Hierarchical Spatial Data Structures for Computer Graphics

TU Berlin - CG 2 – 2008

Tamy Boubekeur
Overview

• Spatial Grids
• Dimensional Trees
  – Quadtrees
  – Octrees
• kD-Trees
• BSP-Trees
• Implementation
• Comments
3D Objects

- Geometry + Appearance + Animation + Physics attributes + ..... 
- Representations are application-dependent
- Explicit geometry usually required
  - Several geometric representation for 3D models

- Today: just P, a set of points in the 3D space
  - Geometric sample or surface element p in P = point \{x, y, z\}

Book: *Fundations of Multidimensional and Metric Data Structures*, by Hanan Samet
Manipulating 3D Geometry

• Fields:
  – Geometric modeling
  – Image synthesis
  – Digital Animation

• Local processing
  – Avoiding to consider the entire model for each “question” the program may ask

→ Object partitioning

• Cutting the object in pieces
• Considering only the neighboring pieces when processing a given spatial location
Grids

• The straightforward solution!
  – Generate a 3D grid embedding the model
  – Classify the elements according to the cell they intersect
    • 3D rasterization
  – efficient, simple, enough in many cases
  – not adaptive, many empty cells and/or many over-populated cell
  – Still O(n)
    • Reduce the constants, not the order of complexity
  – The popular solution: trees!
Hierarchical Data Structures

• Everywhere in graphics:
  – Modeling
    • *Clustering, analysis, simplification, reconstruction, LOD generation, etc*
  – Rendering
    • *Ray tracing, photon mapping, radiosity, visibility, ambient occlusion, etc*
  – Animation
    • *Collision detection, crowd simulation, physically-based deformations, etc*
  – Virtual Reality
    • *Scene graph, LOD selection, parallel computing, etc*
Hierarchical Data Structures

• Main structures:
  – **kD-Tree** [Bentley 1975]: orthogonal organization of a set of samples
  – **BSP-Tree** [Fuchs et al. 1980]: « Binary Space Partition » tree, a binary and recursive space subdivision performed with hyperplanes and organized in a binary tree
  – **Quadtree/Octree** [Jackins & Tanimoto 1980]: dimensional structure (1-4 in 2D, 1-8 en 3D)
  – **BVH**: Bounding volume hierarchy

• Can be combined for a specific application.
Hierarchical Data Structures

• Standard questions to answer:
  – **Point intersection**: which partition contains a given 3D point?
  – **Ray intersection**: how to order partitions along a ray?
  – **Neighborhood query**: considering a point p, what are
    • the k nearest samples?
    • the set of samples within a distance r?
  – **Walking / General spatial ordering**
QuadTree

- Recursive 1-4 split in the plane (4 children by node)
- Internal node: « symmetric axis-aligned structure »
Octree

- The 3D version of the quadtree
- A special case of *bounding cube hierarchy*
- *Static* clustering policy
  - Split cubes in 8 equal cubes

![Level 0](image1.png)  ![Level 1](image2.png)

Note: surface partition induced by volume partition
A very simple recursive implementation

```c
TYPE OctreeNode {
    OctreeNode children[8];
    Data data;  // Bounding Cube + misc. data (e.g. 3D points, color, etc)
}
OctreeNode buildOctree (Data data) {
    OctreeNode node;
    if (stopCriteria (data))
        init (node, data);  // fill children with NULL and affect data
    else
        Data * childData[8];
        dataSpatialSplit (data, dataChild);  // 8-split of the bounding cube and partitioning of data
        for (int i = 0; i < 8; i++)
            node.children[i] = buildOctree (childData[i]);
        node.data = NULL;
    return node;
}
Can also be implemented in an array (no pointer, quasi-perfect tree) or in a hashtable (Morton code)
```
**kD-Tree**

- Orthogonal binary tree
- Can be seen as a product of spatial orderings
- At each level:
  - Compute the bounding box
  - Split the box the longest edge

**Algorithm:**

```java
KDNode buildKDTree (PointList P) {
    BBox B = computeBoundingBox (P);
    Point q = findMedianSample (B,P);
    Node n;
    Plane H = plane (q, maxAxis (B))
    n.data = <q,H>;
    PointList Pu = upperPartition (P, H);
    PointList Pl = lowerPartition (P, H);
    n.leftChild = buildKDTree (Pu);
    n.rightChild = buildKDTree (Pl);
    return n;
}
```
kD-Tree Properties

