High-Performance Graphics (2021)
N. Binder and T. Ritschel (Editors)

# Compression and Rendering of Textured Point Clouds via Sparse Coding

Kersten Schuster, Philip Trettner, Patric Schmitz, Julian Schakib, Leif Kobbelt

Visual Computing Institute, RWTH Aachen University, Germany

colored splats | 1 atom per splat | 0–8 atoms per splat

**Figure 1:** *Our proposed method compresses textured point clouds via sparse coding. We optimize a dictionary of atoms (each* $32 \times 32$ *pixels) and represent each splat texture as an average color plus a weighted sum of these atoms. With colored splats only, all high frequency content is lost (left). When adding a single atom, the quality is already improved significantly (center). Our approach can adapt the number of atoms per splat depending on texture content and the desired target quality.*

**Abstract**

*Splat-based rendering techniques produce highly realistic renderings from 3D scan data without prior mesh generation. Mapping high-resolution photographs to the splat primitives enables detailed reproduction of surface appearance. However, in many cases these massive datasets do not fit into GPU memory. In this paper, we present a compression and rendering method that is designed for large textured point cloud datasets. Our goal is to achieve compression ratios that outperform generic texture compression algorithms, while still retaining the ability to efficiently render without prior decompression. To achieve this, we resample the input textures by projecting them onto the splats and create a fixed-size representation that can be approximated by a sparse dictionary coding scheme. Each splat has a variable number of codeword indices and associated weights, which define the final texture as a linear combination during rendering. For further reduction of the memory footprint, we compress geometric attributes by careful clustering and quantization of local neighborhoods. Our approach reduces the memory requirements of textured point clouds by one order of magnitude, while retaining the possibility to efficiently render the compressed data.*

**CCS Concepts**

• ***Computing methodologies*** → ***Point-based models; Texturing; Image compression;***

## 1. Introduction

Digital 3D reconstructions capture the geometry and appearance of real-world objects and environments based on non-invasive imaging technology. They enable faithful replications of locations and buildings, which can be explored interactively everywhere and at any time. Such reconstructions are increasingly important in areas such as cultural heritage preservation, education in history and architecture, or virtual touring and telepresence applications.

High-precision terrestrial laser range scanners, camera-based depth sensors, or image-based 3D reconstruction by structure-from-motion offer a number of mature technologies to capture real environments with ever-increasing resolution and detail. The resulting datasets typically contain massive amounts of point cloud data combined with a large number of photographs. For high-precision laser range scanners, a reconstructed scene can easily consist of hundreds of high-resolution images and billions of 3D points.

One way to visualize such datasets is to render the point cloud data directly, without intermediate surface mesh extraction. For that, points are expanded to small disks, so-called splats, within their local tangent plane. This is sufficient for visual reproduction, but requires less manual preprocessing work. Automatically generating topologically accurate meshes from 3D scan data is often problematic due to insufficient sampling density [KBH06], and does not improve the visual quality. Point-based rendering techniques don't suffer from this limitation and can yield highly realistic renderings. High-quality splat rendering techniques use photographs for perspectively correct texturing, and perform shading based on a smoothly varying normal field for precise surface approximation [BHZK05].

While high-quality reproductions are possible this way, storing many high-resolution photographs in GPU memory for texture mapping quickly becomes a limiting factor. Existing methods solve the problem by page-based streaming of splat textures [SBMK20]. However, for their vast amount of data (1.2 terabyte on their dataset), they require high-end systems with large storage bandwidths and cause a continuously high CPU load for on-the-fly decompression. This can be prohibitive for many potential applications and target systems.

Our goal in this paper is to enable high-quality reproduction of textured point cloud data without the bandwidth and CPU overhead induced by streaming approaches. Instead, we devise an efficient compression scheme that drastically reduces the memory footprint of such datasets. This enables high-quality textured point cloud rendering on a wider range of target systems, and removes the continuous system load to enable new types of applications.

To this end, we propose a texture compression method that is targeted specifically at textured splat rendering. By using fixed-size snippets of input textures projected onto each splat as the basis of our representation, we leverage the advantages of block-based compression without introducing the typical compression artifacts. As neighboring splats are blended like described in [BHZK05], potentially visible block boundaries are concealed. A sparse coding technique finds an optimized representation of the texture content, that is reconstructed during rendering by weighted blending of a small number of atoms from a learned dictionary. We furthermore propose a compression scheme for the geometric attributes. Splat positions are clustered and encoded as local coordinates, and then quantized together with normals and cliplines into a representation that is suitable for uploading onto the GPU.

The resulting method permits random memory accesses for the compressed texture and geometric data, which is crucial for efficient evaluation during rendering. Our approach can furthermore be adapted to different memory and GPU performance budgets, by choosing a variable dictionary size and number of atoms per splat. We achieve compression ratios that outperform generic texture compression algorithms, while not suffering from visible block artifacts. We achieve PSNR scores comparable to input textures that have been downsampled once, but the perceived visual quality is superior at a much lower memory footprint. With our method, we are able to compress a dataset of 71 million splats, textured with 441 high-resolution photographs, to about 1 GB of GPU memory.

Our contributions to the efficient rendering of large textured point clouds can be summarized as follows:

- Compression of splat textures via sparse dictionary encoding
- Compression of splat geometry via clustering and quantization
- Efficient direct rendering of the compressed per-splat data

## 2. Related Work

### 2.1. Point Cloud Compression

Captured point clouds tend to be quite memory-intensive and thus, their compression is well-researched.

Several existing methods use octrees to efficiently encode positions, and simultaneously provide culling and level-of-detail [SK06]. De-duplicating similar subtrees leads to sparse voxel DAGs [KSA13; DKB*16; VMG17]. Instead of the uniform subdivision of octrees, DE QUEIROZ and CHOU use a *k*-d tree and predictive coding to better adapt to the data [dC16]. FAN et al. use hierarchical clustering for compression and level-of-detail and achieve less than one bit per normal-equipped point at good quality [FHP13].

Traditional image compression schemes can also sometimes be employed. One main source of point clouds is LiDAR sensor data, which is effectively a cylindrical heightmap and can thus directly benefit from traditional image compression [CTMT16]. In a more general setting, HOUSHIAR and NÜCHTER create panorama images of the point cloud and compress those [HN15]. Strong compression is especially required when continuous streams of point clouds are captured. THANOU et al. use a graph-based compression approach that is suited for scans of moving people [TCF16]. TU et al. focus on scenarios where the scanner itself moves and employ SLAM based predictions for higher compression [TTMT17].

The strong predictive capabilities of neural networks can be used for compression as well. TU et al. compress LiDAR data using recurrent networks [TTCT19], and WANG et al. use variational autoencoders for scanned objects [WZM*19].

For rendering, the point clouds must fit into GPU memory and are decoded on-the-fly. From the previous works, only the sparse voxel DAGs are suited for direct rendering. Other approaches that

support this try to find regions of the point cloud that can be approximated by a regular heightmap. These heightmaps can be stored and rendered as progressive quadtrees, the individual entries encoded via vector quantization [SMK07; SMK08].

Due to its importance, there have been several surveys and even standardization efforts by the MPEG for point cloud compression methods [SPB*19; LYL*20; GNK*20].

### 2.2. Sparse Coding

Learning sparse dictionaries to compress or represent data is a powerful approach in many domains. Popular algorithms for learning these dictionaries are matching pursuit [MZ93], orthogonal matching pursuit [RK95] and $k$-SVD [AEB06].

In geometry processing, sparse coding can for example be used to reconstruct surfaces [RXX*17], compute point cloud super-resolution [HCDC18], or provide compact representations of large terrains [GDGP16]. DIGNE et al. build a sparse dictionary of local tangent heightmaps and exploit self-similarity for compression [DCV14]. For more complete surveys, see [XWZ*15] or [LOM*18].

### 2.3. Image Compression

Our use case is rendering of textured point clouds via splats. Here, the amount of texture data exceeds the geometric data. Generic image compression such as PNG or JPEG are hard to decode during rendering, thus we focus on techniques that provide efficient random access.

There are several texture compression algorithms that have hardware and driver support from the graphics cards. Most notably, S3TC [Bro00] and BPTC [WD10], which are supported by all major GPU hardware and graphics APIs. ASTC [NLP*12] provides even better compression, but the hardware support on desktop GPUs is very limited. These methods are all block compression schemes and compress a fixed texture region, e.g. $4 \times 4$ pixels, in a fixed memory space, e.g. 128 bit. This fixed compression ratio enables extremely efficient hardware implementations and predictable performance. In the area of transform-based approaches, DCT [HPLW12] and DWT [MP12; ALM17] texture compression have been proposed and especially the latter can outperform the conventional block-based methods in terms of compression performance. For our use case, the compression ratios are typically not high enough.

