R-trees

Adapting out-of-core techniques to modern memory architectures

Sebastian Sylvan (ssylvan@rare.co.uk)
Principal Software Engineer, Shared Technology Group, Rare
This talk

Two main themes

- A neat data structure for spatial organization: R-trees
- A strategy for finding cache-friendly alternatives to traditional algorithms

Will use the Xbox 360 for specific examples

Mostly applicable to PS3 too (I think! 😊)
The problem

- Current console architectures are... “pointer challenged”
- L2 cache miss \( \approx 600 \text{ cycles!} \)
  
  \textit{And that’s on a good day!}

- A very common reason for poor performance

- Can’t “deal with it later”
  
  Cache considerations impact data structure choices
  
  Must take it into account \textit{early}, or end up in local optimum
It’s very easy to end up in a local optimum if you don’t consider cache concerns up front and choose a data structure which has inherently poor cache behaviour (e.g. a KD-tree). You can easily end up optimizing for cache behaviour (compressing nodes, rearranging them in memory), within the constraints of a cache-hostile data structure. If cache misses are a primary concern, it’s worth trying a data structure that attacks that problem more directly.
The Solution

- Code shouldn’t be written as if memory access was cheap
  - (because it’s not)
  - Treat memory access like disk access
- Main tool at our disposal: **dcbt**
  - “Pre-fetch”, whenever possible
  - “Block fetch”, for unavoidable cache misses
    - Pay one cache miss instead of 8!
  - Opportunities for these rarely occur in the wild without intentional set up
History repeats itself

How do we write code that exposes opportunities for pre-fetching and block fetching?

Database and out-of-core people have dealt with this problem for several decades!

Their algorithms are tuned to:
- reduce number of fetches
- deal with large blocks of data at a time
  - *This is exactly what we want!*

Steal their work!
  - For spatial index structures: R-trees
R-trees

- Essentially just an AABB-tree
  - But with some specific properties that pay off...
  - ... and loads of prior work we can steal for the details
- Nodes are big fixed size blocks of child AABBs and pointers
  - AABB for a node stored in its parent
  - Makes sense given the access pattern
  - Some slackness (usually up to 50%)
  - Reduces frequency of topology changes
- Terminology: An “m-n R-tree” means all nodes (except root) has between m and n children
2-3 R-trees

2-3 R-tree as an example, in practice we want much bigger nodes (e.g. 16-32 R-trees)
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
2-3 R-trees
Main Operations on R-trees

Insert/Delete
- Saw insert. Pick leaf, split if overflowing, insert new node into parent. (keeps it balanced!)
- Delete: Remove child from node, fill hole with last child
- Strictly speaking should handle underflow
  - typically by re-inserting all remaining nodes
  - In practice, delete is often deferred; only remove when node is completely empty

Range queries (frustum, sphere, AABB etc.)
- Fairly standard:
  - Check each child’s AABB against range
  - If it intersects, recurse until we’re at a leaf
  - If it’s fully outside range, return nothing
  - If it’s fully in range, return all children with no further tests
R-trees, benefits

- Cache friendly data layout
- No rigid subdivision pattern
- High branching factor
  - shorter depth, fewer fetches, more work within each node
- Prefetching (breadth-first traversal)
  - Stack (depth first): current node can change which node comes next
  - Queue: We know which node is next – so prefetch it!
- Many children per node
  - Unroll tests to hide VMX latency
  - Hides the pre-fetch latency
R-trees, benefits

- Dynamic objects doable
  - Even if objects move, the topology remains valid
    - Just need to propagate any AABBs changes to parent
  - May eventually end up performing poorly
    - But will still be correct
  - Defer re-insertions until object has moved “a lot”
    - Adjusting AABBs is much cheaper than a re-insert
R-trees – Insertion

1. How to pick which child node to insert into
2. Seems somewhat irrelevant for performance!
   - As long as you do something relatively sensible
   - Splitting seems to have a bigger impact
3. Simple strategy:
   - Pick the child node which would be enlarged the least by putting the new object into it
R-trees – Splitting

- Happens when a node overflows
- Loads of different strategies here...
  - Linear, Quadratic and Exponential from the original R-tree paper... Simple, but perform fairly poorly
  - Many other techniques... No time to talk about them all!
- For static data, can bulk load
  - Mainly STR or Static/Packed Hilbert R-trees
  - We’re mainly concerned with dynamic R-trees so we won’t linger on this...
- Let’s come back to splitting...

Note: R*-trees and Hilbert R-trees are generally considered the best splitting techniques w.r.t. query performance for dynamic trees. cR-trees tend to do well too...
R-trees – Incremental refinement

Garcia et al.

“On Optimal Node Splitting for R-Trees”

Figured out a fast way to do optimal splitting...

…but query performance boost was small...

“Thus a second contribution is to demonstrate the near optimality of previous split heuristics”

* Ouch!

