Fast Parallel Construction of High-Quality Bounding Volume Hierarchies

Tero Karras
Timo Aila
Ray tracing comes in many flavors

Interactive apps
1M–100M rays/frame

Architecture & design
100M–10G rays/frame

Movie production
10G–1T rays/frame

© Activision 2009, Game trailer by Blur Studio

Courtesy of Delta Tracing

Lucasfilm Ltd.™, Digital work by ILM

© Activision 2009, Game trailer by Blur Studio

Courtesy of Dassault Systèmes

Courtesy of Columbia Pictures
Effective performance

\[ \text{effective ray tracing performance} = \frac{\text{number of rays}}{\text{rendering time}} \]
Effective performance

Effective ray tracing performance = \frac{\text{number of rays}}{\text{rendering time}}

Rendering time = \text{time to build BVH} + \frac{\text{number of rays}}{\text{ray throughput}}

“speed”

“quality”
Effective performance

**Effective ray tracing performance** = \( \frac{\text{number of rays}}{\text{rendering time}} \)

**Rendering time** = \( \text{time to build BVH} + \frac{\text{number of rays}}{\text{ray throughput}} \)

Interactive apps, Architecture & design, Movie production

- Speed matters
- Both matter
- Quality matters

Mrays/s vs. effective performance vs. number of rays per frame
Effective performance

- SODA (2.2M tris)
- NVIDIA GTX Titan
- Diffuse rays

![Graph](https://via.placeholder.com/150)

*Number of rays per frame vs. effective performance*
Effective performance

SBVH
[Stich et al. 2009]
(CPU, 4 cores)

Build time dominates

number of rays per frame

Mrays/s

effective performance

1M 10M 100M 1G 10G 100G 1T
Effective performance

HLBVH
[Garanzha et al. 2011]
(GPU)

SBVH
[Stich et al. 2009]
(CPU, 4 cores)
Effective performance

HLBVH
[Garanzha et al. 2011]
(GPU)

Our method
(GPU)

SBVH
[Stich et al. 2009]
(CPU, 4 cores)

number of rays per frame

Mrays/s

0 50 100 150 200 250 300 350 400 450

1M 10M 100M 1G 10G 100G 1T

effective performance
Effective performance

- Best quality–speed tradeoff for wide range of applications

30M–500G rays/frame

97% of SBVH

Effective performance vs. number of rays per frame

Mrays/s

0 50 100 150 200 250 300 350 400 450

1M 10M 100M 1G 10G 100G 1T

Best quality–speed tradeoff for wide range of applications
Treelet restructuring

Idea
- Build a low-quality BVH
- Optimize its node topology
- Look at multiple nodes at once
Treelet restructuring

- **Idea**
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- **Treelet**
  - Subset of a node’s descendants
Treelet restructuring

- Idea
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- Treelet
  - Subset of a node’s descendants
Treelet restructuring

Idea
- Build a low-quality BVH
- Optimize its node topology
- Look at multiple nodes at once

Treelet
- Subset of a node’s descendants
- Grow by turning leaves into internal nodes
Treelet restructuring

- **Idea**
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- **Treelet**
  - Subset of a node’s descendants
  - Grow by turning leaves into internal nodes
Treelet restructuring

- **Idea**
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- **Treelet**
  - Subset of a node’s descendants
  - Grow by turning leaves into internal nodes
Treelet restructuring

**Idea**
- Build a low-quality BVH
- Optimize its node topology
- Look at multiple nodes at once

**Treelet**
- Subset of a node’s descendants
- Grow by turning leaves into internal nodes
Treelet restructuring

- **Idea**
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- **Treelet**
  - Subset of a node’s descendants
  - Grow by turning leaves into internal nodes
Treelet restructuring

**Idea**
- Build a low-quality BVH
- Optimize its node topology
- Look at multiple nodes at once

**Treelet**
- Subset of a node’s descendants
- Grow by turning leaves into internal nodes
- Largest leaves → best results
Treelet restructuring

