Exploiting Coherence in Time-Varying Voxel Data

Viktor Kämpe[1] Sverker Rasmuson[1] Markus Billeter[12] Erik Sintorn[1] Ulf Assarsson[1]

[1]Chalmers University of Technology [2]VMML, University of Zürich

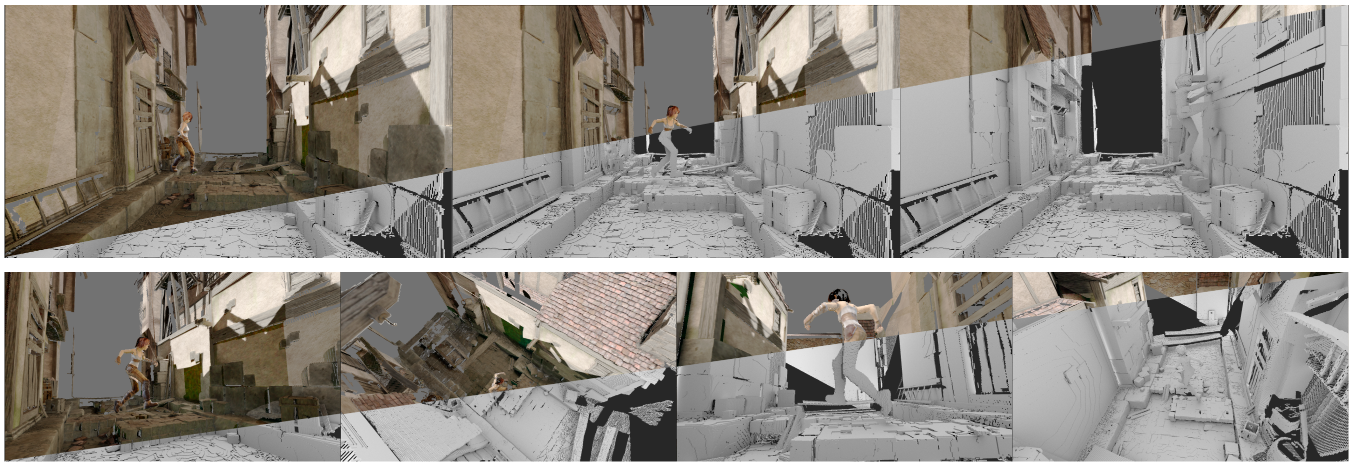


Figure 1: *We introduce an efficient encoding of time-varying binary voxel data, the temporal DAG, which we use as the geometric representation for free viewpoint video. The geometry of this sequence consists of 70 frames of voxel data at a spatial resolution of* 2048^3*. Encoded as a temporal DAG, the memory consumption is only 1.86 MBytes. The top row of images shows three different time steps from a single novel viewpoint, and the bottom row shows four additional views of the second time step. The geometry is visualized with ambient occlusion and with colors reconstructed from four color rgb-camera streams.*

Abstract

We encode time-varying voxel data for efficient storage and streaming. We store the equivalent of a separate sparse voxel octree for each frame, but utilize both spatial and temporal coherence to reduce the amount of memory needed. We represent the time-varying voxel data in a single directed acyclic graph with one root per time step. In this graph, we avoid storing identical regions by keeping one unique instance and pointing to that from several parents. We further reduce the memory consumption of the graph by minimizing the number of bits per pointer and encoding the result into a dense bitstream.

Keywords: time-varying, voxel grid, directed acyclic graph, free viewpoint video

Concepts: •**Computing methodologies** → **Computer graphics;** *Volumetric models;*

1 Introduction

Geometry scanned by, for instance, depth cameras can be represented in a raw point-sample format but the points are often processed to produce a surface representation. Dense voxel grids, and two-level grids, have been demonstrated as good geometric representations in surface reconstruction methods [Kazhdan et al. 2006; Izadi et al. 2011; Chen et al. 2013; Nießner et al. 2013]. While being appropriate for surface-reconstruction methods, they consume too much memory, especially at high spatial resolutions, to be a viable option for streaming and storage of reconstructed geometry. For time-varying geometry, with a reconstructed surface per time step, the memory consumption becomes even more infeasible. A commonly used method to reduce the memory consumption of grids is to exploit coherence in the data.

Sparseness is one type of spatial coherence which can be utilized to efficiently represent large uniform (or empty) regions [Meagher 1982]. Translational coherence is another type of spatial coherence, which can be used to encode regions that are identical under spatial translation [Kämpe et al. 2013].