But led to a key insight: “research should focus on global rather than local optimization”

R-trees – Incremental refinement

- Non-local, incremental
- Can do this for e.g. 5% of insertions, or do many iterations whenever we have time to spare
- Can give *better* performing trees than bulk loading!
- Can reshape *any* R-tree into a well-formed one
- If we do this, we can get away with murder for insertions/splitting!
R-trees – Incremental refinement

_starts at a specific node, \( n \) (typically a leaf)_

**First, recursively optimize the parent of \( n \) (unless root)**
- Helps ensure that we get “good” siblings for the next steps

**Then try to merge \( n \) with an overlapping sibling**
- Improves space utilization

**Else, for each overlapping sibling to \( n \), take the union of its and \( n \)’s children, and then split that**
- Redistributes child nodes between siblings, improving overlap, empty space etc.
R-trees – Splitting, part deux

Due to refinement, splitting strategy isn’t *that* crucial
Even if we slightly screw up the first time through, we get
multiple chances to fix it

However, must ensure “diagonal” clusters work well
Incremental refinement algorithm generates these
situations
E.g. Ang/Tan splitting method doesn’t handle this well
(splits along coordinate axes)
R-trees – Splitting, part deux

Split using k-means clustering ($k=2$)
Standard clustering algorithm, not just for R-trees
Performs well

Simple:
Randomly pick two nodes as “means”
Repeat until means converge:
- Assign each node to the closest mean
- Compute new means for these clusters

See:
**Revisiting R-tree Construction Principles**
Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis

For more on using k-means as the splitting algorithm, though this paper splits in >2 parts, we don’t need to do that – the incremental refinement fixes up flaws better.
K-means example...
K-means example...

Pick two seed objects, A and B, at random...
K-means example...

Classify objects by closest mean...
K-means example...

Compute new means...
K-means example...

Classify objects by closest mean...
K-means example...

Compute new means...

A

B
K-means example...

- Classify objects by closest mean...

- Convergence! Done!
R-trees – Splitting, part deux

- Adapt k-means for AABBs instead of points

  Need to define what “distance” and “mean” means for AABBs

  - Let “distance to mean” be the diagonal of the minimal AABB that includes both the box and the mean
    - Gives a reasonable estimate of dissimilarity
  - Let “mean of a cluster” be the “center of gravity”
From the very sharp initial drop we should NOT conclude that merely increasing the branching factor will yield incredible performance benefits, rather we should conclude that the R-tree algorithm performs poorly at low branching factors. This makes perfect sense when you think about it, as the condition of the R-tree is largely determined by the splitting strategy’s ability to perform a “good” split of all the objects in the subtree at a given node by only dealing with the immediate children. For very low branching factors (e.g. 4) the immediate children are unlikely to provide sufficient information about the “shape” of objects under the node. Once we reach branching factors of 20-24 though, it’s much more likely that the distribution of the immediate children is a good representation of the overall distribution of objects in that subtree, which means that a good split of the immediate children is likely to be a good split of the entire sub tree under the node in question.

The main thing to take home from the graph is the large jump between the curve with no cache control, and the other curves.

Note the jump of the “All optimizaions” line when you go from 8 to 9 cache lines. That’s because if you try to issue 9 dcbt’s the CPU will stall for the first 8, meaning it can’t proceed and work the current node (hiding the fetch latency). This effectively disables pre-fetching. In the real world there may be other things putting pressure on memory, so it may be wise to use only 7 or perhaps 6 cache lines to leave some room for other threads to read form memory.
R-trees summary

- Fast, block-based, AABB trees
  With all the usual benefits of hierarchical trees
- Very cache-friendly
- SIMD friendly
- No “bulk loading” required
- Lots of prior work to exploit
A Reusable Strategy?

- Have problems with cache misses?
  Has similar been done in an out-of-core scenario?
  Steal it! Replace “disk block” with “memory block”
  - Use dcbt and dcbl28 on these memory blocks!

- R-trees: my first attempt at this
  Wanted cache-friendly spatial index
  Figured it would be block-based, but knew no details
  Deliberately looked up how database people do it

- I hope this strategy will work again in the future!
Questions?

Contact: ssylvan@rare.co.uk

Tons of papers on this, here are two overviews as a starting point:

R-trees: Theory and Applications
Manolopoulos, Nanopoulos, Padopoulos, Theodoridis; Springer 2006

Multidimensional and Metric Data Structures
H. Samet; Morgan Kaufman Publishers, 2006

The original paper:
R-Trees: a Dynamic Index Structure for Spatial Searching
A. Guttman; Proceedings ACM SIGMOD, pp. 47-57, 1984
Backup slides!
Future work

- Compress nodes, reduce memory/cache usage
  - May lead to too high branching factor?
  - May be slower due to decompression into VMX vectors...
  - Node size can be smaller – just fetch several nodes at a time. This is an advantage over out-of-core techniques.
  - We only need our blocks to be piecewise contiguous.
- Other BVs
  - Maybe spheres.. Maybe 8-DOP... Maybe have different BVs at leafs vs nodes?
- Implement on GPU
  - Very small cache. Even more dependent on good cache behaviour
  - Each ALU lane could do one AABB, we may not need hacks like “packet tracing” to get good SIMD performance
- Spatial joins
  - Usually not considered a standard tool for game applications
  - But this is bread and butter for DB people so it’s natural to come across it if you’re reading R-tree literature
  - Makes many $O(n \log n)$ operations $O(\log^2 n)$
- More benchmarks/comparisons (e.g. hgrid etc.?)
The function `BlockFetch` inserts the required number of `dcbt` calls for a given value through some recursive template metaprogramming. Same trick works for `dcbz128`. 

```cpp
template<unsigned int N>
__forceinline void BlockFetchHelper(void* p)
{
    BlockFetchHelper<N-1>(p);
    __dcbt( CACHE_LINE*(N-1), p );
}

template<>
__forceinline void BlockFetchHelper<0>(void* p) {}

template< typename T >
__forceinline void BlockFetch( T* p )
{
    BlockFetchHelper<sizeof(T)/CACHE_LINE>(p);
}
```