Thesis for the degree of Doctor of Philosophy

On sparse voxel DAGs and memory efficient compression of surface attributes for real-time scenarios

Dan Dolonius

CHALMERS

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2022

On sparse voxel DAGs and memory efficient compression of surface attributes for real-time scenarios
Dan Dolonius
ISBN: 978-91-7905-631-5

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5097
ISSN 0346-718X

Technical Report report no 212D
Department of computer Science and Engineering
Research group: Computer Graphics
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Phone: +46(0)32 772 1000

Contact information:
Dan Dolonius
Department of computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Phone: +46(0)70 7769178
Email: dan.dolonius@gmail.com
URL: http://www.cse.chalmers.se/~dolonius/

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2022

On sparse voxel DAGs and memory efficient compression of surface attributes for real-time scenarios

Dan Dolonius
Department of Computer Science and Engineering
Chalmers University of Technology

Thesis for the degree of Ph.D of Engineering

Abstract

The general shape of a 3D object can expeditiously be represented as, e.g., triangles or voxels, while smaller-scale features usually are parameterized over the surface of the object. Such features include, but are not limited to, color details, small-scale surface-normal variations, or even view-dependent properties required for the light-surface interactions. This thesis is a collection of four papers that focus on new ways to compress and efficiently utilize surface data in 3D for real-time usage.

In **Paper IA** and **IB**, we extend upon the concept of *sparse voxel DAGs*, a real-time compression format of a voxel-grid, to allow an attribute mapping with a negligible impact on the size. The main contribution, however, is a novel real-time compression format of the mapped colors over such sparse voxel surfaces.

Paper II aims to utilize the results of the previous papers to achieve *uv*-free texturing of surfaces, such as triangle meshes, with optimized run-time minification as well as magnification filtering.

Paper III extends upon previous compact representations of view dependent radiance using *spherical gaussians* (SG). By using a *convolutional neural network*, we are able to compress the light-field by finding SGs with free directions, amplitudes and sharpnesses, whereas previous methods were limited to only free amplitudes in similar scenarios.

Keywords: voxel, geometry, octree, directed acyclic graph, compression, surface properties, filtering, neural networks, spherical gaussians, light field

Acknowledgements

First and foremost, I am very grateful for my mother Amelie, my fathers Leif and Per, my sister Emma, my partner Freja, and my mother in law Catharina for their continuous support and encouragement. I would also like to thank my supervisor Ulf, my co-supervisor Erik, and my co-workers at Chalmers; Sverker, Alexandra, and Roc. Further, I would like to thank my co-students during my master studies; Pierre, Edwin, and Damiano for all the interesting discussions we have had, and still do. And finally, a special thanks to the dogs Ankan, Nelson, and Lizzie for that little extra bump of emotional support.

Dan Dolonius
Göteborg, February 2022

List of Appended Publications

This thesis is a summary of four publications. The second publication (**IB**) is an extension to the first publication (**IA**) but is included since the contributions are arguably enough to constitute a paper on its own.
References to the papers will be made with roman numerals.

Paper IA - Dan Dolonius, Erik Sintorn, Viktor Kämpe, Ulf Assarsson,
Compressing Color Data for Voxelized Surface Geometry (original),
I3D '17 Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Article No. 13 (Best paper award).

Paper IB - Dan Dolonius, Erik Sintorn, Viktor Kämpe, Ulf Assarsson,
Compressing Color Data for Voxelized Surface Geometry (extension),
IEEE Transactions on Visualization and Computer Graphics,
(Volume: 25, Issue: 2, Aug. 18 2017, Pages: 1270 - 1282).

Paper II - Dan Dolonius, Erik Sintorn, Ulf Assarsson,
UV-free Texturing using Sparse Voxel DAGs,
Computer Graphics Forum,
(Volume: 39, Issue: 2, Jul. 13 2020, Pages: 121 - 132).

Paper III - Roc Ramon Currius, **Dan Dolonius**, Erik Sintorn, Ulf Assarsson,
Spherical Gaussian Light-field Textures for Fast Precomputed Global Illuminations,
Computer Graphics Forum,
(Volume: 39, Issue: 2, Jul. 13 2020, Pages: 133 - 146).

Other papers and work co-authored by Dan Dolonius:

- Viktor Kämpe, Erik Sintorn, **Dan Dolonius**, Ulf Assarsson,
  Fast, Memory-Efficient Construction of Voxelized Shadows,
  IEEE Transactions on Visualization and Computer Graphics,
  (Volume: 22, Issue: 10, Oct. 1 2016, Pages: 2239 - 2248).

- Ulf Assarsson, Markus Billeter, **Dan Dolonius**, Elmar Eisemann, Alberto Jaspe, Leonardo Scandolo, Erik Sintorn,
  Voxel dags and multiresolution hierarchies: from large-scale scenes to pre-computed shadows,
  Eurographics (Tutorials),
  (2018, Pages: 9 - 11)

Table of Contents

Part I

Introductory chapters

Chapter 1

Introduction

In the physical domain, we perceive objects through their interaction of light on an atomic or even quantum level, e.g., by blocking, reflecting, and transmitting photons. However, we can make our lives a bit easier by realizing that some high-level concepts, such as for instance silhouettes and shadows, can be sufficiently reasoned about on a macroscopic scale, while other properties, such as the smoothness, of a surface resides in the microscopic domain (scratches and dust). Further, effects such as refraction, diffraction and metallic surfaces might even need some degree of atomic or quantum scaling to be fully explained [32].

In the realm of computer graphics where we often want to render, in some sense, physical objects, the general representations is as follows. On a macroscopic level, we let the surface geometry of the object be approximated by simple primitives, such as triangles, whereas on the microscopic and lower levels, we approximate the material of the object with statistical distributions and/or analytical functions. This allows us to efficiently simulate how the light interacts with the object, and by increasing the number of primitives and using more physically accurate distributions, we can render more complex and realistic scenes but at the expense of rendering times and memory consumption.

Suffice to say, just naively increasing complexity will yield diminishing returns as the memory and processing requirements grows much faster than what the hardware can provide. We thus need to find ways to more efficiently represent these objects, as well as cheaper methods of rendering them, without sacrificing quality.

This thesis aims to improve and extend existing representations in order to decrease the memory footprint while still being inexpensive to visualize with high quality.

1.1 Thesis structure

The first part of this thesis is organized as follows. Chapter 2-3 describes existing concepts and methods on which this thesis expands upon. Chapter 4 offers some brief problem statements which this thesis aims to overcome. Chapter 5 is a summary of the papers addressing named problems, and chapter 6 is left for discussion and future work.

The second part consists of the appended papers.

Chapter 2

Object representation

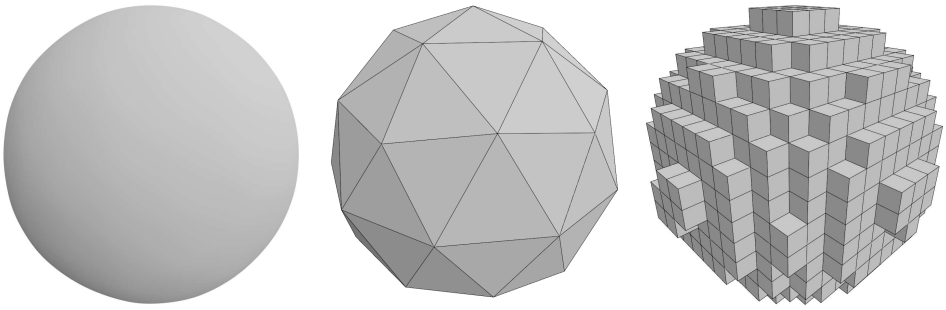


Figure 2.1: Left: Original. Middle: Triangulated. Right: Voxelized.

