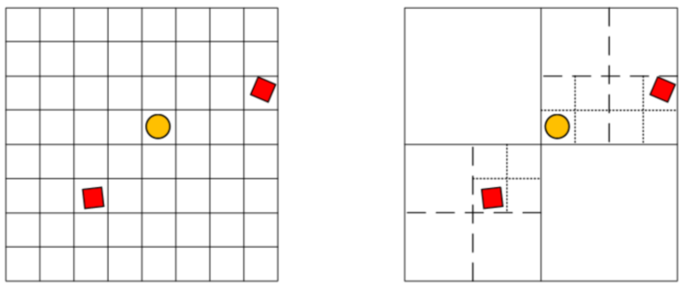
Quad/Oct-Trees

jodavis42@gmail.com

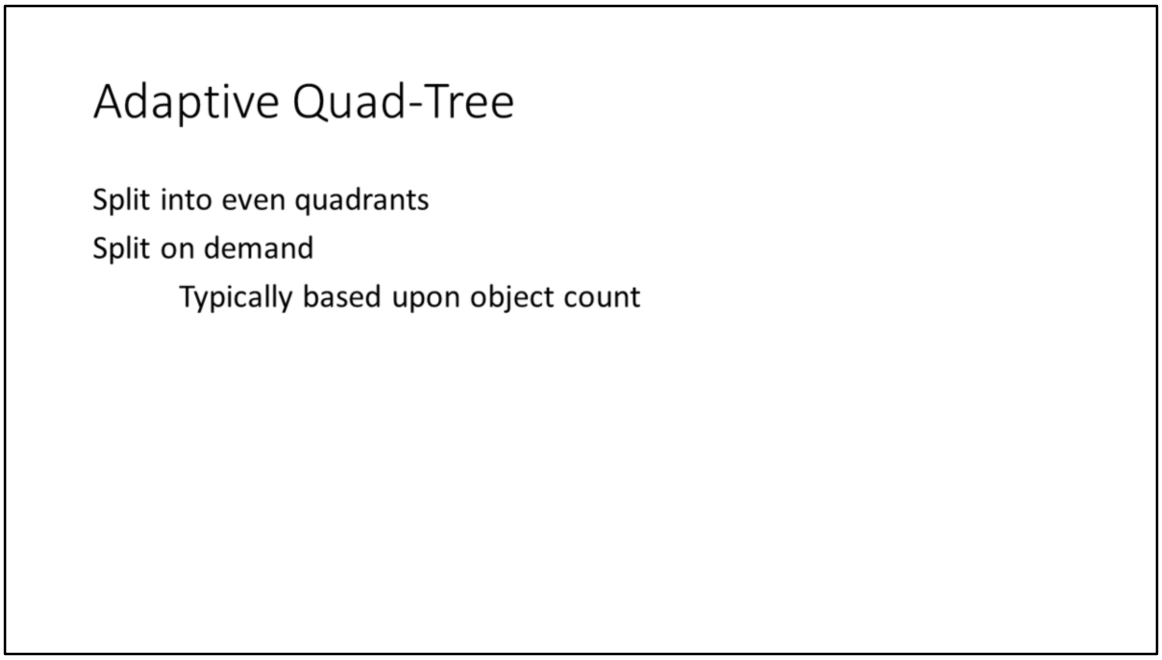
Quad-Trees

Attempt to fix uniform grid size issues



One of the root problems with uniform grids (although we addressed most of them with h-grids) is that they contain a lot of detail where we don’t need them. In particular this was the case with very spread out objects. Another issue with them was being unable to store varying sized objects easily.

One alternative to this is to use a quad or oct-tree. This lecture will mostly just talk about quad-trees as they’re much easier to draw, but everything should extend easily to oct-trees. A quad-tree sub-divides space into 4 (typically equal) sections recursively.



