CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Neural Compression of Material Properties using a Geometry-Associated Feature Hierarchy

Master's thesis in Computer science and engineering

Pascal Walloner

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Neural Compression of Material Properties using a Geometry-Associated Feature Hierarchy

Pascal Walloner

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Neural Compression of Material Properties using a Geometry-Associated Feature Hierarchy

Pascal Walloner

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisors: Magnus Olausson & Tore Levenstam, Rapid Images AB
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Neural Compression of Material Properties using a Geometry-Associated Feature Hierarchy

Pascal Walloner
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Texture maps are ubiquitous in 3D rendering to encode material properties like albedo color, surface normal vectors, roughness coefficients and many more. Applying them to the surface of a 3D object requires a mapping from the object's surface to the flat texture plane (*uv*-mapping), which may introduce artifacts through unavoidable distortion and seams.

In this work, we propose a novel neural approach to encoding surface material parameters without the need for *uv*-mapping. We build on recent research in the field of neural function approximation in computer graphics [1] [2], which achieves efficient compression of texture data by training a machine learning model. By parameterizing surface positions in relation to their mesh triangle, we adapt previous approaches to circumvent the *uv*-mapping step.

The evaluation of our prototype shows that our method is capable of encoding detailed, high-resolution textures at satisfying quality, while encoding multiple material channels in a single representation. We evaluate our method on a selection of datasets with a broad range of geometry and texture characteristics. We observe that certain characteristics challenge our method more than others. Compression rates range from 66.6% to 8.3% across the examined datasets.

Our outlook discusses, among other points, how limitations regarding subpar performance on meshes with low vertex-density could be overcome in future work. Furthermore, we lay out a possibility how our method's hierarchical structure could be leveraged to realize low-pass texture filtering.

Keywords: texture compression, machine learning, neural networks, input encoding, GPU, *uv*-mapping.

Acknowledgements

I would like to thank my supervisor Erik Sintorn and my company advisor Tore Levenstam at Rapid Images, for their valuable guidance and support during the work on this project.

Pascal Walloner, Gothenburg, 2024-06-26

Contents

List of Figures

List of Tables

1

Introduction

Texture maps are a ubiquitous tool in 3D rendering to encode material properties on object surfaces. In modern applications they represent not only color, but also surface normal vectors or other material parameters, e.g. surface roughness, or specular reflectivity. Using texture maps with the goal of rendering close to realistic images comes with two major challenges:

Firstly, using texture maps requires a mapping of the 3D object's surface to the flat texture plane. Such a mapping is called *uv*-mapping. Creating it typically requires flattening curved parts of the surface, which introduces distortion. Other sections of the mesh might be separated entirely, turning edges between them into seams. Distortion can lead to artifacts through unevenly distributed texture space, while seams can cause discontinuities on the surface. Finding a *uv*-mapping that minimizes artifacts caused by distortion and seams, or places them in regions where they are less noticeable, is a task that generally requires artist input. It is desirable to minimize the time artists spend on this task, as it entails little artistic expression but is primarily a necessity that arises from the technical limitations of *uv*-mapped texture maps.

The second challenge that comes with the use of texture maps is compression. Modern renderers with the goal of producing realistic images require high-resolution texture maps, which typically consume large amounts of memory. However, memory in fast low-level caches on the GPU is limited. Thus, efficient compression of texture data is necessary. Vaidyanathan et al. [2] propose a neural approach to texture compression. They train a machine learning model to fit the texture function, mapping (u, v) texture coordinates to a vector of material properties (color, normal, roughness etc.). The model they construct is based on previous research by Müller et al. [1], who associate learnable parameters (called features) with positions in input space. In the case of texture compression, features are associated with points on 2-dimensional grids that span the texture plane. Using their neural approach, Vaidyanathan et al. [2] achieve efficient compression rates and propose a model that serves as a single representation for all material channels. This means that one model can encode albedo color, normal vectors and all other channels, eliminating the need for multiple texture maps. However, since the model is queried by (u, v) texture coordinates, it still requires a mesh whose surface is *uv*-mapped to the texture plane.

With this work, we aim to develop a novel neural compression technique that asso-

ciates features with points on the object's surface instead. This way, we adapt the approach by Müller et al. [1] to work without the need for explicitly *uv*-mapped meshes. Instead, we implicitly parameterize the mesh surface and use this parameterization to access the relevant features. By not requiring meshes to be *uv*-mapped we not only remove the creation of *uv*-mappings from the artists' work flow, but also completely circumvent the problem of artifacts arising through distortion and seams in the mapping. At the same time, we aim to benefit from the advantages of neural texture compression, specifically, good compression rates while storing all material channels in a single representation.

2

Fundamentals

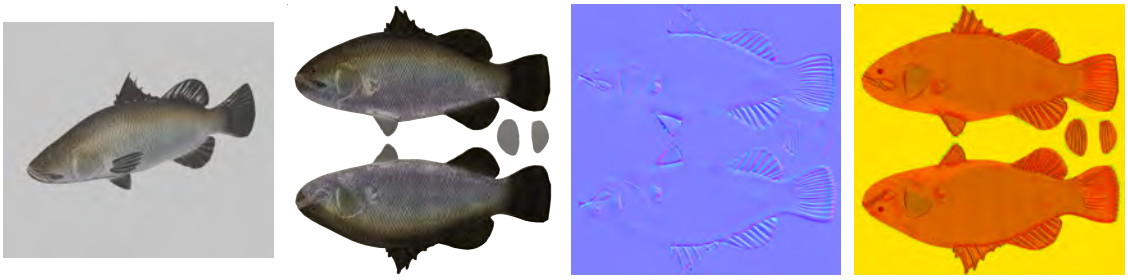
This chapter highlights important prerequisites to neural material compression.

2.1 Texture Mapping

Texture maps are images containing data about an objects surface properties [3]. They are typically 2-dimensional images which the surface of the object is mapped to. This mapping is called texture- or *uv*-mapping and is realized by giving each vertex a 2D coordinate (u, v) representing its position in the texture plane. Surface points between vertices are then mapped by interpolating adjacent vertices' *uv* coordinates. This way, surface detail can be represented at a greater resolution than the model's vertex density.

Defining a good *uv*-mapping for a given object is no trivial task. It is often a trade-off between using the rectangular real-estate of the texture map as efficiently as possible and minimizing distortion and seams in the mapping that lead to unpleasant artifacts. For objects with complex surface geometry it is often topologically impossible to find a mapping that is free from distortion or seams. In this case, the artist has to carefully distribute them to locations where they have little impact on the final appearance of the object.

Despite this, texture maps are still very commonly used, as they are a versatile tool to encode all kinds of surface properties. The most common property texture maps encode is surface color. Texture maps are also used to encode surface properties that play a role in simulating the interaction of light with object surfaces, especially since physically-based rendering (PBR) methods have gained traction. These include surface normal vectors, as well as coefficients like roughness, metallicness or specular reflectivity. Texture maps are also commonly used to store pre-computed lighting information through techniques like ambient occlusion or shadow mapping. Texture maps typically store information in four channels: red, green, blue and alpha (r, g, b, a). If more than four channels are required to store surface detail, it is necessary to use multiple texture maps. Figure 2.1 shows what the unwrapped texture maps for different channels of material properties of a mesh may look like.



(a) (b) (c) (d)

Figure 2.1: A rendered image (a) of the *BarramundiFish*-model [4] and its unwrapped texture maps: (b) albedo channels (r, g, b), (c) normal channels (n_x, n_y, n_z), (d) metallic, occlusion and roughness channels.

2.2 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are one of the simplest and most versatile kinds of artificial neural network models. This section briefly outlines their functionality. For a more detailed explanation, we refer the reader to Lindholm et al.'s book [5] on machine learning techniques.

A perceptron maps an input vector $\mathbf{x}$ to a scalar output using a function

$$f(\mathbf{x}) = a(\mathbf{w} \cdot \mathbf{x} + b),$$

where the weight vector $\mathbf{w}$ and bias b are learnable parameters. $a : \mathbb{R} \to \mathbb{R}$ is a typically non-linear activation function breaking up the model's linearity and allowing it as a whole to represent non-linear functions. Typical choices for activation functions are *sigmoid*, *tanh* or rectified linear unit (*ReLU*).

MLPs are perceptrons structured in a layered network, where the concatenated outputs of the perceptrons in layer n form the input vector to all perceptrons in layer $n + 1$. The first layer is called the input layer. Its activations represent the input to the model. It is followed by a number of hidden layers and an output layer. A network like this is called fully connected. We denote an MLP as a function $\mathbf{y} = \mathbf{mlp}(\mathbf{x}, \phi)$, where ϕ denotes the learnable parameters, meaning weights and biases of all perceptrons. Figure 2.2 shows a schematic of the information flow in an MLP.

The function an MLP represents is defined by its learnable parameters $\mathbf{w}$ and b. To find values for these, the model is trained on a training data set. This data set contains numerous examples of correct input-output-pairs. For each input instance $\mathbf{x}$ in the training set, the corresponding output label $\mathbf{y}_{\text{lab}}$ is compared to the models predicted output $\mathbf{y}_{\text{pred}}$ using a loss function $L(\mathbf{y}_{\text{lab}}, \mathbf{y}_{\text{pred}})$. A typical choice for L is the mean squared error (MSE) function. After each iteration, learnable parameters are updated using gradient descent to minimize L. Gradient descent updates a parameter by subtracting the partial gradient of the loss function w.r.t the parameter. Determining partial gradients happens via backpropagation: First, partial gradients w.r.t each of the output perceptrons' activations are computed. These serve as the

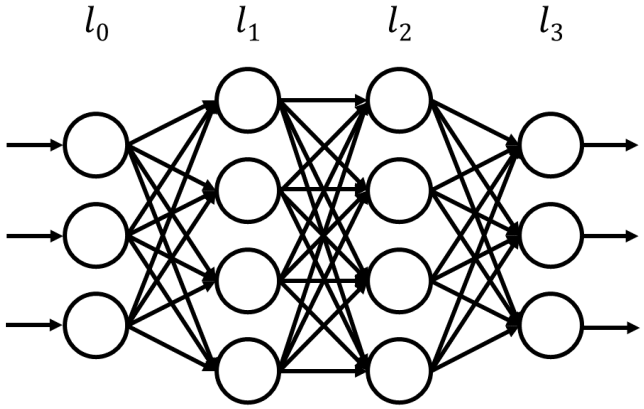


Figure 2.2: A multilayer perceptron with a 3-node input layer l_0, two 4-node hidden layers l_1 and l_2, and a 3-node output layer l_3. Arrows denote how a node's activation is fed forward to each node of the next layer, making the network fully connected. Each connecting arrow is associated with a weight acting as a factor to the previous activation.

basis to compute the partial gradients w.r.t the output perceptrons' parameters. Recall that a layer's parameters are composed of its weights $\mathbf{w}$ and biases b as well as the previous layer's activations. Gradients w.r.t weights and biases represent the updates to the respective learnable parameters, while gradients w.r.t the previous layer's activations allow for a recursive repetition of the algorithm to determine updates to learnable parameters of all layers. This recursion is the core of the backpropagation algorithm.

Having computed all parameters' gradients, updates occur by multiplying the partial gradients with a learning rate λ and subtracting the product from the parameters' current values. In the beginning, with randomly or zero-initialized parameters, updates will be quite drastic as the model's predictions will likely differ greatly from the desired output $\mathbf{y}_{\text{lab}}$. After a number of iterations, however, the model will converge to a local minimum of the loss function L.

If the model is able to represent the function sampled by the training data and does not suffer from overfitting to deviations in the training data, it will serve as a good estimator after convergence. Whether or not this is the case is depends greatly on layer width and number of layers in the network, as well as the characteristics of the training data.

2.3 Input Encodings

The characteristics of the data the MLP is trained on have a significant impact on approximation quality. MLPs represent low-frequency features significantly better than those of high frequency [6]. This is especially disadvantageous in the context

of computer graphics, where a common use case for MLPs is to represent images or texture data, meaning, passing a position $\mathbf{x}$ in 2D space to the model should yield an output representing the color of the image at $\mathbf{x}$. In this case, the model's bias toward low-frequency features would express itself through overblurring artifacts, hiding high-frequency detail in the approximation.

To combat this bias, it is useful to encode the input to the MLP in a way that hides the frequency of characteristics in the data. Instead of evaluating the MLP directly on the input, the input is passed through an encoding function **enc** before:

$$\mathbf{y} = \mathbf{mlp}(\mathbf{enc}(\mathbf{x}), \phi) \tag{2.1}$$

This input encoding can be thought of as a prepended layer to the MLP. Input encodings typically map inputs to some higher dimensional space where distances between encoded inputs do not correlate strongly with distances between original inputs. Mildenhall et al. [7] encode positions in 3D space using a frequency encoding

$$\gamma(p) = (\sin(2^0\pi p), \cos(2^0\pi p), ..., \sin(2^{M-1}\pi p), \cos(2^{M-1}\pi p)), \tag{2.2}$$

where $\gamma(p)$ is applied to each entry p of the input vector $\mathbf{x}$.