There is a plethora of ways to create and represent virtual objects, besides the basic primitives. An object can be scanned, using e.g., LIDAR or stereoscopic matching, resulting in a point cloud. It can be modelled using parametric surfaces, such as *Non-uniform rational B-splines* or *Subdivision surfaces*. There are also pure functional representations, e.g., where the object is represented as mathematical functions creating a *Signed Distance Field*, which can be rendered by sampling the field and classifying the samples to be inside or outside the geometry whether their value is within a certain threshold. While the different methods all have their own advantages, a common issue is that they are either inefficient to store or to render. Point clouds require a lot of memory and do not really support non-rigid animations, while parametric solutions and distance fields has too many degrees of freedom to be rendered with maximum efficiency. Thus, in order to allow real-time applications or reduce rendering times (power consumption) in offline applications, these representations are often converted to simple primitives, either before or during rendering, except for some niche cases.

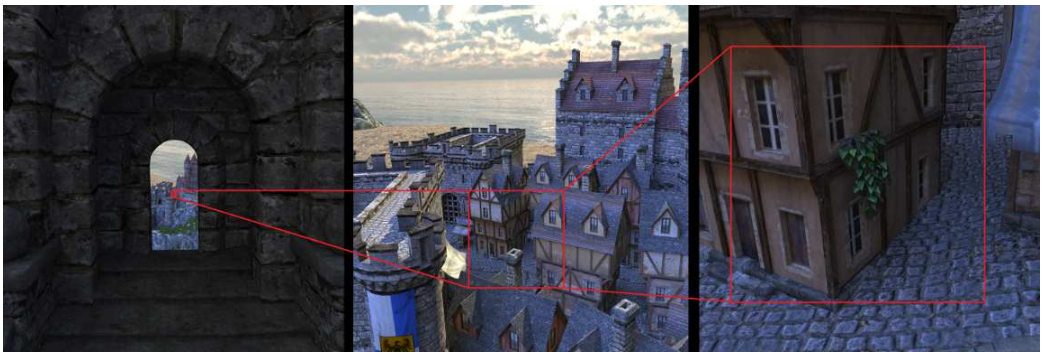


Figure 2.2: An example of how a pixel can encompass many small triangles.

2.1 Surface geometry

The most common primitive, as hinted in the introduction, is a *triangle* (see Figure 2.1). A collection triangles sharing edges is called a *triangle mesh*, whereas a collection of independent triangles is called a *triangle soup*. The advantage of a triangle is that it is a very simple representation, which results in fewer edge and degenerate cases. For example, there is only one plane that can be defined by the three vertices of a triangle, while the four vertices of a quad can define up to two pairs of two planes, of which desired set is ambiguous unless extra information is provided. Having so few edge cases has given rise to the *Graphics Processing Units* (GPUs), which is hardware initially dedicated to render massive amount of triangles in parallel through a process called rasterization.

During rasterization, the triangles are projected onto a virtual plane, the so called raster image, which is a collection of pixels. The triangle closest to the camera which overlaps the pixel center is then sampled to yield the final color for that pixel. Herein lies a problem. Consider Figure 2.2, where the camera is inside a tower, overlooking a town far away. The red square in the leftmost image represents one pixel in our image, and as we zoom in, we see that this pixel encompass and entire building, potentially made of thousands of triangles. If we only take one sample for this pixel, this means that only one triangle will represent that pixel, resulting in severe aliasing artifacts. Another issue is that, while inside the tower, keeping all the triangles that can not be seen, on the GPU, can be both wasteful or impossible due to limited amounts of GPU memory. Thus, in order to combat such issues, the models can first be generated with different *levels of detail* (LOD) and stored on disk. Depending on the distance to the object to be rendered, the most suitable detail level is loaded onto the GPU. However, creating such

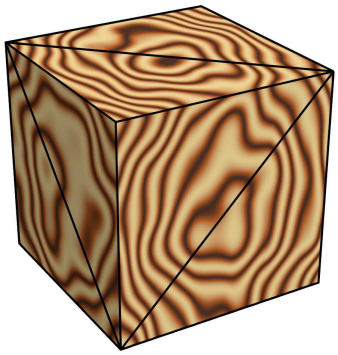


Figure 2.3: A textured cube composed of 12 triangles.

models is extra overhead for artists, and while there exist tools for automatic downsampling, extra tweaking is still necessary to create high quality models.

An interesting alternative, is *voxels*, which is short for *volume element*, analogous to *pixel / picture element* (see Figure 2.1). To represent geometry as voxels, they can be stored in a binary 3D grid, where 0 represents empty voxels and 1 when the voxel contains geometry. This naive and direct approach however is not very efficient and does not solve any of the named problems with triangles. The benefit is that it is easier to create LODs with voxels, as we can easily downsample the grid using the simple rule that downsampled voxel contains geometry only if any composing voxels do. Further, the memory overhead may also be considerably reduced by only storing the non-empty voxels in a so called *Sparse Voxel Octree* (SVO). We will revisit and expand upon this concept in later chapters.

2.2 Surface properties

So far, we have only addressed the macroscopic properties, the geometry, of the object. More often than not, we also need to specify the microscopic properties, such as the material of the object. The material properties may vary on the model of the material, but common properties are how it scatters light (rough/shiny), transmission (transparent/opaque), and the reflected wavelengths (color), and on rare occasions, it is sufficient to store this on a per primitive basis. However, as can be seen in Figure 2.3, we would need around a million triangles to produce the pattern, compared to only twelve for the geometry. Since a non-degenerate triangle defines only one plane in 3D space, it should be trivial to realize that we can uniquely define every point inside the triangle by mapping the vertices to a 2D plane. Thus, by

adding just two extra properties to the vertices (the x and y coordinates on the 2D plane, a so called *uv*-map), we can define every point on the surface of the triangle and use a 2D grid as a *lookup table* (LUT) to query the desired surface attributes during rendering. Such a LUT is called a *texture*, for which each element is called a *texel*. Note that while 2D textures is the common use case, texturing is by no means restricted to just 2D. For example, 1D textures can be sufficient for creating gradients and 3D textures can be useful for volumetric objects.

However, individually mapping each triangle would result in a triangle soup, since there can be no shared vertices as in a triangle mesh. In many cases, this will increase the memory overhead by about a factor of 6×. Further, if we want to have filtered lookups, they would only be filtered on a per-triangle basis, resulting in aliasing and discontinuities.

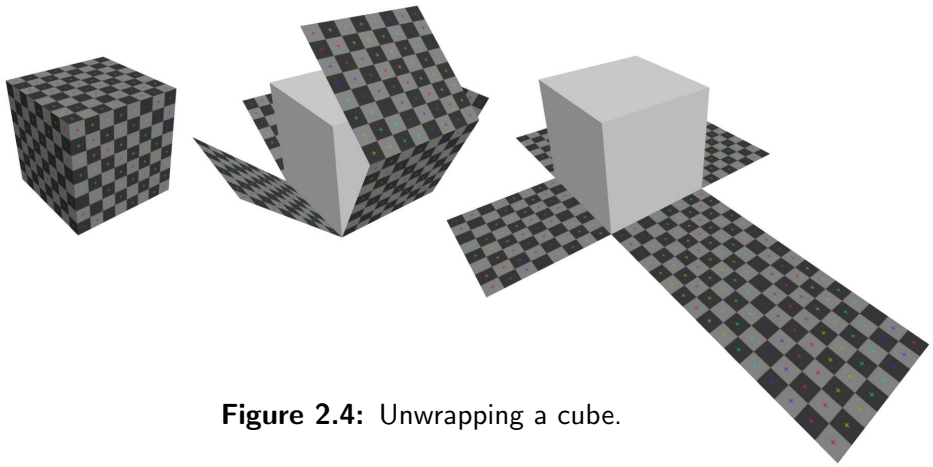


Figure 2.4: Unwrapping a cube.

