INF584 - High Resolution Sparse Voxel DAGs

Léon Zheng

March 2019

Abstract

The Github repository of this project can be accessed at the following link: `https://github.com/leonzheng2/INF584-High_Resolution_Sparse_Voxel_DAGs`.

1 Context

1.1 Motivation

Due to increasing interest for real-time rendering in video game industry, evaluating secondary rays for effects like shadowing, reflection or indirect illumination is becoming more and more necessary. Such ray tracing algorithm needs to be efficient: fast ray-geometry queries should be fast enough to increase the number of queries. That is why acceleration structure are needed for geometry, but it has to be compact enough so that it can fit the strict memory budget of the GPU.

1.2 Sparse Voxel Octree

1.2.1 Principle

Sparse Voxel Octree (SVO) is an acceleration structure which encodes efficiently the geometry of a scene. Consider a grid subdivised in N^3 voxels, where N is the resolution of the scene. A voxel encodes '1' if it contains geometry, and '0' if it is empty. The octree is a hierarchical structure of those voxels, where close voxels are regrouped in the same voxels subtree, sharing a common node. The sparsity consist of culling away all empty voxels, thus achieving data structure compression.

However, the memory cost of an SVO is still too high for high resolution scenes, with typically $N^3 = 128K^3$, which therefore limits the use of SVO for medium resolution effects of shadowing, or indirect illumination effects.

1.2.2 Existing SVO Builder

There exists several algorithms for building SVO, such as out of core construction [3]. We will implement the code provided by the paper's author in [2] as

our code base.

2 Contributions

In order to get a more compact data structure for encoding efficiently the scene's geometry, the paper's authors have designed an algorithm to reduce a Sparse Voxel Octree into a minimal Sparse Voxel Directed Acyclic Graph (SVDAG), which is a factorized version of the SVO. The core idea is to merge identical subtrees, which encode a similar local geometry of the scene.

The authors showed that the SVDAG is a more compact representation than the SVO, having a decreasing rate in nodes between 28× and 576×, depending on the scene's regularity. As a consequence, the SVDAG can fit in memory for high resolution scenes. Moreover, the authors showed that SVDAG does not require decompression, and thus can be traversed efficiently for ray tracing algorithms, allowing high-quality secondary-rays effects to be evaluated in a voxelized scene at high resolution.

3 Implementation

The C++ code is available in the Github repository in the abstract.

3.1 Classes

3.1.1 DAGBuilder

The main methods of the class `DAGBuilder` is to load in memory the SVO, reduce the SVO into a SVDAG, and write the SVDAG in an output file.

To achieve this, it has the following attributes:

- `octreeNodes`: a vector containing all the SVO nodes
- `DAGNodes`: a vector containing all the DAG nodes
- `DAGLevels`: a vector of levels of nodes in the DAG, where the first level contains only the root, and the level i contains the children of nodes from level $i-1$.

The constructor `DAGBuilder(std::string filename)` instantiates an object ready to read the `.octree` and `.octreenodes` files given by the `filename` string.

3.1.2 DAGNode

My implementation contains a `DAGNode` class, which represents one node of the DAG, with the following attributes:

- `id`: the index of the node in the vector `DAGNodes`

- `children`: a vector containing the indices of the node's children
- `parents`: a vector containing the indices of the node's parents
- `childmask`: a 8-bit variable indicating the position and the number of children

The `DAGNode` has an overloaded `operator<` method so that we can perform sorting on instances of this class. If `A` and `B` are two instances of `DAGNode`, `A` is smaller than `B` if it has a smaller `childmask`; if it is the same value, we use the lexicographic order on the children indices of the `children` vector.

3.2 DAG building algorithm

3.2.1 Load the SVO in memory, initialize the unreduced DAG

The first step is to read the `.octreenodes` files. We can use, in order to read the nodes of the file, methods provided by the `Node` class in the file `Node.h`, like `readNode`. After reading all the nodes, the variable `DAGNodes` is fixed, containing all the `DAGNode` objects with a given index.

The second step is to build the `DAGLevels` variable, which will be used for the reducing algorithm.

3.2.2 Reducing algorithm

Once the DAG is initialized, we can start the reduction, by applying the algorithm 1, which uses two main methods:

- `void sortLevel(size_t L)`: sorts all the nodes of the level L of `DAGLevels`
- `void groupLevel(size_t L)`: after sorting, regroups identical DAG nodes at level L (same childmask, same children pointers) and updating the parents at level $L-1$ so that they point to the merged vertices.

Result: The `DAGNodes` elements in `DAGNodes` are updated so that they represent the minimal DAG
for *L: size_of(DAGLevels)-1* $\rightarrow 0$ **do**
  sortLevel(L);
  groupLevel(L);
end

Algorithm 1: Reducing into minimal DAG

3.3 Implementation results

The obtained SVDAG file is much more compact than the original SVO, as we can see in the figure 1. The reduction rate achieved is $4\times$.

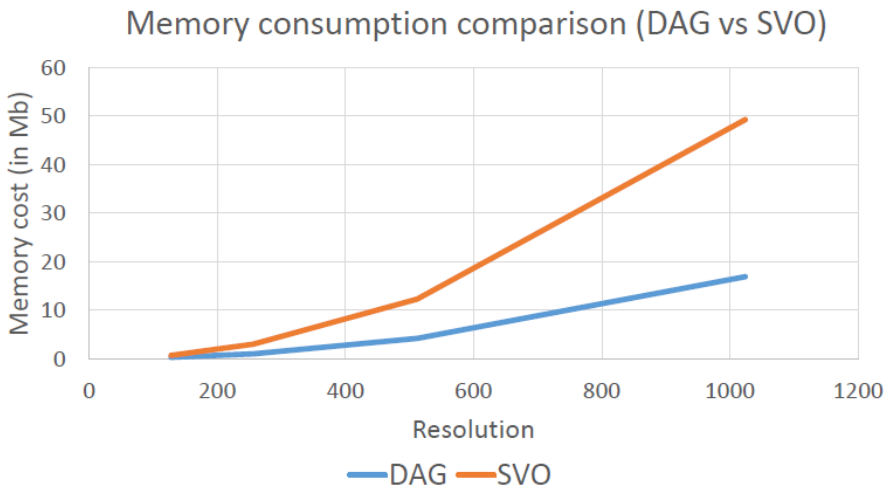

Figure 1: Memory consumption comparison between SVO and DAG of the Lucy mesh, depending on the resolution.

4 Limitations, possible improvements

The DAG is an efficient way to compact the geometry of a scene, by identifying identical local geometry. But it cannot have a material representation.

It is also possible to continue to reduce the DAG by using Symmetry-aware SVDAG, like in [1], which exploits symmetric transformations to identify similar local geometry, modulo a simple transformation like a mirror transformation or a rotation, and merge them in an efficient way.

References

[1] E. Gobbetti A. J. Villanueva, F. Marton. Ssvdags: Symmetry-aware sparse voxel dags. 2016.

[2] J. Baert. Out-of-core svo builder v1.6.1. `https://github.com/Forceflow/ooc_svo_builder`. Accessed: 2019-03-18.

[3] P. Dutré J. Baert, A. Lagae. Out-of-core construction of sparse voxel octrees. *HPG*, 2013.