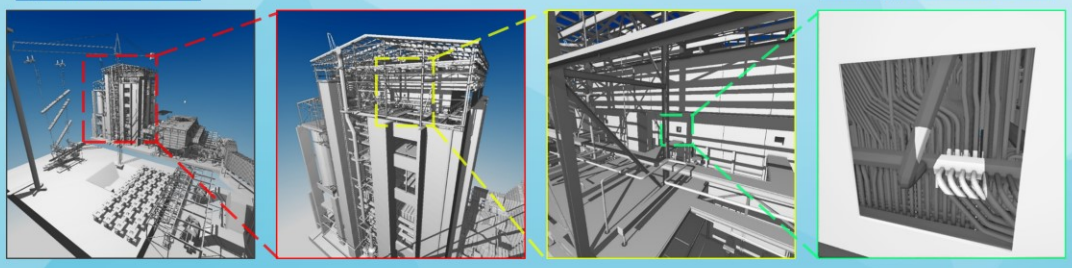
Voxel DAGs and Multiresolution Hierarchies
From Large-Scale Scenes to Pre-computed Shadows

Advanced DAG encodings

Alberto Jaspe Villanueva
ajaspe@crs4.it

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



EUROGRAPHICS 2018

This work was financed by
REGIONE AUTÒNOMA DE SARDIGNA
REGIONE AUTONOMA DELLA SARDEGNA
POR FESR SARDEGNA 2014-2020
TDM
POR FESR 2014 2020/1.2.2 Del 66/14, 13/12/2016
CRS4
UNIONE EUROPEA
Fondo europeo di sviluppo regionale
REPUBBLICA ITALIANA

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!)

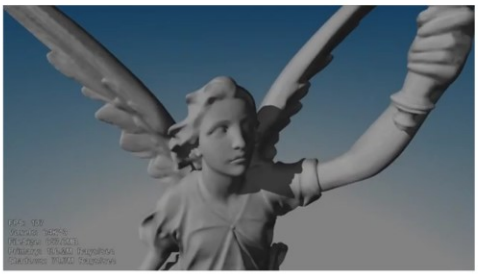
Related Work

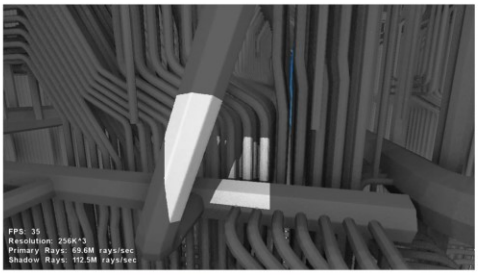
- Very hot topic with lots of contributions, not only DAGs
  - **Voxelization**
    [Crassin and Green, 2012] [Pätzold and Kolb, 2015] …
  - **Voxelized representation and render**
    [Laine and Karras, 2011 & 2012] [Crassin et al. 2009, 2012] [Schnabel and Klein, 2006] …
  - **Trees encoding & hashing**
    [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

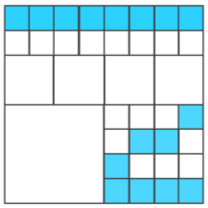
- **Advantages**
  - ~ 50% more compact
  - Similar tracing speed
  - Able to build massive models

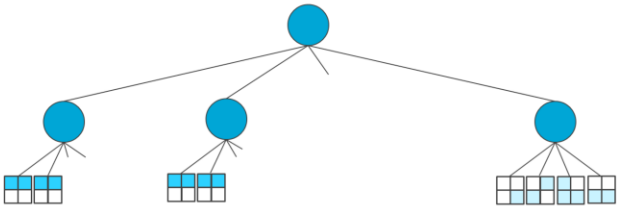




Quick recap – Sparse Voxel Octrees (SVO)

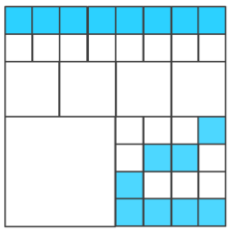
- Introduce sparsity in a full-octree
  - Compression by avoiding to encode empty space
  - Grows quadratically with resolution
  - Very fast traversal methods
- Encoding requires 1 pointer per node

