Transforming Hierarchical Data Structures – a PSVDAG–SVDAG Conversion Algorithm

Branislav Madoš, Norbert Ádám

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9/A, 042 00 Košice, Slovakia
e-mail: branislav.mados@tuke.sk, norbert.adam@tuke.sk

Abstract: This paper examines the issues of domain-specific hierarchical data structures, based on directed acyclic graphs, dedicated to the representation of the geometry of three-dimensional scenes. In this paper, the authors introduce two versions (out-of-core and semi-out-of-core) of an algorithm to transform hierarchical data structures – pointerless sparse voxel directed acyclic graphs, into sparse voxel directed acyclic graphs. Pointerless sparse voxel directed acyclic graphs are not suitable for immediate traversing, due to the absence of pointers to the child nodes; however, they are suitable for archiving and streaming, as they have a more compact binary-level representation. Sparse voxel directed acyclic graphs, on the other hand, allow quick traversing during visualization or other forms of processing, since their nodes include pointers to child nodes. The disadvantage of this, is that the binary-level representation, requires more operating memory or secondary storage space. Both hierarchical data structures – sparse voxel directed acyclic graphs and pointerless sparse voxel directed acyclic graphs – and both versions of the proposed conversion algorithm are described in the first part of the paper. Results of tests, performed on various models – previously surface polygonal models, stored in the Wavefront Technologies geometry definition file format (OBJ) – now voxelized to the respective resolutions, are summarized in the second part of the paper. The binary-level representation lengths of both data structures, along with the time consumption of both versions of the proposed conversion algorithms, are detailed in the last part of the paper.

Keywords: pointerless sparse voxel octrees; PSVO; sparse voxel octrees; SVO; pointerless sparse voxel directed acyclic graphs; PSVDAG; sparse voxel directed acyclic graphs; SVDAG; hierarchical data structures; volume dataset; three-dimensional image; lossless data compression

1 Introduction

Hierarchical data structures (HDS), based on the use of octant trees (octrees) and Directed Acyclic Graphs (DAGs), are popular solutions to represent the geometry of three-dimensional scenes, especially if the scenes are voxelized from three-dimensional polygonal surface models [1]. In voxelized scenes, the uncompressed

geometry is represented as a three-dimensional regular grid, using 1 b per voxel – active (filled) voxels are encoded as "1" bits and passive (empty) voxels are encoded as "0" bits. In such scenes, most voxels are passive (often, these represent 99.99% of all voxels, or even more).

Hierarchical data structures, based on octrees and directed acyclic graphs, can cope with this sparsity and use it in lossless data compression, and that is why they can store scene geometry using unprecedentedly small space. A compression level of 10^{-5} bits per voxel (b/vox) can be achieved in case of a 64 K^3 scene (65536 × 65536 × 65536 voxels), using 315.3 MB of space to store the particular scene geometry [2] [3]. In comparison, to store this geometry in its uncompressed form, as a regular three-dimensional grid of 1b/vox, one needs 256 TB of space. Another common feature of voxelized scenes based on polygonal surface models is the high probability of occurrence of identical subspaces. Hierarchical data structures use this feature to achieve lossless compression, via Common Subtree Merging (CSM). Applying this technique, octrees are transformed into DAGs.

There are octree-based hierarchical data structures allowing space-saving representation of the scene geometry without having any child node pointers encoded in their nodes – these are Pointerless Sparse Voxel Octrees (PSVOs). Pointerless hierarchical data structures are suitable for archiving and streaming purposes, but are less suitable for immediate traversing. Their disadvantage is that they don't allow using the CSM technique. Sparse Voxel Directed Acyclic Graphs (SVDAGs) include pointers for quick traversing and CSM. Pointers, though, require a significant amount of space in SVDAGs. However, the possibility to use CSM can outweigh this disadvantage. That is why, in total, SVDAGs can be more space-efficient, compared to PSVOs.

In our previous research, we developed Pointerless Sparse Voxel Directed Acyclic Graphs (PSVDAGs) [4] – hierarchical data structures incorporating advantages of both PSVOs and SVDAGs [5]. Due to pointerless encoding, this data structure saves space, and, due the concept of variable-length Labels/Callers, it allows the use of CSM. PSVDAGs can exceed the compression ratio of PSVOs and SVDAGs. However, the absence of pointers makes this data structure unsuitable for quick traversing. That is why the PSVDAG data structure incorporates a feature – active child node count (ACHNC) – allowing quick transformation into SVDAGs. In this paper, we propose and describe two versions of the conversion algorithm allowing the transformation of PSVDAGs into SVDAGs: an out-of-core and a semi-out-of-core version.

The contribution of the paper is in the following:

The design of two versions of the PSVDAG – SVDAG conversion algorithm: an out-of-core and a semi-out-of-core version.

Section 2, hereof summarizes the related works in the field of multi-dimensional data linearization, hierarchical data structures used to represent the geometry of

three-dimensional scenes and out-of-core algorithms for the creation of those data structures. Due to the vast number of papers published in the field, only closely related works are mentioned.

Section 3, provides a description of the SVDAG and PSVDAG hierarchical data structures, not only their formal description using the Backus-Naur Form, but also a brief explanation of their fundamental properties.

Section 4, is an introduction to the main contribution of the paper, i.e. the out-of-core and semi-out-of-core versions of the conversion algorithm that transforms PSVDAGs into SVDAGs.

Section 5, describes the results of the tests performed on various three-dimensional scenes, originally stored in the Wavefront Technologies geometry definition file format (OBJ), subsequently voxelized to the respective resolutions and then stored as PSVDAGs. Using the algorithm described in Section 4, the transformation of their representation from PSVDAG into SVDAG was tested.

Section 6, the last section of the paper, draws conclusions from the test results stated in the previous section of the paper.

2 Related Work

Due to the vast number of papers published in this field, this section lists only the closely related works.