Temporal coherence can be applied to time-varying data to efficiently store regions that are very similar in different time steps. Difference coding, for instance, considers consecutive time steps and can be combined with sparse spatial encoding [Ma and Shen 2000].

In this paper, we further increase the amount of coherence possible to exploit by searching within, as well as between, time steps to encode only the regions of the time-varying voxel grid that are unique under spatio-temporal translation. This allows us to encode regions as identical where previous methods cannot, e.g, two regions at both different spatial position and different (not necessarily adjacent) time steps. We encode the voxel grids as a single directed acyclic graph (DAG), where two identical regions are encoded by pointing out the same, uniquely stored, subgraph. We keep a start node per time step in the DAG, and traversing the structure from a start node is identical to traversing an octree from the root node. Surfaces are stored in this common structure, whether they are static or dynamic.

I3D '16, February 27-28, 2016, Redmond, WA
ISBN: 978-1-4503-4043-4/16/03
DOI: http://dx.doi.org/10.1145/2856400.2856413

The coherence automatically reduces the memory consumption of geometry that, for instance, is alternating between being static and dynamic, or is composed of both static and dynamic parts. The coherence is not encoded with explicit annotation in the nodes of the DAG and the traversal of the structure, e.g., during ray tracing, is just as simple as for a static octree.

We believe that a compact voxel representation of time-varying surface data has numerous applications. We mainly target a geometry representation for free viewpoint video (FVV), but the proposed method is independent of the origin of the voxel data and the purpose of playback. We believe that our structure is a suitable alternative for FVV due to: 1) simple traversal, regardless of encoded coherence, 2) no restriction on topology, and 3) low memory consumption. Avoiding coding of surface topology allows, for instance, capture of scenes with difficult and continuously changing topologies that are very hard to encode efficiently with triangle meshes. Good memory performance is of crucial importance since an unrestricted camera means that the surface can be viewed from an arbitrary direction and from an arbitrary distance, which makes the demand for resolution virtually insatiable.

We also introduce a compression step that encodes the DAG to a non-traversable state for streaming or storage. The compression and decompression is very fast and reduces the memory consumption further by a factor of 2-3.

2 Related Work

A comprehensive overview of work related to time-varying geometry is out of the scope of this work. In this section, we mainly focus on time-varying voxel grids and only briefly compare to other commonly used geometric representations like triangle meshes and point clouds.

2.1 Coherence in Voxel Grids

Geometry stored as a dense grid has a predictable, but very high, memory consumption. An octree allows for spatially homogeneous regions to be encoded without further subdivision. Efficient encoding of uniform regions has been used extensively, and the special case when uniform regions are restricted to empty regions, is often referred to as sparse voxel octrees (SVO). Kämpe et al. [2013] exploit that surfaces in a voxel grid exhibit many identical regions at different spatial locations, and encode the grid in a directed acyclic graph that only needs to store the unique regions.

Time-varying voxel grids can be encoded with a difference encoding to avoid storing regions that change very little between consecutive frames. Ma and Shen [2000] combine octree encoding of individual time steps with difference encoding by identifying spatial regions that are identical in two or more consecutive time steps. The subtree of such a spatial region is only stored for the first frame and in each consecutive time step, until the region alters, the region is encoded with a single pointer to the subtree.

2.2 Encoding of Sparse Voxel Octrees

The encoding of SVOs is often adapted to specific use cases. To reduce the memory consumption of SVOs, a single pointer per node can point to consecutively stored children. By exploiting locality of references, the majority of the pointers can be encoded with 2 bytes while still maintaining traversability [Laine and Karras 2010]. The SVO can be uniquely represented even without pointers by storing the sparseness information for each node (the bits indicating whether a child is empty or not) in a well defined order, e.g., breadth-first-order [Schnabel and Klein 2006], but before reconstruction of pointers it is only possible to traverse the SVO in that pre-defined order.

2.3 Depth Maps

In depth-image–based rendering, the geometry is represented by depth maps. For time-varying geometry, the stream of depth images can be encoded with methods similar to conventional video codecs. Pece et al. [2011] distribute 16-bit depth values in three 8-bit channels (rgb) to reduce the error when using lossy compression with available codecs for conventional video. The widespread availability of conventional video decoders would make it easy to decode such a depth stream even in hardware, but the codecs are not designed specifically for depth images. Müller et al. [2013] present an extension to the high efficiency video coding (HEVC) for multi-view depth streams to compress blocks of depth values in and between views combined with exploitation of temporal coherence and evaluate with the MPEG benchmark for auto-stereoscopic displays. The benchmark consisting of sequences shot from either two or three depth cameras mounted a short distance apart in a linear array and facing in the same direction [MPEG 2011].