Encodings like this achieve a decoupling of frequencies between the original and encoded input space. However, they do not contain any learnable parameters, i.e. all information gained during training needs to be stored in the networks weights and biases. In many cases, it is more efficient to store learnable parameters in a more task-specific way as part of the encoding layer. This allows for much smaller network sizes, which greatly benefits training and evaluation performance.

We denote such a parameterized encoding by an encoding function $\mathbf{enc}(\mathbf{x}, \theta)$ with encoding parameters θ. To train input encoding parameters, we compute the encoding functions gradient $\frac{\partial}{\partial\theta}\mathbf{enc}$ with respect to the learnable parameters. After the MLP's backward pass, the computed gradients are used to backpropagate input layer gradients to the encoding parameters θ and update them analogously to how weights and biases in the MLP itself are updated. This points out differentiability as an important requirement to the encoding function.

One such parametrized encoding is the *Multiresolution Hashgrid Encoding* (MHE) proposed by Müller et al. [1]. They distribute learnable parameters, called features, that are associated with nodes in a hierarchy of grids that span the input space. To encode an input position they interpolate the feature vectors associated with the corners of the surrounding grid cell at each hierarchy level. Since linear interpolation within grid cells is trivially differentiable, the encoding parameters' gradients are efficient to compute. Figure 2.3 shows an overview of their algorithm.

As the core part of this work, we develop and evaluate a task-specific parameterized input encoding to encode surface positions on 3D meshes by interpolating feature vectors in a similar way. We follow Müller et al.'s [1] approach in that we use linearly interpolated feature vectors to encode surface positions. However, instead of naively spanning object space with a 3D MHE grid hierarchy, our technique performs interpolation on a mesh-associated grid hierarchy of triangles spanning the objects

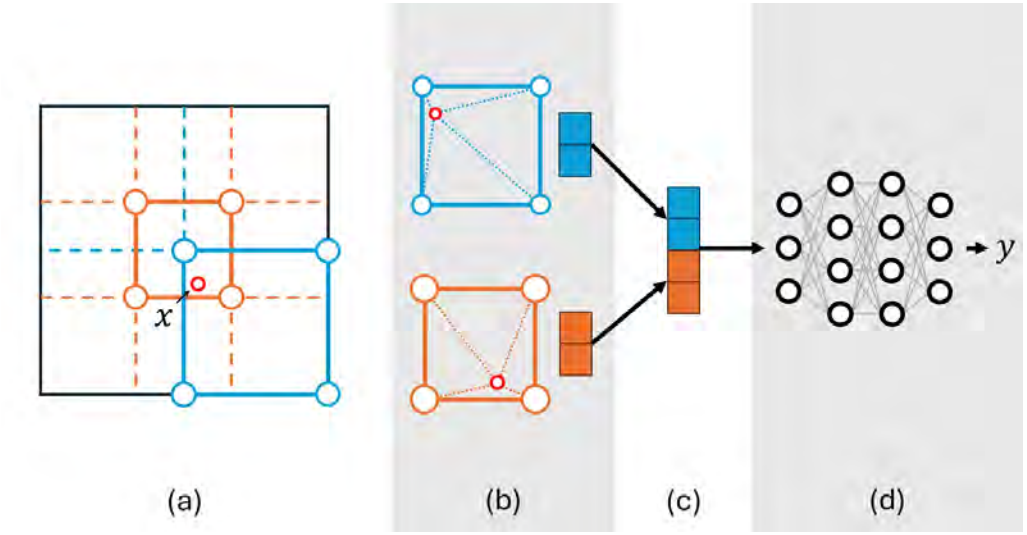


Figure 2.3: Overview of the multiresolution hash encoding (MHE) by Müller et al. [1]. (a) Sample position x is mapped to the surrounding cell on each level of the grid hierarchy. (b) Feature vectors associated with the corners of the grid cell are interpolated at each hierarchy level. (c) Interpolated feature vectors from each level are concatenated into one vector, forming the encoded input. (d) The encoding's output serves as the input to the MLP. In the backward pass, input layer gradients are backpropagated to update feature vectors.

surface. This allows for a geometry-aware encoding of surface positions preventing the misinterpretation of spatial correlations between unrelated surface positions, which is a common problem in approaches that aim to encode surface properties in 3D data structures [8] [9]. The feature vector-based approach, however, allows us to employ a small MLP of only a few narrow layers. Müller et al. [10] develop a, so called *fully fused* network architecture, which allows for significantly faster training and evaluation times of small MLPs. Encoding surface positions - regardless if parametrized in object space or on the surface domain - without learnable feature vectors, e.g. using a frequency encoding [7], requires a significantly larger MLP as all texture information needs to be captured in its weights and biases. These network sizes would far exceed the limitations of the *fully fused* architecture and lead to significantly worse throughput, making real-time applications impossible.

3

Related Work

3.1 Neural Function Estimators in Rendering

The field of neural function approximation in computer graphics has made numerous significant advances in recent years. Mildenhall et al. [7] lay groundbreaking foundations with their work on *Neural Radiance Fields* (NeRFs). NeRFs are a neural architecture that aims to approximate the scene radiance function, which maps spatio-directional coordinates $(\mathbf{x}, \omega)$ to the appropriate radiance (r, g, b). The model Mildenhall et al. [7] propose essentially consists of an MLP. As described in section 2.3, MLPs suffer from a strong bias toward learning low-frequency signals [6], which they solve using a frequency-based positional input encoding.

Müller et al. [1] build on this by proposing MHE (see section 2.3). By delegating learnable parameters to the input encoding, they allow the MLP to be significantly smaller, improving training and evaluation performance.

A substantial amount of recent research has been conducted on how task-specific input encodings with learnable parameters can benefit neural rendering techniques. Vaidyanathan et al. [2] build on MHE to develop a neural architecture that captures 2D textures at different detail levels. They use correlations between different channels of material properties to efficiently compress them into a single neural representation. While they achieve good compression rates, texture data is still accessed via traditional *uv*-coordinates. Liu et al. [11] propose an optimized encoding for sparse field data. Instead of a uniform grid, they associate feature vectors with nodes in a sparse voxel octree [12]. This way they use less memory for features in regions with little relevant scene content. Takikawa et al. [13] use an octree-based structure to construct an efficient neural representation of signed distance functions.

Müller et al. [10] achieve great strides in training and evaluation performance of narrow MLPs. They argue that for narrow networks, such as those used in conjunction with a parametrized input encoding, the limiting factor for performance on modern GPUs is memory traffic rather than computational cost. On-chip memory like low-level caches or shared memory of modern GPUs is significantly faster than global memory or VRAM. The authors achieve a significant speedup by splitting large batches into small chunks that can be processed in parallel by one thread block each. These are small enough that the narrow layers' weights and activations fit into fast on-chip memory, cutting down on slow traffic to global memory. With

their work on MHE [1], they allow for the use of sufficiently small MLPs in many applications to benefit from their *fully fused* architecture [10].

We use their framework of fast *fully fused* MLPs [14] in our prototype as our input encoding also allows for small enough MLPs.

3.2 Alternatives to Manual *uv*-Mapping

Dolonius et al. [9] aim to circumvent *uv*-mapping by following an approach to encode surface properties in 3D model space using sparse voxel DAGs [15], a spatial data structure generalized from sparse voxel octrees [12]. However, Benson et al. [8] mention a crucial challenge with object space surface parameterization. Proximity of two points in object space does not necessarily provide any information about their proximity on the mesh surface. A thin, two-sided wall exemplifies this well. Points on opposite sides of the wall might only be separated by its thickness, while being on completely unrelated parts of the surface. If the spatial data structure fails at distinguishing between them, points on one surface might falsely be attributed to the other, leading to erroneous cross-surface interpolation artifacts.

Other work has aimed at decreasing the time spent by artists manually *uv*-mapping surfaces of 3D meshes by streamlining the process. Approaches range from general to highly task-specific. Chen et al. [16] propose a neural network model which learns to map 3D surfaces to 2D *uv*-space. Deng et al. [17] propose a more task-specific approach for *uv*-mapping models of human faces with images captured from facial recognition. They fit a 3D morphable model to a dataset of multiview images and use it in combination with a deep convolutional neural network to complete *uv*-mappings extracted from single in-the-wild images. Earlier work by Tarini et al. [18] devises a function that maps each surface point to texture space, solely based on its 3D coordinate. They adapt this function to a parametrization of the mesh surface. Thus, the mapping can be reused for geometrically similar meshes, such as different LODs of an object, independent of the exact tessellation. Poranne et al. [19] formulate *uv*-mapping as a minimization problem. Their method generates distortion-minimizing *uv*-maps from 3D geometry and constraints input by the user in an interactive workflow. These works provide significant advances to the automation of the *uv*-mapping process. However, the inherent drawbacks of mapping 3D surfaces to the 2D texture plane remain, especially for meshes with intricate geometry. Distortion and seams are generally unavoidable. With this work we aim to circumvent the need for *uv*-mapping entirely by training a neural network model to fit a function that maps positions parameterized on the mesh's surface to texture parameters directly.

4

Method

We represent material properties as a vector-valued function Φ which maps a position $\mathbf{x}$ on the object's surface to a vector of material properties $\mathbf{y} \in \mathbb{R}^n$. In a simple case, $\mathbf{y}$ might be a 3-dimensional vector representing the red, green and blue channel of the object's surface color. In more complex cases $\mathbf{y}$ is a larger vector containing more material properties, such as surface normals, roughness or specular reflectivity.

We encode Φ using a multi-layer perceptron (MLP), i.e. evaluating the MLP at a given surface position $\mathbf{x}$ should yield the corresponding vector of material properties $\mathbf{y}$. However, MLPs show a strong spectral bias toward learning low-frequency functions [6]. Thus, using surface position in linear space as an unencoded input to the network is not viable to represent texture data containing high-frequency detail. Strong overblurring artifacts would be the undesirable result. Furthermore, capturing all characteristics of a detailed high-resolution texture would necessitate a large network with enough parameters to store all the required information. The depth of the network has a strong influence on training and evaluation times. Additionally, narrow networks allow for efficient memory use when implemented on modern GPUs as shown by Müller et al. [10]. In short, it is highly desirable to keep the MLP as small and with as little parameters as possible. To fight the spectral bias, but at the same time allow for the representation of detailed, high-resolution surfaces, we follow the approach by Müller et al. [1] to outsource learnable parameters to a prepended input encoding. We store these as a set of learnable feature vectors, each associated with a position on the mesh surface. This way, we compute $\mathbf{y} = \Phi(\mathbf{x})$ using the composition of a parameterized encoding function $\mathbf{e} = \mathbf{enc}(\mathbf{x}, \theta)$ and an MLP $\mathbf{y} = \mathbf{mlp}(\mathbf{e}, \phi)$ with learnable parameters θ and ϕ.

This chapter describes the construction of $\mathbf{enc}(\mathbf{x}, \theta)$ in detail.

Hyperparameter Name	Symbol	Value
Number of entries per feature vector	F	2 to 8
Number of hierarchy levels	L	4 to 8
Hash table size per hierarchy level	T	2^{17} to 2^{22}
Number of quantization bins	N	16, 256

Table 4.1: List of hyperparameters to the input encoding. If feature quantization is disabled, parameter N is irrelevant.

The encoding function depends on a number of hyperparameters, described in the

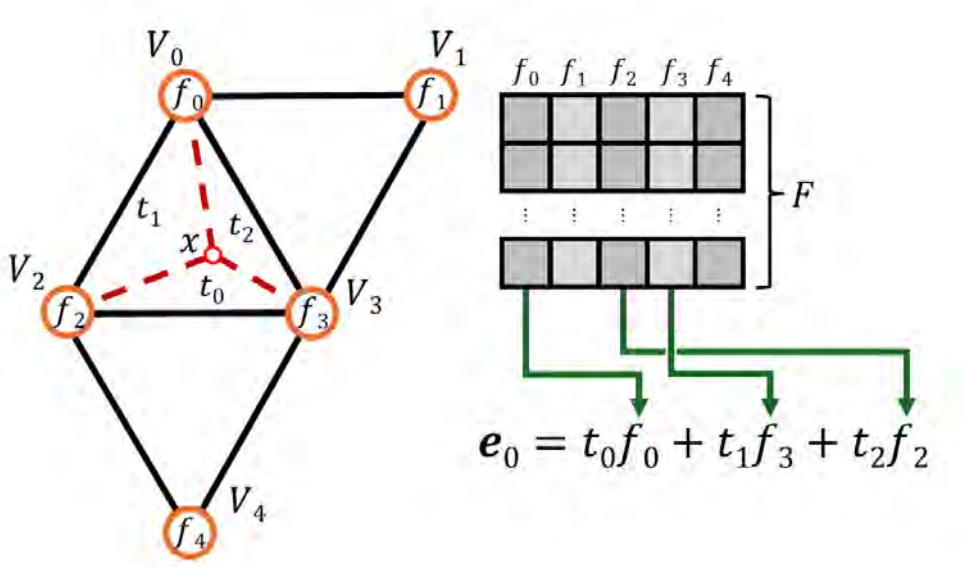


Figure 4.1: Feature vector interpolation on the coarsest hierarchy level: Sample position $\mathbf{x}$ is located on mesh triangle (V_0, V_3, V_2). Associated feature vectors f_0, f_3 and f_2 are interpolated with $\mathbf{x}$'s barycentric coordinates t_0, t_1 and t_2 as weights to compute the coarsest component $\mathbf{e}_0$ of the encoding output.

following sections. Table 4.1 gives an overview of these along with their symbols and ranges of typical values.