- Idea
  - Build a low-quality BVH
  - Optimize its node topology
  - Look at multiple nodes at once

- Treelet
  - Subset of a node’s descendants
  - Grow by turning leaves into internal nodes
  - Largest leaves → best results

- Valid binary tree in itself

\[ n - 1 = 6 \]
\[ n = 7 \]

treelet internal nodes

treelet leaves
Treelet restructuring

Idea

- Build a low-quality BVH
- Optimize its node topology
- Look at multiple nodes at once

Treelet

- Subset of a node’s descendants
- Grow by turning leaves into internal nodes
- Largest leaves → best results

Valid binary tree in itself

- Leaves can represent arbitrary subtrees of the BVH
Treelet restructuring

- Restructuring
  - Construct optimal binary tree for the same set of leaves
  - Replace old treelet
Treelet restructuring

- Restructuring
  - Construct optimal binary tree for the same set of leaves
  - Replace old treelet

- Reuse the same nodes
  - Update connectivity and AABBs
  - New AABBs should be smaller
Treelet restructuring

- Restructuring
  - Construct optimal binary tree for the same set of leaves
  - Replace old treelet

- Reuse the same nodes
  - Update connectivity and AABBs
  - New AABBs should be smaller
Treelet restructuring

- Restructuring
  - Construct optimal binary tree for the same set of leaves
  - Replace old treelet

- Reuse the same nodes
  - Update connectivity and AABBs
  - New AABBs should be smaller
Treelet restructuring

- Restructuring
  - Construct optimal binary tree for the same set of leaves
  - Replace old treelet

- Reuse the same nodes
  - Update connectivity and AABBs
  - New AABBs should be smaller

- Perfectly localized operation
  - Leaves and their subtrees are kept intact
  - No need to look at subtree contents
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Input triangles

One triangle per leaf
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel LBVH [Karras 2012]

60-bit Morton codes for accurate spatial partitioning
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal [Karras 2012]

Restructure multiple treelets in parallel
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal [Karras 2012]
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal [Karras 2012]
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal
[Karras 2012]
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal
[Karras 2012]
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Parallel bottom-up traversal
[Karras 2012]
Processing stages

Initial BVH construction

Optimization

Post-processing

Parallel bottom-up traversal

[Karras 2012]

Strict bottom-up order
→ no overlap between treelets
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Rinse and repeat (3 times is plenty)
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Collapse subtrees into leaf nodes
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Collect triangles into linear lists
Prepare them for Woop’s intersection test [Woop 2004]
Processing stages

- Initial BVH construction
- Optimization
- Post-processing

Fast GPU ray traversal [Aila et al. 2012]
Processing stages

Triangle splitting

Initial BVH construction

Optimization

Post-processing

Fast GPU ray traversal
[Aila et al. 2012]
Processing stages

- Triangle splitting
- Initial BVH construction
- Optimization
- Post-processing
- Fast GPU ray traversal [Aila et al. 2012]

---

**DRAGON (870K tris)**
- NVIDIA GTX Titan
- 23.6 ms / 30.0 ms

---

<table>
<thead>
<tr>
<th>Stage</th>
<th>NVIDIA GTX Titan</th>
<th>No splits</th>
<th>30% splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial BVH construction</td>
<td>0.4 ms</td>
<td>5.4 ms / 6.6 ms</td>
<td>17.0 ms / 21.4 ms</td>
</tr>
<tr>
<td>Triangle splitting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td>1.2 ms</td>
<td></td>
<td>1.6 ms / 1.6 ms</td>
</tr>
<tr>
<td>Post-processing</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cost model

- Surface area cost model
  
  \[ SAH := C_i \sum_{n \in I} \frac{A(n)}{A(\text{root})} + C_t \sum_{l \in L} \frac{A(l)}{A(\text{root})} N(l) \]

