# *Data Structures & Algorithms for Geometry*

- Agenda:
  - Quiz #1.
  - Assignment #1 due.
  - Bounding volumes.
    - Overview
    - Creation
    - Intersection
  - Assignment #2 assigned.


© Copyright Ian D. Romanick 2007

# *What about convex hulls?*

- Putting them off until later in the term.
  - For 3D hulls, we really need some data structures that we haven't covered yet.
  - I'm going to bump space partitioning by a couple weeks.
  - Move polygon representation up.
  - Insert convex hulls.
- Updated sylabus available on-line.
  - Please refer to it for updated reading assignments.
  - You *are* doing the reading, right?


© Copyright Ian D. Romanick 2007

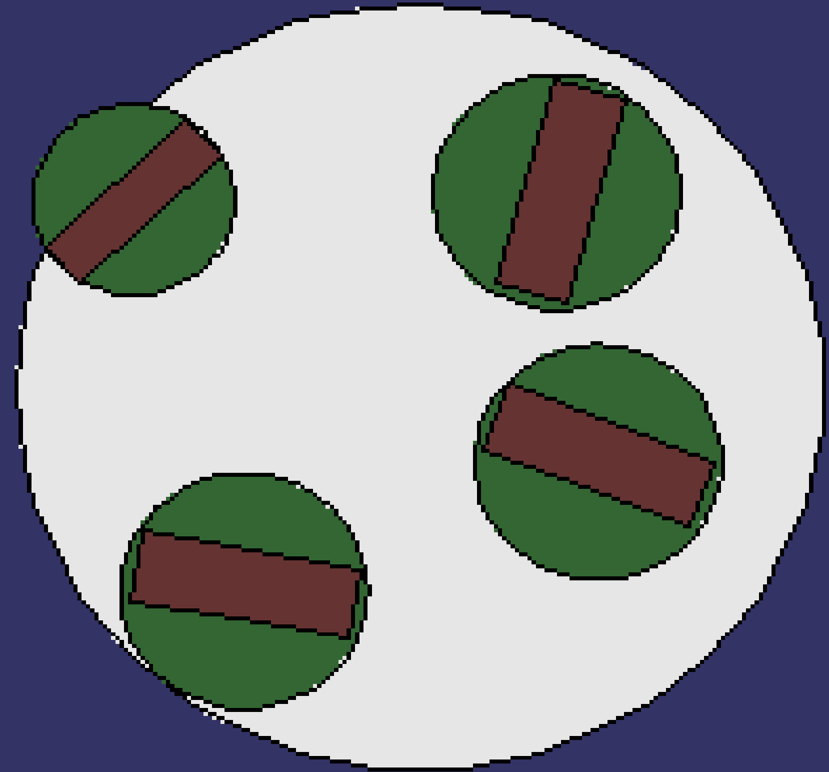
# Bounding Volume Hierarchies

- BVs containing BVs containing BVs etc.
- Arrange BVs in a tree-like structure.
  - Each larger BV stores references to its immediate sub-BVs.
- Similar to space partitions, but some key differences.
  - Multiple BVs at the same level of the tree may occupy the same space.
  - Objects typically only stored in one BV.


© Copyright Ian D. Romanick 2007

# Parent-child Property

- Each parent BV contains its child BVs.
  - Makes things easier.
  - Not required!
    - Parent BV need only contain the *objects* in the child BVs.
    - Top level circle (right) contains all boxes, but not all sub-circles.




© Copyright Ian D. Romanick 2007

# Desirable BVH Characteristics

- The nodes within any subtree should be “near” each other.
  - Farther down in the tree, the nodes should be nearer.


© Copyright Ian D. Romanick 2007

# Desirable BVH Characteristics

- The nodes within any subtree should be “near” each other.
  - Farther down in the tree, the nodes should be nearer.
- Each node should be of minimal volume.
  - Just like BVs!


