted sum of two functions – f_i and g_i. The first function is parameterized using a subset of the 10 artist control parameters, we call $\alpha_\mathbf{d} \in [0, 1)^2$ and the second function using some subset of geometry dot-product, we call $\mathbf{h_d} \in [0, 1)^3$, as shown below:

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

The accuracy is improved with more terms but empirical observations show $K = 3$ is sufficient to attain accuracy over 35dB. Due to reasons similar to *Disney-metallic*, the artist parameter 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 maybe less accurate but 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 3 visualizes several c_* terms.

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

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

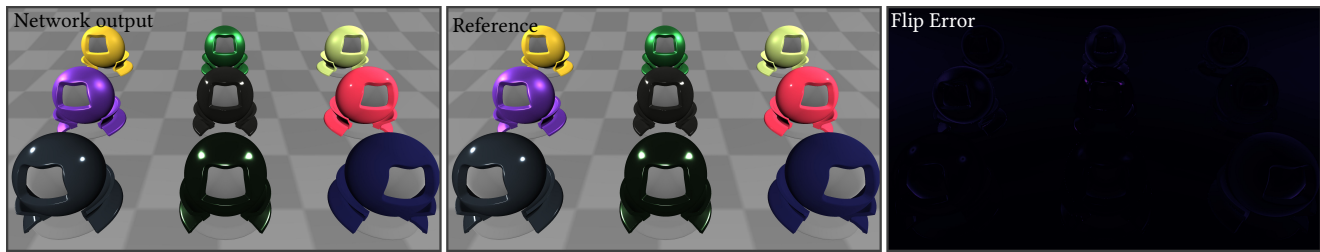| Dimension/ Interpolation	O/P Channels	Associated Output
16x16 / Bilinear	3	Anisotropic GGX
16x16 / Bilinear	1	Smith Masking
16x16 / Bilinear	4	Disney Diffuse, Clear coat masking
16x16 / Bilinear	4	Metallic, Clear coat, Sheen
16 / Linear	2	Clear coat gloss

Figure 2: Figure visualizes the final output of *Disney BRDF* approximation for a variety of material configuration.
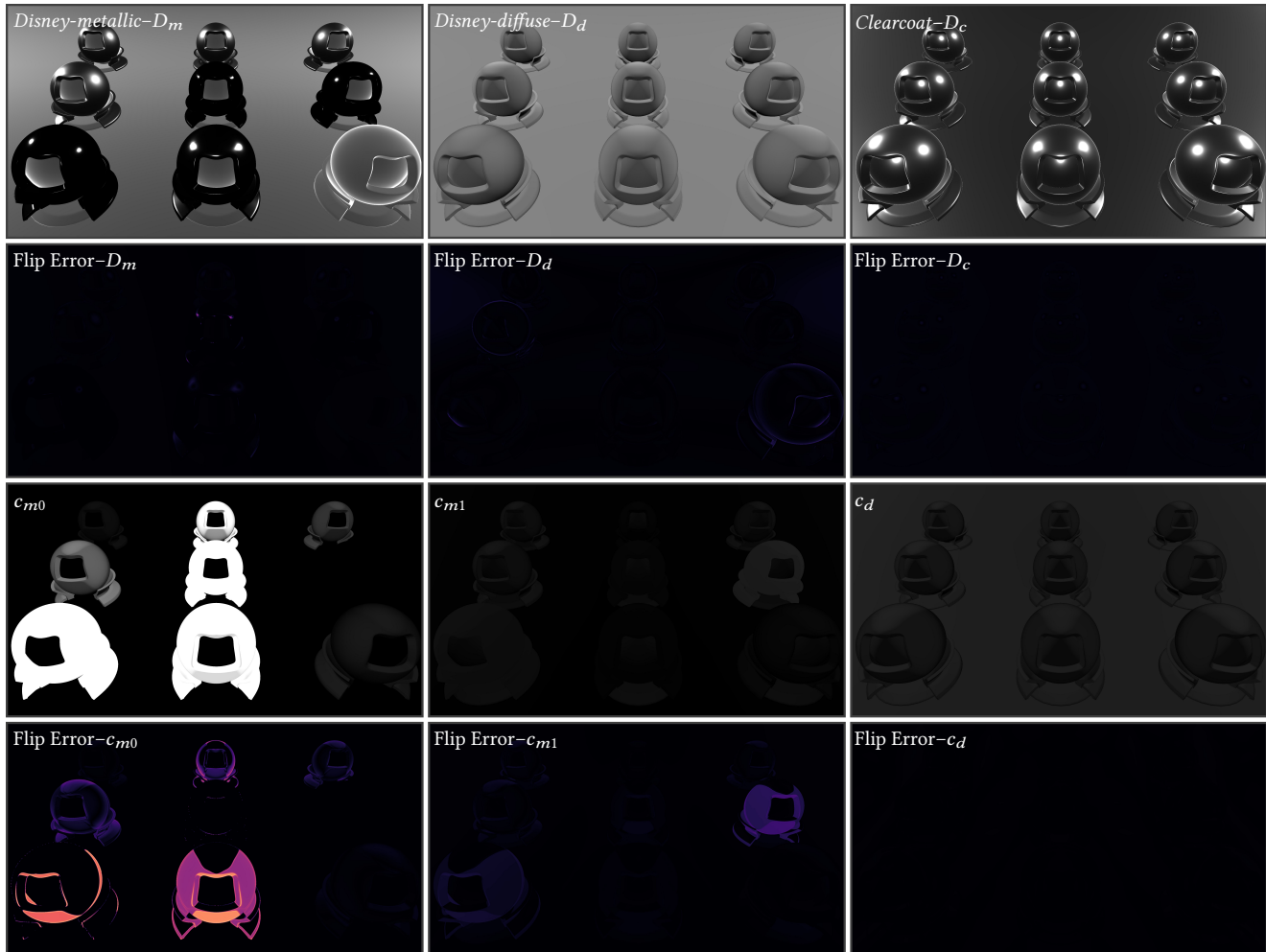