Adaptive Quad-Tree

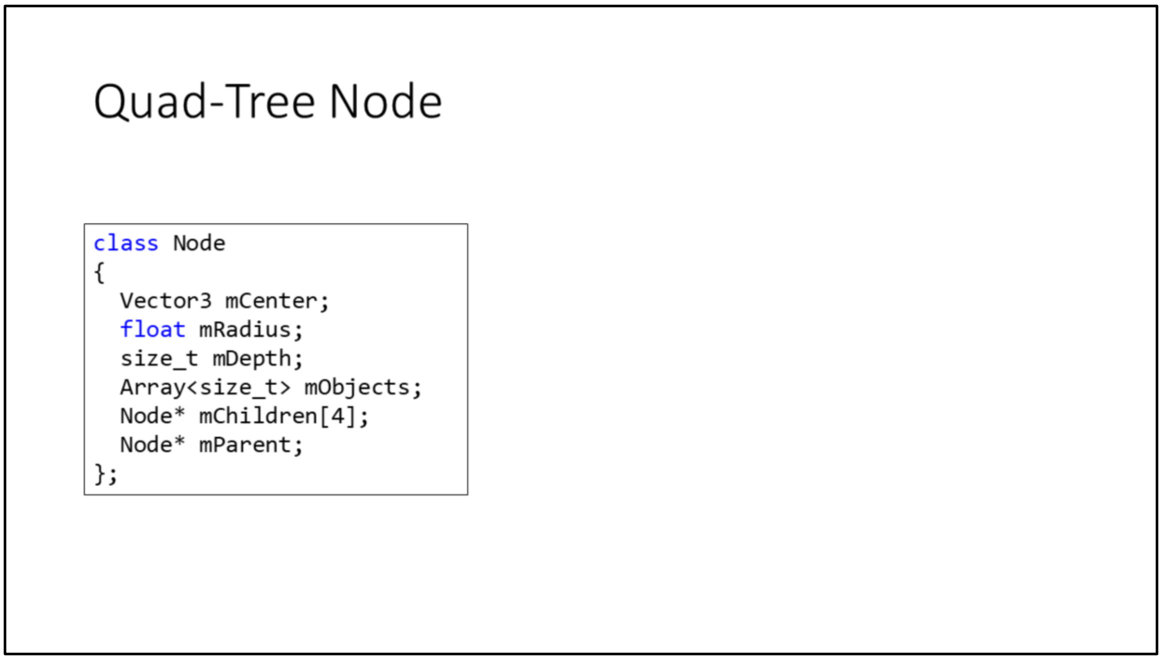
Split into even quadrants

Split on demand

    Typically based upon object count

Since we only subdivide when needed, this is an adaptive quad-tree. This lecture will focus on dynamic objects and in particular the quad-tree being online, meaning we don’t have all of the data up-front. Because of this quad-trees typically subdivide into equal quadrants as we have no good knowledge of where else to make the split and it would change over time.

Also, since this is an adaptive tree we subdivide on demand. Typically subdivision is controlled by a max object count. More on subdividing will be addressed when discussing object insertion.

