Real-Time Neural Materials using Block-Compressed Features

Clément Weinreich, Louis de Oliveira, Antoine Houdard, Georges Nader

Volume xx (20yy), Number z, pp. 1–13

Real-Time Neural Materials using Block-Compressed Features

C. Weinreich†, L. De Oliveira†, A. Houdard† and G. Nader†

Ubisoft La Forge

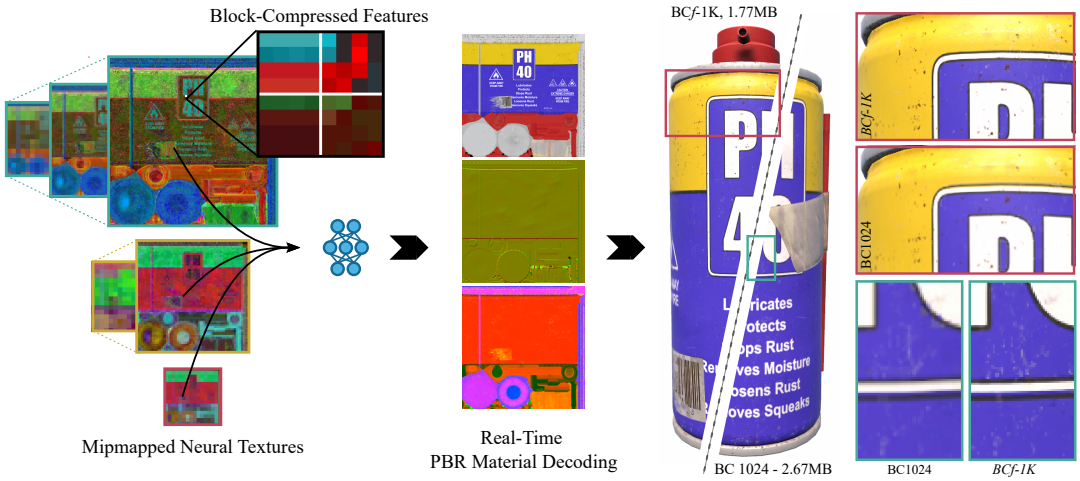


Figure 1: *PBR material texture set is decoded in real-time from our Block Compressed neural features (BCf) resulting in an image that is visually sharper than standard BC textures of similar resolution.*

Abstract

Neural materials typically consist of a collection of neural features along with a decoder network. The main challenge in integrating such models in real-time rendering pipelines lies in the large size required to store their features in GPU memory and the complexity of evaluating the network efficiently. We present a neural material model whose features and decoder are specifically designed to be used in real-time rendering pipelines. Our framework leverages hardware-based block compression (BC) texture formats to store the learned features and trains the model to output the material information continuously in space and scale. To achieve this, we organize the features in a block-based manner and emulate BC6 decompression during training, making it possible to export them as regular BC6 textures. This structure allows us to use high resolution features while maintaining a low memory footprint. Consequently, this enhances our model's overall capability, enabling the use of a lightweight and simple decoder architecture that can be evaluated directly in a shader. Furthermore, since the learned features can be decoded continuously, it allows for random uv sampling and smooth transition between scales without needing any subsequent filtering. As a result, our neural material has a small memory footprint, can be decoded extremely fast adding a minimal computational overhead to the rendering pipeline.

1. Introduction

The continuous challenge of real-time rendering systems is to improve the graphics quality while reducing both the memory cost and the evaluation time. Recent progress in graphics hardware and rendering algorithms has gotten us closer to this goal. For instance, hardware accelerated ray-tracing [Fen22] facilitates real-time dynamic global illumination and drastically improves the quality of shadows, reflections and refractions [WNK22, OLK*21, DNSD22]. Besides that, micropolygon-based data struc-

† Equal Contribution, order determined by coin toss

tures [MMT23, KSW21] allows the rendering of extremely detailed and dense scenes. In terms of visual aesthetics, the Physically Based Rendering (PBR) framework has been the standard in real-time applications for well over a decade [MHH*12, MHM*13, HMB*20]. In this framework, a material is composed of multiple texture layers (such as albedo, normals, metalness, etc.) with each layer serving a distinct role in accurately representing various aspects of the material's visual characteristics. Enhancing the visual quality of materials typically involves the process of either layering multiple elements [HdR23] or increasing the texture resolution, which affects their computational cost and memory footprint, respectively. Consequently, rendering hyperrealistic materials in real-time applications, such as video games, remains somewhat restricted.

Recent work has shown the potential of neural networks to model and represent material properties [SRRW21, ZZW*21, FWH*22, XWH*23]. This neural approach aims at replacing traditional PBR textures with a collection of learned latent features, also known as neural textures [TZN19], alongside a neural network, usually a Multi-Layer Perceptron (MLP). In this context, the network plays a crucial role in decoding the learned information and reconstructing the original material. While neural approaches have proven successful in accelerating the rendering of complex appearances [ZRW*23] and compressing high-resolution PBR materials [VSW*23], its integration in consumer oriented real-time applications such as video games is not straightforward. The trained latent features have a relatively big memory footprint as they are often stored in GPU memory using an uncompressed format. Recent work by Vaidyanathan *et al.* [VSW*23] proposed to overcome this issue by reducing the resolution and heavily quantize the values of these features. However, to compensate for the loss in resolution and precision, the associated decoder network is rather large and thus computationally intensive which leads slow evaluation time. As a result, achieving real-time performance remains challenging, and depends on specific hardware extensions that are supported on only the most recent high-end hardware. The method we present addresses these challenges. Our goal is to design a neural material model that can be integrated into a rendering pipeline and achieve real-time performance without having to rely on any specific hardware acceleration.

To do so, we present a framework capable of (1) learning the material information at any point and scale and (2) leveraging hardware-based Block Compression (BC) format to store the learned neural features. Our features can thus be decoded continuously at any scale in uv space, making it possible to reconstruct the material information with one sample per pixel and achieve smooth transition between various scales. This removes the need for any subsequent filtering step. Furthermore, encoding our neural features as regular block compressed textures reduces their memory footprint enabling us to increase their resolution. This, in turn, directly influences the complexity of the decoder network, making it considerably simpler and significantly reduces computational time.

The paper is organized as follows. Section 2 quickly details existing material representation and compression methods. We then present the technical aspects of our neural material learning framework and block based features in sections 3 and 4 respectively. In section 5 we describe how to practically use our framework to learn a standard PBR material and integrate it in a real-time rendering pipeline. Finally, we present our results in section 6 and conclude by discussing some limitations and future work in section 7.

2. Related Work

This section provides an overview of the various methods to represent and store materials that are relevant to our work. We first focus on the traditional texture based approaches (sec. 2.1) then highlight the more recent field of neural representation methods (sec. 2.2).

2.1. Texture Materials

The material properties of a three-dimensional object can be thought of as a multi-dimensional signal, which associates every point (u, v) on its two-dimensional surface with the parameter space of its appearance model. In a real-time rendering context, this is primarily done via texture mapping [AMHH18] where the material properties are discretely stored in a collection of textures with dimensions $h \times w \times c$. Here, h and w denote the spatial resolution of the textures, and c represents the total number of channels across all texture layers in the set. Increasing the visual fidelity of materials primarily relies on either adding more layers [HMB*20] or increasing their resolution which has led to a significant rise in memory requirements. To address this issue, textures are stored in GPU memory in a compressed format. Due to the object's arbitrary position and orientation, material information are sampled from the textures at random locations. It is therefore necessary for the texture's compression format to allow for random-access and filtering so that the information can be decompressed at any point in uv space when needed in real-time. This makes standard image compression techniques, such as JPEG [Wal92, AVAB*19], as well as newer neural based ones [BMS*18, CSTK20] not suitable for this application as they require unpacking the entire image before being able to access pixel information.

