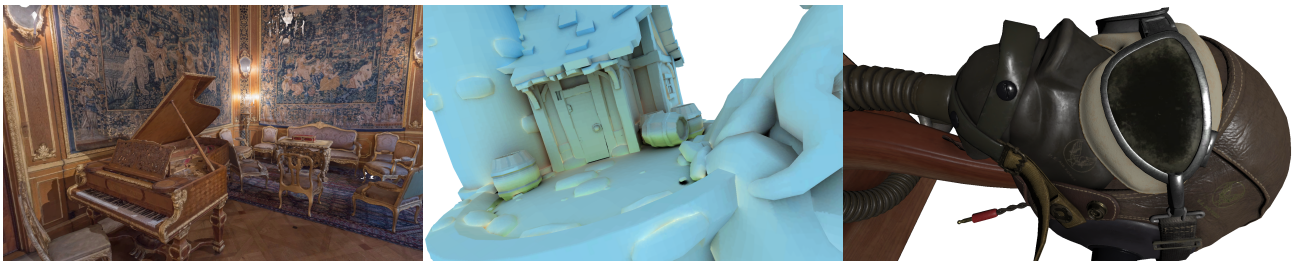
EUROGRAPHICS 2020 / U. Assarsson and D. Panozzo
(Guest Editors)

Volume 39 (2020), Number 2

UV-free Texturing using Sparse Voxel DAGs

D. Dolonius[1], E. Sintorn[1] and U. Assarsson[1]

[1]Chalmers University of Technology, Sweden



(a) **(b)** **(c)**

Figure 1: *For full HD and compressed textures, a)* THEGREATDRAWINGROOM *uses one quad-linear-filtered texture lookup in voxel space (the analogy to standard trilinear mipmap filtering of 2D textures) per pixel at* 5 *ms per frame; b)* SKYHOUSE *uses multisampling of one baked irradiance texture at 2 ms; and c)* FLIGHTHELMET *uses three quad-linear-filtered textures for albedo, normal, occlusion, roughness, and metallic at 5 ms in total. No uv mapping required.*

Abstract

An application may have to load an unknown 3D model and, for enhanced realistic rendering, precompute values over the surface domain, such as light maps, ambient occlusion, or other global-illumination parameters. High-quality uv-unwrapping has several problems, such as seams, distortions, and wasted texture space. Additionally, procedurally generated scene content, perhaps on the fly, can make manual uv unwrapping impossible. Even when artist manipulation is feasible, good uv layouts can require expertise and be highly labor intensive.

This paper investigates how to use Sparse Voxel DAGs (or DAGs for short) as one alternative to avoid uv mapping. The result is an algorithm enabling high compression ratios of both voxel structure and colors, which can be important for a baked scene to fit in GPU memory. Specifically, we enable practical usage for an automatic system by targeting efficient real-time mipmap filtering using compressed textures and adding support for individual mesh voxelizations and resolutions in the same DAG. Furthermore, the latter increases the texture-compression ratios by up to 32% *compared to using one global voxelization, DAG compression by* $10-15\%$ *compared to using a DAG per mesh, and reduces color-bleeding problems for large mipmap filter sizes.*

The voxel-filtering is more costly than standard hardware 2D-texture filtering. However, for full HD with deferred shading, it is optimized down to 2.5 ± 0.5 *ms for a custom multisampling filtering (e.g., targeted for minification of low-frequency textures) and* 5 ± 2 *ms for quad-linear mipmap filtering (e.g., for high-frequency textures). Multiple textures sharing voxelization can amortize the majority of this cost. Hence, these numbers involve 1-3 textures per pixel (Fig. 1c).*

CCS Concepts

• ***Computing methodologies*** → ***Texturing;***

1. Introduction

Texturing is a ubiquitous and core part of 3D modeling and rendering. However, standard texturing requires some *uv* mapping, and for many scenarios, such as for light-map computations and other global-illumination parameters, *uv* unwrapping has to be performed such that no surface points share*uv* coordinates. I.e., the mapping has to be injective.

UV unwrapping is hard to automate in a universally satisfying way, since an object's surface generally cannot be unwrapped without artificial seams and undesired distortion. This causes varying texture resolution over the surface domain and problems with correct texel interpolation and mipmap filtering over the seams. The latter often requires padding between patches and hence, limited mipmap-filter sizes. Other drawbacks include vertex duplication and wasted *uv* space.

For a solution that must be reliable and fully automated, it is at-

tractive to avoid these problems, even at the cost of run-time performance. Common examples include architectural visualization and CAD models. An option is to use a voxel space for the texel mappings [BD02; DKB*16; DSKA18; LH06; LK10; CNLE09]. This avoids *uv* coordinates, since there is an implicit, injective surface-to-texel mapping by the surface's 3D position. If the voxel structure is sparse with no empty voxels, this mapping may also be bijective, such that texture space is not wasted. Using voxel space allows generality of surface models, e.g., triangles, quads, implicit or parametric surfaces, ISO surfaces, lines, point clouds, subdivision surfaces, etc. Animated and/or skinned objects are trivially supported by using the objects rest pose for the texel mapping [BD02].

Dado et al. [DKB*16] and Dolonius et al. [DSKA18] show that colors stored in a voxel data-structure can be heavily compressed while maintaining good quality and real-time lookup performance. Their underlying geometry data is stored as a Sparse Voxel DAG [KSA13] that is raytraced to produce an image, but in this paper we consider using the same data structure only as a means of storing surface colors, to be used in a standard rasterization pipeline. We build on the method by Dolonius et al. [DSKA18] due to the high compression ratios vs. other real-time formats. Textures with an injective or bijective surface-to-texel mapping can be significantly more memory demanding than standard repeated textures. Therefore, the use of texture compression can be important for fitting the textures in GPU memory. To aid the reader, Section 3 provides a quick recap of how to represent colored voxels with DAGs.

Nevertheless, directly using the voxel structure presents a number of problems as compared to traditional texture mapping. Here is the list of problems, followed by our contributions:

- Using a global voxelization of the entire scene can cause disjointed surface parts to fall into the same voxel and thus share a texel. This is a classic issue for voxel-based textures and can manifest itself as color bleeding between nearby surfaces [BD02].
- The above problem is even more pronounced when using mipmap hierarchies and large filter sizes. Voxel colors of separate surfaces may erroneously be averaged together at coarser mipmap levels.
- A solution that inherently allows per-mesh resolutions and voxel sizes can be desirable over explicitly orchestrating fully separate voxel structures (i.e., DAGs).
- Fast filtered texture lookups are important to maintain real-time performance. Thus, optimized accesses are vital.

Color bleeding can be avoided by assigning conflicting parts to separate voxelizations. This is often considered unattractive. Instead, solutions such as normal flags that prevent the use of too high mip levels at problematic regions are favored [BD02].

In contrast, to target the first three of the listed problems, we create a voxelization per mesh by default, potentially of individual grid resolutions and voxel sizes, and develop a solution that can store these separate mesh voxelizations in the same DAG despite resolution (and hence DAG-height) differences. The DAG contains a root per mesh, but lower nodes are typically shared. A DAG-traversal path then becomes unique for each voxel and mesh, which automatically prohibits neighboring voxels of separate meshes to incorrectly be blended together during mipmap generation. This alleviates filtering with filter sizes up to full mesh sizes, which is a problem for standard *uv*-unwrapping-based methods (Mesh Color Textures being an exception [Yuk17]).

If disjoint parts of a mesh share a voxel, the color-bleeding problem remains. When manual intervention is allowed, an artist could split the mesh into submeshes [BD02], thereby creating a separate voxelization per such submesh. Alternatively, undesirable original seams between meshes can be removed by merging the meshes, thereby smoothing the seams by allowing filtering over the original mesh borders.

Storing all voxelizations in the same DAG improves the compression of the geometric information. As we will demonstrate, allowing individual voxelizations per mesh increases the texture compression of the voxel colors by up to 32 percent.

Using 16 individual texture lookups for quadlinear-filtering (two mip levels times eight texels) is costly [DSKA18]. Instead, we demonstrate how to optimize the magnification- and minification-filtering. For full HD, the timings are as follows: 2.5 ± 0.5 ms for a custom multisampling filter, 5 ± 2 ms for quad-linear mipmap filtering, and about 1 ms for nearest neighbor filtering. The quad-linear filter would typically be used for high-frequency textures and is a factor 2.5 ± 0.2 times faster than a straight-forward solution that just takes 16 individual texture lookups. The multisampling filter could, for instance, be used for more low-frequency textures.

