Acceleration Data Structures for Ray Tracing

Travis Fischer and Nong Li (2007)
Outline

Introduction/Motivation

Bounding Volume Hierarchy

Grid/Voxel

Octrees

kd-Trees/SAH

Other acceleration data structures

Miscellaneous optimizations
Bounding Volumes

• Extents and Bounding Volumes:
  – bound each complex object with simpler one
    • examples of simpler volumes: sphere, cuboid
  – if bounding volume isn’t visible, neither is object inside it!
  – can put multiple objects into one volume: more efficient

• Great for ray-tracing pipeline!
  – quick reject: check ray against bounding volume first
  – quicker reject: check group of rays (frustum) against bounding volume of object

• Very easy to implement and can offer noticeable speedups

• Works well if object has tight bounding box
  – the better the bounding box, the more difficult it is to compute (ie. chair)

• Many data structures use Axis Aligned Bounding Boxes (AABB)
  – AABB is a cuboid such that each edge is parallel to either the x, y, or z axis
Bounding Volume Hierarchy (1/3)

A technique for organizing entire model

- Extends the idea of bounding boxes by combining adjacent bounding boxes to create a tree hierarchy
  - bottom-up construction

- Repeatedly group bounding volumes of nearby objects together until entire scene is bounded
  - finding objects that are close to each other can be very difficult (naïve $O(n^2)$, can be done in $O(n \log n)$ with sorting)
  - Problem of having tight bounding boxes worsens.

- Could use original scene graph: bounding volumes at nodes = union of child bounding volumes
  - easy to construct...
  - scenegraph may be logically organized but may not be spatially organized
Bounding Volume Hierarchy (2/3)

Applications in Ray Tracing

- Raytracing: if ray intersects parent, check the children
Summary

• BVH can work very well for a subset of scenes and is easy to implement.

• Performance can improve a lot with additional user input
  – the scene gives better clues how to group objects
  – works well for video games (you know the scenes beforehand)

• Does not handle arbitrary scenes
  – will have no performance gains
  – examples:
    • close-up of very detailed mesh
    • landscape with grass and trees
Grids (1/2)

Partitioning Space

• Instead of bounding objects bottom-up, break up space into regularly-sized cells
  - easy/fast to construct -- great for animated scenes
  - can use line-scanning algorithms for traversal
  - think of cells as pixels -- ray traversal as scan-converting a line
  - because of this, can be easily implemented in hardware

• 2D Example:

  What’s wrong with this approach?
Grids (2/2)

Pros/Cons

- ‘Teapot in a Dome’ syndrome\(^1\)
  - unbalanced, some cells are clearly more important than others
  - how to determine the best cell size?
- Why don’t we just use a finer grid?
  - expense of stepping through grid cells during traversal
  - analogous to super-sampling from image processing unit: increasing resolution on monitors decreases visible effects of aliasing, but only at the expense of a significant amount of overhead (memory/processing)
- Not hierarchical
  - Grid partitioning is still useful for animated scenes: moving one object does not affect other objects in grid
- In general, however, we’d like a smarter, more adaptive solution

\(^1\) Teapot Dome Scandal during Warren G. Harding’s Presidency
Octrees\(^1\) (1/5)

**Adaptive data structures**

- Combine advantages of BVH and grids
  - scene agnostic (grids)
  - adaptive and hierarchical (BVH)
  - viewer independent
- Similar to grid except voxels do not have to be the same size.
  - more voxels where geometry is densest
- All nodes in the tree are AABBs

\(1\) Octrees are based on Warnock’s algorithm for hidden surface removal
Octrees (2/5)

Construction

- Rather than bottom up construction (BVH), construct top down.
- Find the bounding box of the scene
  - this is the root of the tree and contains all the primitives
- At every iteration, partition the current node into 8 octants
  - easy to do if bounding boxes are AA
- Split the primitives at the current node into the octants
  - if a node spans a split, put it into both octants
  - recur on the new nodes
  - terminate at a maximum depth or if a voxel has sufficiently few primitives
- There are variations that can have slightly better performance
  - collapse empty neighbors
Octrees (3/5)

**Traversal**

- Begin at the root node.

- If node is a leaf node, perform intersection on all of its children.