Closer to our method are approaches based on sparse coding that learn dictionaries to encode blocks of images [ZGK11; PPKD15; SD17]. These methods often require significant computation time to encode the image but tend to have larger compression ratios for the same quality. Every block in the reconstructed image is a linear combination of a small number of dictionary atoms, which enables efficient random access. The results in [SD17] reveal that quality and compression ratio can be comparable to JPEG and JPEG2000 which, however, do not provide the crucial random access. BRYT and ELAD furthermore compare $k$-SVD to dimensionality reduction via Principal Component Analysis (PCA) for compression of images divided into fixed-sized square patches. They show that $k$-SVD performs significantly better in terms of the root-mean-square error (RMSE) for a given memory budget.

### 2.4. Rendering

There are different approaches to render large point clouds efficiently. This is in essence a surface reconstruction problem from a sampled input.

The first type of method renders colored splats. ZWICKER et al. reconstruct the surface color using an elliptical weighted average filter [ZPVBG01]. BOTSCH et al. improve the lighting by phong shading on elliptical splats [BSK04]. For increased performance, BOTSCH, HORNUNG, ZWICKER, and KOBBELT propose a multi-pass algorithm that uses a depth pre-pass to blend splat attributes in a narrow band around the top-most surface and compute the final color via deferred shading [BHZK05].

Point clouds can also be rendered as individual points followed by an image-space reconstruction step. MARROQUIM et al. use a push-pull procedure to interpolate between rendered points [MKC07]. In [SEMO14], approximate convex hulls are computed and a hidden point removal operator is used for image-space point cloud rendering.

For large, dense point clouds, rendering each point as a single pixel might be visually feasible and the challenge is to process the large number of points. WIMMER and SCHEIBLAUER devise an out-of-core algorithm that efficiently renders an view-appropriate sub-sampling of a point cloud that is too large for the CPU memory [WS06]. GÜNTHER et al. show that the typical triangle pipeline might be suboptimal for point rendering and how a GPGPU-only pipeline can be faster [GKLR13]. SCHÜTZ et al. present a progressive rendering system for rendering point clouds with multiple hundreds of millions of points [SMOW20].

Finally, combinations of splats and textures have been used to render high-fidelity captures of real-world data. WAHL et al. fit quads to the point cloud and creates a texture atlas for them [WGK05]. Similarly, GARCÍA et al. fit elliptical splats to build the atlas [GPBM15]. Often, the textures are captured from cameras and splats can be projected onto the camera image plane for texturing. An efficient way to dynamically select which camera to use for minimal occlusion is proposed in [YGW06]. SCHMITZ et al. use ellipsoidal splats and virtual texturing to render a 1.2 terabyte dataset in real-time for VR [SBMK20].

Our approach can be seen as a combination of dictionary-based image compression with a point cloud renderer based on blended textured splats, similar to [BHZK05].

## 3. Textured Point Cloud Compression and Rendering

### 3.1. Overview

Our method achieves high texture compression ratios by leveraging a sparse coding strategy on fixed-size texture snippets for each splat. It creates a set of dictionary entries, which we call atoms, that is optimized for the specific texture content of the input data.

To this end, we first project each splat into its associated camera

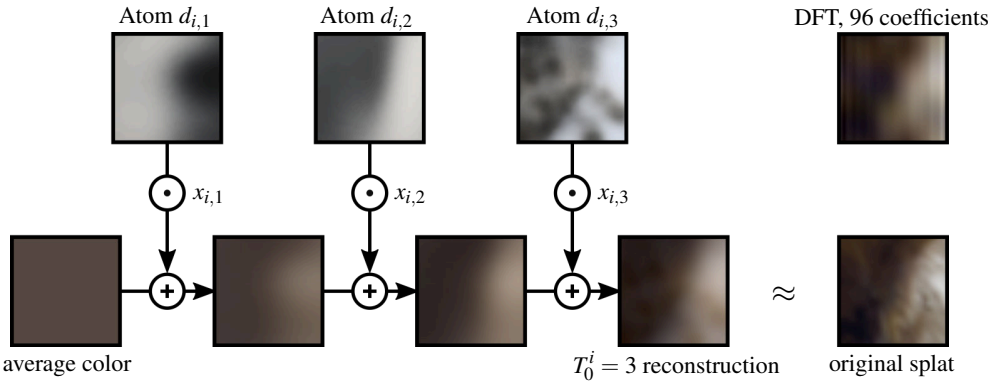



**Figure 2:** *The basis of our compression method is a learned dictionary that we use to reconstruct splat textures from a sparse representation. Each splat stores an average color and a variable number of atom indices $d_{i,T}$ and weights $x_{i,T}$. During rendering, we sample each atom from the dictionary, multiply by $x_{i,T}$, and add it to the average color. The diagram shows an example reconstruction of a splat using 3 atoms. Similarly to frequency-domain techniques, our learned dictionaries tend to capture low frequencies first. In practice, we use as many atoms as needed to reach a given quality target, with a global limit of 8 atoms. For comparison, the top right image shows a DFT reconstruction from the dominant $3 \times 32$ coefficients which has roughly the same quality as our depicted reconstruction. Our data-adaptive dictionary provides sparser reconstructions than the generic DFT basis.*

texture and resample it to a fixed-size representation (Section 3.2). Note that this is slightly different from conventional texture compression, where textures are divided into blocks of fixed size: we consider every splat to be a resampled texture portion of fixed resolution so that the terms splat texture and block are interchangeable. On these uniform blocks of texture content, a sparse decomposition is performed via the $k$-SVD algorithm [AEB06], that yields a set of dictionary atoms and a varying number of weight coefficients that encode the texture with a desired approximation quality (Section 3.3). The motivation for using $k$-SVD for data reduction is two-fold. On the one hand, it was shown to provide good tradeoffs between image reconstruction quality and compression ratio (cf. [SD17]). On the other hand, reconstructing texture portions via linear combinations of a small number of atoms is possible so that random access becomes feasible. An example splat and its first three atoms is shown in Figure 2. For comparison, we added a reconstruction from a Discrete Fourier Transform (DFT) using the 32 largest coefficients per color channel. The number of coefficients has been chosen so that the reconstruction quality is roughly comparable to our depicted reconstruction. Even though the number of coefficients could be reduced, e.g. by distributing them non-evenly on the color channels (as is done for JPEG by converting the color space to YCbCr), it is easy to see that our overcomplete dictionary can better adapt to the data with small numbers of coefficients.

The dictionary indices and corresponding weights for each splat are then quantized to yield a memory-efficient representation on the GPU that is suitable for immediate rendering (Section 3.4). Similarly, the geometric attributes of each splat are quantized to further reduce the GPU memory footprint. Splat positions are first encoded as local coordinates within spatial neighborhoods (Section 3.5).

Finally, the compressed splat representation is rendered, by decoding the geometric data in the vertex shader, and reconstructing the texture content per output pixel in the fragment shader via linear combination of the dictionary atoms (Section 3.6).

### 3.2. Splat Texture Generation

The input to our method is a number of camera views with an intrinsic and extrinsic camera matrix and associated rectified photographs. Each splat in the input dataset is assumed to have uniform size in object space and stores a position and normal vector plus an identifier of its associated camera view. To leverage a sparse dictionary coding strategy, our input textures are first resampled into fixed-size snippets for each splat in the dataset.

First, we compute the world coordinates of each splat's four corner points. Due to their rotational symmetry about the normal axis, the choice of the tangent space is arbitrary, but must be consistent between splat resampling and rendering. For example, it can be computed from the orientation of the normal and its alignment with the global global coordinate axes.

The splat corners are then projected into the texture of the associated camera view. We apply projective texture mapping with bilinear interpolation to resample the input texture to a fixed-size grid of texels that is aligned with the splat's local tangent directions. We resample the full rectangle area instead of just the circular inner part to use in the subsequent dictionary generation step. On the one hand this simplifies the texture lookup, because the boundary does not need special treatment, and on the other hand enables to use hardware-support for mipmap generation. We found $32 \times 32$ splats to be the sweet spot in our test case: the resampling quality is very high (52.63 dB) while data size is still manageable.

### 3.3. Splat Texture Compression

The total amount of data of the generated splat textures is typically much larger than that of the input textures due to overlapping of splats. Apart from the overlapping areas, there is further redundancy because of repeating details or uniformly colored areas that spread over multiple splats. In order to store splat textures compactly, we propose to learn a dictionary of splat-sized atoms using

the *k-SVD* algorithm [AEB06] and perform the sparse coding of splats via *Orthogonal Matching Pursuit* [MZ93; RK95].