With deferred shading, one filtered voxel lookup per pixel (or a few for transparency) is realistic. Furthermore, the computational overhead for multiple textures, each compressed individually but sharing voxel structure, is relatively small. Our method can also be used side-by-side with standard uv-based hardware-accelerated texturing.

2. Related work

A vast set of methods have been proposed to perform automatic *uv* unwrapping and target the problems of seams and distortions [SPGT18; PTH*17; SLMB05; ZMT05; KLS03; SWG*03; SCGL02; LPRM02; DMA02]. An advantage of those methods is that after the *uv* unwrapping has been done, they can typically utilize, to a varying degree, the existing texture-filtering hardware, thus making real-time texture lookups and filtering very fast during rendering.

For surface parameterization, there are two surveys [HPS08; SPR06]. Yuksel et al. provide a recent overview of previous work on alternatives to standard texture mapping [YLT19; TYL17].

Hiding Seams Two reasons why a *uv* map of a continuous surface requires cuts, or seams, are: 1) undesired texture-resolution distortion, and 2) when the surface is not topologically a disk. Due to its added memory cost, an undesirable feature inherent to creating seams during *uv* unwrapping is *vertex duplication*, i.e. that vertices may have to be split into two or more vertices with separate *uv* values. Additionally, the splits cause problems with visible seams when doing mipmap filtering over their borders.

By reassuring that topologically adjacent triangles with an artificial seam between them are separated in *uv* space by a pure integer translation and possibly rotations of 90 degrees and duplicating these triangles' texel neighborhoods, the seams can be made invisible on the screen, despite filtering [RNLL10]. However, this puts restrictions on filter sizes corresponding to the width of the added texel neighborhoods.

For purely cylindrical or toroidal texture maps, Tarini presents a very simple and efficient solution without the need for duplicate vertices and only with a small modification in the vertex and fragment shader [Tar12]. By enforcing quadrilateral surface patches mapped to square charts in texture space, Purnomo et al. achieve seamless mipmap filtering [PCK04]. Liu et al. create a linear operator where its null space represents seamless texture borders [LFJG17].

Connectivity-based representations Ptex has become increasingly popular, where the idea is to assign a texture per face, and comes in both offline [BL08] and real-time flavors [McD13; MB11]. Mesh colors [YKH10; Yuk16] define an implicit *uv* mapping per triangle by using only one resolution value per triangle. Filtering over triangle edges can also elegantly be handled by assuring that texels or colors along triangle edges are centered on the edges and vertices.

Mesh Color Textures [Yuk17] enable artifact-free filtering also for mip-map levels coarser than a color per vertex, by for those levels compute pre-filtered per-vertex colors but keeping the storage resolution to a color per vertex. For high tessellations, on par with the texture resolution, the storage overhead could be larger than the 33% of standard mip maps. However, standard hardware texture lookups can be used, via a custom four-dimensional *uv* format. Mallett et al. [MSY19] introduce a hardware modification for anisotropic filtering. A mesh-color texture, at least for its finer mipmap levels, visually contains a similar block coherence as Dolonius et al. [DSKA18] that we use.

Volume-based Parametrizations PolyCube-Maps suggests splitting objects into cubic regions of which the surface colors can be mapped onto cube maps [THCM04]. Lefebvre and Dachsbacher suggest using a few orthogonal 2D maps per octree node [LD07]. Volume-Encoded UV-maps automatically compute an injective mapping from 3D space to 2D space by assigning *uv* coordinates in a coarse 3D grid [Tar16]. Cuts may still occur, and seam artifacts could appear when using filtering. The technique is less general when tiny features are important, compared to the resolution of the 3D *uv* texture.

Sparse Volumetric Textures Adaptive texture maps [KE02] remove unused texture space by packing texture tiles, thereby discarding unused tiles, and adding an indirection lookup. Benson and Davis [BD02] suggest using an octree for mapping and storing of texture colors. Then, Lefebvre et al. [LHN05] show how to adapt this to the GPU. Brick maps (e.g., in Renderman) use a brick of, for instance, 8^3 sparse voxels per octree node, which also significantly reduces the depth of the tree [CB04]. It is reasonable to expect the hardware-friendly layout to enable speed optimizations we cannot use, at the cost of compression capabilities (e.g., inter-brick compression would be difficult). Perfect Spatial Hashing circumvents the expensive tree traversal by instead using a hash function [LH06]. However, the precomputation times and hash-table overheads can be substantial, and although that method and the later work by Garcia et al. [GLHL11] strive to maintain spatial coherency, their solutions do not directly lend themselves to efficient usage of compressed textures in real time.

Gigavoxels [CNLE09] target real-time voxel rendering of Sparse Voxel Octrees (SVOs) [LK10] with stackless traversal using ray casting. SVOs can be compressed both geometrically [KSA13; VMG16] and for the colors [DSKA18; DKB*16] by merging identical subgraphs into a sparse voxel DAG and adding a non-obstructive mapping from node to texel.

Compared to directly rendering a voxel representation, our rendering is performed using the artist's original triangle model. It can often be sufficient to use a significantly lower texture resolution and triangle-tessellation resolution (the latter perhaps by two orders of magnitude) than a corresponding voxel resolution would require for acceptable surface representation without apparent blockiness.

3. Recap of Sparse Voxel DAGs with Colors

A Sparse Voxel Octree (SVO) is an octree where each node represents a voxel and its color. A Sparse Voxel DAG achieves a high compression ratio by considering the geometric information only (empty or non-empty voxel) and removing duplicates of identical subgraphs recursively bottom up [KSA13].

Storing color information in the nodes would lower the probability of having identical subgraphs. Instead, per-voxel colors are supported by, for each DAG node, storing a relative color-offset value that directly corresponds to the amount of non-empty voxels in that node's subgraph (1 for a leaf). Since this information only depends on the geometry, it does not affect the compression possibilities. The colors are stored separately in a one- or two-dimensional texture [DKB*16; DSKA18].

A DAG is traversed identically to an SVO. However, by a careful summation of the color-offset values during traversal from the DAG root to a voxel location, the color index can be retrieved. The index represents a depth-first traversal order of the SVO nodes along a 3D Morton curve. This preserves significant amounts of the two-dimensional surface-color coherence, which is essential for compression. Dolonius et al. implement a custom real-time, block-compression format of the resulting one-dimensional voxel-color array (see Section 5.1-*Decoding colors*).

4. Method Overview

Voxelization Each mesh is voxelized to an individual DAG that stores the mesh colors in voxel space. We create a non-empty voxel for each voxel that intersects the mesh surface. If the surface already has an assigned texture pattern that should be transferred into the voxel colors, then we initiate the voxel color to a filtered value of the texels falling inside the voxel.

If a mesh has more than one texture, for instance representing

global-illumination information of separate frequencies and thus of different resolutions, we generate a DAG per resolution and mesh. Alternatively, we recommend using the same resolution for all textures of a mesh and relying on the texture compression being higher for data of lower frequencies.

Either way, when using the voxel structure, it would be easy to do a trilinear-filtered texture lookup at a 3D position in space if the voxel colors would be stored at the voxel corners or if at least a neighborhood of 2^3 voxel colors exist [BD02]. However, that would require storing colors not only in the geometry-containing voxels but also in many of their neighbors, which could drastically increase the memory cost. Instead, we only store a color at the center of surface-intersecting voxels and modify the logic of the real-time trilinear filtering accordingly.

Merging DAGs Next, we merge all generated DAGs into one and then compress the voxel colors using the texture compression.

Rendering During real-time rendering, we render each mesh to a shared G-buffer using hardware rasterization with the fragment shader outputting an encoded representation of the meshID, the sample's 3D position, and desired filter level. It should be noted that this information is independent of the number of global textures used and their individual texture resolutions.

Color lookups Finally, in a deferred-shading pass, filtered color values per pixel are retrieved using a DAG-traversal kernel and texture-decompression kernel implemented in CUDA. The latter kernel is run once per used global texture.

5. Implementation

Construction of data For each mesh of a scene, we create a separate DAG with its compressed one-dimensional texture and later merge all of them into one DAG. The textures are simply merged by appending them sequentially.

Double-sided surfaces with a per-face texture are treated as a mesh per face (this is also common in modeling programs). The two DAGs are then automatically collapsed into one (as described in the next paragraph), keeping just the two individual roots and thus maintaining individual voxel colors per face.

Merging DAGs of different heights When meshes have independent resolutions, their associated DAGs can have different heights. However, it is possible to merge all DAGs into one DAG by virtually aligning their leaf levels (i.e., regarding their leaf levels as being at the same depth) and accordingly letting their roots start at different heights (see Figure 2, upper). Then, level by level, the DAG nodes are merged into one new combined level by appending and then reducing them. As for the original algorithm [KSA13], the non-unique nodes have their parent pointers updated to point to the surviving node instance, thereby exploiting the added compression possibilities.