4.1 Geometry-Associated Feature Vectors

The input to the encoding is a position $\mathbf{x}$ on the mesh's surface. More specifically, $\mathbf{x}$ is a tuple $(k, \mathbf{t})$ of an integer k serving as an identifier to the mesh triangle $\mathbf{x}$ is on and a position on the triangle expressed using barycentric coordinates $\mathbf{t} = (t_0, t_1, t_2)$ [20].

The encoding works with learnable parameters called features. They are vectors of set length F and are associated with points in the mesh's geometry. At the coarsest level, feature vectors are linked to mesh vertices. To determine the encoded output $\mathbf{e}_0$ at the coarsest level, feature vectors at vertices belonging to triangle k are linearly interpolated according to barycentric coordinates $\mathbf{t}$ (see figure 4.1). Feature vectors at vertices are shared between adjacent triangles. Thus, feature vector interpolation is continuous across triangle edges. If this was not the case, discontinuities between triangles would be visible in the output.

Since linear interpolation is trivially differentiable, input layer gradients of the MLP computed during the backward pass can be propagated to feature vectors, making them learnable parameters. After training convergence, most material property information is held in feature vectors instead of network weights. This allows for a very small network with few narrow layers.

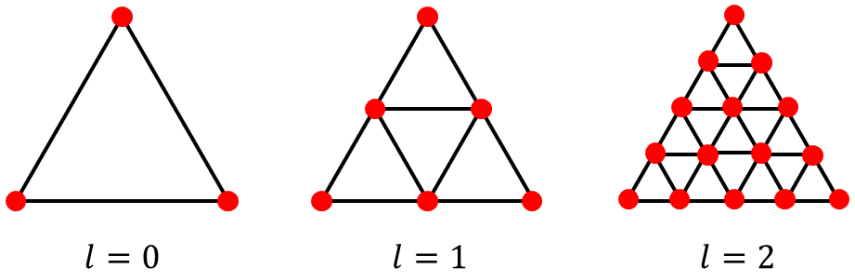


Figure 4.2: Feature Hierarchy: Hierarchy level l's feature vector positions (red) on each triangle are defined by the vertex positions of the $2^l - 1$ subdivided triangle.

4.2 Feature Hierarchy

As feature vectors contain the majority of the information required to reconstruct surface detail, interpolating them on a coarse scale leads to strong interpolation artifacts, such as overblurring. For this reason we employ a hierarchy of feature vectors. The hierarchy is comprised of L levels where feature vectors in each level capture detail of a different granularity.

Each hierarchy level is constructed by subdividing mesh triangles according to Loop's subdivision scheme [21]. Loop's subdivision scheme subdivides mesh triangles into identical subdivision triangles that are congruent to the original. Feature vector positions are then defined by the vertex positions in the subdivided mesh. Note that no actual mesh subdivision occurs. We merely use the subdivision scheme to define positions that feature vectors are associated with. We call a Loop-subdivided triangle with n additional edge vertices per edge n-Loop-subdivided. Feature vector positions at each hierarchy level l are defined by the vertex positions of the $2^l - 1$-Loop subdivided version of each mesh triangle. Figure 4.2 shows the structure of a Loop-subdivided triangle at levels $l = 0, 1, 2$.

Loop's subdivision scheme is especially suitable as it distributes subdivision vertices - and thus feature vectors - evenly and creates non-elongated triangles. These properties ensure stable and uniform feature vector interpolation at each hierarchy level. Furthermore, the fact that subdivision triangle edges are parallel to those of the original triangle allows for an easy mapping of barycentric coordinates (t_0, t_1, t_2) to the corresponding subdivision triangle. Some subdivision triangles (e.g. the center one at $l = 1$) are rotated by 180 degrees compared to the mesh triangle. We call these non-edge-aligned.

Feature vector interpolation happens for all hierarchy levels in parallel and independently. To determine the interpolated feature vector $\mathbf{e}_l$ at hierarchy level l, $\mathbf{x}$ is mapped to the subdivision triangle it falls onto at level l. Interpolation is then carried out between feature vectors at the subdivision triangle's vertices using barycentric coordinates on the subdivision triangle as weights. Interpolated feature vectors $\mathbf{e}_l$ across all levels are then concatenated to form the final encoding output vector $\mathbf{e}$. Figure 4.3 shows an overview of this process.

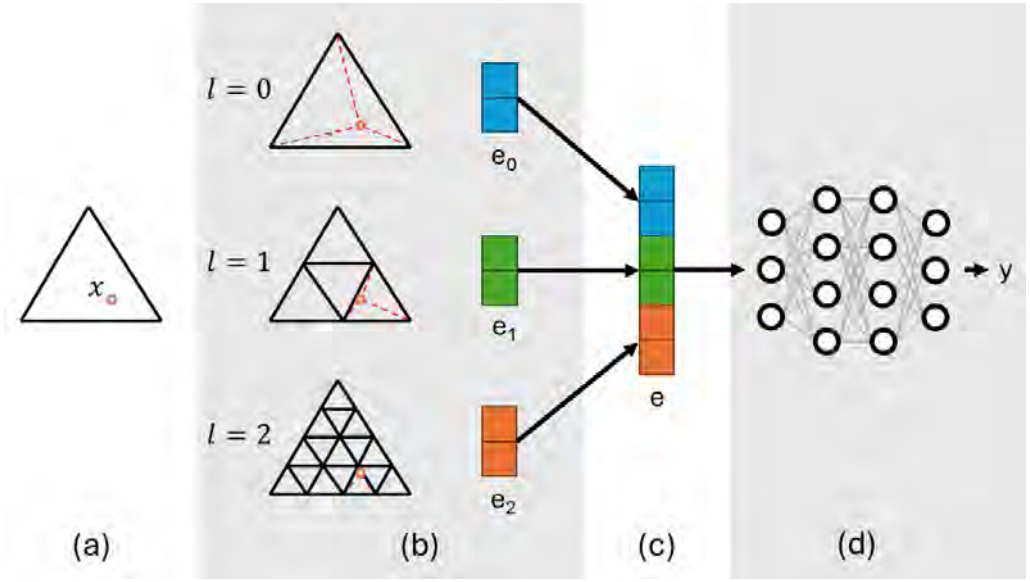


Figure 4.3: Overview of the model: (a) Sample position $\mathbf{x}$ on mesh triangle. (b) At each hierarchy level l: interpolate adjacent feature vector entries according to subdivision to calculate $\mathbf{e}_l$. (c) Construct encoded input $\mathbf{e}$ by concatenating $\mathbf{e}_0$ through $\mathbf{e}_{L-1}$. (d) Use $\mathbf{e}$ as input to MLP to compute $\mathbf{y} = \mathbf{mlp}(\mathbf{e}, \phi)$.

Note that feature vectors at different hierarchy levels are fully distinct - even if they are associated with the same surface position - as they contain information about texture detail at different scales of granularity. Furthermore, it is important to ensure that feature vectors that influence interpolation for samples on several mesh triangles (i.e. those at mesh vertices or along mesh edges) are shared. If feature vectors at mesh edges were kept distinct per mesh face, edge discontinuities occur in the encoding output. These can not be sufficiently resolved by the MLP. Section 5.1.3 discusses this in more detail.

4.3 Feature Hashing

With each finer hierarchy level the number of feature vectors it is comprised of grows exponentially. To limit the impact of this on the memory footprint of the encoding's parameters, Müller et al. [1] suggest a hard limit T to the number of feature vectors per level. The total number of learnable encoding parameters is thus bounded by $T \cdot L \cdot F$. At coarse levels with fewer than T feature vectors this limit poses no problem and each feature vector can be stored independently. At finer levels where the number of feature vectors exceeds T, they use a hash function to map each of them to an entry in a hash map of exactly T feature vectors. In this case hash collisions between feature vectors occur. Note that collisions only occur when samples refer to feature vectors associated with *different* surface positions, but those feature vectors map to the same position in the hash map. It is *not* a collision if two samples refer to the feature vector at the same surface position and thus access the same hash map location.

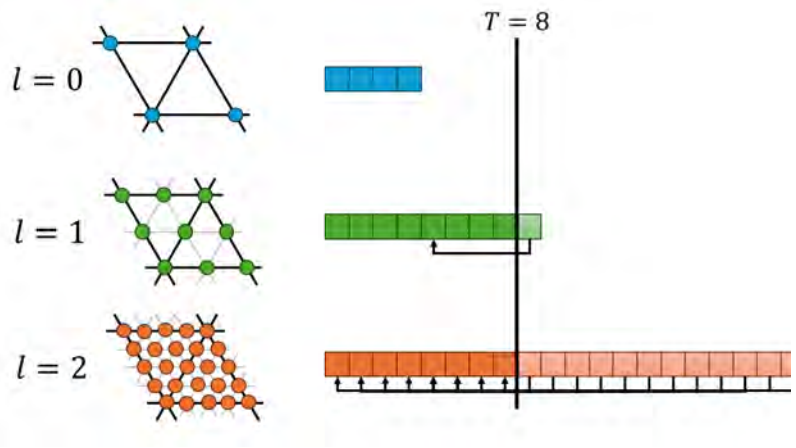


Figure 4.4: Schematic of a feature hashing example: As the coarse hierarchy level $l = 0$ contains fewer than T feature vectors, each of them is stored separately and no collisions occur. Levels $l = 1$ and $l = 2$ contain more than $T = 8$ feature vectors. Thus they are mapped to locations in a hash map with T entries.

Müller et al. [1] do not resolve these collisions explicitly. When two training samples collide, their updates to the shared hash map entry average. Thus the more important sample, i.e. the one with the larger absolute gradient w.r.t. the feature vector entry, will dominate during training. They argue that this is sufficient as, at different hierarchy levels, hash collisions are likely to occur between different feature vectors. Thus detail that can not be properly represented on one level due to a hash collision will likely be captured by another one. Remaining hash collision artifacts can also be corrected by the subsequent MLP.

To reduce the memory footprint of our model's features, we employ a similar hashing approach. The original MHE [1] uses a hash function $h : \mathbb{Z}^d \to \mathbb{Z}_T$ mapping integer coordinates of feature vectors in a d-dimensional grid to hash map locations. Since we do not associate feature vectors with positions in uniform grids, but rather surface positions $(k, \mathbf{t})$ on 3D meshes, given by an integer triangle ID k and barycentric coordinates $\mathbf{t}$, an adapted hash function is necessary. This hash function needs to fulfill a number of properties:

- Samples on adjacent faces may refer to the same feature vector for interpolation if it is associated with a position along a shared edge or at a shared vertex. Positions on shared edges or vertices can be validly named with k being any of the adjacent faces. However, this ambiguity must be resolved so that samples referring to a feature vector at the same surface position and hierarchy level always refer to the same feature vector in memory. Otherwise, they are effectively treated as separate feature vectors leading to edge discontinuities.
- As stated above, for the implicit collision handling to work, hash collisions should not be repeated at different hierarchy levels. This means that if two feature vectors at positions p_1 and p_2 on one hierarchy level collide, feature vectors at the same positions on another level should not collide.
- To ensure efficient use of memory, the hash function should be surjective,

meaning each cell of the hash map is accessible.

- Since the hash function is evaluated frequently during training and evaluation, it needs to be efficiently computable.

Section 5.1.3 describes the hash function we use in detail.

Figure 4.4 shows an example of how feature hashing limits memory footprint.

4.4 Feature Quantization

Vaidyanathan et al. [2] suggest another measure to reduce the memory footprint of stored feature vectors. They quantize feature vector entries into N discrete bins. Thus only $\log_2(N)$ bits per feature vector entry are necessary to store encoding parameters. Without quantization, features entries are stored as 16-bit floating point numbers. A quantization into 16 bins would thus lead to a 75% decrease in memory footprint. We employ this method with the aim to optimize memory consumption and evaluate its impact.