For real-time rendering, block compression methods [CDF*86, INH99] have long been the standard for compressing material textures. There are seven variants of the BC format (BC1-BC7) supported in DirectX [D3D], each designed for specific types of image data. All BC formats divide the image into block of 4x4 pixels and operate under the assumption that the colors in each block exhibits minimal variation and are evenly distributed along one or more line segments within the RGB color space. This means that each block can be represented by a very small color palette. In this context, the information in each block is encoded by storing the two endpoints of each segment and indexing each pixel according to its position in the RGB space (fig. 2). Reconstructing the pixel value is simply done by blending the two endpoints proportionally to the index value. The more recent BC6 and BC7 formats are the ones using more than one segment per block. These formats, designed for floating points and RGBA data respectively, improve the compression quality by separating the pixels inside a block into several groups, each having its own set of endpoints. The groups are chosen from a set of predefined partitions. In this case, each block stores more than one set of endpoints as well as the partition number. However, BC formats can only compress textures

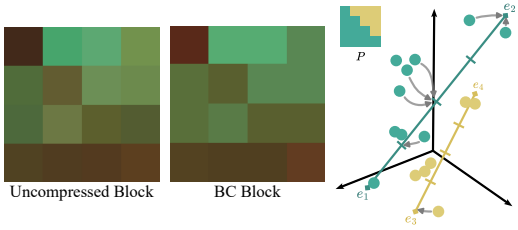


Figure 2: *Block Compression algorithms encode a block of* 4×4 *pixels with a set of endpoints forming one or multiple line segments and index each pixel based on it's projected position in the RGB space. In the case where two or more line segments are stored, the pixels are separated into groups according to a pre-defined partition P.*

with up to four channels which is not suited for high dimensional materials. To overcome this, the material data is separated into several textures that are compressed independently. ASTC [NLP*12] is another popular block-based texture compression technique. It is more flexible than BC as it supports various block dimensions including non-square blocks and can even handle 3D textures.

2.2. Neural Materials

Over the last few years, advancement in neural rendering [TTM*22] have shown that it is possible to represent a digital signal, such as a material M, with a neural network f_η by minimizing over its weights η the following quantity:

$$\sum_{i,j} \|f_\eta(u_i, v_j) - M_{i,j}\|^2, \tag{1}$$

where $M_{i,j}$ is the pixel value of the target material and (u_i, v_j) are their corresponding local coordinates. This is an overfitting problem where the weights of the network are optimized such that the network's output matches the target image at a given pixel. Thus, the capability of the network is key to perfectly recover the original image. For instance, simple coordinate based networks [MST*20] are not capable of dealing with high frequency details. To improve the network's reconstruction capacity, positional encoding [TSM*20, CRSL22] where the input (u, v) coordinates is encoded as a vector generated from a periodic function, is usually employed. However, this requires a large network to accurately reconstruct the original image which makes it not suitable for real-time evaluation. Using trainable spatial features [MESK22, CXW*23, CLS*23, SP23], *i.e.*, neural textures, drastically reduces the size of the network and improves the reconstruction quality. In this setting, the input coordinates are used to sample in the neural textures using bilinear interpolation and the resulting feature vector is given as input to the neural network.

The idea of pairing a set of discrete spatial features and a neural network have proven to be popular for material representation. For instance, Rainer et al. [RJGW19] uses an encoder-decoder architecture to compress a large set of Bidirectional Texture Function layers. The encoder is trained to generate a latent code for each texel. This code is then used by the decoder, in conjunction with a light and view direction to output a single RGB value. Kuznetsov et al. [KMX*21] combined a pyramid of neural textures with a fully connected network to learn the material properties at different scales. Their model is capable of rendering materials with intricate parallax effects on an infinite plane but fails to generalize to curved surfaces. This was later done in [KWM*22]. By adding curvature and transparency as part of the network's input and output respectively. More recently, Zeltner et al. [ZRW*23] have used a set of hierarchical textures and two MLPs to bake complex film-quality appearance. Here, the first network learns the material's reflectance and the seconds produces importance-sampled directions. This model is about three times faster than standard node-based multi-layers materials when integrated into a path-tracing pipeline.

Despite this progress, these methods often overlook the issue of storage size. In practice, learned neural features are stored in an uncompressed format which can be quite large to practically use in a real-time environment, especially when considering the memory capacity of mainstream hardware. Vaidyanathan et al [VSW*23] addressed this issue and demonstrated how neural material representations can be more efficient than standard texture compression techniques at storing PBR material information. To do so, they reduce the resolution of the neural features as much as possible and aggressively quantize their values. This made it possible to compress PBR textures at very low bit-rates, up to 0.2 bits per-pixel per-channel (bppc). However, integrating this neural material decompression in current rendering pipeline and achieving real-time performance requires access to the latest high-end hardware. Moreover, since the model from [VSW*23] only learns the material information at fixed coordinates, it is essential to couple the neural decompression with a filtering operation [FWSP23] in order to minimize flickering and aliasing artefacts. This introduces additional computational overhead as it entails decompressing multiple samples per pixel.

In the following, we introduce a novel neural material representation using Block-Compressed features (BCf) specifically designed to be integrated into a traditional real-time rendering pipeline at minimal computational overhead.

3. Neural Material Framework

A PBR material M is constituted of a set of properties. These properties are represented at a coordinate (u, v) by a vector of dimension c. In a standard 3D application, this data is stored in a discrete fashion through a texture of size $w \times h \times c$ and mapped onto a 3D object. At render time, this information is sampled from the textures as follows:

$$\mathcal{F}(M)(u, v, s) \in \mathbf{R}^c, \tag{2}$$

where $\mathcal{F}$ is a filtering operation and s is a scale value. The filtering is essential here. It accounts for the misalignment and the difference in area between the screen pixel and its corresponding texels, caused by the 3D object's arbitrary distance and orientation. Our neural material model aims at replacing the material textures with a set of neural features containing abstract information and a decoder network. The role of the decoder here is to reconstruct the material data at a given point and scale using the learned features. Figure 3 gives an overview of our neural material framework.

Let $\{T_{\theta_0}, \ldots, T_{\theta_n}\}$ be a set of neural features of size

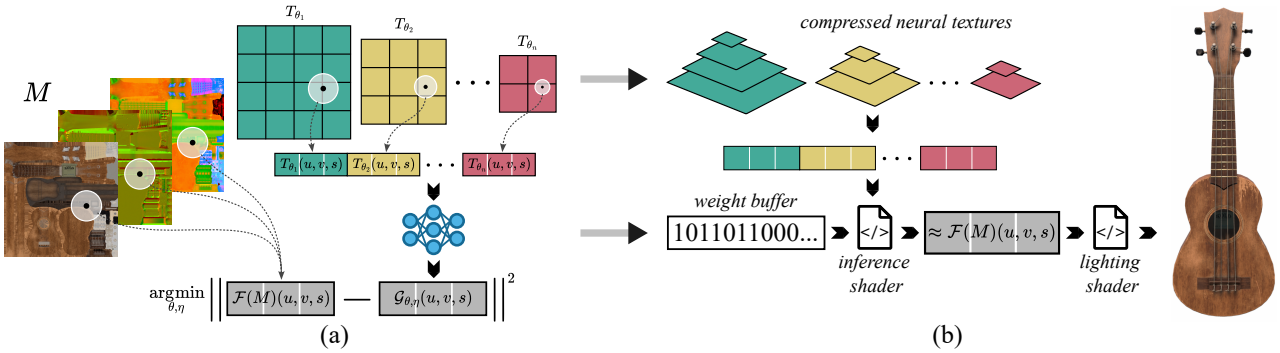


Figure 3: *Overview of our neural material framework. (a) The neural features T_{θ_i} and the MLP f_η are fitted through backpropagation to match the filtered material $\mathcal{F}(M)$. (b) After training, the neural features T_{θ_i} are exported as mipmapped texture sets that can be sampled by the engine and the weights η of the MLP are exported as a binary buffer. A shader is used to perform the MLP inference after trilinearly sampling the neural texture, outputting the filtered material $\mathcal{G}_{\theta,\eta}(u,v,s) \approx \mathcal{F}(M)(u,v,s)$. Finally, The renderer can perform the shading step as usual.*

$(w_0,h_0,d_0),\ldots,(w_n,h_n,d_n)$ with trainable parameters $\{\theta_0,\ldots,\theta_n\}$. And let f_η be a fully connected neural network of input size $\sum_{i=1}^{n} d_i$ and output size c with trainable parameters η. To reconstruct the material information at a point (u,v) with respect to a scale s, we sample each of the features, concatenate the resulting values and pass it to the neural network as follow:

$$\mathcal{G}_{\theta,\eta}(u,v,s) = f_\eta\left(T_{\theta_0}(u,v,s),\ldots,T_{\theta_n}(u,v,s)\right). \tag{3}$$

