Computer Graphics

- Spatial Index Structures -

Philipp Slusallek
Overview

• **Last lecture**
  – Overview of ray tracing
  – Ray-primitive intersections

• **Today**
  – Acceleration structures
    • Bounding Volume Hierarchies (BVH)
    • Grids
    • Octrees
    • K-D trees
  – Ray bundles
Acceleration Strategies

• **Tracing rays in $O(n)$ is too expensive**
  – Need hundreds of millions rays per second
  – Scenes consist of millions of triangles

• **Faster ray-primitive intersection algorithms**
  – Do not reduce asymptotical complexity, “just” a constant

• **Reduce complexity through *pre-sorting data***
  – Spatial index structures
    • Dictionaries for ray tracing
  – Eliminate intersection candidates
    • Can reduce complexity to $O(\log n)$
    • Theoretical worst case still $O(n)$ – highly unlikely

• **Tracing of continuous bundles of rays**
  – Exploit coherence of neighboring rays, amortize cost among them
    • Frustum tracing, cone tracing, beam tracing, ...
Spatial Index Structures

- **Motivation:**
  - Fewer candidates that need to be tested for intersections

- **Approaches**
  - Spatial partitioning
    - (Hierarchically) partition of space or objects
      - Binary space partitions (BSP), K-D trees, Octrees
      - Bounding volume hierarchies (BVH)
      - Grids, hierarchies of grids
  - Directional partitioning (not very useful)
  - 5D partitioning (space and direction, once a big hype)
    - Close to pre-compute visibility for all points and all directions
    - Easily causes memory problems for the index
SPATIAL PARTITIONING
Aggregate Objects

- Object that holds groups of objects
- Stores bounding volume and pointers to children
- Useful for instancing and Bounding Volume Hierarchies (BVH)
Bounding Volumes

• **Observation**
  – Bound geometry with BV
  – Only compute intersection with children if ray hits BV

• **Sphere**
  – Very fast intersection computation
  – Often inefficient because too large

• **Axis-aligned bounding box (AABB)**
  – Very simple intersection computation (min-max)
  – Sometimes too large

• **Non-axis-aligned box**
  – A.k.a. „oriented bounding box (OBB)“
  – Often better fit
  – Fairly complex computation

• **Slabs**
  – Pairs of half spaces
  – Fixed number of orientations/axes: e.g. x+y, x-y, etc.
    • Pretty fast computation
  – Becomes more costly for 3D
Bounding Volume Hierarchy

• **Definition**
  – Hierarchical partitioning of the *set of objects*

• **BVHs form a tree structure**
  – Each inner node stores a volume enclosing all sub-trees
  – Each leaf stores a volume and pointers to objects
  – All nodes are aggregate objects

• **Object partitioning**
  – Usually every object appears exactly once in the tree
  – Can also allow for multiple instancing of objects or subtrees
    • Need to store a transformation (→ later)
Bounding Volume Hierarchy

- Hierarchy of groups of objects
BVH Traversal (1)

- **Accelerate ray tracing**
  - By eliminating intersection candidates

- **Traverse the tree**
  - Only if root node is intersected by the ray
BVH Traversal (2)

- **Accelerate ray tracing**
  - By eliminating intersection candidates

- **Traverse the tree**
  - Consider only objects in nodes intersected by the ray
BVH Traversal (3)

- **Accelerate ray tracing**
  - By eliminating intersection candidates
- **Traverse the tree**
  - Cheap traversal instead of costly intersection
  - Skip 2nd child if (non-overlapping) intersection found in 1st child
• **Accelerate ray tracing**  
  – By eliminating intersection candidates

• **Traverse the tree**  
  – Cannot stop in cases like this!!
Object vs. Space Partitioning

- **BVHs partition objects into groups**
  - They are still a spatial index structure

- **Next: Space partitioning**
  - Subdivide space in regions
  - Organize them in a structure (tree or table)
Uniform Grids

• **Definition**
  – Regular partitioning of space into equal-size cells
  – Each cell holds a reference to all the objects overlapping the cell

• **Bounding box of the grid**
  • For each object
    – Compute object bounding box
    – Expand grid bounding box

• **Resolution**
  – Want number of cells in $O(n)$
  – Resolution in each dimension $\propto \sqrt[3]{n}$
  – Usually $R_{x,y,z} = d_{x,y,z} \sqrt[3]{\frac{\lambda n}{v}}$
    • $d$: diagonal (vector)
    • $n$: number of objects
    • $v$: volume of the bounding box
    • $\lambda$: density (user-defined)
Uniform Grid Construction

• For each object:
  – Compute bounding box
  – Determine (conservative) lattice coordinates of the box
  – For each cell within the 3D lattice range
    • Add object reference to cell
Uniform Grid Construction – V2

- For each object:
  - Compute bounding box
  - Determine lattice coordinates of the box
  - For each cell within the 3D lattice range
    - Add object reference to cell only if the object intersects cell boundary