© Copyright Ian D. Romanick 2007

# *Desirable BVH Characteristics*

- The nodes within any subtree should be “near” each other.
  - Farther down in the tree, the nodes should be nearer.
- Each node should be of minimal volume.
  - Just like BVs!
- Nodes near the root are more important than nodes near the leaves.
  - Prune as many objects as soon as possible.


© Copyright Ian D. Romanick 2007

# *Desirable BV Characteristics (cont.)*

- Volume overlap of sibling nodes should be minimal.
  - Overlap may force traversal of multiple subnodes. Yuck!


© Copyright Ian D. Romanick 2007

# Desirable BV Characteristics (cont.)

- Volume overlap of sibling nodes should be minimal.
  - Overlap may force traversal of multiple subnodes. Yuck!
- Hierarchy should be balanced w.r.t. node structure *and* content.
  - Balanced structure is just like regular search trees.
  - Balanced content (i.e., number of objects in subtrees) allows earlier pruning of objects from consideration.


© Copyright Ian D. Romanick 2007

# *Desirable BV Characteristics (cont.)*

- Worst case performance should not be much worse than average case.
  - Want to avoid sudden drops in framerate.


© Copyright Ian D. Romanick 2007

# Desirable BV Characteristics (cont.)

- Worst case performance should not be much worse than average case.
  - Want to avoid sudden drops in framerate.
- Generate without human intervention.
  - Automatically generate BVH from data without artists or (worse) programmers having to help it / do the work.


© Copyright Ian D. Romanick 2007

# Desirable BV Characteristics (cont.)

- Worst case performance should not be much worse than average case.
  - Want to avoid sudden drops in framerate.
- Generate without human intervention.
  - Automatically generate BVH from data without artists or (worse) programmers having to help it / do the work.
- Memory usage should be low.
  - Just like BVs.


© Copyright Ian D. Romanick 2007

# *BV Cost*

$$T = N_v C_v + N_p C_p + N_u C_u + C_o$$

- $N_v$ – number of volume-volume tests
- $C_v$ – cost of volume-volume test
- $N_p$ & $C_p$ – number & cost of primitive tests
- $N_u$ & $C_u$ – number & cost of node updates
- $C_o$ – fixed, one-time cost (i.e., translate one BVH to the other BVHs coordinate space).


© Copyright Ian D. Romanick 2007

# *BV Cost (cont.)*

- Why is this important to think about?


© Copyright Ian D. Romanick 2007

# *BV Cost (cont.)*

- Why is this important to think about?
  - All of the values are interrelated.
    - Decreasing BV's volume may reduce $N_v$ but increase $C_v$.
  - Typically $C_p \gg C_v$.
    - Decreasing $N_p$ at the expense of increasing $N_v$ is generally a win.
  - Generally, gives a framework to compare the expected performance of different BVHs.


© Copyright Ian D. Romanick 2007

# *Tree Degree*

- What is “tree degree”?


© Copyright Ian D. Romanick 2007

# *Tree Degree*

- What is “tree degree”?
  - Number of branches (children) from each node.
- What difference does it make?


© Copyright Ian D. Romanick 2007

# *Building a BVH*

- Three common strategies for building trees.


© Copyright Ian D. Romanick 2007

# *Building a BVH*

- Three common strategies for building trees.
  1. Insertion – Build the hierarchy incrementally by adding one element at a time.


© Copyright Ian D. Romanick 2007

# *Building a BVH*

- Three common strategies for building trees.
  1. Insertion – Build the hierarchy incrementally by adding one element at a time.
  2. Top-down – Recursively subdivide the data into subnodes.
     - *Really* easy to implement.
     - Doesn't result in the best tree.


© Copyright Ian D. Romanick 2007

# Building a BVH

- Three common strategies for building trees.
  1. Insertion – Build the hierarchy incrementally by adding one element at a time.
  2. Top-down – Recursively subdivide the data into subnodes.
     - *Really* easy to implement.
     - Doesn't result in the best tree.
  3. Bottom-up – Group leaf nodes together repeatedly until there's only one node left.
     - Most complicated to implement.
     - Results in better trees than top-down.