- A beautiful algorithm
  - Spatial sorting
  - Dimension independent
  - Few numerical issues (for point sets)
- Very popular for nearest neighbors search
  - Range search: find all samples within a distance $r$ to the $p$
    - Trivial to implement using sphere-box intersection tests
  - kNN search: find the $k$ nearest neighbors
    - Various methods:
      - Direct: use a fixed size priority queue to order the nodes (O($k \log N$) for a balanced tree)
      - Iterative: loop over a guess/adjust range search, better in parallel context (GPU, multicore, etc)
- Raytracing
  - Can handle packets of rays
  - Geometric bias to balance a kD-Tree
    - e.g. Surface Area Heuristic, or SAH for triangles
BSP Tree

- Binary Space Partition Tree
- On each node: an hyperplane, with arbitrary location and orientation
BSP Tree

- Cutting planes:
  - Axis Aligned
    - Very similar to a kD-Tree
  - PCA analysis (*top-down*)
    - Principal Component Analysis on the set of sample P
      - Compute the covariance matrix M of the set
      - Eigen analysis of M
    - Use the centroid of P and the eigen vector associated to the largest
eigen value as a split plane.
  - Variational partitioning (Lloyd relaxation)

- **Difference kD-Tree / axis-aligned BSP?**
  - kD-Tree: *organization* structure, axis-aligned by definition
    - more efficient for point sets
  - BSP: more accurate spatial partitioning (the “pyramid” example)
    - more accurate

*Nothing fixed : you decide !*
Comparison

- **Adaptivity** at a given depth
  - Octree
  - kD-Tree
  - BSP-Tree

- **Simplicity** of code/Construction **Time**
  - Octree
  - kD-Tree
  - BSP-Tree

*Good trade-off*

*Efficient heuristics to guarantee log (N) queries*

*Static ray-tracing*
Comparison

Grid

Octree

BSP
Bounding Volume Hierarchy

- BVH vs Octree/kd-Tree/BSP-Tree:
  - Node $\neq$ sum of its children
  - BV of the same BVH may intersect
  - BVH nodes have arbitrary valence
  - Arbitrary BV shape
    - Usually convex for simple intersection test
  - Construction principle:
    1. Find the smallest BV of $P$
    2. Split BV in $n$ sub-BV
    3. Classify $P$ in the $n$ sub-BV and restart
- Classical BV: Bounding Sphere (BS), Axis-Aligned Bounding Box (AABB), Oriented Bounding Box (OBB)
Bounding Volume Hierarchy

Almost useless for geometric processing

Very successful for dynamic models:
- **BD-Trees** for scalable collisions
- BVH for interactive raytracing of dynamic scenes
- Superior to kD-Trees when the construction time matters
  - no triangle split

Very successful in virtual reality:
*Inventor, OpenSG, OpenSceneGraph, NVSG, VRML, X3D, COLLADA*… almost all scenes graphs are “enriched” BVH
Partitioning criteria

• When the recursive spatial subdivision should stop?
  – Max depth: uniform, hard to predict in general
  – Density-based: less than m samples per node
  – Error-driven: when the m samples of node no more violate a certain predicate
    • Geometric error metric (L2, QEF, mean/max curvature, etc)
    • Visual error metric (saliency, color variance, shader complexity)
    • etc

Note: in many applications, bucket trees
Implementation

• Pointers: simple, easy to manipulate
• Explicit offset: good for « complete » trees
  – None NULL children for internal nodes
• Hash-table: use a spatial location key
  – e.g. the Morton code
  – Good for optimization
• Construction scheme:
  – Breadth-first (BF): use a queue, good when there is a « budget » of nodes
  – Depth-first (DF): use a stack (implicit in recursive calls)
Implementation

• Highly programmable GPUs, multi-core CPUs: what about the parallel construction?
  – More or less still an open problem

  – One simple strategy: start with BF to “feed” enough threads, then switch to DF
    • Clearly sub-optimal for high density variation
    • Good for cache, map easily to CUDA for instance
    • The BF/DF switch is not always clear (early DF?)
Optimization

• Partial completion:
  – Made *perfect* the first M levels of the tree
    • *Direct* access up to depth M
    • Shrink significantly the query time for a negligible amount of memory

• Lifting scheme
  – « Local offset » for table implementation
  – Offset quantization on less bits

• « Use internal nodes »
  – In many applications, can describe roughly “what’s going on” in their sub-tree.
Exotic Trees

• Many trees are specialized graphics data structure:
  – **Volume-Surface Tree**: combines octree and local quadtrees for surface partitioning and processing
  – **Tile-Trees**: tree of « cube faces » for texturing surfaces
  – **Bounding Interval Hierarchy**
  – **Tetrahedra Hierarchy**: canonical finite element structure for physics simulations (volume partitioning)
Final word

• Recursive grids
  – Grids made hierarchical
  – Generalization of Octrees
  – Dual or primal

  – In practice: only few levels are necessary
  – Very convenient when accurate memory usage is not mandatory