Put differently, we first concatenate all the DAGs' leaf levels into one new leaf level. Secondly, we remove identical node duplicates. Then, we repeat these two steps identically for the next level above,

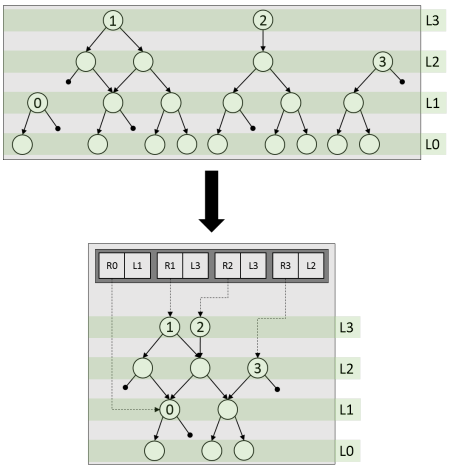


Figure 2: *Upper) Separate DAGs of different heights, i.e., corresponding to different voxel resolutions, can be merged into one combined DAG by first aligning their leaf levels and then merging all levels. Lower) We track each root's new position in the combined DAG by using an array of elements (RX, LY), where RX means aa new index for the root of DAG X and LY its new level, Y.*

thereby merging that level of all DAGs into one new concatenated level. This is repeated level by level until there are no higher levels for any of the DAGs.

Once we have aligned the depths of all DAGs' leaf levels, our approach is practically identical to the original algorithm by Kämpe et al. [KSA13]. However, we also keep an array of node indices specifying the root node for each separate mesh in the DAG. These root indices will therefore often point to nodes not residing at the root level of the resulting combined DAG (see Figure 2, lower).

Using the data During triangle rasterization, a sample's voxel-space position is used to perform the filtered color lookup. For deferred shading, we use a G-buffer to store, per pixel, the index to the mesh's root-node, the voxel-space coordinate, and the desired filter level. We use a 32-bit float per x, y, z to describe the coordinate, i.e., typically with much higher precision than the voxel resolution, in order to represent a sub-voxel location for the sample of the surface. This index and position is later used in a CUDA kernel that traverses the DAG from the root specified by the root index and returns the desired color index given the path and mipmap level (see Section 5.1). If G-buffer size is an issue, there is room for optimizing the per-pixel information, but we have left that out.

Minification Colors are stored in all nodes of the DAG. A parent's color is the average of its children. We sample the level above and below our desired mip level and do an interpolation on the fractional part. The mip level, m, is calculated, from the screenspace derivatives of the position, here called $\frac{\partial F}{\partial x}$ and $\frac{\partial F}{\partial y}$, as:

$$m = \log_2(\max\left(\left\|\frac{\partial F}{\partial x}\right\|, \left\|\frac{\partial F}{\partial y}\right\|\right))$$

For the analogy to standard trilinear mipmap-filtered texture lookups, we have to perform quad-linear filtering, since we do the

filtering in 3D voxel space and not 2D texture space. For each of the two mipmap levels closest to m, we perform trilinear filtering of the existing colors in the surrounding block of 2^3 voxels. Then, we interpolate between the two resulting colors, based on the fractions of m and $1-m$. Since we access up to 2^3 colors for both mipmap levels, this results in up to 16 texel accesses. However, we optimize these accesses heavily.

The implementation will be detailed in Section 5.1. There, we will also describe a faster, approximate option based on multisampling that only performs four texel accesses per fragment on average. It is beyond our scope here to seek the best multisampling filter, since that depends on a huge number of factors and is a vast topic [MN88; MP11]. Instead, the intention is to present one filter that we empirically found to work well for low-frequency textures, demonstrating that higher performance than for full tri-linear mipmap filtering can be achieved and with satisfying quality.

Magnification When $m \leq 0$, we do a trilinear-filtered lookup using only mipmap level zero. This requires up to 8 voxel-color accesses, i.e., half the cost of filtering between two mipmap levels.

5.1. CUDA kernels

A naive tri-linear filtering for one mipmap level would perform eight independent lookups from which a weighted average is calculated. We propose two methods to accelerate this computation. The first method utilizes caching to avoid unnecessary traversals and global memory reads. The second is an approximate solution where we multi sample in screen space in order to reduce the number of required samples. While the second one shows some aliasing artifacts for high-frequency textures, there is little to no artifacts for lower frequencies such as e.g., light maps.

Our trilinear sampling is composed of two parts. First, we calculate an array of up to eight color indices using a CUDA kernel. Secondly, this index array is used by a separate kernel to decode the colors and compose the final color by a weighted average.

To achieve mip mapping, the kernels are run twice, once for each relevant mip-map level. While it is indeed possible to do all sixteen lookups in one kernel, we still resort to two kernel calls in order to reduce register pressure to achieve maximum occupancy.

Color index lookup For a filtered sample in a voxel block of size 2^3, we need a fast way to find the ≤ 8 voxel-color indices. In our implementation, the DAG's voxels are stored in a Morton order where x is the most and z the least significant direction ($z \rightarrow y \rightarrow x$). Given a sample, s, with its path, the voxel position of the bottom, lower, left corner of the block is given as $p = \lfloor s/2^L - 0.5 \rfloor$, where $L \geq 0$ is the desired mipmap level (i.e., DAG level), counting the leaf level as level zero.

The important issue is to minimize the DAG traversal and maximize cache coherence by visiting the eight voxels along their global Morton order. The voxels' Morton codes are trivially found by bit swizzling their x, y, z coordinates. Therefore, it would be possible to sort the eight voxels on their Morton codes and visit them in that order. However, the following approach is faster, with identical result, where we utilize that the sorted order can be found by only sorting three values based on properties of the voxel position p.

To maximize subgraph coherence, when visiting the eight voxels, we want to traverse any immediate DAG siblings before voxels belonging to another DAG parent, in order to avoid traversal from the root in favor of the closest common parent. In our block of 2^3 voxels, any two voxel neighbors in direction i, where i is x, y, or z, will share parent at some specific level, l_i. Hence, we can use a traversal stack of only three elements, each storing a parent-node index.

Additionally, if a, b, c is some permutation of the step dimensions x, y, z, the Morton-visiting order of the eight voxels can be written as $[(p), (p+\hat{a}), (p+\hat{b}), (p+\hat{a}+\hat{b}), (p+\hat{c}), (p+\hat{c}+\hat{a}), (p+\hat{c}+\hat{b}), (p+\hat{c}+\hat{a}+\hat{b})]$, where $\hat{a}, \hat{b}, \hat{c}$ are the unit vectors for a, b, c. We will now explain how to compute the correct permutation order, a, b, c for the block.

We can sort p's x, y, and z coordinates based on their first-bit cleared, fbc(), i.e., first zero-bit position from the least-significant bit. In other words, we sort on how much a direction increases the position along the Morton curve. The sort can be done in only three compare-and-swap instructions, implementing a small stable sort that utilizes that, in the case of equality, a step in z is smaller than a step in y, which is smaller than a step in x, according to the Morton order we use. This will result in some order $\text{fbc}(a') < \text{fbc}(b') < \text{fbc}(c')$, where a', b', c' are the values of p's x, y, z coordinates. This directly gives the requested permutation a, b, c of x, y, z.

Since we traverse directly neighbouring voxels in a sorted order, we also know in which order to read the stack. If we define the stack elements as e_a, e_b, e_c where the subscripts correlate to the ordering of the permutation, i.e., e_a for the smallest change and e_c for the largest, then we read the stack elements in order $\{e_a, e_b, e_a, e_c, e_a, e_b, e_a\}$. Finally, we can compute the parents' level offsets w.r.t. the sample level as $o_d = \text{fbc}(p_d) + 1, d \in \{a, b, c\}$ (Figure 3). Thus, we know exactly at what level to update each stack element.

During the traversal and at each level, we check if we should cache the current node index (parent) and the current color index, for each direction. When traversal has finished for a voxel position, we write its color index to texture memory if there exists geometry for that voxel, and a no-index sentinel otherwise. In case of the latter, the index caching during the traversal may still be beneficial for the next voxel. For compression efficiency, DAGs often use leaf nodes of $(4x4x4)$ voxels [KSA13]. If we traverse to the bottom of the DAG, the next voxel of our 2^3 block would likely fall in the same $(4x4x4)$ leaf. Thus, we also cache the 64-bit leaf mask in order to avoid unnecessary reads from global memory.