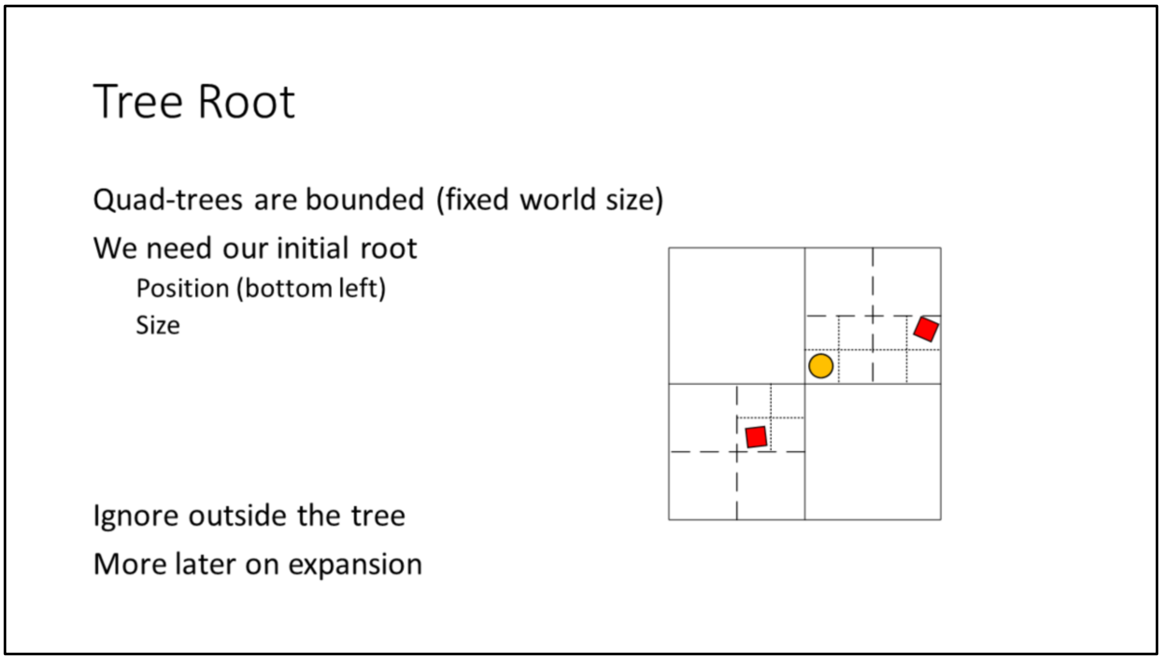


Quad-Tree Node

```
class Node
{
  Vector3 mCenter;
  float mRadius;
  size_t mDepth;
  Array<size_t> mObjects;
  Node* mChildren[4];
  Node* mParent;
};
```

Real quick we need to address what a node needs to store. Much of this data can be implicitly computed but for simplicity I’m showing a fairly complete initial node.

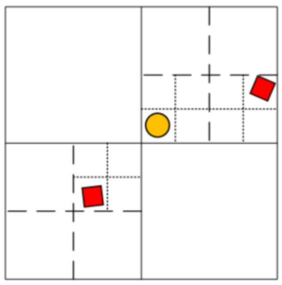
As each node is basically a box, we need to store it’s center and size (radius). To make certain computations easier storing the depth can also be nice. Also we need to store what objects are contained in this cell. Finally, as this is a tree we need to store the parent and children so we can traverse up and down the tree.



Tree Root

Quad-trees are bounded (fixed world size)

We need our initial root
- Position (bottom left)
- Size



Ignore outside the tree

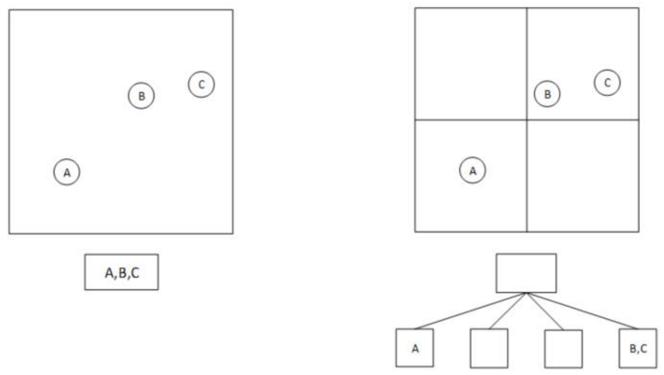
More later on expansion

To start with, we'll discuss the quad tree as being a rooted tree with a fixed bounds. To create the first node that everything gets inserted into we need to know the root node's position and size. For a user, specifying the node center is more convenient but for our internal math calculations bottom left is easier.

If our game has areas of a fixed size then this is not a problem, however if our world is unbounded then we should either use a different spatial partition or allow the tree to grow. More details on this at the end.