Quantizing features this way leads to information loss and thus potentially negatively impacts visual quality. To reduce the negative impact on visual quality, we follow Vaidyanathan et al.'s approach to simulate quantization errors during training. After updating them in the backward pass, feature entries are clamped to the quantization range $[Q_{min}, Q_{max}]$. In the forward pass, uniform noise in the range of a bin's width $[-\frac{Q_{max}-Q_{min}}{2N}, \frac{Q_{max}-Q_{min}}{2N}]$ is added to feature vector entries to simulate quantization errors. After approximately 90% of training iterations, feature vector entries are fixed to the quantized value at the center of their respective bin. The remaining training iterations serve to optimize network parameters to quantized features. We perform quantization on the interval $[Q_{min}, Q_{max}] = [-1, 1]$.

Splitting a balanced interval $[Q_{min}, Q_{max}]$ (balanced meaning $Q_{min} = -Q_{max}$) into an even number of bins results in a bin border at 0. Thus, zero-valued parameters would be quantized to either the next highest or next lowest bin's center. To prevent this, Jacob et al. [22] suggest shifting the quantization range by half a bin width to make one bin center align with 0. Vaidyanathan et al. [2] achieve an improvement in visual quality this way. On the other hand, if the neural representation relies on non-zero feature entries with low absolute value, they might be falsely zeroed-out.

5

Implementation

To examine the viability of neural material compression, we implement and evaluate a prototype [23]. The prototype is based on *tiny-cuda-nn*s [14], a fast MLP implementation by Müller et al.. They achieve impressive performance through highly efficient use of the GPU, which they program using CUDA [24]. We extend their work by contributing an input encoding which implements a neural model for compression of material properties as described in chapter 4. Furthermore, we provide a framework to train and evaluate the model composed of the input encoding and a narrow MLP. The following sections describe the implementation of both parts in detail.

5.1 Input Encoding

We implement both the forward and backward pass of the input encoding efficiently using CUDA kernels.

5.1.1 Forward Pass

The forward pass reads a batch of unencoded surface positions represented by a $3 \times N$ matrix $\mathbf{X}$, where N is the number of instances in the batch. Unencoded input instances are stored as 3-dimensional column vectors $\mathbf{x} = (k, t_0, t_1)^T$ with triangle ID k and barycentric coordinates t_0 and t_1. t_2 is implicit, since for barycentric coordinates of points on the triangle enclosed by the vertices it always holds that

$$t_0 + t_1 + t_2 = 1 \tag{5.1}$$

The task of the forward pass is to encode $\mathbf{X}$ into a $L \cdot F \times N$ matrix $\mathbf{E}$ of encoded surface positions. Each column in $\mathbf{E}$ represents an encoded instance, comprised of L concatenated F-dimensional interpolated feature vectors. The structure of $\mathbf{E}$ determines the structure of the grid of threads the forward pass kernel is executed on. We employ a 2-dimensional thread grid that matches the dimensions of $\mathbf{E}$. Thus, each thread is responsible for writing one entry in $\mathbf{E}$. A thread at position (i, j) in the thread grid interpolates entry $f = j \mod F$ among the three feature vectors corresponding to the vertices of the subdivision triangle which $\mathbf{x} = \mathbf{X}_{:i}$ maps to at hierarchy level $l = \lfloor \frac{j}{F} \rfloor$.

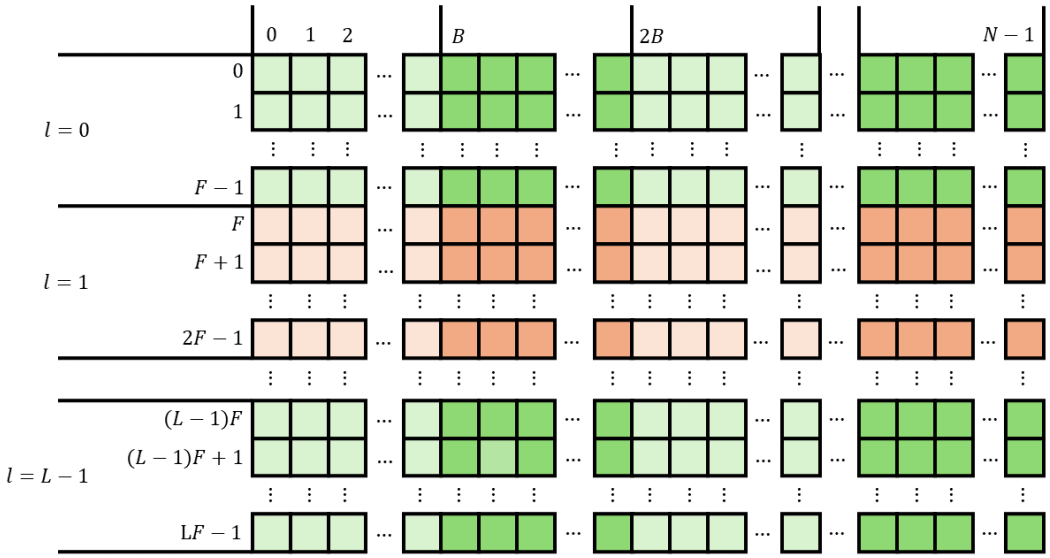


Figure 5.1: Thread Block Structure Schematic: A cell in the grid corresponds to a thread of the CUDA kernel. Each column represents an encoded input vector $\mathbf{e}$, which is composed by the concatenation of L interpolated feature vectors - one for each hierarchy level l. Cell blocks of a color compose a thread block. Threads are structured into blocks of dimensions $F \times B$. This way, each block only contains threads operating at the same hierarchy level, which improves cache coherency.

We structure the grid into thread blocks of size $F \times B$, where we choose B as a constant so that $F \cdot B = 512$. The block structure aligns so that each thread in a block operates on the same hierarchy level. Thus only a subset of feature vectors are ever accessed in each block, improving per-block cache coherency. Figure 5.1 shows a schematic of the kernel structure.

To encode $\mathbf{x}$, each thread follows a four step process. l denotes the hierarchy level the thread operates at:

1. Map $\mathbf{x}$ to the corresponding subdivision triangle at hierarchy level l and determine the positions of its vertices. These are the positions the relevant feature vectors are associated with.

2. If one or more of them lie on a vertex or an edge of the original mesh and are thus shared among two or more mesh triangles, resolve the feature vector position so that it is independent of the mesh triangle the feature vector is accessed from.

3. Hash the resulting feature vector positions to determine their locations in memory.

4. Interpolate the three feature vectors' fth component and write the result to $\mathbf{E}_{ji}$.

At hierarchy level l the mesh triangle is 2^{l-1}-Loop-subdivided (see section 4.2). Step

one becomes straight forward after the realization that the non-fractional parts of $2^l t_0$, $2^l t_1$ and $2^l t_2$ change exactly when an edge on the subdivided triangle is crossed. Thus, the subdivision triangle $\mathbf{x}$ maps to is identified by $W = (\lfloor 2^l t_0 \rfloor, \lfloor 2^l t_1 \rfloor, \lfloor 2^l t_2 \rfloor)$. The alignment of the subdivision triangle can be determined by $W_0 + W_1 + W_2 \mod 2$. What remains is to assemble the three feature vector positions at W's vertices by picking the corresponding components from $\frac{1}{2^l}(\lfloor 2^l t_0 \rfloor, \lfloor 2^l t_1 \rfloor, \lfloor 2^l t_2 \rfloor)$ and $\frac{1}{2^l}(\lceil 2^l t_0 \rceil, \lceil 2^l t_1 \rceil, \lceil 2^l t_2 \rceil)$ based on the subdivision triangle's alignment.

We explain steps two and three, resolving of edge ambiguities and hashing of feature positions, in the next section 5.1.3. After hashing feature vector positions, their memory locations are known and can be read from and written to.

Finally, in step four, the relevant feature vectors $\mathbf{f}^{(0)}$, $\mathbf{f}^{(1)}$ and $\mathbf{f}^{(2)}$ are sampled at the memory locations provided by the hash function, interpolate their fth entry based on $\mathbf{x}$'s position on the subdivision triangle, and write the result to $\mathbf{E}_{ji}$:

$$\mathbf{E}_{ji} = \{2^l t_0\} \cdot \mathbf{f}_f^{(0)} + \{2^l t_1\} \cdot \mathbf{f}_f^{(1)} + \{2^l t_2\} \cdot \mathbf{f}_f^{(2)} \tag{5.2}$$

Or in case the subdivision triangle is non-edge-aligned:

$$\mathbf{E}_{ji} = (1 - \{2^l t_0\}) \cdot \mathbf{f}_f^{(0)} + (1 - \{2^l t_1\}) \cdot \mathbf{f}_f^{(1)} + (1 - \{2^l t_2\}) \cdot \mathbf{f}_f^{(2)} \tag{5.3}$$

where $\{x\} = x - \lfloor x \rfloor$.

5.1.2 Backward Pass

The kernel for the backward pass has a structure identical to that of the forward pass. Its task is to propagate gradients from the MLP input layer to the feature vectors and update them accordingly.

Each thread (i, j) reads one entry $\nabla\mathbf{E}_{ji}$ from the encoded input gradient matrix $\nabla\mathbf{E}$, which corresponds to the MLP's input layer gradients. Determining the relevant feature vectors based on $\mathbf{x} = \mathbf{X}_{:i}$ works identically to steps one through three in section 5.1.1.

To propagate input layer gradients to the relevant feature vectors, each thread evaluates the derivative of the interpolation function 5.2/5.3 w.r.t the three relevant feature vectors $\mathbf{f}^{(0)}$, $\mathbf{f}^{(1)}$ and $\mathbf{f}^{(2)}$:

$$\frac{\partial L}{\partial \mathbf{f}^{(k)}} = \frac{\partial \mathbf{E}_{ji}}{\partial \mathbf{f}^{(k)}} \frac{\partial L}{\partial \mathbf{E}_{ji}} \tag{5.4}$$

$$= \frac{\partial \mathbf{E}_{ji}}{\partial \mathbf{f}^{(k)}} \nabla\mathbf{E}_{ji} \tag{5.5}$$

$$= \{2^l t_k\} \nabla\mathbf{E}_{ji} \tag{5.6}$$

Or in case the subdivision triangle is non-edge-aligned:

$$\frac{\partial L}{\partial \mathbf{f}^{(k)}} = (1 - \{2^l t_k\}) \nabla\mathbf{E}_{ji} \tag{5.7}$$

with $k = 0, 1, 2$. Computing the partial derivative of the input encoding with respect to some feature vector $\mathbf{f}^{(k)}$ is quite trivial as it is simply the corresponding linear interpolation weight $\{2^l t_k\}$ from the forward pass. Feature vectors are updated by subtracting the respective gradient. Updates to features are carried out as atomic operations since multiple concurrent threads might operate on the same feature vector.

5.1.3 Feature Hashing

Both the forward and the backward pass rely on a mapping from a surface position $(k, \mathbf{t})$ to the memory location of a feature vector. This mapping is realized by a hash function.

Feature vectors whose positions lie on vertices or edges may be validly referred to by different surface positions (see figure 5.2). However, all valid references to a feature vector's position have to be hashed to the same memory location. If this was not the case, they would effectively be treated as *different* feature vectors, which would lead to discontinuities at mesh edges. To prevent this, we pre-compute a data structure that holds the IDs of adjacent faces at each vertex/edge. Before hashing a feature vector position to retrieve its memory location, it is mapped to the equivalent position on the face with the lowest ID k that shares this feature vector. This way the ambiguity is resolved.

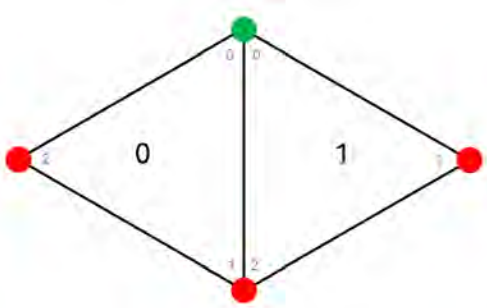


Figure 5.2: Feature Vector Naming Ambiguity: The feature vector associated with the highlighted vertex (green) may be referred to by $(0, (1, 0, 0))$ or $(1, (1, 0, 0))$.