For these reasons, we ideally want to map as many shared vertices as possible. This is achieved by a process called unwrapping, which is illustrated in Figure 2.4, where certain vertices are split so that the model can unfold to the 2D-plane (a.k.a, *cuts*). In general, there is no perfect unwrapping as there will always be a compromise by preserving continuity, angles, or area, where neglecting any of each will result in different kinds of artifacts or distortions. The discontinuities arise where the cuts are placed, resulting in jarring artifacts called *seams*, forcing artists to strategically place cuts in areas less likely to be viewed, while keeping the distortions to a minimum. For a comprehensive guide on alternative texture mapping techniques, please refer to the course notes on *Rethinking Texture Mapping* [45] by Cem Yuksel et al.

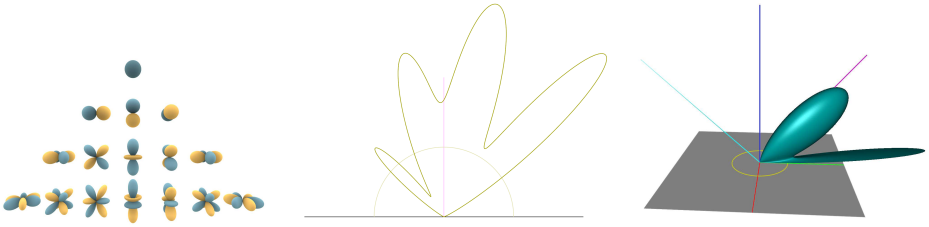


Figure 2.5: Left: SH. Middle: SG. Right: BRDF.
(SH by Inigo Quilez https://www.shadertoy.com/view/lsfXWH)

2.3 Sampling and Filtering

The rendering of an image is in essence a sampling problem, where the sample points are the pixels and the virtual scene is the signal. In accordance with the *Nyquist–Shannon sampling theorem*, we indeed suffer aliasing artifacts if e.g., the projected texture's resolution exceeds the sample resolution. In such cases, a solution is to remove high frequencies with a low-pass filter. As for textures (and voxel grids), they can trivially be pre-filtered by downsampling the grid, which for textures is commonly known as a *mip-map hierarchy*. Texture aliasing can thus be avoided by sampling from the texture which best matches the target resolution after projection. There will however be artifacts, visible as discontinuities when adjacent texels are sampled with different mipmaps. The solution is to, for each texel, interpolate between the suitable mipmaps, generally using a first degree polynomial. Not surprisingly, we call this a linear filter for minification.

On the other hand, we might also suffer from oversampling when the projected texture resolution is lower than the target resolution, resulting in discontinuities and block-like artifacts. In this case we can, at run time, mitigate these problems by upsampling the texture, simply by sampling the adjacent texels relative the sampling point, p, and interpolate with respect to the distance from p and the texel centers, i.e., a bi-linear filter for magnification. Combining these two methods is equivalent to sampling a cube, a tri-linear filter. This idea is easily generalizeable to higher dimensions, e.g., a tri-linear filter for voxel magnification, and a linear filter for minification, resulting in a quad-linear filter.

2.4 View dependent properties

These days, practically all physically-based renderings of virtual scenes are based on solving the light transport equation [11],

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f(p, \omega_o, \omega_i) L_i(p, \omega_i) |cos(\theta_i)| d\omega_i,$$

where the exitant radiance L_o must be equal to the emitted radiance L_e plus the scattered incident radiance L_i over all incoming directions ω_i over the hemisphere Ω, at a surface point p. How the light is scattered is defined by the *bidirectional reflectance distribution function* (BRDF), f, and a common such distribution is the *Cook-Torrance BRDF* which is a micro-facet model, defined as

$$f(p, \omega_o, \omega_i) = \frac{F(\omega_o, \omega_h) G(\omega_i, \omega_o, \omega_h) D(\omega_h)}{4(\boldsymbol{n} \cdot \omega_i)(\boldsymbol{n} \cdot \omega_o)},$$

where F is a *Fresnel function*, G a *geometric attenuation function* and, D a *normal distribution function*. Finally, $\boldsymbol{n}$ is the normal of the surface, θ_i is the angle between the normal and incident direction, and ω_h is the half-vector between the normal and the incoming direction. In Figure 2.5, we see an example of the Cook-Torrance BRDF, given the incoming direction (cyan), normal (blue).

While it is most common to store view-independent properties such as colors, normals, and BRDF specific parameters to calculate the view dependent result on the fly, we need simpler and approximate calculations for real time scenarios, or long rendering times for realistic high quality images. For perfectly specular surfaces we can precompute the incoming radiance from light sources far away by a single *environment map* [3], and this technique has later been extended to allow glossy surfaces by pre-convoling the environment map with respect to the surface's BRDF [22]. While it is possible to capture local radiance by introducing multiple local environment maps for different points in the scene, the excessive memory overhead inhibits it to be practically useful in most cases.

Another option is to instead approximate the radiance transfer [38] with *Spherical Radial Basis Functions*, (SRBF), which then are convolved with the BRDF.

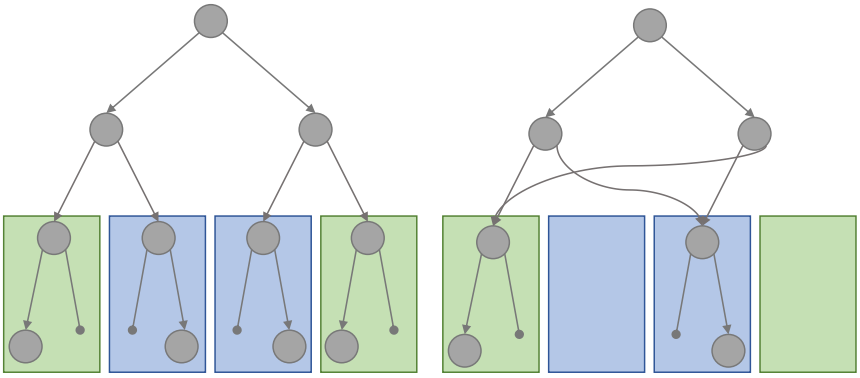
The most common basis is probably *Spherical Harmonics* (SH), which is a set of particular orthogonal polynomial basis functions defined over the surface of a sphere and is also frequently used in physics, e.g., for computations of electron configurations, representation of gravitational or magnetic fields, and solutions of the Schrodinger equation. For representation of the first few, see Figure 2.5. While they are practical to work with for diffuse or

rough surfaces [30], SHs become increasingly problematic for glossy surfaces and high-frequency lighting environments, as more coefficients are needed to avoid ringing artifacts due to the nature of these polynomials.

Another popular basis is *Spherical Gaussians*(SG), which is a special case of the *von Mises-Fisher distribution*, and is mathematically defined as $G(\boldsymbol{v}; \boldsymbol{u}, \lambda, \boldsymbol{\mu}) = \boldsymbol{\mu} e^{\lambda}(\boldsymbol{v} \cdot \boldsymbol{u} - 1)$, where $\boldsymbol{u}$ is the *axis*, $\boldsymbol{\mu}$ the *amplitude*, and λ the *sharpness*. By summing up several SGs, arbitrary distributions can be approximated, as can be seen in the middle image of Figure 2.5 that is constructed by four SG lobes with different parameters. In the work by Green et al. [8], they introduce a hybrid method where the diffuse, direct and indirect terms, for view-independent effects were modeled using SHs or wavelets, while the higher frequency glossy terms were modeled using SGs. In the work by Tsai et al. [40], they point out that since the model by Green et al. is restricted to model only specular effects with SGs, it is unclear whether effects such as all-frequency shadows could be handled. Instead, they propose a unified framework which also allows rendering of all-frequency shadows, even though they are at a disadvantage with representing highly specular BRDF's. Succeeding works [41, 10, 43] have improved on this concept by, for instance, allowing spatially varying BRDF's, dynamic scenes, and introducing anisotropic SGs for better reconstructions.