In this framework, learning the material M boils down to optimizing θ and η such that:

$$\hat{\theta},\hat{\eta} = \arg\min_{\theta,\eta} \left\|\mathcal{G}_{\theta,\eta} - \mathcal{F}(M)\right\|^2. \tag{4}$$

This allows us to train a model such that it simulates a given filtering operation. In practice, we use a batched stochastic gradient descent to perform this optimization where the gradient is computed on a batch of random values of (u,v,s). This leads us to minimize the following loss:

$$\sum_{(u,v,s)\in B} \left\|\mathcal{G}_{\theta,\eta}(u,v,s) - \mathcal{F}(M)(u,v,s)\right\|^2. \tag{5}$$

Once trained, we export the learned features as textures and store the model's weights in a binary buffer. At render time, the neural textures and the weights are used to reconstruct the material information. To make neural materials more appealing, it is important to reduce their memory footprint and to make their memory size match the traditional material texture. There are two possible strategies here. The first one consists in using low bitrate and low resolution neural features [VSW*23]. However, this approach leads to a more complex decoder architecture, making the material reconstruction more computationally expensive. The second aims at storing the neural features more efficiently by using a compression algorithm. We chose the latter and propose to store them in the BC6 format as it is designed to handle floating point data and allows for random-access. Additionally, these compressed features can be handled like any traditional texture, which considerably simplifies the neural material's decoding at rendertime. However, naively compressing the learned features at the end of the training

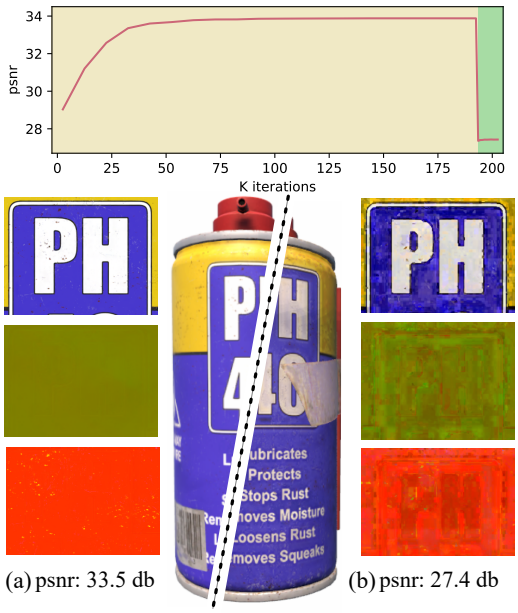


Figure 4: *Neural material reconstructed from (a) raw unconstrained neural features and (b) compressed ones. The naive compression of the neural features will lead to artifacts severely affecting the visual quality.*

will lead to artifacts that severely impact the visual quality of the reconstructed material (fig. 4). In order to tackle this issue, we propose a specific block compressed neural feature parameterization that is compatible with the BC6 format.

4. Block Based Neural Features

In this section, we detail our block-based neural features. The goal here is to be able to store the trained neural features as BC6 textures without affecting the visual quality and use them in a real-time en-

vironment. To do so, we structure the features in blocks of 4×4 and define the parameters on a per-block basis (sec. 4.1). Then, we design a forward pass that emulate the BC6 decompression (sec. 4.2) and takes advantages of hardware texture samplers (sec. 4.3). This makes it possible to directly export the neural features as BC6 textures without the need for a subsequent compression step and use them into real-time environment with a minimal computational overhead.

4.1. Block parameterization

In the BC6 setting, an image is stored by encoding the endpoints value for each 4×4 block and indexing each pixel according to its distance to the corresponding line segment (fig. 2). The dual partition mode of BC6 divides the pixels inside each block into two regions allowing for two lines segments per block.

We design the neural features to mimic this behaviour. Given a feature layer T_{θ_i} of size $w \times h \times 3$, we force its values within a block of size 4×4 to lie on two lines segments. This is done by design, we structure our block-based neural features by modeling the parameter set θ_i such that each block of size 4×4 is modeled with

$$\begin{cases} l_1 = \{e_1, e_2\} \subset \mathbf{R}^3 \\ l_2 = \{e_3, e_4\} \subset \mathbf{R}^3 \\ x = \{x_1, \dots, x_{16}\} \subset [0,1] \end{cases}, \tag{6}$$

where l_1 and l_2 are two sets of endpoints for the first and second line segments respectively and x pixel index represented as the relative position of each one of the sixteen pixels on the corresponding line segment. To determine which pixel belongs to which segment, we attach to each block an integer, $k \in \{0, \dots, 31\}$, linking to a binary mask of the corresponding partition. We refer the reader to the DirectX BC documentation [D3D] for all the technical details regarding the pre-defined partitions.

Reconstructing the material information (eq. 3) requires sampling each neural feature T_{θ_i} at position (u,v) and scale s. This consists of performing the BC6 decompression and then filtering the corresponding value according to a certain strategy (fig. 5). In order to train our block-based model, it is necessary to backpropagate through this decompression. Additionally, it is important to choose a filtering strategy that does not incur additional computation overhead. The latter is crucial to be able to integrate our neural material model in rendering pipelines.

4.2. Trainable BC6 decompression

In the two partition mode, a BC6 block stores two sets of quantized endpoints, the pixel indices and a partition ID. The decompression operation uses this data to recover back the original information by mixing the endpoints proportionally with the index values. According to the BC6 standard specification [D3D], this operation is non-linear and is done in three steps. First, the endpoints of each block are unquantized as follow :

$$\bar{e}_i = \frac{a 2^{16} e_i + 2^{15}}{2^b}, \quad i = 1, \dots, 4. \tag{7}$$

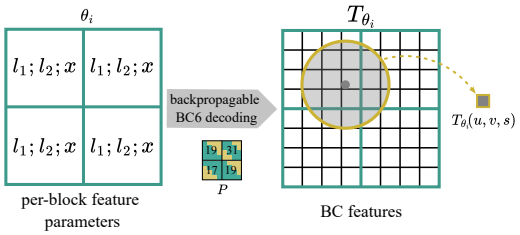


Figure 5: *Evaluating T_{θ_i} at position (u,v) and scale s is done by performing a BC6 decompression then filtering the result with respect to the given scale. Our simulated BC6 decoding makes it possible to backpropagate through this operation and train the per-block neural features.*

Where $a = {}^{31}/_{64}$ for the unsigned BC6 mode, $a = {}^{31}/_{32}$ for the signed BC6 mode and b is the number of bits used to quantize the endpoints. This essentially maps the endpoint values into a valid range. The second step consists in interpolating the endpoints linearly:

$$y = P_k \odot (e_1 + 2^q x(\bar{e}_2 - \bar{e}_1)) + \neg P_k \odot (\bar{e}_3 + 2^q x(\bar{e}_4 - \bar{e}_3)), \tag{8}$$

where P_k is the binary mask associated with partition k and q is the number of bits used to quantize the pixel indices. This will give a value $y \in [-31743, 31743]$ and whose bits are finally re-intrepreted as a half precision number [iee19]. This cast is a non-linear transformation that can be simulated with the following operation

$$w = 2^{h(y)-14} \left(\frac{y}{1024} - h(y) \right), \tag{9}$$

where $h(y) = \max\left(\lfloor {}^{(y-1)}/_{1024} \rfloor - 1, 0 \right)$.

Therefore, to simulate the hardware BC6 decompression for each pixel within a block, we unquantize and mix the endpoints using eq. (7) and eq. (8), and finally simulate the bit re-interpretation operation with eq. (9) resulting in the final value w. All these operations being almost-everywhere differentiable, it allow us to backpropagate through the BC6 decompression, when the partitions are fixed.

4.3. Hardware compliant filtering

In a real-time rendering context, sampling a texture at a particular scale is often done via mipmapping. This technique consists in explicitly storing in memory a version of the texture at different scales, also known as mip, and performing trilinear filtering. This involves doing a texture lookup and bilinear filtering on the two closest mipmap levels (one higher and one lower), and then linearly interpolating the results. In general, this approach stabilizes sampling performance as it fixes the number of processed pixels for each query independently of the scale values.