Linearization of multi-dimensional data. Space-Filling Curves (SFC), introduced by Peano and Hilbert at the end of the 19th Century [6] [7], are used to linearize multi-dimensional data. Very popular in computer graphics is the Morton order [8]. Hilbert Space Filling-Curves (HSFCs) are used in computer science for better locality preserving [9]. Examples of linearization of a two-dimensional space using Morton order and Hilbert SFCs of levels 1 and 2 are depicted in Figure 1.

Hierarchical data structures for three-dimensional data representation. Octrees – hierarchical data structures – have been used for representation of three-dimensional scenes for decades. Works from the 1980s include Srihari [10], Rubin and Whitted [11], Jackins and Tanimoto [12] and Meagher [13] [14] [15], to name only a few. Different encoding schemes of child node header tags in PSVO leveraging fixed-length and variable-length header tags appeared in [16].

Not only homogenously empty subtrees, but also homogenously filled subtrees were removed: PSVOs – hierarchical data structures allowing the removal of not only homogenously empty subtrees, but also those homogenously filled with any symbol from a defined set of symbols – were proposed in [17]. The symbol set can be encoded using a fixed-length or variable-length binary-level representation.

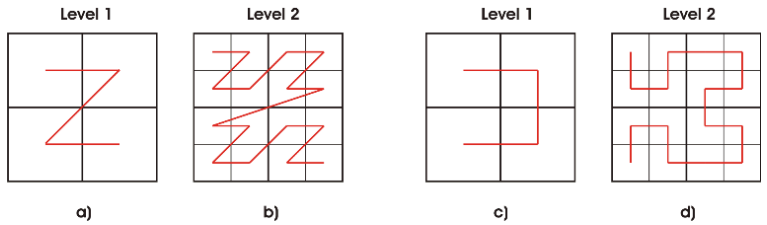


Figure 1
Linearization of the two-dimensional space of various levels,
using a, b) Morton order and c, d) Hilbert curves

Efficient Sparse Voxel Octrees (ESVOs), introduced in 2010, are sparse voxel octrees, in which whole subtrees may be efficiently replaced with 32 b-long contour information (i.e. a 24 b-long contour pointer and an 8 b-long contour mask) [18].

High Resolution Sparse Voxel Directed Acyclic Graphs (SVDAGs), proposed in 2013, are DAGs incorporating pointers to the child nodes into their internal nodes and allowing common subtree merging, when two or more child node pointers point to the same node of the data structure [5]. Child node pointers are 0 b and 32 b-long, respectively. Each part of the data structure is 32 b-aligned. In 2016, Symmetry-aware Sparse Voxel Directed Acyclic Graphs (SSVDAG) were introduced in [2] [3]; these incorporate pointers to the child nodes of 0 b, 16 b, 32 b and 33 b lengths, which is why 2 b header tags of the child node mask with an overall length of 16 b are used. Three bits of child node pointers are used to encode a reflective symmetry transformation (separately, in each main axis of the scene) when common subtree merging is used, for improved lossless compression of the data structure. CSM is applicable not only to identical subtrees, but also to subtrees identical after applying reflective symmetry transformation.

SSVDAG* is a small modification of the SSVDAG data structure, proposed in 2019 in [19], replacing 33 b-long pointers with another 16 b-long pointer without the symmetry transformation representation. This allows a higher compression ratio but smaller overall size of the binary representation of the data structure, due to the smaller overall addressing space of pointers in each level of data structure.

Pointerless Sparse Voxel Directed Acyclic Graphs (PSVDAGs) were proposed in 2020 in [4]. This hierarchical data structure allows common subtree merging using the concept of Labels/Callers, with variable-length and their frequency-based compaction. See subsection 3.2 for further details.

Out-of-core construction of Sparse Voxel Octrees (SVOs) from triangle meshes was introduced in [20] [21]. The algorithm consists of two basic steps. The first is a voxelization process, in which the triangles representing the scene form an intermediate product – a high-resolution three-dimensional voxel grid. In the second step, this intermediate product is transformed into SVOs. The algorithm

allows the size of the binary representation of the input triangle mesh, the output SVO, and the intermediate product – a three-dimensional voxel grid generated at a high resolution and represented by a Morton order – to exceed the available computer memory by far. Compared to the in-core algorithm, it uses only a fraction of the memory and has comparable processing time.

3 Hierarchical Data Structures

This section describes the SVDAG and PSVDAG data structures – the formal description in the Backus-Naur Form (BNF) is accompanied by a brief explanation of the basic features of those data structures.

3.1 Sparse Voxel Directed Acyclic Graphs

The formal description of the SVDAG hierarchical data structure using Backus-Naur Form is as follows:

SVDAG ::= (n) <NODE>

NODE ::= <INODE> | <LNODE>

INODE ::= <CHNM> (p) <BIT> <PTS>

LNODE ::= <CHNM> (q) <BIT>

CHNM ::= (8) <HT>

*PTS ::=(1) * (8) <PT>*

PT ::= (r) <BIT>

HT ::= <BIT>

BIT ::= "0" | "1"

(1)

Where the following applies:

<SYM> - mandatory non-terminal symbol SYM,

"sym" – terminal symbol sym,

(n) <SYM> - symbol SYM, concatenated n times,

(n)(m) <SYM> - symbol SYM, concatenated from n to m times*

| - alternative

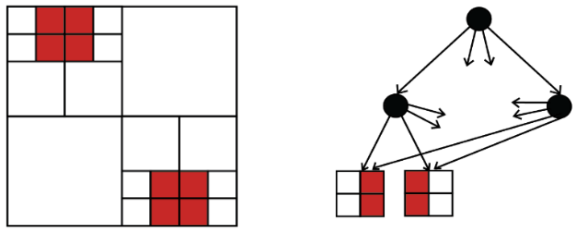
Juxtaposition – concatenation.

SVDAGs represent the three-dimensional grid of voxels R that comprises N^3 voxels, where $N \geq 2$; $N = 2^m$. If $N = 2$, the root node of the SVDAG data structure is constructed as the leaf node *LNODE*. If N >2, the root node of the SVDAG data structure is constructed as the internal node *INODE*.