After resolving feature vector naming ambiguities it remains to hash the resulting feature vector position to a memory location. Recall that at each hierarchy level, the hash map stores at most T distinct feature vectors. Thus the hash function should map a surface position, annotated with a hierarchy level, (k, t_0, t_1, l) to an index i, $0 \leq i < T$ into the hash map. As described in section 5.1.1, t_2 is implicit. To simplify the definition of the hash function, barycentric weights t_0 and t_1 are mapped to integer values $W_0 = 2^l t_0$ and $W_1 = 2^l t_1$. Since feature vectors are equidistantly distributed, W_0 and W_1 are ensured to have no fractional component. We define the hash function $h : \mathbb{Z}^4 \to \mathbb{Z}$ as:

$$h(k, W_0, W_1, l) = (\pi_0 k \oplus \pi_1 W_0 \oplus \pi_2 W_1 \oplus \pi_3 l) \mod T \tag{5.8}$$

This is analogous to the spatial hash function proposed by Teschner et al. [25] and used by Müller et al. [1] for their multiresolution hashgrid encoding. $\pi_{\{0,\dots,3\}}$

are large prime numbers and $\oplus$ denotes the bitwise *xor*-operation. Although, as opposed to the parameters of Müller et al.'s [1] spatial hash function, k and l are not strictly spatial integer coordinates, h fulfills all requirements described in section 4.3. Multiplications and *xor*-operations are very fast to compute, it covers the entire range of the hash map and hashes feature vectors dependent on their hierarchy level l. Thus it is not prone to repeating hash collisions across several hierarchy levels. Müller et al. [1] improve cache coherency by choosing one $\pi_i = 1$. We follow this approach and choose $\pi_0 = 2\,654\,435\,761$, $\pi_1 = 1$, $\pi_2 = 805\,459\,861$ and $\pi_3 = 5\,389\,456\,063$.

5.2 Training and Validation

The final model consists of the input encoding and an MLP of four hidden layers with 64 neurons each, using a ReLU activation function. MLPs of this size perform efficiently on our hardware with Müller et al.'s [14] *fully fused* implementation. We train the model on a ground truth dataset of a 3D mesh and its surface material properties. To evaluate the neural representation after training convergence, we validate it on a data set of mesh surface positions that map to the pixels of a projected image of the object. A forward pass of such a dataset through the model yields an image of predicted material parameters, which we compare to the ground truth to evaluate the models performance.

5.2.1 Generation of Training Data

Training data consists of pairs $(\mathbf{x}, \mathbf{y})$ of surface positions $\mathbf{x}$ and corresponding material parameters $\mathbf{y}$. We generate training data in batches on-the-fly during training using a CUDA kernel.

First, a position $\mathbf{x} = (k, \mathbf{t})$, with mesh triangle k and barycentric coordinates $\mathbf{t}$, is sampled uniformly from the mesh's surface. Triangle k is drawn from a discrete distribution with each triangle weighted by its surface area. This requires a precomputed look-up table of the distribution's cumulative density function. Sampling triangle k accordingly means traversing the CDF via binary search, which introduces a runtime complexity of $O(\log n_{\text{faces}})$ and a significant amount of control flow to the CUDA kernel. In practice, this proves to cause a meaningful increase in training time. Ideally, training data generation should occur in constant time with minimal control flow. It turns out that for many meshes - especially those with faces of approximately consistent size - sampling surface triangles from a uniform distribution does not cause a significant decrease in representation quality, while allowing for greatly improved training times. Barycentric coordinates $\mathbf{t} = (t_0, t_1, t_2)$ on triangle k can be sampled uniformly as follows:

$$t_0 = 1 - \sqrt{r_0} \tag{5.9}$$

$$t_1 = \sqrt{r_0}(1 - r_1) \tag{5.10}$$

$$t_2 = \sqrt{r_0} r_1 \tag{5.11}$$

with independent random variables r_0 and r_1 uniformly distributed in $[0, 1]$ [26].

Labeling training instances with a desired material property output $\mathbf{y}$ requires ground truth data. In real use cases this data might be provided in the form of a very high-resolution version of the mesh containing material properties as vertex data (e.g. obtained via photogrammetry) or a 3D texture, sampled at the mesh surface. For the purpose of this prototype, we use *uv*-mapped meshes and material data defined by texture maps as ground truth training data due to their abundant availability. We interpolate *uv*-coordinates of adjacent vertices and sample the texture map(s) of material properties at the corresponding position to determine $\mathbf{y}$. This does not limit the conclusions about compression quality that can be drawn from the prototype's evaluation. However, in a real world use case this would defeat one of the main advantages of the method, which is not having to provide a *uv*-mapping for the mesh at all.

5.2.2 Training and Validation

We train the model on on-the-fly generated batches of training data based on a given mesh and ground truth material properties. Batches consist of 2^{18} instances. We use an Adam [27] optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.99$, a learning rate of 10^{-2}, and a mean squared error loss function (MSE). There are loss functions that match the human perception of visual artifacts better, e.g. ꟻLIP [28] or SSIM [29]. However, these tend to be pixel neighborhood-dependent, which is incompatible with the on-the-fly generation of random training samples we use. Learnable parameters are stored as half-precision floats and are initialized by uniformly drawing from the interval $[-10^{-4}, 10^{-4}]$. We run 5000 training iterations with one batch each, which is approximately as long as it takes the model to converge plus some margin. If feature quantization is enabled, we run 500 additional iterations after freezing the encoding parameters to optimize network weights.

After convergence, we let the model predict a validation set. The validation set consists of a batch of surface positions that correspond to those that pixels would map to in a projected view of the object. We generate validation sets at a resolution of 1280×720 pixels. Using the model to predict surface properties for validation sets yields a set of output images that correspond to the projected view. The number of prediction images depends on the number of material channels encoded. We run benchmarks with six encoded channels, namely albedo (r, g, b) and surface normals (n_x, n_y, n_z). Thus, per view, two prediction images are generated. We compare these to ground truth images of the same view to evaluate the models representation quality. Chapter 6 describes the setup and results of the evaluation in detail.

6

Evaluation

We evaluate the method using five textured meshes that expose the model to different properties. We train the model to fit the meshes' surface properties and evaluate the visual quality of the neural representation, the time it takes to train and query as well as its memory footprint. Section 6.1 introduces the data sets we use. We examine the impact of the model's hyperparameters in section 6.2. Finally, we present the results of our method's evaluation in section 6.3.

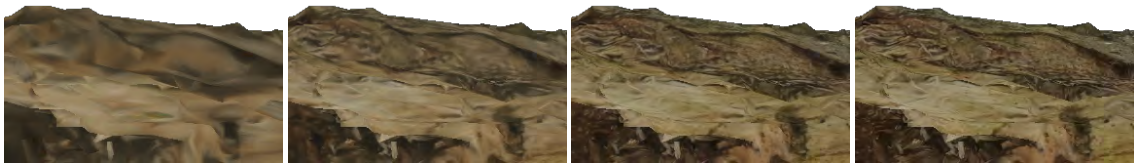
Evaluation takes place on an Intel i9-9980XE CPU and an NVIDIA RTX 2080 Ti GPU.

6.1 Data sets

BarramundiFish is one of the GlTF-Sample-Models [4]. *Apple*, *Stick* and *TreeStump* [30] are photogrammetry models with high vertex density. *StickLow* is a low-fidelity version of *Stick* with approximately 20% of the original triangle count. Since the distribution of feature vectors is highly dependent on tessellation, we include *StickLow* to examine how applying the method to meshes of different tessellation fidelity affects performance and representation quality. All models come with texture maps for surface albedo (r, g, b) and surface normals (n_x, n_y, n_z), which we use as ground truth data to train and evaluate the model. Table 6.1 lists the data sets with their respective triangle count and texture resolution.

Name	Triangle Count	Reference Texture Resolution
BarramundiFish	3 864	$2\,048 \times 2\,048$
Apple	50 000	$4\,096 \times 4\,096$
Stick	49 999	$4\,096 \times 4\,096$
TreeStump	499 239	$4\,096 \times 4\,096$
StickLow	9 999	$4\,096 \times 4\,096$

Table 6.1: Mesh data sets used for evaluation with the respective triangle count and reference texture resolution.



$L = 2$ $L = 4$ $L = 6$ Reference

Figure 6.1: Surface albedo of *StickLow* as represented by models configured with different hierarchy level counts L. Low values for L lead to significant overblurring artifacts, as feature vectors are interpolated on a scale too coarse to capture all texture detail.

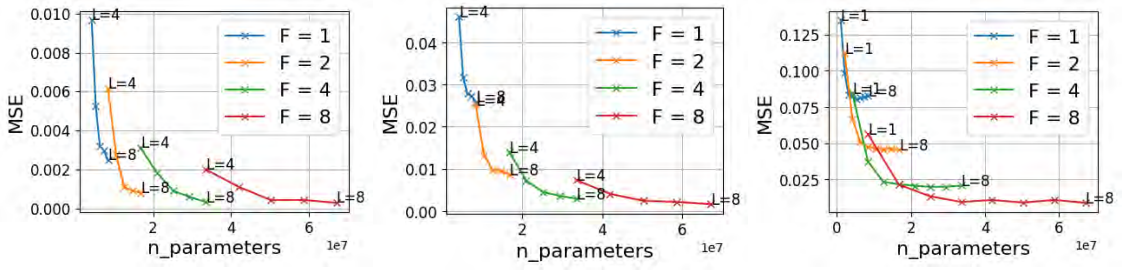
6.2 Hyperparameter Tuning

Poorly configured hyperparameters may cause insufficient quality of the neural surface representation. The optimal hyperparameter configuration, however, may greatly depend on the characteristics of the mesh and its surface texture. For example, since the coarsest hierarchy level links feature vectors directly with mesh vertices, its resolution directly depends on mesh tessellation. Thus, the number of hierarchy levels L necessary to represent the finest level of detail may differ based on the mesh's vertex density. Section 4.3 describes how collisions in the hash map lead to averaging of feature vectors. It turns out that this has a much more visible impact on high contrast textures. Thus, such data sets might require larger hash map sizes T for optimal representation quality. This section highlights how each of the hyperparameters impacts representation quality.

6.2.1 Number of Hierarchy Levels L and Feature Vector Length F

We start by evaluating the impact of the number of hierarchy levels L and the number of feature vector entries F. We train the model on each of the data sets using every combination of values for hyperparameters $L \in \{1, \dots, 8\}$ and $F \in \{1, 2, 4, 8\}$. We log representation quality after convergence measured by mean squared error (MSE) and its memory footprint measured by the number of learnable floating point encoding parameters.

Increasing the total number of learnable encoding parameters allows the model to store more information and thus increases representation quality. However, the hyperparameters that influence the number of total distinct learnable parameters (F, L and T) are not created equal. Only increasing F and keeping L low, for example, cannot benefit representation quality without bound. The highest representable detail frequency is governed by the spacing of feature vectors on the finest hierarchy level. If the finest hierarchy level is too coarse to represent full surface detail, interpolation artifacts, such as overblurring, remain visible. Figure 6.1 shows the extent of these artifacts across hierarchy level counts L for one of the data sets. In



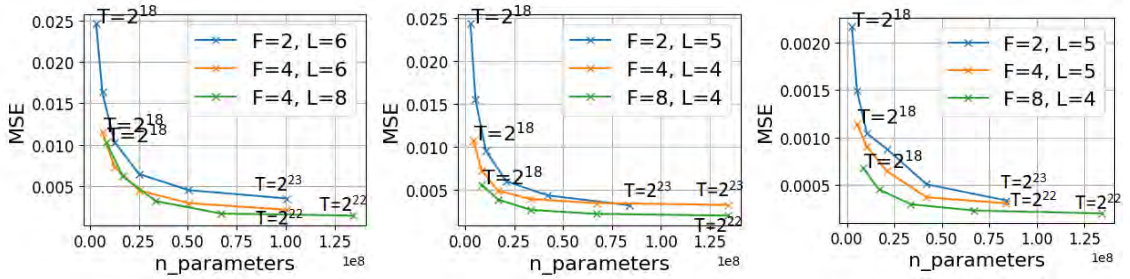
BarramundiFish *StickLow* *TreeStump*

Figure 6.2: Plots showing total number of learnable encoding parameters (x-axis, in units of 10^7) against MSE of the neural representation after convergence (y-axis). Each data point represents memory footprint and representation quality for one model configuration. Data points on the same colored curve share the same feature vector dimensionality F, but differ in number of hierarchy levels $L \in \{1, \ldots, 8\}$ as labeled. Hashmap size T is not varied and set to a constant 2^{20}. The data sets depicted increase in vertex density from left to right (see table 6.1).

this case, L should be increased to add finer hierarchy levels. Conversely, the effect of additional hierarchy levels on high-vertex density meshes is low, since the finest necessary feature vector resolution might be reached with few hierarchy levels.

We want to remark at this point that this overblurring behaviour may turn out useful as a filtering method. In texture filtering, the aim is to eliminate high-frequency signals in the texture that would lead to artifacts if not sampled at a high enough rate. Thus, only sampling the neural representation at coarse levels might yield a blurred, low-frequency version of the objects surface that could turn out useful for filtering purposes. Testing this hypothesis, however, is outside the scope of this work.