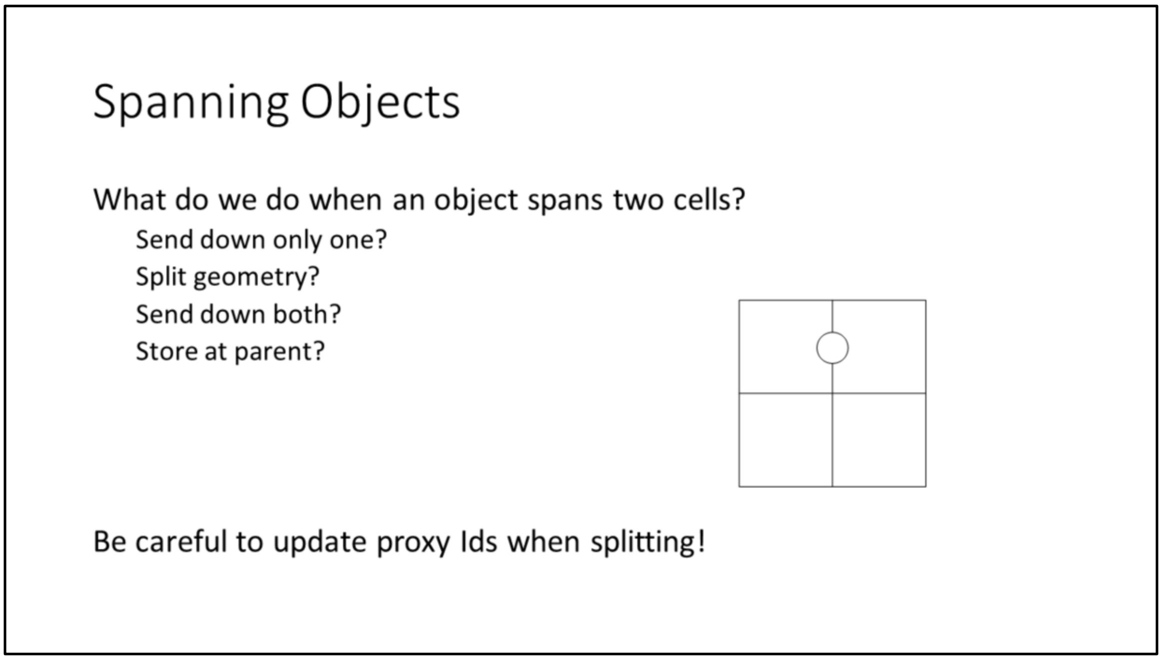
Quad-Tree: Insertion

Insert into root until split is needed



When we first insert into the tree our objects will just go into the object list on the root node. As this is an adaptive tree we need to choose some point at which to split the node. Object count is the most typical way to do this. The main idea with object count is to limit how big n^2 can grow to within a cell. That being said we still typically need a termination condition of a max depth to prevent infinite recursion.

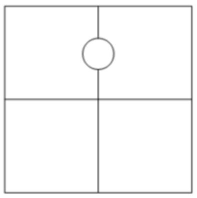
When we do split we need to pass each object in the current node down to the new children. Obviously if a node fully contains the object's bounding volume then we can just pass it down to the child.



Spanning Objects

What do we do when an object spans two cells?

- Send down only one?
- Split geometry?
- Send down both?
- Store at parent?



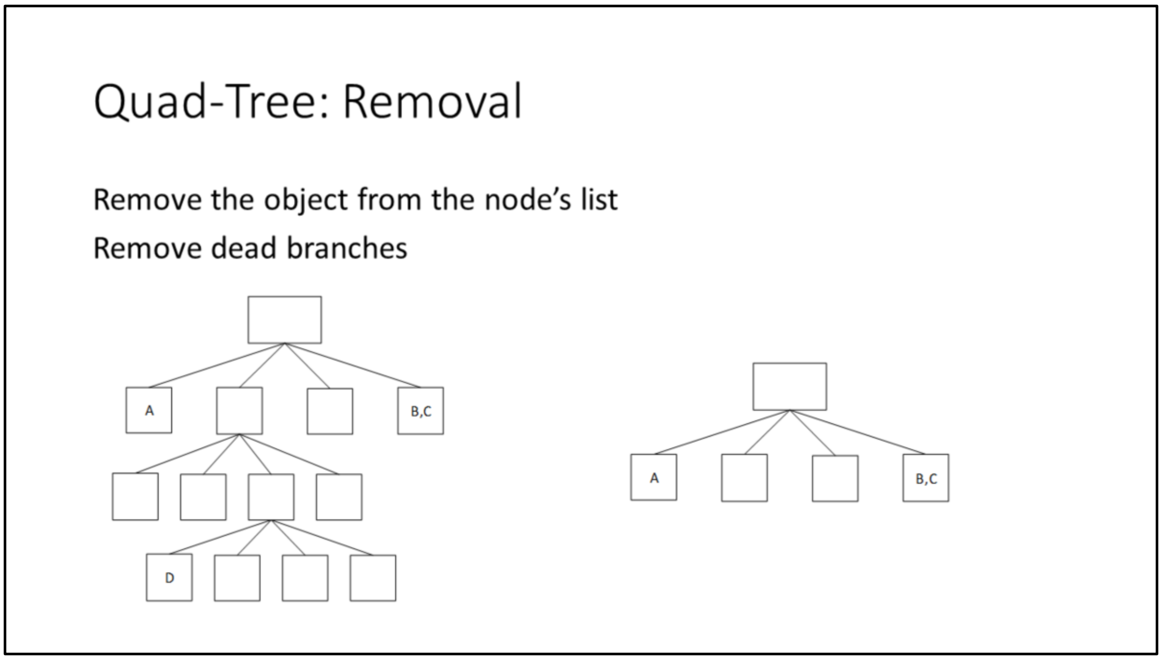
Be careful to update proxy Ids when splitting!