Since our goal is to integrate the neural material model in a real-time environment, we propose to rely on trilinear interpolation to filter the neural features T_{θ_i}. This makes it possible to exploit hardware accelerated texture filtering to sample the corresponding neural textures during rendering. To do so, we consider that each feature layer T_{θ_i} is composed of a pyramid with S_i block-based mips,

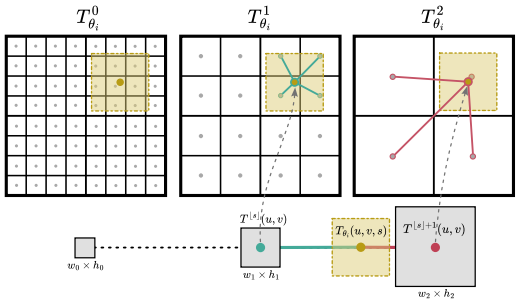


Figure 6: *When sampling a feature at a scale level s, the block compressed features are filtered using a trilinear interpolation. This requires sampling the two closest mip levels $\lfloor s \rfloor$ and $\lfloor s \rfloor + 1$ using a bilinear interpolation than linearly mixing the results.*

$T_{\theta_i} = \{T^0_{\theta_i}, \dots, T^{S_i}_{\theta_i}\}$, of decreasing sizes. Each block-based mip has independent parameters that will be adjusted during the training process. In this setting, the scale parameter $s \in [0, S_i]$ refers to the level at which the features T_{θ_i} are sampled. This is done as follows:

$$T_{\theta_i}(u,v,s) = \lambda T^{\lfloor s \rfloor+1}(u,v) + (1-\lambda) T^{\lfloor s \rfloor}(u,v), \qquad (10)$$

where, $\lambda = s - \lfloor s \rfloor$ and $T^k_{\theta_i}(u,v)$ is the bilinear interpolation from the k^{th} block based mip at coordinates (u,v) (fig. 6).

5. Implementation details

In this section, we present all the practical details related to the encoding and decoding of PBR material information with our block based neural features.

5.1. Material encoding

Model structure In our implementation, we use 4 sets of neural block-based features $\{T_{\theta_0}, T_{\theta_1}, T_{\theta_2}, T_{\theta_3}\}$ with size $w_i = h_i = 2^{n_i}$. As described in section 4.3, each T_{θ_0} consists of a pyramid with several block-based mips of decreasing power of two sizes. We use a simple MLP with RELU activations as our decoder network and adjust the size of the output according to the endcoded material.

Partition selection strategy As mentioned in section 4.2, we can backpropagate through the simulated BC6 decompression only if the partition IDs for each block are fixed. In order to maximise the reconstruction quality, it is necessary to select the most optimal partition for each block. However, choosing the one with the lowest error during training is not straightforward. Indeed, the optimal values of l_1, l_2 and x_i might vary significantly depending on which partition k is considered. This means every-time the partition changes, these parameters need to re-adapt to the new set of pixels resulting in training instability. To overcome this issue, one approach would be to learn in parallel all the parameters for each possible partition for each block. This is unpractical and would increase both the memory and time needed to learn the material model. Instead of that, we propose to consider the partitions as hyper-parameters, and thus fix them before the training. Since randomly setting their values would possibly lead to non-optimal partition attribution, we therefore propose to learn the material model in two steps. First, we perform a quick training using a set of features with unconstrained parameters, i.e., move freely in 3D space and not constrained on a line segment. Then, we initialize the block-based features by compressing the unconstrained features with the BC6 algorithm. This provides us both a partition selection strategy and a relevant initialization of the parameters within each block. In our experiments, we train the model for 5k iterations with unconstrained parameters, then use the result to initialise the block based features. While this does not guarantee the selection of optimal partitions, we found that it improves the reconstruction quality and consistency of the results.

Reference material sampling In our experiments, we consider reference materials with S mipmaps and sample them at random during training by doing a bicubic filtering on the closest two mips then linearly interpolating the results. More precisely, we process batches of 512x512 uniformly sampled uv-grids, for each batch we also sample a continuous scale parameter s uniformly in $[0, S]$ where S is the number of mipmap. Both the reference material and the neural material (sec. 4.3) are then sampled at (u,v,s) and the loss is computed as the mean squared error between them. For more details on the training parameters, we refer to the results in section 6.

5.2. Real-time Decoding

We export each trained feature layer as a mipmapped BC6 texture. The block-based structure of these features simplifies this operation as it only involves the quantization and encoding of the endpoints and pixel indices with 6 bits and 3 bits respectively. The network's weights are stored directly in a binary buffer as fp16 values since their size is negligible.

Multiple methods can be employed to integrate our model into a real-time renderer. Our implementation is rather straightforward and simply consists in loading the neural textures and weights in GPU memory and proceed as follows (fig 3). First, we use the partial derivatives of the input pixel's uv coordinates to compute the sampling scale s_i for each neural texture T_{θ_i}

$$s_i = \log_2 \left(\max \left(\left| \partial uv / \partial u \right|, \left| \partial uv / \partial v \right| \right) \right) + b_i, \qquad (11)$$

where $b_i = \log_2 \left(\max(h_i/h, w_i/w) \right)$ is a bias value that depends on the neural texture's and render resolutions, respectively $w_i \times h_i$ and $w \times h$. Then, we sample each of the neural textures using the GPU's hardware filtering capabilities and pass the resulting values to the decoding shader. Within this shader, the sampled values processed through a sequence of functions that perform vector-matrix multiplications with the model's weights and the return the material information. When rendering a scene with multiple neural materials, we store all the different networks weights in a single buffer and use a material id value as an offset to access a specific model's weights. Finally, the pixel is shaded.

6. Results

6.1. Experimental setup

Evaluation dataset We gather a dataset of 19 materials from polyhaven.com, 12 of which are accompanied with a 3D model. All of

Table 1: *our Neural block-compressed feature configuration.*

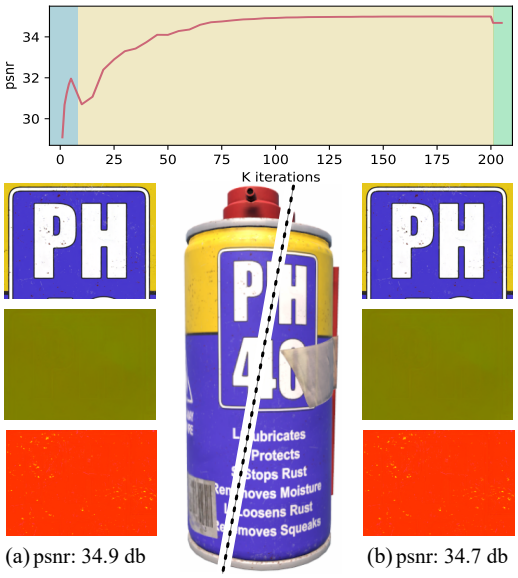
	T_{θ_0}		T_{θ_1}		T_{θ_2}		T_{θ_3}	
	res	*mips*	*res*	*mips*	*res*	*mips*	*res*	mips
BCf-0.5k	512	8	256	7	128	6	64	5
BCf-1k	1024	9	512	8	256	7	128	6
BCf-2k	2048	10	1024	9	512	8	256	7
BCf-2k++	2048	10	2048	10	512	8	256	7

the material information consist of at least 9 channels that are stored in a set of three textures : albedo, normals, and arm. The albedo layer has 3 channels and contains the diffuse color information. The normals layer has 3 channels and contains surface normals with respect to a local tangent frame. In practice, our model learns the x and y components since the z value can be reconstructed. The arm layer has 3 channels and contains the ambient occlusion, surface roughness and metalness parameters. 8 of the materials in the dataset contains an additional displacement channel that is stored in its own texture. Note that our model is not restricted to learn only this specific material representation but can handle materials with arbitrary number of layers.