Decoding colors Our decoding algorithm is, in essence, the same as for one texel access at a time [DSKA18] but optimized for a Morton-ordered sequence of texel indices (e.g., eight). For the convenience of the reader, the next paragraph presents a summary of the original algorithm to retrieve the color for one texel.

All texels are stored in a one-dimensional sequence along a three-dimensional Morton order in voxel space and are then compressed by a custom block-compression method, as follows. The texels are sequentially split into compressed *macro blocks* of 16K texels, where each macro block is composed of smaller *blocks*. The macro block is composed of an index to the first block within its

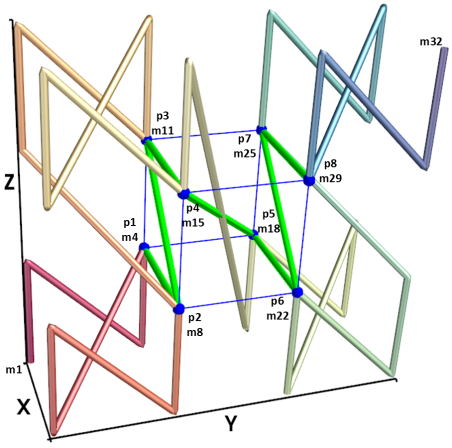


Figure 3: *Coherent sample lookups - finding the step order (green) of eight sample indices (blue), $p1,...,p8$, along a global Morton order ($z \rightarrow y \rightarrow x$). Assume the global index for p is (0,1,1). Then, $p1 = (0,1,1)$ and $p8 = (1,2,2)$. Since $fbc(p1_x) < fbc(p1_z) \leq fbc(p1_y)$, we have a sample ordering as ($x \rightarrow z \rightarrow y$). As $fbc(p1_x)$= 0, we also know that the shared parent for p1 → p2, p3 → p4, p5 → p6, and p7 → p8 is one level above; i.e., for p1 → p2, it is the parent with children m1,...,m8. Similarly, since $fbc(p1_z)$= 1, we know that the shared parent for p2 → p3 and p6 → p7 is two levels above. The same holds for p4 → p5, as $fbc(p1_y)$= 1.*

16K range and an index to a global array of per-texel weights. Each block, within the macro block, consists of a header specifying two colors used for interpolation, a texel-index offset representing the block's start-texel index in the macro block, a weight-index offset, and the number of bits per weight for that block (which may be zero). The algorithm itself performs a binary search on the blocks using the texel-index offset to find the block responsible for the texel. Finally, weights and end-point colors are extracted and are used to interpolate the final color.

Here follows the description of our new optimization. From the color-index-lookup step, we have received eight Morton-ordered voxel samples, as voxel locations and their texel indices, and as such the texel indices are ordered as well. Invalid samples (i.e., for empty voxels) are already marked with a no-index sentinel. We have also cached, in texture memory, the voxel-sample order and the fractional part of the surface-sample path (i.e., for the surface position of which we are computing a trilinear-filtered color). These will be used for the trilinear weighting of the voxel samples when their colors have been decoded.

A block generally spans a range of texels. Thus, we can expect several of our desired samples to share block, especially at lower mip-map levels. Therefore, when we decode a voxel sample, we also cache all of the current block header (colors and metadata) and the metadata of next block headers to avoid reads from global memory. Since the texel indices are sorted, we can first check the next sample's index against the index stored in the next cached block header. If it is lower, the texel index belongs to the same block and no binary search is required. If the texel index does not belong to the same block but is still contained in the same macro block, then we can at least cache the lower bound of the binary search in order

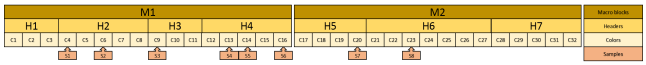


Figure 4: *Caching block headers during sample lookups. In this example, the first sample (S1) of up to 8 samples (S1-S8) is found by a binary search in macro block M1 and thus between the blocks specified by header H1 and H4. Then, to find S2, no new binary search is necessary, since S2 is in the same block as S1. To find S3 only a binary search between H3 and H4 is needed, as S2 is the latest sample. Eventually, to find S7 that is located in another macro block (M2), a new binary search (between H5 and H7) is needed.*

to save some iterations for the next voxel sample. Again, this is also possible since the indices are sorted (see Figure 4 for an example).

We decode and weight the data for the first decoding pass by using the before-mentioned cached data. Similarly to bilinear filtering, the trilinear filtering calculates a weighted average of the eight neighboring voxels. Our weight for a sample is $w_i = (1 - d_{i_x}) * (1 - d_{i_y}) * (1 - d_{i_z})$, where d_i is the component-wise distance from the voxel center to the sample position, which is trivially derived from the fractional part of the original sample position. Since not all samples are valid, they can be discarded when calculating the weight. I.e., for the eight samples with colors, $C_i, i \in [0,7]$, and weighs, w_i, where χ_i is 1 for valid samples and 0 otherwise, we have the trilinear sample, S, as:

$$S = \frac{\sum \chi_i C_i w_i}{\sum \chi_i w_i}$$

In the second pass, for the next mipmap level, we also read back the result of the first pass to perform the final mipmap weighting.

Approximative multisampling Quad-linear mipmap filtering is not always necessary for smooth and flicker-free filtering, depending on the maximum frequency present in the texture. Arguably, for smooth-enough textures, even a nearest neighbor filter will suffice. To demonstrate that higher performance with maintained quality often can be achieved, e.g,. for scenarios such as baked light maps, we will here present a filter based on multi sampling, which we found to work well, quality-wise, for our test scenes while doubling the texture-lookup performance.

We can speed up the minification filter by distributing the texture samples according to the classic screen-space pattern shown in Figure 5. This can also be seen as a shifted Quincunx [01] pattern and bears resemblance to Flipquad [Ake02]. This allows a pixel to effectively retrieve four filtered texture samples at the cost of two. Additionally, for each filtered texture sample, we use linear filtering between the two nearest mipmap levels and nearest-neighbor filtering within each level. The color-index-lookup kernel is launched once for each pixel and processes two sample positions and both mipmap levels, i.e., it processes four texel accesses in total. Since the samples are now in screen space, the two mipmap levels of a sample correspond to a child and its direct parent. Hence, for one such path, no stack is required.

We only use the approximative multisampling for minification. For magnification, we use trilinear filtering at mip level 0, since that is both fast and correct. As minification often will query nodes

higher up in the DAG, the stack will be less useful. Therefore, we drop it completely and start from the root for both sample positions, in favour of letting the compiler allocate the registers more freely elsewhere.

In order to maximize cache coherency, we want to perform the sample lookups along the Morton order (just as for quadlinear filtering kernel). This can be achieved by presorting the indices or processing them in ascending order, where we choose the latter. For the final weighting, we perform a gather pass where the samples are weighted by the distance from their voxel center to the surface-sample position (see Figure 5).

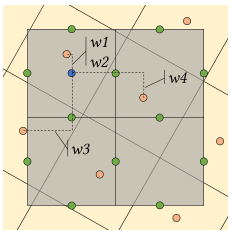


Figure 5: *Our multisampling scheme for custom minification filtering. Gray boxes are pixels, the blue dot is the pixel center (sample position), green dots are multi-sample positions, yellow boxes are voxels, and orange dots are voxel centers.*

6. Results

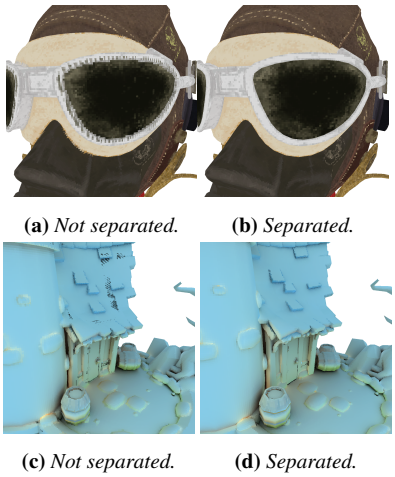
We present the results for three scenes, highlighting different properties of our algorithm. First, we have the FLIGHTHELMET scene, where three textures are used: albedo, normals, and a texture with occlusion, roughness, and metallic packed into three components, to show that we can achieve a similar resolution as standard texture maps with a lower memory overhead, as there is no wasted texture space.

We also show that color compression can be improved by partitioning the mesh into separate sets since that will present a higher coherency of similar attributes. Furthermore, we show that such partitions can avoid the color-bleeding artifacts for large filter sizes. As the artist usually already has a model consisting of different parts, the only overhead is specifying the faces of different parts where we want to avoid color bleeding, e.g., for nearby disjoint surface parts, such as for the glasses and straps (see Figure 6).