Figure 3: Figure visualizes the intermediate outputs of *Disney BRDF* approximation under a variety of material configuration. Notice errors in c_{m0} is masked by the D_m term as the product $c_{m0}D_m$ is jointly optimized while D_m is also optimized separately.

rest of the artist parameters and albedo from a uniform distribution. We sample the emitter and camera direction from a hemispherical distribution with normal vector 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

outlined in table 1, are required. The output of the primary arrays are encoded as standard 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,
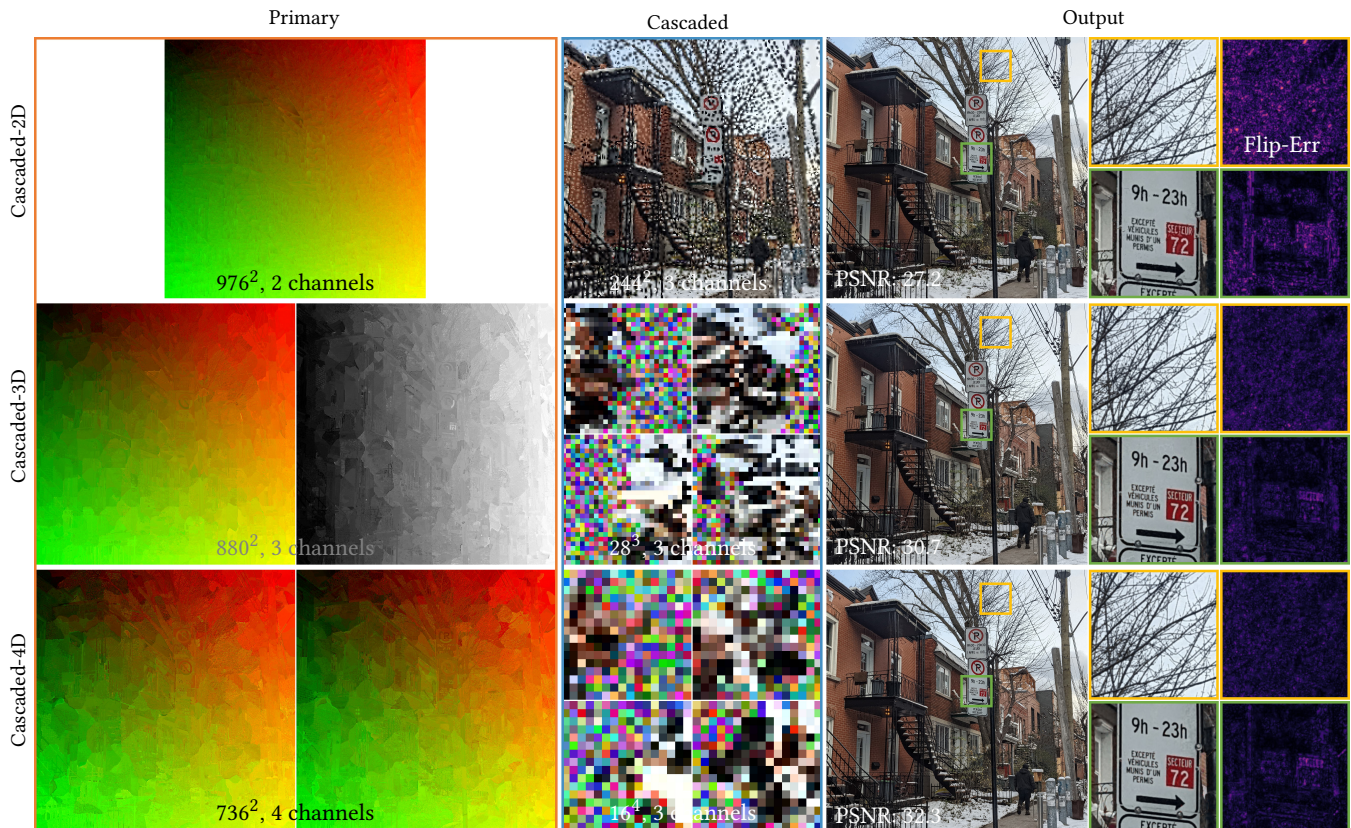
Figure 4: Figure visualizes the contents of the primary and cascaded arrays for varying (vertically) network configurations. All network compresses a 2k image by 6× using *differentiable indirection*. In case of 3D and 4D cascaded arrays, 2D slices of the multi-dimensional volume is visualized in the 2nd and 3rd row. Result improve with the dimension of the cascaded array.