© Copyright Ian D. Romanick 2007

# Top-down

```
BVHNode *build_BVH(Entity *e, int num_e)
{
    BoundingVolume *bv = new BoundingVolume(e, num_e);
    BVHNode *node = new BVHNode(bv);

    if (num_entity < threshold) {
        node->is_leaf = true;
    } else {
        int first_half_count = divide_entities(e, num_e);
        node->child[0] = build_BVH(& e[0],
            first_half_count);
        node->child[1] = build_BVH(& e[first_half_count],
            num_e - first_half_count);
    }

    return node;
}
```


© Copyright Ian D. Romanick 2007

# *Top-down Partitioning*

- The lynch pin in the whole deal is `divide_entities`.
  - As coded, assumes each entity will be in exactly one set.
  - *Not* the only strategy.
- How do we decide where to divide the set?


© Copyright Ian D. Romanick 2007

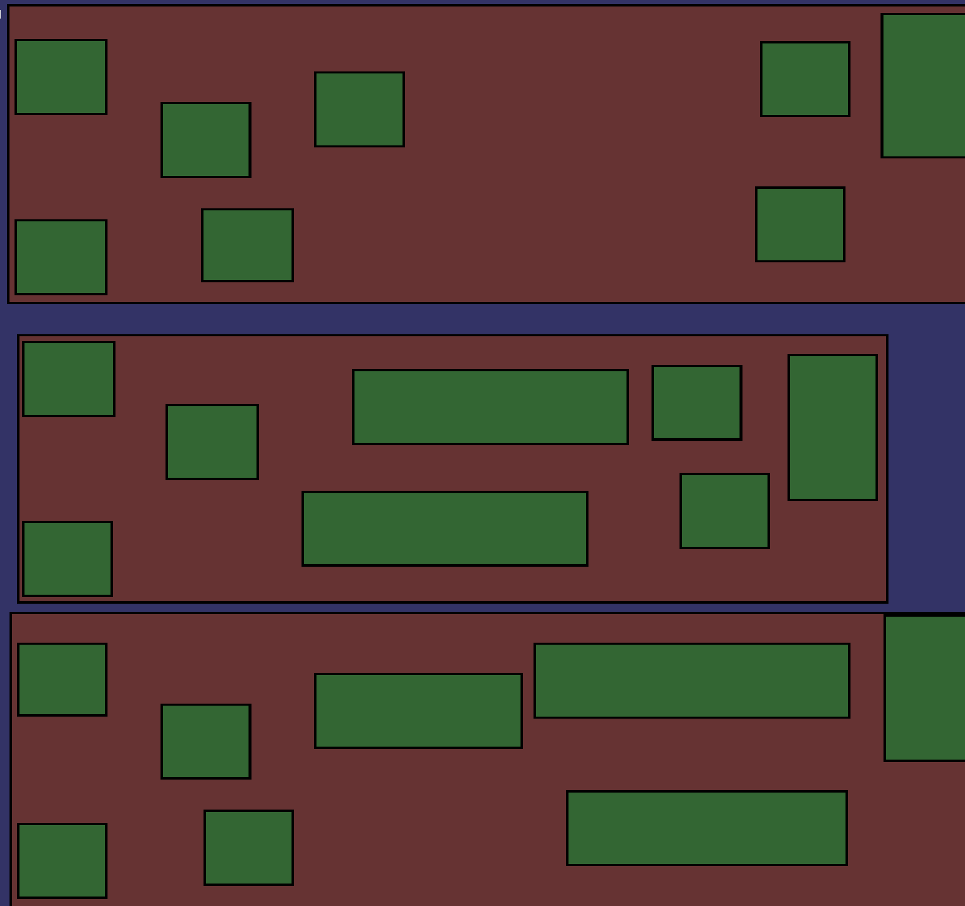
# Top-down Partition Strategies

- “Median-cut” is a common strategy.
  - Select an axis.
    - Long axis of the OBB, AABB, k-DOP, etc. is a common choice.
  - Project all entities onto this axis and sort by position.
  - The first half is one subnode, and the second half is the other subnode.