Each internal node (*INODE*) of the data structure comprises a Child Node Mask (*CHNM*) and an array of pointers (*PTS*). The *CHNM* comprises 8 b, where each potential child node is represented by a 1 b header tag (HT). If HT is set to 0, the related subDAG of the grid *R* is homogenously filled by passive voxels and there is no need for pointer representation, because the subDAG is pruned out. If HT is set to 1, it represents the subDAG of the grid *R*, in which at least one of the voxels is active, and therefore the child node is present in the data structure and the related pointer *PT* is included in the *PTS*. The *CHNM* is concatenated with *p* reserved bits, to align the size of this part of the node. The next part of the node is the *PTS*, containing 1 to 8 pointers *PT*, each of them *r* bits long. The order of *HTs* in the *CHNM* and that of the pointers in the *PTS* depends on the selected linearization.

Two or more pointers from different nodes at the same level *n* of the data structure and even from the same node may point to the same address and therefore to the same node (their child node) in level $n + 1$ of the data structure. This allows common subtree merging and further lossless compression without any decompression overhead in comparison to the octree version of the data structure, which does not allow two pointers to point to the same node. In SVDAGs proposed in [5], each part of the node and therefore each node and the overall data structure is 32 b-aligned, when parameters *p* and *q* are set to 24 and *r* is set to 32.

Figure 2 shows an example of a two-dimensional (for the sake of simplicity) grid of pixels and the corresponding directed acyclic graph that can be transformed into a binary-represented SVDAG. Each node has four potential child nodes in the Morton order [8]. Passive pixels are set in white, active pixels in red. Two subgrids of 4×4 pixels are pruned out along with four 2×2 subgrids. Each of the two other subgrids of 2×2 pixels is represented in the grid twice; that is why common subtree merging is performed two times. For linearization, we used the Morton order.



a) b)

Figure 2

Example of a) a two-dimensional grid of pixels with passive voxels (white) and active voxels (red); and b) a directed acyclic graph that can be transformed into a sparse voxel directed acyclic graph using common subtree merging

3.2 Pointerless Sparse Voxel Directed Acyclic Graphs

The formal description of the PSVDAG hierarchical data structure using the Backus-Naur Form is as follows:

$$
\begin{aligned}
&\textit{PSVDAG ::= <NODE>}\\
&\textit{NODE ::= <INODE> | <LNODE>}\\
&\textit{INODE ::= <ACHNC> <CHNM>}\\
&\textit{ACHNC ::= (3) <BIT>}\\
&\textit{CHNM ::= (1)*(8) <HT>}\\
&\textit{HT ::= “00“ | “01“ <LAB> <NODE> | “10“ <CAL> | “11“ <NODE>}\\
&\textit{LAB ::= <SIZ><VAL>}\\
&\textit{CAL ::= <SIZ><VAL>}\\
&\textit{SIZ ::= (5)*<BIT>}\\
&\textit{VAL ::= (1)*(32) <BIT>}\\
&\textit{LNODE ::= (8) <BIT>}\\
&\textit{BIT ::= “0“ | “1“}
\end{aligned}
\tag{2}
$$

A PSVDAG represents a three-dimensional grid of voxels R comprising N^3 voxels, where $N \geq 2;\ N = 2^m$. The *PSVDAG* data structure comprises the root node (*NODE*) that represents the overall 3D scene. If $N = 2$, the root node of the PSVDAG data structure is constructed as the leaf node *LNODE*. If N > 2, the root node of the PSVDAG data structure is constructed as the internal node *INODE* and is further iteratively decomposed into 8 child nodes, dividing the scene into 8 suboctants. Each node can be either an *INODE* or an *LNODE (*if the node is located in the last level of the hierarchical data structure).

The *INODE* consists of the *Active Child Node Count* (*ACHNC*) and the *Child Node Mask* (*CHNM*). The *ACHNC* is a 3 b unsigned integer value showing the number of active child nodes of the particular node (using a value decremented by 1). It allows potential lossless compression of the *CHNM*. The *ACHNC* facilitates the reconstruction of pointers in the PSVDAG–SVDAG transformation process. The *Child Node Mask* comprises header tags *HT* ordered according to the selected linearization. All *HT*s indicating passive child nodes (set to “00”), placed beyond the last active child node *HT* (the active child node indicator *HT* is encoded as “01”,”10” or ”11”) in the *CHNM* are omitted, as it can be seen in Figure 3, in which 4 *HT*s are omitted.

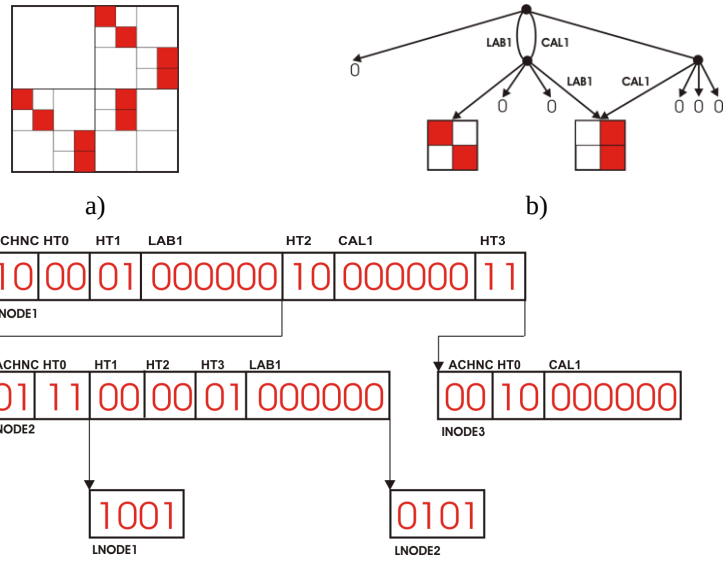
ACHNC	HT0	HT1	HT2	HT3	HT4	HT5	HT6	HT7
001	11	00	00	01	00	00	00	00

Figure 3

An example of the node structure, where *ACHNC* set to 1 indicates 2 active child nodes (here: HT0 and HT3) and four omitted HTs (HT4 to HT7). For the sake of simplicity, only *HT* indicators are represented, without concatenated Labels and Callers.