While streaming of multi-view depth has been demonstrated for setups of similar views, it has, to our knowledge, not yet been demonstrated to scale sublinearly with the number of views for a general setup with many overlapping views from widely different positions and with different orientations. We believe that a native 3D representation will have advantages in simplicity and storage when it comes to avoiding redundant encoding of surfaces seen from multiple views. Native 3D representations also allow for reconstructed surfaces that, for instance due to depth complexity, cannot be represented by a small number of depth maps.

2.4 Triangle Meshes

Triangle meshes can be very efficiently rendered due to rasterization hardware in GPUs. There are also numerous tools to create, manipulate and simplify triangles meshes. Artist generated content is often extended with animation rigs, which simplify the animation procedure and require very little animation data to be stored, or transmitted, per frame. Animated meshes that are captured by, e.g., depth cameras can however be very costly to store and transmit as all vertex data must be encoded, each frame. Lengyel [1999] encodes time-varying meshes without rigging information by clustering vertices and matching their trajectories to affine transforms, which is shown to work well for meshes of constant topology when the vertex trajectories are from skeletal animation, but not as well when fitted to captured data. Beeler et al. [2011] assume surfaces for which a single mesh topology can be defined and used for all frames, and show how that single mesh can be fitted to the captured geometry of each frame. With over 1M vertices per frame, streaming vertex positions at 24 fps still consumes a considerable amount of bandwidth. Collet et al. [2015] allow mesh topology to change at keyframes, and quantize the per-vertex data for intermediate frames to 16bits; This data is then compressed with run-length encoding. The system isolates the performing actor from the background and creates reconstructed meshes of only 10k-20k triangles by identifying perceptually important areas (e.g. faces and hands). This data can be compressed to require 4-8Mbps. In cases where single characters cannot be isolated, where motifs of higher geometric complexity are captured, or where a partially dynamic background must be captured, the amount of triangles required can be much higher (a frame in an animated feature film, for instance, usually contains several million polygons) and the bandwidth required will increase proportionally.

3 The Temporal DAG

In the first part of this section, we describe how we find coherence in a sequence of sparse voxel octrees and how we encode this in a single directed acyclic graph with a root per time step. While this actually allows time-varying resolutions without modifications, we will assume that the grid resolutions are identical for all time steps. In the second part, we describe how we reduce the number of bits required for the pointers in the DAG.

3.1 Coding Coherence

To find coherence, we extend the method presented by Kämpe et al. [2013] to multiple time steps by keeping a root per time step. Instead of just searching for spatial coherence within a single time step, we search for coherence within all time steps. Even though the input data is different, the method is the same. We search for coherence in a bottom-up method by arranging the nodes of the SVOs (of all time steps) in a list per level and processing a level at a time. I.e., level 0 will contain all the SVO root nodes, in time-sequential order. Similarly, level n contains all the SVO nodes at levels n for all time steps. The order is insignificant for correctness, as long as validity of child pointers (max 8 per node) is ensured.

We search in this new SVO for all subtrees that are identical except for their spatial or temporal position. Since the spatial position in an SVO is implicit by the path taken during traversal, and the time step is implicit by the root we start traversal from, the problem is reduced to only finding identical nodes in the node lists of each level.

Identical nodes are found by sorting the list of nodes. Each node consists of a child mask (8 bits) to indicate whether the eight children are containing geometry or not, and eight pointers (8×32 bits) to point out the children, and the comparison operator for sorting simply compares this data. After sorting, all identical nodes are adjacent in the list, and extracting the set of unique nodes becomes trivial. The nodes of the parental level are then updated to point to the corresponding unique nodes. In a tree, all pointers are unique, but after the pointer update, nodes in the parental level may become identical, and so we proceed to extract the set of unique nodes one level at the time. When the top level is reached, we have encoded all coherence within each frame as well as between frames.