© Copyright Ian D. Romanick 2007

# Median Cut

- Median cut is easy to implement, but has problems:




© Copyright Ian D. Romanick 2007

# *Other Partitioning Strategies*

- Other heuristics:
  - Minimize sum of volumes
  - Minimize largest volume
  - Minimize intersection volume
  - Maximize child node separation
- In reality, no one heuristic is perfect.
  - Implement a primary and adjust if the next heuristic scores poorly.
  - Repeat for all heuristics or until a heuristic passes without adjustment.


© Copyright Ian D. Romanick 2007

# *Partitioning Axis Selection*

- Infinite number of possible partitioning axes.
  - Like the problem of selecting the basis for OBB.


© Copyright Ian D. Romanick 2007

# *Partitioning Axis Selection*

- Infinite number of possible partitioning axes.
  - Like the problem of selecting the basis for OBB.
- Aligned axes of BV type.
  - Local X, Y, Z axes, k-DOP axes, etc


© Copyright Ian D. Romanick 2007

# Partitioning Axis Selection

- Infinite number of possible partitioning axes.
  - Like the problem of selecting the basis for OBB.
- Aligned axes of BV type.
  - Local X, Y, Z axes, k-DOP axes, etc
- Axes of parent BV.
  - Create OBB of objects in parent, use these axes.


© Copyright Ian D. Romanick 2007

# *Partitioning Axis Selection*

- Infinite number of possible partitioning axes.
  - Like the problem of selecting the basis for OBB.
- Aligned axes of BV type.
  - Local X, Y, Z axes, k-DOP axes, etc
- Axes of parent BV.
  - Create OBB of objects in parent, use these axes.
- Axis through most distant points.
  - Approximate with most separated points on AABB.


© Copyright Ian D. Romanick 2007

# Partitioning Axis Selection

- Infinite number of possible partitioning axes.
  - Like the problem of selecting the basis for OBB.
- Aligned axes of BV type.
  - Local X, Y, Z axes, k-DOP axes, etc
- Axes of parent BV.
  - Create OBB of objects in parent, use these axes.
- Axis through most distant points.
  - Approximate with most separated points on AABB.
- Axis of greatest variance.


© Copyright Ian D. Romanick 2007

# *Split Point*

- Once an axis is selected, how is a split point on the axis selected?


© Copyright Ian D. Romanick 2007

# *Split Point*

- Once an axis is selected, how is a split point on the axis selected?
- Several common ways:
  - Median of projected object centroids.
  - Mean of projected object centroids.
  - Median of projected BV extents.
  - Pick best of *n* evenly spaced points along axis.


© Copyright Ian D. Romanick 2007

# Bottom-up

- More complex to implement.
- Slower.
- But usually results in better BHVs.
  - If it runs as a pre-process, it probably doesn't matter that it's slower.

 © Copyright Ian D. Romanick 2007

# *Bottom-up*

- Create a BV for each object.
  - Store these BVs in an “active” list.
- Select 2 or more BVs to merge.
  - Remove old BVs from active list.
  - Add new parent BV to active list.
- Lather, rinse, repeat until only one BV remains in active list.
  - This is the root of the BVH.


© Copyright Ian D. Romanick 2007

# *Merging Strategy*

- The lynch pin here is the method used to select nodes to merge.


© Copyright Ian D. Romanick 2007

# Merging Strategy

- The lynch pin here is the method used to select nodes to merge.
- Brute force: find pair of nodes in active list that merge to form least-volume BV.
  - $O(n^2)$ for the search, must be repeated (n-1) times results in $O(n^3)$.  Ouch.
  - Heuristic other than least-volume can be used.