The meaning of *HT* indicator values is as follows:

- **00** – a passive child node header tag indicating that there is no active voxel in the corresponding suboctant of the grid R. There is no further information concatenated (the passive subtree is pruned out).
- **01** – an active child node header tag indicating a child node being the root node of a subtree present in the data structure multiple times. The header tag indicator is concatenated with the Label along with the representation of that child node.
- **10** – an active child node header tag indicating a child node being the root node of a subtree present in the data structure multiple times. The HT is concatenated with the Caller. No other information is needed, because the whole subtree is pruned out.
- **11** – an active child node header tag, concatenated with the root node of the subtree, present in the data structure only once.



10 00 01 000000 01 11 1001 00 00 01 000000 0101 10 000000 11 00 10 000000

d)

Figure 4

An example of a) a two-dimensional grid of 8×8 pixels, b) a quadtree with labels and callers marked to illustrate the relation to the PSVDAG, c) the structure of PSVDAG nodes and d) the final binary representation of the PSVDAG data structure in its version for 2D where only two bits are used for the active child node count (ACHNC)

PSVDAGs are pointerless data structures allowing merging of common subtrees (CSM). If two or more subtrees are identical in the data structure, only one full decomposition of the subtree is present in the data structure: in its root node, the *HT* is set to "01", concatenated with the Label. For all other instances of this subtree, the *HT* is set to "10" and a Caller (with the same binary representation as the Label) is used, but the subtree is not decomposed (its root node and whole decomposition is pruned out and the Caller represents the link to the template subtree used instead when traversing the data structure). This way, complete representations of subtrees can be referenced in PSVDAGs, multiple times. When depth-first traversing the PSVDAG, to the first occurrence of this common subtree, the Label is assigned and to all other occurrences, the Caller.

The Label has two components – label size (*SIZ*), which is a 5 b unsigned integer and value (*VAL*). *SIZ* represents the length of the value *VAL* in bits, decremented by 1. If *SIZ* is *n*, the length of value is *n + 1* bits. *VAL* can be from the range of $<0; 2^{n+1}-1>$. For each level *l* of the data structure nodes, the labels are generated separately, from the smallest value of *SIZ* and *VAL* (SIZ = 0 and VAL = 0). For each following label in a particular level, the value of *VAL* is incremented and when the capacity of this size of *VAL* is filled, size *SIZ* is incremented and *VAL* is set to 0 for the next label. Taking only *SIZ* and *VAL* into account, each Label within level *l* is unique. Considering level *l*, *SIZ* and *VAL*, each label within the whole data structure is unique – see Figure 4.

The Caller structure follows the same principles as that of the Label, when it comprises *SIZ* and *VAL*. A Caller with the same values of *l*, *SIZ* and *VAL* as the particular Label points to the subtree labelled with this particular Label. This allows common subtree merging, using the concept of variable-length Labels/Callers.

To each level *l* of the data structure, frequency-based compaction is applied, when all nodes referenced from level *l - 1* more than once are ordered by their frequency of referencing in descending order and labels generated for this level of nodes are ordered by their binary-representation length *SIZ* and value of the *VAL* in ascending order. The most frequently referenced node in the level is therefore assigned the label with the shortest binary-level representation and vice versa. This allows to obtain the lowermost possible number of bits forming Labels/Callers for this principle of Label/Caller-encoding.

Leaf nodes (*LNODE*) contain one bit per voxel – active voxels are set to 1, passive voxels are set to 0. In the leaf node, voxels are ordered according to the selected linearization (Morton order [8] in Figure 4).

For detailed information regarding PSVDAG encoding principles and features, refer to [4].

4 The Proposed Conversion Algorithm

One of the basic features of PSVDAGs is that considering binary-level encoding, internal nodes are not encoded as uninterrupted sequences of bits, but quite the opposite: binary-level representations of internal nodes are intermittent, when child node representations are inserted. Internal node representation can be therefore distributed across the whole data structure. Child node pointers are absent, while some of them are replaced by Labels and Callers having a variable-length binary-level representation.

The *CHNM* of the internal nodes can have a variable-length binary representation and *HT* indicators are encoded using 2 b. On the contrary, in SVDAGs, internal nodes are encoded as uninterrupted sequences of their binary-level representation, the *CHNM* has a constant length with 1b *HT*s and child nodes are referenced using constant-length pointers.

The conversion algorithm must therefore perform four important tasks:

- Extract the binary-level representation of the PSVDAGs internal nodes and transform them into compact sequences of binary-level representation, as used in SVDAGs
- Transform the PSVDAG *CHNM* into the SVDAG *CHNM*, i.e. transform 2 b PSVDAG *HT* indicators into 1b SVDAG *HT*s
- Generate child node pointers referencing child nodes in SVDAG internal nodes and those that are missing in PSVDAG HDS
- Replace PSVDAG Labels and Callers by SVDAG pointers to the child nodes in SVDAG in the way allowing CSM

The binary-level representation of the PSVDAG is the linearized form of the HDS, obtained as the product of depth-first traversing of the particular directed acyclic graph. When processed by the transformation algorithm, it is transformed into a SVDAG in one pass, progressively generating particular internal and leaf nodes of the target SVDAG, processing only one SVDAG node at a time, for each level of nodes. During the processing, we use a data structure having a significantly shorter binary representation than the whole input PSVDAG and the output SVDAG and store this structure in the main memory of the computer. During the transformation, a Label Transformation Table (LTT) – to convert Labels/ Callers into child node pointers – is needed; the binary representation of the LTT can be considerably larger. Thus, if this table is stored in the main memory of the computer, the algorithm is considered to be the semi-out-of-core version and if the LTT table is stored in secondary storage, it is considered to be the out-of-core version.

When the conversion of the PSVDAG internal node starts, the nearest free address in the SVDAG is assigned to this node as the final address where this node will be stored as the new SVDAG internal node; it will be finalized after its conversion.

