KD-Tree for Animated Scene

Copyright © 2010 by Yong Cao
KD-Tree for Animated Scene

[Diagram showing a KD-tree with nodes labeled A, B, C, D and regions Y1, Y2, X, t_min, t_max.]

Copyright © 2010 by Yong Cao
Another Case

A large tree structure change.
A totally new tree!
Solution

Solution 1: Rebuild kd-tree each frame
- Rebuild kd-tree in a lazy manner, approximate SAH (Surface Area Heuristics) [Hunt et al. 06]
- Can just move objects bounding boxes around and transform rays (for hierarchical movement) [Wald et al. 03]
- Motion decomposition, fuzzy kd-trees [Günther et al. 06]

Solution 2: use different hierarchical structure
Hierarchical Representations for Dynamic Ray Tracing

- **Bounding volume hierarchies (BVHs)**
  - [Wald et al. 06b, Boulos et al. 06, Lauterbach et al. 06]

- **Grids**
  - [Wald et al. 06a]
Ray Tracing Dynamic Scenes Using BVHs

[Lauterbach et al. 06]

Dinesh Manocha, Christian Lauterbach
University of North Carolina at Chapel Hill

Copyright © 2010 by Yong Cao
Bounding Volume Hierarchies: BVHs

- Tree of bounding volumes (sphere, AABB, OBB, k-DOP, spherical shells, etc.)
- Each bounding volume encloses “nearby” primitives
- Parent node primitives are union of children node primitives
Spatial partitioning vs. Object Hierarchies

- **Spatial partitioning:**
  - space is subdivided into disjoint regions (e.g. grid, kd-tree, octree, ...)

- **Object hierarchy:**
  - groups or clusters of objects/primitives are subdivided (BVH, s-kd-tree)
Spatial partitioning vs. Object Hierarchies

- Implications for ray tracing
  - Spatial partitioning: Objects referenced in multiple nodes (overlap in object space)
  - BVH Hierarchies: Nodes can overlap each other (overlap in 3D space)
  - Spatial partitioning allows easier front-to-back ordering
BVHs for intersection tests

- Widely used for intersection computations
  - Ray tracing
  - Visibility culling: view frustum and occlusion culling
  - Collision and proximity computations
  - Other applications
Pretty simple:
- Start from root
- If ray intersects AABB, try all children, too:
  - is inner node: recurse on both children
  - is leaf node: intersect with primitive(s)
Naïve implementation far slower than kd-tree!
Why are BVHs slower?

- Intersection test more costly
  - Up to 6 ray-plane intersections for AABB (slabs test)
  - Just 1 for kd-tree

- No front-to-back ordering
  - Cannot stop after finding first hit

- Nodes take more space
  - 32 bytes vs. 8 bytes
On the other hand...

- AABBs can provide tighter fit automatically
  - No empty leaves, tree does not need to be as deep
  - Primitives only referenced once
    ⇒ less nodes in hierarchy
- #nodes known in advance (2n-1)
  - (if 1 primitive/leaf)
More Importantly ...

- AABBs can provide tighter fit automatically
  - No empty leafs, tree does not need to be as deep
  - Primitives only referenced once
    - Less nodes in hierarchy
- #nodes known in advance (2n-1)
  - (if 1 primitive/leaf)
- Can be updated easily!
Hierarchy updates

- What does updating mean?
  - Underlying geometry changes
  - Update will ensure correctness of hierarchy without rebuilding it

- Should be faster than rebuild
Dynamic Scenes: updating BVHs

- Post-order traversal of BVH
  - Update children's AABB, then update own
  - At leaf level, update from primitives
  - Also update additional information such as axis

- O(n) time
  - Usually a few ms for small scenes
  - May become too long for large models!
Dynamic scenes: BVH degradation

- Quality of BVH may decrease over animation
  - Update does not change tree topology
  - Rebuild may be necessary
  - How to detect?

- In worst-case scene:
  - Performance dropping an order of magnitude over 20 animation frames
  - Not as bad for normal scenes, though
Quality degradation

- Use heuristic to detect degradation
- Assume performance lower when BVHs contain lots of empty space:

```
  child 1
  
  child 2
  
  after build

  child 1
  child 2
  
  after updates
```
Rebuild heuristic

How to measure quality?

- Use ratio of surface area parent to children
  - \( \frac{SA(\text{parent})}{SA(\text{child1}) + SA(\text{child2})} \)
  - Save on rebuild for each node (4 bytes/node)

- On each update: compare to initial value

- Sum up differences and normalize

- If above threshold: initiate rebuild
  - ~30-40% work well in practice
Results

Video
Ray Tracing Animated Scenes using Coherent Grid Traversal

[Wald et al. 06a]

I Wald, T Ize, A Kensler, A Knoll, S Parker
SCI Institute, University of Utah
Coherent Grid Traversal

- A new traversal techniques for uniform grids
- … that makes packet/frustum traversal compatible with grids
- … thus achieves performance competitive with fastest kd-trees
- … and which allows for per-frame rebuilds (dynamic scenes)
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

Copyright © 2010 by Yong Cao
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

BVH
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

- BVH
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

 BVH

Copyright © 2010 by Yong Cao
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

Copyright © 2010 by Yong Cao
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

Copyright © 2010 by Yong Cao
What’s so special about grids?