Model configuration. In our experiments, our models consisted of four mipmapped square shaped block compressed features. Table 1 details the resolution of each of the feature layers along with the corresponding number of mips. Our decoder network consist of a small MLP with 12 inputs and one hidden layer of dimension 16. The output of the network depends on the number of channels in the material texture set. For the rest of this paper, we will refer to our model by the name of its block compressed features configuration and the size of its network. For instance BCf-0.5K refers to the model with a resolution of T_{θ_0} equal to 512.

Training parameters. We use the Adam stochastic gradient descent algorithm [KB14] with an exponential decay learning rate scheduler. We set a different learning rate for both the block compressed features and the MLP in order to balance the gradient backpropagation. We train the models for a total of 205k iterations in two phases. In the first phase we train our model with unconstrained features for 5k iterations using learning rates of 5×10^{-2} and 10^{-3} for the features and the MLP respectively, and a decay parameter $\gamma = 0.9995$. The second phase starts by initializing the block compressed features from unconstrained ones as described in section 5.1. Then we train the model for 200k iterations using a learning rate of 10^{-2} for the features and 10^{-3} for the MLP with decay parameter $\gamma = 0.9999$. For reference, our BCf-1K requires about 140 minutes to be trained for the 205k iteration on an NVIDIA RTX2070 with our implementation using the PyTorch library [PGM*19]. Note that we observed that the model already outputs decent results after 10k steps, in approximately 15 minutes.

Compared methods. We compare our method with standard BC compression, ASTC [NLP*12] and our implementation of Vaidyanathan et al.'s NTC [VSW*23]. Both BC and ASTC compressed textures were generated using NVIDIA's texture tools exporter [NVI] with a compression quality set to the highest possible setting. For BC compression, we used the BC5 format to store the normal layer and the BC1 for the rest for rest of the texture set (the albedo, armand displacement textures when available). For ASTC, we used a 12×12 blocks as it matches the size of our BCf-1K configuration. In the context of our experiments, where the reference material resolution is 2048×2048, the highest resolution of NTC's features grid of set to 512 for NTC0.2 and NTC0.5, and 1024 for NTC1.0.



(a) psnr: 34.9 db (b) psnr: 34.7 db

Figure 7: *Our training process comprises three stages: A warmup stage (blue) with unconstrained neural features. The main training stage (yellow) with Block-based features initialised from the result of the previous stage. Finally, a finetune stage (green) the BC features are quantized. The results at the end of the second stage (a) and third stage (b) are visually identical.*

Considered metrics. Quantifying the visual quality of an image is still an open problem as no metric can effectively align with human perception. This is especially the case when the types of distortions introduced by the compression methods are different. For instance, traditional BC and ASTC methods tends to have more blocky artefacts while neural methods such as ours and NTC tend to exhibit color shift and feature bleeding. For this reason we mainly rely on the PSNR value as it is directly linked to the loss we are optimizing. In this sense, the PSNR can be seen more as a proxy for the method's capacity to approximate the reference data and not as a measure of visual quality. We also include SSIM [WBSS04] and FLIP [ANA*20] values as they are supposed to be more in line with human perception. We compute our metrics for each mip level and aggregate the values as described by Vaidyanathan et al. in [VSW*23].

6.2. Reconstruction quality

Block-compressed features. Figure 7 shows the evolution of the PSNR value throughout the entire training. We start by training a set of unconstrained embedding and use them to initialise the BC

Table 2: *Evaluation of the reconstruction metrics (PSNR, SSIM and FLIP) on the whole dataset. For PSNR, we averaged the mean-squared error over all the mips (as proposed in [VSW*23]) and all the models then computed the PSNR.*

	BCf–0.5K	BCf–1K	BCf–2K	BCf–2K++	NTC0.2	NTC0.5	NTC1.0	BC		ASTC12	
								1024	*2048*	*1024*	*2048*
PSNR (↑)	28.70	31.83	36.02	37.86	31.90	33.92	37.13	28.49	36.16	26.82	30.48
SSIM (↑)	0.85	0.90	0.95	0.96	0.89	0.92	0.93	0.84	0.96	0.78	0.89
FLIP (↓)	0.142	0.099	0.069	0.059	0.089	0.069	0.057	0.099	0.036	0.143	0.082
size (MB)	0.44	1.77	7.08	11.08	0.93	2.26	4.53	2.91	11.65	0.53	2.11

features at the 5000th iteration as stated in section 5.1. It is expected that the PSNR value drops at this point since the network's input undergoes a change due to the alteration of the embeddings. The BC constrained model is trained for 200K iterations, after which we quantize the trained end-points and indices. Following this, we freeze the values of the BC features and proceed with training for 1000 more iterations to fine-tune the network. Although quantizing the trained features leads to a slight reduction in PSNR value, its visual impact is negligible. The material information reconstruction from the raw unquantized fp16 block compressed features (fig. 7 (a)) and from the exported BC6 block features (fig. 7 (b)) are visually indistinguishable.

Visual Performance. We trained all our model configurations against a reference 2K material and compared them to the methods mentioned in section 6.1. The results are gathered in table 2 and illustrated in figure 8. In these experiments, the texture set is reconstructed using a regular *uv* grid aligning with the center of pixels and the metrics are evaluated by comparing the result with the reference 2K material. Our model outperforms standard BC and ASTC textures as it does not exhibit blocky artifacts and is capable of reconstructing sharper visuals with less memory (see fig. 9). It is not surprising to see that the resolution of the neural features has a direct impact on the capacity of the model to reconstruct the material. The higher the resolution, the better the reconstruction. However, our model's performance does not scale equally with the increase in resolution and can output a slightly smoother result when compared with NTC [VSW*23] (see fig. 9). This is due to our minimalist neural architecture that relies on a very small MLP with one hidden layer of size 16 and an input of size 12 to reconstruct a very complex signal which is particularly well suited for real-time performance. While naively increasing the size of the hidden layers does improve the quality, it causes the inference time to increase in a quadratic manner. We refer to the supplementary material for additional experiments with larger networks and other configurations.

The main advantage of our approach with respect to NTC, is its capacity to output filtered material information. This is particularly important since, in a 3D environment, material texture sets are almost never perfectly aligned with the screen's pixel. The misalignment between the 3D object and the viewport, along with changes in distance, results in the material being sampled at random points within the *uv* domain and across continuous scales. NTC cannot handle this in a straightforward manner since it is trained by considering a fixed uv grid and lod values. For instance, in fig. 10, we

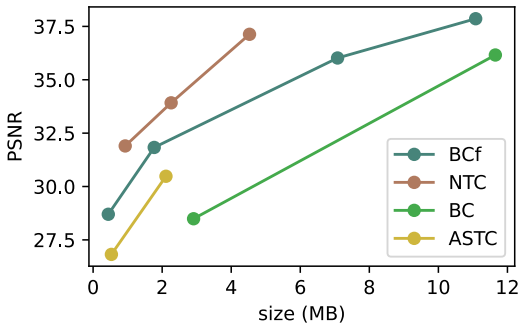


Figure 8: *Reconstruction PSNR versus size in megabytes (MB) for all the methods presented in table 2.*

compare our method to NTC by evaluating the result while constantly changing the lod value. It shows that NTC's performance is not stable when opting to reconstruct the material with 1 sample per pixel (spp) and requires to be paired with a filtering operator. In this case, a bilinear or trilinear filtering is not suited for real-time rendering as it requires decoding multiple samples per pixel. While temporal stochastic filtering can mitigate this issue, its reconstruction quality is dependant on several factors (jittering pattern, noise distribution, accumulation policy, etc) and does result in a smoother reconstruction in motion (fig. 11). Our approach, on the other hand does not require any filtering and is capable to reconstruct the material texture with 1spp. Moreover, the values outputted by our decoder remains stable across time (fig. 11) and mip levels (fig. 10) and can even handle extreme magnifications (fig. 12) . This is not surprising since our model is trained to emulate a continuous filtering operation. In this case our network plays the role of a decoder and a filter at the same time.

