Voxel DAGs and Multiresolution Hierarchies
From Large-Scale Scenes to Pre-computed Shadows

Advanced DAG encodings

Alberto Jaspe Villanueva
ajaspe@crs4.it

CRS4 :: Visual Computing
http://vic.crs4.it

EUROGRAPHICS 2018
Focus

• How to further **increase compression rates of geometric data** by exploiting symmetries in addition to replication

• How to **efficiently encode geometric data** by exploiting the skewed distribution of references to shared nodes

• How to achieve this with still an efficient traversal

(Just geometry. For **attribute compression**, next talk!)
Related Work

• Very hot topic with lots of contributions, not only DAGs
  • Voxelization
    [Crassin and Green, 2012] [Pätzold and Kolb, 2015] ...
  • Voxelized representation and render
  • Trees encoding & hashing
    [Lefevre and Hoppe, 2011] [Duan et al. 2017] ...
  • Volume compression surveys
    [Beyer et al. 2015] [Balsa et al. 2013] ...

• Symmetry-aware compression
  [Jaspe et al. 2016, 2017]

• Entropy encoding
  [Jaspe et al. 2016, 2017] [Dado et al. 2016] [Kampe et al. 2016] ...
SSVDAGs

- **Symmetry-aware**
  loseless voxel structure
- Compact **GPU-friendly** encoding
- Build and traversal methods

**Advantages**
- ~ 50% more compact
- Similar tracing speed
- Able to build massive models
Quick recap – Sparse Voxel Octrees (SVO)

- Introduce sparsity in a full-octree
  - Compression by avoiding to encode empty space
  - Grows quadratically with resolution
  - Very fast traversal methods
- Encoding requires 1 pointer per node
Quick recap – Sparse Voxel DAG (SVDAG)

- Avoiding multiple encodings of same subtrees [Kämpe et al. 2013]
  - Tree becomes a directed graph
  - Same traversal method as SVOs
- Up to x100 or more reduced number of nodes SVOs
- Encoding requires 1 pointer per each node child
SSVDAGs approach

Reduce number of nodes by merging subtrees equal up to a similarity transform

\[ A \neq B \]
\[ T(A) = B \]

Chosen transform: **Reflection Symmetries**
- Geometric meaning
- Expected in real-world scenes
Reflection Symmetries

- Mirror transform along main 3 planes
  - Pointer tag is encoded with only 3 bits (X,Y,Z)

- Useful properties for building and traversing
  - Order of application does not matter
    \[ T_x(T_y(N)) = T_y(T_x(N)) \]
  - Cyclic (applying just means inverting the tag)
    \[ T_x(T_x(N)) = N \]
Reflection Symmetries

- 2D canonical base: 7 nodes / 16
- 3D canonical base: 46 nodes / 256
SSVDAGs Building

- **Bottom-up** process
- **One pass per level**: sort – transform – match – merge
- Additional data respect to DAG construction
  - Invariant masks: 3 bits per node (only during building)
  - Transformation tags: 3 bits per pointer
- Implementations
  - Out-of-core: memory-mapped, external-memory arrays (whole levels)
  - In-core: partitioning in memory-fitting subtrees and reduction

![Diagram showing the process from mesh to SSVDAG](image-url)
Building process – Leaves level

• Symmetry detected just using the voxel bit mask

1. Look for every leaf to its canonical form
2. Remap and tag pointers with its transformation
3. Mark invariants
Building process — Example

- Symmetry detected just using the voxel bit mask

1. Look for every leaf to its canonical form
2. Remap and tag pointers with its transformation
3. Mark invariants
Building process — Inner Levels

• Search for symmetries in the whole subtree, without checking down to the leaves
Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and match nodes
Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and match nodes

\[ T_x(s) \]

\[
\begin{array}{c}
1 \quad 2 \\
3 \quad 4
\end{array}
\]
Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations 
   \textbf{Transform} and match nodes
Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and **match** nodes

For each child pointer
   Check if both are equal
   If pointed node is not invariant, check transformation tag
Building process — Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and match nodes
3. Merge matches and update pointers from upper level
Building process — Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and match nodes
3. Merge matches and update pointers from upper level
4. Update invariants
Building process — Example
Building process — Example
Building process – Example

SVO (10 nodes)

SVDAG (9 nodes)

SSVDAG (4 nodes)
Compact encoding

- **Goal**: minimize size with minimum performance impact
- Every pointer must be tagged (XYZ reflection bits)
- GPU-friendly memory layout

- **Steps**
  - Leaf grouping
  - Frequency-based pointers compaction
  - Variable bitrate memory alignment
Compact encoding

- Leaf grouping
  - $2^3$ leaves good for construction
  - $4^3$ leaves better for encoding and traversal speed

\[ \text{Min} \quad \text{1 child: } 32\text{bits} + 8\text{bits} = 5 \text{bytes} \]
\[ \text{Max} \quad \text{8 children: } 8 \times (32\text{bits} + 8 \text{bits}) = 40 \text{bytes} \]

V.S.

