Ray Tracing Performance

Zero to Millions in 45 Minutes

Gordon Stoll, Intel
Ray Tracing Performance

Zero to Millions in 45 Minutes?!

Gordon Stoll, Intel
Goals for this talk

• Goals
  – point you toward the current state-of-the-art ("BKM")
    • for non-researchers: off-the-shelf performance
    • for researchers: baseline for comparison
  – get you interested in poking at the problem

• Non-Goals
  – present lowest-level details of kernels
  – present "the one true way"
Acceleration Structures

• Previous (90s) BKM was to use a uniform grid
  – very high raw traversal speed
  – performance is not robust
• Current BKM is the kD-tree
  – comparable performance even on regular scenes
  – very good build technique handles irregular scenes
  – other grids, octrees, etc, might as well just use kD-tree
  – packet tracing was the icing on the cake
Acceleration Structures

• Couldn’t leave well enough alone
  – several new schemes to be described later
  – lots of stuff up in the air right now
  – But…

• The good tree building technique applies across the board.

• General optimization principles are similar.
kD-Trees
kD-Trees
kD-Trees
kD-Trees
Advantages of kD-Trees

• Adaptive
  – Can handle the “Teapot in a Stadium”

• Compact
  – Relatively little memory overhead

• Cheap Traversal
  – One FP subtract, one FP multiply
Take advantage of advantages

• Adaptive
  – You have to build a good tree

• Compact
  – At least use the compact node representation (8-byte)
  – You can’t be fetching whole cache lines every time

• Cheap traversal
  – No sloppy inner loops! (one subtract, one multiply!)
“Bang for the Buck” ( !/$ )

A basic kD-tree implementation will go pretty fast…

…but extra effort will pay off big.
Fast Ray Tracing w/ kD-Trees

- Adaptive
- Compact
- Cheap traversal
Building kD-trees

• Given:
  – axis-aligned bounding box (“cell”)
  – list of geometric primitives (triangles?) touching cell

• Core operation:
  – pick an axis-aligned plane to split the cell into two parts
  – sift geometry into two batches (some redundancy)
  – recurse
Building kD-trees

- **Given:**
  - axis-aligned bounding box ("cell")
  - list of geometric primitives (triangles?) touching cell

- **Core operation:**
  - pick an axis-aligned plane to split the cell into two parts
  - sift geometry into two batches (some redundancy)
  - recurse
  - termination criteria!
“Intuitive” kD-Tree Building

• Split Axis
  – Round-robin; largest extent

• Split Location
  – Middle of extent; median of geometry (balanced tree)

• Termination
  – Target # of primitives, limited tree depth
“Hack” kD-Tree Building

• Split Axis
  – Round-robin; largest extent

• Split Location
  – Middle of extent; median of geometry (balanced tree)

• Termination
  – Target # of primitives, limited tree depth

• All of these techniques stink.
“Hack” kD-Tree Building

• Split Axis
  – Round-robin; largest extent

• Split Location
  – Middle of extent; median of geometry (balanced tree)

• Termination
  – Target # of primitives, limited tree depth

• All of these techniques stink. Don’t use them.
“Hack” kD-Tree Building

• Split Axis
  – Round-robin; largest extent

• Split Location
  – Middle of extent; median of geometry (balanced tree)

• Termination
  – Target # of primitives, limited tree depth

• All of these techniques stink. Don’t use them.
  – I mean it.
Building good kD-trees

• What split do we really want?
  – Clever Idea: The one that makes ray tracing cheap
  – Write down an expression of cost and minimize it
  – Greedy Cost Optimization

• What is the cost of tracing a ray through a cell?

\[
\text{Cost(cell)} = C_{\text{trav}} + \text{Prob(hit } L) \times \text{Cost(L)} + \text{Prob(hit } R) \times \text{Cost(R)}
\]
Splitting with Cost in Mind
Split in the middle

- Makes the L & R probabilities equal
- Pays no attention to the L & R costs
Split at the Median

- Makes the L & R costs equal
- Pays no attention to the L & R probabilities
Cost-Optimized Split

- Automatically and rapidly isolates complexity
- Produces large chunks of empty space
Building good kD-trees

- Need the probabilities
  - Turns out to be proportional to surface area
- Need the child cell costs
  - Simple triangle count works great (very rough approx.)
  - Empty cell “boost”

\[
\text{Cost(cell)} = C_{\text{trav}} + \text{Prob(hit L)} \times \text{Cost(L)} + \text{Prob(hit R)} \times \text{Cost(R)}
\]

\[
= C_{\text{trav}} + \text{SA(L)} \times \text{TriCount(L)} + \text{SA(R)} \times \text{TriCount(R)}
\]
Termination Criteria

• When should we stop splitting?
  – Clever idea: When splitting isn’t helping any more.
  – Use the cost estimates in your termination criteria

• Threshold of cost improvement
  – Stretch over multiple levels (?)