This address is 32 b-long; its value will be used also when child node pointers to this child node will be constructed. Using the ACHNC of the PSVDAG node, which is also stored in the operating memory during node transformation and which was read as the first part of the PSVDAG node when its transformation started, it is possible to determine the length of the binary-level representation of the corresponding SVDAG node, although not all components of this node have been read and transformed yet. Therefore, it is possible to evaluate the next free address in the SVDAG for the next node to be transformed. This allows to determine addresses and binary-level representation lengths for all nodes processed at this time. As soon as one of those nodes gets finally processed, this node may be stored at the final address in the target HDS immediately.

The *ACHNC* of internal PSVDAG node stores the number of active child nodes of the corresponding SVDAG internal node, decremented by 1. In the SVDAG node, each active child node will have its 32 b child node pointer and another 32 b will comprise the 8 b *CHNM* of the node and 24 reserved bits (to align this part of the node to 32 b). The size of the SVDAG node $SVDAG_{nodesize}$ in bits can be calculated from the *ACHNC* of the related PSVDAG node using this formula:

$$SVDAG_{nodesize} = (ACHNC + 2) * 4\ [B] \quad (3)$$

Another component describing the SVDAG node is an 8 b vector, representing its child node mask *CHNM*. While in PSVDAGs, the *CHNM* of the child nodes is represented using 2 b *HT* indicators and some of those *HT*s can be omitted, in SVDAGs, there are always eight 1 b *HT*s in the *CHNM*. When transforming the PSVDAG *CHNM*, the passive child node – i.e. the node encoded in the PSVDAG node as the "00" *HT* indicator – is transformed into *HT* "0" in SVDAG. Active child nodes – nodes encoded in the PSVDAG node with the "01", "10" or "11" *HT* indicator – are transformed into *HT* "1" in SVDAG. *HT*s omitted in the PSVDAG node are inserted into the SVDAG, encoded as *HT* "0". Due to the selected linearization, the HT order in the SVDAG HDR will be the same as the one in the PSVDAG HDR.

Another part of the SVDAG node is the array of pointers *PTS*, where each pointer has a constant – 32 b – length. Their number varies from 1 to 8. When processing PSVDAG nodes, each processed node has an array for 8 potential pointers, each 32 b-long. Each processed node has also a pointer *ptpt* showing where the next pointer can be stored in the *PTS* array of the node.

When constructing a node pointer to the child node that is indicated by HT "11" in the PSVDAG *CHNM*, in the SVDAG, the address of the child node will be – after its completion – the next free address, stored in the variable *nnadr*. This address is also used as the value of the pointer to this child node. Considering that this child node is referenced by the pointer in the HDS only once, there is no need to store the value of this pointer in the operating memory after it has been generated and stored into the node and after the child node's address has been assigned. After this step, the algorithm starts the conversion of the child node.

When constructing the child node pointer for the active child node that is indicated by the *HT* set to "01" in the PSVDAG *CHNM*, the corresponding Label of the child node – containing *SIZ* and *VAL* – is read. Level *l*, representing the level of the particular node, is also known. The next free address in the SVDAG, stored in the *nnadr* variable, is assigned to this child node and this child node will be stored at this address in the SVDAG after its finalization. This address is used as the value of the pointer to this child node and as the value of the corresponding Label in the *LTT*. After the pointer is generated and stored into the node and the address is assigned to the child node, the processing of this child node starts.

During the PSVDAG data structure transformation, there is need to store information about addresses assigned to the particular labels, because those addresses will be used one or more times when processing the Callers. That is why the particular Labels and their addresses are progressively stored in the Label Transformation Table (*LTT*). For each triplet [*level*, *SIZ*, *VAL*] of a particular Label, the address assigned to this Label is written into the *LTT*. If the *LTT* is stored in the operating memory of the computer, the semi-out-of-core version of the algorithm is used; if the *LTT* is stored in the secondary storage of the computer, the out-of-core version of the algorithm is used.

When constructing the child node pointer for the child node indicated by the HT set to "10" in the PSVDAG *CHNM*, the related Caller of the child node – containing *SIZ* and *VAL* – is read. Level *l*, representing the level of the particular node, is also known. The value of the pointer to this child node that will be used when constructing the SVDAG node is then read from the LTT table, where it may be found using the triplet [*level*, *SIZ*, *VAL*] of the Caller when the same triplet of the Label is found and the assigned address is used as the pointer.

For each constructed internal node of the SVDAG, the 32 b target address, stored in *adr* is known. The number of active child nodes, calculated from the *ACHNC*, is stored in the 8 b *achnc* variable. There is a 32 b *chnm* variable, representing the node's *CHNM*. The *pts* array may store at most 8 child node pointers, 32 b each. The number of node pointers actually processed for this node is stored in the 8 b *ptpt*. Each node is therefore represented in memory by the 336 b structure *node*, consisting of *adr*, *achnc*, *chnm*, *pts* and *ptpt*. The array *nodes* comprise one *node* structure for each level of internal nodes of the HDS. After each pointer is added to the *pts* array of the particular *node*, the algorithm checks if it was the last pointer expected for this node. If yes, the node is finalized and stored at the *adr* address in the SVDAG. The related *node* structure is then initialized and prepared for processing another node of the same level.

The proposed conversion algorithm has three steps. ***Step 3*** is called iteratively.

Step 1 *nnadr = 0; level = 0;*

if (*maxl == 1*) ***Step 2; end;***

if *(maxl >1)* S***tep 3; end;***

```
Step 2   readLNODE();
         writeLNODE();
         nnadr += 4;
         return;
Step 3   level++;
         nodes[level].achnc = readACHNC() + 1;
         nodes[level].adr = nnadr;
         nnadr += (nodes[level].achnc + 1) * 4;
         nodes[level].ptpt = 0;
         until (nodes[level].achnc > nodes[level].ptpt) {

         ht = readHT();
            switch (ht) {

                  case 0:  updateCHNM(nodes[level].chnm, 0);
                           break;
                  case 1:  updateCHNM(nodes[level].chnm, 1);
                           [siz, val] = readLabel();
                           putInLTT(level, siz, val, nnadr);
                           updatePTS(nodes[level].pts[ptpt++], nnadr);
                           if (level < lmax-2) Step 3;
                           if (level == lmax-2) Step 2;
                           break;
                  case 2:  updateCHNM(nodes[level].chnm, 1);
                           [siz, val] = readCaller();
                           adr = getFromLTT(level, siz, val);
                           updatePTS(nodes[level].pts[ptpt++], adr);
                           break;
                  case 3:  updateCHNM(nodes[level].chnm, 1);
                           updatePTS(nodes[level].pts[ptpt++], nnadr);
                           if (level < lmax-2) Step 3;
                           if (level == lmax-2) Step 2;
                           break;
            }
         }
         writeINODE(nodes[level]);
         return;
```

