Overview

- Acceleration structures
  - Spatial hierarchies
  - Object hierarchies

- Interactive Ray Tracing techniques
  - Optimized hierarchy construction
  - Optimized hierarchy traversal - Ray packets
Last lecture

- Ray Tracing
  - For each pixel: generate ray
    - For each primitive:
      - Does ray hit primitive?
        - Yes: Test depth / write color

- That means linear time complexity in number of primitives!
Acceleration structures

- **Goal:**
  fast search operation for ray that returns all intersecting primitives
  - Then test only against those
  - Operation should take sub-linear time

- In practice: conservative superset
Acceleration structures

- **Broad classification:**
  - Spatial hierarchies
    - Grids
    - Octrees
  - Kd-trees, BSP trees
  - Object hierarchies
    - Bounding volume hierarchies
Spatial Hierarchies: Grids

- Building a grid structure
  - Start with bounding box
  - Resolution: often $\sim 3\sqrt{n}$
  - Overlap or intersection test
- Traversal
  - 3D-DDA
  - Stop if intersection found in current voxel
Grids

- Grid traversal
  - Requires enumeration of voxel along ray $\rightarrow$ 3D-DDA
  - Simple and hardware-friendly

- Objects in multiple voxels
  - Store only references
  - Need to avoid multiple intersection computations
    - Store (ray, object)-tuple in small cache (e.g. with hashing)
    - Do not intersect if found in cache
Grids

- Grid resolution
  - Strongly scene dependent
  - Cannot adapt to local density of objects
    - Problem: „Teapot in a stadium“
  - Possible solution: hierarchical grids
Hierarchical Grids

- **Simple building algorithm**
  - Recursively create grids in high-density voxels
  - Problem: What is the right resolution for each level?

- **Advanced algorithm**
  - Separate grids for object clusters
  - Problem: What are good clusters?
Spatial Hierarchies: Octree

- Hierarchical space partitioning
  - Adaptively subdivide voxels into 8 equal sub-voxels recursively
  - Result in subdivision

- Problems
  - Rather complex traversal algorithms
  - Slow to refine complex regions
Spatial Hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane
kD-Trees
kD-Trees
kD-Trees
kD-Trees
Spatial Hierarchies: kd-trees

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
BSP-Trees

- Recursive space partitioning with half-spaces
- Binary Space Partition (BSP):
  - Splitting with half-spaces in arbitrary position
- Kd-Tree
  - Splitting with axis-aligned half-spaces
Object Hierarchies: BVHs

- Different approach: subdivide objects, not space
  - Hierarchical clustering of objects
  - Each cluster represented by bounding volume

- Binary tree
  - Each parent node fully contains children
Bounding volumes

- Practically anything can be bounding volume
  - Just need ray intersection method

- Typical choices:
  - Spheres
  - Axis-aligned bounding boxes (AABBS)
  - Oriented bounding boxes (OBBs)
  - k-DOPs

- Trade-off between intersection speed and how closely the BV encloses the geometry
BVH traversal

- Recursive algorithm:
  - Start with root node
  - For current node:
    - Does ray intersect node’s BV? If no, return
    - Is inner node?
      - Yes, recurse on children
    - Is leaf node?
      - Intersect with object(s) in node, store intersection results

- Note: can’t return after first intersection!
Overview

- Acceleration structures
  - Spatial hierarchies
  - Object hierarchies
- Interactive Ray Tracing techniques
  - Optimized hierarchy construction
  - Optimized hierarchy traversal - Ray packets
Optimized hierarchy construction

- The way the hierarchy is constructed has high impact on traversal performance
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?

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

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

= C_{trav} + \text{SA(L)} \times \text{TriCount(L)} + \text{SA(R)} \times \text{TriCount(R)}
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
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
Just Do It

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

- Acceleration structures
  - Spatial hierarchies
  - Object hierarchies

- Interactive Ray Tracing techniques
  - Optimized hierarchy construction
  - Optimized hierarchy traversal - Ray packets
Acceleration of kD-Tree Traversal

- **Compact node structure**
  - Relatively little memory overhead

- **Fast traversal step**
  - One FP subtract, one FP multiply
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 (!)
Acceleration of kD-Tree Traversal

- Compact node structure
  - Relatively little memory overhead
- Fast traversal step
  - One FP subtract, one FP multiply
kD-Tree Traversal Step