Alternatively, and contrary to partitioning meshes to avoid color bleeding, the visibility of seams between intersecting geometry can be reduced by letting different meshes share the same mesh partition (see Figure 7).

Figure 8 compares nearest and linear filtering modes for regular textures and DAGs at a similar resolution. Figure 9 compares the frame-to-frame flickering between multisampling and quadlinear filtering. The straps in the textscFlightHelmet with a regular high-frequency pattern cause aliasing for multisampling, especially when the pattern is aligned diagonally to the samples. (As a trade-off, it is possible to remove the visible artifacts for multisampling by biasing the mip-map level one level up at the cost of over blur.) For the low-frequency regions, the flickering is undetectable.

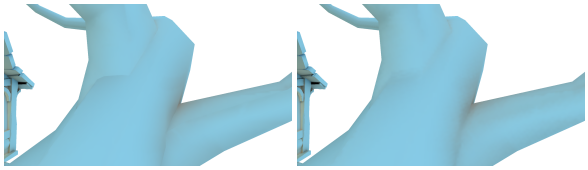
In the second scene, SKYHOUSE, we have baked irradiance to show that the multi-sampling scheme is highly suitable for such scenes. The low-frequency information also lends itself well to the texture compression. The third scene, THEGREATDRAWINGROOM, is a scanned scene with one large high-frequency texture, which is problematic to *uv* unwrap.



(a) *Not separated.* **(b)** *Separated.*

(c) *Not separated.* **(d)** *Separated.*

Figure 6: *Benefits of separating meshes. Top row is* FLIGHTHELMET *and bottom is* SKYHOUSE. *As can be seen in (a) and (c) the underlying geometry causes unwanted bleeding, e.g., in (a) where we have the dark colored glass close to the white rim, or in (c) where parts on the interior with no irradiance causes color bleeding, which we see from an interior supporting beam being too close to the exterior roof.*



(a) *Texture - Geometry seam.* **(b)** *DAG - Seamless.*

Figure 7: *By merging separate meshes into the same mesh partition, the filtering in voxel space can reduce undesired seams compared to pure filtering along surface space.*

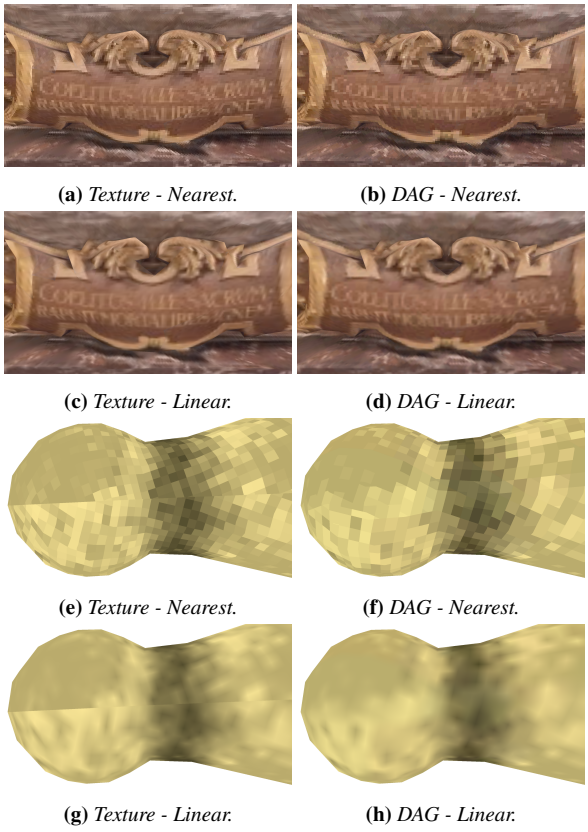
6.1. Compression

Depending on the model, storing one DAG for the entire mesh can require more memory than storing *uv* coordinates at its vertices; on the other hand, using a DAG no memory is wasted to store unused texels in the texture image. Also, the number of *uv*-coordinates depends on the resolution of the mesh (for example, it roughly quadruples if the mesh undergoes a subdivision step), whereas the DAG size depends only on the texture resolution, and is usually relatively small (especially for planar surfaces).

In Table 1, we compare the total size of DAG + colors against standard textures + *uv* attributes. For comparison purposes, this is done without texture compression for any of the methods. All texels are of roughly the same projected size, and the meshes are voxelized such that the average distance in texture space is the same in voxel space, i.e., the original texture resolution is roughly maintained. We also compare against a virtual, perfect *uv* mapping without wasted texels between *uv* patches. I.e., all unused texels are discarded in that comparison (see Table 1 - Used texels). Without texture compression, we see that the DAG + colors are around $0.7\times$ smaller than the textures and around $1.5\times$ larger if we had a perfect uv map.

	FLIGHTHELMET	SKYHOUSE	THEGREAT DRAWINGROOM
	Mb	Mb	Mb
DAG	6.4	5.1	18.2
UV	0.4	0.2	4.0
Factor	16.8×	23.7×	4.6×
	Mb (#elements)	Mb (#elements)	Mb (#elements)
Voxels	68.7 (22.9M)	48.7 (16.2M)	177.1 (59.0M)
Texture / Used texels	122.7 (40.9M) / 51.5 (17.2M)	67.1 (22.4M) / 36.3 (12.1M)	268.4 (89.5M) / 129.3 (43.1M)
Factor	0.6× / 1.3×	0.7× / 1.3×	0.7× / 1.4×
Total	0.6× / 1.4×	0.8× / 1.5×	0.7× / 1.5×

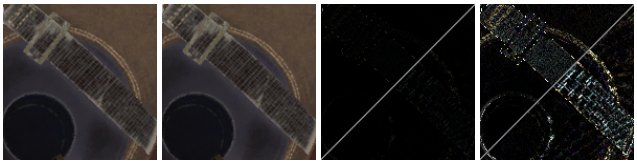
Table 1: *Comparing memory consumption of DAG + Voxels (colors) against UV + textures, without texture compression for either method. Factor corresponds to DAG/UV and Voxels/Texels, respectively. Total represents (DAG+Voxels)/(UV + Texels).*



(a) *Texture - Nearest.* **(b)** *DAG - Nearest.*
(c) *Texture - Linear.* **(d)** *DAG - Linear.*
(e) *Texture - Nearest.* **(f)** *DAG - Nearest.*
(g) *Texture - Linear.* **(h)** *DAG - Linear.*

Figure 8: *Comparisons of nearest and linear filtering modes for regular textures and DAG. The nearest filter selects the closest mipmap and nearest texel within.*

Merging DAGs We have shown how to represent individual DAGs by one common DAG. In Table 2, we compare sizes and compression ratio between voxelizing the full scene as one global mesh in one DAG vs. voxelizing separate meshes into DAGs and then merging them into one. We chose to present the data for the irradiance texture of SKYHOUSE, representing a low-frequency texture,



(a) *Frame 1.* **(b)** *Frame 2.* **(c)** *Diff.* **(d)** *Diff* ×10.

Figure 9: *Evaluation of multisampling flickering during motion for high- and low-frequency regions. (a) Frame 1 with multisampling. (b) Frame 2, with the multisampling offset such that the screen samples land 0.5 pixels from frame 1. (c) bottom right: Difference of frame 1 and 2 aligned to demonstrate scale of flickering at high-frequency regions and no detectable flickering at low-frequency regions. (c) top left: Diff. of the same but using quad-linear filtering. (d)* 10× *amplified difference.*

and the albedo texture of FLIGHTHELMET, representing a high-frequency texture. As THEGREATDRAWINGROOM is a scanned scene, the mesh proved cumbersome to manually separate in a reasonable way and, therefore, is left out in this comparison.

We want to highlight that, while we gain $10-15$ percent of DAG compression by merging compared to not merging, the resulting sizes also become close to as if we had never separated the meshes in the first place. More noteworthy is the fact that by separating meshes, there is also a positive impact on color compression by $12-32$ percent, which is a bonus besides avoiding color bleeding.

6.2. Timings

To illustrate the benefit of the optimizations, Table 3 presents timings for the three different test scenes, comparing our quad-linear mipmap filtering and multi-sampling scheme vs. nearest-neighbor filtering. The nearest-neighbor filter selects the closest mipmap and nearest texel within. We use an RTX2080 for all the tests, at a resolution of 1920×1080.