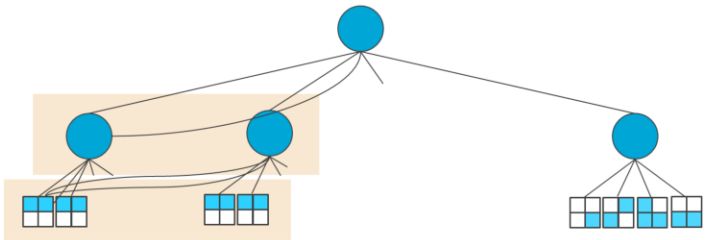




Quick recap – Sparse Voxel DAG (SVDAG)

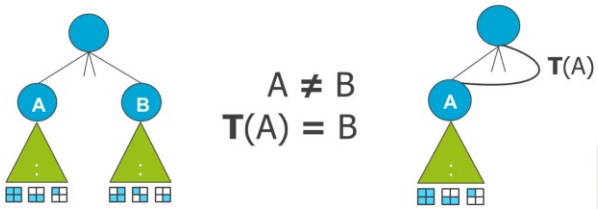
- Avoiding multiple encodings of same subtrees [Kämpe et al. 2013]
  - Tree becomes a directed graph
  - Same traveral method as SVOs
- Up to x100 or more reduced number of nodes SVOs
- Encoding requires 1 pointer per each node child





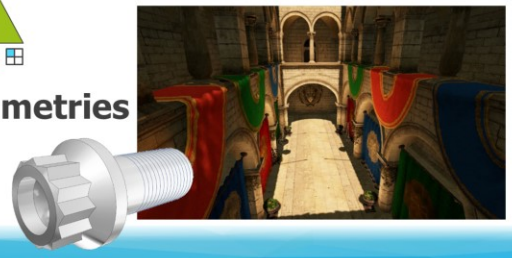
SSVDAGs approach

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



Chosen transform: **Reflection Symmetries**

- Geometric meaning
- Expected in real-world scenes

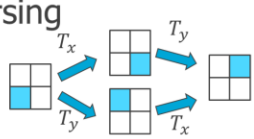


Reflection Symmetries

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

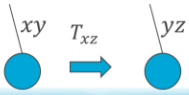
- Useful properties for building and traversing
  - Order of application does not matter

$$T_x(T_y(N)) = T_y(T_x(N))$$

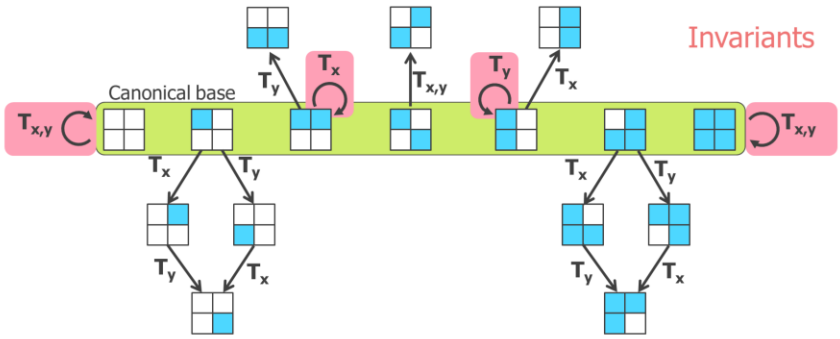


  - Cyclic (applying just means inverting the tag)

$$T_x(T_x(N)) = N$$



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

Building process – Leaves level

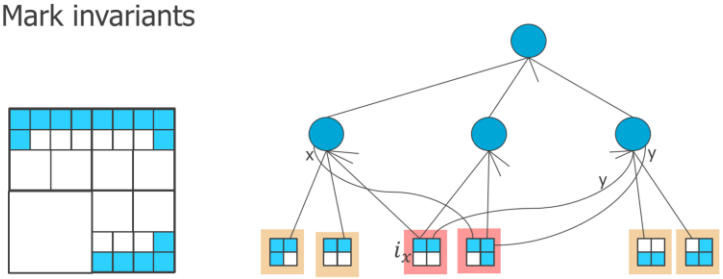
- Symmetry detected just using the voxel bit mask

