Acceleration Structure for Ray Tracing

K-D Tree Traversal
Locate the primitives that intersect with a ray, FAST! (Avoid brute-force approach)
Background

- Similarity search or nearest neighbor search
- Applications:
  - Statistics
  - Content-based retrieval
  - Data-mining and Pattern recognition
  - Geographic information system (GIS)
    - where is the closest post office
  - Graphics
Naïve Approach

- Given a query point $X$.
- Scan through each point $Y$
- Takes $O(N)$ time for each query!
We can speed up the search for the nearest neighbor:
- Examine nearby points first.
- Ignore any points that are further than the nearest point found so far.
k-d tree is a multidimensional binary search tree.

Recursively partitions points into axis aligned boxes. One axis at a time.
We start with a list of n-dimensional points.
We can split the points into 2 groups by choosing a dimension \( X \) and value \( V \) and separating the points into \( X > V \) and \( X \leq V \).
We can then consider each group separately and possibly split again (along same/different dimension).
We can then consider each group separately and possibly split again (along same/different dimension).
We can keep splitting the points in each set to create a tree structure. Each node with no children (leaf node) contains a list of points.
We will keep around one additional piece of information at each node. The (tight) bounds of the points at or below this node.
KD-Tree Construction

- Use heuristics to make splitting decisions:
- Which dimension do we split along? **Widest**
- Which value do we split at? **Median of value of that split dimension for the points.**
- When do we stop? **When there are fewer then** $m$ **points left OR the box has hit some minimum width.**
We traverse the tree looking for the nearest neighbor of the query point.
Examine nearby points first: Explore the branch of the tree that is closest to the query point first.
Examine nearby points first: Explore the branch of the tree that is closest to the query point first.
When we reach a leaf node: compute the distance to each point in the node.
When we reach a leaf node: compute the distance to each point in the node.
Then we can backtrack and try the other branch at each node visited.
Each time a new closest node is found, we can update the distance bounds.
Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.
Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.
Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.
KD-Tree in Ray Tracing

Query: given a ray $r(t)$, find the primitives that intersected with the ray
KD-Tree in Ray Tracing
KD-Tree in Ray Tracing

Copyright © 2010 by Yong Cao
KD-Tree Traversal
**KD-Tree Traversal**

![Diagram of KD-Tree Traversal]

Copyright © 2010 by Yong Cao
Adaptivity

- Can handle the “Teapot in a Stadium”
  - Octree is not.

Compactness

- Relatively little memory overhead
  - 8 bytes only (function as BSP-tree, not more compact)

Cheap Traversal

- One FP subtract, one FP multiply
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
- 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.**
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?

Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)
Splitting with Cost in Mind
• Makes the L & R probabilities equal
• Pays no attention to the L & R costs
• Makes the L & R costs equal
• Pays no attention to the L & R probabilities

Copyright © 2010 by Yong Cao
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.)

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

- When should we stop splitting?
  - 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
Fast Ray Tracing w/ kD-Trees

- **adaptive**
  - build a cost-optimized kD-tree w/ the surface area heuristic
- **compact**
- cheap traversal
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 (!)
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

Copyright © 2010 by Yong Cao
**kD-Tree Traversal Step**

- **t_split**
- **t_min**
- **t_max**
- **split**

Copyright © 2010 by Yong Cao
kD-Tree Traversal Step

```
t_min
t_max
```

Copyright © 2010 by Yong Cao
Given: ray \( P & iv (1/V), t_{\text{min}}, t_{\text{max}}, \)  
\( \text{split\_location, split\_axis} \)

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

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
while ( not a leaf )
    \( t_{\text{split}} = ( \text{split\_location} - \text{ray}\rightarrow\text{P}[\text{split\_axis}] ) \times \text{ray\_iV}[\text{split\_axis}] \)
    if \( t_{\text{split}} \leq t_{\text{min}} \)
        continue with far child // hit either far child or none
    if \( t_{\text{split}} \geq t_{\text{max}} \)
        continue with near child // hit near child only
// hit both children
push (far child, \( t_{\text{split}}, t_{\text{max}} \)) onto stack
continue with (near child, \( t_{\text{min}}, t_{\text{split}} \))
Reference

- Slides from Siggraph 2005 course on “Ray Tracing Performance” by Gordon Stoll
- Slides from “KD-Tree Acceleration Structures for a GPU Raytracer” by Tim Foley and Jeremy Sugerman