Quad-linear timings Our experiments indicate that for the *index computation*, our optimized tri-linear version is approximately only 3 times slower than for nearest sampling, whereas a naive implementation, requiring 8 independent lookups, would be expected to

			Size in Mb DAG before / after merge	Size in millions Number of colors	Size in Mb Raw / compressed colors	Compression % of original size
FLIGHTHELMET	512^3	One DAG	0.5	1.5	4.5 / 0.9	20.3
		Separate DAGs	0.66 / 0.60	1.6	4.7 / 0.7	15.4
	1024^3	One DAG	1.9	6.2	18.5 / 2.9	15.8
		Separate DAGs	2.3 / 2.1	6.4	19.2 / 2.5	12.9
	2048^3	One DAG	6.7	25.0	75.1 / 8.9	11.9
		Separate DAGs	7.7 / 6.9	25.6	76.7 / 7.9	10.3
	4096^3	One DAG	23.0	101.2	303.5 / 25.2	8.3
		Separate DAGs	25.3 / 23.2	102.2	306.5 / 22.6	7.4
SKYHOUSE	512^3	One DAG	0.54	1.2	3.7 / 0.8	21.3
		Separate DAGs	0.74 / 0.64	1.3	4.0 / 0.7	17.1
	1024^3	One DAG	1.8	5.0	15.0 / 2.0	13.0
		Separate DAGs	2.4 / 2.0	5.2	15.8 / 1.7	10.7
	2048^3	One DAG	6.2	20.4	61.1 / 4.8	7.9
		Separate DAGs	7.6 / 6.3	20.8	62.3 / 4.1	6.6
	4096^3	One DAG	19.8	82.0	246.0 / 11.5	4.67
		Separate DAGs	22.7 / 19.6	82.8	248.5 / 9.7	3.9

Table 2: *In the* FLIGHTHELMET *scene, the high-frequency albedo texture is used, and in* SKYHOUSE*, we use the low-frequency irradiance texture. Both are compressed using an error threshold of 6.4 for color ranges in* $[0,255]$.

be 8 times slower. The *color decoding* of eight samples is around $3-4\times$ slower than one, instead of 8. In total, for the two trilinear mipmap samples, we see that together they are only $6-7\times$ slower than one nearest-neighbor texel lookup instead of 16 times.

Multisampling timings One multi-sampled fragment requires the calculation of four texel samples (two per pixel times two mipmap levels) for the eight weighted samples per pixel (four per pixel times two mipmap levels), and are meant to approximate the 16 samples needed for a quad-linear mipmap-filtered sample. For the sample-index lookup, the two mipmap levels correspond to a child and its immediate parent, which can be fetched in one traversal. Together with our caching, the sample-index lookup for 4 such indices is only about $1.5-2\times$ slower than one nearest sample (i.e., $2-2.7\times$ faster per sample). Computing only 4 samples per pixel instead of 16 results in a total index-computation speedup of $8-10\times$.

There are also fewer samples to decode (4 compared to 16 for quad-linear filtering). The timings in Table 3 show that these four are only $2.5-3\times$ slower than one nearest-neighbor sample, i.e., $5.3-6.4\times$ faster than 16 nearest samples.

In total, combined with trilinear sampling for magnification, we see in Table 3 that multisampling is $2.5-3.5\times$ slower than 1 nearest sample, i.e., $4.5-6.4\times$ faster than 16 nearest samples. For all our timings, the numbers are mostly bound by global-memory reads.

Multiple textures sharing voxelization The major cost is the DAG traversal to retrieve the color indices; the decoding step is cheap (see Table 3). Note that FLIGHTHELMET uses three separate quad-linear-filtered textures in < 5 ms in total. The decoding step is optional when compression is not needed, in which case the only overhead would be reading the data directly from memory and weighting the samples, thereby pushing the numbers further to our favour. However, we do not exploit this opportunity in the reported numbers.

7. Discussion

Filtering at high mipmap levels For traditional texture maps, color bleeding problems can occur when the filter includes texels from disjoint texture patches. In our case, color bleeding occurs whenever two disconnected parts of the surface happen to share one voxel.

Despite a separate voxelization per mesh, errors could still occur internally for those meshes, but the problem is reduced since individual meshes often contain more surface regularity than a whole object and such a mesh perhaps also shares a similar continuous texture pattern over its surface.

Another problem is that, under extreme minification filters, when the filter size is comparable to the mesh size, the color of back-facing parts of the surface, which are occluded, will wrongly contribute to the averaged color of the visible parts. This minor limitation is shared by most other texture mapping approaches, including the standard one.

Splitting a mesh into sub-meshes to avoid color bleeding in one region will disable filtering over the new submesh borders. This is often undesirable. We believe that these seams can be automatically removed in a preprocess of the DAG and mipmap generation, using some memory overhead but without render-time overhead. However, this remains as future work.

Other Limitations Our method is significantly more computationally heavy compared to traditional texture mapping. A typical application is to store post-modeling information, such as automatically computed light maps or other global-illumination information. The texture compression can be attractive for material textures. Either way, since we use a DAG root per mesh, creating a DAG from a selected combination of DAG-converted meshes is done by essentially appending their node levels. The inter-DAG compression is optional, with speed limited by the GPU radix sorting (in the order of 1B nodes/s). The color-index information and the compressed per-mesh textures are unaffected during this process. While not being part of our contributions, the texture com-

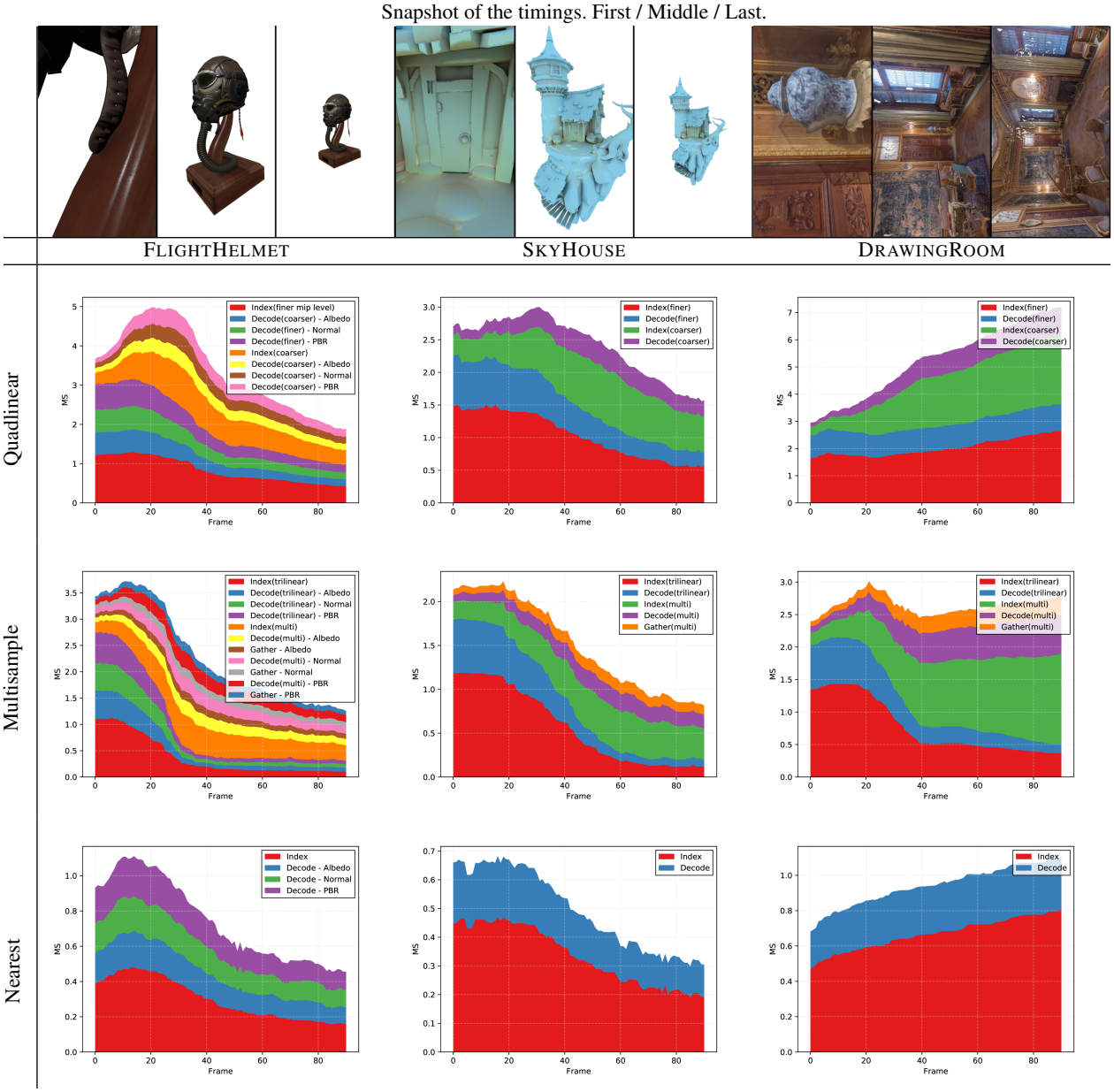


Table 3: *Timings for different scenes. For the* FLIGHTHELMET *scene, we decode three textures: RGB albedo, RG normals, and an RGB PBR material storing occlusion, roughness, and metallic. The* SKYHOUSE *scene decodes one RGB texture of baked irradiance. The* THEGREAT-DRAWINGROOM *scene decodes scanned RGB data in one texture.*

pression takes in the order of a few seconds for our non-optimized single-threaded CPU version. Voxelization and DAG compression takes about 1 ms per 1M voxels and thus in the order of tens of milliseconds for our reported scenes and resolutions.

