



SIGGRAPH2005

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”



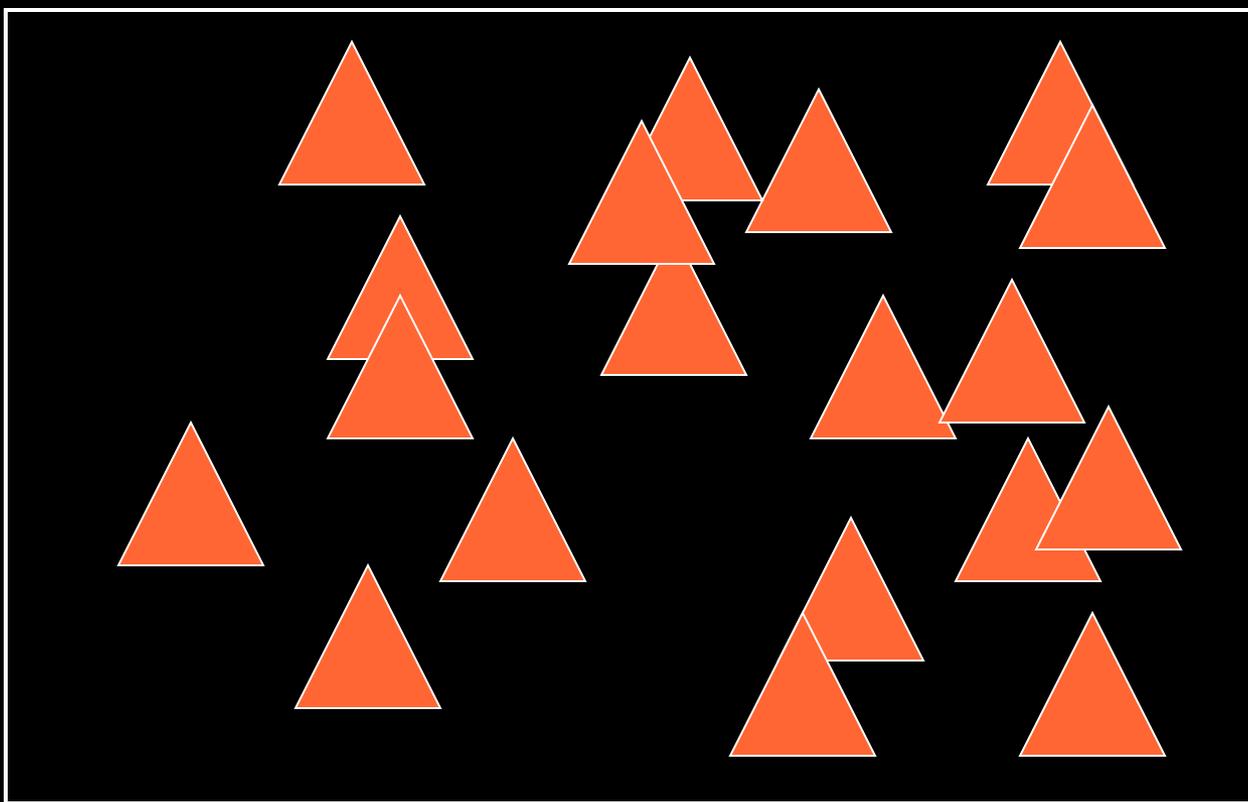
Acceleration Structures

- BKM is to use a kD-tree (AA BSP)
- Previous BKM was to use a uniform grid
 - Only scheme with comparable speed
 - Performance is not robust
 - No packet tracing algorithm
- Other grids, octrees, etc...just use a kD-tree.
- *Don't use bounding volume hierarchies.*