6.3. Decoding performance

In order to better illustrate the computational overhead of our neural material model, we rendered a textured full-screen quad at 1080p, 1440p and 2160p (4K). We then compared the total rendering time of our BCf–1K with rendering times obtained using the same setup but with standard BC textures. The results are presented in table 3. They clearly show that our method is well suited for real-time environments as the computational overhead on a 4K resolution is only 0.6ms for a full 4K screen. The small computational overhead is a direct consequence of our block compressed neural features. Ex-

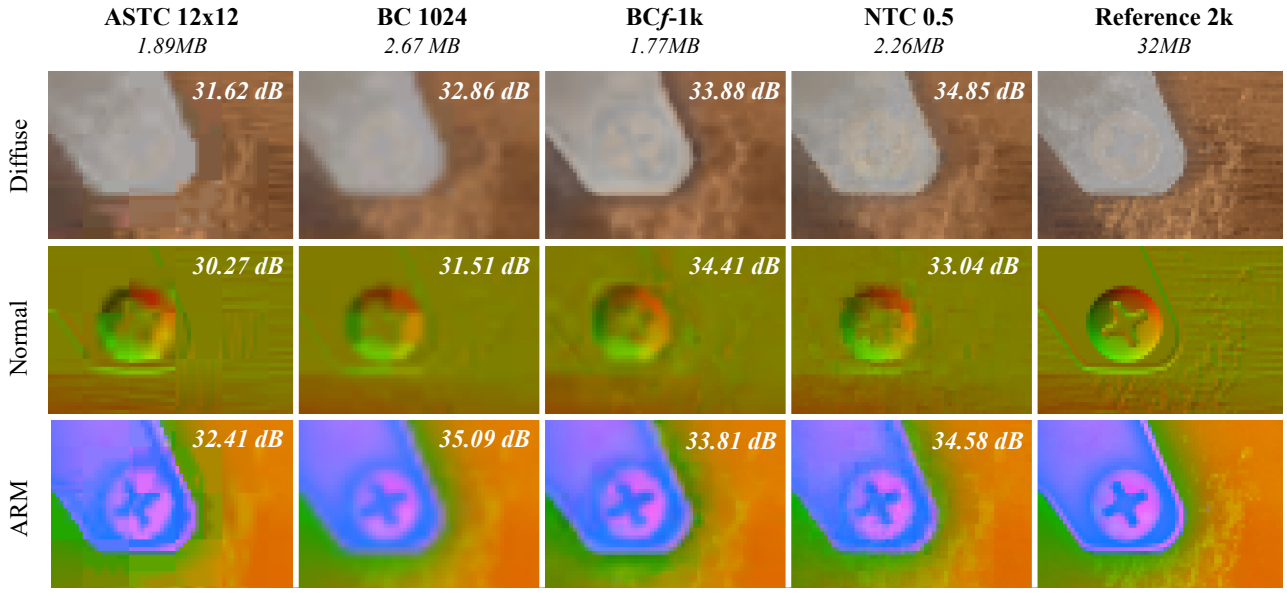


Figure 9: *Close-up crop of the Ukulele's material texture set. Comparison between ASTC, BC, our BCf-1K, NTC0.5 methods and the reference 2K material.*

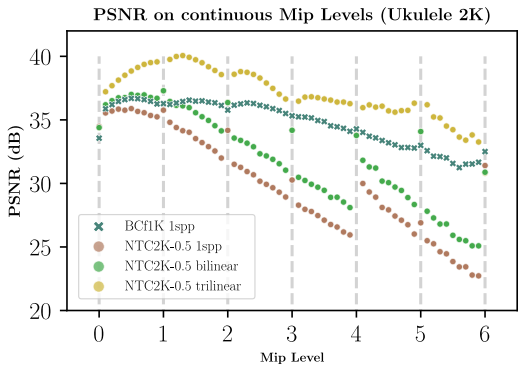


Figure 10: *Evolution of the PSNR reconstruction metric over the scales on the Ukulele's material.*

porting these features as BC6 textures makes their memory footprint small which makes it possible to increase their resolution. This allows us to rely on a very small and fast decoder network to reconstruct the material. More importantly, it enables the use of hardware texture filtering operations to sample the features at render-time which minimizes the overhead.

6.4. Quantization vs. Compression

As mentioned in section 3, there are two possible strategies that can be adopted when it comes to storing the learned neural features with fewer bits. We can either quantize their values or maintain them at a high bitrate and compress them. Both approaches are not straightforward to implement. Simply reducing the number of bits downgrades quality. This can be mitigated by increasing features's dimension but requires reducing the resolution to maintain manageable memory usage. For instance, it is possible to match the size of our BCf-1K profile with features of 4 channels quantized with two bits, or by dividing their resolution by two and increasing the number of channels to 8 and quantizing with 4 bits. Figure 13 plots the PSNR values of these configuration for the ukulele model and shows that, when pairing the features with a small network, it is more beneficial to have high bitrate features that are compressed than high dimensional low bitrate values. On average, at equal resolution, neural features with 3 channels stored according to be BC6H format performs similarly to ones with 5 channels and quantized with 4 bits resulting in a memory reduction of 40% (fig. 14). Increasing the number of channels and increasing the resolution without increasing the complexity of the network is not a viable strategy. The resolution of the features has a major impact on the reconstruction quality that cannot be compensated by increasing the number of channels and more importantly the larger input will overwhelm the small network and impact the result in a negative way. A larger network and more complex architecture are, thus, needed to fully take advantage of the higher number of input features and compensate for the lower resolution and bitrate as demonstrated by Vaidyanathan et al. [VSW*23].

Table 3: *Average time to render a textured full-screen quad using an NVIDIA RTX2070.*

	BC Textures	BCf-1K	
		total time	*neural overhead*
1080p	0.376 ms	0.555 ms	0.179 ms
1440p	0.614 ms	0.924 ms	0.310 ms
2160p	1.684 ms	2.324 ms	0.640 ms

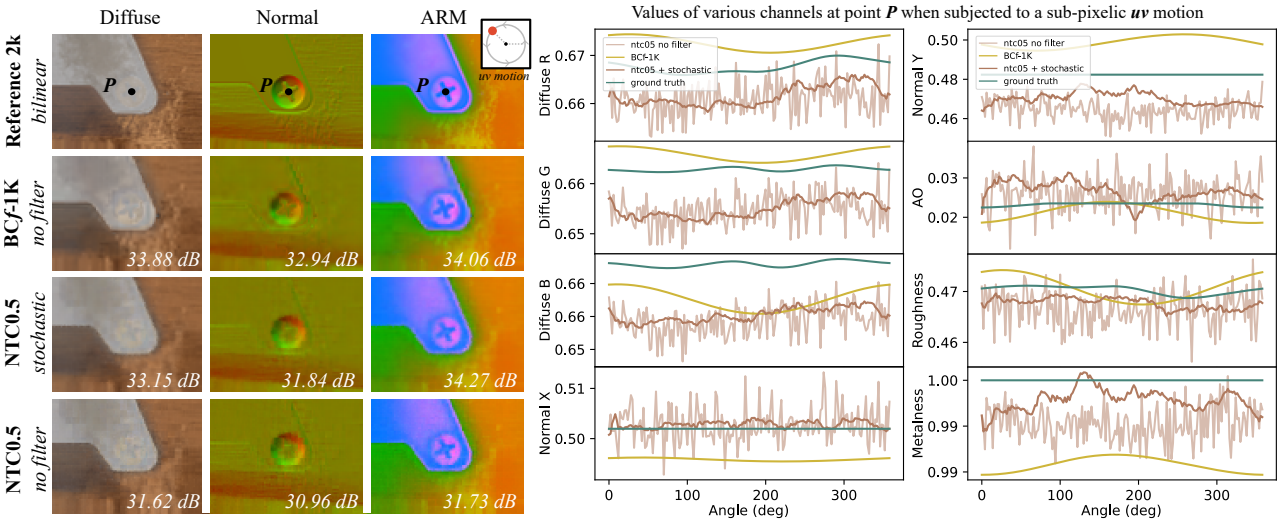


Figure 11: *Close-up crop of the reconstruction of the Ukulele's material when subject to a circular motion. We also track the values of each channel throughout time at point P. NTC is not stable when it is evaluated center of the pixel and requires some filtering which smoothen the result. Our approach does not require any filtering and output values that are stable temporally.*