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

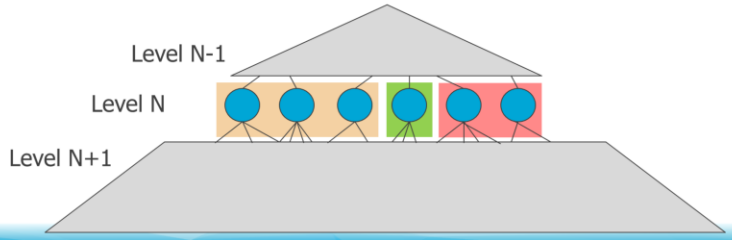


Building process – Inner Levels

- 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

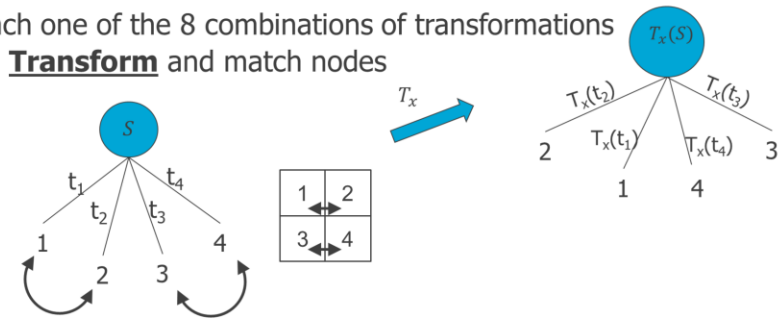


Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster

   For each one of the 8 combinations of transformations

   <u>Transform</u> and match nodes

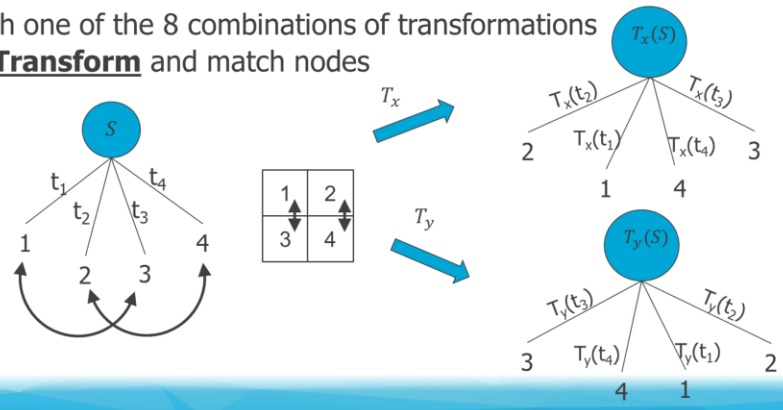


Building process – Inner Levels

1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster

   For each one of the 8 combinations of transformations

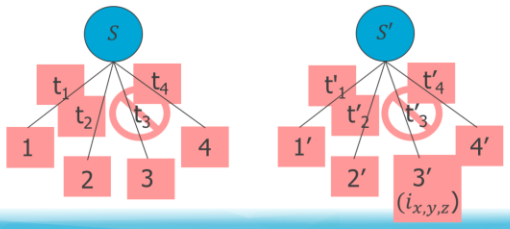
   <u>Transform</u> and match nodes



Building process – Inner Levels

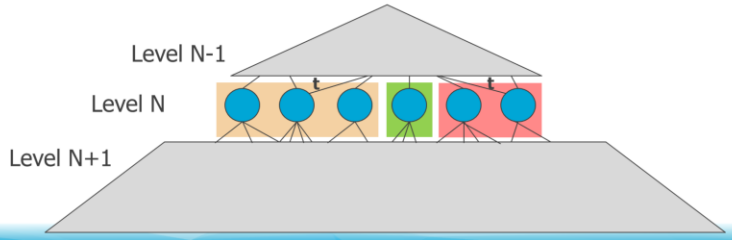
1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and **match** nodes