Unfortunately we have to deal with objects that span more than one cell. We have a few options:

Only send down 1 cell based on object position
Split the object and send each piece down the respective side
Insert into all overlapping cells
Store at the parent cell

Inserting into only 1 cell isn't really a good idea because we break the fundamental rule that a bounding volume needs to contain the object. Splitting the object works, but is better suited for static trees. Inserting into all overlapping cells has a similar problem. They both would require a lot of extra work during insertion and removals and have to be careful to avoid duplicate pairs. This just leaves the choice of keeping the object at the current node's level.

One extra thing we have to be careful of here during insertion is to not let the proxy id get screwed up. If we use a direct reference to the node the object is in then when we subdivide we have to update the id. We may not have access to each object's proxy at some random point in time so I recommend storing a box array (similar to what we did with h-grids).



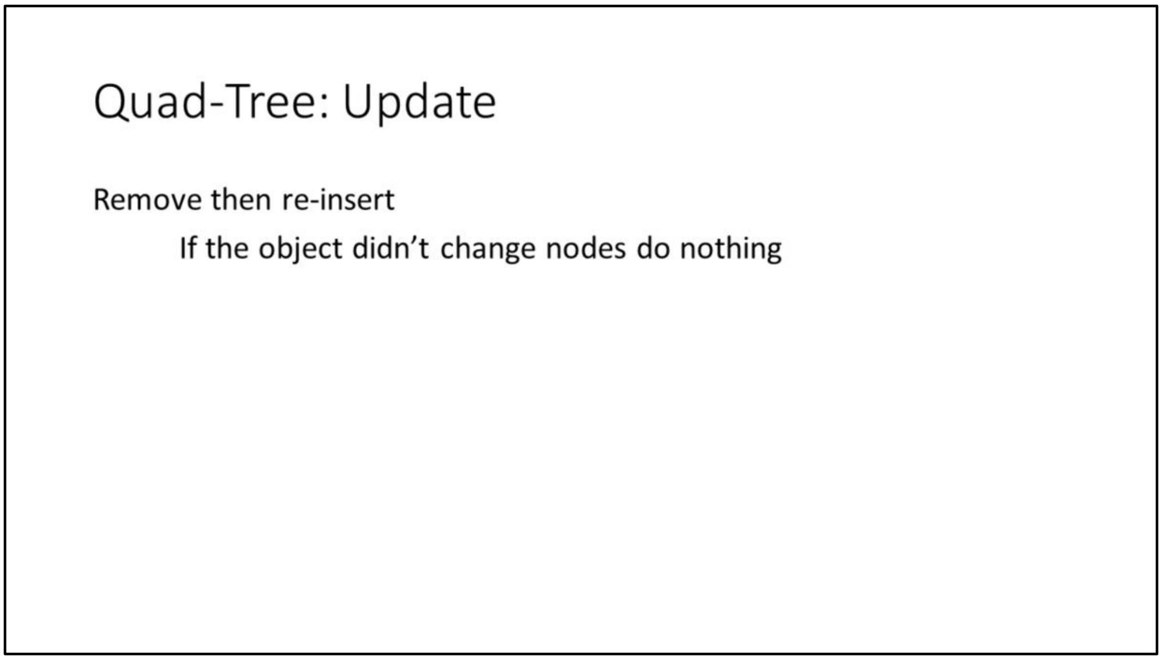
Quad-Tree: Removal

Remove the object from the node's list

Remove dead branches

Object removal is fairly straightforward: find the node the object is in and remove the object. Using the proxy system this should be an instant look-up, but even without that we should be able to walk an object's aabb to find the cell it was in.