© Copyright Ian D. Romanick 2007

# Improved Merging Strategy

- Use brute force as basis of improved method.
  - For each node, calculate the best node for it to pair with.
    - Store both nodes and the resulting volume in a priority queue.
  - Loop, removing head node from queue.
    - Validate stored size.
      - May have changed if either node was previously removed.
    - If size still smallest, calculate pairing for new node, insert in queue.
    - If size not still smallest, re-insert in queue.


© Copyright Ian D. Romanick 2007

# Insertion

- Find location to insert node with least cost.
  - Cost metric is typically along the lines of volume added to BV and all parent BVs.
  - Large objects will be inserted near the top of the tree, small objects will be inserted near the bottom.
  - Far away objects will be inserted near the top of the tree.


© Copyright Ian D. Romanick 2007

# *Insertion Strategies*

1. Pick the child node with the least insertion cost, recurse on that child.
   - Search cost is O(lg n), with n searches.  Total cost is O(n lg n).
2. Use best-first search of entire tree.
   - Search cost is worst case O(n), with n searches. Worst case cost is $O(n^2)$, O(n lg n) typical case.
   - *May* be more expensive.
   - Results in better tree.
     - Uses global information instead of just local information.


© Copyright Ian D. Romanick 2007

# Goldsmith-Salmon Incremental Method

- Developed for ray tracing.
- Cost of tree is proportional to surface area of all nodes.
- Insert primitive in subtree with least added cost.
  - Object and leaf node are paired under new node: $\Delta C = 2 \times$ Area(new node).
  - Object added as child of existing node: $\Delta C = k \times$ (Area(new node) – Area(old node)) + Area(new node).
  - Object added lower in tree: $\Delta C = k \times$ (Area(new node) – Area(old node))


© Copyright Ian D. Romanick 2007

# *Break*


© Copyright Ian D. Romanick 2007

# *Traversal*

- Same traversal orders as “regular” trees:


© Copyright Ian D. Romanick 2007

# *Traversal*

- Same traversal orders as “regular” trees:
  - Depth first – uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring.


© Copyright Ian D. Romanick 2007

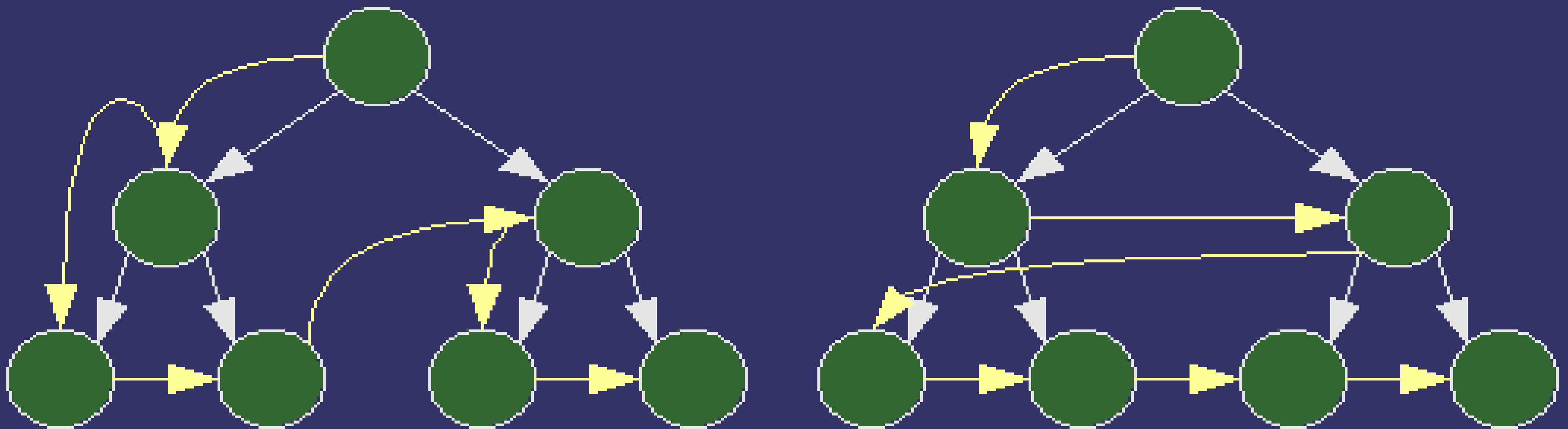
# Traversal

- Same traversal orders as “regular” trees:
  - Depth first – uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring.
  - Breadth first – Begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

Definitions from wikipedia.org.