Optimization:
Check if whether an object intersects a given grid cell / AABB.
**Uniform Grid Traversal**

- **3D DDA (modified Bresenham algorithm, more later)**
  - Find lattice coordinates of starting cell
    - From intersection point with box if ray starts outside
    - From camera position if inside
  - Loop:
    - For each primitive in current cell
      - Check if ray intersects the primitive
      - If closest (positive) hit along the ray so far, record hit
    - If closest hit recorded is inside cell, then exit loop
    - Step to next cell in the structure
      - Exit loop if grid boundary is reached
Uniform Grid Traversal

• **Mailboxing**
  – Single primitive can be inserted in many cells
  – Avoid multiple intersections
  – Keep track of intersection tests
    • Per-object cache of ray IDs
      – Problem with concurrent access
    • Per-ray cache of object IDs
      – Data local to a ray
Nested Grids

- **Problem: „Teapot in a stadium“**
  - Uniform grids cannot adapt to local density of objects
- **Nested Grids**
  - Hierarchy of uniform grids
  - Each cell has a (potentially different) grid in itself

Cells of uniform grid
(colored by # of intersection tests)

Same for two-level grid
Octrees and Quadtrees

- **Octree (3D)**
  - Hierarchical space partitioning
  - Each inner node contains 8 equally sized voxels
    - Split in the middle

- **In 2D: Quadtree**

- **Adaptive subdivision**
  - Adjust depth to local scene complexity (e.g. # of objects in cell)
BSP Trees

• **Definition**
  – Binary Space Partition Tree (BSP)
  – Recursively split space with planes
    • Arbitrary split positions
    • Arbitrary orientations

• **Often used for visibility computation**
  – E.g. in games (Doom)
  – Enumerating objects in back to front order
kD-Trees

- **Definition**
  - **Axis-Aligned** Binary Space Partition Tree
  - Recursively split space with axis-aligned planes
    - Arbitrary split positions
    - X, Y or Z orientations
kD-Tree Example (1)
kD-Tree Example (2)
kD-Tree Example (3)
kD-Tree Example (4)
kD-Tree Example (5)
kD-Tree Example (6)
kD-Tree Example (7)
kD-Tree Traversal

- **“Front-to-back” traversal**
  - Traverse child nodes in order along rays

- **Termination criterion**
  - As soon as surface intersection is found (in current cell)

- **Maintain manual stack of sub-trees to traverse**
  - More efficient than recursive function calls
kD-Tree Traversal (1)

Current: A
Stack:
kD-Tree Traversal (2)

Current: B
Stack: C

Diagram of a kD-tree with nodes A, B, C, D, L1, L2, L3, L4, L5. The tree is traversed and the current node and stack are indicated.
kD-Tree Traversal (3)
kD-Tree Traversal (4)
kD-Tree Traversal (5)

Current: C

Stack:
kD-Tree Traversal (6)
kD-Tree Traversal (7)

Current: **L4**
Stack: **L5**  **L3**
kD-Tree Traversal (8)
kD-Tree Traversal (9)
kD-Tree Properties

• **kD-Trees**
  – Split space instead of sets of objects
  – Split into disjoint and fully covering regions

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

• **Compact representation**
  – Relatively little memory overhead per node
  – Node stores:
    • Split location (1D), child pointer (to both children),
      Axis-flag (often merged into pointer)
    • Can be compactly stored in 8 bytes
  – But replication of objects in (possibly) many nodes
    • Can greatly increase memory usage

• **Cheap traversal**
  – One subtraction, multiplication, decision, and fetch
  – But many more cycles due to dependencies & HW pipelines
Overview: kD-Trees Construction

- Adaptive
- Compact
- Cheap traversal
Building kD-trees

• **Given:**
  – Axis-aligned bounding box (“cell”)
  – List of geometric primitives (triangles?) in box

• **Core operation:**
  – Pick an axis-aligned plane to split the box into two parts
  – Sift geometry into two batches (left/right, some redundancy)
  – Recurse
  – Termination criteria!
Intuitive kD-Tree Building

- **Split axis**
  - Round-robin; largest extent

- **Split location**
  - Middle of extent or median of geometry (balanced tree)

- **Termination**
  - Target # of primitives, limited tree depth
Intuitive 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 are NOT very clever :-)**
Building good kD-trees

- **What split do we really want?**
  - Clever idea: the one that makes ray tracing cheapest
  - Write down an expression of cost and minimize it
  - Cost optimization

- **What is the cost of tracing a ray through a cell?**
  - Fixed cost per tree node
  - Plus probability of hitting left child
    - Given that I have hit current node
  - Times cost of left child
  - Same for the right node

\[
\text{Cost(cell)} = \text{traversalCost} + \text{Prob(hit L | hit P) } \ast \text{Cost(L)}
\]
\[
+ \text{Prob(hit R | hit P) } \ast \text{Cost(R)}
\]
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 (almost) empty space
Building good kD-trees

- **Need the probabilities**
  - Assume uniform directional distribution of rays
  - Probability turns out to be proportional to *surface area* (SA)
  - Area of outer surface of bounding box (not its volume)

- **Need the child cell costs**
  - Simple *triangle count* works great (rough approx.)

