Plinto
// Declare two variables that refer to a pawns.
var pawn P, Q;

// Here is a function that makes use of P.
// It displays some information about P.
function MyFunction()
{
    // Set P's enemy to Q.
    P.Enemy = Q;

    // Tell P to play his running animation.
    P.PlayRunning();
}

Potentially Interesting Objects

- Objects register queries
- Often restricted to clients
- Often implicit
- Server returns relevant objects
Discovery Today

- Almost universally distance based
- “Let me know about everything within x meters”
- Games use imposters
- Only simple for centrally controlled content
Shortcoming

- Distance doesn’t capture what’s important
Solid Angle

- Object could be far away, but big
- Object could be nearby, but small

- We care about apparent size or solid angle

- “Let me know about all objects that are bigger than n pixels on my screen”
Query Comparison

- For same # of objects, better quality
- For same quality, fewer objects
Non-visual?

- Discovery, not communication
- Reflects real “discovery”
Outline

• Motivation
• Previous Work
• BVH++
• Continuous Queries and Cuts
• Distributed Queries
Implementation

• How do we efficiently evaluate these queries?
• All objects with solid angle < x
• Standing query - continuous stream of updates
• Not much guidance from CG literature
Implementation

• Doesn’t seem too different from distance query
• Just a slightly different constraint
• Maybe we can reuse existing techniques
Bounding Volume Hierarchy
Bounding Volume Hierarchy
Bounding Volume Hierarchy
Bounding Volume Hierarchy
Why does this work?

• Conservative check on entire group
• If boundary of bounding volume is out of range, contained objects must be too
• Quickly cull large subtrees
Outline

- Motivation
- Previous Work
- BVH++
- Continuous Queries and Cuts
- Distributed Queries
Solid Angle Queries

- Use same BVH
- Bounding volumes must have larger solid angle
- Recurse if bounding volume satisfies constraint
Results

• 50,000 objects (1m) randomly placed over 1 km$^2$
• 50 queriers
• Branching factor: 10
• Solid angle comparisons:
  • Brute force: 50,000
  • BVH: 60,466 (No culling!)
What went wrong?

• **Too** conservative
What went wrong?

- Too conservative
BVH++

- Track largest object in subtree
- Check against virtual worst-case object
- As close as possible
- As big as possible
BVH++

- Track largest object in subtree
- Check against virtual worst-case object
- As close as possible
- As big as possible
Results

- Solid angle comparisons:
  - Brute force: 50,000
  - BVH: 19,631
Branching Factor

- Higher branching factor potentially allows better culling
Branching Factor

- Future evaluation
- Some operations on tree are linear in branching factor
- Maintenance cost vs. traversal cost
- Timing based - depends on optimized implementation
Outline

• Motivation
• Previous Work
• BVH++
• Continuous Queries and Cuts
• Distributed Queries
Continuous Queries

• Good approach for one-off queries
• Could just reevaluate periodically
  • How frequently?
• Wasted effort - check same nodes repeatedly, even if they don’t change
Query Cuts

- Maintain “cut” representing stopping points of evaluation
- Leaves
- or internal nodes which do not satisfy constraint
Query Cuts
**Query Cuts**

- **Cut**: ordered list of **CutNodes**
- **CutNode**: stores references to parent Cut, BVH node
- **BVH Node**: stores bounds, max object size, unordered set of **CutNodes**
Query Cut Updates

- Query change (position, solid angle) causes results to change
- Traverse cut, pushing it up or down
  - Sequence of all children failing test pushes cut up
  - Cut pushed down as usual
Query Cut Updates

- Sequence of all children failing test pushes cut up
Query Cut Updates

- Sequence of all children failing test pushes cut up
Query Cut Updates

• Sequence of all children failing test pushes cut up
Query Cut Updates

- Cut pushed down as usual
Query Cut Updates

- Cut pushed down as usual
Query Cut Updates

- Cut pushed down as usual
Query Cut Updates

- Due to object change (position, size)
- or addition, removal
- Update BVH
  - Reevaluate cuts, pushing up or down
  - BVH nodes store pointers to cut nodes to quickly determine which nodes in which cuts require adjustment
Query Cut Updates