➢ Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

![BVH](image1)
![Kd-tree](image2)
![Octree](image3)
![Grid](image4)
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

Copyright © 2010 by Yong Cao
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

[Images of BVH, Octree, Grid, Kd-tree structures]
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures
What’s so special about grids?

- Since 70’ies: Lots of different RT data structures

→ Of all these, grid is only that is not hierarchical!
What’s so special about grids?

- Grid is not hierarchical…
  - Much simpler to build (similar to 3D rasterization, very fast)
    - Build-times in the paper: 2.2M “Soda Hall” in 110 ms

- Ideally suited for handling dynamic scenes
  - Full rebuild every frame, no restrictions at all!
What is so special about dynamic scenes?

- All of the recent advancements of RT are for kd-trees!
  - Pre-2000: Tie between grids and kd-trees...
  - [Wald ’01]: New concept → “coherent ray tracing” (for kd-tree)
    - Trace “packets” of coherent rays → 10x faster than single rays
  - [Woop ’05]: First RT hardware prototype → RPU (for kd-tree)
  - [Reshetov ’05]: New concept → “multilevel ray tracing” (kd-tree)
    - Trace packets using bounding frusta → another 10x faster than CRT!
- But: (good) kd-trees are (too) costly to build…
Ray Tracing & Dynamic Scenes

- SIGGRAPH ‘05: Dynamic Scenes huge problem
  - Ray tracing has become very fast (MLRT: ~100fps)
  - If ray tracing is to ever replace rasterization, it must support dynamic scenes (games…)
  - But: All our fast RT algos are for kd-trees…
  - … and kd-trees can’t do dynamic scenes …
Ray Tracing & Dynamic Scenes

- SIGGRAPH ‘05: Dynamic Scenes huge problem
- Since then, lots of research
  - Lazy kd-tree construction (Razor [Stoll, Mark ‘06])
  - Fast BVH and kd-tree construction (yet unpublished)
  - Motion decomposition [Günther et al. ‘06]
  - Dynamic BVHs [Wald et al. ‘06, Lauterbach et al. ’06]
  - Hybrid BVH/kd-trees [Woop ‘06, Havran ‘06, Wachter ‘06, ...]
  - Coherent Grid Traversal [Wald et al. ’06]
Using grids for dynamics – Where’s the problem?

- 2005: Grid too slow to traverse (vs kd-tree)…
- Fact: Fast RT needs “packets” & “frusta” concepts
  - Traverse multiple packets over same node of DS
- Rather simple for hierarchical data structures…
  - Test both children in turn for overlap w/ packet
  - If child overlaps: traverse it, else: skip it.
    - (it’s as simple as that)
- … but not for grids
Packets & grids: “Non-trivial task”
- In which order to test the nodes? ABCD or ABDC?
- What to do when packet diverges?
  - 3DDDA etc break in that case…
- Split diverging packet?
  - Quickly degenerates to single-ray traversal…
- Fix by re-merging packets?
  - Non-trivial & costly …
Coherent Grid Traversal

- First: Transform all rays into “canonical grid space”
  - i.e., \([0,0,0]-[Nx,Ny,Nz]\)
Coherent Grid Traversal

Idea: Consider only frustum, not “set of rays”
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
Coherent Grid Traversal

- Idea: Consider only frustum, not "set of rays"
  - Traverse "slice by slice" instead of "cell to cell"
  - Pick "major traversal axis" (e.g., max component of 1st ray)
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
- Traverse “slice by slice” instead of “cell to cell”
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
- Traverse “slice by slice” instead of “cell to cell”
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
- Traverse “slice by slice” instead of “cell to cell”
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
- Traverse “slice by slice” instead of “cell to cell”
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
- Traverse “slice by slice” instead of “cell to cell”
- For each slice, compute frustum/slice overlap
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
  - Traverse “slice by slice” instead of “cell to cell”
  - For each slice, compute frustum/slice overlap
Coherent Grid Traversal

- **Idea:** Consider only frustum, not “set of rays”
  - Traverse “slice by slice” instead of “cell to cell”
  - For each slice, compute frustum/slice overlap
    - Float-to-int gives overlapped cell IDs
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
  - Traverse “slice by slice” instead of “cell to cell”
  - For each slice, compute frustum/slice overlap
    - Float-to-int gives overlapped cell IDs
    - Intersect all cells in given slice
Coherent Grid Traversal

- Idea: Consider only frustum, not “set of rays”
  - Traverse “slice by slice” instead of “cell to cell”
  - For each slice, compute frustum/slice overlap
    - Float-to-int gives overlapped cell IDs
    - Intersect all cells in given slice
  - Loop: incrementally compute next slice’s overlap box
    - 4 additions…
Coherent Grid Traversal