Chapter 3

Sparse voxel octrees and DAGs



(a) Original SVO. **(b)** Transformed to DAG.

Figure 3.1: Illustration of SVO to dag transformation. Green and blue boxes represents identical subtrees. For brevity, the octree is represented as a binary tree.

As explained earlier, storing even the minimum unit of data (a bit) per voxel in a voxel grid is infeasible at larger resolutions. Arguably, a voxelized surface will be very sparse, as thus de-facto compression algorithms will perform reasonably well over the entire volume, due to the low entropy, for offline storage. However, decoding of the compressed data will be far too slow in a real-time application, and thus we need a format suitable for such situations. A sparse voxel octree (SVO) is a data structure where a volume is recursively subdivided along each canonical axis such that there are eight smaller volumes (children) composing the larger volume (parent). Further, by subdividing in an explicit order, we can implicitly store the children such that all is needed is an 8-bit mask for the nodes in the SVO, sacrificing practicality in favour of compression.

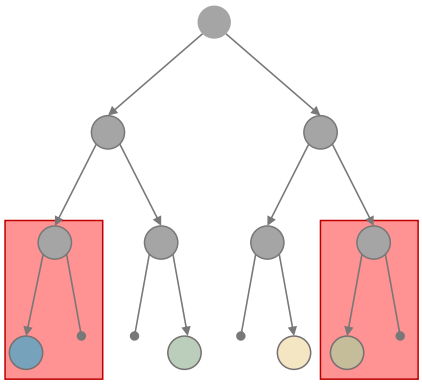


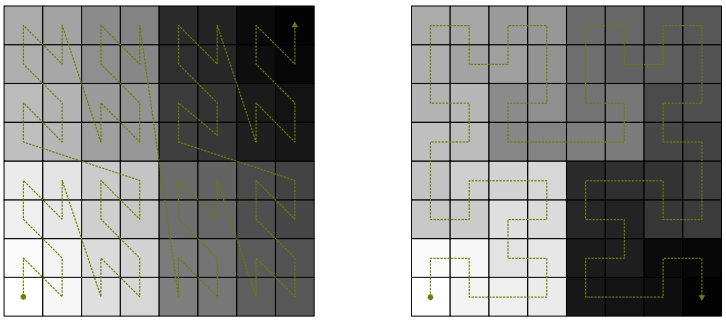
Figure 3.2: Red boxes represent previously identical subtrees which become different when color information is added to leaf nodes.

While the SVO is a more compact alternative than a grid, the memory requirements still grow unmanageable at extreme resolutions. Kampe et al. [13] realized that since there are only 2^8 variations of the leaf[1] masks, by merging identical leaves, the final level can be made significantly more compact. Similarly, recursively storing only the unique subtrees, they manage to drastically compress the entire SVO in what now is called a *Sparse Voxel DAG*[2] (see Figure 3.1).

However, with this, what is gained in compression is lost in practicality. While an SVO can store arbitrarily large payloads in the leaves, the DAG does not share this feature, as adding extra bits of information will substantially reduce the number of identical elements, thus abolishing the compression (see Figure 3.2). While storing the geometry attributes in the leaf nodes in an SVO provides an easy mapping, this will also render any significant compression of the attributes themselves nearly impossible. When attribute compression is desired, a more practical approach is to decouple the attributes from the SVO so they can be processed independently.

In the paper *Out-of-Core Construction of Sparse Voxel Octrees* [1], Baert et al. describe how a SVO can be constructed very efficiently in a streaming manner by first ordering the voxel grid along a *Morton Space-Filling Curve*. The concept of space-filling curves was introduced by the mathematician Giuseppe Peano in 1890 and describes a continuous surjection from the unit interval to unit square [28]. As it turns out, this also holds for N-dimensional hypercubes, and in the case of a voxel grid when countable and finite, it may also serve as an injection, i.e., we can map from a curve to a grid and back. The Morton curve is widely adopted due to its simplicity as well as

[1]The children of the lowest level of the SVO
[2]DAG - Directed Acyclic Graph



(a) Morton curve **(b)** Hilbert curve

Figure 3.3: Example of two common space-filling curves. The path is shown in green, and each cell is color coded to show how coherency is preserved.

locality preserving features, and other examples of applications includes, e.g., optimizing 2D memory accesses or faster construction of bounding volume hierarchies [26, 19]. Refer to Figure 3.3 for an example of the Morton and also popular Hilbert curve. As Dado et al. [6] points out, a depth first traversal of an SVO or DAG will preserve locality, and we realise that by explicitly ordering the children, as in [1], we can sort the attributes relatively coherently along any space-filling curve we so desire. This provides a good foundation for attribute compression, and similarly to how we can replace the data stored in the leaves with a more light-weight attribute index for a reasonably sized SVO, we can achieve the same effect with a DAG by storing a leaf counter in the nodes.

Chapter 4

Problem statements

4.1 Compressing voxel-property information

The elegance of an imlpicit mapping in the SVO is unfortunately lost when compressing it to a DAG, since adding extra bits of information, e.g., color, in the leaves will significantly hinder compression, as previously identical leaves (and thus subtrees) will now be unique, which can be seen in Figure 3.2. What we need is to be able to decouple the attributes from the DAG, e.g., by keeping track of non-empty or empty voxels as in [6, 42], as this will not change the uniqueness of the nodes. As the number of voxels grow, the raw attribute data will rather quickly be the dominant factor in terms of memory, rendering the DAG practically redundant. In the paper by Dado et al. [6], we saw the first attempt of a lossy compression format for real-time scenarios, using color palettes, reminiscent of *png* images.

Admittedly, neither SVOs or DAGs are not as fast as a hardware 3D texture as we need to do approximately $log_2 n$ more data reads without hardware support, where n is the original grid resolution. If filtering is desired, this will result in up to $8 \times -16\times$ times more reads, further inhibiting the use in real-time scenarios.

4.2 View dependency

In real-time scenarios, it is popular to represent both the light field and BRDF as either SHs or SGs, as it allows for an efficient convolution with the incoming illumination. If we restrict the SGs to only have a variable amplitude, we can efficiently compute the parameters using non-negative

Figure 4.1: Using only pre-convolved environment map vs ground truth.

least-squares methods[1]. Since SHs is an orthonormal basis, the least squares projection can be performed, in real-time, by a simple integration of the light field against the basis functions [37, 15]. Consequentially, both SHs and SGs[2] have the added benefit that the parameters are trivially filterable, but with the downside that we require higher order SHs and more SGs to represent higher frequency effects. Removing the constraint of fixed direction and sharpness for SGs allows for much more efficient approximations using e.g., *gradient descent* (GD) or *expectation maximization* (EM) algorithms, which is perfectly viable if only independent sets of SGs are desired. However, if filterable SGs are required, the problem becomes significantly non-trivial as there is no guarantee that the the ordering of lobes of different sets of SGs is preserved, using GD or EM naively.