Fixed 8 bytes
Compact encoding

• Frequency-based pointer compaction
  • Distribution of references to nodes highly skewed

![Graph showing frequency distribution for Power Plant 64k³ with levels 11, 12, 13, and 14.]
Compact encoding

- Frequency-based pointer compaction
  - Distribution of references to nodes highly skewed
  - Sort nodes by nº references

- Variable index size
  - [1 .. 8K] : 13 bits
  - [8K .. 527M] : 29 bits
  - [527M .. 1,074M] : 30 bits
Compact encoding

Leaves nodes encoding

64 bits 4^3 leaf

Inner nodes encoding

children bitmask
00
01
00
10
00
11
00
00

reflection bits

x y z

13 bits pointer

reflection bits

x y z

29 bits pointer

reflection bits

x y z

30 bits pointer

(1st bit = 1)

node

00 No child

01 13 bits pointer child

10 29 bits pointer child

11 30 bits pointer child

Total node size

Min: 4 bytes (1 "13-bits" child)

Max: 34 bytes (8 "30-bits" children)
Traversal / Ray-casting

- Same as for SVOs/DAGs but need to keep track of transformation upon descent in the tree
  - Transformation state: 3 bits \([x, y, z]\)
  - When traverse new node: XOR with pointer’s tag

- Using the stacked transform
  - **Remap children indices**, one bitwise operation
  
  \[
  \text{transformedIdx} = \text{currentTransform XOR selectedChildIdx}
  \]

- **Transform the ray**
  - Reflect ray’s origin (voxel entry point)
  - Inverse direction
Results – Datasets

Scanned scenes
- Lucy (28 Mtris)
- David (56 Mtris)

Gaming scenes
- Crytek Sponza (282 Ktris)
- San Miguel (7.8 Mtris)

CAD scenes
- Boeing 777 (350 Mtris)
- Power Plant (12 Mtris)
Results – Compression performance (64k³)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SSVDAG</th>
<th>SVDAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lucy</td>
<td>0.85</td>
<td>1.61</td>
</tr>
<tr>
<td>6.3 GVox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>642 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td>1.91</td>
<td>0.99</td>
</tr>
<tr>
<td>41.1 GVox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>487 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cristoza</td>
<td>0.06</td>
<td>0.43</td>
</tr>
<tr>
<td>12.0 GVox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>315 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Miguel</td>
<td>0.16</td>
<td>0.24</td>
</tr>
<tr>
<td>5.6 GVox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>86 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boeing 777</td>
<td>0.79</td>
<td>1.47</td>
</tr>
<tr>
<td>24.6 GVox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,314 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results – Compression performance

• Average node reduction *only by symmetries*: ~21%
• Average data memory size reduction *only by encoding*: ~23%

• **Together,** they achieve a data **reduction average of 52.4%**
  • Symmetries favor pointer compaction

• Conversion time performance not aimed for real-time
  • Few seconds for lowest resolutions (2k³, 4k³, 8k³)
  • Up to few minutes for biggest
Results – Rendering

GLSL full-stack 3DDDA Raycaster

Symmetries overhead: ~ 1%
Encoding overhead: ~ 15%


720p – Nvidia GTX 980
Summary

- **Efficient lossless encoding** of voxelized geometry
- Identical subtrees by a **reflection transformation** are stored only once
- Node reduction is combined with pointer compaction by **entropy encoding**
- Probed in a variety of scenes, it achieves ~50% **data size reduction** respect raw DAGs, with **similar traversal speed**

- Next problems for **next talk!**
  - How to define a mapping to attributes
  - How to compress non-geometric data
  - How to render colored models efficiently

Further explanations and source code in JCGT SSVDAGs paper site
http://jcg.org/published/0006/02/01/