The input of the proposed conversion algorithm is represented by the stream of bits of the binary-level representation of the PSVDAG (described in subsection 3.2), and *maxl*, the number of node levels in this data structure (*maxl > 0*). Node levels are from the range of *<0; maxl - 1>*: the root node is located in level 0, the internal nodes are located in levels *<0; maxl - 2>*, the leaf nodes are located in level *maxl - 1*.

Step 1 of the algorithm sets the next node variable *nnadr* to 0, as well as the PSVDAG node level indicator, the variable *level*. Subsequently, it has to be tested if the root node of the PSVDAG data structure is a leaf node or an internal node. If the input parameter *maxl* is set to 1, the root node of the PSVDAG is a leaf node, ***Step 2*** of the algorithm is performed and then execution of the algorithm ends. If the input parameter *maxl* > 1, the root node of the PSVDAG is an internal node, ***Step 3*** of the algorithm is performed and then the execution of the algorithm ends.

Step 2 processes the leaf node of the PSVDAG, encoded in the PSVDAG as an 8-bit vector, where each bit represents one voxel. ***readLNODE()*** reads 8 bits from the input stream. ***writeLNODE()*** writes an 8-bit vector representing the leaf node of the SVDAG data structure and adds 24 reserved bits to the output stream at the address *nnadr*. The value of *nnadr* (in bytes) is then incremented by 4.

Step 3 processes an internal node of the PSVDAG. The *level* value is incremented and the *ACHNC*, a 3-bit vector, is read from the input stream. Its incremented value shows how many child node pointers will the SVDAG node have. Its final address is set to the *nnadr* value and stored in *adr*. The value of *nnadr* (next new node address) is computed, and pointer *ptpt* is set to 0.

Until the last active child node *HT* is read, the following is repeated:

ht is read using ***readHT()***.

If the *ht* is set to 0 (“00“), *chnm* in the *node* structure of SVDAG nodes, stored in *nodes* array, is updated using ***updateCHNM()***, setting the corresponding HT to 0.

If *ht* is set to 1 (“01“), *chnm* in the *node* structure of SVDAG nodes, stored in the *nodes* array, is updated using ***updateCHNM()***, setting *HT* to 1. The Label is read using ***readLabel()*** and *LTT* table is updated using ***putInLTT()***, using the particular *level*, *siz* and *val* values to set the Label address to *nnadr*. The pointer array *pts* of the particular structure *node* is updated using ***updatePTS()*** – the new pointer is written into this array and *ptpt*, showing the number of pointers stored for this node, is incremented. Then, child node construction starts and – depending on the level in which it is stored – ***Step 3*** (if the child node is an *INODE)* or ***Step 2*** (if it is an *LNODE*) is performed.

If *ht* is set to 2 (“10“), *chnm* in the *node* structure of SVDAG nodes, stored in the *nodes* array, is updated using ***updateCHNM()***, setting the related *HT* to 1. The caller is read using ***readCaller()***, and from the *LTT* table, the address is retrieved for particular *level*, *siz* and *val* values, using ***getFromLTT()***. The pointer

array *pts* of the particular structure *node* is updated using ***updatePTS()*** **–** the new pointer is written into this array and *ptpt*, showing the number of pointers stored for this node, is incremented.

If *ht* is set to 3 ("11"), *chnm* in the *node* structure of SVDAG nodes, stored in the *nodes* array, is updated using ***updateCHNM()***, setting the related *HT* to 1. The pointer array *pts* of the particular structure *node* is updated using ***updatePTS()*** **–** the new pointer is written into this array and *ptpt*, showing the number of pointers stored for this node, is incremented. Then, child node construction and – depending on the level in which it is stored – ***Step 3*** (if the child node is an *INODE)* or ***Step 2*** (if it is an *LNODE*) is performed.

After all active child node *HT*s are processed, the particular INODE is written to the output using ***writeINODE()***.

5 Results

Various 3D scenes, originally stored in Wavefront OBJ geometry definition file format (examples of those scenes with their visualizations are in Figure 5), were used for test purposes. The "Angel Lucy" model consisted of 488 880 triangles, the "Skull" model had 80 016 triangles and the "Porsche" model had 22 011 triangles. These were voxelized to different resolutions, ranging from 128^3 to 1024^3 ($1\ K^3$). The voxelized scenes were then encoded as PSVDAG hierarchical data structures.

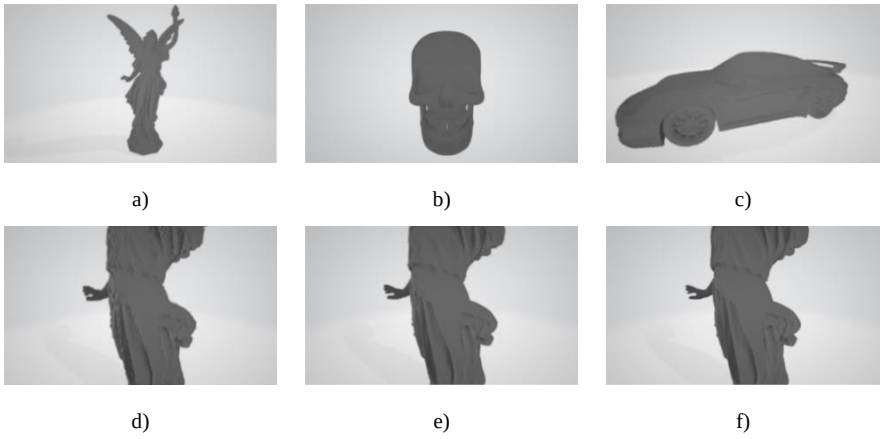


Figure 5
Visualization of voxelized scenes for testing purposes: a) "Angel Lucy" 512^3; b) "Skull" 512^3; c) "Porsche" 512^3; d) detail of the "Angel Lucy" 256^3 model; e) detail of the "Angel Lucy" 512^3 model; f) detail of the "Angel Lucy" $1\ K^3$ model

Details of particular models and their particular voxelizations to the respective resolutions can be found in Table 1.