In real-time applications, using the Cook-Torrance BRDF, it is common to pre-convolve the environment map with the normal distribution function and precompute the expensive convolution in a 3D LUT while also moving the remaining terms of the light transport equation outside of the integral. Those terms are then evaluated only for the perfect specular direction, and as such, the error of this estimation will be worse the rougher the material. Nevertheless, in practice this is sufficient for unoccluded reflections. The issue with pre-convolving the environment map however, is that, since it ignores local occlusions, in scenarios with lots of occlusion, it can have detrimental effects on the final result, such as the indoor scene in Figure 4.1.

[1]https://mynameismjp.wordpress.com/2016/10/09/sg-series-part-5-approximating-radiance-and-irradiance-with-sgs/

[2]With fixed sharpness and directions

Chapter 5

Summary of Included Papers

In this Section, we will describe the main contributions of each paper:

Paper IA - A novel method for compressing surface properties stored with a voxelized scene representation, which outperforms previous work by up to 1.8× at similar quality.

Paper IB - An extension to the previous method in which compression is further improved by up to 2.5×.

Paper II - An optimization on attribute lookups in DAGs for fast filtering and uv-free texturing.

Paper III - A novel optimization technique using CNNs to find filterable SGs.

5.1 Paper IA - Compressing Color Data for Voxelized Surface Geometry (Original)

5.1.1 Problem

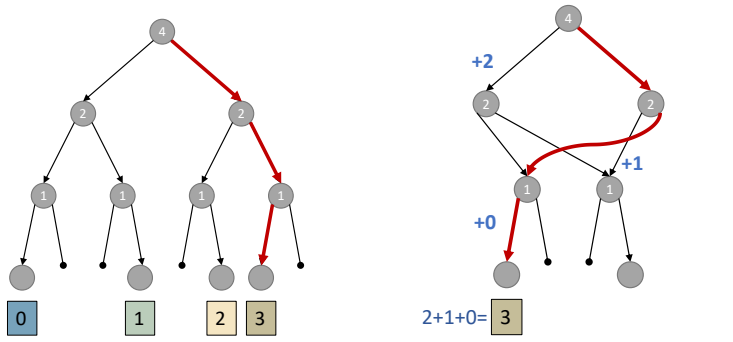
In a triangle mesh, the attribute data is commonly stored as a texture. Since the *uv* mapping is continuous over one, up to several, triangles, the data is generally coherent and is therefore suitable for compression; e.g., *jpeg* and *png* for offline storage or *bc* and *astc* for real-time scenarios. Unfortunately this is not the case for sparse data as with an SVO [18] or DAG [13]. While it is possible to implicitly map the geometry of an SVO by storing the attributes in the leaves, a DAG lacks this feature. Further, since the map is a sparse 3D set instead of a dense 2D, conventional compression formats do not perform well, or even apply. Further, even if applicable, formats such as the BC family has a fixed compression ratio of 4-8 bits per pixel, while ASTC also has a fixed ratio of 0.89-8 bits texel[1].

As shown in [42, 6], it is possible to introduce a leaf counter to the DAG nodes, such that an attribute index can be computed by a running sum while keeping the DAG nodes fingerprint intact, thus decoupling the attributes from the geometry with only a small overhead (the leaf counter). Dado et al. further provide the first attempt at a compression format for the decoupled data, and while they achieve impressive compression ratios, their format is a bit complex and suffer when the whole color space is evenly utilized, due to their approach on using color palettes.

5.1.2 Method

While [42, 6] use a per-pointer leaf count, we instead opt for a per-node counter (see Figure 5.1). Since our DAG is stored in a 32-bit array, where the elements are (8-bit child mask + 24-bit padding) and ($n\times$32-bit child pointers), we can use the 24 padding bits as our leaf counter. Using 24 bits for the leaf counter is generally sufficient for DAG resolutions up to 1024^3, and for larger resolutions we can just store an offset for the upper levels into a separate 32-bit counter array. These nodes are few, and in our experiments, we found that the overhead was generally far less than 0.1%. This way, we have minimal impact to the DAG size instead of almost doubling it, in the case of per-pointer counters. The drawback is that we need to read the preceding childs counter for each node, resulting in extra memory reads. In

[1] for 2D textures



(a) SVO: Color indices can be trivially mapped to nodes.

(b) DAG: Storing total number of leaf nodes preserves identical nodes.

Figure 5.1: Example how a desired color index (red path) can be found by a running sum of preceding child nodes total number of leaf nodes.

our experiments, we found that while per-pointer counter lookups is around $1.2\times - 1.9\times$ faster, it also results in a $1.5\times - 1.8\times$ larger DAG.

In this paper, we investigate two methods of compressing colors, both of which rely on first ordering the voxels along a 1D space-filling curve.

Regarding the first method, we argue that if we map this 1D array to a 2D array, again using a space-filling curve, the colors will often be reasonably coherent, and thus we will be able to utilize conventional compression algorithms.

In the second method, we introduce a novel compression format, inspired by the BC and ASTC [27, 9] families of compression formats, since they are real time formats with almost zero overhead. In these formats, the texture is split into blocks, e.g., 4×4 texels, of which each contain a set of base colors and a cheap per-texel value, which together are used to reconstruct the original color for each of the texels. Similarly, in our format, we divide the color array into blocks, with the difference that our blocks are not of fixed size. For each block, we assign two base colors (e.g., 2×16 bits) and a per-color weight (e.g., 3 bits). The base colors (B_1, B_2) and the weight (w) are chosen such that the original color (C) is reconstructed by a linear interpolation; $C = wB_1 + (1 - w)B_2$. Additionally, since the blocks are of variable length, we also need to store the first voxel index for the range of voxels assigned to that block (see Figure 5.2).

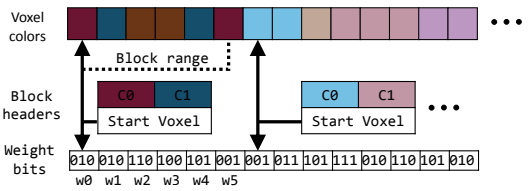


Figure 5.2: Visualization of data layout for fixed weight bit width.

The base colors and the first voxel index is stored together ($2 \times 16 + 32$ bits) in which we call *the header*, all of which is stored in a 32-bit array, and the weights are stored in a separate array. This way, the headers are of fixed size, which we will take advantage of during the decoding, explained later. Since the weights are always of fixed size, the means of compression are to minimize the number of headers, or in other words, find as large blocks as possible.

We achieve this by first specifying an error threshold, meaning that the *mean square error* (MSE) of the reconstructed color may not be larger than that threshold. The gist of the encoding is as follows.

1. Each color is considered a block.
2. Sequentially try to merge a block with its left or right neighbour, whichever has the lowest MSE after merging. If both merges violate the error threshold, the block is considered done. This involves finding new base colors and weights for the union of the two blocks considered.
3. When all blocks are processed, we start a new iteration with the newly merged blocks and repeat until all blocks are considered done.

The decoding of a color first requires finding the voxel index, which has already been covered. Given this index, we perform a binary search on the headers' first voxel index[2] to find the block spanning the color, as well as a direct lookup in the weight array using the same index. With this, we have all we need to reconstruct the color, the base colors (included in the header), and the weight.

5.1.3 Contributions

We have introduced an alternative to map attributes to a DAG. While lookup times are a bit more expensive, the memory overhead is practically negligible compared to previous methods, which almost doubled the size of the DAG. The main contribution is, however, a novel compression format for sparse 3D data that outperforms previous attempts in both compression/quality accompanied by a straight forward implementation. We also investigate and

[2]Which is only possible because of their fixed size.

demonstrate how it is possible to instead map the data to a texture in order to utilize conventional compression formats, such as *jpeg* for offline storage or *astc* for real-time scenarios.