- Some BVH updates cause major adjustments to tree
- Merge/split
- ... and splits can even percolate all the way up to the root
- insertion where all ancestors are full
Query Cut Updates

- Current approach
  - “Lift” all cuts to stable point
  - Mark for retraversal and update
  - Simple but expensive
Query Cut Updates

• Current approach
• “Lift” all cuts to stable point
• Mark for retraversal and update
• Simple but expensive
Query Cut Updates

- Current approach
  - “Lift” all cuts to stable point
  - Mark for retraversal and update
  - Simple but expensive
Query Cut Updates

- Current approach
- “Lift” all cuts to stable point
- Mark for retraversal and update
- Simple but expensive
Query Cut Updates

• Any approach that:
  • ensures the cut remains **structurally valid**
  • marks the cut for re-traversal using update procedure

• will work

• More complicated solution might keep cut closer to final cut, making update cheaper
Considerations

• Updating queries:
  • due to object motion can be expensive
  • due to query motion almost always better than re-evaluating one-off query

• Polling might be cheaper for frequent movement, especially of objects
  • or when reevaluating position due to velocity frequently
Considerations

- Trees for static vs. dynamic objects
- Also reduces maintenance cost by isolating frequently adjusted nodes in their own tree
- Static tree: cut-based, event-driven queries
- Dynamic tree: poll-based queries
Reporting Results

- Deltas on result set (additions, removals)
- Sort by solid angle
- Prioritize additions

- But probably not the dominant factor in quality over time
- Downloading content likely much slower
Outline

• Motivation
• Previous Work
• BVH++
• Continuous Queries and Cuts
• Distributed Queries
Distributed

- Each space server starts out only with local objects
Distributed

- Space servers aggregate queries and send one query to each other space server

SS 1

Local

A

B

Q(1 -> 2)

SS 2

Local

C

D

Q(2 -> 1)
Distributed

• Results from these queries used to maintain global tree
Distributed

- Global tree answers queries from connected objects
What about updates?

• Loc takes care of updates
• PlntO triggers subscriptions in Loc automatically
• For other servers and connected objects
Server Discovery

• How do servers know to query each other?

• PlntO service

• Collects region, max object size, and query from space servers

• Answers queries about which servers have objects that might satisfy query

• Same data structure works!
PlntO Conceptual Model

PlntO Service

Space Servers

SS I

A C D B

SS N

E F G H

Objects
Evaluations

- # of servers
- Very limited in range query (4 or 8 neighbors)
- Potentially all servers for solid angle
- # of objects in global tree
- Efficiency of query aggregation
- # objects returned in object query / # objects returned by server queries?
Insertion

- Recursively select best child node
- Min volume increase
Insertion

- Recursively select best child node
- Min volume increase
• Recursively select best child node
• Min volume increase
Insertion

- Split nodes as necessary
- Possibly reorders children
- May obtain new root
Insertion

- Split nodes as necessary
- Possibly reorders children
- May obtain new root
Insertion

- Split nodes as necessary
- Possibly reorders children
- May obtain new root
Deletion

- Find leaf node (hash table)
Deletion

- Find leaf node (hash table)
Deletion

- Remove
- Merge parent’s children if:
  - All children are leaf nodes
  - Fewer children than branching factor
Deletion

- Remove
- Merge parent’s children if:
  - All children are leaf nodes
  - Fewer children than branching factor
Deletion

- Remove
- Merge parent’s children if:
  - All children are leaf nodes
  - Fewer children than branching factor
Deletion

- Remove
- Merge parent’s children if:
  - All children are leaf nodes
  - Fewer children than branching factor
Deletion

- Remove
- Merge parent’s children if:
  - All children are leaf nodes
  - Fewer children than branching factor
Split

- Select two seeds for new nodes
- e.g. pair of objects (nodes) that cause worst increase in volume
- Add each object to one group
  - based again on which causes the least harm