# Image Synthesis

# Final Course Project

# “High Resolution Sparse Voxel DAGs”

Dmitry Gaynullin

March 2021

## Context of the problem

With increased interest in the video-gaming industry in evaluating secondary rays for different graphical effects such as reflections, indirect illumination and shadows for real-time rendering it’s important to use acceleration structures for fast ray-tracing algorithms. Also, as the quality of the rendered scenes is increasing it’s important for these data structures to be quite compact to fit in limited GPU memory.

So, the final goal of this project is to reduce the amount of memory occupied by the most frequently used acceleration structure.

## Project Description

### Sparse Voxel Octree

Recently, Sparse Voxel Octree (SVO) has shown quite promising results as secondary scene representation. Let’s suppose, that we have a graphical scene with the resolution equals to N, this scene is subdivided by 3D grid in $N^3$ voxels. These voxels would encode the presence of the geometry by 1 and its absence by 0. So, this octree works as follows: it’s a tree where each volume is divided in 8 parts hierarchically and close voxels would be regrouped in the same subtree and they would share the same parent node. This tree is sparse because all empty nodes (subtress with all children voxels containing zeroes) are deleted from the tree for less memory consumption. But at high resolutions the SVOs are still much too memory expensive, this limits the usage of this data structure for non-high resolution effects.

### Current implementations of SVO

There exists several libraries and algorithms for SVO construction for the 3D scene. The most used library is presented in [1] and it’s implementation of the method presented in article [2] by the same authors. We will use this library and classes for SVO-structure implementations in our code.

### Compression method presented in the article

First of all, the article “High Resolution Sparse Voxel DAGs” [3] presents the method of more efficient encoding of scene’s geometry and propose the algorithm of compressing the Sparse Voxel Octree (SVO) into a Sparse Voxel Directed Acyclic Graph (SVDAG) without information loss. The principal idea of this algorithm is to recursively (from leaves to the root) replace the identical subtrees by a pointer to one of these subtrees, so we could encode a similar local geometry of the scene.

After that, the authors prove that ray tracing algorithms could traverse the tree as efficiently as on a decompressed tree, this allows use of SVDAG structure for secondary-rays effects in high resolution scenes.

Finally, they confirmed empirically that this transformation reduces the amount of memory occupied by the data structure, the reduction of nodes varied from 2 (for lowest resolution) to 576 (for highest resolution), depending on symmetry and regularity of the scene. Also, they compared the traversal speeds of the algorithm with previous states-of-the-art results.

## Description of the implementation

### Algorithm overview

First of all, the SVDAG class reads nodes from .octreenodes file using the API from the library [1]. After that the SVDAG.SVDAG_nodes is filled with nodes and that utility structure SVDAG.SVDAG_levels is built .So, non-reduced data structure is constructed.

After this initialization, SVDAG::reduce is applied and it works like that:

```
for cur_level in SVDAG_levels.size()-1 downto 0:
    sort_level(cur_level)
    group_level(cur_level)
endfor
```

where two support methods are used:

- **sort_level**(cur_level) - sorts the nodes on cur_level of SVADAG_levels in-place
- **group_level**(cur_level) - merge identical (same pointers to children and these pointers are in correct order) nodes on a cur_level and update the pointers on a parent level (cur_level-1).

## Implementation details

The implementation consists of several classes:

**SVDAG_node -** one node of the SVDAG it has several fields:

**Id** - index of the node in a SVDAG.SVDAG_nodes

**children -** an vector of children indices

**parents** - an vector of parents indices (in non-reduced SVDAG contains only one parent)

**childmask -** 8 bit mask, bit i tells if i-th children is non-empty and contains a child node.

Also, operators < and == were overloaded for sorting used in an algorithm. < compares childmask values first and children indices in lexicographical order after that.

**SVDAG -** contains complete SVDAG graph:

**SVDAG** non default constructor read the .octree file and constructs the object, filling its values with octree nodes

**octree_nodes** - temporary vector containing all the octree nodes read from a file, before SVDAG_nodes is filled

**SVDAG_nodes** - vector of all SVDAG nodes, constructed from octree nodes

**SVDAG_levels** - 2d array (vector of vectors). SVDAG_level[L] contains all the indices of nodes for a level L-1, 0 level contains the root.