and 2 divisions. The extra arithmetic is required to combine the output of the lookups into the final *BRDF* output. To compare compute instructions as a single number, we assume additions and multiplications require 1 FLOP, divisions 2 FLOPs, square root and logarithms 4 FLOPs each. These are likely conservative estimates on modern hardware and low power hardware may require more FLOPs for complex instructions. Plugging these values, we estimate the required number of FLOPs for reference implementation is about 300 FLOPs. We reduce the number of FLOPs by a factor of 0.8 to account for optimization inefficiencies in our implementation and arrive at a 240 FLOPs estimate for reference implementation. Our approximate implementation requires 5 lookups and 41 FLOPs.

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 cascaded arrays. 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 presumed spatially redundant data. Images are natural – such as photographs taken by a smartphone camera or synthetic – such as video-game textures. Images are also used as containers to store other forms of data such as *normal* [Vries 2023a; Zhu et al. 2022] maps used to alter shading computations or *displacement* [Donnelly 2005; Szirmay-Kalos and Umenhoffer 2008] 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* 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 FLOPs, memory bandwidth required to decode a pixel and the overall representation size is crucial.

Block compression [Delp and Mitchell 1979; Nystad et al. 2012] techniques exploit spatial locality of images by regressing a line
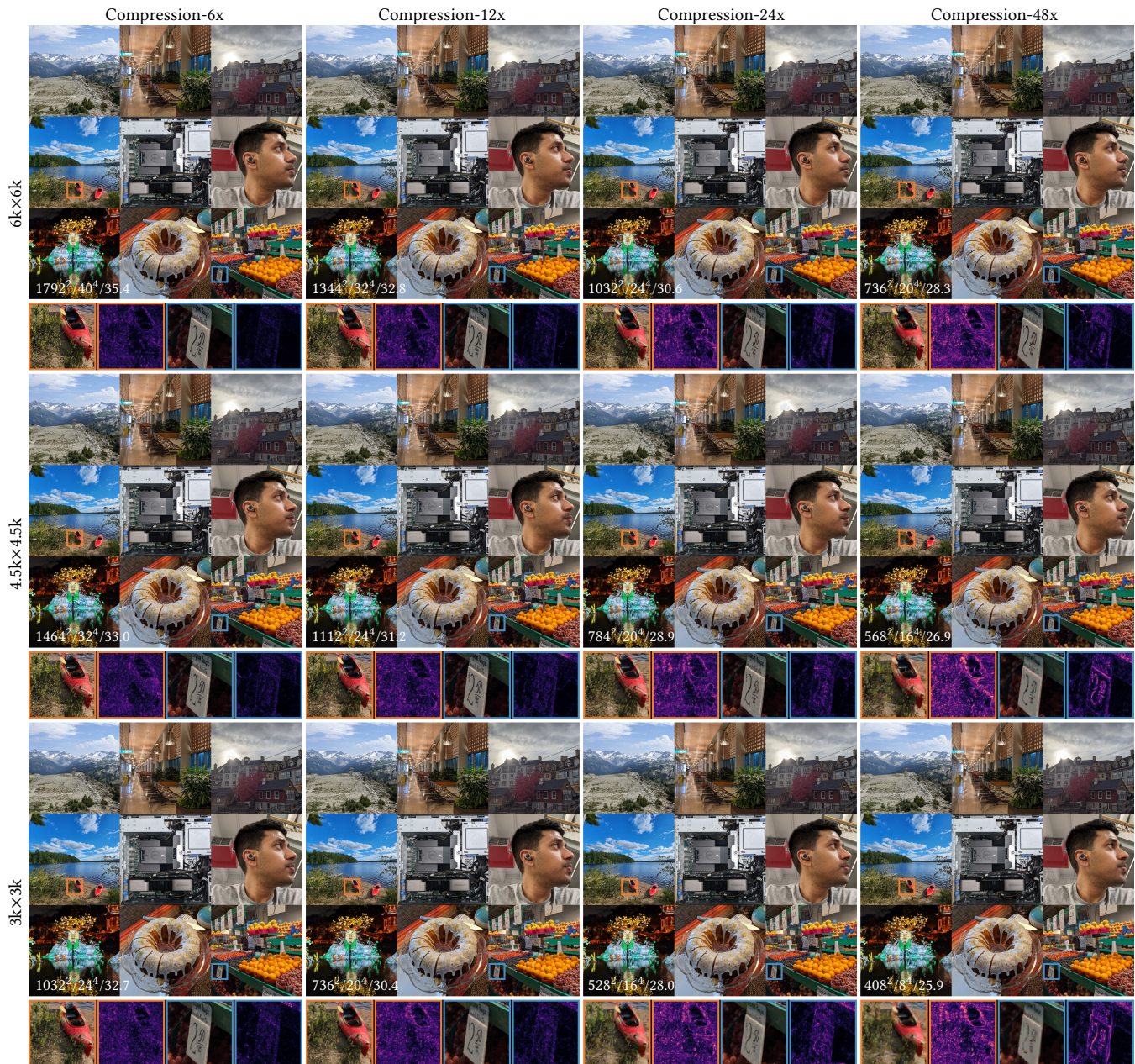
