Acceleration: Spatial Trees

Christian Miller
CS 354 - Fall 2011
You probably noticed

- Your raytracer is slow
- Really slow
Why is that?

• Every pixel on the screen spawns a set of rays
• That set of rays will hit very, very little of the scene
• However, we’re currently checking every ray against everything in the scene
• Surely we can cull the number of things each ray needs to check
Spatial data structures

• A spatial data structure is a way of storing the scene such that certain queries can be performed efficiently

• An N-dimensional database

• We’re usually interested in performing distance, ray casting, and pairwise interaction operations

• Several different types
Spatial partitions

• You have a space, and its full of objects

• The idea: carve the space up somehow, and keep track of which objects are in which subspaces

  • Do this hierarchically

• When making a query of the data structure, you can usually find ways to skip entire subspaces (and the objects in them)
Spatial grid

• Define a 3D grid in space, then add each object inside a cube to a list of objects for that cube

• During traversal, rasterize the ray onto the grid (using Bresenham’s or similar), then perform intersection checks against all cells it overlaps

• Setup and traversal are very fast, but it can require a lot of memory and tweaking
Spatial grid
**Spatial grid**

- Find bounding box for your scene
• Discretize at some resolution
Spatial grid

- Add overlapping objects to a list for each grid cell
Spatial grid

• Optionally check for intersections
Spatial grid

- Rasterize ray through grid, only check for intersections in those cells
Quad / Octrees

- If there are lots of empty cells, we might want to create cells adaptively
- Take a bounding box of the scene, carve it into 4 pieces (2D) or 8 pieces (3D)
- This makes a hierarchical representation of the scene
Quad / Octrees
Building quadtrees

- Objects are stored in lowest node that can fit them
Traversing quadtrees

• Intersect ray with the current node, if it hits:
  • Do regular intersection testing on all objects in the node
  • Recurse to all children of the node
• This skips all objects in boxes the ray doesn’t intersect
BSP trees

- Short for Binary Space Partition
- Take the space, pick a plane that cuts it in half
- This gives you two subspaces, and you can recurse into them
- Keep building the tree until each leaf has less than a certain number of objects
- If an object straddles the plane, clip it or just add it as is to both child nodes
BSP Trees

Diagram showing a BSP tree with nodes labeled A, B, C, D, E, F, and G. The tree structure is represented on the right side of the image, with 'splitting plane' indicating a plane that divides the space.
BSP traversal for rays

- Find out how the ray intersects the split plane
  - Recurse on the near side first, then the far side
  - You can ignore a subspace if it’s behind the ray origin
- Once you hit a leaf node, check the ray against all objects in that node
  - If you find an intersection, return it immediately
BSP traversal

- Figure out which subspace to traverse first (A)
BSP traversal

- Check objects in A
BSP traversal

- Hit triangle, return intersection
BSP traversal

- Figure out which subspace to traverse first (A)
BSP traversal

- Check objects in A, no intersection found
BSP traversal

- Check objects in B, return intersection with circle
Traversals

- Traversal goes from $O(n)$ to $O(lg \ n)$
- Allows you to skip intersection tests with most objects in the scene, at the cost of checking plane intersections at each node
  - These are still way cheaper
- You have to build the data structure before you can use it, and that can be very expensive
- Only effective if you pick good split planes!
Picking a split plane

- You want to carve up space such that roughly half of the objects are on each side of the plane
  - Maximizes the work saved during traversal
- A common way to do this ($O(n \ lg \ n)$ build time):
  - Every triangle in a subspace lies in a plane
  - Try each of those planes and see how well they split the data
  - Just take the best one of those as the split plane
BSP build cost

- There are many ways of picking split planes, but the decent ones are very expensive
- BSP trees are usually built as a preprocess
  - Frequently done for game levels
- If you’re not careful, you can spend as long building the tree as you do rendering!
KD-tree

• Like a hybrid spatial grid / BSP tree
• Split planes are arbitrary, but axis-aligned
Picking KD split plane

- Pick an axis (usually the maximum dimension)
- Take the median of all object centroids along that axis (called median-split)
- There are other methods, but this is simple and still performs well
- Still $O(n \log n)$, but constant is much lower than building a general BSP tree
KD Traversal

- A bit more efficient than general BSP traversal
- Can skip half-spaces that ray doesn’t intersect
Bounding volume hierarchy (BVH)

- Another idea for spatial data structures
- You probably already have a scene graph defining the relative transforms of objects
- Add bounding volume information (like a box) to each object, then union those boxes together up the hierarchy
- During traversal, start at the root and only check the contents of a box if you hit the box itself
Types of BVs

- Bounding boxes are the most common choice, but you can pick any shape that surrounds the object.
- Spheres, cylinders, ellipsoids, convex hulls, k-DOPs, etc.
- Generally works well as long as the scene graph doesn’t get too complicated.
Sphere tree

- Each object is bounded by a sphere, and the internal nodes bound each group
**BVH efficiency**

- Usually about as fast as KD-trees during traversal
- Build time is not really considered, since you assume it comes from a preexisting scene graph
- Requires pre-existing knowledge of scene hierarchy
- You can even combine the two methods:
  - Have a BVH for all objects, then put a KD-tree at the leaf nodes to make complex object queries faster
Using Spatial Data Structures

- Remember: rays are not the only queries that can be accelerated
  - Closest object, intersection, K-nearest, etc.
- One data structure usually helps with many different queries
Usage breakdown

• BSP tree: any time you can preprocess something, including static meshes or models

• KD-tree: quickly build and traverse a scene, excellent for K-nearest neighbors

• BVH: any time you already have a scene graph, but no acceleration structure to go with it

• Other types for more specific purposes...