5.2 Paper IB - Compressing Color Data for Voxelized Surface Geometry (Extension)

5.2.1 Problem

Given the fixed bit rate of the weights (bpw) in **Paper IA**, the theoretical optimal compression ratio at e.g., 3 bpw, is 12%, in the infeasible scenario where only one header is required for all colors. Thus, it would make sense to have a variable bit rate, where higher bit rates can span larger areas, decreasing the headers footprint at the cost of weights, while lower bit rates might be suitable for large regions of similar or identical colors. However to achieve this, there are two, not mutually exclusive, major hurdles we need to overcome:

1. How do we store the headers for this variable bits per weight. Naively storing a bit pointer per header will definitively inhibit compression as these consequently are required to be very large.

2. How to choose the per-header bit rate, so that we consume as little memory as possible.

Further, we also need to address another shortcoming of the previous paper, as it lacked the implementation for more than one LOD, which will result in aliasing when many voxels are projected onto the same pixel. This is not a problematic task as the method is already explained in **Paper IA**, but an investigation on how interleaving colors from different levels affects compression needs to be performed.

5.2.2 Method

We can reduce the weight-bit pointer size by first segregating the underlying color array into several smaller sets of the same size, which we call *macro blocks*. These macro blocks can then be compressed independently, where the contained header's voxel index and weight-bit pointer only need to be much smaller, local offsets relative the macro block, as can be seen in Figure 5.3. Using a size of 2^{14} colors for the macro blocks allows us to use 0-4 bits for the weights such that the required information can neatly be packed in the 32-bit section of the previous format's block header containing the voxel start index. We only need 14 bits to specify the macro-local voxel index, 16 bits for the macro-local bit pointer, and two bits to specify the number of bits per weight within that block. However, as the attentive reader might have noticed, two bits for the weight bit-width can only enumerate four different

cases, e.g., 1-4 bpw. We manage to enumerate the fifth case by realising that we never need all the bits set for the bit pointer, leaving this value free as a sentinel. The overhead is thus 1/128 extra bits per color for the macro-block header, plus a small extra overhead due to the fact that a block that ideally would reside in the border between two macro block will be forced to be (up to) two blocks instead. Naturally, and as the paper shows, this overhead is insignificant.

Our encoding relies on first creating a tree structure of blocks of different weight bit-rates. Using the same algorithm as in **Paper IA**, we first construct the leaf-level blocks at 0 bpw and then use those blocks to recursively create new blocks with higher bit rates. To find the optimal cut through this tree, we begin with the parents of the leaves, comparing the sum of the memory overhead of blocks with lower bit rates against the parent block (they describe the same set of colors), keeping the one(s) with lowest overhead. This is then repeated recursively with higher bit rates, resulting in an optimal cut when we reach the root.

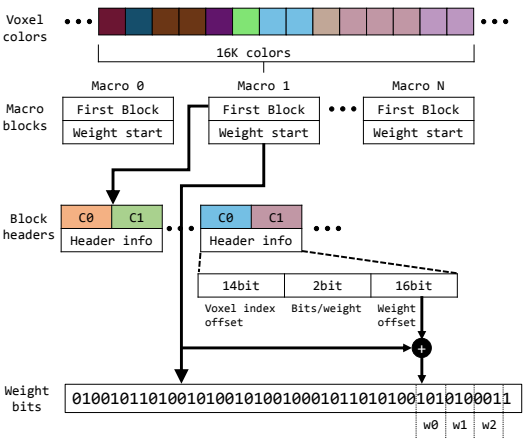


Figure 5.3: Visualization of data layout for variable weight bit width. © 2017 IEEE.

During decoding, the macro blocks are directly indexable given a voxel index[3]. Using the header index of the comprising and directly following macro blocks results in a range of header candidates. Similarly as in **Paper IA**, we perform a binary search to find the correct header and decode the color. Note however that an added benefit is that the binary search is now faster as we have implicitly restricted the range of header candidates through the macro blocks.

5.2.3 Contributions

Extending the previous compression format to allow variable bits per weight significantly decrease the compressed size given the same error thresholds.

[3]Explained in **Paper IA**

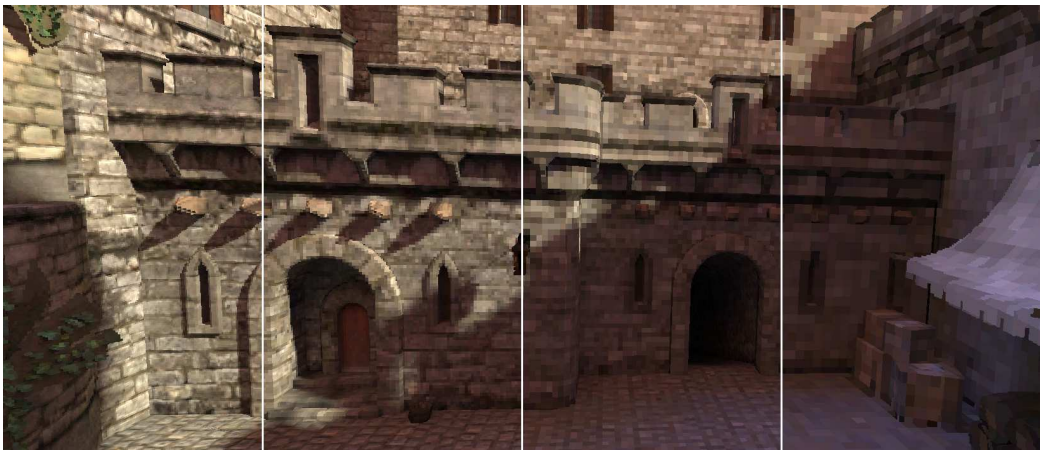


Figure 5.4: Different levels of mip mapping.

Further, we show that the decoding is as fast, or faster, than the previous format thanks to the reduced binary search range given by the macro blocks. Finally, we implemented LOD (Figure 5.4) and compared compression rates. The concern was that the scattering of internal nodes would decrease coherency and thus negatively affect compression. However, as it turns out, this concern was misplaced, as the color of the internal nodes are similar enough to actually yield better compression rates.

5.3 Paper II - UV-free Texturing using Sparse Voxel DAGs

5.3.1 Problem

While sparse voxel DAGs solve many problems, triangles are still the preferred primitives for many applications, since they offer other advantages for which there is yet no competitive alternative using voxels. However, UV-mapping a triangle mesh is not a trivial endeavour. Producing a high quality map requires the artist to carefully mind seams and distortions while not wasting texture space, which requires skill and time. While automatic methods do exist [35, 29, 36, 46, 14, 33, 39, 21, 7], the end result is far from the quality of a hand-crafted map.

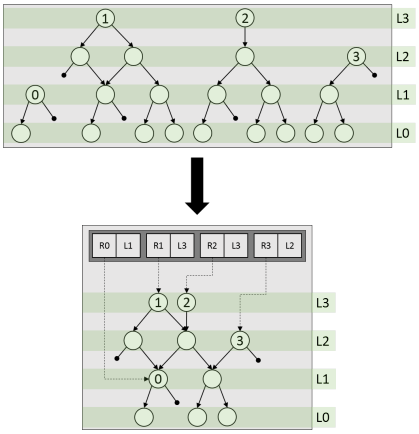


Figure 5.5: Merging DAGs.

An interesting alternative is to instead use *Sparse Volumetric Textures* [16, 2, 20]. These are all viable alternatives in their own right but nevertheless suffer from, for instance, seams, bleeding, memory overhead, globally uniform resolution or non-realtime performances.

In this paper, we expand upon these ideas, utilising the results from **Paper IB** to address the following problems:

- Fast filtered lookups in a DAG with compressed colors.
- Remove color bleeding due to disjointed surface parts of the object sharing the same voxel.
- Handle mipmap hierarchies where the problem above becomes even more pronounced as the increased voxel sizes and larger filter sizes will encompass more geometry.
- Allowing per-mesh resolutions using a single DAG.

