Computer Graphics

- Spatial Index Structures -

Philipp Slusallek
Overview

• **Last lecture**
  – Basic ray tracing
  – Intersection computations

• **Today**
  – Advanced acceleration structures
    • Grids
    • kd-Trees
    • Octrees
    • Bounding Volume Hierarchies
  – Ray bundles
Motivation

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

• **Reduce complexity through pre-sorting data**
  – Spatial index structures
    • Dictionaries of objects in 3D space
  – Eliminate intersection candidates as early as possible
    • Can reduce complexity to $O(\log n)$
  – Worst case complexity is still $O(n)$
    • *Private exercise: Come up with a worst case example*
Acceleration Strategies

- **Faster ray-primitive intersection algorithms**
  - Does not reduce complexity, “only” a constant factor (but relevant!)

- **Less intersection candidates**
  - Spatial indexing structures
  - (Hierarchically) partition space or the set of objects
  - Examples
    - Grids, hierarchies of grids
    - Octrees
    - Binary space partitions (BSP) or kd-trees
    - Bounding volume hierarchies (BVH)
  - 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

- **Tracing of continuous bundles of rays**
  - Exploits coherence of neighboring rays, amortize cost among them
    - Frustum tracing, cone tracing, beam tracing, ...
Aggregate Objects

- Object that holds groups of objects
- Conceptually stores bounding box and pointers to children
- Useful for instancing (placing collection of objects repeatedly) and for Bounding Volume Hierarchies
Bounding Volumes

• **Observation**
  – BVs (tightly) bound geometry, ray must intersect BV first
  – Only compute intersection 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
Bounding Volume Hierarchies (BVHs)

- **Definition**
  - Hierarchical partitioning of a 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
  - Usually every object appears once in the tree
    - Except for instancing
Bounding Volume Hierarchies (BVHs)

- Hierarchy of groups of objects
• **Accelerate ray tracing**
  – By eliminating intersection candidates

• **Traverse the tree**
  – Consider only objects in leaves intersected by the ray
• **Accelerate ray tracing**
  – By eliminating intersection candidates
• **Traverse the tree**
  – Consider only objects in leaves intersected by the ray
**BVH traversal (3)**

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

- **Traverse the tree**
  - Consider only objects in leaves intersected by the ray
  - Cheap traversal instead of costly intersection
Object vs. Space Partitioning

• **BVHs partition** objects into hierarchical 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

• Resolution
  – Want: number of cells in $O(n)$
  – Resolution in each dimension proportional to $3\sqrt{n}$
  – Usually $R_{x,y,z} = d_{x,y,z} \frac{3\lambda n}{V}$
    • $d$: diagonal (vector), $n$: #objects, $V$: volume of Bbox
    • $\lambda$: density (user-defined)
Uniform Grid Traversal

- **Grids are cheap to traverse**
  - 3D-DDA, modified Bresenham algorithm (see later)
  - Step through the structure cell by cell
  - Intersect with primitives inside non-empty cells

- **Mailboxing**
  - Single primitive can be referenced 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 (better!)
Nested Grids

• **Problem: „Teapot in a stadium”**
  – Uniform grids cannot adapt to local density of objects

• **Nested Grids**
  – Hierarchy of uniform grids
  – Each cell is a grid itself

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

Same for two-level grid
Octrees and Quadtrees

- **Octree**
  - Hierarchical space partitioning
  - Each inner node contains 8 (2x2x2 grid) equally sized voxels

- **Quadtree**
  - 2D “octree”

- **Adaptive subdivision**
  - Adjust depth to local scene complexity
BSP Trees

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

- **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
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 the current node)

• **Maintain stack of sub-trees still to traverse**
  – More efficient than recursive function calls
kD-Tree Traversal (1)
kD-Tree Traversal (2)
kD-Tree Traversal (3)

Current: L2
Stack: C
kD-Tree Traversal (4)

Current: C

Stack: C
kD-Tree Traversal (5)

Current: \[C\]

Stack:
kD-Tree Traversal (6)

Current: D
Stack: L3
kD-Tree Traversal (7)
kD-Tree Traversal (8)
kD-Tree Traversal (9)

Current: Δ △
Result: Δ
Stack: L5 L3
**kD-Tree Properties**

- **kD-Trees**
  - Split space instead of sets of objects
  - Split into disjoint 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 instruction dependencies
Overview: kD-Trees Construction

- Adaptive
- Compact
- Cheap traversal
Exploit 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!)
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 are not very clever**
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?**
  - Surface Area Heuristic (SAH)
    \[
    \text{Cost(cell)} = \text{C\_trav} + \text{Prob(hit L)} \times \text{Cost(L)} + \text{Prob(hit R)} \times \text{Cost(R)}
    \]
    - Cost of traversal of the inner node itself, plus
    - Relative probability of hitting one child, times
    - Cost of hitting that child
    - Same for other child
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* (SA)
  – *Not* the volume

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

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

\[
= C_{\text{trav}} + \frac{SA(L)}{SA(c)} \times \text{TriCount}(L) + \frac{SA(R)}{SA(c)} \times \text{TriCount}(R)
\]
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
  – But 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
    - Predefined locations (fixed number of bins)
  - Sort the triangle events or bin them
  - Walk through candidates to find minimum cost split

- **Characteristics of the tree you’re looking for**
  - Deep and thin
  - Typical depth of 50-100,
  - About 2 triangles 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)
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?
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…
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 Operation

- **Maintain on a stack**
  - Entry and exit distance to node (t_near and t_far)

- **Three cases**
  - t_split > t_far: Go only to near node
  - t_near < t_split < t_far: Go to both
  - t_split < t_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_at_split = ( split_location - ray->origin[spli_axis] ) * ray->inv_dir[spli_axis]
    if (t_split <= t_min)  // hit either far child or none
        continue with (far child, t_split, t_far)
    if (t_split >= t_max)  // hit near child only
        continue with (near child, t_min, t_split)
    // hit both children
    push (far child, t_split, t_max) onto stack
    continue with (near child, t_min, t_split)
}
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!
  – Not covered here
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**

- **Variation: Light buffer (for shadow rays)**
  - Lazy and conservative evaluation
  - Store last found occluder 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
Distribution Ray Tracing

• Formerly called Distributed Ray Tracing [Cook`84]
• Stochastic Sampling of
  – Pixel: Antialiasing
  – Lens: Depth-of-field
  – BRDF: Glossy reflections
  – Lights: Smooth shadows from area light sources
  – Time: Motion blur
Beam and Cone Tracing

- **General idea:**
  - Trace continuous bundles of rays

- **Cone Tracing:**
  - Approximate collection of ray with cone(s)
  - Subdivide into smaller cones if necessary

- **Beam Tracing:**
  - Exactly represent a ray bundle with pyramid
  - Create new beams at intersections (polygons)

- **Problems:**
  - Clipping of beams?
  - Good approximations?
  - How to compute intersections?

- **Not really practical !!**
Beam Tracing
Packet Tracing

• **Approach**
  – Combine many similar rays (e.g. primary or shadow rays)
  – Trace them together in SIMD fashion
    • All rays perform the same traversal operations
    • All rays intersect the same geometry
  – Exposes coherence between rays
    • All rays touch similar spatial indices
    • Loaded data can be reused (in registers & cache)
    • More computation per recursion step → better optimization
  – **Overhead**
    • Rays will perform unnecessary operations
    • Overhead low for coherent and small set of rays (e.g. up to 4x4 rays)