To avoid the peak memory demand of keeping all unreduced SVOs in memory, the reduction can be applied on subsets of nodes before making a final reduction that produces the same final DAG. Kämpe et al. [2013] apply the reduction on a spatial subregion at a time, before the final reduction. With time-varying voxel data, it is reasonable to assume that the construction of SVOs will happen a frame at a time, and therefore we first apply the reduction per frame and then we merge the per-frame-DAGs into the final DAG. The final merging can be performed on all frames at once, which requires the least work. Another option is to do the final merging progressively, merging the nodes of a frame directly into the final DAG. The progressive merging does more work, but has an even smaller peak memory demand and enables progressive capture and processing of geometry.

When all coherence is found, and the topology of the DAG is fixed, we compact the DAG by removing allocated pointers for non-existing children. We store the nodes consecutively in memory with a single child mask, padded to a 32bit word, followed by the 32bit pointers for each existing child. Whenever we traverse the nodes of the DAG in a non-predefined order, e.g., while ray casting, this is the format we use.

3.2 Compression

When memory consumption is more important than traversal speed, e.g., during streaming and storage, we compress the DAG into a dense bitstream where the number of bits per pointer is greatly reduced. While the DAG is compressed, it can only be traversed in a pre-defined order, but a decompression step recovers the traversable format again.

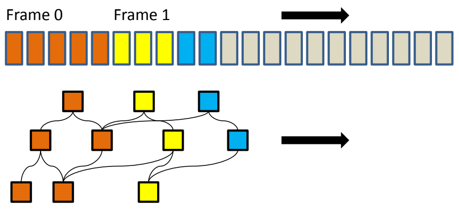


Figure 2: *Frame-first ordering of nodes. All nodes of a frame are stored before the nodes of the next frame.*

3.2.1 Variable Pointer Size

In preparation for decreasing the size of pointers, we rearrange the nodes in chronological order based on the time step in which they are first referenced; they may be referenced in several time steps (see Figure 2). One good property of the chronological order is that the streaming of nodes happens in the same order as they will be requested during playback. Another good property is that it lets us determine a subset of the nodes that a particular pointer may point to, since nodes will not have children in a future frame, thus allowing us to code the pointer with fewer bits:

$$\text{bits per pointer} = \lceil \log_2(\#\text{nodes in subset}) \rceil$$

We further reduce the subset of nodes by sorting the nodes within each time step in breadth first order (see Figure 3). Since nodes are restricted to point to the level below, it is now easy to determine a dense range of nodes that the pointers may reference. The needed number of bits per pointer will vary per frame and level, and we provide the sizes as integers in a header to the bitstream.

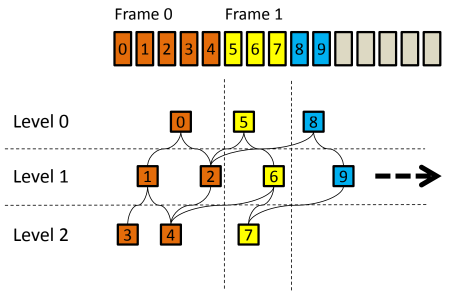
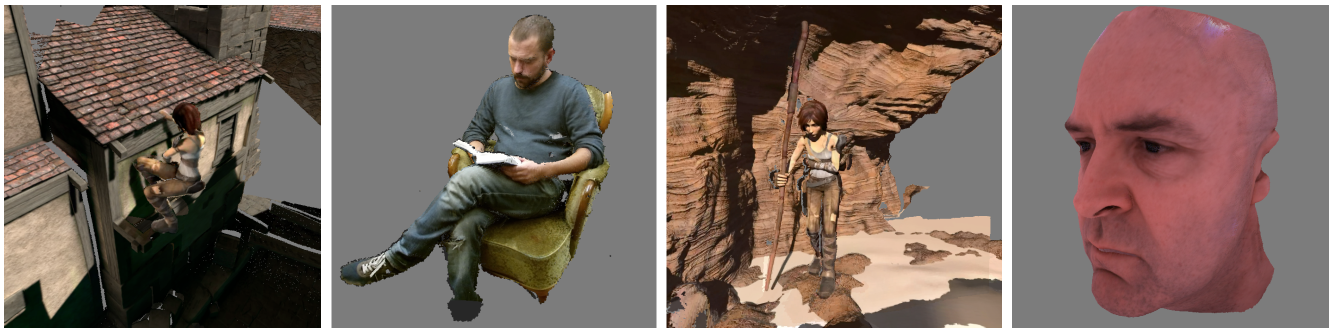


Figure 3: *We sort the nodes in frame-first order and within each frame in breadth-first order. This gives a well-defined order, and we exploit this to reduce the size of pointers.*