- $t_{\text{min}}$
- $t_{\text{split}}$
- $t_{\text{max}}$

Diagram showing the traversal step with $t_{\text{split}}$ as the central point and $t_{\text{min}}$ and $t_{\text{max}}$ as the endpoints.
kD-Tree Traversal Step
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 = (split_location - ray->P[split_axis]) * ray_iV[split_axis]

if t_at_split > t_min
    need to test against near child
If t_at_split < t_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
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
Ray Packets

- Often have large set of rays close together

- **Idea:**
  trace rays in coherent groups (ray packets)
Ray coherence

- How does this change tracing?
  - Traversal and object intersection work on group of rays at a time
  - Also generate secondary (shadow, ...) rays in packets

- General idea:
  - If one ray intersects, all are intersected
BVH traversal

- Recursive algorithm:
  - Start with root node
  - For current node:
    - Does ray intersect node’s BV? If no, return
    - Is inner node?
      - Yes, recurse on children
    - Is leaf node?
      - Intersect ray with all object(s) in node, store intersection results
BVH packet traversal

- Recursive algorithm:
  - Start with root node
  - For current node:
    - Does any ray intersect node’s BV? If no, return
    - Is inner node?
      - Yes, recurse on children
    - Is leaf node?
      - Intersect all rays with all object(s) in node, store intersection results
Ray packet advantages

- Why does this make things faster?
  - Less memory bandwidth: nodes/objects only loaded once for rays in packet
  - Allows data parallel processing!
    - Current CPUs: e.g. Intel SSE
    - All GPUs

- Disadvantage:
  - Rays can be intersected with objects/nodes they would never hit!
Data parallelism

- Essentially vector operations (SIMD)
  
- Operations work on all elements in parallel

\[ v_1 \oplus w_1 \]
\[ v_2 \oplus w_2 \]
\[ v_3 \oplus w_3 \]
\[ v_4 \oplus w_4 \]
\[ v_5 \oplus w_5 \]
\[ v_6 \oplus w_6 \]
\[ v_7 \oplus w_7 \]
\[ v_8 \oplus w_8 \]
Data parallelism

- Vector operations usually as fast as a single scalar operation
  - Intel SSE: 4-wide vectors
  - GPUs: 8-32 wide
  - Cell: 4-wide
  - Vector processors: up to 64,000!
SIMD ray processing

- Can use SIMD units to parallelize for rays
  - Each vector has one component from one ray
  - Thus, can process small group at same speed as one ray!
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
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)
- Can borrow many tricks we know from kd-Trees
  - SAH build, SIMD, frustum triangle test, interval arithmetic, ...

On the other hand...

- 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 borrow many tricks we know from kd-Trees
  - SAH build, SIMD, frustum triangle test, interval arithmetic, ...
- Most important: can be updated for dynamic scenes
Hierarchy updates

- What does updating mean?
  - Underlying geometry changes
  - Update will ensure **correctness of hierarchy** without rebuilding it
- Should be faster than rebuild
Ray Tracing Dynamic Scenes

- **Alternative 1:** Handle dynamic objects separately
  - Keep dynamic objects out of ADS, intersect separately
  - Only works for very few ‘dynamic’ objects

- **Alternative 2:** Rebuild ADS every frame
  - (Super-)linear in #objects
  - Only applicable to tiny scenes
  - Note: This might change for future HW…
Ray Tracing Dynamic Scenes

- Alternative 3: Update ADS for dynamic objects
  - Problem 1: How to avoid degradation of ADS?
  - Problem 2: How to *efficiently* update ADS?
  - Kd-trees: Quite problematic
    - Original cost estimates (SAH) wrong/useless after geom. changes
    - Updating often not (easily) possible
      - E.g., can’t move root split: would need to rebuild all sibling nodes
  - But: works reasonably well for BVH
Choosing a structure

- There is no ‘best’ acceleration structure
  - All have pros and cons
- Grid:
  + fast construction
  - bad for high local detail (teapot/stadium)
Choosing a structure

- There is no ‘best’ acceleration structure
  - All have pros and cons

- kd-tree:
  - fast traversal
  - expensive build, only static scenes
Choosing a structure

- There is no ‘best’ acceleration structure
  - All have pros and cons
- BVH:
  + can be updated for dynamic scenes
  - traversal more expensive than kd-tree
Summary

- Use acceleration structure
  - Spatial hierarchies: grid, kd-trees
  - Object hierarchies: BVH
- Fast construction and traversal algorithms
- Other considerations: memory layout, vector processing, ...