Figure 6.2 plots the performance and memory footprint of the model depending on hyperparameters F and L with constant $T = 2^{20}$. Ideally, the model yields a low representation error while keeping a small memory footprint. Thus, desirable configurations lie close to the bottom-left corner of the plot. Data sets with low vertex density (*BarramundiFish* and *StickLow*) show a significant improvement in representation quality with every added hierarchy level. Only at approximately $L = 8$ does the impact diminish. Furthermore, their curves overlap strongly in representation quality, which indicates a comparatively low impact of higher feature vector dimensionalty F. Data sets with high vertex density (e.g. *TreeStump*) show a different behaviour. Curves flatten out already around $L = 4$, which is to be expected as the coarsest hierarchy level is already quite fine. Thus only few additional levels are necessary to match the highest frequency of detail present in the training data. However, a strong impact of feature vector dimensionality F can be observed. We attribute this to the lack of coarser levels. Low-frequency detail needs to be stored within features of fine hierarchy levels. Larger feature vectors allow for the storage of more data per hierarchy level and thus potentially improve representation of low-frequency detail.



StickLow *Stick* *Apple*

Figure 6.3: Plots showing total number of learnable encoding parameters (x-axis, in units of 10^8) against MSE of the neural representation after convergence (y-axis). Each data point represents memory footprint and representation quality for one model configuration. Each colored line contains data points that share values for F and L. Data points along each colored line differ in their value for $T \in \{2^{18}, \ldots, 2^{24}\}$ as labeled.

Furthermore, we observe configurations with $F = 1$ perform significantly slower. This is because processing data in vectors can benefit from the GPU's single instruction-multiple data (SIMD) capabilities. Thus we do not use configurations with one-dimensional feature vectors for evaluation.

6.2.2 Hash Map Size T

To examine the impact of the per-level hash map size T, we train models again using three configurations of L and T with varying hash map sizes $T = 2^e$ with $e \in \{18, .., 24\}$. We evaluate representation quality and memory footprint as before.

Figure 6.3 plots the performance and memory footprint of the model depending on T. As before, desirable configurations appear close to the bottom left corner of the plot. It is clearly visible that increasing hash map sizes yields significant quality improvements up to an inflection point beyond which returns diminish. This holds for all examined datasets and configurations.

We observe that configurations with high values for F and L generally yield better representation quality, even when applying more aggressive hashing to reduce their memory footprint to be comparable with configurations lower values for F and L. This observation is consistent across all datasets. We infer that it is generally preferable to choose generous values for hyperparameters F and L and achieve efficient memory use through smaller hash map sizes T.

Small hash map sizes lead to more frequent collisions. The implicit collision handling strategy averages updates to feature vectors that share a location in the hash map. More frequent hash collisions thus lead to more aggressive averaging across updates to otherwise unrelated feature vectors. This leads to a loss of saturation in the neural representation (see figure 6.4). Decreasing hash map sizes even further leads to strong noise artifacts. This is interesting, as Müller et al. [1] primarily observe overblurring.

$T = 2^{18}$ $T = 2^{20}$ $T = 2^{22}$ Reference

Figure 6.4: Surface albedo of *TreeStump* as represented by models configured with different hash map sizes T. Low values for T lead to more aggressive averaging among updates to unrelated feature vectors. This leads to global averaging of albedo colors in the model's representation, decreasing saturation in images rendered from models with low hash map sizes. Note how the rich green color of the moss, clearly visible in the reference, appears less pronounced the lower the value we choose for T.

How severely this behaviour affects visual quality depends on the characteristics of the surface texture. Besides that, the mesh's vertex count also has an influence, as it is directly linked to the number of feature vectors. If a larger number of feature vectors is mapped to a hash map of the same size, more collisions occur.

6.3 Results

This section presents the results of the evaluation of our method. Based on how the different hyperparameters impact the model's performance, as described in the previous section, we pick three hyperparameter configurations per dataset to carry out evaluation. For each dataset, we pick a *quality-optimized* configuration, which aims to test how well the model can represent the dataset given a large amount of memory. In other words, its aim is to find those properties that the model struggles to represent even if memory is not a limiting factor. On the other end of the spectrum, we pick a *memory-optimized* configuration with the goal of representing a dataset at satisfying quality with a minimal memory footprint. In this case, we allow for slight artifacts that we deem not to impact overall visual quality severely. As a middle ground, we also pick a *balanced* configuration that aims to find a compromise between the other two. Table 6.2 lists the hyperparameter values for each dataset and configuration.

First, we examine the visual quality of the neural representation in section 6.3.1. Section 6.3.2 focuses on the memory footprint the model's parameters impose. Finally, we evaluate the model's training and query times in section 6.3.3.

6.3.1 Visual Quality

We evaluate visual quality qualitatively and quantitatively. MSE serves as a good metric to measure how well the model fits the data it has been trained on. However, it has shown to be unsuitable to estimate how well a generated image resembles a reference according to a viewer's perception. Human judgement of image artifacts

Data Set	*BarramundiFish*			*Apple*			*Stick*		
Configuration	m	b	q	m	b	q	m	b	q
L	8	8	8	4	4	4	4	4	4
F	2	2	2	8	8	8	8	8	8
T	19	20	21	17	19	21	18	20	21
$n_{\text{params}} \times 10^{-7}$	0.8	1.7	3.4	0.4	1.7	6.7	0.8	3.4	6.7
MSE $\times 10^2$	0.13	0.10	0.06	0.09	0.06	0.05	0.56	0.26	0.22
ꟻLIP (RGB)	0.049	0.060	0.065	0.029	0.017	0.015	0.051	0.042	0.034
ꟻLIP (Normal)	0.058	0.092	0.055	0.033	0.021	0.020	0.090	0.094	0.054

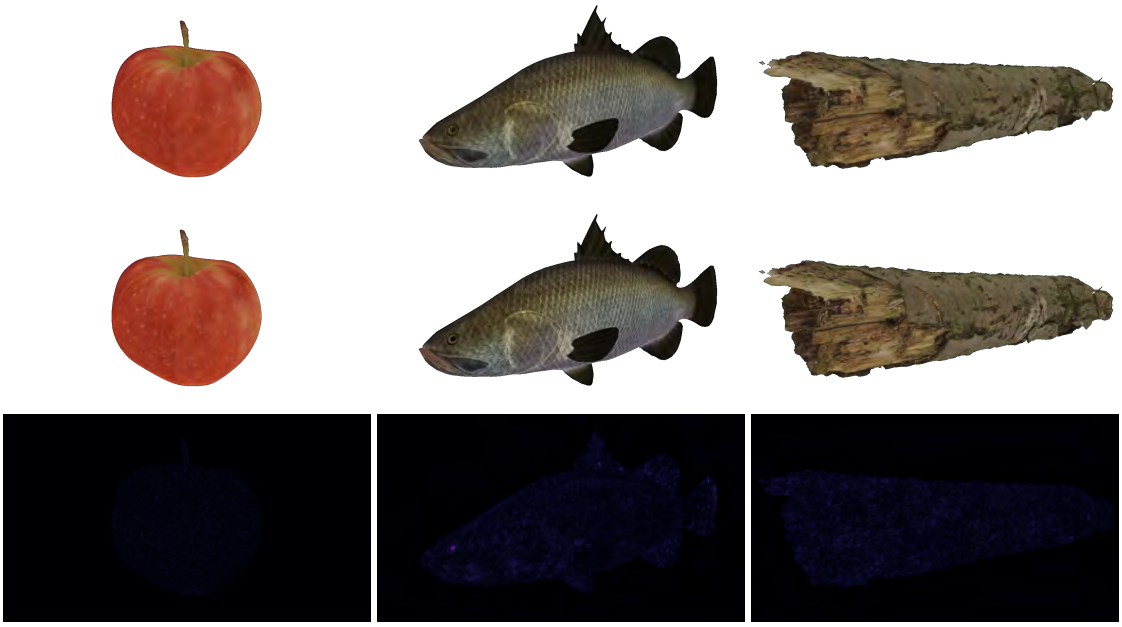
Data Set	*TreeStump*			*StickLow*		
Configuration	m	b	q	m	b	q
L	4	4	4	6	8	8
F	8	8	8	4	4	4
T	19	20	21	20	20	21
$n_{\text{params}} \times 10^{-7}$	1.7	3.4	6.7	2.5	3.4	6.7
MSE $\times 10^2$	1.63	0.94	0.58	0.46	0.30	0.18
ꟻLIP (RGB)	0.115	0.079	0.057	0.050	0.046	0.039
ꟻLIP (Normal)	0.106	0.087	0.072	0.075	0.059	0.054

Table 6.2: Hyperparameter values and aggregated quantitative measurements for each dataset and configuration after convergence. Configurations include *memory-optimized* (m), *balanced* (b) and *quality-optimized* (q). Note that ꟻLIP values are projection-dependent and thus not comparable across datasets.

not only depends on per-pixel color differences, but also on preservation of larger structures in the image. Furthermore, the noticeability of artifacts to the human eye also depends on brightness and contrast in the region.

For this reason, we extend our set of evaluation metrics by ꟻLIP [28]. ꟻLIP is an image comparison method by Andersson et al. [28]. It aims to highlight regions in the image where a viewer would notice artifacts if they flipped between the test image and the reference, taking into account surrounding features like brightness and contrast. ꟻLIP provides a heatmap, but can also aggregate results down to a single value by computing the weighted median of the histogram of per-pixel values. The disadvantage of image-based quality evaluation methods, like ꟻLIP, however, is that they inevitably require the projection of model samples to an image. Thus, they are generally not suitable to serve as a means of comparison across projections or datasets. In our case, *Stick* and *StickLow* are an exception to this rule as they represent the same object with different fidelity and the images we use to evaluate them use the same projection. Furthermore, the number of per-pixel channels in the image domain is limited to three. Thus, evaluating all represented material properties may require multiple images. In our case, where we represent three color and three normal channels, two images are necessary for a full comparison.

Table 6.2 shows MSE and ꟻLIP metrics for each dataset and configuration. Figure 6.5 compares images generated by sampling the albedo (r, g, b) channels from each datasets neural representation, trained using *memory-optimized* hyperparame-



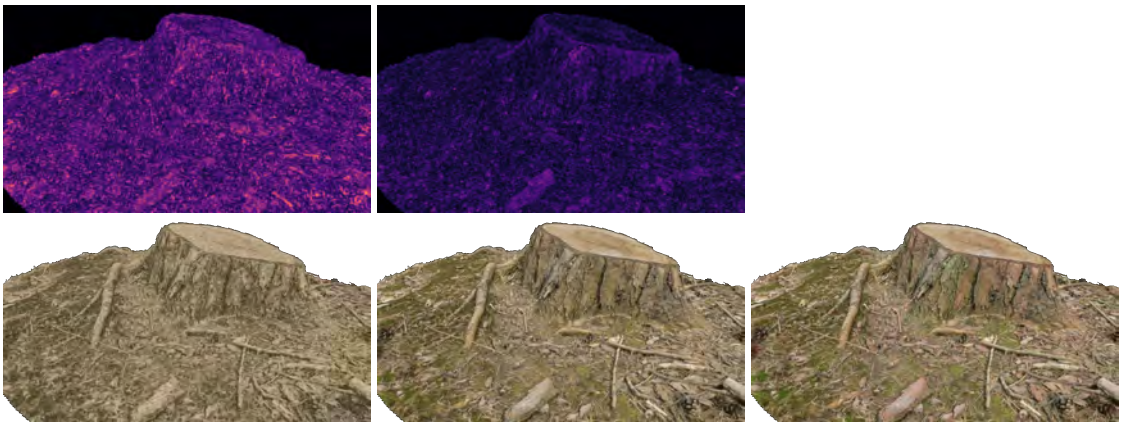
Apple *BarramundiFish* *Stick*

Figure 6.5: Visual comparison between neural representation (top) and reference (middle). ꟻLIP heatmap (bottom) to highlight differences. Albedo channels are sampled from the *memory-optimizes* neural representations for each dataset and projected to an image. At these settings *Apple*, *BarramundiFish* and *Stick* achieve compression rates of approximately 8.3%, 66.6% and 16.6% respectively.

ter configurations.

Our method performs best on the *Apple* dataset. Even the *memory-optimized* configuration allows for a representation of the texture with barely any noticeable artifacts, while achieving impressive compression rates. With the *balanced* and *quality-optimized* configurations, we find the images rendered from the neural representation indistinguishable from the reference. This is also reflected by the MSE and ꟻLIP metrics. *Stick* achieves similar compression rates but is slightly more prone to minor blurring artifacts and desaturation at low hash map sizes.