3.2.2 Implicit Pointers

When a pointer value is restricted to a subset containing only one node, that pointer value is implicit. We code two types of pointers implicitly: pointers that are the first reference to a new node and pointers that are identical in the previous frame. When we find an opportunity to encode a pointer implicitly, we replace the original child mask (8-bit) with an extended child mask (16-bit) that encodes each child to one of the following:



ALLEY : 2048^3 grid, 70 frames
source: 4 virtual static cameras

KINECT : 512^3 grid, 480 frames
source: 3 Kinect cameras

BEAST : 1024^3 grid, 213 frames
source: 5 virtual moving cameras

FACE : 1024^3 grid, 347 frames
source: triangle meshes
2.4M tris (1.2M verts) per frame

Figure 4: *Test sets for evaluation. All images are raytraced using our DAG data structure. The shading is a convex combination of colors sampled in the original camera shots. The sample positions are determined by projecting the primary hit onto each camera plane, and the color samples are weighted by a cosine factor (for the angle between the primary ray and the camera direction) and a binary visibility factor (determined by ray tracing).*

e_0 :	Empty
e_1 :	Occupied and identical to previous frame
e_2 :	Occupied and reference to new node
e_3 :	Occupied and explicit pointer

To distinguish between normal 8-bit child masks and 16-bit extended child masks, we start each node in the bitstream with a bit that indicates the type of child mask.

Pointer values that are encoded as ”reference to new node” are easy to convert into explicit pointers by keeping track of the first non-referenced node in each level. Whenever an implicit pointer to a new node is encountered during linear traversal of the bitstream, it refers to the first non-referenced node of the level below, due to the pre-defined order.

Pointer values that are encoded as ”identical to previous frame” are well defined in a tree, due to the one-to-one mapping between regions and nodes, and the implicit pointer value can be substituted with the pointer to the node of the corresponding spatial region in the previous frame. In a directed acyclic graph, a node can be reached by several different traversal paths, and hence it can describe several regions in a single frame. Traversing to these regions in the previous frame may result in different nodes and, therefore, a more rigorous definition is required. One solution is to enumerate all possible traversal paths from the root (of the current frame) that end up in a specific node, and specify which of the enumerated paths to use to recover the implicit pointer value in the previous frame. The number of possible paths can be vastly different, and specifying the number of bits for the enumeration plus the actual enumeration may unfortunately consume more bits than we seek to save. We choose a simplified version, where we enumerate the paths according to breadth-first order, and only encode a pointer implicitly when recovery is possible with the first path to the node. The recovery is then also simplified by our bit stream being stored in breadth-first order, which means that our linear traversal of the bitstream corresponds to breadth-first traversal.

4 Results

We compare the performance of our temporal DAG to storing a separate DAG per frame, to storing a separate SVO per frame, and to a difference tree coding [Ma and Shen 2000]. First, we evaluate the ability to code coherence by the number of resulting nodes. Secondly, we evaluate the final memory consumption, which is affected by the encoding of the nodes.

Our test data consist of four sequences of time-varying voxel grids, representing different use cases (see Figure 4). The ALLEY and BEAST sets are produced from an open source movie project by the Blender foundation by combining point clouds from several virtual cameras into SVOs. We produce the point clouds by rendering depth maps of resolution 2048×2048 from a few very different viewpoints (see auxiliary video). For the third sequence, FACE, we voxelize triangle meshes (one per frame) of a facial performance, captured and reconstructed by Beeler et al. [2011]. Finally, for the KINECT sequence, we have captured a performance using three Kinect (version 2) cameras simultaneously. The obtained depth streams (512x424) were de-noised (temporally and spatially) using two simple bilateral filters and the resulting point clouds were cropped to a world-space bounding box. The average number of points inserted per frame is 126k.

Table 1: *Node Count of the first frame, consecutive frames, and in total.*

	First frame		Average in consecutive frame				Total			
	Thousands of Nodes		Thousands of Nodes				Millions of Nodes			
	Trees	DAGs	SVO	Diff. Tree	DAG	Temp. DAG	SVO	Diff. Tree	DAG	Temp. DAG
ALLEY	1340	150	1340	17.4	150	3.45	93.7	2.54	10.4	0.388
KINECT	24.1	5.91	24.6	8.68	6.00	2.58	11.8	4.18	2.88	1.24
BEAST	359	60.1	316	294	54.5	33.4	67.4	62.7	11.6	7.15
FACE	674	74.4	670	518	70.9	30.8	232	180	24.6	10.7