Figure 5: Figure shows the effect of compression (horizontally) on varying image resolution (vertically). While the 1st row compresses a native resolution image, the 2nd and 3rd row compresses a pre-downsampled version of the native image. We use a 2D-primary/4D-cascaded network configuration for all images in this figure. The resolution of the primary, cascaded arrays, and the associated output PSNRs are provided for all images. We note, it is more difficult to compress an already downsampled image, as indicated by a reducing PSNR along each column. Cutouts underneath each image highlight two interesting regions as zoomed-in views along with associated *FLIP* error map.

through a block of pixels and only storing an index identifying a point on the line for each pixel. The technique also stores the end-points of a line per block. Our technique may be considered a generalized extension of block compression where we learn the indices (in the primary array) and the regressed line (in the cascaded array) using gradient decent and *differentiable indirection.* There are several possible arrangements for primary and cascaded array in our case. Figure 4 shows a hyper-parameter sweep across various primary/cascaded arrangements such as 2D/2D, 2D/3D, and 2D/4D.

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

Table 3: Network configuration details for texture 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

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), and 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 is 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:

$$B_{comp} = 4N_p^2 + kN_c^4$$
$$= 4\rho^2 N_c^2 + kN_c^4, \text{ using } \rho = \frac{N_p}{N_c}. \quad (3)$$

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 $e - B/B_{comp}$ 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* [Simonyan and Zisserman 2015] 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. As such, we only use *MAE* as loss. We use *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 $\leq$ 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 primary and varying (2-4D) cascaded

network configurations. Figure 5 shows the effect compression (6-48×) on a range of resolutions.