The per-splat RGB textures, as well as the dictionary entries that encode them, are treated as vectors of dimension $n = 3 \cdot s^2$, where $s$ is the resampled splat texture size (e.g. 32). Following the notation of [AEB06], we approximate the non-constant part $y_i$ of a splat (the splat texture with the average color subtracted) as linear combination of a small number $T_0^i$ of learned atoms:

$$y_i \approx \sum_{T=1}^{T_0^i} x_{i,T} d_{i,T} \tag{1}$$

The coefficient $x_i$ denotes the influence of the dictionary entry (atom) $d_i$ when reconstructing the splat $y_i$. $T_0^i$ is the smallest number of atoms necessary to reconstruct $y_i$ under a given error bound, and further restricted by a global upper bound $T_0$:

$$T_0^i \leqslant T_0, \quad \forall i \in \{1, ..., \#\text{splats}\} \tag{2}$$

Note that the constant part (the splat's average color) is eventually added for the final reconstruction but is not part of the optimization. It would be possible to not regard the color separately and also reconstruct it using the dictionary. However, that would require at least one texture fetch even for uniformly colored splats.

The dictionary $D = [d_1, ..., d_K] \in \mathbb{R}^{n \times K}$ consists of $K$ atoms of dimension $n$. In the case of a 12 bit dictionary and $32 \times 32$ RGB splats, $D$ has the size $3072 \times 4096$.

For the compression of all splat textures $Y = \left[y_1, ..., y_{\#\text{splats}}\right]$, we want to find a dictionary $D$ and coefficients $X = \left[x_1, ..., x_{\#\text{splats}}\right]^T$ so that

$$\|Y - DX\| \to \min \quad s.t. \ \|x_i\|_0 \leqslant T_0 \quad \forall x_i \in X \tag{3}$$

Generally speaking, we want to find the dictionary $D$ that best describes the splat data while requiring the atom count per splat to be less than a predefined number $T_0$. This requirement ultimately enforces the sparsity in the coefficient vectors $x_i$, which is important for an efficient reconstruction.

After initializing the dictionary $D$ with a random subset of our inputs $Y$, we train it by alternatingly performing two operations:

1. Find the best sparse coding $X$ of inputs $Y$ given the current dictionary $D$.
2. Find the best dictionary $D$ for the sparse coding $X$.

For the first step, we employ the greedy *Orthogonal Matching Pursuit* [MZ93; RK95]. We successively project all dictionary atoms onto the input splats and subtract the projections that decrease the residual errors the most. Inputs and atoms are required to be normalized, so that the weight coefficient can be computed as the scalar product between the two vectors. The described procedure is repeated until either the residual error is below a predefined threshold or a maximum number of used atoms $T_0$ is reached. Throughout our experiments we set $T_0 := 8$ as we found the moderate error decrease for larger $T_0$ could not justify the increased evaluation/rendering cost (cf. Figure 9). By orthogonalizing the result, the residual error can often be decreased further. Given the input vector and the best atoms to approximate it (found in a greedy fashion), a linear regression provides optimal atom coefficients in the least-squares sense.

Once all input splats have been approximated and we obtained up to $T_0$ atoms and accompanying coefficients for each of those, the dictionary can be updated using the *k-SVD* algorithm [AEB06]. Every dictionary atom $d_k \in D$ is updated independently and we collect all input splats $y_i$ that use it in their approximate reconstructions. To determine the influence of $d_i$ on the reconstruction of a splat that uses it, the contribution of all other (up to $T_0 - 1$) atoms is subtracted from the respective input vector $y_i$. This process is repeated for all splats that use $d_k$ so that we receive a variety of residual vectors that should be matched by an updated $d_k$ as well as possible. For that, we construct a matrix $E_k$ with those vectors as columns. The number of rows is the dimensionality of the splat data, e.g. 3072 for $32 \times 32$. Computing a singular value decomposition $E = U \Sigma V^T$ enables the creation of a rank-1-approximation. The first column of $U$ yields the updated version of the normalized atom $d_k$, while the multiplication of $\Sigma(1,1)$ with the first row of $V^T$ are the new $d_k$-coefficients for the splats that use this atom in their reconstructions. After optimizing all atoms $d_k$, the updated dictionary $D$ can be used in the next sparse coding iteration.

The full optimization procedure is stopped after a fixed number of iterations, or when the average reconstruction error does not decrease significantly anymore. Convergence is guaranteed, since both steps cannot increase the approximation error. However, it should be noted that the found solution is not necessarily a global optimum. The output of the algorithm is the dictionary $D$ and for each input splat the average splat sRGB color and up to $T_0$ pairs of atom indices into $D$ and corresponding weights.

### Implementation Notes

In the following we want to share some details that will be helpful to implement our method.

To find the best-matching atom for each each splat during the sparse coding step, we do not have to test all atoms with linear time complexity. Instead, we can employ an accelerated search via a k-d tree and $L_2$ distances. Since the atoms are normalized, ordering of distances between the query splat and an atom is the same for $L^2$ and cosine distance if both vectors are located on the same hemisphere. Therefore, the original atom as well as its negative are added to the search space to also take solutions pointing away from the query into account which is accounted for by a negative atom weight. The k-d tree search can be further accelerated by employing a *best bin first* strategy [BL97] which yields good approximations at reduced search time.

It should be noted that the atom search during sparse coding, as well as the dictionary updates, can be highly parallelized. During the dictionary update step, AHARON et al. observed faster convergence when using already-updated atoms for subsequent updates in the same iteration [AEB06]. Due to the small size of our dictionaries, we chose to keep two independent copies (one to read from and one to write into), which enables us to perform the update in a highly parallel fashion without locking.

When computing the largest eigenvector for the dictionary update, we already have a good initial estimate. Therefore, we do not

need a costly SVD, but can rely on a few power iterations, where 10 was typically sufficient in our test cases.

The size of our resampled splat textures (approx. 200 GB for $32 \times 32$ RGB splats) can exceed the available memory budget during dictionary learning. Furthermore, the memory requirements of the update step largely depend on how many splats have been assigned to a particular atom. For very small dictionaries and large splat counts, the average number of occurrences for every atom is very high and can result in excessive memory usage. This can be circumvented by implementing the dictionary learning in an out-of-core fashion by performing the update steps independently per atom. The relevant splats can be loaded from disk on demand and do not have to be kept in main memory.

### 3.4. Splat Textures and Atom Quantization

While the original texture data requires significantly more space than the compressed splat data, the latter must not be ignored either. Each splat references a small, variable number of atom indices and weights. At an average of four atoms per splat, 32 bit integer indices and floating point weights would amount to 32 byte per splat, requiring 2.27 GB for our test dataset of 71 million splats.

Thus, we limit the index size to $k$ bit, which results in a dictionary size of $2^k$ atoms, and heavily quantize the weights. Furthermore, we split the dataset into batches according to the number of required atoms, making that information implicit. This has the added benefit of speeding up the rendering (cf. Section 3.6.1). Note that the quantization of the atom weight should be incorporated into the sparse coding: the additional error from quantizing the contribution of the $n$th atom can be considered when choosing the $n+1$-th atom.

The size of the atoms has a strong influence on the compression rate and feasible quantizations. Small atoms (e.g. $8 \times 8$ or $16 \times 16$) save some space by reduced dictionaries and correspondingly small atom indices. However, a much higher number of splats is necessary to achieve a desired object-space resolution. On the other hand, large atoms ($64 \times 64$ and above) require large dictionaries to accurately reconstruct the splat textures. So large, in fact, that the size of the dictionary becomes non-negligible. Thus, our experiments indicate $32 \times 32$ as the compression sweet spot and will be used for the rest of this paper. Note that for regions where much lower texture resolutions would suffice, rendering time is usually not wasted because of automatically fetching from coarser mipmaps. Non-power-of-two textures are theoretically possible, but interfere with mipmapping, especially if the dictionaries are not stored in an array texture.

Given this texture size, we evaluated atom indices of 8, 12, and 16 bit and corresponding dictionaries with 256, 4096, and 65 536 atoms. The weight quantization was chosen as the smallest possible range that does not introduce large reconstruction errors and is convenient for packing into GPU buffers. We settled for 4, 4, and 8 bit, leading to the three evaluated versions D8W4, D12W4, and D16W8. Table 1 lists the total sizes and bit rates on our sample dataset.

The average splat texture color is stored as 24 bit sRGB color per splat. Approaches to further quantize the color, e.g. the common `R5G6B5` compression (5 bit red, 6 bit green, 5 bit blue), lead to a visible quality degradation, e.g. because of banding.

### 3.5. Geometry Compression

Our main focus lies on compressing the texture data, which dominates the memory consumption. After our dictionary compression, the texture data stored in the dictionary is now negligible, but we added 63–109 bits per splat for average color, atom indices, and weights.