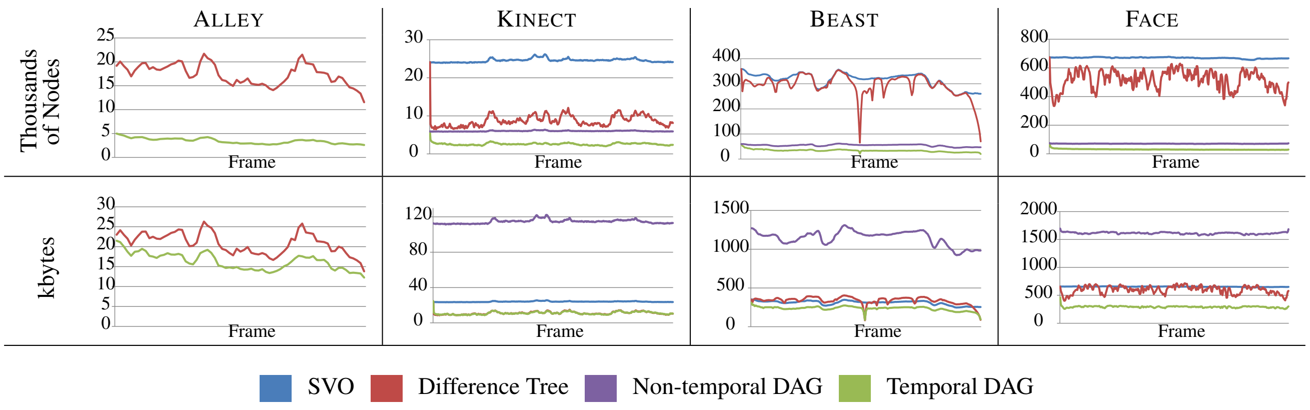


Figure 5: *Distribution of nodes and memory consumption over the frames. In* ALLEY*, the SVO, the non-temporal DAG, and the inital frame is omitted due to orders of magnitude difference in values (see Table 1 and 2).*

4.1 Coherence

Our temporal DAG utilizes a superset of the coherence of the difference tree, which in turn utilizes a superset of the coherence of an SVO. The node count of the temporal DAG will therefore always be less (or equal) to that of a difference tree, and the node count of a difference tree will always be less (or equal) to that of an SVO per frame (see Table 1 for total number of nodes).

Both the difference tree and the temporal DAG can exploit the temporal coherence of geometry that is mainly static, while the SVOs and non-temporal DAGs cannot. This shows in the first two data sets, ALLEY and KINECT. ALLEY contains only static geometry except for a moving character and a swinging bucket hanging in a rope. The SVOs and the non-temporal DAGs consume a nearly constant number of nodes ($\sim$1.3M and 150k) per frame (see Table 1). The difference tree and the temporal DAG both consume a large number of nodes in the first frame (1.3M and 150k nodes) but they require significantly less nodes for the consecutive frames (17k and 3.5k nodes on average) due to the abundance of static geometry. The KINECT sequence shows a man in a chair, reading aloud from a book and gesturing. The torso, legs and chair are stationary (except for noise), while the head, arms and hands are dynamic. The temporal DAG and the difference tree exploit the temporal coherence and need significantly fewer nodes for consecutive frames than they need for the first frame. Again, the SVO consumes the same amount of nodes per frame throughout the sequence.

When the data sets do not contain much static geometry, the amount of coherence in the difference tree coding is significantly reduced. The third data set, BEAST, contains two characters moving through a static landscape, but the voxel grid is moving with the characters, making the whole world move in the voxel grid. The difference tree only removes a few percent of the nodes compared to an SVO per frame. The temporal DAG has almost an order of magnitude fewer nodes, which shows that there exists a lot of coherence even in the dynamic geometry.

The fourth data set, FACE, is captured with static cameras but the reconstructed surface is never static, on a macro scale due to the dynamic facial performance and on a micro scale due to noise in the capture and surface reconstruction. The difference tree very often consumes the same amount of nodes as an SVO, but occasionally some coherence can be found (see Figure 4). The number of nodes in the temporal DAG is about 5% compared to the SVO and the difference tree.