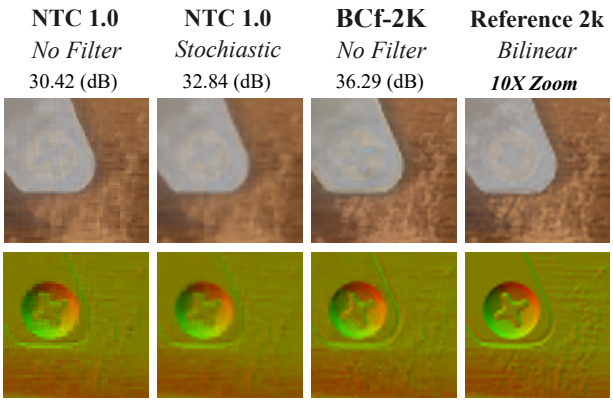


Figure 12: *Reconstuction the Ukulele's texture set with* 10× *magnification. Without any filtering NTC exhibit blocky artifacts and high frequency noise. Pairing it with a stochiastic filter will yield a smoother result. Our model can handle this without any filtering.*

6.5. Impact of BC partition selection

By design, the introduced BC6 constraints restrain the model capability compared to unconstrained features. The more the partition shape is aligned with the content of the data in a 4×4 block, the better the compression. Thus randomly selecting the partitions or forcing a static one is not optimal and will lead to lower reconstruction performance. Figure 15 shows that by initializing the partitions from pre-trained unconstrained neural features, as discussed in section 5.1, yields better result in the long run. In general, the warmup stage is not required to be long. We found that usually 5K iterations are sufficient for the shape of the unconstrained features to stabilise resulting in more suitable partitions. On our hardware, this takes up to 3 minutes.

7. Limitations and Future Work

In this section we discuss some of the limitations of our neural material model and offer avenues for future work.

Reconstruction artifacts Our reconstructed material can experience layer bleeding and color shift. Since our model is exploiting correlation between the channels and projecting the data into a smaller dimensional space, we believe these artifacts occur when there is small correlation or their absolute values are too divergent. In addition, the use of a decoding network consisting of only one hidden layer makes our model struggle with reconstructing and separating the data from the material layer. Even increasing the network size is hardly an option considering the potential overhead. It should be possible to overcome these issues by designing a decoder network that separates the reconstruction of uncorrelated layers. In addition to that, we could learn a tone mapping operator to map each output to the corresponding values. The challenge here, is to keep the decoding network small in order to maintain real-time performance, so we leave this to future work.

Higher quality reconstruction Depending on the profile, our model may struggle to learn high-frequency details which results in a slightly over-smooth reconstruction. This is a direct consequence of having neural features with only 3 channels and a very small decoder network, which prevent the effective use of positional encoding that could have contributed additional high frequency information as shown in [VSW*23]. Increasing the number of channels is key here. However, it is not straightforward in the context our BC neural features as the BC6H format is only capable of compressing RGB images. One way to overcome this is to group the high dimensional features into sets of 3 and process them independently. In this context, our BCf-2k++ configuration can be seen as one where we have three neural layers where the first one has 6 channels.

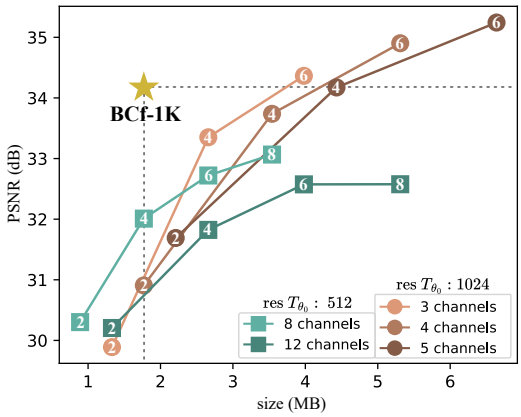


Figure 13: *The reconstruction performance of the Ukulele's material using neural features with various number of channels and quantization rate versus Block Compressed ones. The quantization rate is written in white.*

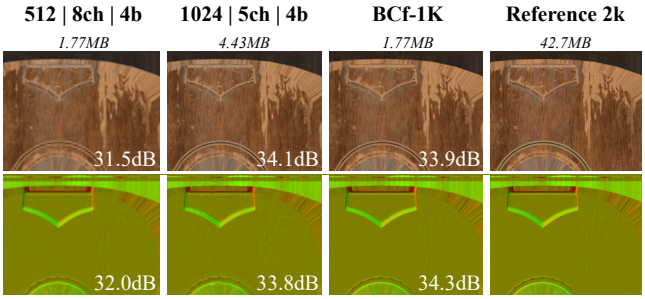


Figure 14: *Texture set reconstruction obtained with neural features of various configurations. Our BCf-1K configuration matches up visually with neural features of dimension 5 and quantized on 4 bits while being 40% smaller in size.*

Faster material encoding In this current version, our PyTorch implementation of the training pipeline requires about 140 minutes for 200K iteration steps. For now, we mainly focused on designing a model with a fast decoder which is critical for real-time applications such as video games. Reducing the training time is equally important as it is key in making such methods less computationally intensive and more practical in dynamic environments where the assets keep changing. There are several avenues that we can employ to improve on this. For instance, it is possible to train a generic encoder to initialize the embedding from the reference material in order to have a better starting point and train for fewer iterations as in [ZRW*23]. Additionally, we could improve the sampling strategy during training and sample only where it matters. This could also have an impact on the visual quality as the network will focus on minimizing the errors only where it matters.

Simulating more complex filtering Even with simple MLP decoders, our architecture is able to emulate simple texture filtering operations such as bilinear and bicubic filtering. We believe that this idea can be explored further by training models to emulate more elaborate filtering operations. For instance it would be possible to sparsely sample the neural textures in an anisotropic manner, i.e., along the major ellipse axis, and train them against a high density anisotropic filtered ground truth. This would make it possible to emulate high quality anisotropics filtering with few samples.

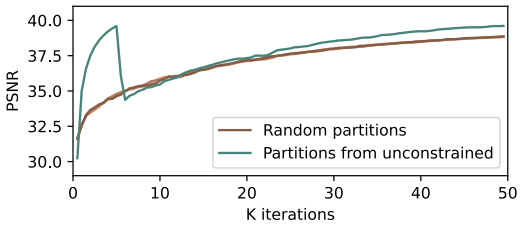


Figure 15: *Initializing the partitions of the neural BC feature from a set of unconstrained features yields better result in the long run. After 50k iterations, our initialisation heuristic is ahead by 1 PSNR unit compared to a randomized initialisation.*

8. Conclusion

This work introduces a novel block-compressed feature layer along with a continuous neural material encoding framework. This allows us to design an encoding and a decoding method that fits well within the real-time rendering pipeline constraints. We demonstrate the ability of our method to compress PBR materials efficiently and decompress them in real-time in a shader on consumer-level hardware. By taking into account memory and time constraints, and by taking advantage of existing hardware operations, we make neural textures usable at large scale in a real-time rendering pipeline, such as a video game engine. We hope that our work could serve as a starting point for future work and open the door for more complex use cases, such as learning directional materials or more complex filtering.

Acknowledgements

We thank Dr. Heqi Lu for all the discussions and advice and Arnaud Schoentgen for proofreading the paper. We also thank all the anonymous reviewers for their insightful comments.