- For internal nodes, iterate through the node’s children and compute intersection between ray and child’s bounding box
  - if it intersects, recur on child
Octrees (4/5)

Traversal
Octrees (5/5)

Summary

- Octrees are able to handle arbitrary scenes with good performance
- Expected runtime $O(\log n)$ per ray.
  - exponential speedup over linear solution
- For scenes where primitives are distributed very non-uniformly, octrees will perform terribly
  - in practice, these scenes are fairly common
  - octrees can take many subdivisions to zone in on complex geometry, yielding deeper, inefficient trees
Motivation

- Leads us towards a solution which takes into account the relative cost of stepping through grid cells with the possible intersection tests performed
  - would like to quickly isolate areas of high complexity and produce large empty spaces which we can step through quickly during traversal

- Definition: a **kd-Tree** is a *k*-dimensional, axis-aligned binary tree.
  - called a quadtree if *k* equals 2
  - axis-aligned => quicker traversal
  - binary tree => unlike octree, choose one axis to split along at each node

- Main challenge with kd-trees used for spatial partitioning is determining where to position the split plane at a given node (including which axis to split along)
  - would like split plane to reflect geometry of objects
  - degenerate kd-tree is identical to its octree equivalent
  - let’s look in-depth at how we would select one split for a single node
**Choosing a Split-Plane**

- **2D Example:**

  - How do we split up this “node” in space optimally?
  - Split at middle?
    - During traversal, ray is equally likely to enter left and right sides.
    - But cost of entering right side is much higher.
  - Split at median?
    - Cost of entering each side during traversal in terms of possible intersection tests is approximately equal.
    - But a ray is much more likely to enter left-hand side – much greater area.

**Cost-Optimized Split**

- Attempts to balance the cost of entering a node (intersection tests) with probability of entering that node.
- Isolates geometric complexity rapidly, creating large, empty nodes that can be quickly discarded during traversal.

**Where to split children?**

- Left child requires no further subdivision.
- Right child split roughly at middle/median.
- No clear winner for future splits in right child; will look like octree from this point onwards (objects now uniformly distributed).
- Difference is that we have isolated this geometry after only one split whereas octree would take longer (objects not isolated).

---

Andries van Dam  
November 10, 2009  
Acceleration Data Structures  
15/35
kd-Trees (4/7)

**Construction**

- Pseudocode for constructing kd-Tree:

  ```plaintext
  Given AABB of current node and list of primitives L
  if there are too few primitives
    create leaf node and return
  else
    choose split axis a (i.e. one of the k dimensions)
    choose split plane p (along axis a)
    determine which primitives in L intersect left child and right child respectively
    recursively subdivide left child
    recursively subdivide right child
  ```

- Selecting a good split plane is crucial to performance.

- Constructing cost-optimal kd-tree is NP-complete, so we do the best we can locally at a given node and use a greedy **Surface Area Heuristic** as an approximation (see next slide)
kd-Trees (5/7)

Surface Area Heuristic

- Pseudocode for construction with SAH:

  Given AABB of current node and list of primitives $L$
  
  foreach possible split position
  
  $$成本 = \text{Cost}_{\text{traversal}} + \text{Prob}_{\text{left}} \times \text{Cost}_{\text{left}} + \text{Prob}_{\text{right}} \times \text{Cost}_{\text{right}}$$
  
  keep track of split plane which minimizes this cost

- Probability calculation assumes rays will be distributed evenly throughout space
  
  - $\text{Prob}_{\text{child}} = \text{ratio of child AABB’s surface area to parent node’s surface area}$
  
  - $\text{Cost}_{\text{child}} = \text{number of primitives contained in child}$
  
  - $\text{Prob}_{\text{child}} \times \text{Cost}_{\text{child}} = \text{Expected cost of entering child node}$

- Number of possible split positions is infinite
  
  - which ones should we consider?
  
  - only consider splits at extrema of primitives on split axis

- Complexity Analysis: do high construction costs nullify the advantage we gain during traversal?
  
  - naïve construction takes $O(n^2 \log n)$
  
  - how can we make construction faster?

- Sort primitives along split axis
  
  - at each node $\Rightarrow O(n \log^2 n)$
  
  - once at the root $\Rightarrow O(n \log n)$
  
  - memory I/O is generally the limiting factor
kd-Trees (6/7)

**Traversal**

- Can use identical traversal algorithm as octree

- We can use early termination during traversal
  - this technique also applies for previous data structures

- Instead of traversing the two children in arbitrary order, pick the child that will be hit by the ray first.
  - first child is often called “frontside child”
  - second child is “backside child”

- If there is an intersection in the frontside child that is earlier than the intersection to the backside child’s bounding box, no need to traverse backside child
**Edge Cases**

- **Traversal edge cases** – must be really careful when early termination is safe
  - the ray traverses node F before node G. When traversing F, the ray intersects the green rectangle but you can not terminate yet.
  - must compare intersection $t$ values with $t$ values for bounding boxes.