- **Idea:** Consider only frustum, not “set of rays”
  - Traverse “slice by slice” instead of “cell to cell”
  - For each slice, compute frustum/slice overlap
    - Float-to-int gives overlapped cell IDs
    - Intersect all cells in given slice
  - Loop: incrementally compute next slice’s overlap box
    - 4 additions…
Expensive setup phase
- Transform rays to canonical grid coordinate system
- Determine major march direction (simple)
- Compute min/max bounding planes (slopes and offsets)
- Compute first and last slice to be traversed (full frustum clip)

But: Very simple traversal step
- Overlap box update: 4 float additions (1 SIMD instruction)
- Get cell IDs: 4 float-to-int truncations (SIMD…)
- Loop over overlapped cells (avg: 1.5-2 cells per slice)
Traversal fast, but ...

- Grid usually less efficient than kd-tree
Traversal fast, but …

- Grid usually less efficient than kd-tree
- Cannot adapt to geometry as well → more intersections
Traversal fast, but ...

- Grid usually less efficient than kd-tree
  - Cannot adapt to geometry as well $\rightarrow$ more intersections
  - Tris straddle many cells $\rightarrow$ re-intersection
Grid usually less efficient than kd-tree
- Cannot adapt to geometry as well → more intersections
- Tris straddle many cells → re-intersection

First sight: Frustum makes it worse…
Traversal fast, but …

- Grid usually less efficient than kd-tree
  - Cannot adapt to geometry as well → more intersections
  - Tris straddle many cells → re-intersection
- First sight: Frustum makes it worse…
  - Rays isec tris outside “their” cells
Traversal fast, but ...

- Grid usually less efficient than kd-tree
  - Cannot adapt to geometry as well → more intersections
  - Tris straddle many cells → re-intersection
- First sight: Frustum makes it worse...
  - Rays isec tris outside “their” cells
  - Re-isec aggravataed by width of frustum
Traversal fast, but …

- Grid usually less efficient than kd-tree
- First sight: Frustum makes it worse…
- But: Two easy fixes
Traversal fast, but ...

- Grid usually less efficient than kd-tree
- First sight: Frustum makes it worse...
- But: Two easy fixes
  - Bad culling $\rightarrow$ SIMD Frustum culling in Packet/Tri Isec [Dmitriev et al.]

Outside frustum $\rightarrow$ cull!
Traversal fast, but ... 

- Grid usually less efficient than kd-tree
- First sight: Frustum makes it worse...
- But: Two easy fixes
  - Bad culling $\rightarrow$ SIMD Frustum culling in Packet/Tri Isec [Dmitriev et al.]
  - Re-intersection: Mailboxing [Haines]

Mailbox detects re-intersection
**CGT efficiency**

- **Surprise: Mailboxing & Frustum culling very effective**
  - Both standard techniques, both limited success for kd-trees
  - Grid & Frustum: Exactly counter weak points of CGT ...
    - “Hand”
      - Grid w/o FC & MB: 14 M ray-tri isecs
      - Grid with FC & MB: .9 M ray-tri isecs (14x less)
      - Kd-tree: .85M ray-tri isecs (5% less than grid)

- And: cost indep of #rays $\rightarrow$ very cheap (amortize)
Results
Impact of Method: Compare to single-ray & kd-tree

- **Comparison to single-ray grid**
  - Fast single-ray traverser, macrocell if advantageous, ...
  - Speedup 6.5x to 20.9x, usually ~10x

- **Comparison to kd-tree**
  - To OpenRT: 2x-8x faster (2M Soda Hall: 4.5x)
  - To MLRT: ~3x slower (but much less optimized)
  - Tests performed on “kd-tree friendly” models
Overall Performance

- Build time: Usually affordable even on single CPU...
- Traversal results (1024^2, dual 3.2 GHz Xeon PC)
  - X/Y: X=raycast only; Y=raytrace+shade+texture+shadows

- "Hand" 16K triangles 34.5/15.3 fps
- "Poser" 78K triangles 15.8/7.8 fps
- "Toys" 11K triangles 29.3/10.2 fps
- "Marbles" 8.8K triangles 57.1/26.2 fps
- "Fairy" 174K triangles 3.4/1.2 fps
Discussion

- **Comparison to state-of-the-art BVH or kd-tree**
  - Somewhat harder to code and “get right” than, e.g., BVH
  - Usually somewhat slower (~1.5x-3x)
  - More susceptible to incoherence & teapot-in-stadium cases
    - Pure frustum tech.: Visits all cells in frustum even if not touched by any ray!

- **BUT:**
  - It works at all! (Who’d have thought 12m ago?)
  - ~10x faster than single-ray grid
  - Benefits better from additional coherence (4x AA at 2x cost)
  - “Maybe” better suited for regular data or special HW (Cell, GPUs)
  - Most flexible wrt dynamic → no limitation at all
Conclusion

- Have developed a new technique that
- Makes grid compatible with packets & frusta
- Is competitive with BVHs and kd-trees
- Most general in handling dynamic scenes
References