- **Surface Area Heuristic (SAH) – Important!!**

\[
\text{Cost(cell)} = \text{traversalCost} + \text{Prob(hit L | hit P)} \times \text{Cost(L)}
\]

\[
+ \text{Prob(hit R | hit P)} \times \text{Cost(R)}
\]

\[
= \text{traversalCost} + \frac{\text{SA(L)}}{\text{SA(P)}} \times \text{TriCount(L)} \times \text{intersectionCost}
\]

\[
+ \frac{\text{SA(R)}}{\text{SA(P)}} \times \text{TriCount(R)} \times \text{intersectionCost}
\]
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 decision over multiple levels, to avoid local minima

• 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)
    • Based on BBox of triangles (in/out events) or
      – Cost function is linear in between, so min/max must be at edge
    • **Predefined locations (fixed (small) number of bins)**
      – For large N the SAH is typically very smooth
  – Sort the triangle events or **bin them**
    • Binning can be done in constant time
  – Walk through candidate planes to find minimum cost split
  – Sift objects (while doing the computation for next level!!)
    • **Streaming operation!!**

• **Characteristics of the tree you’re looking for**
  – Deep and thin
  – Depth often 30 and more
  – A few objects per leaf
  – Big empty cells
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)
Fast Ray Tracing w/ kD-Trees

- Adaptive
- 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...**
  - Merge them into the lower bits of the pointer
No Bounding Box!

- kD-Tree node corresponds to an AABB
- Doesn’t mean it has to “contain” one
  - 6 floats: 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

• **Use the idea of “treelets”**
  – Small parts of the tree that are generated and stored together
  – Can have optimized layout (less/no pointers)
  – Can be stored in one cache line (efficient traversal)
Fast Ray Tracing w/ kD-Trees

- Adaptive
- Compact
- Cheap traversal
kD-Tree Traversal Operation

- **Maintain on a (manual) stack**
  - Entry and exit distance to node \((t_{\text{near}}\text{ and } t_{\text{far}})\)
  - Plus node still to be traversed

- **Three cases**
  - \(t_{\text{split}} \geq t_{\text{far}}\) Go only to near node
  - \(t_{\text{near}} < t_{\text{split}} < t_{\text{far}}\) Go to both
  - \(t_{\text{split}} \leq t_{\text{near}}\) Go only to far node

- **Near and far depend on direction of ray!**
kD-Tree Traversal: Inner Loop

given(node, t_near, t_far)

while ( ! node.isLeaf() )
{
    t_split = ( split_location - ray->P[split_axis] ) * ray_iV[split_axis]
    if (t_split <= t_near)
        continue with (far_child, t_near, t_far) // hit far child only
    if (t_split >= t_far)
        continue with (near_child, t_near, t_far) // hit near child only
    if (hit both children)
        push (far_child, t_split, t_far) onto stack
        continue with (near_child, t_near, t_split)
}

ray_iV is inverse of (normalized) direction vector (length along ray to do go one unit along that axis)
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
Summary: Exploit Advantages

• **Adaptive**
  – You have to build a good tree
  – Build a cost-optimized kD-tree w/ the surface area heuristic

• **Compact**
  – At least use the compact node representation: use 8-byte nodes
  – You can’t be fetching whole cache lines every time
  – Lay out your memory in a cache-friendly way

• **Cheap traversal**
  – No sloppy inner loops! (one subtract, one multiply!)
Can it go faster?

- How do you make fast code go faster?
- Parallelize it!
  - Not covered here
Bounding Volume Hierarchies

• **Today: Best Known Method (BKM)**
  – Traversal a bit more expensive (handling of overlap of BVs)
  – Building BVHs can be much faster
    • Using streaming construction (based on binary splits and SAH)
    • Often used to rebuild BVHs for entire dynamic objects per frame
DIRECTIONAL APPROACHES
Directional Partitioning

- **Applications**
  - Useful only for rays that start from a single point
    - Camera
    - Point light sources
  - Preprocessing of visibility
  - Requires scan conversion of geometry
    - For each object locate where it is visible
    - Expensive and linear in # of objects

- **Generally not used for primary rays**
  - GPUs can do this very efficiently

- **Variation: Light buffer**
  - Lazy and conservative evaluation
  - Store occluder that was found in directional structure
  - Test entry first for next shadow test
Ray Classification

- **Partitioning of space and direction [Arvo & Kirk´87]**
  - Roughly pre-computes visibility for the entire scene
    - What is visible from each point in each direction?
  - Very costly preprocessing, cheap traversal
    - Improper trade-off between preprocessing and run-time
  - Memory hungry, even with lazy evaluation
  - Seldom used in practice

![Diagram](image-url)