Table 1
Parameters of particular models, voxelized to the particular resolutions and stored as PSVDAGs, for test purposes

		Active voxels					
		Angel Lucy		**Skull**		**Porsche**	
Resolution [vox]	**Voxels [×2^{20}]**	**[×10^3]**	**[%]**	**[×10^3]**	**[%]**	**[×10^3]**	**[%]**
128^3	2	22,48	8,78	74,10	28,95	54,20	21,17
256^3	16	91,52	4,47	298,85	14,59	233,04	11,38
512^3	128	366,58	2,24	1192,04	7,28	969,11	5,91

By converting PSVDAGs into SVDAGs, the space needed to store the scene geometry increased. In case of the particular models and voxelization resolutions, the increase ranged from 3.01-fold (in case of the "Skull" model at 1 K^3 scene voxelization resolution) to 3.70-fold (in case of the "Angel Lucy" model at 128^3 scene voxelization resolution). So, in general, higher voxelization resolutions led to smaller inflation rates. See Table 2 for absolute sizes of the PSVDAGs and SVDAGs (in KB) for the respective models and voxelization resolutions.

Table 2
Size of particular models at various voxelization resolutions, stored as PSVDAG and SVDAG hierarchical data structures

Size [KB]		**Resolution**		
		128^3	**256^3**	**512^3**
Angel Lucy	**PSVDAG**	8.62	35.10	132.37
	SVDAG	31.88	127.01	462.85
Skull	**PSVDAG**	28.11	104.31	366.48
	SVDAG	100.39	356.75	1182.45
Porsche	**PSVDAG**	15.80	61.28	227.13
	SVDAG	56.78	222.41	788.76

During the conversion process, it was necessary to maintain 32 b addresses assigned to the individual Labels in the LTT table. The smallest number of these addresses (182) was needed for the Angel Lucy 128^3 model, requiring 0.71 KB of space. The largest number of these addresses (9179) was needed in the case of the Skull 512^3 model, requiring 35.86 KB of space.

The semi-out-of-core version of the conversion algorithm that stores the Label Transformation Table in the operating memory, transformed PSVDAGs into SVDAGs with a higher data throughput and therefore faster for each model and voxelization resolution, in comparison to the out-of-core algorithm storing the Label Transformation Table in the secondary storage.

Table 3
Time in seconds and data throughput in MB for particular models at various voxelization resolutions – conversion from PSVDAG hierarchical data structure into SVDAG hierarchical data structure, for the semi-out-of-core (SOoC) and out-of-core (OoC) versions of the algorithm

Time [s] **Data throughput [KB/s]**		**Resolution**		
		128^3	**256^3**	**512^3**
Angel Lucy	**SOoC**	0.28*1 112*2	1.14 111	4.62 100
	OoC	0.30*1 106*2	1.21 105	4.73 98
Skull	**SOoC**	0.81*1 124*2	3.24 110	13.59 87
	OoC	0.85*1 118*2	3.43 104	14.44 82
Porsche	**SOoC**	0.44*1 129*2	1.97 113	7.33 108
	OoC	0.47*1 121*2	2.04 109	7.51 105

*1 Conversion time in seconds
*2 Data throughput measured on the output in KB/s

The time required for the conversion of the particular models and voxelization resolutions ranged from 0.28 s for the "Angel Lucy" model at 128^3 resolution and the semi-out-of-core version to 14.44 ms for the "Skull" model at 512^3 resolution and the out-of-core version. The time consumption of the out-of-core version was 1.024–1.057-times higher than that of the semi-out-of-core version.

Data throughput was 82 KB/s – 129 KB/s and was strongly affected by the seek operation, performed when storing the completed nodes in the final output file. Output data throughput at the level of 34.5 MB/s was achieved in the case when a fully in-core transformation was performed on the model "Skull" having a 512^3 resolution. Both the PSVDAG and the LTT structures were stored in the computer's operating memory, together with the SVDAG hierarchical data structure, which was consecutively stored into the secondary storage after its finalization.

Conclusions

This paper examined the issues related to three-dimensional scene geometry representation, using domain-specific hierarchical data structures, building on previous work in the field – Pointerless Sparse Voxel Directed Acyclic Graphs. This data structure is well suited for archiving and streaming purposes; however, quick traversing requires reconstruction of pointers. That is why PSVDAG incorporates a feature that facilitates pointers reconstruction. Two versions – an out-of-core and a semi-out-of-core – of the PSVDAG–SVDAG conversion

algorithm were proposed and introduced as the contribution of this paper, along with test results that were performed on different models voxelized to various resolutions.

During the conversion of PSVDAGs into SVDAGs, pointers are generated. Pointers take significant amount of space in SVDAGs, so conversion causes inflation of the space required for storing this HDS, in the operating memory of computer or in the memory of the graphics card. For particular models and voxelization resolutions, the resulting inflation ranged from 3.01-fold (in case of the "Skull" model at a 1 K^3 scene voxelization resolution) to 3.70-fold (in case of the "Angel Lucy" model at a 128^3 scene voxelization resolution). In general, higher voxelization resolutions need more space for Labels/Callers in PSVDAGs, which results in a relatively smaller inflation ratio when converting to the SVDAG data structure. Considering conversion time, the out-of-core version of the algorithm (storing the Label Transformation Table on secondary storage), has a disadvantage, compared to the semi-out-of-core version of the algorithm (storing the Label Transformation Table in the operating memory of the computer). Thus, the out-of-core version of the algorithm was 1.024 to 1.057 times slower than the semi-out-of-core version of the algorithm, for all models and voxelization resolutions. When testing the in-core version of the algorithm, a data throughput 34.5 MB/s was achieved.