SIGGRAPH2005

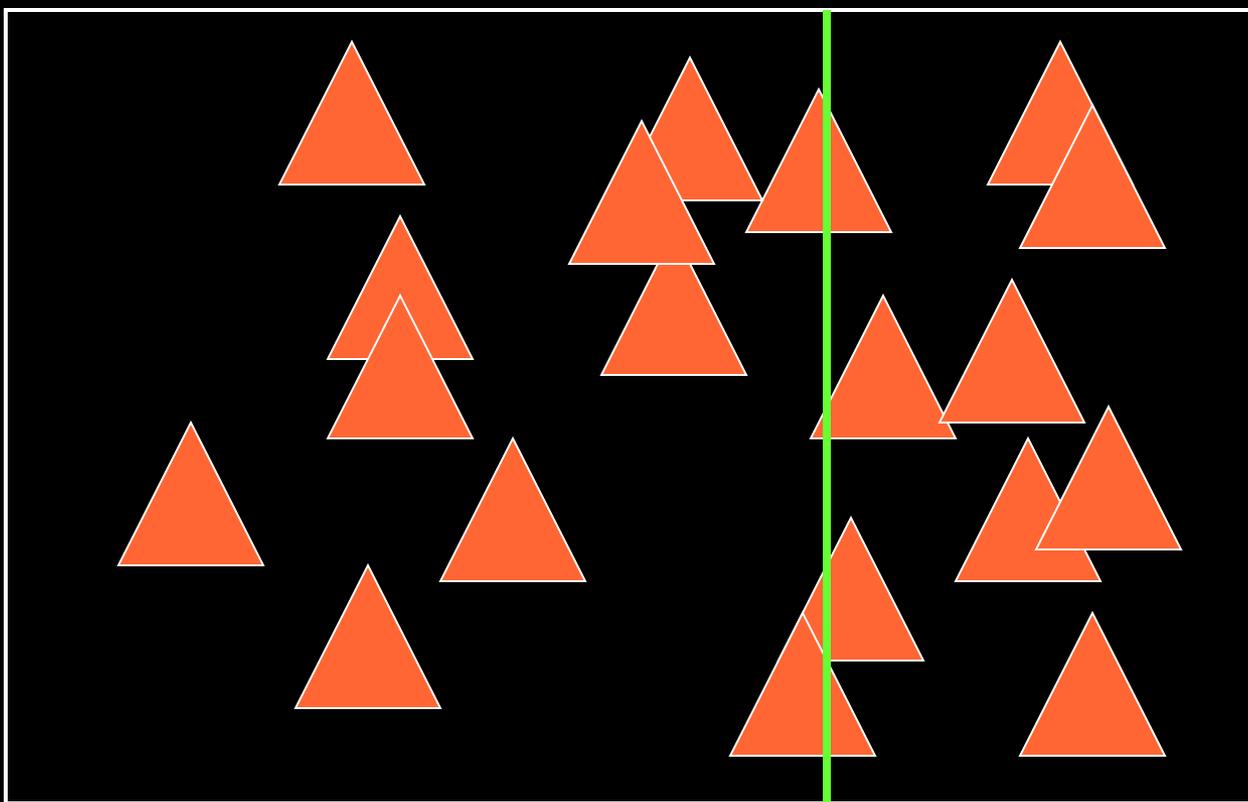
kD-Trees





SIGGRAPH2005

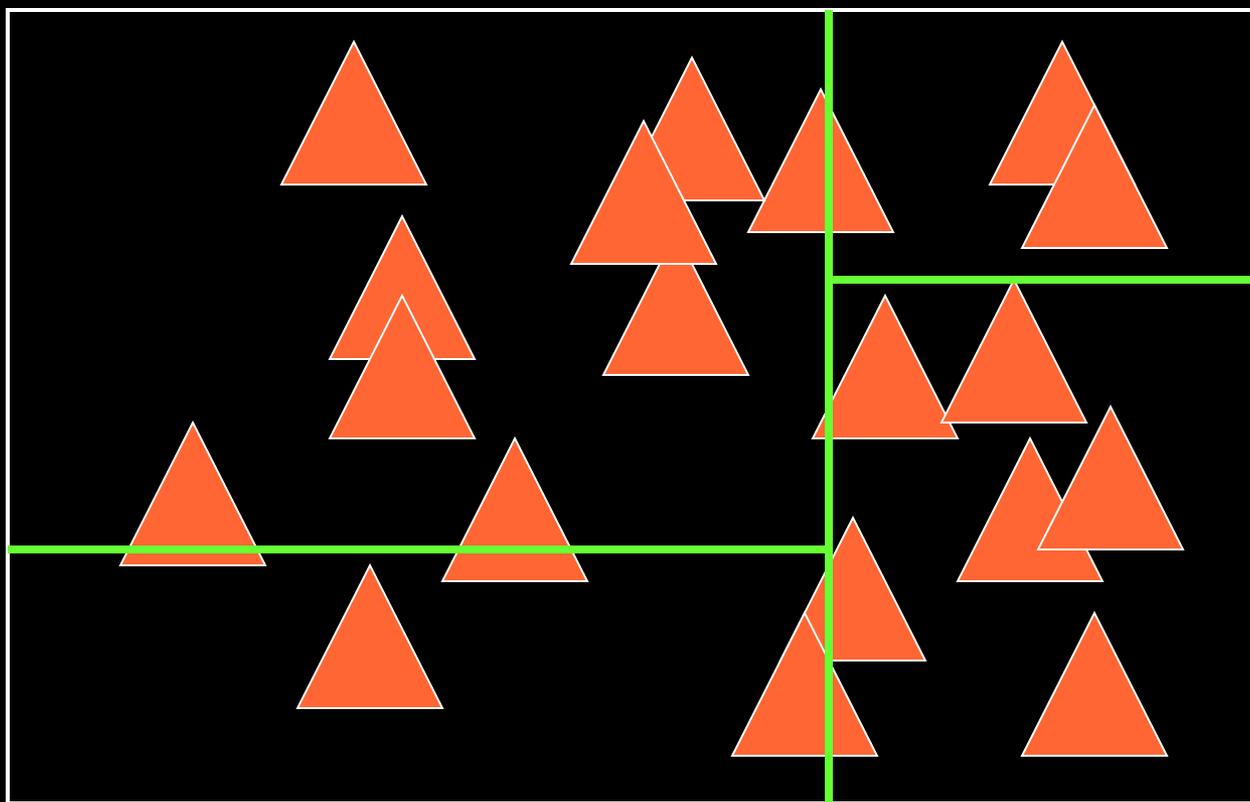
kD-Trees



kD-Trees



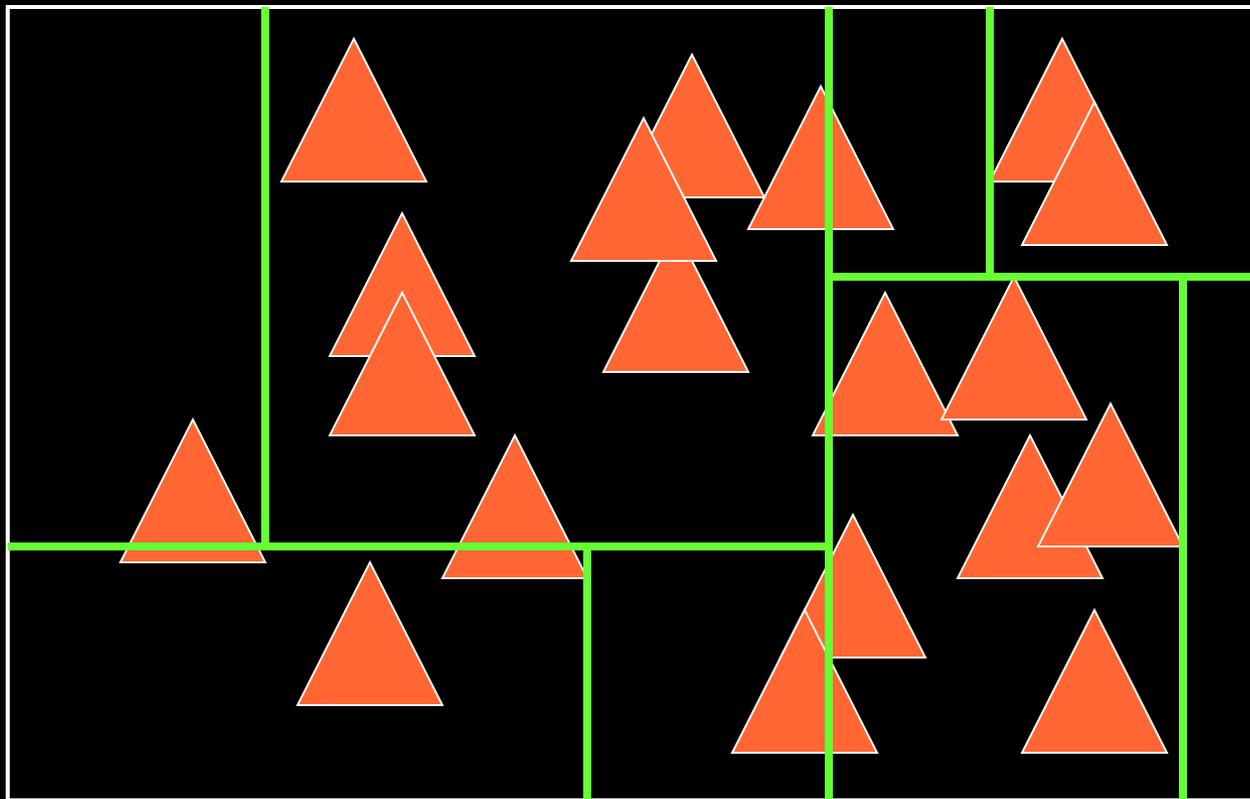
SIGGRAPH2005





SIGGRAPH2005

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!)



SIGGRAPH2005

“Bang for the Buck” (!/\$)

A basic kD-tree implementation will go pretty fast...
...but extra effort will pay off *big*.