- Track cost and triangle count of each subtree

- Minimize SAH cost of the final BVH
  
  - Make collapsing decisions already during optimization
    → Unified processing of leaves and internal nodes
Optimal restructuring

- Finding the optimal node topology is NP-hard
  - Naive algorithm $\rightarrow \mathcal{O}(n!)$
  - Our approach $\rightarrow \mathcal{O}(3^n)$

- But it becomes very powerful as $n$ grows
  - $n = 7$ treelet leaves is enough for high-quality results

- Use fixed-size treelets
  - Constant cost per treelet
    $\rightarrow$ Linear with respect to scene size
### Optimal restructuring

<table>
<thead>
<tr>
<th>Treelet size</th>
<th>Layouts</th>
<th>Quality vs. SBVH *</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>15</td>
<td>78%</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td>85%</td>
</tr>
<tr>
<td>6</td>
<td>945</td>
<td>88%</td>
</tr>
<tr>
<td>7</td>
<td>10,395</td>
<td>97%</td>
</tr>
<tr>
<td>8</td>
<td>135,135</td>
<td>98%</td>
</tr>
</tbody>
</table>

* SODA (2.2M tris)

Number of unique ways for restructuring a given treelet

Ray tracing performance after 3 rounds of optimization
Optimal restructuring

Almost the same thing as tree rotations [Kensler 2008]

<table>
<thead>
<tr>
<th>Treelet size</th>
<th>Layouts</th>
<th>Quality vs. SBVH</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>15</td>
<td>78%</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td>85%</td>
</tr>
<tr>
<td>6</td>
<td>945</td>
<td>88%</td>
</tr>
<tr>
<td>7</td>
<td>10,395</td>
<td>97%</td>
</tr>
<tr>
<td>8</td>
<td>135,135</td>
<td>98%</td>
</tr>
</tbody>
</table>

Variation a lot between scenes

Limited options during optimization → easy to get stuck in a local optimum

* SODA (2.2M tris)
Optimal restructuring

<table>
<thead>
<tr>
<th>Treelet size</th>
<th>Layouts</th>
<th>Quality vs. SBVH *</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>15</td>
<td>78%</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td>85%</td>
</tr>
<tr>
<td>6</td>
<td>945</td>
<td>88%</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td><strong>10,395</strong></td>
<td><strong>97%</strong></td>
</tr>
<tr>
<td>8</td>
<td>135,135</td>
<td>98%</td>
</tr>
</tbody>
</table>

* SODA (2.2M tris)

Can still be implemented efficiently

Consistent across scenes

Surely one of these will take us forward 😊

Further improvement is marginal
Algorithm

- Dynamic programming
  - Solve small subproblems first
  - Tabulate their solutions
  - Build on them to solve larger subproblems

Subproblem:
- What’s the best node topology for a *subset* of the leaves?
Algorithm

input: set of \( n \) treelet leaves

\[
\text{for } k = 2 \text{ to } n \text{ do}
\]

\[
\text{for each subset of size } k \text{ do}
\]

\[
\text{for each way of partitioning the leaves do}
\]

look up subtree costs

calculate SAH cost

end for

record the best solution

end for

end for

reconstruct optimal topology

Process subsets from smallest to largest

Record the optimal SAH cost for each
Algorithm

**input:** set of $n$ treelet leaves

for $k = 2$ to $n$ do
  
  for each subset of size $k$ do
    
    for each way of partitioning the leaves do
      
      look up subtree costs
      calculate SAH cost
    end for
  end for
  record the best solution

end for

reconstruct optimal topology

Exhaustive search: assign each leaf to left/right subtree

We already know how much the subtrees will cost

Backtrack the partitioning choices
### Scalar vs. SIMD

<table>
<thead>
<tr>
<th>Scalar processing</th>
<th>SIMD processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each thread processes one treelet</td>
<td>32 threads collaborate on the same treelet</td>
</tr>
<tr>
<td>Need many treelets in flight</td>
<td>Need few treelets in flight</td>
</tr>
<tr>
<td>✗ Spills to off-chip memory</td>
<td>✓ Data fits in on-chip memory</td>
</tr>
<tr>
<td>✗ Doesn’t scale to small scenes</td>
<td>✓ Easy to fill the entire GPU</td>
</tr>
<tr>
<td>✓ Trivial to implement</td>
<td>✗ Need to keep all threads busy</td>
</tr>
</tbody>
</table>