The geometric data per splat is a 3D position, a normal, and optionally a clipline for improved rendering of sharp features. Uncompressed, this amounts to 288 bit per splat and thus cannot be ignored. While offline compression methods can often reduce this to 10 bit and below, they are difficult to adapt to rendering in a memory-constrained setting, i.e. when the data must be decompressed on-the-fly. Instead, we use heavy quantization with additional clustering for positions. The result is roughly 54 bits per splat.

#### 3.5.1. Position

Laserscans of entire scenes are usually not amenable to direct quantization of their point positions. The bounding box is too large and quantization would either lead to unacceptable loss of quality or low compression ratios. Instead, we decompose the scene into small boxes of $K$ points each. For each of these chunks, we store a floating point position and size. The memory cost is well amortized over the $K$ points. As each chunk, except potentially the last, has exactly $K$ points, the association between point and chunk is implicit: given point index $i$, the chunk has index $\lfloor i/K \rfloor$. Each splat then only stores its relative position with a total amount of 24 bit and linearly interpolates the chunk bounding box. The distribution of the 24 bits depends on the shape of the box and is proportional to the binary logarithms of the extents. E.g., if the side lengths of a chunk are $a$, $2a$ and $4a$, the distribution of bits is 7, 8 and 9.

We compute the chunks via a simple recursive median split strategy. Given a point cloud, we compute its axis-aligned bounding box and split along the largest axis. For that, we sort the points along the coordinate axis that we want to split. The actual split is not performed at the exact median, but at the index that is a multiple of $K$ and closest to the median, thus ensuring that all chunks except the last always have exactly $K$ points. We recursively apply this scheme down to chunks of size $K$.

For our test dataset, we use $K = 128$. The average distance error is 0.326 mm at a splat radius of 20 mm. Each splat stores its position in 24 bit with additional 1.75 bit amortized from the per-chunk data. Figure 3 shows an excerpt of the chunks computed for our dataset.

We implemented and compared this simple scheme against the oriented bounding box fitting from [LK11]. However, this increased the preprocessing time significantly while slightly decreasing the quality. The reason for that is that the additionally stored orientation requires to increase $K$, which increases the error at equal data size per position.

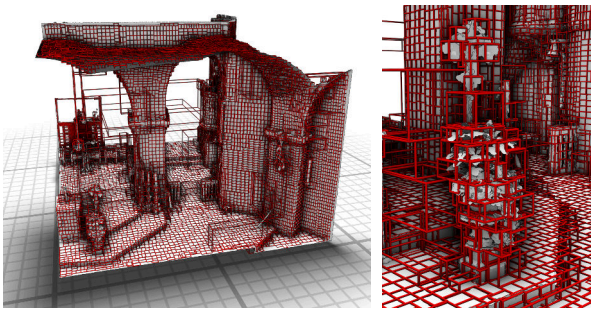


**Figure 3:** *A median split strategy is used to compute clusters of $K = 128$ splats (red bounding boxes). A cut-out and a close-up of our dataset is shown. The splat positions can be heavily quantized by storing them relative to their chunk bounding box. The per-chunk extra data is negligible when amortized over all splats.*

**Figure 4:** *Per-splat cliplines are used to improve reconstruction of sharp features. This is especially visible at silhouettes of edges. A close-up example is shown with cliplines (left) and without (right).*

#### 3.5.2. Normal and Clipline

Two additional geometric attributes are stored per splat. First, each splat has a normal vector that defines a tangent space and thus a local coordinate system for the splat texture. Second, our dataset also includes optional cliplines that have been pre-computed for splats with strong depth discontinuities in the laser scanner coordinate system. Roughly 6% of the splats have such an oriented line in tangent space (cf. 3.2) that clips the otherwise round splat [ZRB*04]. These are used to better represent sharp features as can be seen in Figure 4.

We use 28 bit to encode normals and cliplines. The first 16 bit contain octahedron-mapped world-space normals [MSS*10]. The cliplines are stored as line equation coefficients ($ax + by + c = 0$) with 4 bit per coefficient. They are normalized w.r.t. our chosen splat radius $r = 2\,\text{cm}$.

### 3.6. Compressed Splat Rendering

Our rendering extends the surface splatting method of [BHZK05] and consists of four passes (cf. Figure 5):

1. a compute shader for frustum culling of chunks and draw call generation
2. a *visibility pass* that fills the depth buffer with splats that are slightly pushed backwards
3. an *attribute pass* that accumulates weighted colors and normals of our textured splats
4. a *resolve pass* that computes the weighted averages and optionally applies relighting

As part of the position compression of Section 3.5, the splats are already sorted into chunks with known axis-aligned bounding boxes. The first pass is a compute shader that fills a GPU buffer with indirect draw commands and is invoked once per chunk. If the chunk intersects the view frustum, a draw call is generated for 4 vertices (a quad) and 8192 instances (the per-chunk splats). Otherwise, an empty draw call is created to maintain a 1-to-1 mapping from chunk index to draw command index. We did not observe a performance gain from a compaction step. The splats can now be rendered by a single `glMultiDrawArraysIndirect` call.

The visibility pass is close to a typical depth pre-pass: only a depth buffer is bound and all splats are rendered with a fragment shader that discards fragments outside the spherical splat shape or outside the clipping plane. The major difference to a normal depth pre-pass is that all depths are pushed back by a small world-space margin (2–5 cm in our scenes) to allow blending of splats close to the surface. In our experiments, applying this margin in the vertex shader is considerably faster than modifying `gl_FragDepth`, even when using a conservative depth specification. Figure 6 shows the effect and importance of the margin.

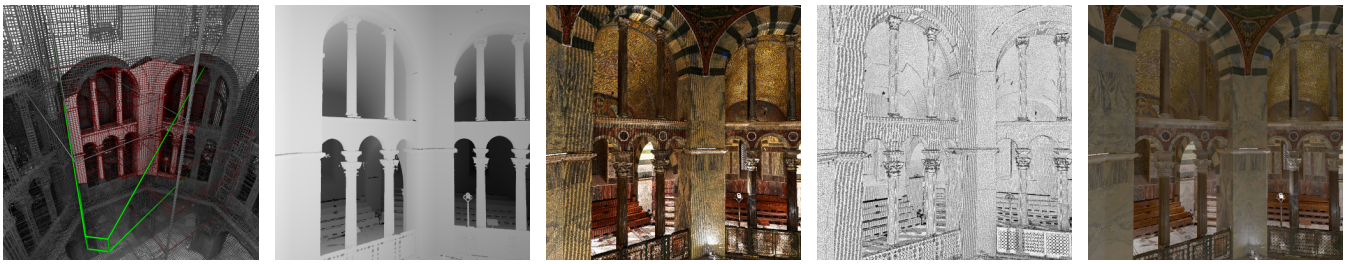
The attribute pass uses additive blending and writes into a `RGBA16F` render target to accumulate weight and weighted splat color. The weight of a fragment is chosen as $1 - d^2$ for simplicity, where $d \in [0,1]$ is the normalized distance to the splat center. Since the purpose of this is to prevent popping artifacts rather than to perform interpolation, a more sophisticated weighting does not improve the quality. If relighting should be supported, another `RGBA16F` target can be used to accumulate normals.

In the resolve pass, a fullscreen post-process divides the accumulated weighted color by the accumulated weight to obtain the output color. Additionally, the optional relighting and a gamma-correction is applied.

#### 3.6.1. Splat Shader

Splat-based rendering tends to have significant overdraw. Our test dataset has 3–5 splats on average that contribute to the final pixel. Thus, the fragment shaders have a considerable performance impact, and our profiling shows that the rendering is typically bandwidth-limited, especially for higher resolutions. Due to the chunk-based culling, the vertex shaders have limited impact on the runtime and we can decode the per-splat geometry and atom information in each vertex.