• Threshold of cell size
  – Absolute probability so small there’s no point
Building good kD-trees

• Basic build algorithm
  – Pick an axis, or optimize across all three
  – Build a set of “candidates” (split locations)
    • BBox edges or exact triangle intersections
  – Sort them or bin them
  – Walk through candidates to find minimum cost split

• Characteristics of highly optimized tree
  – “stringy”, very deep, very small leaves, big empty cells
BKM for BVH building as well
BVH Version of Build
Just Do It

• Benefits of a good tree are not small
  – not 10%, 20%, 30%...
  – several times faster than a mediocre tree

• Warning #1
  – get a good set of scenes for testing
  – scanned models aren’t representative, just easy to find

• Warning #2
  – beware the NOP
Building Trees Quickly

• Very important to build good trees first
  – otherwise you have no foundation for comparison (!!!)
• Don’t give up SAH cost optimization!
  – approximate, sure, but don’t just drop it
• Luckily, lots of flexibility…
  – axis picking (“hack” pick vs. full optimization)
  – candidate picking (bboxes, exact; binning, sorting)
  – termination criteria (“knob” controlling tradeoff)
Building Trees Quickly

• Huge amount of work happening right now
• I’m going to defer to later speakers
Fast Ray Tracing w/ kD-Trees

• adaptive
  – build a cost-optimized tree w/ the surface area heuristic
• compact
• cheap traversal
What’s in a node?

• A kD-tree internal node needs:
  – Am I a leaf?
  – Split axis
  – Split location
  – Pointers to children
Compact (8-byte) nodes

- kD-Tree node can be packed into 8 bytes
  - Leaf flag + Split axis
    - 2 bits
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - One 32-bit pointer
Compact (8-byte) nodes

- kD-Tree node can be packed into 8 bytes
  - Leaf flag + Split axis
    - 2 bits
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - One 32-bit pointer

- So close! Sweep those 2 bits under the rug…
No Bounding Box!

• kD-Tree node corresponds to an AABB
• Doesn’t mean it has to *contain* one
  – 24 bytes
  – 4X explosion (!)
Memory Layout

• Cache lines are much bigger than 8 bytes!
  – advantage of compactness lost with poor layout

• Pretty easy to do something reasonable
  – Building depth first, watching memory allocator
Other Data

- Memory should be separated by rate of access
  - Frames
  - << Pixels
  - << Samples [ Ray Trees ]
  - << Rays [ Shading (not quite) ]
  - << Triangle intersections
  - << Tree traversal steps

- Example: pre-processed triangle, shading info…
Fast Ray Tracing w/ kD-Trees

- adaptive
  - build a cost-optimized tree w/ the surface area heuristic
- compact
  - use an 8-byte node
  - lay out your memory in a cache-friendly way
- cheap traversal
kD-Tree Traversal Step
kD-Tree Traversal Step

\[ t_{\text{split}} \quad t_{\text{min}} \quad \text{split} \]

\[ t_{\text{max}} \]
kD-Tree Traversal Step

- t_min
- t_max
- t_split
- split
kD-Tree Traversal Step

Given: ray P & iV (1/V), t_min, t_max, split_location, split_axis

\[ t_{at\_split} = (\text{split\_location} - \text{ray->P}[\text{split\_axis}]) \times \text{ray\_iV}[\text{split\_axis}] \]

if \( t_{at\_split} > t_{\text{min}} \)
    need to test against near child

If \( t_{at\_split} < t_{\text{max}} \)
    need to test against far child
Optimize Your Inner Loop

• kD-Tree traversal is the most critical kernel
  – It happens about a zillion times
  – It’s tiny
  – Sloppy coding will show up

• Optimize, Optimize, Optimize
  – Remove recursion and minimize stack operations
  – Other standard tuning & tweaking
while ( not a leaf )

    t_at_split = (split_location - ray->P[split_axis]) * ray_iV[split_axis]

if t_split <= t_min
    continue with far child       // hit either far child or none

if t_split >= t_max
    continue with near child     // hit near child only

// hit both children
push (far child, t_split, t_max) onto stack
continue with (near child, t_min, t_split)
Can it go faster?