For each child pointer
Check if both are equal
If pointed node is not invariant,
check transformation tag



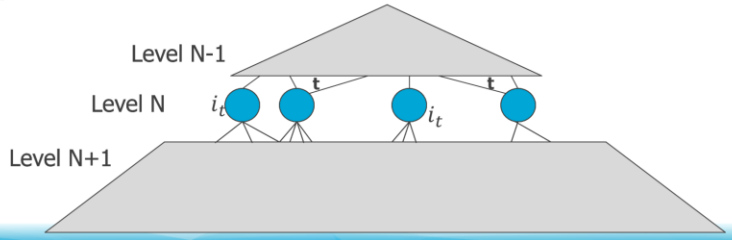
Building process – Inner Levels

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

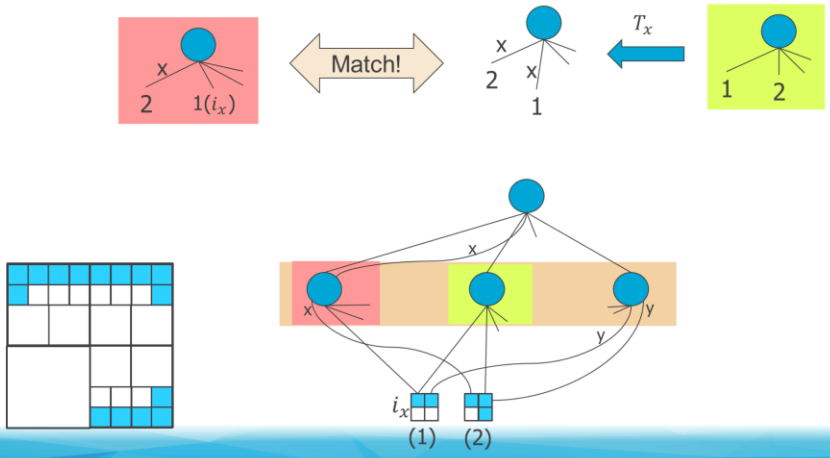


Building process – Inner Levels

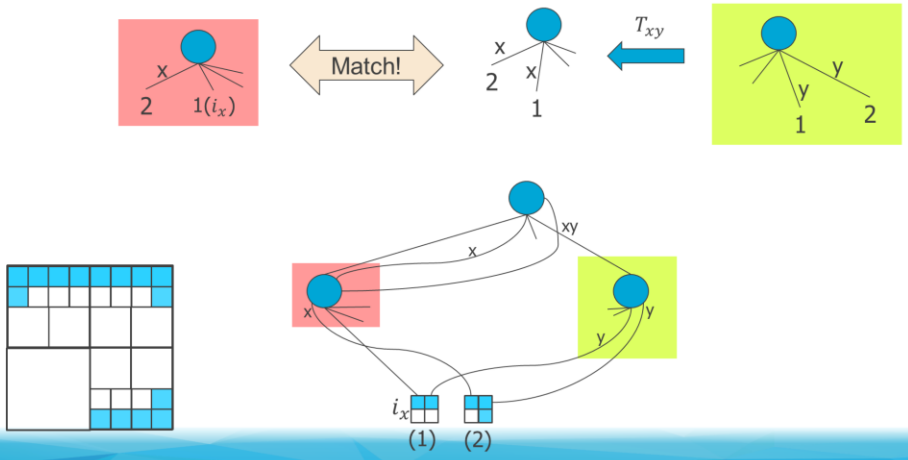
1. Sort & cluster nodes, using keys of sorted pointers
2. For each cluster
   For each one of the 8 combinations of transformations
   Transform and match nodes
3. Merge matches and update pointers from upper level
4. Update invariants