SIGGRAPH2005

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

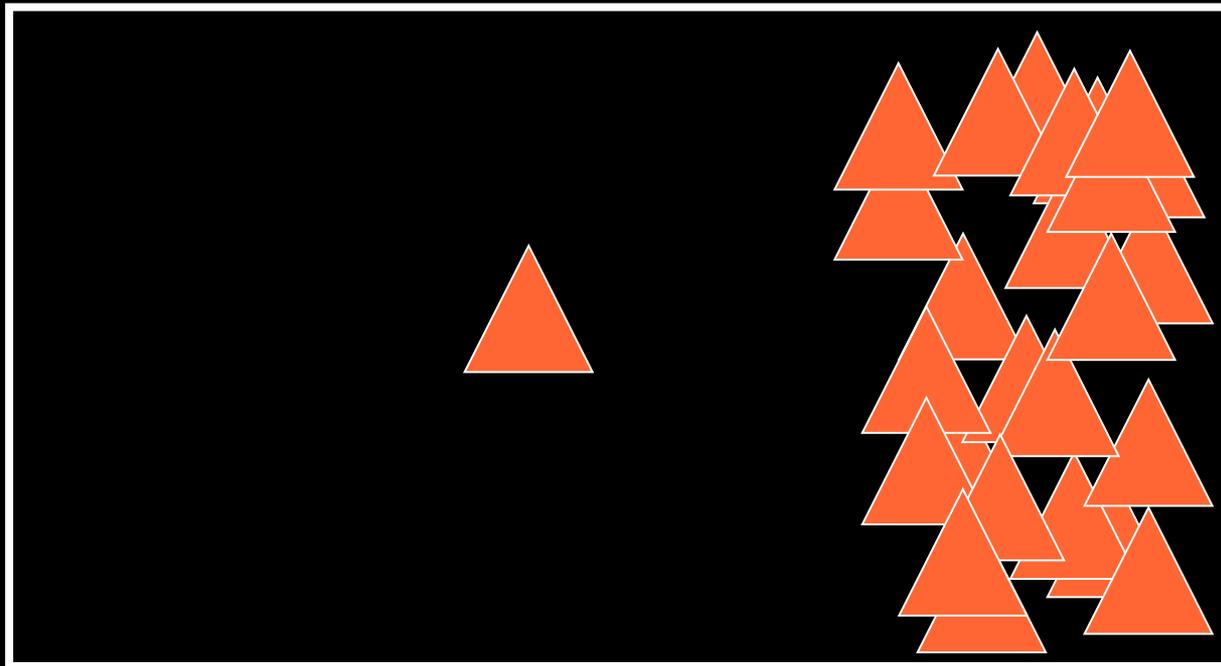
- What split do we really want?
 - Clever Idea: The one that makes ray tracing cheap
 - Write down an expression of cost and minimize it
 - *Cost Optimization*
- What is the cost of tracing a ray through a cell?

$$\text{Cost}(\text{cell}) = C_{\text{trav}} + \text{Prob}(\text{hit L}) * \text{Cost}(\text{L}) + \text{Prob}(\text{hit R}) * \text{Cost}(\text{R})$$