Acknowledgement

This research was supported by the Slovak Research and Development Agency, project number APVV-18-0214 and by KEGA 002TUKE-4/2021 Implementation of Modern Methods and Education Forms in the Area of Cybersecurity towards Requirements of Labour Market.

References

[1] C. Crassin, F. Neyret, S. Lefebvre and E. Eisemann, GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D. 10.1145/1507149.1507152

[2] A. J. Villanueva, F. Marton and E. Gobbetti, Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes, Journal of Computer Graphics Techniques (JCGT), May 8, 2017, Vol. 6, No. 2, p. 30, 2017, ISSN 2331-7418, http://jcgt. org/ published/0006/02/01/

[3] A. J. Villanueva, F. Marton, and E. Gobbetti, SSVDAGs: Symmetry-aware Sparse Voxel DAGs. In Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '16) February 27-28, 2016, Redmond, WA, USA, pp. 7-14, ACM, New York, NY, USA, ISBN: 978-1-4503-4043-4/16/03, DOI: https://doi.org/ 10.1145/285 6400.2856420

[4] L. Vokorokos, B. Madoš and Z. Bilanová, PSVDAG: Compact Voxelized Representation of 3D Scenes Using Pointerless Sparse Voxel Directed Acyclic Graphs", In: Computing and Informatics: Computers and Artificial Intelligence - Bratislava (Slovakia), Vol. 39, No. 3 (2020), pp. 587-616, [print] - ISSN 1335-9150

[5] V. Kämpe, E. Sintorn, and U. Assarsson., High Resolution Sparse Voxel DAGs. ACM Transactions on Graphics. 32, 4, Article 101 (July 2013), pp. 8, ISSN 0730-0301, DOI: https://doi.org/10.1145/2461912.246 2024

[6] A. Laszloffy, J. Long and A. K. Patra, Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive finite element simulations. Parallel Computing, 26, ISSN 1765-1788

[7] H. Sagan, Space-Filling Curves, Springer Verlag, 1994, ISSN 978-1-4612-0871-6, ISBN 978-0-387-94265-0, DOI 10.1007/978-1-4612-0871-6

[8] G. M. Morton, A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, Research Report. International Business Machines Corporation (IBM), Ottawa, Canada, 20, p. 20, March 1st, 1966, Available: https://dominoweb.draco.res.ibm.com/reports/Morton1966.pdf

[9] D. Hilbert, Via the continuous mapping of a line onto a patch of area. Mathematical annals (orig. Über die stetige Abbildung einer Linie auf ein Flächenstück. Mathematische Annalen) 38 (1891), pp. 459-460

[10] S. N. Srihari, Representation of Three Dimensional Digital Images. Technical Report No. 162, Department of Computer Science, State University of New York at Bufallo, Amherst, New York, pp. 26, 1980

[11] S. M. Rubin, T. Whitted, A 3-Dimensional Representation for Fast Rendering of Complex Scenes. Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80), ACM, 1980, pp. 110-116, doi: 10.1145/800250.807479

[12] C. L. Jackins and S. L. Tanimoto, Octrees and Their Use in Representing Three-Dimensional Objects. Computer Graphics and Image Processing, 1980, Vol. 14, No. 3, pp. 249-270, doi: 10.1016/0146-664X(80)90055-6

[13] D. J. R Meagher, Octree Encoding: A New Technique for the Representation, Manipulation, and Display of Arbitrary 3-D Objects by Computer. Technical Report No. IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, 1980

[14] D. J. R. Meagher, Geometric Modeling Using Octree Encoding. Computer Graphics and Image Processing, 1982, Vol. 19, No. 2, pp. 129-147, doi: 10.1016/0146-664X(82)90104-6

[15] D. J. R. Meagher, The Octree Encoding Method for Efficient Solid Modeling. Technical Report IPL-TR-032, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 1982

[16] B. Madoš, E. Chovancová and M. Hasin, Evaluation of Pointerless Sparse Voxel Octrees Encoding Schemes Using Huffman Encoding for Dense Volume Datasets Storage, In: ICETA 2020: 18th IEEE International conference on emerging elearning technologies and applications: Information and communication technologies in learning: proceedings - Denver (USA): Institute of Electrical and Electronics Engineers pp. 424-430, ISBN 978-0-7381-2366-0

[17] B. Madoš, N. Ádám and M. Štancel, Representation of Dense Volume Datasets Using Pointerless Sparse Voxel Octrees With Variable and Fixed-Length Encoding, IEEE 19th World Symposium on Applied Machine Intelligence and Informatics, SAMI 2021, Herľany, Slovakia, January, 21-23, 2021, p. 6

[18] S Laine and T. Karras, Efficient Sparse Voxel Octrees. In Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games. I3D '10, Washington D.C, pp. 55-63, ACM Press, New York, NY, USA, ISBN: 978-1-60558-939-8, DOI: 10.1145/1730804.1730814

[19] P. Čerešník, B. Madoš, A. Baláž and Z. Bilanová, SSVDAG*: Efficient Volume Data Representation Using Enhanced Symmetry-Aware Sparse Voxel Directed Acyclic Graph, In: IEEE 15th International Scientific Conference on Informatics: proceedings - New York (USA): Institute of Electrical and Electronics Engineers, pp. 70-75 [print] - ISBN 978-1-7281-3178-8

[20] J. Baert, A. Lagae and Ph. Dutré, Out-of-core Construction of Sparse Voxel Octrees. In Proceedings of the 5th High-Performance Graphics Conference. HPG ’13, July 19-21, Anaheim, California, pp. 27-32, ACM, New York, NY, USA, ISBN: 978-1-4503-2135-8/13/07

[21] J. Baert, A. Lagae and Ph. Dutré, Out-of-Core Construction of Sparse Voxel Octrees, Computer Graphics Forum Vol. 33, No. 6, pp. 220-227, ISSN 0167-7055, https://doi.org/10.1111/cgf.12345