**SVDAG_main** - is the main class which builds all the previous blocks together and handles IO interactions.

## Results and performance measurements

All the models for performance testing were taken from “The Stanford 3D Scanning Repository” [4]. For experiments I tested several models: Stanford Bunny, Dragon, Lucy and Asian Dragon.

| Grid Size | Number of nodes in thousands | | | Consumed memory in Mb | | | Time for converting in ms |
|---|---|---|---|---|---|---|---|
| | SVDAG | SVO | SVO/SVDAG | SVDAG | SVO | SVO/SVDAG | |
| 2048 | 2031.8 | 8214.7 | 4.04 | 64.50 | 188 | 2.91 | 6100 |
| 1024 | 505.0 | 2049.3 | 4.06 | 16.10 | 47 | 2.92 | 1350 |
| 512 | 124.3 | 509.3 | 4.10 | 4.00 | 11.7 | 2.93 | 302.48 |
| 256 | 29.9 | 125.4 | 4.18 | 1.01 | 2.9 | 2.88 | 72.276 |
| 128 | 7.0 | 30.2 | 4.32 | 0.24 | 0.724 | 2.99 | 18.32 |
| 64 | 1.6 | 7.1 | 4.46 | 0.06 | 0.171 | 3 | 3.731 |

Table. Comparison of SVO and SVDAG for Lucy model with the change of grid size (Performance was counted on Intel i5 7200U processor)

I tested the algorithm with these models and find out that the number of nodes in SVO doesn't depend very much on the complexity of the model (number of vertices and triangles) and it's much more dependant on a grid size, so below I present the results for Lucy model as it had the least number of nodes and, because of that, it was possible to evaluate the model for higher grid sizes.

.

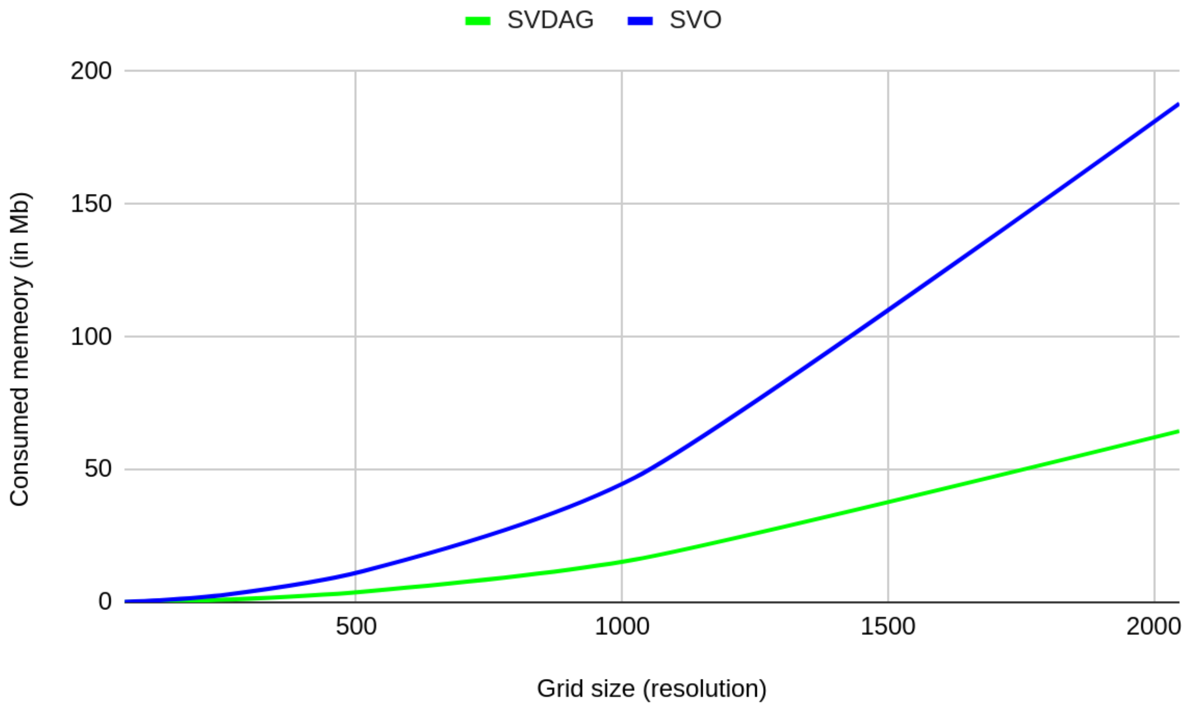

Figure. Memory consumption comparison between SVO and SVDAG for the Lucy model, depending on the grid size

The obtained results for the model presented in the table and on the figure. The memory consumption was reduced by 3x after SVO reduction. Also, the results show that for the same model reduction factor (both for memory and number of nodes) reduces and the time of transformation grows with increase of the grid size, which is quite predictable.

## Conclusions

As a conclusion, the implementation allows to transform SVO structure in SVDAG and the DAG representation is quite an efficient way of compressing the geometry of the scene based on the regularities in it. Material representation could be added to the structure in the future and some improvements in memory layout of the DAG could be made. For example, some symmetric properties of the model could be used to increase compression rate further.

## References

[1] J. Baert et al. Out-Of-Core SVO Builder v1.6.3.
https://github.com/Forceflow/ooc_svo_builder
[2] J. Baert et al. Out Of Core Construction of Sparse Voxel Octrees. HPG 2013
[3] V. Kampe et al. High Resolution Sparse Voxel DAGs.
http://www.cse.chalmers.se/~uffe/HighResolutionSparseVoxelDAGs.pdf
[4] The Stanford 3D Scanning Repository.
http://graphics.stanford.edu/data/3Dscanrep