Parallelize over subproblems using a pre-optimized processing schedule (details in the paper)
## Scalar vs. SIMD

<table>
<thead>
<tr>
<th>Scalar processing</th>
<th>SIMD processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each thread processes one treelet</td>
<td>32 threads collaborate on the same treelet</td>
</tr>
<tr>
<td>Need many treelets in flight</td>
<td>Need few treelets in flight</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>✗ Spills to off-chip memory</td>
<td>✓ Data fits in on-chip memory</td>
</tr>
<tr>
<td>✗ Doesn’t scale to small scenes</td>
<td>✓ Easy to fill the entire GPU</td>
</tr>
<tr>
<td>✓ Trivial to implement</td>
<td>✓ Possible to keep threads busy</td>
</tr>
</tbody>
</table>

- Each thread processes one treelet
- Need many treelets in flight

- 32 threads collaborate on the same treelet
- Need few treelets in flight

- Spills to off-chip memory
- Doesn’t scale to small scenes
- Trivial to implement
Quality vs. speed

- Spend less effort on bottom-most nodes
  - Low contribution to SAH cost
  - Quick convergence

- Additional parameter $\gamma$
  - Only process subtrees that are large enough
  - Trade quality for speed

- Double $\gamma$ after each round
  - Significant speedup
  - Negligible effect on quality
Triangle splitting

- Early Split Clipping [Ernst and Greiner 2007]
- Split triangle bounding boxes as a pre-process

Large triangle

Bounding box is not a good approximation

Split it!
Triangle splitting

- Early Split Clipping [Ernst and Greiner 2007]
  - Split triangle bounding boxes as a pre-process

Resulting boxes provide a tighter bound

Keep going until they are small enough
Triangle splitting

- Early Split Clipping [Ernst and Greiner 2007]
  - Split triangle bounding boxes as a pre-process

Keep going until they are small enough
Triangle splitting

- Early Split Clipping [Ernst and Greiner 2007]
- Split triangle bounding boxes as a pre-process

Treat each box as a separate primitive

Triangle itself remains the same
Triangle splitting

- Shortcomings of pre-process splitting
  - Can hurt ray tracing performance
  - Unpredictable memory usage
  - Requires manual tuning

- Improve with better heuristics
  - Select good split planes
  - Concentrate splits where they matter
  - Use a fixed split budget
Split plane selection

- Reduce node overlap in the initial BVH

Root node partitions the scene at its spatial median
Split plane selection

- Reduce node overlap in the initial BVH

Left child

Right child
Split plane selection

- Reduce node overlap in the initial BVH

If a triangle crosses the plane...

...the bounding boxes will overlap

Use the same spatial median as a split plane
Split plane selection

- Reduce node overlap in the initial BVH

Use the same spatial median as a split plane

No overlap
Split plane selection

- Reduce node overlap in the initial BVH

Splitting one triangle does not help much

Need to split them all to get the benefits
Split plane selection

- Reduce node overlap in the initial BVH
Split plane selection

- Reduce node overlap in the initial BVH

Same reasoning holds on multiple levels
Split plane selection

- Reduce node overlap in the initial BVH

Look at all spatial median planes that intersect a triangle

Split it with the dominant one
Algorithm

1. Allocate memory for a fixed split budget
Algorithm

1. Allocate memory for a fixed split budget

2. Calculate a *priority value* for each triangle
Algorithm

1. Allocate memory for a fixed split budget

2. Calculate a *priority value* for each triangle

3. Distribute the split budget among triangles
   - Proportional to their priority values
Algorithm

1. Allocate memory for a fixed split budget

2. Calculate a *priority value* for each triangle

3. Distribute the split budget among triangles
   - Proportional to their priority values

4. Split each triangle recursively
   - Distribute remaining splits according to the size of the resulting AABBs
Split priority

\[ \text{priority} = \left( 2^{-\text{level}} \cdot (A_{aabb} - A_{ideal}) \right)^{1/3} \]

