Razor: Multi-resolution ray tracing for dynamic environments

William Mark,
Warren Hunt,
Gordon Stoll
How to make the Z-buffer obsolete

• Do everything a Z-buffer can do
• And, do some things better
  – Users must value the new capabilities
Do everything a Z-buffer can do

- 60 frames/sec
  - Complex scenes, shaders, and effects
  - Anti-aliased pixels (4x AA or more)
Do everything a Z-buffer can do

- 60 frames/sec
  - Complex scenes, shaders, and effects
  - Anti-aliased pixels (4x AA or more)
- Dynamic scenes
  - Any kind of motion
  - Simple interface for applications to use
Do everything a Z-buffer can do

- 60 frames/sec
  - Complex scenes, shaders, and effects
  - Anti-aliased pixels (4x AA or more)
- Dynamic scenes
  - Any kind of motion
  - Simple interface for applications to use
Do something better than the Z-buffer

- Conceptual simplicity
- Partially transparent surfaces
- Simple global illumination effects
  - Artifact-free hard shadows, reflection, refraction
Do something better than the Z-buffer

- Conceptual simplicity
- Partially transparent surfaces
- Simple global illumination effects
  - Artifact-free hard shadows, reflection, refraction
- Expensive global illum. and approximations
  - Soft shadows, ambient occlusion, …
  - Unifying theme: Many secondary rays
Do something better than the Z-buffer

- Conceptual simplicity
- Partially transparent surfaces
- Simple global illumination effects
  - Artifact-free hard shadows, reflection, refraction
- Expensive global illum. and approximations
  - Soft shadows, ambient occlusion, ...
  - Unifying theme: Many secondary rays
Razor: An experimental system

- Targeted at future interactive ray tracing
- Key architectural features of Razor:
  - Fast kd-tree build
    - On demand
    - From scene graph
  - Surfaces represented as grids of vertices
  - Multi-resolution
Razor team

Peter Djeu
Rui Wang
Warren Hunt
Ikrima Elhassan

Gordon Stoll – presenting next
William Mark
Outline for this talk

- Dynamic Scenes – Bill and Warren
- Secondary Rays – Gordon
- Anti-aliasing and displacement – Gordon
Dynamic scenes

Acceleration structure must be rebuilt or updated each frame

Image: Okan Arikan
Image: Guendelman et al, SIGGRAPH 2005
System issues for dynamic scenes

- What acceleration structure?
- How is it rebuilt or updated?
- How is it integrated with the rest of the system?

Must reconsider the role of the acceleration structure in a ray tracing system
Optimize for the common case

- Most geometry is not visible
  - (for complex scenes)
- Most movement is spatially coherent
  - Even “unstructured” movement such as particle systems.
- Most movement has frame-to-frame coherence
An efficient solution should:

- Only do work for visible geometry
- Use scene structure
  - Don’t reduce to a polygon soup
  - Use mesh structure
  - Use scene-graph or equivalent info
- Use frame-to-frame coherence
  - Update the sort rather than re-doing it
Trend is to trace more secondary rays

- Traversal cost will grow faster than build cost
- So, use a good acceleration structure
  - Even if it’s a bit more expensive to build
  - Spatially-adaptive structures are best
    (kd, B-kd, some BVH)
  - Use cost metric such as surface area heuristic
The Dilemma

- We need a “fast” acceleration structure
  - E.g. kd-tree built with surface area heuristic
- But:
  - Fast acceleration structures are hard to update
    - (although some recent progress on this)
  - Fast acceleration structures are not application friendly
    - Has anyone built a scene graph organized as a kd-tree??
Our solution: Use two data structures

Scene graph (a sloppy bounding-volume hierarchy)

kd-tree (or other high-quality acceleration struct)

Update each frame

Rebuild lazily each frame
Using two data structures avoids compromises

- Scene graph is:
  - natural for application to use
  - easy to update

- Kd-tree is efficient for ray traversal
Maintaining a scene graph is a well-understood task

- Modern Z-buffer rendering engines do it
- Must update (some) of the bounding volumes each frame
- Efficient hierarchical approaches:
  - Techniques from collision detection (Dinesh Manoch’s talk)
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph

Scene graph

kd-tree
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph

Scene graph

kd-tree

RAY
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from KD-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph.
Lazy build uses pointers from kd-tree to scene graph
Lazy build uses pointers from kd-tree to scene graph

Scene graph

kd-tree

And so on…

RAY
Scene graph hierarchy provides two benefits during kd-tree build

- Enables lazy build – skip unnecessary work
  - 16x more occluded polygons increases build time <5%
- Aggregates geometry – lowers sorting costs
  - Most useful near top of kd-tree
  - Sort 30 nodes instead of 10,000,000 polygons.
  - The scene graph provides an approximate pre-sort
Using scene graph isn’t always sufficient for fast kd-tree build

- Scene graph with high fan-out
  - E.g. 50K polygon character as one node
- Scene graph with sloppy bounding volumes
  - In the limit, reduces to a polygon soup
  - Sorting costs go up as pre-sort quality goes down
Two strategies to speed up non-hierarchical kd-tree build

• Treat meshes as a special case
  – No reason to discard useful information!
  – Speeds up build near the leaves

• Use approximation to surface area heuristic
  – Avoid sorting
  – Speeds up build at any level of the tree
Treat connected surfaces as a special case

- Use polygon grid (or mesh) as atomic unit of geometry for kd-tree
- Use a second acceleration structure at each kd-tree leaf node
  - Razor uses hierarchical bounding volume built from polygon grid
  - Works well due to structure of polygon grid
Fast kd-tree building heuristics
(or, how to get a good tree without sorting 50,000 polygons)

Warren Hunt
Surface area heuristic

- Surface area heuristic (SAH) produces high quality acceleration structures
  - For Kd-trees or BVHs
SAH summary

1) Generate candidate split locations
   - typically min and max for every triangle

2) Evaluate SAH cost function at every location

3) Pick location with lowest cost
The SAH cost function

\[ C(S) = P(left(S))C(left(S)) + P(right(S))C(right(S)) \]

- Probabilities (P) may be computed directly
  - From split location ‘S’
- Costs (C) require counting geometric primitives
  - Count number of primitives that overlap child
Standard algorithm for evaluating costs requires sort

- Sort geometry along split axis
- Allows cost function to be evaluated at every possible split location
- For large scenes, sort is slow.
Alternative: Approximate cost function via sampling

- Evaluate cost function at a few locations
- Approximate full function via interpolation
Evaluating the cost function for one split location is fast

- One pass over geometry
  - No sort
- Can be very fast:
  - Using SIMD operations, can evaluate several split locations in one pass.
  - Read-only operation
  - Parallelizable
The split-cost function

Cost

Location along split axis

Split cost
With costs left and right

Cost

Location along split axis

Split cost

Left-child cost

Right-child cost
First-cut approximation

Cost

Location along split axis

- Split cost
- Left-child cost
- Right-child cost
Additional adaptive samples

Cost

Location along split axis

Split cost
Left-child cost
Right-child cost
SAH approximation almost as good as full SAH

<table>
<thead>
<tr>
<th>Scene</th>
<th>Median Split</th>
<th>Approximate SAH</th>
<th>Full SAH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courtyard</td>
<td>7.90 s</td>
<td>4.96 s</td>
<td>4.86 s</td>
</tr>
<tr>
<td>Monster</td>
<td>5.94 s</td>
<td>5.00 s</td>
<td>4.91 s</td>
</tr>
</tbody>
</table>

- **Results details:**
  - Traversal times for 512x512 image, 32x super-sampling and 4 shadow rays per sample.
  - SAH approximation uses 8 initial + 8 adaptive samples.
Conclusion: SAH approximation is effective

- SAH approximation allows fast build time
- Traversal time is almost as low as full SAH
Dynamic scenes summary (1 of 2):
Use two data structures

- Scene graph
  - Application friendly
  - Provides approximate sort
  - Updated each frame

- **Kd-tree** (or similar, such as B-KD)
  - Ray-tracing friendly
  - Provides good sort
  - Rebuilt lazily each frame
Dynamic scenes summary (2 of 2): Rebuild kd-tree quickly

- Use hierarchy from scene graph
- Approximate the SAH
  - But use a good approximation

Hybrids of this approach with others are possible