1.4 Neural texture sampling

Using uv-mapped textures in 3D scenes requires appropriate *texture filtering* to avoid aliasing. Aliasing occurs when 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 size of 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 [Boissé 2023], the latter is computed using shader derivatives. Shader derivatives are *numerical finite difference* derivatives of a quantity w.r.t. x-axis (horizontal, called *ddx*) and y-axis (vertical, called *ddy Direct 3D*) 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 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, identified by its resolution N_{p0}. Second primary corresponding to the pixel footprint input, called N_{p1}. The shapes of the arrays are provided in the table 3.

We use a single cascaded array with 4-dimesnions. The resolution of the cascaded is not uniform along all dimensions. N_c is the resolution of the sides attached to N_{p0} (primary array for uv) and N_{lod} is the resolution of the side attached to N_{p1} (primary array for footprint). For instance, we would compute the volume of the cascaded array as $N_{lod} \times N_c^3$.

We set $N_{lod} \approx log_2(N_{base})$, where N_{base} is the base texture resolution. We further fine tune N_{lod} with 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

$$B_{comp} = 3N_{p0}^2 + N_{p1} + kN_c^3 N_{lod}$$
$$= 3\rho^2 N_c^2 + N_{p1} + kN_c^3 N_{lod}, \text{ using } \rho = \frac{N_{p0}}{N_c}. \quad (4)$$

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.

Training. To generate the target data for training, we emulate 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, only used for target data generation, works by generating a mip-chain of the base texture and *tri-linearly* interpolate between the mip-levels according to uv-coordinates and pixel footprint. Thus the proxy sampler is a soft-emulation of the actual hardware. 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 (LOD) in the mip-chain. We pass the same inputs (uv and pixel footprint) 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 the detailed and less samples from the coarser mip-maps. One way achieve this is to sample the pixel footprint values from an exponential distribution, however, exponential distribution tends to put too few samples for less detailed mip-maps. The samples however, are better distributed by simply raising the samples collected from a uniform distribution to the power of n as described in previous paragraph. This section calculates the optimal value of n. We generate the pixel footprint samples (x_f) as

$$x_f = u^n, \ u \sim U(0, 1), \quad (5)$$

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

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

We calculate the *cumulative distribution function* of X_f as

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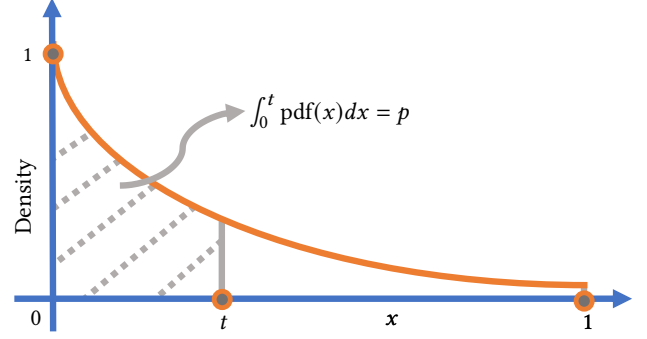
Figure 6: Figure illustrating the fraction of samples p below a threshold t as being equivalent to evaluating the *CDF* at t.

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

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

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

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 $exp(-\lambda_e x)$.

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

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

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

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

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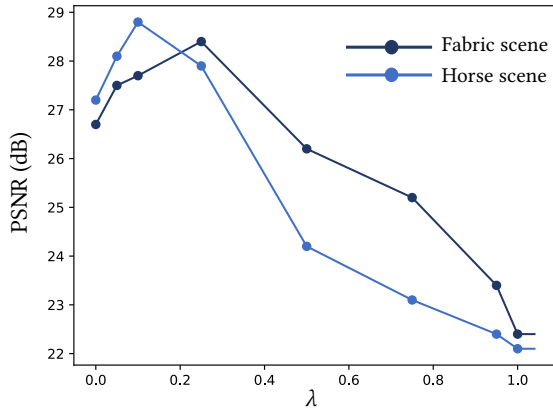
Figure 7: Plot illustrating variation in quality with increasing λ for two scenes. λ indicates the fraction of shading loss used for training as discussed in 1.5

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 learned *Disney BRDF* as a differentiable fixed function layer - i.e. we freeze the contents of the pre-trained decoder arrays in our approximate *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 learned 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*. 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 figure 1, 13 of the main paper shows an improvement in quality with at $\lambda = 0.1, 0.25$ for the two scenes 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 uses the *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 grid-representation, albeit with some minor modification – we remove the *MLP* used at the output of the RGB field and replace it with *positional encoding* (*PE*). The view directions are mixed in with the latent vectors from the grid representation using *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.