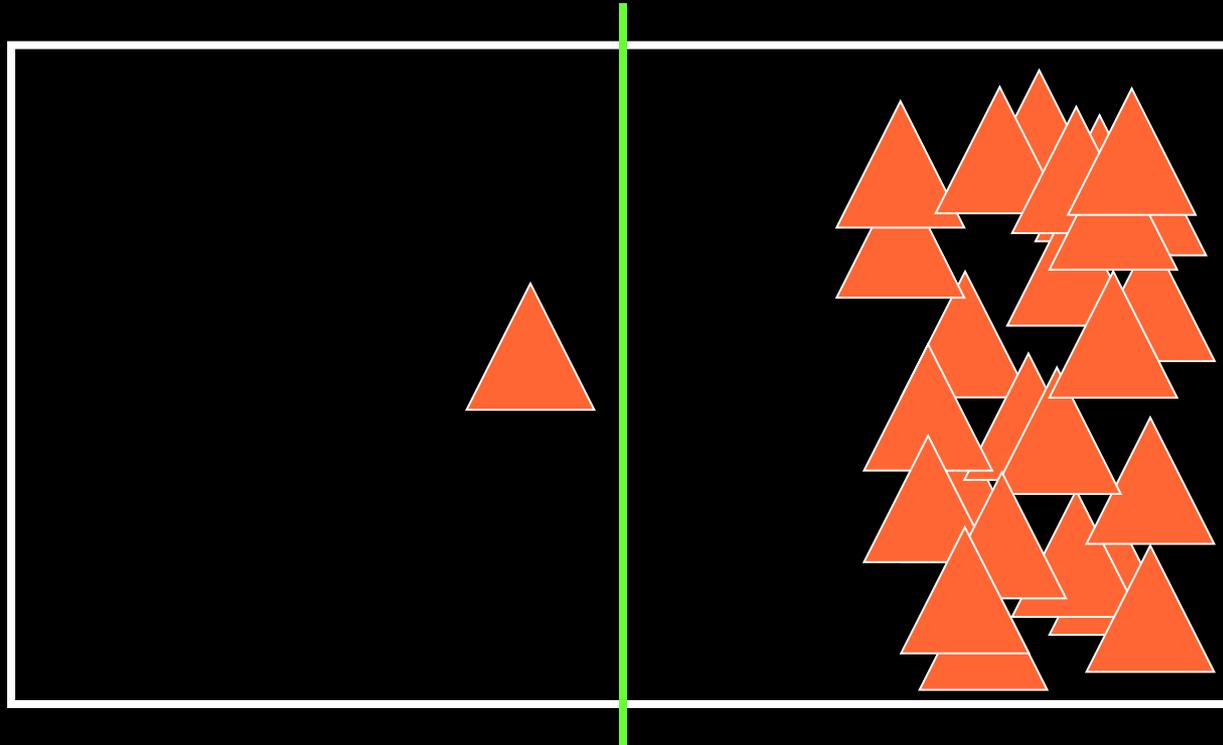
SIGGRAPH2005

Splitting with Cost in Mind





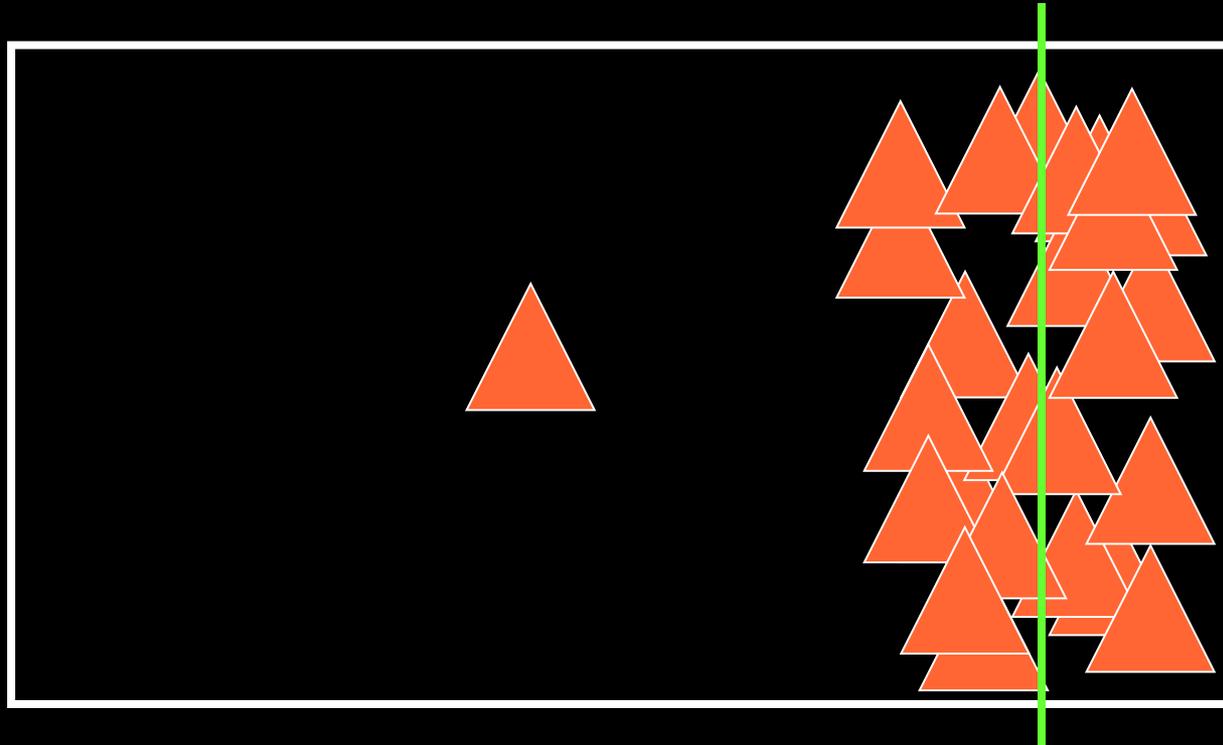
Split in the middle



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



Split at the Median

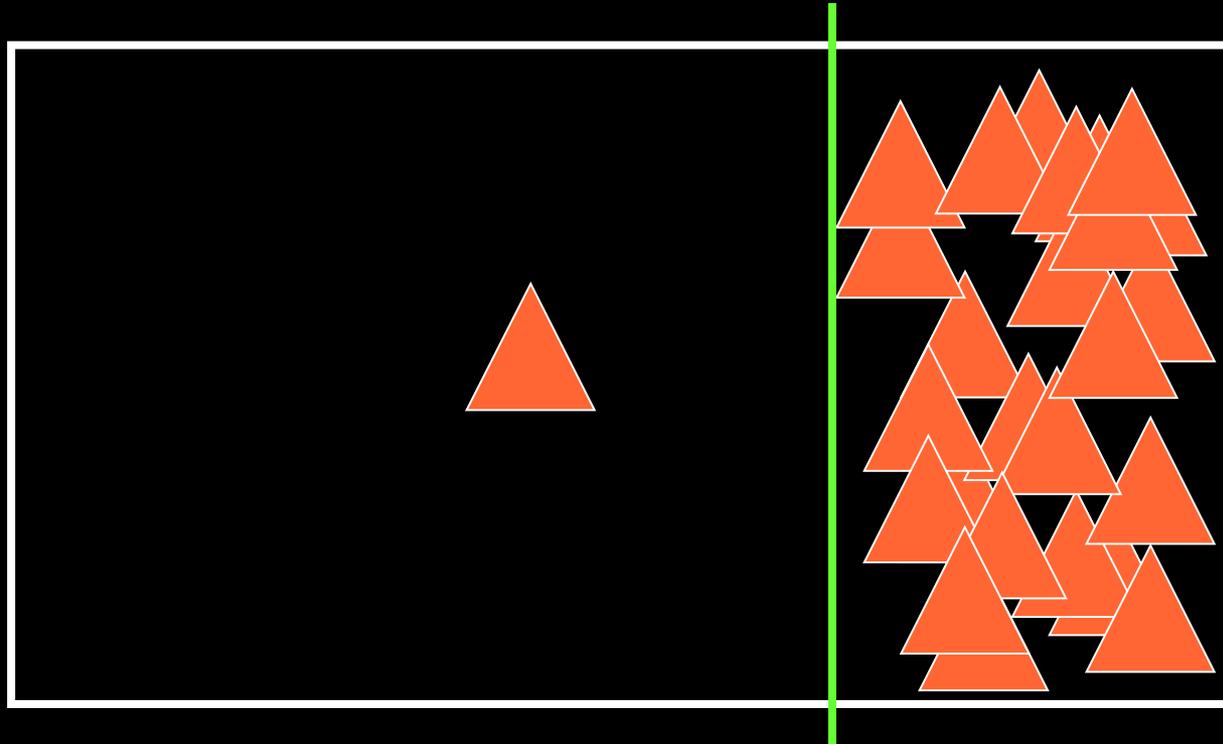


- Makes the L & R costs equal
- Pays no attention to the L & R probabilities



SIGGRAPH2005

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.)

$$\begin{aligned}\text{Cost}(\text{cell}) &= C_{\text{trav}} + \text{Prob}(\text{hit L}) * \text{Cost}(\text{L}) + \text{Prob}(\text{hit R}) * \text{Cost}(\text{R}) \\ &= C_{\text{trav}} + \text{SA}(\text{L}) * \text{TriCount}(\text{L}) + \text{SA}(\text{R}) * \text{TriCount}(\text{R})\end{aligned}$$



Termination Criteria

- When should we stop splitting?
 - Another 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 or bins to find minimum cost split
- Characteristics you’re looking for
 - “stringy”, depth 50-100, ~2 triangle leaves, big empty cells



SIGGRAPH2005

Just Do It

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



Building kD-trees quickly

- Very important to build good trees first
 - otherwise you have no basis for comparison
- Don't give up cost optimization!
 - Use the math, Luke...
- Luckily, *lots* of flexibility...
 - axis picking (“hack” pick vs. full optimization)
 - candidate picking (bboxes, exact; binning, sorting)
 - termination criteria (“knob” controlling tradeoff)



Building kD-trees quickly

- Remember, profile first! Where's the time going?
 - split personality
 - memory traffic all at the top (NO cache misses at bottom)
 - sifting through bajillion triangles to pick one split (!)
 - hierarchical building?
 - computation mostly at the bottom
 - lots of leaves, need more exact candidate info
 - lazy building?
 - change criteria during the build?



SIGGRAPH2005

Fast Ray Tracing w/ kD-Trees

- adaptive
 - build a cost-optimized kD-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...