BarramundiFish is representable at satisfying quality. However, on closer inspection of the areas highlighted by ꟻLIP, slight artifacts in the representation of high-frequency features become visible - especially on the fins and around the eye. The *quality-optimized* configuration suffers less from these. Interestingly, it performs worse in overall color representation, showing some regions of the surface slightly too bright. We attribute this behaviour to characteristics of the MSE loss function. MSE potentially punishes large color deviations on few pixels (i.e. high-frequency artifacts) more severely than a slightly misrepresented color across a more widespread area. This highlights how MSE is only of limited use as an estimator for visual quality and how our method might benefit from a loss function that more closely matches human perception of visual quality. Note how MSE values for *BarramundiFish* do not correlate with ꟻLIP values (see table 6.2).



$N = 16$ $N = 256$ Reference

Figure 6.6: Comparison of albedo images (bottom) of *TreeStump* using different quantization bin counts N. The top row shows ꟻLIP heatmaps to visualize the difference to the reference.

Appendix A shows images of all datasets rendered using *memory-optimized*, *balanced* and *quality-optimized* configurations.

Feature quantization discretizes features into a set of bins and thus leads to a loss of information captured in the neural representation. Vaidyanathan et al. [2] observe only a slight decrease in visual quality as a result of this. We evaluate the impact of feature quantization by comparing the visual quality of quantized models trained using simulated quantization errors (see section 4.4) against that of non-quantized models.

We observe severe artifacts and noise for all datasets when quantizing features to $N = 16$ bins, i.e. 4 bits per feature vector entry. $N = 256$, i.e. quantizing to 8 bits per feature vector entry, shows significant improvements but still suffers from visible artifacts. Figure 6.6 shows a quantized representation of *TreeStump*. In contrast to what Vaidyanathan et al. [2] find, we do not observe an increase in quality by shifting the quantization interval to align the center of a bin with zero. Instead, we find this to significantly worsen representation quality, although this effect decreases with higher bin counts N. We hypothesize that this is due to all features within the zero-aligned bin interval being mapped to zero and thus their influence effectively being ignored completely. If a model is reliant on many features with low enough absolute value to land in the zero-aligned bin interval, this may have a significant impact on the model's representation quality.

6.3.2 Memory Footprint

We evaluate the model's memory footprint by the memory necessary to store the MLP's and the encoding's learnable parameters. All learnable parameters are stored as 16-bit half-precision floats. We use an MLP with 4 hidden layers of 64 nodes each in every configuration, arriving at a network parameter count of 16 384. The majority, however, are the encoding's parameters, given by the product of the number of

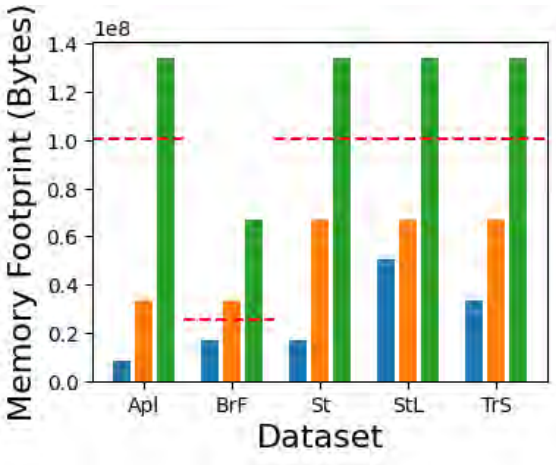


Figure 6.7: Memory footprint of learnable parameters by dataset (*Apple*, *BarramundiFish*, *Stick*, *StickLow*, *TreeStump*) and configuration: (*memory-optimized* (blue), *balanced* (orange), *quality-optimized* (green)). The red dotted line indicates the total memory footprint of uncompressed reference texture maps for comparison.

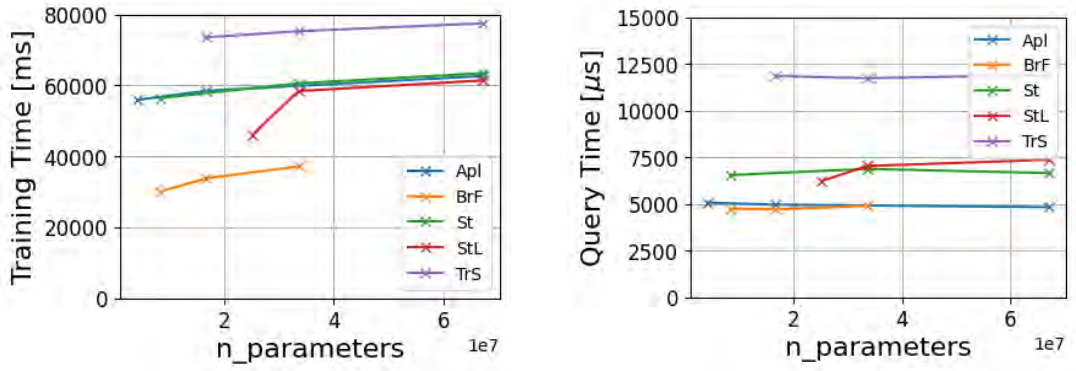
hierarchy levels L, the number of distinct feature vectors per level T and the number of entries per feature vector F.

Figure 6.7 plots the model's memory footprint by dataset and configuration. *Apple* compresses best. The visual quality of its representations hardly suffers from any artifacts even at small hash map sizes of $T = 2^{17}$, which we use for its *memory-optimized* configuration. With *Apple* we achieve a compression rate of approximately 8.3% at nearly indistinguishable visual quality.

TreeStump and *Stick* are more prone to artifacts arising through reduced hash map sizes. In *TreeStump*'s case it is a decrease in color saturation, while *Stick* starts showing blurring artifacts. Thus, we can only achieve a compression rate of approximately 33.3% and 16.6% respectively before visual quality degrades beyond an acceptable amount.

Interestingly, datasets with low vertex density (*BarramundiFish* and *StickLow*) perform worst. With *BarramundiFish* and *StickLow* we achieve a compression rate of approximately 50.0% and 66.6% respectively, before encountering noticeable artifacts. As explained in the previous section, we attribute *BarramundiFish*'s poor performance in part to the choice of MSE as a loss function. However, the tendency of low vertex density datasets performing worse suggests that that might not be the only reason. The feature hierarchy on meshes with low vertex density starts at a coarser level and thus requires more of them to fit the highest frequency detail in the texture. Therefore, it could be hypothesized that features on coarse hierarchy levels carry less information. Investigating this hypothesis, however, is outside the scope of this work.

Feature quantization enables a reduction of each feature's memory consumption to only four bits (with $N = 16$) at the cost of some visual quality. We find that quantization to $N = 256$ bins, i.e. 8 bits per feature vector entry, in some cases still



Training Time Query Time

Figure 6.8: Plots of training and query time per data set and hyperparameter configuration. Data points on a shared colored line represent different hyperparameter configurations for the same data set. Configurations from left to right (per line): *memory-optimized*, *balanced*, *quality-optimized*. The resolution of the query framebuffer is 1280×720.

allows for satisfying visual quality without severe artifacts. In these cases, feature quantization may be a suitable measure to decrease the model's memory footprint, as it leads to a reduction of 50%. However, we find that halving hash map sizes T tends to result in significantly better visual quality than feature quantization while yielding the same memory advantage. For this reason, we discourage the use of feature quantization.

6.3.3 Performance

We evaluate the model's training and query times. Since training happens offline, good training times are desirable to favor artist workflow, but are not critical to the real time performance of the system. Queries to the model, i.e. sampling the object's surface, are expected to happen every frame in real time applications. Therefore, it is critical that query times are low.

Figure 6.8 plots training and query times across all data sets and hyperparameter configurations. Training times range from approximately 30 to 80 seconds. We observe significant impact of F and L on training performance as these parameters have direct influence on the number of threads launched during forward and backward passes of the encoding. Additionally, F and L contribute to the total parameter count $T \cdot L \cdot F$. A large number of learnable parameters causes poorer cache coherence, which further slows down performance. Thus, T's impact on the model's performance is smaller than that of F and L. The plot shows this behaviour clearly: *TreeStump*'s configurations only vary in T and thus show minimal difference in training times, while the *memory-optimized* configuration used with *StickLow* uses less hierarchy levels and thus outperforms the training times of *StickLow*'s *balanced* configuration by a significantly bigger margin.

Query times range from approximately 5 to 12 milliseconds. Taking into account

that one query is able to process texture samples for an entire framebuffer (or multiple) in parallel, these times allow for real-time applications. We achieve these by leveraging Müller et al.'s [10] *fully fused* MLP architecture. Query times exhibit similar behaviour in relation to hyperparameters F, L and T as training times. However, since only one forward pass is necessary, the impact they have is smaller and, in the case of T, negligible.

7

Conclusion

We contribute a novel method to compress material data defined on 3D mesh surfaces. Our neural approach employs a hierarchy of learnable parameters associated with surface positions defined via mesh triangle subdivision. We parametrize surface positions using a triangle identifier and barycentric coordinates and map them directly to feature vectors. This way we completely circumvent the need for *uv*-mapping. We implement our method as an input encoding to a small MLP, and train the composite model on a dataset of pairs of randomly sampled surface positions and corresponding material properties. After convergence, texture information is stored in network weights and feature vectors, the latter of which make up the vast majority of the model's parameters. By hashing feature vectors on a per-hierarchy level basis, similar to MHE by Müller et al. [1], to a hashmap of constant size, we achieve good compression rates in the range of approximately 8% to 33% for high vertex density meshes. The implicit collision handling strategy shows good preservation of visual quality. Furthermore, we examine the viability of quantizing features [22] [2] down to a few bits and adapting the training process to simulate quantization errors. However, our results show that the visual artifacts introduced by this measure are severe. In all examined cases, we achieve the same reduction in memory footprint with better preservation of visual quality through reducing hash map sizes.

Our prototype [23] makes use of Müller et al.'s [10] *fully fused* MLP architecture. This enables us to achieve real-time capable query rates and keeps training times in the order of one minute.

7.1 Limitations and Future Work

Despite the promising results, our method still shows some limitations.

We evaluate the approach using datasets of meshes with surface albedo channels (r, g, b) and surface normals (n_x, n_y, n_z) to demonstrate how these can be unified into a single neural representation to make use of correlations among them. Modern renderers often use significantly more channels, among which are specular reflectivity, surface roughness, metallicness or pre-baked lighting data like ambient occlusion. Many of these may correlate with each other, allowing for a more efficient memory usage when combined into one neural representation that can make use of these correlations.

Our method links feature vectors on the coarsest hierarchy level directly with mesh vertices. Our results show that meshes with low vertex density, i.e. those that need more hierarchy levels to capture the the highest-frequency texture detail, require more memory for parameters to preserve comparable visual quality. This observation hints at less efficient memory usage with features on coarse hierarchy levels. Future work might examine how the method could benefit from decoupling the coarsest hierarchy level's resolution from mesh tessellation. Another approach could be to adapt the number F of entries per feature vector depending on the hierarchy level to more closely match the amount of information stored in its features.

Furthermore, our method subdivides each mesh triangle into a hierarchy of the same depth. Especially with meshes that contain triangles of varying size or surface representation of varying detail, a more adaptive approach might be able to distribute features more efficiently. A solution could be to construct an adaptive hierarchy with different depths per triangle based on a pre-computed analysis of surface detail frequencies on each triangle. However, challenges like edge discontinuities might need to be solved.

We observe overblurring when representing surface detail with too coarse hierarchy levels. While this is undesirable behaviour under the goal to match the reference as closely as possible, it might turn out useful when trying to find a neural counterpart to low-pass texture filtering. Future work could examine how querying the model only at coarser hierarchy levels may serve as a low-pass filtering method. This may require adapting the training process by adding instances that reflect this behaviour.

The real-time viable performance of our method is largely due to Müller at al.'s [10] *fully fused* MLP architecture being able to process an entire framebuffer of queries in parallel at fast rates. However, all queries must be to the same model. This limits the viability of the method in its current state for scenes comprised of meshes with different neural surface representations. Query times would add up and scale with the number of models, quickly diminishing performance beyond real-time viability. Future work that aims at incorporating our approach into a real-time renderer may need to solve this problem. Specialized hardware to process queries to multiple *fully fused* networks in parallel might pose a solution.

Our results confirm that, in some cases, MSE as a loss function does not serve well as an estimator for visual representation quality (see our results for *BarramundiFish*, section 6.3.1). Other functions, like ꟻLIP [28] or SSIM [29], approximate visual quality significantly better. However, these are often neighborhood-dependent, i.e. the value of a sample depends on adjacent samples. This is not compatible with our on-the-fly method of generating training data, as instances are randomly sampled in parallel and independently of each other. Thus, further research is necessary to adapt the method to work with better loss functions.

7.2 Ethical Considerations

The ethical considerations that need to be taken with research on compression of material data are minimal. Material data does not contain any sensitive or personal

information and all our evaluation datasets are publicly available. However, societal and environmental aspects may need to be considered.