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 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

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

SDF representation   Image compression
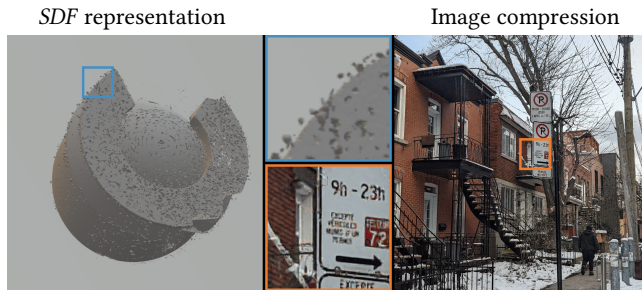
Figure 8: Figure illustrating common artefacts with *SDF* representation and image compression. The *floater* artefacts seen on the surface of geometry is mitigated using truncated *SDF*. For image compression, artefacts like nonlinear-seams may appear at high compression rate – 48× in this case.

2 KNOWN ISSUES AND FIXES

Differentiable indirection generally works out of the box; however, we document some known issues, their fixes, and potential improvements. Figure 8 shows some of the issues we encountered when using *DIn*. When compressing 2D images, we noticed voronoi-seam like artefacts in the compressed representation, especially at higher compression ratios. Similar issue manifest itself in 3D as *floater* artefacts in *SDF* representation as shown in the figure 8.

Analysis. Remember that we initialize the primary array with **monotonically** increasing identity uv-map and let gradient descent locally distort the uv-map. The key to reducing artefact is controlling this distortion. Ideally, for an artefact free output, we should preserve monotonicity in the primary array. However, forcing monotonicity may also reduce the quality of compression, as the primary array cannot adjust freely. These opposing forces must be balanced depending on the circumstances.

Fixes. We start with two simple strategies. These are the first line of defense against artefacts and often extremely effective. First strategy involves controlling the magnitude of the gradient. For a given compression, we aim to limit the amount of unnecessary distortion (non-monotonic) to the identity uv-map stored in the primary. Large gradients may put unnecessary kinks in the fabric of primary which may not iron itself out during training. So we explicitly or implicitly control the gradient going from the cascaded to primary. Some example of implicit gradient control are as follows:

- Pre-initialized cascaded array: The magnitude of the gradient going from cascaded to primary depends on correctness of the values stored in the cascaded. Hypothetically, if the cascaded already contains values that perfectly match the output, there is no gradient distorting the identity uv-map in the primary. As such, initialize the cascaded with good values as often as possible. In case of *SDF* and *NeRF* cascaded is initialized with down-sampled version of the uncompressed representations. For shading and textures, we use a constant DC-value, generally the mean of the output.
- Better loss and bounded output: Avoid losses that produce unbounded output while operating on bounded values. An

example of bad loss function in our context is *MAPE*, generally used in for training *SDFs*. The loss is infinite when the reference input is close to zero. This leads to spurious zero crossing in the compressed representation due to extremely high gradients close to a geometry surface, manifesting as *floater* artifacts. We replace *MAPE* with *MAE* that uniformly penalize all values. However, we also quantize the output of the *SDF* represntation to ±1, a.k.a *truncated sdf* to emphasis the surface boundary without using unbounded loss values. In case of shading, which often produces unbounded output, we learn the reciprocal of the output which has bounded range and avoids bad gradients.

- Smaller cascaded array: We highlight that the backpropagating gradients are amplified proportional to the resolution of the cascaded array. When situation permits, it is better to use smaller cascaded arrays, minimizing the risk of large gradients. We think this information may be vital for developing multi-level or deep indirection where such amplification may play crucial role in training stability.

Next we provide some explicit forms of gradient control:

- Learning rate scheduler: A learning rate scheduler, often built-in the standard libraries such as *PyTorch*, is a very useful form of gradient control. The purpose of a scheduler is to gradually lower the learning rate to reduce overshooting the optima when large (modestly) gradients are presents.
- Gradient clipping: If it is necessary to manually restrict gradients, clipping might be the easiest solution. It is straightforward to access the gradient tensor of the primary array and clip the values before the next gradient-descent step. We determine the clipping bounds in the following paragraph. Usually gradient clipping is **not** required when other gradient control measures are set appropriately.