4.2 Memory Consumption

We compare the overall memory consumption of our temporal DAG to a DAG per frame (that only exploits spatial coherence), SVOs, and difference coded trees. We encode the non-temporal DAG with the layout proposed by Kämpe et al. [2013] that requires 4 bytes per child mask and 4 bytes per pointer. We encode the SVOs with only implicit pointers to new nodes, which require 1 bit per child to indicate if it exists or not. The SVO then consumes 8 bits per node, resulting in a structure similar to that presented by Schnabel and Klein [2006]. We encode the difference tree with our own method, since Ma and Shen [2000] do not provide implementation details on how they encode individual nodes, which is necessary for a comparison of memory consumption. We use a simplified version of our DAG encoding for the difference tree, where we omit all explicit pointers and use the implicit pointers throughout. The enumeration per child then becomes:

e_0 :	Empty
e_1 :	Occupied and identical to previous frame
e_2 :	Occupied and reference to new node

Enumerating each child individually would require 2 bits, resulting in 16 bits per node. We instead enumerate the child mask per node with $3^8 = 6561$ enumerations which only requires 13 bits per node. For leaf nodes, each child is either set or not set and we encode them with an 8-bit child mask instead. We also use an 8-bit child mask for internal nodes of the first frame since there is no previous frame.

For mostly static geometry, in ALLEY and KINECT, both the difference tree and the temporal DAG show similar memory consumption. Both are good at finding the static geometry, but the temporal DAG also finds spatial coherence and temporal coherence of non-adjacent frames and, for ALLEY, the temporal DAG consumes 1.9 MByte compared to 2.7 MByte for the difference tree (see Table 2). The non-temporal DAG and the SVO cannot exploit temporal coherence and therefore is much more memory intensive with 199 Mbyte and 89 Mbyte of memory consumption.

For fully dynamic environments, like BEAST and FACE, the difference tree is close to the SVO in memory performance. The node count of the difference tree is always less (or equal), but the slightly larger nodes, averaging 8-10 bits instead of 8 bits, makes the memory consumption vary from slightly lower to slightly higher. The non-temporal DAG averages 160-190 bits per node which is too high to be amortized by the available amount of spatial coherence per frame. The temporal DAG consistently has the best overall memory performance, even though its nodes requires considerably more bits

Table 2: *Total memory consumption and bit rate when streaming 24 frames per second.*

		ALLEY	KINECT	BEAST	FACE
Mbyte	SVO	89.3	11.2	64.3	222
	Diff. Tree	2.68	5.18	70.0	203
	DAG	199	53.7	239	547
	Temp. DAG	1.86	5.15	48.8	99.3
Mbit/s	SVO	245	4.50	57.9	123
	Diff. Tree	7.35	2.07	63.1	112
	DAG	545	21.5	215	302
	Temp. DAG	5.10	2.06	44.0	54.9

than the tree nodes with an average of 40-80 bits per node. The ability to encode coherence comes with a cost per node, but the overall memory consumption is more than compensated by the reduction of nodes.

4.3 Encoding and Decoding Performance

The time required for reducing the input data (one SVO per frame) to a single DAG depends on input geometry, grid resolution, and the total number of frames. For grid resolutions of 1024^3, the reduction takes in the order of 1 second per frame for an unoptimized single-thread CPU implementation. The time taken for pointer compression (Section 3.2) is negligible in comparison. Pointer decompression is performed as a linear sweep over the data read from disk with a cheap conversion from implicit to explicit pointers (see Table 3).

Table 3: *Single threaded decode timings for temporal DAG in milliseconds on an Intel Core i7 2630QM at 2GHz.*

	ALLEY	KINECT	BEAST	FACE
Unpacking bitstream	51.4	185	1000	1540
Implicit RTNN	12.6	58	248	416
Implicit RITPF	30.6	205	590	973
Total	99.7	543	1850	2940

RTNN: Reference to new node.
RITPF: Reference identical to previous frame.

5 Conclusion and Future Work

We find a significant amount of coherence in time-varying voxel grids and encode the coherence with a single directed acyclic graph. The DAG requires us to store a large number of pointers, but we reduce the number of bits per pointer and show that many of the pointer values can be stored implicitly. This makes the memory performance of the DAG superior to voxel representations that encode less coherence, for all data sets we have tested. There are, of course, pathological cases where the coherence will not be sufficient to compensate for the higher memory consumption per node.

For longer sequences of time-varying voxel data, a single directed acyclic graph may require too much working memory, since we accumulate more and more nodes which monotonically increase the memory consumption for each frame. Similarly to conventional video codecs, the entire sequence may be divided into many shorter clips that can be encoded with separate DAGs. This limits the amount of nodes to keep in memory, but increases the total number of nodes in the entire sequence. More elaborate methods of composing several shorter clips could selectively keep, discard, or copy nodes from a previous clip to a new clip. Such methods could be a superset of keyframe based approaches. The formulation of such a strategy is left for future work.