- How do you make fast code go faster?
- Parallelize it!
Ray Tracing and Parallelism

• Classic Answer: Ray-Tree parallelism
  – independent tasks
  – # of tasks = millions (at least)
  – size of tasks = thousands of instructions (at least)

• So this is wonderful, right?
Parallelism in CPUs

- Instruction-Level Parallelism (ILP)
  - pipelining, superscalar, OOO, SIMD
  - fine granularity (~10s of instructions “window” tops)
  - easily confounded by unpredictable control
  - easily confounded by unpredictable latencies

- So...what does ray tracing look like to a CPU?
No joy in ILP-ville

• At <1000 instruction granularity, ray tracing is anything but “embarrassingly parallel”

• kD-Tree traversal (CPU view):
  1) fetch a tiny fraction of a cache line from who knows where
  2) do two piddling floating-point operations
  3) do a completely unpredictable branch, or two, or three
  4) repeat until frustrated

PS: Each operation is dependent on the one before it.

PPS: No SIMD for you! Ha!
Split Personality

- **Coarse-Grained parallelism (TLP)** is perfect
  - millions of independent tasks
  - thousands of instructions per task

- **Fine-Grained parallelism (ILP)** is awful
  - look at a scale <1000 of instructions
    - sequential dependencies
    - unpredictable control paths
    - unpredictable latencies
    - no SIMD
Options

• Option #1: Forget about ILP, go with TLP
  – improve low-ILP efficiency and use multiple CPU cores
• Option #2: Let TLP stand in for ILP
  – run multiple independent threads (ray trees) on one core
• Option #3: Improve the ILP situation directly
  – how?
• Option #4: …
...All of the above!

- multi-core CPUs are here in spades
  - better performance, better low-ILP performance
  - on the right performance curve
- multi-threaded CPUs are already here
  - improve well-written ray tracer by ~20-30%
- packet tracing
  - trace multiple rays together in a packet
  - bulk up the inner loop with ILP-friendly operations
Packet Tracing

- Very, very old idea from vector/SIMD machines
  - Vector masks

- Old way
  - if the ray wants to go left, go left
  - if the ray wants to go right, go right

- New way
  - if any ray wants to go left, go left with mask
  - if any ray wants to go right, go right with mask
Key Observations

• Doesn’t add “bad” stuff
  – Traverses the same nodes
  – Adds no global fetches
  – Adds no unpredictable branches

• What it does add
  – SIMD-friendly floating-point operations
  – Some messing around with masks

• Result: Very robust in relation to single rays
How many rays in a packet?

- Packet tracing gives us a “knob” with which to adjust computational intensity.
- Do natural SIMD width first
- Real answer is potentially much more complex
  - diminishing returns due to per-ray costs
  - lack of coherence to support bigger packets
  - register pressure, L1 pressure
- Makes hardware much more likely/possible
Beyond Packets

• Diminishing returns limit packet size
  – cost of any operation is still linear in packet size
  – run out of overhead to amortize away

• Frustums and/or Interval Representations
  – constant time operations on a whole packet
  – MLRT, Alex Reshetov, SIGGRAPH ’05
    • “inverse frustum culling”, interval traversal
  – Carsten Benthin’s thesis
Fast Ray Tracing w/ kD-Trees

- **Adaptive**
  - build a cost-optimized tree (w/ surface area heuristic)

- **Compact**
  - use an 8-byte node
  - lay out your memory in a cache-friendly way

- **Cheap traversal**
  - optimize your inner loop
  - trace packets
Getting started...

• Read PBRT (yeah, I know, it’s 1300 pages)
  – great book, pretty decent kD-tree builder
• Read Wald’s thesis and Benthin’s thesis
  – lots of coding details for this stuff
• Track down the interesting references
• Learn SIMD programming (e.g. SSE intrinsics)
• Use a profiler.
Getting started...

- Read PBRT (yeah, I know, it’s 1300 pages)
  - great book, pretty decent kD-tree builder
- Read Wald’s thesis and Benthin’s thesis
  - lots of coding details for this stuff
- Track down the interesting references
- Learn SIMD programming (e.g. SSE intrinsics)
- Use a profiler. I mean it.
If you remember nothing else

- “Rays per Second” is measured in millions.