For maximum performance, we split the input point cloud into batches by the number of atoms needed (0–8 in our tests). Every subset has their own chunks and a shader that is specialized by the number of atoms. We start with the per-splat average color, which is read in the vertex shader and does not require a texture fetch. Each splat texture consists of a weighted sum of atoms, which are sampled from the dictionary and added in the fragment shader. In



**(a)** *culling* **(b)** *depth buffer* **(c)** *accumulated colors* **(d)** *accumulated weight* **(e)** *resolved color*

**Figure 5:** *Our rendering (extended from [BHZK05]) consists of four passes. First, a compute shader performs culling of splat chunks and populates a draw call buffer (a). Then, a visibility pass renders slightly pushed back splats to allow accumulation within a small margin (b). The attribute pass uses additive blending to accumulate weights and weighted colors in two render targets (c,d). Finally, a post-process resolve pass computes the properly blended output color (e). The stripe pattern in the accumulation targets is an artifact of the input splat distribution.*

**Figure 6:** *Instead of rendering only the front-most splat fragment, we use the multi-pass rendering of [BHZK05] to average the contribution of all splats within a certain margin. The figure shows an example without margin (left) and with a* 2 cm *one (right). The margin is essential for hiding the underlying splat structure.*

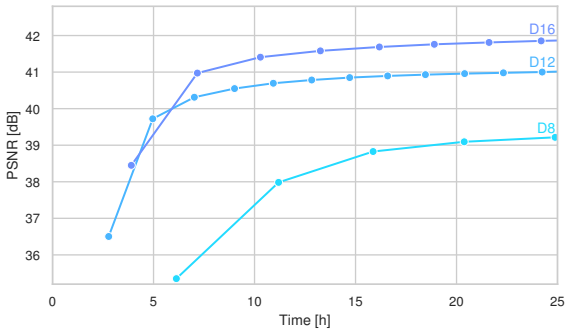



**Figure 7:** *Learning performance for three different dictionaries with 16 bit, 12 bit and 8 bit indices, respectively. Every dot marks one finished iteration (sparse coding + k-SVD dictionary update). The graphs show data for approximately one day and roughly indicate the convergence behavior. Note that the (quality decreasing) weight quantization is not performed here, but in a later subsequent single sparse coding step.*

contrast to traditional uncompressed rendering, this incurs more texture fetches but is significantly more cache-friendly for small dictionaries. In low-contrast areas, where splats are approximated by uniform color (i.e. 0 atoms), our method does not perform any texture fetches and can even be faster than the baseline. For large dictionaries, cache misses slow down rendering significantly. Table 3 and Section 4 evaluate this behavior in more detail.

Compared to the method of [BHZK05], we added the compute shader for culling and indirect draw call generation. However, our main contribution is the direct rendering of compressed splats, especially the on-the-fly reconstruction of the splat texture via weighted dictionary atoms.

## 4. Results and Discussion

We evaluated our method on a dataset of 71 million splats, acquired by multiple fused laser scans. Additionally, 441 camera images of size $3680 \times 2456$ were taken and each splat is annotated with the index of such a camera image. The implementation was performed in C++ and OpenGL using the Clang 7 compiler. To evaluate the rendering performance, an NVidia GeForce GTX 1080 with 8GB VRAM was used. The dictionary learning was performed on a cluster node with two Intel Xeon Gold 6136 CPUs (each with 12 cores and 24 threads @ 3.0 GHz) and 384 GB RAM.

Timings are given in Figure 7. For the 12 bit and 16 bit dictionaries, quality did not increase much after one day of training. However, for the 8-bit dictionary, the *k*-SVD step is extremely slow (between 3 and 5 hours) due to the high number of input residuals distributed among the 256 dictionary atoms. Within the following 64 hours, its quality increased by only 0.5 dB (not shown in the plot). For every splat, the sparse coding step was cut-off after a quality of 40 dB (or a maximum number of 8 atoms) was reached and took between 1 and 1.5 hours for the smaller dictionaries and up to 2 hours for the 16 bit dictionary.

The baseline renders the splats and, in the fragment shader, projects the world position into the image space of the annotated camera image to sample from the appropriate texture. We call this the "LOD 0" or "original" rendering. Our test dataset is too large for many graphics cards, and a straightforward compression would use DXT1 or BPTC or even downsample the input texture. In con-

| name | splat data (tex) | texture data | total data | bytes / splat | $\tau_{all}$ | $\tau_{texture}$ | quality (PSNR) |
|---|---|---|---|---|---|---|---|
| LOD 0 (original) | 142 MB | 15.94 GB | 16.56 GB | 233.2 | 1.0 | 1.0 | $\infty$ dB |
| LOD 0 BPTC | 142 MB | 5.31 GB | 5.93 GB | 83.5 | 2.8 | 2.9 | 45.59 dB |
| LOD 0 DXT1 | 142 MB | 2.66 GB | 3.28 GB | 46.2 | 5.0 | 5.7 | 40.04 dB |
| LOD 1 | 142 MB | 3.99 GB | 4.61 GB | 64.9 | 3.6 | 3.9 | 38.47 dB |
| LOD 1 BPTC | 142 MB | 1.33 GB | 1.95 GB | 27.5 | 8.5 | 10.9 | 36.55 dB |
| LOD 1 DXT1 | 142 MB | 0.66 GB | 1.29 GB | 18.2 | 12.8 | 20.1 | 35.07 dB |
| ours D8W4 | 514 MB | 1 MB | 0.99 GB | 14.0 | 16.7 | 31.2 | 38.10 dB |
| ours D12W4 | 577 MB | 0.02 GB | 1.07 GB | 15.1 | 15.5 | 27.1 | 39.09 dB |
| ours D16W8 | 966 MB | 0.27 GB | 1.71 GB | 24.1 | 9.7 | 13.0 | 41.36 dB |
| per splat texture | 0 MB | 290.82 GB | 291.30 GB | 4102.8 | | | |

**Table 1:** *Size of the 71 million splat dataset that we use for evaluation. The original texture data is a set of 441 camera images onto which the splats are projected. Geometry has been quantized (cf. 3.5) and all texture data contains 5 mipmap levels. Our method can be used with different dictionary sizes and weight quantizations (e.g. D8W4 is an 8 bit dictionary with 4 bit weights). We also compare against downsampled and block-compressed versions of the original data. As a reference, we also compute the size if each splat had their own* $32 \times 32$ *texture, i.e. the same size as our dictionary atoms. The total amount of data includes the* 477 MB *geometric data and is additionally shown in bytes per splat. The compression factor is shown including texture and geometric data ($\tau_{all}$) and texture-related data only ($\tau_{texture}$). Finally, we measure the quality of each compressed dataset as PSNR over all splats.*

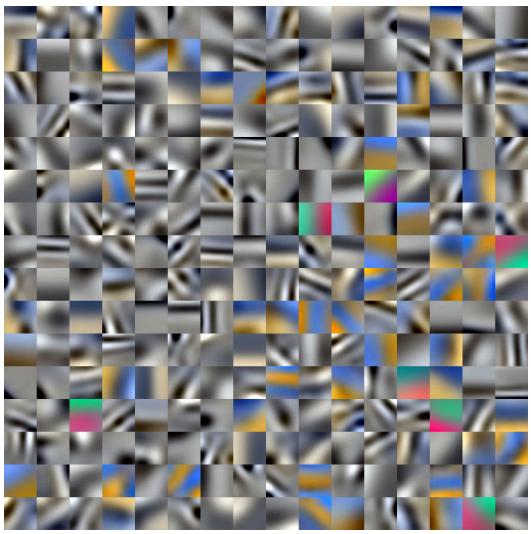


**Figure 8:** *A learned dictionary with 256 atoms of size* $32 \times 32$ *each. For the visualization, pixel values are scaled and mapped from* $[-1,1]$ *to* $[0,1]$*, i.e. grey represents 0. Each splat stores its average color, thus each atom has zero mean. Splat textures are encoded as a weighted sum of a small number of atoms.*

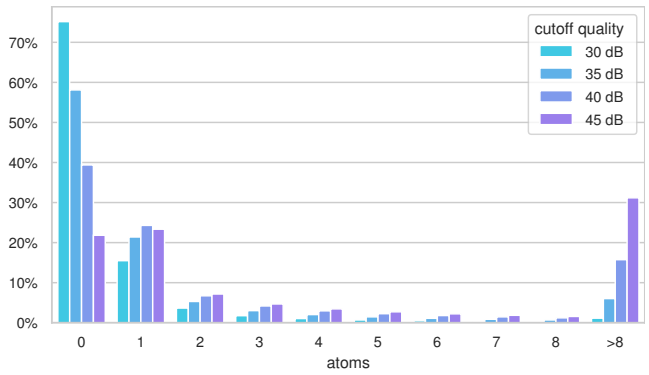



**Figure 9:** *The number of atoms per splat is variable and depends on the target cutoff quality, which effectively adapts to the local texture complexity. For different qualities, the distribution of atom counts are shown (for D16W8, the 16 bit dictionary with 8 bit weights). Note that splats store their average color explicitly, and thus might meet the quality target with 0 atoms, e.g. in homogeneous regions. To achieve* 40 dB *quality, more than 80% of the splats need no more than 8 atoms, while about 40% of the splats require no atoms.*

trast, our method computes a small dictionary of $32 \times 32$ atoms, and annotates each splat with a variable number of atom indices and discretized weights. While in the original rendering, many splats re-use overlapping texture regions, our method reconstructs unique per-splat textures. As a further comparison, we consider the case where each splat has a $32 \times 32$ texture, e.g. stored in a texture atlas.