Building process – Example

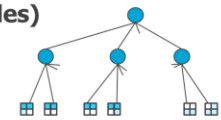


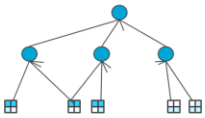
Building process – Example



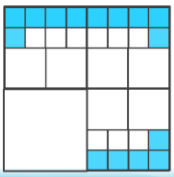
Building process – Example

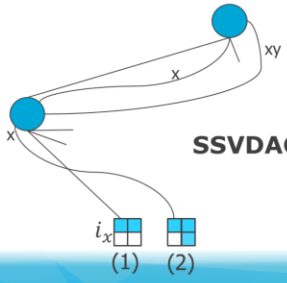
SVO (10 nodes)





SVDAG (9 nodes)





SSVDAG (4 nodes)

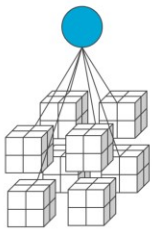
Compact encoding

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

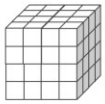
- **Steps**
  - Leaf grouping
  - Frequency-based pointers compaction
  - Variable bitrate memory alignment

Compact encoding

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



V.S.

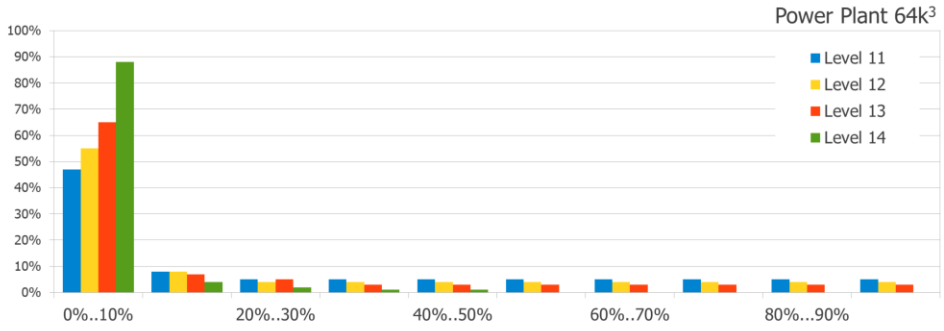


Fixed 8 bytes

Min 1 child: 32bits + 8bits = **5 bytes**
Max 8 children: 8 * (32bits + 8 bits) = **40 bytes**

Compact encoding

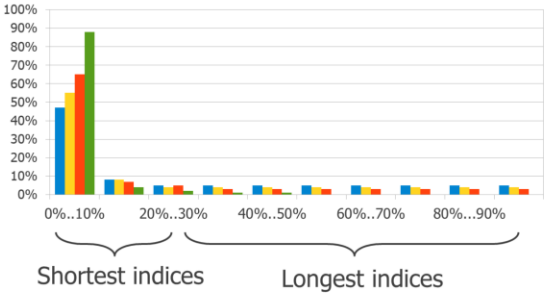
- Frequency-based pointer compaction
  - Distribution of references to nodes highly skewed



Compact encoding

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

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



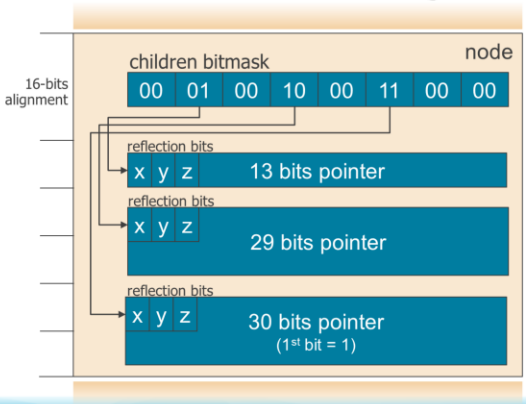
Compact encoding

Leaves nodes encoding

64 bits
4^3 leaf

64 bits
4^3 leaf

Inner nodes encoding



00	No child
01	13 bits pointer child
10	29 bits pointer child
11	30 bits pointer child

Total node size

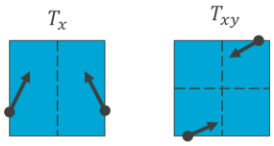
Min: 4 bytes (1 "13-bits" child)
Max: 34 bytes (8 "30-bits" children)