The circumvention of manual *uv*-mapping potentially changes or reduces the work carried out by artists. Throughout our research, we gained the impression that *uv*-mapping is overwhelmingly regarded as an undesirable task among artists. Thus, we do not expect any negative ethical implications on artist workflow. Rather, we see an opportunity for artists to focus on aspects of their work that they consider more creative and fulfilling, improving their productivity and job satisfaction.

Evaluation of our method's energy consumption is out of the scope of this work. While we recognize the importance of considering the environmental impact, our primary focus is on accuracy, memory consumption and performance. Future research could explore optimizing our method to reduce energy usage, contributing to more sustainable computational practices.

Bibliography

[1] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM transactions on graphics (TOG)*, vol. 41, no. 4, pp. 1–15, 2022.

[2] K. Vaidyanathan, M. Salvi, B. Wronski, T. Akenine-Möller, P. Ebelin, and A. Lefohn, "Random-access neural compression of material textures," *arXiv preprint arXiv:2305.17105*, 2023.

[3] T. Akenine-Moller, E. Haines, and N. Hoffman, "Texturing," in *Real-time rendering*, AK Peters/crc Press, 2019, ch. 6, pp. 147–200.

[4] Khronos Group, *glTF Sample Models*, Online, 2017. [Online]. Available: `https://github.com/KhronosGroup/glTF-Sample-Models`.

[5] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, "Chapter 6: Neural networks and deep learning," in *Machine learning: a first course for engineers and scientists*, Cambridge University Press, 2022, ch. 6, pp. 133–161.

[6] N. Rahaman, A. Baratin, D. Arpit, *et al.*, "On the spectral bias of neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 5301–5310.

[7] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[8] D. Benson and J. Davis, "Octree textures," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 785–790, 2002.

[9] D. Dolonius, E. Sintorn, and U. Assarsson, "Uv-free texturing using sparse voxel dags," in *Computer Graphics Forum*, Wiley Online Library, vol. 39, 2020, pp. 121–132.

[10] T. Müller, F. Rousselle, J. Novák, and A. Keller, "Real-time neural radiance caching for path tracing," *arXiv preprint arXiv:2106.12372*, 2021.

[11] L. Liu, J. Gu, K. Zaw Lin, T.-S. Chua, and C. Theobalt, "Neural sparse voxel fields," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 651–15 663, 2020.

[12] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 55–63.

[13] T. Takikawa, J. Litalien, K. Yin, *et al.*, "Neural geometric level of detail: Real-time rendering with implicit 3d shapes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 11 358–11 367.

[14] T. Müller, *tiny-cuda-nn*, version 1.7, Apr. 2021. [Online]. Available: `https://github.com/NVlabs/tiny-cuda-nn`.

[15] V. Kämpe, E. Sintorn, and U. Assarsson, "High resolution sparse voxel dags," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, pp. 1–13, 2013.

[16] Z. Chen, K. Yin, and S. Fidler, "Auv-net: Learning aligned uv maps for texture transfer and synthesis," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 1465–1474.

[17] J. Deng, S. Cheng, N. Xue, Y. Zhou, and S. Zafeiriou, "Uv-gan: Adversarial facial uv map completion for pose-invariant face recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7093–7102.

[18] M. Tarini, "Volume-encoded uv-maps," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–13, 2016.

[19] R. Poranne, M. Tarini, S. Huber, D. Panozzo, and O. Sorkine-Hornung, "Autocuts: Simultaneous distortion and cut optimization for uv mapping," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 6, pp. 1–11, 2017.

[20] E. W. Weisstein, "Barycentric coordinates," *https://mathworld. wolfram. com/*, 2003.

[21] C. Loop, "Smooth subdivision surfaces based on triangles," 1987.

[22] B. Jacob, S. Kligys, B. Chen, *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[23] P. Walloner, *Neural surface: Neural compression of surface properties using a geometry-associated feature hierarchy*, `https://github.com/pacex/neural-surface`, 2024.

[24] NVIDIA Corporation, *CUDA Toolkit*, Version 11.5.0, NVIDIA Corporation, 2022. [Online]. Available: `https://developer.nvidia.com/cuda-toolkit`.

[25] M. T. B. H. M. Müller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *Proc. VMV*, 2003, pp. 1–14.

[26] C. Choy. "Barycentric coordinate for surface sampling." (2015), [Online]. Available: `https://chrischoy.github.io/research/barycentric-coordinate-for-mesh-sampling/`.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, "Flip: A difference evaluator for alternating images.," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, no. 2, pp. 15–1, 2020.

[29] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[30] L. Demes, *Ambientcg*, Online, 3D models available under public domain license., 2019-2021. [Online]. Available: `https://www.ambientcg.com/`.

A

Images

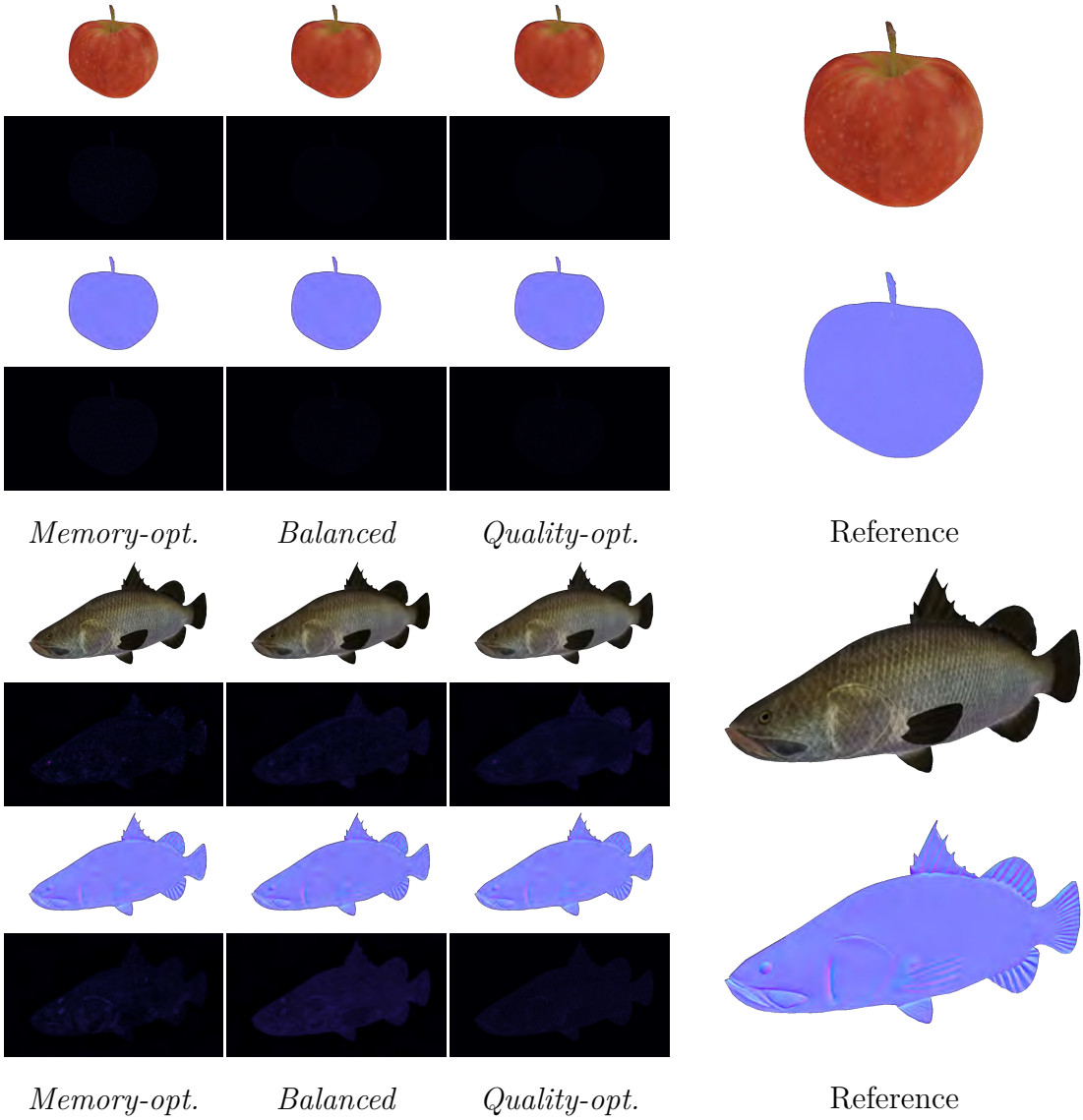


Figure A.1: Images sampling albedo and normal channels of our neural surface representation along with ꟻLIP heatmap to visualize differences to the reference. Datasets (top to bottom): *Apple*, *BarramundiFish*.

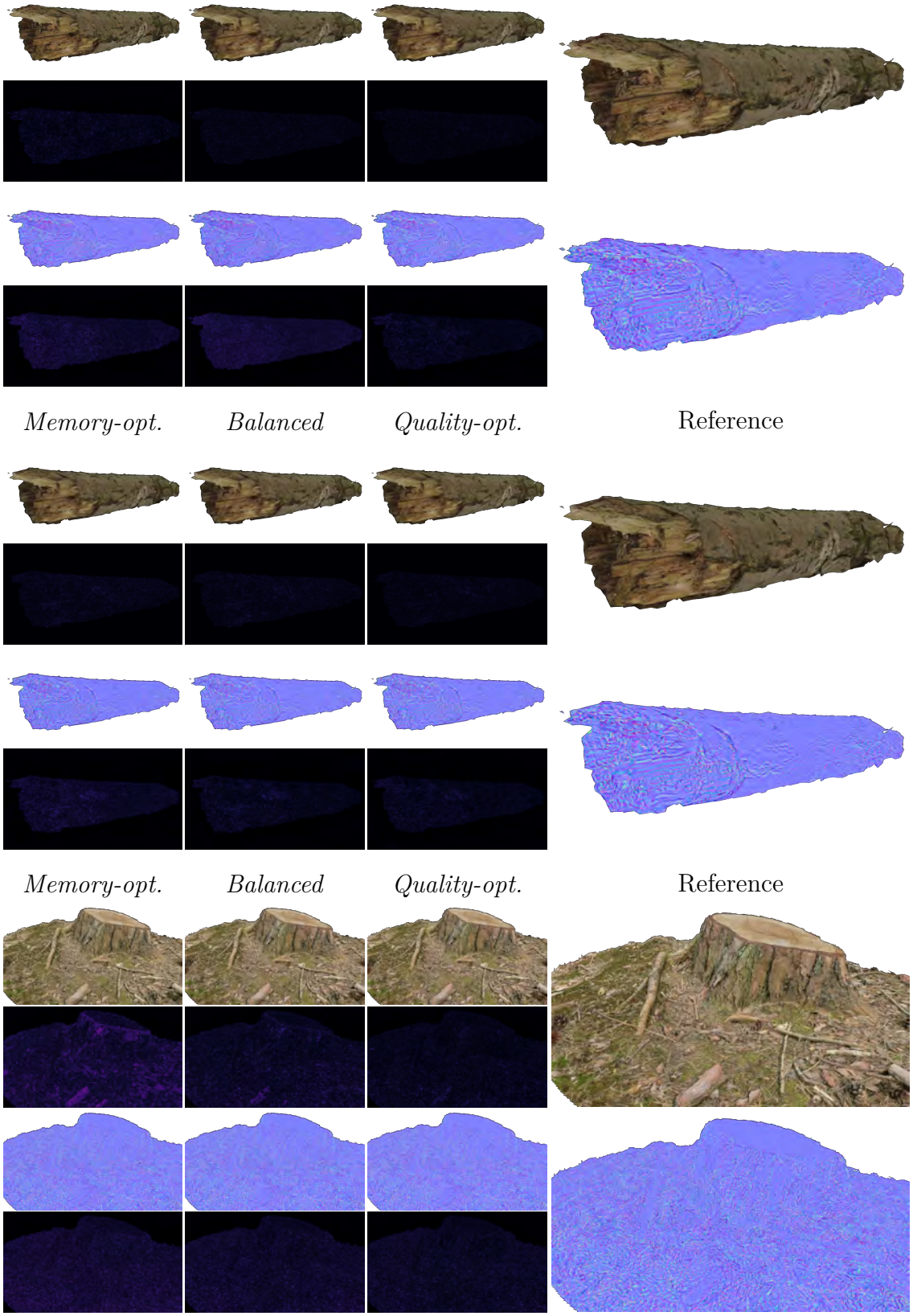


Figure A.2: Images sampling albedo and normal channels of our neural surface representation along with ꟻLIP heatmap to visualize differences to the reference. Datasets (top to bottom): *Stick*, *StickLow*, *TreeStump*.