Basic statistics and a quality measure of these versions are presented in Table 1. For the quality, we reconstruct the $32 \times 32$ texture for every splat and compute the peak signal-to-noise ratio (PSNR) compared to the original (resampled) splat texture. For the evaluated block compression methods, BPTC (BC7) and DXT1 (BC1), encoding was performed through the NVidia driver (version 460.73). Our method offers comparable PSNR at 2–3 times higher compression factors.

Table 2 shows qualitative results as close-up renderings from the captured point cloud. Downsampling and block compression can result in reasonable PSNR, but typically exhibit visible artifacts. Our approach, especially for small dictionaries, tends to result in a loss of high frequencies, but otherwise less visible artifacts.

An example of a learned dictionary is shown in Figure 8. It consists of some low-frequency atoms reminiscent of radial gradients and wavelets and a collection of high-frequency ones. Each splat

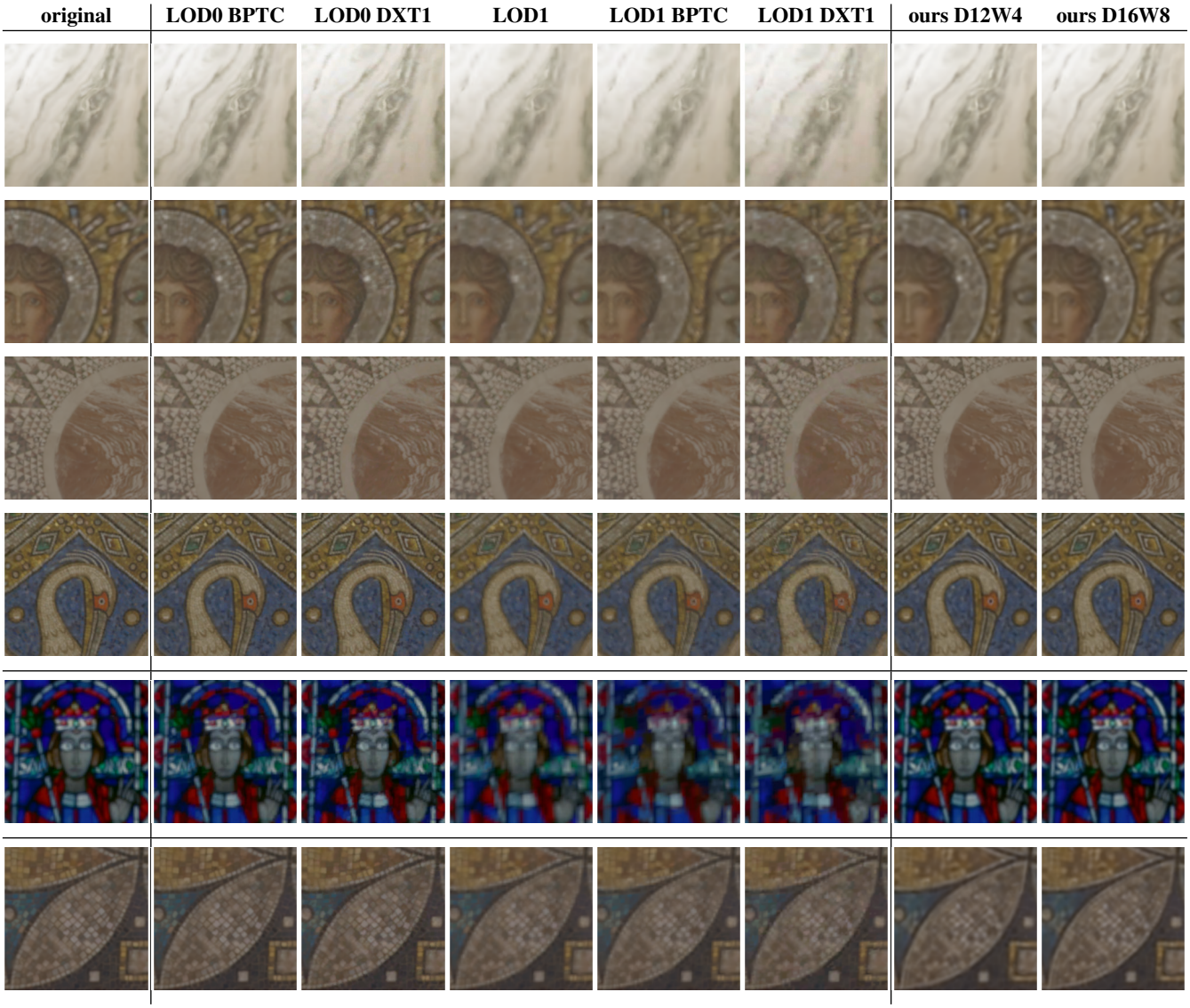



**Table 2:** *Example close-ups from our dataset, compared with different compression options. The first four rows are representative examples. The last two rows show two extreme cases: an example where our reconstruction is as good as the original and an example where our method performs poorly and erases important high-frequency detail. The images are best viewed in the digital version.*

references a variable number of atoms, depending on how many are needed to reach a certain quality target. Figure 9 shows the distribution of atom counts depending on the target quality. Our dataset contains some largely homogeneous regions, and as each splat stores its average color explicitly, a significant amount of splats actually requires zero atoms.

Finally, we evaluate our render performance on several test views in Table 3. Each pass of our rendering (cf. Section 3.6) is measured individually. Culling via compute shader, the visibility pass, and the resolve pass do not depend on the texture compression. Only the attribute pass is evaluated for the different compression versions. The baseline rendering, and its downsampled and block-compressed versions, use large textures and are highly bandwidth-limited. In contrast, our method requires more computations and texture fetches in the fragment shader, but the 12 bit dictionary is extremely small compared to the original dataset. Additionally, if a splat can be approximated by a uniform color (0 atoms), no texture fetch is necessary for rendering it. Thus, our rendering is actually faster than the baseline versions in some cases. A full 16 bit dictionary consumes 268 MB, which does not fit into any GPU cache. This slows down our method, as the atom reconstruction is effectively random access into the dictionary. On the other hand, the small 8 bit dictionary does not increase rendering performance or memory efficiency significantly compared to the 12 bit dictionary, but is 1 dB worse in quality. A promising avenue for future work

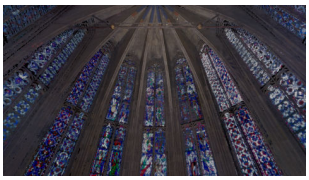
| |  |  |  |
|---|---|---|---|
| visible splats | 4.19 mio. | 3.76 mio. | 3.94 mio. |
| overdraw | 7.65 | 7.09 | 5.91 |
| $t_{\text{culling}}$ | 0.14 ms | 0.14 ms | 0.15 ms |
| $t_{\text{visibility}}$ | 5.12 ms | 4.53 ms | 4.69 ms |
| $t_{\text{attribute}}$ (LOD 0 BPTC) | 7.48 ms | 6.04 ms | 9.32 ms |
| $t_{\text{attribute}}$ (LOD 0 DXT1) | 7.51 ms | 6.03 ms | 9.28 ms |
| $t_{\text{attribute}}$ (LOD 1) | 7.50 ms | 6.06 ms | 9.38 ms |
| $t_{\text{attribute}}$ (LOD 1 BPTC) | 7.39 ms | 5.97 ms | 9.29 ms |
| $t_{\text{attribute}}$ (LOD 1 DXT1) | 7.37 ms | 5.97 ms | 9.33 ms |
| $t_{\text{attribute}}$ (ours D8W4) | 6.67 ms | 6.57 ms | 5.70 ms |
| $t_{\text{attribute}}$ (ours D12W4) | 6.85 ms | 6.96 ms | 5.77 ms |
| $t_{\text{attribute}}$ (ours D16W8) | 13.79 ms | 13.75 ms | 9.01 ms |
| $t_{\text{resolve}}$ | 0.08 ms | 0.08 ms | 0.08 ms |

**Table 3:** *Render statistics and timings of our four passes on an NVidia GTX 1080. The culling, visibility, and resolve passes are mostly independent of the texture compression. In the attribute pass, the splat fragment shaders have to decompress or reconstruct their textures and thus depend on the compression used. With small dictionaries, our method is more cache-friendly and can even result in faster rendering than the baseline. However, large dictionaries incur many cache misses and slow down rendering. Note that we did not evaluate the uncompressed LOD 0, as the texture data did not fit into GPU memory.*

is to reorder the atoms in the dictionary to minimize cache misses within a single splat.

In summary, our method offers a 2–3 times higher compression factor over traditional block-compression methods at comparable quality. While our compression is mainly free of traditional artifacts like visible blocks, aliasing, or color degradation, we instead tend to smooth over noise and high frequencies and non-repeating sharp features (that are not captured in any dictionary atom). When the dictionary is small and atom counts are low, our method can actually be faster than the uncompressed or block-compressed versions due to lower bandwidth utilization. However, large dictionaries suffer from many cache misses and thus degrade performance.