5.3.2 Method

The first step is to acquire a voxel representation of the model, where we store the non-empty voxels intersecting with the geometry, along with a filtered color lookup if we wish to transfer existing colors from the model. As the model may very well contain surface elements that will share voxels, the artist is required to specify individual parts that should not bleed over to each other. These parts are then voxelized and converted into separate DAGs.

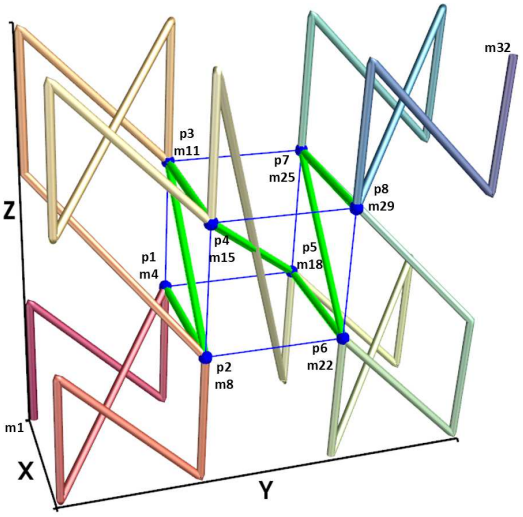


Figure 5.6: The specific ordering for some sample, **p** (inside the blue volume), indicated by the green line.

In order to unify them into one DAG, we merge these DAGs by aligning them vertically at the leaf level (see Figure 5.5) and storing the roots and their starting level in the DAG in a separate array.

For a filtered lookup, we present two alternatives. First a high quality quad-linear filter, which relies on the realization that the neighbouring voxels in the filter kernel will have a specific relative Morton order depending on the sample position (Figure 5.6). This allows us to calculate an optimal traversal of the DAG where we only need to keep a stack of three elements, allowing us to restart from the first shared parent of the current and next voxel, instead of the root. The second alternative is a custom multi-sample scheme which allows even faster minification filtering at the expense of lower quality for high frequency textures.

We also improve on the color decoding since the specific ordering of our samples guarantees that the colors will be at strictly increasing positions in our color structure, which allows us to quickly realize if the next color is in the same or next block, or moving the lower bound for the binary search if it is not.

The selection of independent parts are made in Blender, although, for most cases, the original models were already satisfactory, i.e., using the different meshes composing the model as the independent parts. The creation of the data structure is performed in OpenGL, CUDA, and C++. To render the results, we rasterize the per-pixel object-space position of the model, which is

used in our CUDA kernels as the sampling position for the algorithm. Note that this would also work for animated and skinned meshes, as long as the object-space position can be evaluated.

5.3.3 Contributions

This paper presents an implict mapping using sparse voxel DAGs with compressed textures. We also show how we can avoid color bleeding or seams by letting an artist specify which parts of the model should be independent. While this process requires manual labour, this is typically less complex than constructing a traditional *uv*-map. This also allows for different texture resolution per part by voxelizing the meshes at different resolutions. By merging the individual DAGs for each part, we also improve on the compression ratio of the textures by up to 32%.

We also optimize for quad-linear filtering and multisampling, for which the magnification and minification filtering results in a 2.5±0.2 times speedup compared to the naive approach. Using multisampling instead of minification roughly doubles the speedup, e.g., from 5±2 ms to 2.5±0.5 ms in full HD, on a RTX 2080 graphics card.

5.4 Paper III - Spherical Gaussian Light-field Textures for Fast Precomputed Global Illumination

5.4.1 Problem

Global illumination has a huge impact on realism and fidelity and is therefore highly desired for movies as well as real-time applications. While it is possible to use path tracing to sample the scene thousands of times for each pixel in offline applications, current high-end GPUs only allow a few samples under real-time constraints. Recent methods allow the re-use of previous frames combined with sophisticated denoising techniques [23, 5]. However, they are still too expensive on mid or low-end hardware.

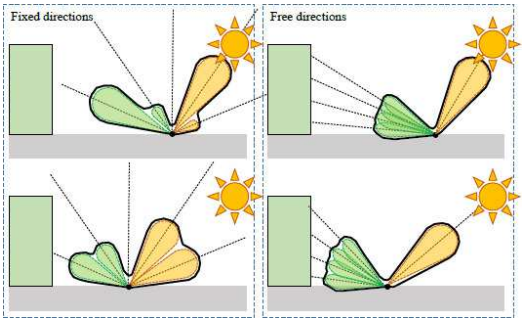


Figure 5.7: Left: Fixed directions cannot capture high frequency changes. Right: More accurate approximation using free directions.

Even though it is possible to precompute and compress indirect illumination over static scenes in the form of SRBFs, the common bases, SH and SG, share some inherent or practical limitations. Using SH, or SG with fixed directions, entails that the lobes of the SRBFs will have sub-optimal directions, as can be seen in Figure 5.7. Theoretically, we are free to choose the directions, amplitudes, and sharpnesses of a set of SGs to find an optimal set of parameters, and indeed it is very much possible to do so, using methods such as GD or EM. The problem lies when we want to estimate the parameters for many sets of SGs that are not independent, if we want to utilize e.g., bi-linear filtering of the parameter samples, as there is no guarantee for the ordering of parameters.

Solving a system of interdependent sets of SG parameters is far from trivial, and in this paper we present a method that solves this optimization problem. Moreover, we also introduce *illumination-weighted environment visibility* in order to remedy the limitations of using a pre-convolved environment map.

5.4.2 Method

We first create a unique *uv*-parameterization of the scene, in which for each texel a 2D *light-field image*[4] is computed and stored on disk. Realising that *neural networks* characteristically will produce similar outputs given similar inputs, we can train a *convolutional neural network*(CNN) to estimate the parameters of the SGs, which thus will be implicitly ordered since adjacent light-fields will also be similar. We achieve this by simply evaluating the SGs to predict the light-field image and backpropagating the error through the network. The parameters are then stored in 2D textures allowing hardware texture lookups and filtering.

Our illumination-weighted environment visibility is computed, for each texel, by taking the ratio of the pre-convolved environment map with and without visibility. Using the same principle as the light-field, this is also approximated as a set of SGs, allowing a high-quality, real-time estimation of the reflected light by evaluating the SGs in the direction of interest and multiplying the result with the pre-convolved environment map.

5.4.3 Contributions

In this paper, we show that introducing extra degrees of freedom for the SG axis and sharpness, in combination with the illumination-weighted environment visibility, we can (with fewer lobes) significantly improve rendering quality and performance, for static scenes, compared to previous methods. We achieve good quality global-illumination renderings in full HD in 1-2ms, using a RTX 2080 graphics card.

[4]Typically $128x128$ half float RGB images, for all directions in our implementation.

Chapter 6

Discussion and Future Work

Figure 6.1: Using SGs for view dependent radiance and visibility in a DAG of resolution 256^3.

The desire for more complex and performant 3D representations will probably not fade any time soon. One recent example is Unreal Engine's new vir-

tualized geometry system, named NANITE[1], which allows multiple orders of magnitude increase in geometry complexity. While this is a triangle-based approach, there is also active research in DAGs, such as lossy compression [17], interactive editing [4], or hybrid solutios such as *Merged Multiresolution Hierarchies* [34]. Furthermore, there has been a surge of papers, inspired by *Neural Radiance Fields* (NeRF) [24], where a neural network is trained to synthesize novel views given a sparse set of images. As an example, *PlenOctress* [44] proposes a more efficient variant of NeRF where they pre-tabulate the NeRF into an octree-based radiance field and store the view dependence as spherical harmonics. In the paper, they state that the full models are of order 2GB in size for a sparse 512^3 grid, which they then reduce by a factor of $20\times - 30\times$ by a more aggressive thresholding, downsampling, and quantization. Regarding limitations, they admit that their octree-based representation is much larger than that of NeRF, which inspires us about the possibility to instead use what we have learned from the included papers in this thesis for a performant and more compact representation. In fact, a similar attempt was made, where we used a modified CNN from **Paper III** to approximate outgoing radiance and visibility and employ **Paper II** for fast filtered look-ups. While showing promising results, the paper is not yet finished and is hence omitted (see Figure 6.1 for preliminary results).