We have shown our method to be feasible for applications such as (streamable) free viewpoint video (FVV), where difficult surface topologies and a combination of static and dynamic content is common, and where the memory requirements are high. This has been done both for artificial and recorded data.

Only lossless encoding has been considered in this work. To guarantee a limited bit rate, for arbitrary data, the compression has to be lossy. Noisy input data will lessen the effectiveness of the compression, and pre-filtering might be necessary to hit a specific bit rate. The interpretation of lossy in the context of a temporal DAG can be rather different compared to an SVO and a difference tree, and this is also a direction of future work that we consider.

Finally, we note that little effort has gone into optimizing our method for speed, and it is not yet fast enough for real-time encoding, which would be a requirement for, e.g., a video conferencing application. This would also be an interesting area for further exploration.

Acknowledgements

This research was supported by The Swedish Foundation for Strategic Research under grant RIT10-003, and The Swedish Research Council under grant 2014-4559. Assets from the open movie project Sintel by the Blender Foundation was used to produce the ALLEY and BEAST datasets. The meshes for the FACE dataset was provided by Disney Research.

References

BEELER, T., HAHN, F., BRADLEY, D., BICKEL, B., BEARDSLEY, P., GOTSMAN, C., SUMNER, R. W., AND GROSS, M. 2011. High-quality passive facial performance capture using anchor frames. *ACM Trans. Graph. 30*, 4 (July), 75:1–75:10.

CHEN, J., BAUTEMBACH, D., AND IZADI, S. 2013. Scalable real-time volumetric surface reconstruction. *ACM Trans. Graph. 32*, 4 (July), 113:1–113:16.

CHHUGANI, J., AND KUMAR, S. 2007. Geometry engine optimization: cache friendly compressed representation of geometry. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics*, 9–16.

COLLET, A., CHUANG, M., SWEENEY, P., GILLETT, D., EVSEEV, D., CALABRESE, D., HOPPE, H., KIRK, A., AND SULLIVAN, S. 2015. High-quality streamable free-viewpoint video. *ACM Trans. Graph. 34*, 4 (July), 69:1–69:13.

IZADI, S., KIM, D., HILLIGES, O., MOLYNEAUX, D., NEWCOMBE, R., KOHLI, P., SHOTTON, J., HODGES, S., FREEMAN, D., DAVISON, A., AND FITZGIBBON, A. 2011. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '11, 559–568.

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4 (July), 101:1–101:13.

KAZHDAN, M., BOLITHO, M., AND HOPPE, H. 2006. Poisson surface reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SGP '06, 61–70.

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, ACM Press, 55–63.

LENGYEL, J. E. 1999. Compression of time-dependent geometry. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '99, 89–95.

MA, K.-L., AND SHEN, H.-W. 2000. Compression and accelerated rendering of time-varying volume data. In *Proceedings of the 2000 International Computer Symposium-Workshop on Computer Graphics and Virtual Reality*, 82–89.

MEAGHER, D. 1982. Geometric modeling using octree encoding. *Computer graphics and image processing 19*, 2, 129–147.

MÜLLER, K., SCHWARZ, H., MARPE, D., BARTNIK, C., BOSSE, S., BRUST, H., HINZ, T., LAKSHMAN, H., MERKLE, P., RHEE, H., ET AL. 2013. 3d high efficiency video coding for multi-view video and depth data. *IEEE transactions on image processing*.

MPEG, 2011. Call for Proposals on 3D Video Coding Technology , March. ISO/IEC JTC1/SC29/WG11.

NIESSNER, M., ZOLLHÖFER, M., IZADI, S., AND STAMMINGER, M. 2013. Real-time 3d reconstruction at scale using voxel hashing. *ACM Trans. Graph. 32*, 6 (Nov.), 169:1–169:11.

PECE, F., KAUTZ, J., AND WEYRICH, T. 2011. Adapting standard video codecs for depth streaming. In *Proceedings of the 17th Eurographics conference on Virtual Environments & Third Joint Virtual Reality*, Eurographics Association, 59–66.

SCHNABEL, R., AND KLEIN, R. 2006. Octree-based point-cloud compression. In *Symposium on Point-Based Graphics 2006*, Eurographics.