References

[AMHH18] AKENINE-MLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering, Fourth Edition*, 4th ed. A. K. Peters, Ltd., 2018. 2

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. 7

[AVAB*19] ALAKUIJALA J., VAN ASSELDONK R., BOUKORTT S., BRUSE M., COMȘA I.-M., FIRSCHING M., FISCHBACHER T., KLIUCHNIKOV E., GOMEZ S., OBRYK R., ET AL.: Jpeg xl next-generation image compression architecture and coding tools. In *Applications of Digital Image Processing XLII* (2019), vol. 11137, SPIE, pp. 112–124. 2

[BMS*18] BALLÉ J., MINNEN D., SINGH S., HWANG S. J., JOHNSTON N.: Variational image compression with a scale hyperprior. In *International Conference on Learning Representations* (2018). 2

[CDF*86] CAMPBELL G., DEFANTI T. A., FREDERIKSEN J., JOYCE S. A., LESKE L. A., LINDBERG J. A., SANDIN D. J.: Two bit/pixel full color encoding. In *ACM SIGGRAPH conference proceedings* (1986), vol. 20. 2

[CLS*23] CHEN Z., LI Z., SONG L., CHEN L., YU J., YUAN J., XU Y.: Neurbf: A neural fields representation with adaptive radial basis functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2023), pp. 4182–4194. 3

[CRSL22] CHNG S.-F., RAMASINGHE S., SHERRAH J., LUCEY S.: Gaussian activated neural radiance fields for high fidelity reconstruction and pose estimation. In *European Conference on Computer Vision* (2022), Springer, pp. 264–280. 3

[CSTK20] CHENG Z., SUN H., TAKEUCHI M., KATTO J.: Learned image compression with discretized gaussian mixture likelihoods and attention modules. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2020), pp. 7939–7948. 2

[CXW*23] CHEN A., XU Z., WEI X., TANG S., SU H., GEIGER A.: Dictionary fields: Learning a neural basis decomposition. *ACM Trans. Graph.* (2023). 3

[D3D] Texture block compression in direct3d 11. https://learn.microsoft.com/fr-fr/windows/win32/direct3d11/texture-block-compression-in-direct3d-11. Accessed: 2023-09-28. 2, 5

[DNSD22] DATTA S., NOWROUZEZAHRAI D., SCHIED C., DONG Z.: Neural shadow mapping. In *ACM SIGGRAPH 2022 Conference Proceedings* (2022), pp. 1–9. 1

[Fen22] FENG X.: Real-time cluster path tracing for remote rendering. In *ACM SIGGRAPH 2022 Courses* (2022). 1

[FWH*22] FAN J., WANG B., HAŠAN M., YANG J., YAN L.-Q.: Neural layered brdfs. In *Proceedings of SIGGRAPH 2022* (2022). 2

[FWSP23] FAJARDO M., WRONSKI B., SALVI M., PHARR M.: Stochastic texture filtering, 2023. arXiv:2305.05810. 3

[HdR23] HILLAIRE S., DE ROUSIER C.: Authoring materials that matters - substrate in unreal engine 5. In *ACM SIGGRAPH 2023 Courses* (2023). 2

[HMB*20] HILL S., MCAULEY S., BELCOUR L., EARL W., HARRYSSON N., HILLAIRE S., HOFFMAN N., KERLEY L., PATRY J., PIEKÉ R., ET AL.: Physically based shading in theory and practice. In *ACM SIGGRAPH 2020 Courses*. 2020, pp. 1–12. 2

[iee19] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. 5

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and method for fixed-rate block-based image compression with inferred pixel values, 1999. 2

[KB14] KINGMA D. P., BA J.: Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014). 7

[KMX*21] KUZNETSOV A., MULLIA K., XU Z., HAŠAN M., RAMAMOORTHI R.: Neumip: Multi-resolution neural materials. *Transactions on Graphics (Proceedings of SIGGRAPH) 40*, 4 (July 2021). 3

[KSW21] KARIS B., STUBBE R., WIHLIDAL G.: A deep dive into nanite virtualized geometry. In *ACM SIGGRAPH 2021 Courses* (2021). 2

[KWM*22] KUZNETSOV A., WANG X., MULLIA K., LUAN F., XU Z., HASAN M., RAMAMOORTHI R.: Rendering neural materials on curved surfaces. In *ACM SIGGRAPH 2022 conference proceedings* (2022), pp. 1–9. 3

[MESK22] MÜLLER T., EVANS A., SCHIED C., KELLER A.: Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph. 41*, 4 (July 2022), 102:1–102:15. 3

[MHH*12] MCAULEY S., HILL S., HOFFMAN N., GOTANDA Y., SMITS B., BURLEY B., MARTINEZ A.: Practical physically-based shading in film and game production. In *ACM SIGGRAPH 2012 Courses* (2012). 2

[MHM*13] MCAULEY S., HILL S., MARTINEZ A., VILLEMIN R., PETTINEO M., LAZAROV D., NEUBELT D., KARIS B., HERY C., HOFFMAN N., ZAP ANDERSSON H.: Physically based shading in theory and practice. In *ACM SIGGRAPH 2013 Courses* (2013). 2

[MMT23] MAGGIORDOMO A., MORETON H., TARINI M.: Micro-mesh construction. *ACM Transactions on Graphics (TOG) 42*, 4 (2023), 1–18. 2

[MST*20] MILDENHALL B., SRINIVASAN P. P., TANCIK M., BARRON J. T., RAMAMOORTHI R., NG R.: Nerf: Representing scenes as neural radiance fields for view synthesis. 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics* (2012), pp. 105–114. 3, 7

[NVI] NVIDIA: Nvidia texture tools exporter. URL: developer.nvidia.com/nvidia-texture-tools-exporter. 7

[OLK*21] OUYANG Y., LIU S., KETTUNEN M., PHARR M., PANTALEONI J.: Restir gi: Path resampling for real-time path tracing. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 17–29. 1

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., BRADBURY J., CHANAN G., KILLEEN T., LIN Z., GIMELSHEIN N., ANTIGA L., DESMAISON A., KOPF A., YANG E., DEVITO Z., RAISON M., TEJANI A., CHILAMKURTHY S., STEINER B., FANG L., BAI J., CHINTALA S.: Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. 7

[RJGW19] RAINER G., JAKOB W., GHOSH A., WEYRICH T.: Neural btf compression and interpolation. In *Computer Graphics Forum* (2019), vol. 38, Wiley Online Library, pp. 235–244. 3

[SP23] SHIN S., PARK J.: Binary radiance fields. *arXiv preprint arXiv:2306.07581* (2023). 3

[SRRW21] SZTRAJMAN A., RAINER G., RITSCHEL T., WEYRICH T.: Neural brdf representation and importance sampling. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 332–346. 2

[TSM*20] TANCIK M., SRINIVASAN P., MILDENHALL B., FRIDOVICH-KEIL S., RAGHAVAN N., SINGHAL U., RAMAMOORTHI R., BARRON J., NG R.: Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in Neural Information Processing Systems 33* (2020), 7537–7547. 3

[TTM*22] TEWARI A., THIES J., MILDENHALL B., SRINIVASAN P., TRETSCHK E., YIFAN W., LASSNER C., SITZMANN V., MARTIN-BRUALLA R., LOMBARDI S., ET AL.: Advances in neural rendering. In *Computer Graphics Forum* (2022), vol. 41, Wiley Online Library, pp. 703–735. 3

[TZN19] THIES J., ZOLLHÖFER M., NIESSNER M.: Deferred neural rendering: Image synthesis using neural textures. *Acm Transactions on Graphics (TOG) 38*, 4 (2019), 1–12. 2

[VSW*23] VAIDYANATHAN K., SALVI M., WRONSKI B., AKENINE-MÖLLER T., EBELIN P., LEFOHN A.: Random-Access Neural Compression of Material Textures. In *Proceedings of SIGGRAPH* (2023). 2, 3, 4, 7, 8, 9, 10

[Wal92] WALLACE G.: The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (1992), xviii–xxxiv. 2

[WBSS04] WANG Z., BOVIK A., SHEIKH H., SIMONCELLI E.: Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612. 7

[WNK22] WRIGHT D., NARKOWICZ K., KELLY P.: Lumen: Real-time global illumination in unreal engine 5. In *ACM SIGGRAPH 2022 Courses* (2022). 1

[XWH*23] XU B., WU L., HAÅ¡AN M., LUAN F., GEORGIEV I., XU Z., RAMAMOORTHI R.: Neusample: Importance sampling for neural materials. In *ACM SIGGRAPH 2023 Conference Proceedings* (2023). 2

[ZRW*23] ZELTNER T., ROUSSELLE F., WEIDLICH A., CLARBERG P., NOVÁK J., BITTERLI B., EVANS A., DAVIDOVIČ T., KALLWEIT S., LEFOHN A.: Real-time neural appearance models. In *Proceedings of SIGGRAPH* (2023). 2, 3, 11

[ZZW*21] ZHENG C., ZHENG R., WANG R., ZHAO S., BAO H.: A compact representation of measured brdfs using neural processes. *ACM Transactions on Graphics (TOG) 41*, 2 (2021), 1–15. 2