As was described before, $T_0$, the maximum number of atoms per splat, can be chosen freely to trade memory efficiency and rendering performance against reconstruction quality. Independent of the effective number of atoms that were determined during coding, the atom count can be limited dynamically at runtime to reduce the number of texture fetches. For the presented results and timings, we always used all available atoms.

## 5. Limitations and Future Work

In this paper, we focus on uniform geometry and texture density: splats at LOD 0 have a uniform fixed world-space size and a fixed texture resolution. One possible extension of our method would be to relax these requirements and encode splat size and shape in a few bits to support varying splat sizes, elliptical splats, and different texture densities. This would also require minor adjustments to the $k$-SVD for using different per-splat weight masks.

We currently rely on a rather straightforward quantization for the geometric splat attributes. A promising avenue for future work would be to use a sparse voxel octree or DAG for these. While this would decrease the decoding performance, it can still be used for real-time rendering and might save up to 50% memory.

Another issue is that high-frequency details tend to get smoothed by our approach. This is a result of minimizing the pixel-wise squared error, the size of the dictionaries and the way they are optimized. There are different avenues to explore to improve the quality in this regard. First, MSE (and thus PSNR) is a suboptimal metric to measure texture quality. A more perception-oriented error metric will most likely improve the perceived reconstruction quality but might be challenging to optimize dictionaries for. Second, splats can be slightly rotated or moved in tangent direction without changing the represented geometry. This degree of freedom could be used to better align texture content and thus improve visual quality while keeping the dictionary size fixed. Furthermore, the dictionary learning could be improved to make more effective use of the available atom count. Several heuristics proposed by [AEB06] could be investigated, such as purging atoms that are rarely used or too similar. The computationally more expensive FOCUSS algorithm could be evaluated [GR97], alongside a more recent variant that significantly improves its efficiency [HCZX08].

Finally, performance can always be optimized further. Our preprocessing step, the dictionary learning, is quite slow and typically needs hours for our dataset of 71 million splats. We already described several optimizations at the end of Section 3.3, but there is certainly room for improvement. On the rendering side, we are currently strongly bandwidth-limited due to overdraw. Some of the overdraw is inherent to the blended splat rendering, but proper occlusion culling, e.g. via a hierarchical z-buffer, would be beneficial.

While our method focuses on LOD 0, where memory constraints are the most critical, it is also applicable to higher LODs: after subsampling or clustering the splats, their textures can simply be added to our compression described in Section 3.3.

In this paper, we evaluated our method on just a single 3D scan dataset. While it is very diverse in appearance, the effectiveness of our approach needs to be validated on different inputs. It would also be worthwile to compare the achievable compression ratios when performing sparse coding based on generic dictionaries, such as the learned activations of the first layers of a neural network.

## 6. Conclusion

We presented a compression technique for textured point clouds that allows for efficient rendering without prior decompression. This is an important problem, as real-world captured data tends to require immense space that often exceeds GPU memory. Our method compresses geometry and texture and is able to directly render the compressed data.

We compress splat texture data using dictionary learning via $k$-SVD to create a sparse representation. For each splat, this yields an average color, and a variable-sized list of atom indices and weights. The weighted sum of these is an approximation of the original texture. The computed dictionaries exploit the self-similarity in the texture data.

In contrast to a general-purpose blackbox compression, we retain random-access into the texture data and thus enable efficient rendering. As the rendering is done via splat blending, and the dictionary atoms are in full resolution and bit-depth, our method does not suffer from many typical compression artifacts.

We have shown the viability of dictionary-based compression for textured splat rendering. Our approach reduces the memory requirements of textured point cloud datasets by one order of magnitude, while retaining the possibility to efficiently render the compressed data.

## Acknowledgments

This work was partially funded by the European Regional Development Fund within the "HDV-Mess" project under the funding code EFRE-0500038 and by the German Federal Ministry of Education and Research within the "VirtualDisaster" project under the funding code 13N15155. Furthermore, this work was supported by the German Research Foundation within the Gottfried Wilhelm Leibniz programme under the funding code KO 2064/6-1.

## References

[AEB06] AHARON, MICHAL, ELAD, MICHAEL, and BRUCKSTEIN, ALFRED. "K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation". *IEEE Transactions on signal processing* 54.11 (2006), 4311–4322 3–5, 11.

[ALM17] ANDRIES, BOB, LEMEIRE, JAN, and MUNTEANU, ADRIAN. "Scalable texture compression using the wavelet transform". *The Visual Computer* 33.9 (2017), 1121–1139 3.

[BE08] BRYT, ORI and ELAD, MICHAEL. "Compression of facial images using the K-SVD algorithm". *Journal of Visual Communication and Image Representation* 19.4 (2008), 270–282 3.

[BHZK05] BOTSCH, MARIO, HORNUNG, ALEXANDER, ZWICKER, MATTHIAS, and KOBBELT, LEIF. "High-quality surface splatting on today's GPUs". *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* IEEE. 2005, 17–141 2, 3, 7, 8.

[BL97] BEIS, JEFFREY S and LOWE, DAVID G. "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces". *Proceedings of IEEE computer society conference on computer vision and pattern recognition*. IEEE. 1997, 1000–1006 5.

[Bro00] BROWN, PAT. *EXT texture compression s3tc*. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_compression_s3tc.txt. [Online; accessed 12-03-2021]. 2000 3.

[BSK04] BOTSCH, MARIO, SPERNAT, MICHAEL, and KOBBELT, LEIF. "Phong splatting". *Proceedings of the First Eurographics conference on Point-Based Graphics*. 2004, 25–32 3.

[CTMT16] CHENXI TU, TAKEUCHI, E., MIYAJIMA, C., and TAKEDA, K. "Compressing continuous point cloud data using image compression methods". *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. 2016, 1712–1719 2.

[dC16] DE QUEIROZ, R. L. and CHOU, P. A. "Compression of 3D Point Clouds Using a Region-Adaptive Hierarchical Transform". *IEEE Transactions on Image Processing* 25.8 (2016), 3947–3956 2.

[DCV14] DIGNE, JULIE, CHAINE, RAPHAËLLE, and VALETTE, SÉBASTIEN. "Self-similarity for accurate compression of point sampled surfaces". *Computer Graphics Forum* 33.2 (2014), 155–164 3.

[DKB*16] DADO, BAS, KOL, TIMOTHY R, BAUSZAT, PABLO, et al. "Geometry and attribute compression for voxel scenes". *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library. 2016, 397–407 2.

[FHP13] FAN, Y., HUANG, Y., and PENG, J. "Point cloud compression based on hierarchical point clustering". *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*. 2013, 1–7 2.

[GDGP16] GUÉRIN, ERIC, DIGNE, JULIE, GALIN, ERIC, and PEYTAVIE, ADRIEN. "Sparse representation of terrains for procedural modeling". *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library. 2016, 177–187 3.

[GKLR13] GÜNTHER, CHRISTIAN, KANZOK, THOMAS, LINSEN, LARS, and ROSENTHAL, PAUL. "A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds". *Journal of WSCG* 21 (June 2013), 153 3.

[GNK*20] GRAZIOSI, D, NAKAGAMI, O, KUMA, S, et al. "An overview of ongoing point cloud compression standardization activities: Video-based (V-PCC) and geometry-based (G-PCC)". *APSIPA Transactions on Signal and Information Processing* 9 (2020) 3.

[GPBM15] GARCÍA, SERGIO, PAGÉS, RAFAEL, BERJÓN, DANIEL, and MORÁN, FRANCISCO. "Textured Splat-Based Point Clouds for Rendering in Handheld Devices". *Proceedings of the 20th International Conference on 3D Web Technology*. Web3D '15. Heraklion, Crete, Greece: Association for Computing Machinery, 2015, 227–230. ISBN: 9781450336475 3.

[GR97] GORODNITSKY, IRINA F and RAO, BHASKAR D. "Sparse signal reconstruction from limited data using FOCUSS: A re-weighted minimum norm algorithm". *IEEE Transactions on signal processing* 45.3 (1997), 600–616 11.

[HCDC18] HAMDI-CHERIF, AZZOUZ, DIGNE, JULIE, and CHAINE, RAPHAËLLE. "Super-Resolution of Point Set Surfaces Using Local Similarities". *Computer Graphics Forum* 37.1 (2018), 60–70 3.

[HCZX08] HE, ZHAOSHUI, CICHOCKI, ANDRZEJ, ZDUNEK, RAFAL, and XIE, SHENGLI. "Improved FOCUSS method with conjugate gradient iterations". *IEEE transactions on signal processing* 57.1 (2008), 399–404 11.