- Crosses an important spatial median plane?
- Has large potential for reducing surface area?

Concentrate on triangles where both apply...
...but leave something for the rest, too
Results

- Compare against 4 CPU and 3 GPU builders
  - 4-core i7 930, NVIDIA GTX Titan
  - Average of 20 test scenes, multiple viewpoints
Ray tracing performance

- SweepSAH = 100%
- SBVH = 131%

Bar chart showing:
- SweepSAH [MacDonald]
- SBVH [Stich]
- Tree rotations [Kensler]
- Iterative reinsertion [Bittner]

High-quality CPU builders
Ray tracing performance

SBVH = 131%

Almost 2×

67% – 69%

Fast GPU builders

Ray tracing performance

- SweepSAH [MacDonald]
- SBVH [Stich]
- Tree rotations [Kensler]
- Iterative reinsertion [Bittner]
- TRBVH
- TRBVH +30% split
- LBVH [Karras]
- HLBVH [Garanzha]
- GridSAH [Garanzha]

Our method
- No splits
- 30% splits
Ray tracing performance

- 91% of SBVH
- 96% of SweepSAH

Graph comparing performance of various ray tracing methods:
- SweepSAH [MacDonald]
- SBVH [Stich]
- Tree rotations [Kensler]
- Iterative reinsertion [Bittner]
- TRBVH
- TRBVH +30% split
- LBVH [Karras]
- HLBVH [Garanzha]
- GridSAH [Garanzha]
Effective performance

- **SweepSAH** [MacDonald]
- **Tree rotations** [Kensler]
- **Iterative reinsertion** [Bittner]

**Not Pareto-optimal**

![Graph showing performance comparison between different methods](image)

*number of rays per frame*
Effective performance

number of rays per frame

SweepSAH [MacDonald]

SBVH [Stich]
Effective performance

- LBVH [Karras]
- HLBVH [Garanzha]
- GridSAH [Garanzha]
- SweepSAH [MacDonald]

Not Pareto-optimal

number of rays per frame
Effective performance

Number of rays per frame

SBVH [Stich]

LBVH [Karras]

SweepSAH [MacDonald]
Effective performance

- Our method (30% splits)
- Our method (no splits)
- LBVH [Karras]
- SweepleSAH [MacDonald]
- SBVH [Stich]

number of rays per frame
Effective performance

Below 7M → LBVH
Above 60G → SBVH

7M–60G rays/frame → our method is the best choice

number of rays per frame
Conclusion

- General framework for optimizing trees
  - Inherently parallel
  - Approximate restructuring → larger treelets?

- Practical GPU-based BVH builder
  - Best choice in a large class of applications
  - Adjustable quality–speed tradeoff

- Will be integrated into NVIDIA OptiX
Thank you

Acknowledgements
- Samuli Laine
- Jaakko Lehtinen
- Sami Liedes
- David McAllister
- Anonymous reviewers
- Anat Grynberg and Greg Ward for Conference
- University of Utah for Fairy
- Marko Dabrovic for Sibenik
- Ryan Vance for Bubs
- Samuli Laine for Hairball and Vegetation
- Guillermo Leal Laguno for SanMiguel
- Jonathan Good for Arabic, Babylonian and Italian
- Stanford Computer Graphics Laboratory for Armadillo, Buddha and Dragon
- Cornell University for Bar
- Georgia Institute of Technology for Blade