© Copyright Ian D. Romanick 2007

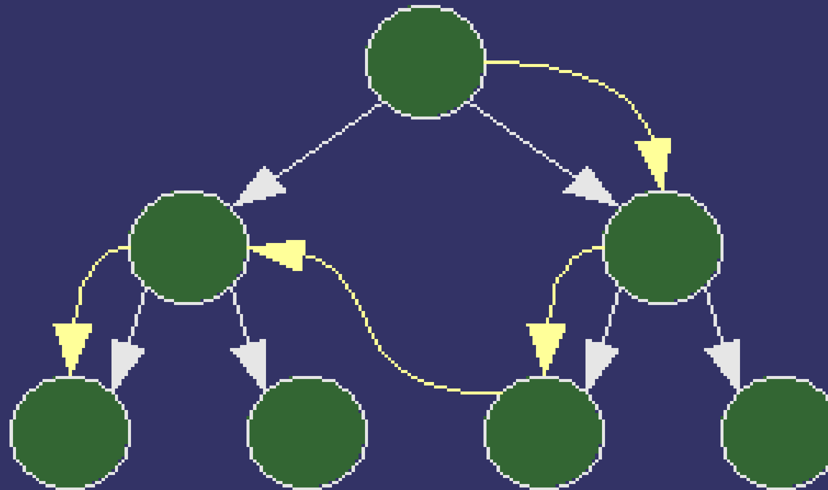
# *Examples*




© Copyright Ian D. Romanick 2007

# *Informed Traversal*

- Best-first search – like breadth-first, but explore the “most promising” unexplored node first.




© Copyright Ian D. Romanick 2007

# Best-First Search

```
BVNode *BVNode::search(BoundingVolume *bv)
{
    BVPriorityQueue q;

    q.insert(this, bv);
    do {
        BVNode *curr = q.dequeue();
        if (curr.is_leaf()) {
            if (curr.test_primitives(bv))
                return curr;
        } else {
            q.insert(curr.child[0], bv);
            q.insert(curr.child[1], bv);
        }
    } while (!q.empty());

    return NULL;
}
```


© Copyright Ian D. Romanick 2007

# BVH / BVH Intersection

- Search is different because we are comparing two trees.


© Copyright Ian D. Romanick 2007

# Basic Recursive Version

```
CollisionResult *BVHNode::collision(BVHNode *other)
{
    if (!this->bv.intersect(other->bv))
        return NULL;

    if (is_leaf() && other->is_leaf()) {
        return collide_primitives(other);
    } else {
        if (this->descend(other)) {
            result = child[0]->collision(other);
            if (result == NULL)
                result = child[1]->collision(other);
        } else {
            result = other->child[0]->collision(this);
            if (result == NULL)
                result = other->child[1]->collision(this);
        }
        return result;
    }
}
```


© Copyright Ian D. Romanick 2007

# *Basic Recursive Version (cont.)*

- `BVHNode::descend` is the key.
  - Making good decisions about which BVH to descend can make a big difference in performance.
  - Should be very simple.
    - Descend the non-leaf node.
    - Descend the non-leaf node with largest volume.
    - Descend the non-leaf node with fewest subnodes.
    - Etc.
- Recursion makes the routine easy to understand but *very* inefficient.


© Copyright Ian D. Romanick 2007

# Simultaneous Recursive Version

- We really want to descend both trees simultaneously.
- Doing so can eliminate some tests.
  - In a worst-case tree, the simultaneous version will perform 2/3$^{rds}$ as many tests.