[HN15] HOUSHIAR, H. and NÜCHTER, A. "3D point cloud compression using conventional image compression for efficient data transmission". *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*. 2015, 1–8 2.

[HPLW12] HOLLEMEERSCH, CHARLES-FREDERIK, PIETERS, BART, LAMBERT, PETER, and Van de WALLE, RIK. "A new approach to combine texture compression and filtering". *The Visual Computer* 28.4 (2012), 371–385 3.

[KBH06] KAZHDAN, MICHAEL, BOLITHO, MATTHEW, and HOPPE, HUGUES. "Poisson surface reconstruction". *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 2006 2.

[KSA13] KÄMPE, VIKTOR, SINTORN, ERIK, and ASSARSSON, ULF. "High resolution sparse voxel dags". *ACM Transactions on Graphics (TOG)* 32.4 (2013), 1–13 2.

[LK11] LARSSON, THOMAS and KÄLLBERG, LINUS. "Fast Computation of Tight-Fitting Oriented Bounding Boxes". Feb. 2011, 3–19. ISBN: 978-1-56881-437-7 6.

[LOM*18] LESCOAT, THIBAULT, OVSJANIKOV, MAKS, MEMARI, POORAN, et al. "A Survey on Data-driven Dictionary-based Methods for 3D Modeling". *Computer Graphics Forum* 37.2 (2018), 577–601 3.

[LYL*20] LIU, H., YUAN, H., LIU, Q., et al. "A Comprehensive Study and Comparison of Core Technologies for MPEG 3-D Point Cloud Compression". *IEEE Transactions on Broadcasting* 66.3 (2020), 701–717 3.

[MKC07] MARROQUIM, RICARDO, KRAUS, MARTIN, and CAVALCANTI, PAULO. "Efficient Point-Based Rendering Using Image Reconstruction". Jan. 2007, 101–108 3.

[MP12] MAVRIDIS, PAVLOS and PAPAIOANNOU, GEORGIOS. "Texture compression using wavelet decomposition". *Computer graphics forum*. Vol. 31. 7. Wiley Online Library. 2012, 2107–2116 3.

[MSS*10] MEYER, QUIRIN, SÜSSMUTH, JOCHEN, SUSSNER, GERD, et al. "On floating-point normal vectors". *Computer Graphics Forum*. Vol. 29. 4. Wiley Online Library. 2010, 1405–1409 7.

[MZ93] MALLAT, STÉPHANE G and ZHANG, ZHIFENG. "Matching pursuits with time-frequency dictionaries". *IEEE Transactions on signal processing* 41.12 (1993), 3397–3415 3, 5.

[NLP*12] NYSTAD, J, LASSEN, A, POMIANOWSKI, A, et al. "Adaptive Scalable Texture Compression". *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings* (Jan. 2012) 3.

[PPKD15] PATI, NIBEDITA, PRADHAN, ANNAPURNA, KANOJE, LALIT KUMAR, and DAS, TANMAYA KUMAR. "An Approach to Image Compression by Using Sparse Approximation Technique". *Procedia Computer Science* 48 (2015). International Conference on Computer, Communication and Convergence (ICCC 2015), 769–775. ISSN: 1877-0509 3.

[RK95] REZAIIFAR, RAMIN and KRISHNAPRASAD, P. "Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition". (June 1995) 3, 5.

[RXX*17] REMIL, OUSSAMA, XIE, QIAN, XIE, XINGYU, et al. "Surface reconstruction with data-driven exemplar priors". *Computer-Aided Design* 88 (2017), 31–41. ISSN: 0010-4485 3.

[SBMK20] SCHMITZ, PATRIC, BLUT, TIMOTHY, MATTES, CHRISTIAN, and KOBBELT, LEIF. "High-Fidelity Point-Based Rendering of Large-Scale 3-D Scan Datasets". *IEEE computer graphics and applications* 40.3 (2020), 19–31 2, 3.

[SD17] SAHOO, ARABINDA and DAS, PRANATI. "Dictionary based Image Compression via Sparse Representation". *International Journal of Electrical and Computer Engineering (IJECE)* 7 (Aug. 2017), 1964 3, 4.

[SEMO14] Machado e SILVA, R., ESPERANÇA, C., MARROQUIM, R., and OLIVEIRA, A. A. F. "Image Space Rendering of Point Clouds Using the HPR Operator". *Computer Graphics Forum* 33.1 (2014), 178–189 3.

[SK06] SCHNABEL, RUWEN and KLEIN, REINHARD. "Octree-based Point-Cloud Compression." *Spbg* 6 (2006), 111–120 2.

[SMK07] SCHNABEL, RUWEN, MÖSER, SEBASTIAN, and KLEIN, REINHARD. "A Parallelly Decodeable Compression Scheme for Efficient Point-Cloud Rendering". *Symposium on Point-Based Graphics 2007*. Sept. 2007, 214–226 3.

[SMK08] SCHNABEL, RUWEN, MÖSER, SEBASTIAN, and KLEIN, REINHARD. "Fast vector quantization for efficient rendering of compressed point-clouds". *Computers & Graphics* 32.2 (2008), 246–259. ISSN: 0097-8493 3.

[SMOW20] SCHÜTZ, MARKUS, MANDLBURGER, GOTTFRIED, OTEPKA, JOHANNES, and WIMMER, MICHAEL. "Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures". *Computer Graphics Forum* 39.2 (2020), 51–64 3.

[SPB*19] SCHWARZ, S., PREDA, M., BARONCINI, V., et al. "Emerging MPEG Standards for Point Cloud Compression". *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.1 (2019), 133–148 3.

[TCF16] THANOU, D., CHOU, P. A., and FROSSARD, P. "Graph-Based Compression of Dynamic 3D Point Cloud Sequences". *IEEE Transactions on Image Processing* 25.4 (2016), 1765–1778 2.

[TTCT19] TU, C., TAKEUCHI, E., CARBALLO, A., and TAKEDA, K. "Point Cloud Compression for 3D LiDAR Sensor using Recurrent Neural Network with Residual Blocks". *2019 International Conference on Robotics and Automation (ICRA)*. 2019, 3274–3280 2.

[TTMT17] TU, C., TAKEUCHI, E., MIYAJIMA, C., and TAKEDA, K. "Continuous point cloud data compression using SLAM based prediction". *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, 1744–1751 2.

[VMG17] VILLANUEVA, ALBERTO JASPE, MARTON, FABIO, and GOBBETTI, ENRICO. "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". *Journal of Computer Graphics Techniques (JCGT)* 6.2 (2017), 1–30. ISSN: 2331-7418 2.

[WD10] WERNESS, ERIC and DANIELL, PIERS. *ARB texture compression bptc*. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_compression_bptc.txt. [Online; accessed 12-03-2021]. 2010 3.

[WGK05] WAHL, ROLAND, GUTHE, MICHAEL, and KLEIN, REINHARD. "Identifying Planes in Point-Clouds for Efficient Hybrid Rendering". *The 13th Pacific Conference on Computer Graphics and Applications*. Oct. 2005 3.

[WS06] WIMMER, MICHAEL and SCHEIBLAUER, CLAUS. "Instant Points: Fast Rendering of Unprocessed Point Clouds". *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG'06. Boston, Massachusetts: Eurographics Association, 2006, 129–137. ISBN: 3905673320 3.

[WZM*19] WANG, JIANQIANG, ZHU, HAO, MA, ZHAN, et al. "Learned point cloud geometry compression". *arXiv preprint arXiv:1909.12037* (2019) 2.

[XWZ*15] XU, LINLIN, WANG, RUIMIN, ZHANG, JUYONG, et al. "Survey on sparsity in geometric modeling and processing". *Graphical Models* 82 (2015), 160–180. ISSN: 1524-0703 3.

[YGW06] YANG, RUIGANG, GUINNIP, DAVID, and WANG, LIANG. "View-dependent textured splatting". *The Visual Computer* 22 (2006), 456–467 3.

[ZGK11] ZEPEDA, JOAQUIN, GUILLEMOT, CHRISTINE, and KIJAK, EWA. "Image Compression Using Sparse Representations and the Iteration-Tuned and Aligned Dictionary". *Selected Topics in Signal Processing, IEEE Journal of* 5 (Oct. 2011), 1061 –1073 3.

[ZPVBG01] ZWICKER, MATTHIAS, PFISTER, HANSPETER, VAN BAAR, JEROEN, and GROSS, MARKUS. "Surface splatting". *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, 371–378 3.

[ZRB*04] ZWICKER, MATTHIAS, RASANEN, JUSSI, BOTSCH, MARIO, et al. "Perspective accurate splatting". *Proceedings-Graphics Interface*. CONF. 2004, 247–254 7.