Voxel-based methods such as ours do not target texture repetition. Tangent-space normal maps require information for tangent vectors. One method is to store tangents in the G-buffer, using the original (non-injective) *uv* map that typically is present if a tangent normal map is present.

Anisotropic Filtering We have not explored anisotropic filtering [MSY19]. Our multisampling scheme, however, uses four independent mipmap-filtered texture lookups per pixel, each with linear filtering between the two nearest mipmap levels and nearest-neighbor filtering within each level. While this could be easily extended to more samples, each of up to quadlinear filtering, efficient full 16× anisotropic filtering that does not just linearly increase costs requires further research.

8. Conclusions

We have demonstrated how to achieve *uv*-free texturing using sparse voxel DAGs with compressed textures. This comes at the cost of not being able to use fully hardware-supported filtering. We show how to optimize magnification and minification filtering 2.5 ± 0.2 times for quad-linear mipmap filtering, e.g., achieving 1-3 texture loopkups per pixel in full HD at 5 ± 2 ms. With our custom multisampling filter (for low-frequency textures), the cost is only 2.5 ± 0.5 ms.

We also explained how to extend the DAGs to handle different resolutions at separate regions by storing several individual DAGs of different heights in one combined DAG, while maintaining full compression capabilities. On top, it lowers problems with large filter widths and improves the texture-compression ratio by up to 32 percent for our test scenes.

9. Acknowledgments

This work was supported by the Swedish Research Councilunder Grant 2014-4559. The FLIGHTHELMET scene is distributed in the glTF Sample Models, from the Khronos Group, donated by Microsoft. The GREATDRAWINGROOM scene is made by the Hallwyl Museum. The SKYHOUSE scene is made by Sander Vander Meiren, with the original name *stylised sky player home dioroma*.

References

[01] *HRAA: High-Resolution Antialiasing Through Multisampling*. Technical brief. NVIDIA Corp. 2001 6.

[Ake02] AKENINE-MÖLLER, TOMAS. "FLIPQUAD: Low-Cost Multisampling Rasterization". *Technical Report 02-04*. Chalmers University of Technology, Apr. 2002 6.

[BD02] BENSON, DAVID and DAVIS, JOEL. "Octree Textures". *ACM Trans. Graph.* 21.3 (July 2002), 785–790. ISSN: 0730-0301. DOI: 10.1145/566654.566652. URL: http://doi.acm.org/10.1145/566654.566652 2–4.

[BL08] BURLEY, BRENT and LACEWELL, DYLAN. "Ptex: Per-face Texture Mapping for Production Rendering". *Proceedings of the Nineteenth Eurographics Conference on Rendering*. EGSR '08. Sarajevo, Bosnia and Herzegovina: Eurographics Association, 2008, 1155–1164. DOI: 10.1111/j.1467-8659.2008.01253.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2008.01253.x 3.

[CB04] CHRISTENSEN, PER H. and BATALI, DANA. "An Irradiance Atlas for Global Illumination in Complex Production Scenes". *Eurographics Workshop on Rendering*. Ed. by KELLER, ALEXANDER and JENSEN, HENRIK WANN. The Eurographics Association, 2004. ISBN: 3-905673-12-6. DOI: 10.2312/EGWR/EGSR04/133-141 3.

[CNLE09] CRASSIN, CYRIL, NEYRET, FABRICE, LEFEBVRE, SYLVAIN, and EISEMANN, ELMAR. "GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering". *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. Boston, Massachusetts: ACM, 2009, 15–22. ISBN: 978-1-60558-429-4. DOI: 10.1145/1507149.1507152. URL: http://doi.acm.org/10.1145/1507149.1507152 2, 3.

[DKB*16] DADO, BAS, KOL, TIMOTHY R., BAUSZAT, PABLO, et al. "Geometry and Attribute Compression for Voxel Scenes". *Computer Graphics Forum* 35.2 (2016), 397–407. DOI: 10.1111/cgf.12841. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12841. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12841 2, 3.

[DMA02] DESBRUN, MATHIEU, MEYER, MARK, and ALLIEZ, PIERRE. "Intrinsic Parameterizations of Surface Meshes". *Computer Graphics Forum* (2002). ISSN: 1467-8659. DOI: 10.1111/1467-8659.00580 2.

[DSKA18] DOLONIUS, DAN, SINTORN, ERIK, KÄMPE, VIKTOR, and ASSARSSON, ULF. "Compressing Color Data for Voxelized Surface Geometry". *IEEE Transactions on Visualization and Computer Graphics* 25.2 (2018), 1270–1282. ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2741480 2, 3, 5.

[GLHL11] GARCÍA, ISMAEL, LEFEBVRE, SYLVAIN, HORNUS, SAMUEL, and LASRAM, ANASS. "Coherent Parallel Hashing". *ACM Trans. Graph.* 30.6 (Dec. 2011), 161:1–161:8. ISSN: 0730-0301. DOI: 10.1145/2070781.2024195. URL: http://doi.acm.org/10.1145/2070781.2024195 3.

[HPS08] HORMANN, KAI, POLTHIER, KONRAD, and SHEFFER, ALIA. "Mesh Parameterization: Theory and Practice". *ACM SIGGRAPH ASIA 2008 Courses*. SIGGRAPH Asia '08. Singapore: ACM, 2008, 12:1–12:87. DOI: 10.1145/1508044.1508091. URL: http://doi.acm.org/10.1145/1508044.1508091 2.

[KE02] KRAUS, MARTIN and ERTL, THOMAS. "Adaptive Texture Maps". *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02. Saarbrucken, Germany: Eurographics Association, 2002, 7–15. ISBN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569046.569048 3.

[KLS03] KHODAKOVSKY, ANDREI, LITKE, NATHAN, and SCHRÖDER, PETER. "Globally Smooth Parameterizations with Low Distortion". *ACM Trans. Graph.* 22.3 (July 2003), 350–357. ISSN: 0730-0301. DOI: 10.1145/882262.882275. URL: http://doi.acm.org/10.1145/882262.882275 2.

[KSA13] KÄMPE, VIKTOR, SINTORN, ERIK, and ASSARSSON, ULF. "High Resolution Sparse Voxel DAGs". *ACM Transactions on Graphics* 32.4 (July 7, 2013). SIGGRAPH 2013 2–5.

[LD07] LEFEBVRE, SYLVAIN and DACHSBACHER, CARSTEN. "TileTrees". *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM SIGGRAPH. ACM Press, 2007. URL: http://www-sop.inria.fr/reves/Basilic/2007/LD07 3.

[LFJG17] LIU, SONGRUN, FERGUSON, ZACHARY, JACOBSON, ALEC, and GINGOLD, YOTAM. "Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution". *ACM Transactions on Graphics (TOG)* 36.6 (Nov. 2017), 216:1–216:15. ISSN: 0730-0301. DOI: 10.1145/3130800.3130897 3.

[LH06] LEFEBVRE, SYLVAIN and HOPPE, HUGUES. "Perfect Spatial Hashing". *ACM Trans. Graph.* 25.3 (July 2006), 579–588. ISSN: 0730-0301. DOI: 10.1145/1141911.1141926. URL: http://doi.acm.org/10.1145/1141911.1141926 2, 3.

[LHN05] LEFEBVRE, SYLVAIN, HORNUS, SAMUEL, and NEYRET, FABRICE. "GPU Gems 2". 2005. Chap. Octree textures on the GPU, 595–613 3.

[LK10] LAINE, SAMULI and KARRAS, TERO. "Efficient Sparse Voxel Octrees". *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: ACM, 2010, 55–63. ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730814. URL: http://doi.acm.org/10.1145/1730804.1730814 2, 3.