When set correctly, gradient clipping should enforce monotonicity in the **primary** array, as discussed in the next paragraph. However, strict monotonicity may be too restrictive for most cases, as it imposes harsh constraint on the optimization process and reduces reconstruction PSNR. In this paragraph, we discuss gradient clipping to minimize large kinks (not necessarily enforce monotonicity) in the primary array. An effective clipping bound is thus the average length of the diagonal of a hyper-cube cell. For a 1-dimensional primary array, the bounds are ±(Array resolution×learning rate)$^{-1}$/2. One may also use these bounds for monitoring purposes to detect spurious behavior. We recommend tuning other parameters discussed previously before any gradient clipping.

This paragraph discusses gradient clipping for enforcing strict monotonicity in the primary. Although from our experience, such is neither required nor recommended. To maintain strict monotonicity, one must assign a clipping range for each cell individually; in other words, a gradient descent step should not increase the value of the current cell beyond the value of the next cell and should not reduce its value below the value of the previous cell. We argue that the non-linearity $\mathcal{F}$ is not required when strict monotonicity is maintained as the content of the array is always positive and

always restricted to $[0, 1)$ range. In a 1-dimensional context, we pin the first array cell to 0 and last array cell to 1; gradients do not affect the values in the boundary cells. For the ith cell of a 1-D array, the lower (ϵ_l) and upper (ϵ_u) clipping bounds are:

$$\epsilon_u[i] < \frac{c[i+1] - c[i]}{2\alpha},$$

$$\epsilon_l[i] > \frac{c[i-1] - c[i]}{2\alpha},$$

where $c[i]$ is the content of the array and α is the learning rate. We clip the gradients in the aforementioned range for the ith cell in the array to enforce strict monotonicity.

We have now covered our first strategy involving controlling the gradient, either implicitly or explicitly; we are ready to discuss our second strategy. Our second strategy involves increasing the number of features or channels in the primary array and accordingly adjusting the dimensionality of the cascaded array. The additional dimensions allows for robustness against sharp kinks/distortions in the primary array. The idea is simple; at any given input location in the primary array, the probability that a large fraction of channels have sharp kinks/distortions at the same location is reduced. For example, it would be bad if all 4 channels in the primary array in figure 4 (last row) had a kink at the exact same location. However, the probability of such event is lowered with increasing channels. Generally, having one or more extra channels/dimensions than required is great for improving robustness against artefacts.

Advanced fixes. We have now discussed the main mechanisms to build robustness against artefacts. We next discuss some more advanced techniques, however, we found they are rarely required in practice. We introduce the concept of soft monotonicity. Unlike strict monotonicity, our goal is to encourage monotonicity in the **primary** array without forcing it. Assuming 1-D array, we define monotonicity as:

$$\mathcal{F}(c[i+1]) - \mathcal{F}(c[i]) > \epsilon,$$

where $c[i]$ indicates the content of the array at i^{th} location, $\mathcal{F}$ is the output non-linearity and $\epsilon \leq 0$ is a small negative value. Note that the left side of the equation is computing a numerical finite difference derivative using *forward difference* method. One may use more advanced numerical derivative such as *central difference* to compute a more accurate finite difference derivative on the left side. On the right side, setting $\epsilon = 0$, would indicate strict monotonicity, while a small negative value indicates some tolerance for non-monotonic behavior. We set $\epsilon \propto -N^{-1}$, where N is array resolution. We put the above equation as regularization as follows:

$$\frac{\kappa}{N} \sum_{i=0}^{N-1} ReLU(\epsilon + \mathcal{F}(c[i]) - \mathcal{F}(c[i+1])),$$

where κ is a positive regularization constant. The regularization term is added with other losses and optimized using gradient descent. One small implementation details is that we first compute $\mathcal{F}(c[i])$, copy the result to another array without gradient tensor, pad the copied array with border values, and offset the indices to compute appropriate finite difference.

Future work. We have outlined all important techniques that we have tested for improving robustness and reducing artefacts. We discuss another potential idea that may help reduce artifacts. Recall our arrays are linearly interpolated. A higher order interpolant such as cubic may improve smoothness and reduce artefacts but at the cost of increased computation and reduced compression capacity due cubic constraints.