- **During construction, what happens if the primitive is in the split plane?**
  - architectural models

- **Rays that are inside/parallel to split plane**
  - divide by 0!
Summary of Raytracing
Acceleration Data Structures

- Bounding Volume
- Hierarchy
- Octrees
- Grids
- kd-tree
Binary Space Partitioning (BSP) Trees (1/4)

- Used for polygon VSD (not raytracing)
- Construct a binary tree with spatial subdivision
  - non axis-aligned version of kd-tree
  - no cost evaluation; choose split plane arbitrarily
- An example:

![BSP-0: Initial Scene](image)
Binary Space Partitioning (BSP) Trees (2/4)

BSP-1: Choose any polygon (e.g., polygon 3) and subdivide others by its plane, splitting polygons when necessary

BSP-2: Process front sub-tree recursively
Binary Space Partitioning (BSP) Trees (3/4)

BSP-3: Process back sub-tree recursively

BSP-4: An alternative BSP tree with polygon 5 at the root
Binary Space Partitioning (BSP) Trees (4/4)

- kd-tree visited nodes front-to-back during raytracing traversal
- whereas BSP nodes are visited in back-to-front order. The far side of a node is the side that the viewpoint is not in (view-dependent)
- very fast to build. Quake III uses this for occlusion culling and to speed up intersection testing

```c
void BSP_displayTree(BSP_tree* tree) {
    if (tree is not empty) {
        if (viewer is in front of root) {
            BSP_displayTree(tree->backChild);
            displayPolygon(tree->root);
            BSP_displayTree(tree->frontChild);
        }
        else {
            BSP_displayTree(tree->frontChild);
            // ignore next line if back-face culling desired
            displayPolygon(tree->root);
            BSP_displayTree(tree->backChild);
        }
    }
}
```

Advanced Techniques (1/2)

Portals

- Indoor spaces are mostly rooms with doorways
- Why draw the geometry in the next room if the door is closed?
- If there is a portal (open door or hallway) only draw geometry visible through the portal
- Really useful as a pre-computation step – geometry visible through a portal remains constant - not too good for outdoor scenes

Advanced Techniques (2/2)

Occlusion Culling

- If a big object fills a good portion of the screen, don’t draw geometry that it covers up
- Many new graphics cards have support for parts of the process

![Without OC – lots of geometry drawn, most is not seen (drawn in blue)](image)

![With OC – algorithm ignores blue geometry behind the hills](image)

- Algorithm:
  - Create list of all objects potentially visible in frustum (per polygon or per shape)
  - For each pair of objects i and j, if i occludes j (i.e. i lies in j’s shadow volume) remove j
  - Terrain can be conservatively clipped by clipping against projection of frustum against terrain projected onto ground plane
- $O(n^2)$! Lots of ways to make this faster:
  - Coorg, S., and S. Teller, "Real-Time Occlusion Culling for Models with Large Occluders", in *1997 Symposium on Interactive 3D Graphics*
- Bad for indoor scenes with lots of small objects
Quiz

• Which data structure would be best to use in the following situations?

• An animation where only the camera moves?
  – kd-Tree

• Rendering scenes in a Pixar-like movie?
  – depends on the shot; most likely a combination of techniques are used

• A fast-paced video game with lots of interactive, moving objects?
  – regular grid or BSP-Tree

• Your ray-tracing assignment in CS123?
  – kd-Tree
Optimization Principles

• Most of your rays are secondary rays (ie: shadow rays)
• Ray tracing performance is very memory I/O bound
  – page and cache misses
Miscellaneous Optimizations

- Mailboxing\(^1\)
- Shadow caching\(^1\)
- Compact node formats\(^1\)
- Contiguous memory layout
- Multithreading
  - Makes mailboxing tricky
- Packet traversal
  - SIMD
- Combining spatial data structures
  - BVH + kd-trees
- More efficient ray-primitive intersections
  - Slab method for boxes\(^1\)
  - Baryocentric coordinates for triangles

- Using these techniques, state of the art ray tracing engines are able to get ~10fps on scenes with 100K primitives

- PhD thesis by Ingo Wald describes these optimizations in more detail and has coding examples!

1) Easy to implement in your raytracer for CS123
Compact Node Representation (kd-tree) 1/3

• The nodes we used for kd-trees contain a lot of redundant information
  – AABB share multiple vertices
  – Parent and child share multiple vertices
  – Various low level optimizations

```c
struct crappyKdNode {
    float min[3];
    float max[3];
    int leafNode;
    int splitAxis;
    float splitPlane;
    int nPrimitives;
    Primitive* primitives;
    kdNode* leftChild;
    kdNode* rightChild;
};
```

• sizeof(crappyKdNode) = 52 Bytes

• We can do much better!
Compact Node Representation (kd-tree) 2/3

- We can represent the same data in 8 Bytes

```c
struct kdNode {
    union {
        kdNode* child;
        unsigned int nPrimitives;
        unsigned int flag;
    };
    union {
        Primitive* primitives;
        float splitPlane;
    };
};
```

- All of the node information is bit packed into the flag.
  - Lower 2 bits contain split axis & leaf boolean
  - Upper bits are either pointer to child (word aligned) or the number of primitives (bit shift)
  - Siblings are adjacent in memory (only need to store pointer to the first one)
Compact Node Representation (kd-tree) 3/3

Traverse left child
Traverse right child
Compact Node Representation (kd-tree) 3/3

Traverse right child
Compact Node Representation (kd-tree) 3/3

Traverse left child
Example

Primitives: 23437 Spheres  Render Time: 135/6 mins