© Copyright Ian D. Romanick 2007

# Simultaneous Recursive Version

```
CollisionResult *BVHNode::collision(BVHNode *other)
{
    if (!this->bv.intersect(other->bv)) return NULL;

    if (is_leaf()) {
        if (other->is_leaf()) {
            return collide_primitives(other);
        } else {
            result = other->child[0]->collision(this);
            if (result == NULL) result = other->child[1]->collision(this);
        }
    } else {
        if (other->is_leaf()) {
            result = child[0]->collision(other);
            if (result == NULL) result = child[1]->collision(other);
        } else {
            result = child[0]->collision(other->child[0]);
            if (result == NULL) result = child[0]->collision(other->child[1]);
            if (result == NULL) result = child[1]->collision(other->child[0]);
            if (result == NULL) result = child[1]->collision(other->child[1]);
        }
        return result;
    }
}
```


© Copyright Ian D. Romanick 2007

# *Break*


© Copyright Ian D. Romanick 2007

# OBB Trees

- Construct top-level OBB using PCA.
- Split along the longest axis.
  - Split at mean of the object vertexes projected onto the split axis.
  - If a primitive straddles the split, put it in the subnode containing its centroid.
  - If that split fails to reduces the number of objects in the child nodes or create two non-empty nodes, select one of the other axes.


© Copyright Ian D. Romanick 2007

# *OBBs Evaluation*

- OBBs make a good fit.
  - If O(m) OBBs would be required, typically $O(m^2)$ spheres or AABBs would be required.
- $N_v$ and $N_p$ tend to me smaller for OBB trees than AABB or sphere trees.
- $C_v$ is about an order of magnitude slower.


© Copyright Ian D. Romanick 2007

# AABB Trees

- Like OBB tree, construct top-down.
- Split along longest axis.
  - Split at the midpoint of the axis.
  - If a primitive straddles the split, put it in the subnode containing its centroid.
  - If all primitives end up in one subnode, re-split by primitive medians.
- Some speed up from only performing 6 of the 15 rotated AABB tests.
  - More false positives but still a win.


© Copyright Ian D. Romanick 2007

# BoxTrees

- Similar to AABB trees.
  - Subdivide each node to directly form the new AABBs.
  - Primitives that straddle the split are put in *both* subnodes.
    - Try all three axes to find the one that causes the fewest straddles.
  - For testing, rotate AABBs to form OBBs.
    - All nodes share the same coordinate space, so the same 15 OBB testing axes can be reused.


© Copyright Ian D. Romanick 2007

# *Sphere Trees via Octree*

- Build octree:
  - Create axis-aligned bounding cube.
  - Recursively subdivide cube into 8 subcubes.
  - Prune away empty cubes.
  - Halt when predefined depth reached or subdivision fails to separate polygons.
- Create sphere centered at each cube.


© Copyright Ian D. Romanick 2007

# *k-DOP Trees*

- Similar to AABB & OBB methods.
- Most research seems to indicate:
  - Split along axis that minimizes the sum of volumes.
  - Various numbers of axes work in different situations.
    - Some say 3 (AABB), some 9 (18-DOP), some 12 (24-DOP).
  - Required storage is *high*.
    - 170k triangles required 69MiB


© Copyright Ian D. Romanick 2007

# Other

- Possible to combine multiple BVs:
  - k-DOPs near the root, AABBs near the leaves.
  - AABBs at all levels except leaves, k-DOPs or spheres at the leaves.
  - Etc.
- Little research has been done in this area.
  - Might be a good way for an undergrad to get published. ;)


© Copyright Ian D. Romanick 2007

# *Next week...*

- Finish BVHs
  - Merging
  - Data layout tuning
- Polygon representation
- Convex hulls
- Assignment #2 due.
- Assignment #3 assigned.


© Copyright Ian D. Romanick 2007

# Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.


© Copyright Ian D. Romanick 2007