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

```
transformedIdx = currentTransform XOR selectedChildIdx
```

  - **Transform the ray**
    - Reflect ray's origin (voxel entry point)
    - Inverse direction



Results – Datasets

Scanned scenes

Lucy (28 Mtris)

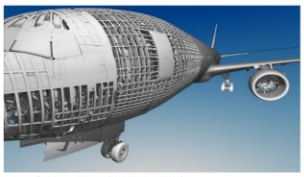
David (56 Mtris)

Gaming scenes

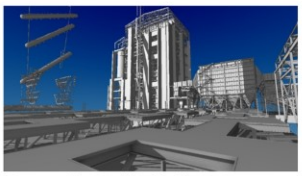
Crytek Sponza (282 Ktris)

San Miguel (7.8 Mtris)

CAD scenes

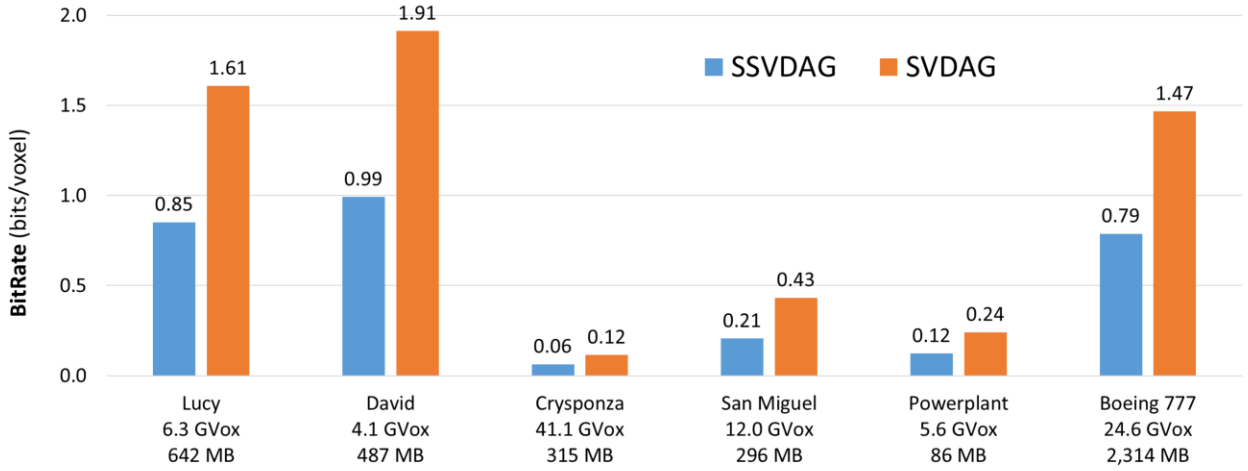


Boeing 777 (350 Mtris)



Power Plant (12 Mtris)

Results – Compression performance ($64k^3$)



Results – Compression performance

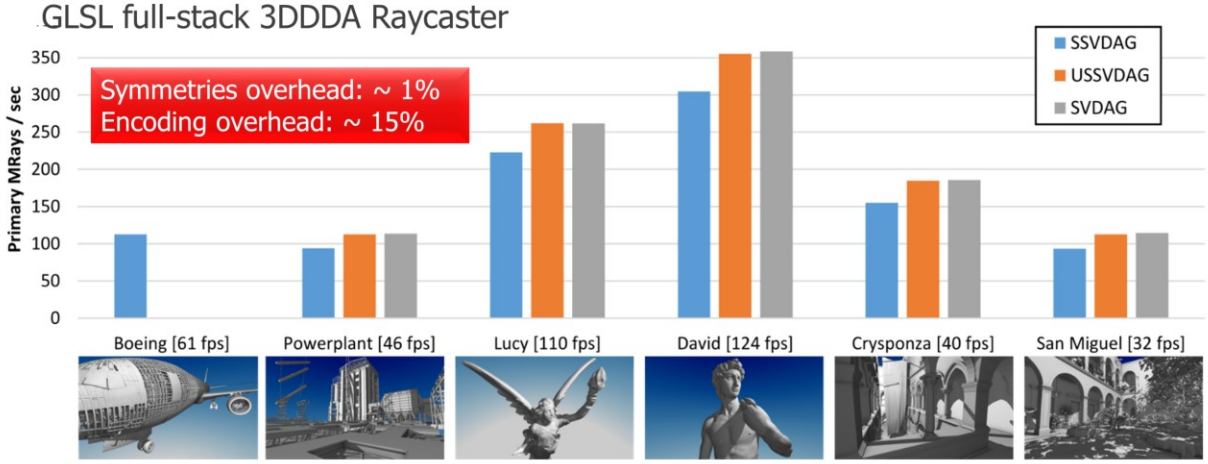
- Average node reduction ***only*** **by symmetries**: **~21%**
- Average data memory size reduction ***only*** **by encoding**: **~23%**