[LPRM02] LÉVY, BRUNO, PETITJEAN, SYLVAIN, RAY, NICOLAS, and MAILLOT, JÉROME. "Least Squares Conformal Maps for Automatic Texture Atlas Generation". *ACM Trans. Graph.* 21.3 (July 2002), 362–371. ISSN: 0730-0301. DOI: 10.1145/566654.566590. URL: http://doi.acm.org/10.1145/566654.566590 2.

[MB11] MCDONALD JR, JOHN and BURLEY, BRENT. "Per-face Texture Mapping for Real-time Rendering". *ACM SIGGRAPH 2011 Talks*. SIGGRAPH '11. Vancouver, British Columbia, Canada: ACM, 2011, 10:1–10:1. ISBN: 978-1-4503-0974-5. DOI: 10.1145/2037826.2037840. URL: http://doi.acm.org/10.1145/2037826.2037840 3.

[McD13] McDonald, John. *Eliminating Texture Waste: Borderless Ptex*. GDC Talk, 2013. Aug. 2013 3.

[MN88] Mitchell, Don P. and Netravali, Arun N. "Reconstruction Filters in Computer-graphics". *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: ACM, 1988, 221–228. ISBN: 0-89791-275-6. DOI: 10.1145/54852.378514. URL: http://doi.acm.org/10.1145/54852.378514 5.

[MP11] Mavridis, Pavlos and Papaioannou, Georgios. "High Quality Elliptical Texture Filtering on GPU". *Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: ACM, 2011, 23–30. ISBN: 978-1-4503-0565-5. DOI: 10.1145/1944745.1944749. URL: http://doi.acm.org/10.1145/1944745.1944749 5.

[MSY19] Mallett, Ian, Seiler, Larry, and Yuksel, Cem. "Patch Textures: Hardware Implementation of Mesh Colors". *High-Performance Graphics (HPG 2019)*. to appear. Strasbourg, France: The Eurographics Association, 2019. ISBN: 978-3-03868-092-5. DOI: 10.2312/hpg.20191194 3, 10.

[PCK04] Purnomo, Budirijanto, Cohen, Jonathan D., and Kumar, Subodh. "Seamless Texture Atlases". *Symposium on Geometry Processing*. Ed. by Scopigno, Roberto and Zorin, Denis. The Eurographics Association, 2004. ISBN: 3-905673-13-4. DOI: 10.2312/SGP/SGP04/067-076 3.

[PTH*17] Poranne, Roi, Tarini, Marco, Huber, Sandro, et al. "Autocuts: Simultaneous Distortion and Cut Optimization for UV Mapping". *ACM Trans. Graph.* 36.6 (Nov. 2017), 215:1–215:11. ISSN: 0730-0301. DOI: 10.1145/3130800.3130845. URL: http://doi.acm.org/10.1145/3130800.3130845 2.

[RNLL10] Ray, Nicolas, Nivoliers, Vincent, Lefebvre, Sylvain, and Lévy, Bruno. "Invisible Seams". *Proceedings of the 21st Eurographics Conference on Rendering*. EGSR'10. Saarbrücken, Germany: Eurographics Association, 2010, 1489–1496. DOI: 10.1111/j.1467-8659.2010.01746.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01746.x 3.

[SCGL02] Sorkine, Olga, Cohen-Or, Daniel, Goldenthal, Rony, and Lischinski, Dani. "Bounded-distortion Piecewise Mesh Parameterization". *Proceedings of the Conference on Visualization '02*. VIS '02. Boston, Massachusetts: IEEE Computer Society, 2002, 355–362. ISBN: 0-7803-7498-3. URL: http://dl.acm.org/citation.cfm?id=602099.602154 2.

[SLMB05] Sheffer, Alla, Lévy, Bruno, Mogilnitsky, Maxim, and Bogomyakov, Alexander. "ABF++: Fast and Robust Angle Based Flattening". *ACM Trans. Graph.* 24.2 (Apr. 2005), 311–330. ISSN: 0730-0301. DOI: 10.1145/1061347.1061354. URL: http://doi.acm.org/10.1145/1061347.1061354 2.

[SPGT18] Schertler, Nico, Panozzo, Daniele, Gumhold, Stefan, and Tarini, Marco. "Generalized Motorcycle Graphs for Imperfect Quad-dominant Meshes". *ACM Trans. Graph.* 37.4 (July 2018), 155:1–155:16. ISSN: 0730-0301. DOI: 10.1145/3197517.3201389. URL: http://doi.acm.org/10.1145/3197517.3201389 2.

[SPR06] Sheffer, Alla, Praun, Emil, and Rose, Kenneth. "Mesh Parameterization Methods and Their Applications". *Found. Trends. Comput. Graph. Vis.* 2.2 (Jan. 2006), 105–171. ISSN: 1572-2740. DOI: 10.1561/0600000011. URL: http://dx.doi.org/10.1561/0600000011 2.

[SWG*03] Sander, P. V., Wood, Z. J., Gortler, S. J., et al. "Multi-chart Geometry Images". *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP '03. Aachen, Germany: Eurographics Association, 2003, 146–155. ISBN: 1-58113-687-0. URL: http://dl.acm.org/citation.cfm?id=882370.882390 2.

[Tar12] Tarini, Marco. "Cylindrical and Toroidal Parameterizations Without Vertex Seams". *Journal of Graphics Tools* 16.3 (July 2012), 144–150. URL: http://vcg.isti.cnr.it/Publications/2012/Tar12 3.

[Tar16] Tarini, Marco. "Volume-encoded UV-maps". *ACM Trans. Graph.* 35.4 (July 2016), 107:1–107:13. ISSN: 0730-0301. DOI: 10.1145/2897824.2925898. URL: http://doi.acm.org/10.1145/2897824.2925898 3.

[THCM04] Tarini, Marco, Hormann, Kai, Cignoni, Paolo, and Montani, Claudio. "PolyCube-Maps". *ACM Trans. Graph.* 23.3 (Aug. 2004), 853–860. ISSN: 0730-0301. DOI: 10.1145/1015706.1015810. URL: http://doi.acm.org/10.1145/1015706.1015810 3.

[TYL17] Tarini, Marco, Yuksel, Cem, and Lefebvre, Sylvain. "Rethinking Texture Mapping". *ACM SIGGRAPH 2017 Courses*. SIGGRAPH '17. Los Angeles, California: ACM, 2017, 11:1–11:139. ISBN: 978-1-4503-5014-3. DOI: 10.1145/3084873.3084911. URL: http://doi.acm.org/10.1145/3084873.3084911 2.

[VMG16] Villanueva, Alberto Jaspe, Marton, Fabio, and Gobbetti, Enrico. "SSVDAGs: Symmetry-aware Sparse Voxel DAGs". *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '16. Redmond, Washington: ACM, 2016, 7–14. ISBN: 978-1-4503-4043-4. DOI: 10.1145/2856400.2856420. URL: http://doi.acm.org/10.1145/2856400.2856420 3.

[YKH10] Yuksel, Cem, Keyser, John, and House, Donald H. "Mesh colors". *ACM Transactions on Graphics* 29.2 (2010), 15:1–15:11. ISSN: 0730-0301. DOI: 10.1145/1731047.1731053. URL: http://doi.acm.org/10.1145/1731047.1731053 3.

[YLT19] Yuksel, Cem, Lefebvre, Sylvain, and Tarini, Marco. "Rethinking Texture Mapping". *Computer Graphics Forum (Proceedings of Eurographics 2019)* 38.2 (2019), 535–551. DOI: 10.1111/cgf.13656. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13656 2.

[Yuk16] Yuksel, Cem. "Mesh Colors with Hardware Texture Filtering". *ACM SIGGRAPH 2016 Talks*. SIGGRAPH '16. Anaheim, California: ACM, 2016, 10:1–10:2. ISBN: 978-1-4503-4282-7. DOI: 10.1145/2897839.2927446. URL: http://doi.acm.org/10.1145/2897839.2927446 3.

[Yuk17] Yuksel, Cem. "Mesh Color Textures". *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 17:1–17:11. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105780. URL: http://doi.acm.org/10.1145/3105762.3105780 2, 3.

[ZMT05] Zhang, Eugene, Mischaikow, Konstantin, and Turk, Greg. "Feature-based Surface Parameterization and Texture Mapping". *ACM Trans. Graph.* 24.1 (Jan. 2005), 1–27. ISSN: 0730-0301. DOI: 10.1145/1037957.1037958. URL: http://doi.acm.org/10.1145/1037957.1037958 2.