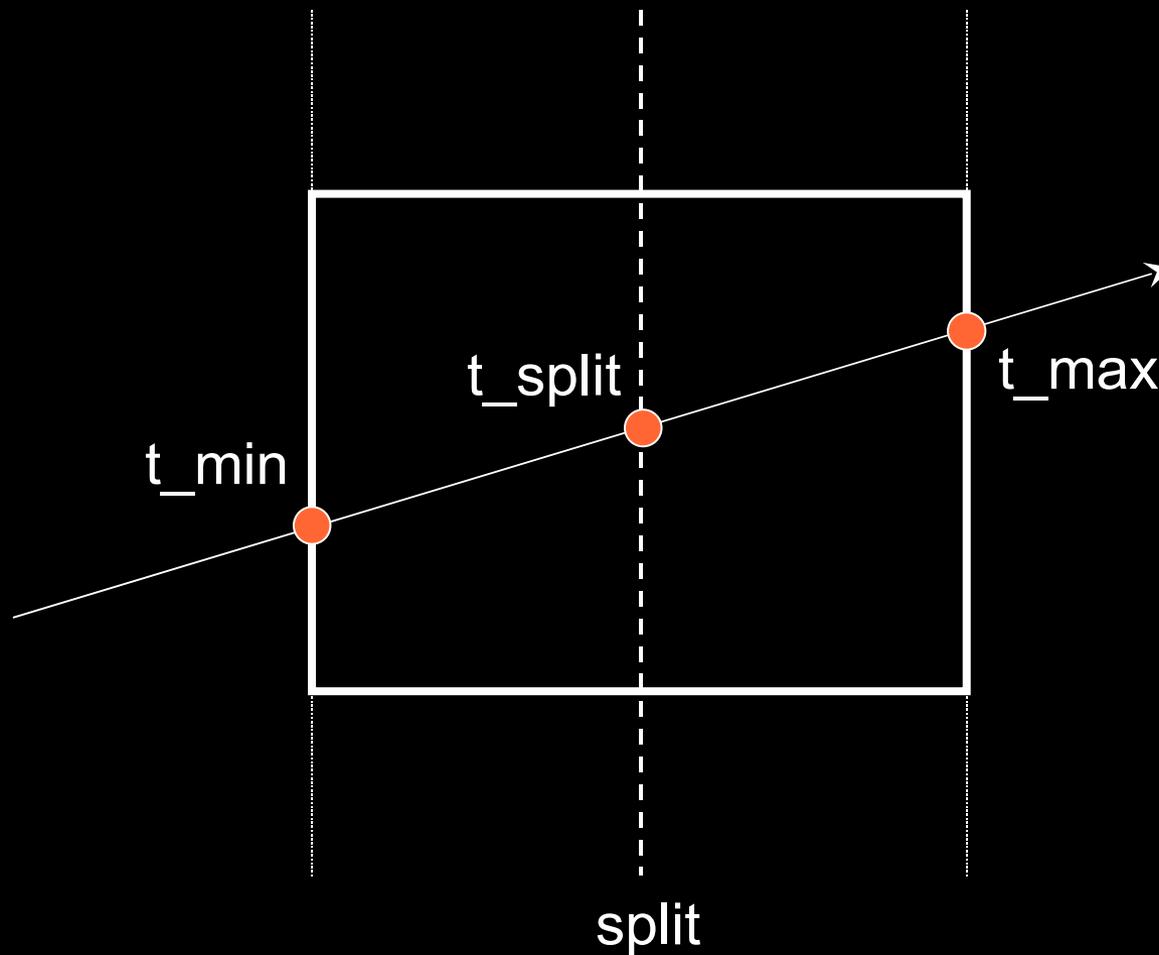
SIGGRAPH2005

Fast Ray Tracing w/ kD-Trees

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

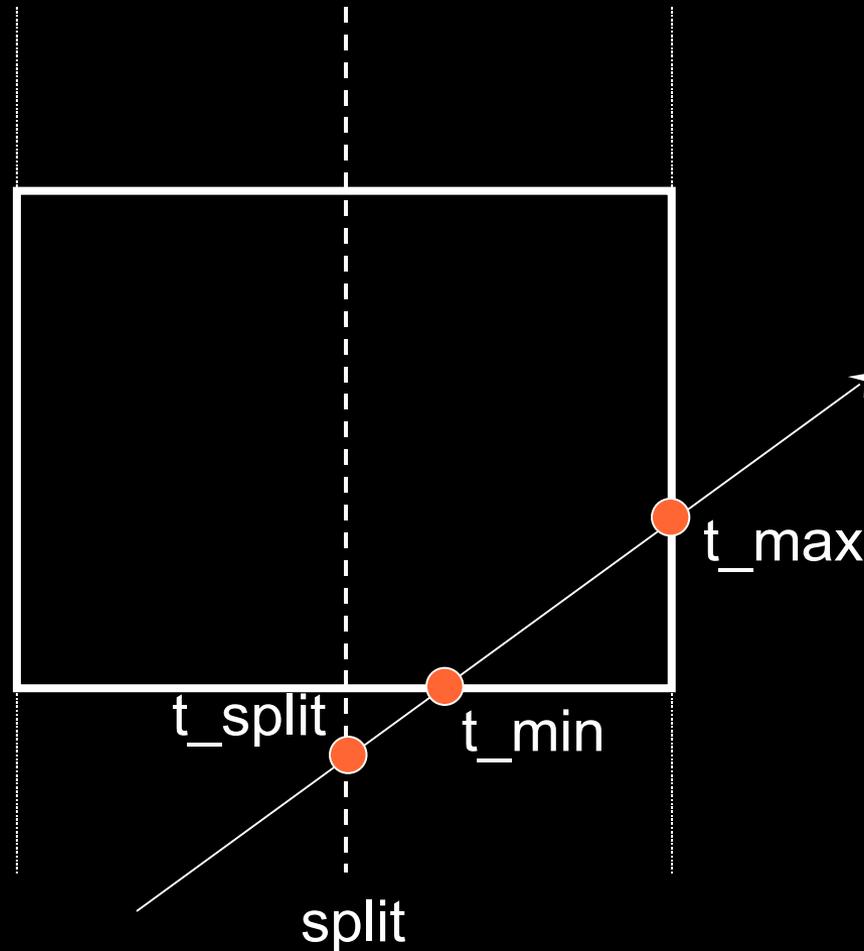


kD-Tree Traversal Step



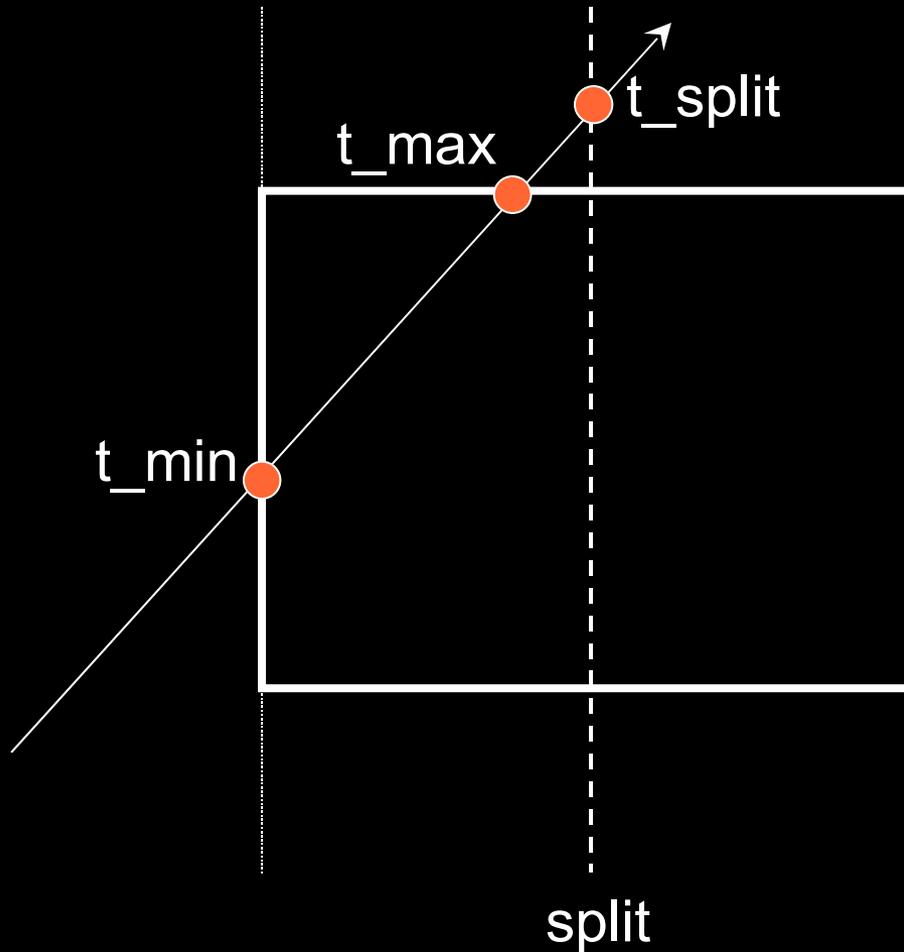


kD-Tree Traversal Step





kD-Tree Traversal Step





kD-Tree Traversal Step

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

$$t_{\text{at_split}} = (\text{split_location} - \text{ray} \rightarrow P[\text{split_axis}]) * \text{ray_iV}[\text{split_axis}]$$

if $t_{\text{at_split}} > t_{\text{min}}$

 need to test against near child

If $t_{\text{at_split}} < t_{\text{max}}$

 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



kD-Tree Traversal

```
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)
```



SIGGRAPH2005

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 (~100 instruction “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 already here (more coming)
 - 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 big packets
 - register pressure, L1 pressure
- Makes hardware much more likely/possible



SIGGRAPH2005

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 Ingo Wald'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 Ingo Wald'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.



SIGGRAPH2005

If you remember nothing else

- “Rays per Second” is measured in *millions*.