Recently, both Google[2] and NVIDIA[3] have shown interest in free viewpoint teleconferencing. However, due to the complexity and constraints for real-time streaming, they either rely on custom hardware and/or limiting the area to faces only. With new research from Rasmusson et al. [31] and Muller et al. [25] that significantly improves on the convergence over NeRF, previous research for time varying DAGs [12], and the included papers in this thesis, we start to see the possibility for general real-time streaming of radiance fields on consumer level hardware.

In 2019, Disney released the assets required to render a scene from the island in the movie Moana (2016)[4]. The uncompressed data for only the geometry is a staggering 93GB, and including animations adds another 131GB. Thus, it would also be worth investigating how well DAG and attribute compression would perform, as well as investigating new use cases such as caching view-dependent properties to speed up authoring, lookdev, or the final rendering.

[1]https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite/
[2]https://blog.google/technology/research/project-starline/
[3]https://nvlabs.github.io/face-vid2vid/
[4]https://www.disneyanimation.com/resources/moana-island-scene/

Bibliography

[1] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. *Computer Graphics Forum*, 33(6):220–227, 2014.

[2] David Benson and Joel Davis. Octree textures. *ACM Trans. Graph.*, 21(3):785–790, July 2002.

[3] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, oct 1976.

[4] V. Careil, M. Billeter, and Elmar Eisemann. Interactively modifying compressed sparse voxel representations. *Computer Graphics Forum*, 39:111–119, 05 2020.

[5] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4), jul 2017.

[6] Bas Dado, Timothy R Kol, Pablo Bauszat, Jean-Marc Thiery, and Elmar Eisemann. Geometry and attribute compression for voxel scenes. In *Computer Graphics Forum*, volume 35, pages 397–407. Wiley Online Library, 2016.

[7] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic Parameterizations of Surface Meshes. *Computer Graphics Forum*, 2002.

[8] Paul Green, Jan Kautz, Wojciech Matusik, and Frédo Durand. View-dependent precomputed light transport using nonlinear gaussian function approximations. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 7–14, New York, NY, USA, 2006. Association for Computing Machinery.

[9] Konstantine I Iourcha, Krishna S Nayak, and Zhou Hong. System and method for fixed-rate block-based image compression with inferred pixel values, September 21 1999. US Patent 5,956,431.

[10] Kei Iwasaki, Wataru Furuya, Yoshinori Dobashi, and Tomoyuki Nishita. Real-time rendering of dynamic scenes under all-frequency lighting using integral spherical gaussian. *Computer Graphics Forum*, 31(2pt3):727–734, 2012.

[11] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[12] Viktor Kämpe, Sverker Rasmuson, Markus Billeter, Erik Sintorn, and Ulf Assarsson. Exploiting coherence in time-varying voxel data. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16, page 15–21, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Trans. Graph.*, 32(4):101:1–101:13, July 2013.

[14] Andrei Khodakovsky, Nathan Litke, and Peter Schröder. Globally smooth parameterizations with low distortion. *ACM Trans. Graph.*, 22(3):350–357, July 2003.

[15] Gary King. *Real-time computation of dynamic irradiance environment maps*, volume 2, pages 167–176. Addison-Wesley Professional, 2005.

[16] Martin Kraus and Thomas Ertl. Adaptive texture maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 7–15, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[17] Remi Laan, Leonardo Scandolo, and Elmar Eisemann. Lossy geometry compression for high resolution voxel scenes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3:1–13, 04 2020.

[18] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[19] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[20] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *GPU Gems 2*, chapter Octree textures on the GPU, pages 595–613. Addison-Wesley Professional, 2005.

[21] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, July 2002.

[22] Josiah Manson and Peter-Pike Sloan. Fast filtering of reflection probes. *Computer Graphics Forum*, 35(4):119–127, 2016.

[23] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. An efficient denoising algorithm for global illumination. In *ACM SIGGRAPH / Eurographics High Performance Graphics*, page 7, July 2017. HPG 2017.

[24] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[25] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *arXiv:2201.05989*, January 2022.

[26] Anthony E Nocentino and Philip J Rhodes. Optimizing memory access on gpus using morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 18. ACM, 2010.

[27] Jorn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 105–114. Eurographics Association, 2012.

[28] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890.

[29] Roi Poranne, Marco Tarini, Sandro Huber, Daniele Panozzo, and Olga Sorkine-Hornung. Autocuts: Simultaneous distortion and cut optimization for uv mapping. *ACM Trans. Graph.*, 36(6):215:1–215:11, November 2017.

[30] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 497–500, New York, NY, USA, 2001. Association for Computing Machinery.

[31] Sverker Rasmuson, Erik Sintorn, and Ulf Assarsson. PERF: performant, explicit radiance fields. *CoRR*, abs/2112.05598, 2021.

[32] Feynman Richard. Qed: The strange theory of light and matter, 1985.

[33] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, pages 146–155, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[34] Leonardo Scandolo and Elmar Eisemann. Directed acyclic graph encoding for compressed shadow maps. In *High Performance Graphics*. ACM, 2021.

[35] Nico Schertler, Daniele Panozzo, Stefan Gumhold, and Marco Tarini. Generalized motorcycle graphs for imperfect quad-dominant meshes. *ACM Trans. Graph.*, 37(4):155:1–155:16, July 2018.

[36] Alla Sheffer, Bruno Lévy, Maxim Mogilnitsky, and Alexander Bogomyakov. Abf++: Fast and robust angle based flattening. *ACM Trans. Graph.*, 24(2):311–330, April 2005.

[37] Peter-Pike Sloan. Stupid spherical harmonics (sh) tricks. In *Game developers conference*, volume 9, page 42, 2008.

[38] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21(3):527–536, jul 2002.

[39] Olga Sorkine, Daniel Cohen-Or, Rony Goldenthal, and Dani Lischinski. Bounded-distortion piecewise mesh parameterization. In *Proceedings of the Conference on Visualization '02*, VIS '02, pages 355–362, Washington, DC, USA, 2002. IEEE Computer Society.

[40] Yu-Ting Tsai and Zen-Chung Shih. All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. *ACM Trans. Graph.*, 25(3):967–976, jul 2006.

[41] Jiaping Wang, Peiran Ren, Minmin Gong, John Snyder, and Baining Guo. All-frequency rendering of dynamic, spatially-varying reflectance. *ACM Trans. Graph.*, 28(5):1–10, dec 2009.

[42] Brent Robert Williams. Moxel dags: Connecting material information to high resolution sparse voxel dags. *Master thesis*, 2015.

[43] Kun Xu, Wei-Lun Sun, Zhao Dong, Dan-Yong Zhao, Run-Dong Wu, and Shi-Min Hu. Anisotropic spherical gaussians. *ACM Transactions on Graphics*, 32(6):209:1–209:11, 2013.

[44] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021.

[45] Cem Yuksel, Sylvain Lefebvre, and Marco Tarini. Rethinking texture mapping. *Computer Graphics Forum (Proceedings of Eurographics 2019)*, 38(2):535–551, 2019.

[46] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27, January 2005.