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

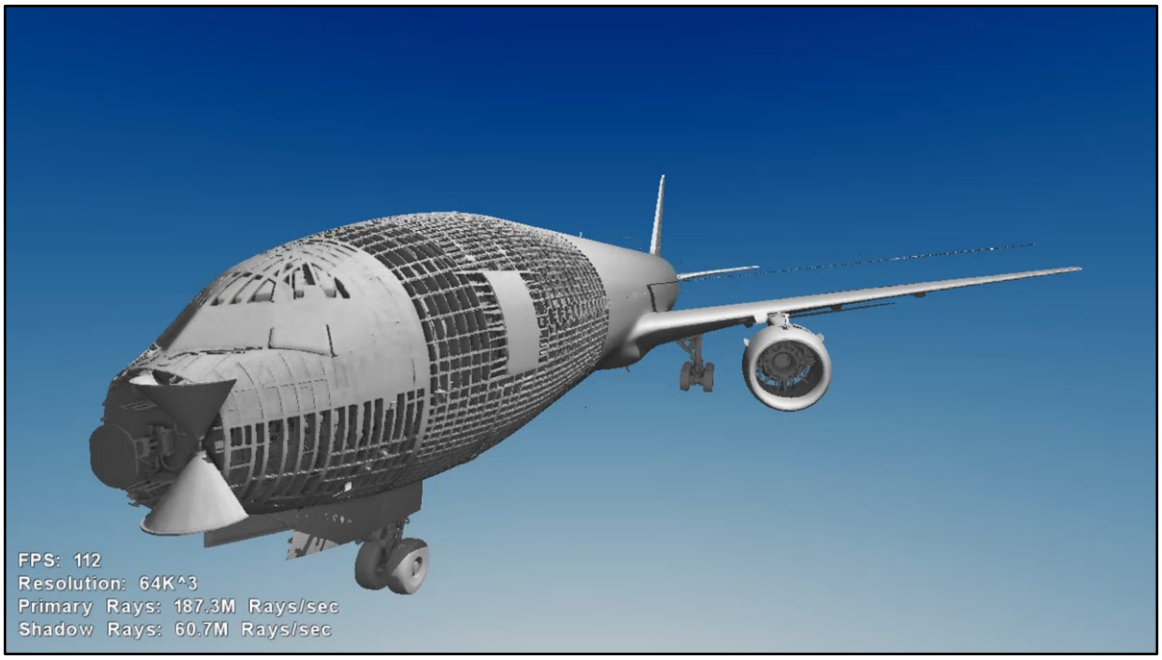
- Conversion time performance not aimed for real-time
  - Few seconds for lowest resolutions ($2k^3$, $4k^3$, $8k^3$)
  - Up to few minutes for biggest

Results – Rendering

GLSL full-stack 3DDDA Raycaster



720p – Nvidia GTX 980


FPS: 112
Resolution: 64K^3
Primary Rays: 187.3M Rays/sec
Shadow Rays: 60.7M Rays/sec

Summary

- **Efficient lossless encoding** of voxelizated 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://jcgt.org/published/0006/02/01/