The only extra thing we want to do is clean-up any nodes that don't need to exist anymore. We have to be a little careful though as a node with zero objects in it can't necessarily be destroyed. We have to make sure that the total (recursive) count of the node is zero before removing it. Similarly we need to traverse back up the tree trying to destroy all of our parent nodes.

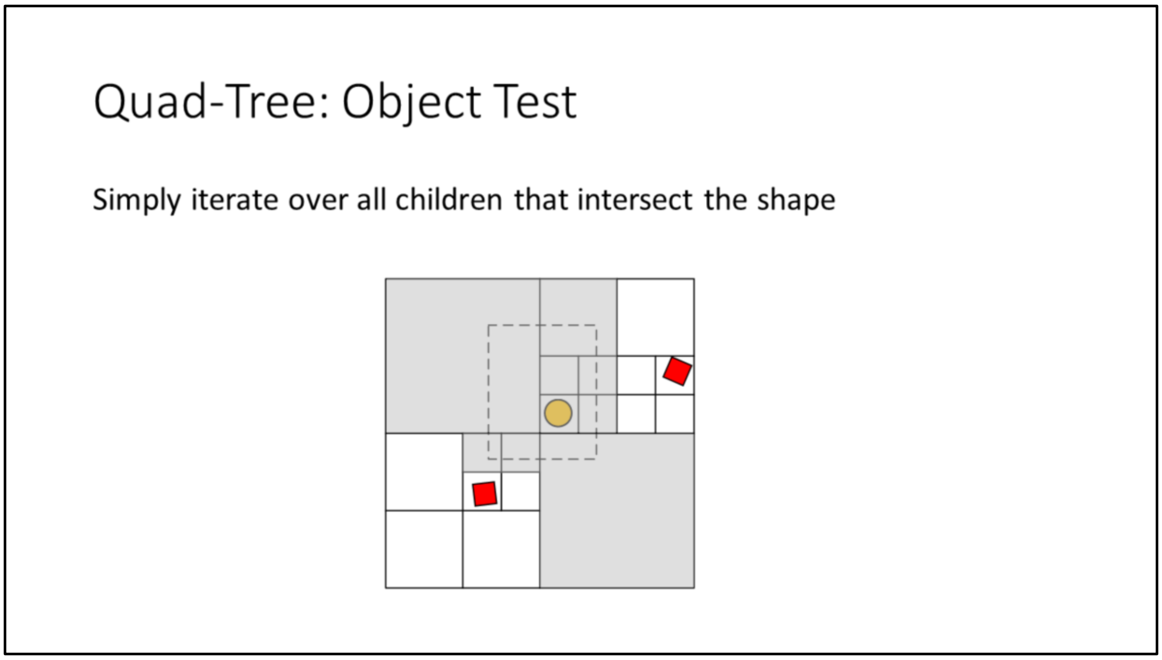


Quad-Tree: Update

Remove then re-insert

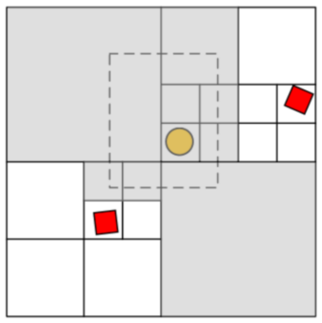
- If the object didn’t change nodes do nothing

The simplest method of performing an object update in a quad-tree is to just remove it then re-insert it. Doing this could cause several wasted calculations due to node deletion and re-creation though. A minor optimization that can be made is to not do anything if the old and new cell would be the same.



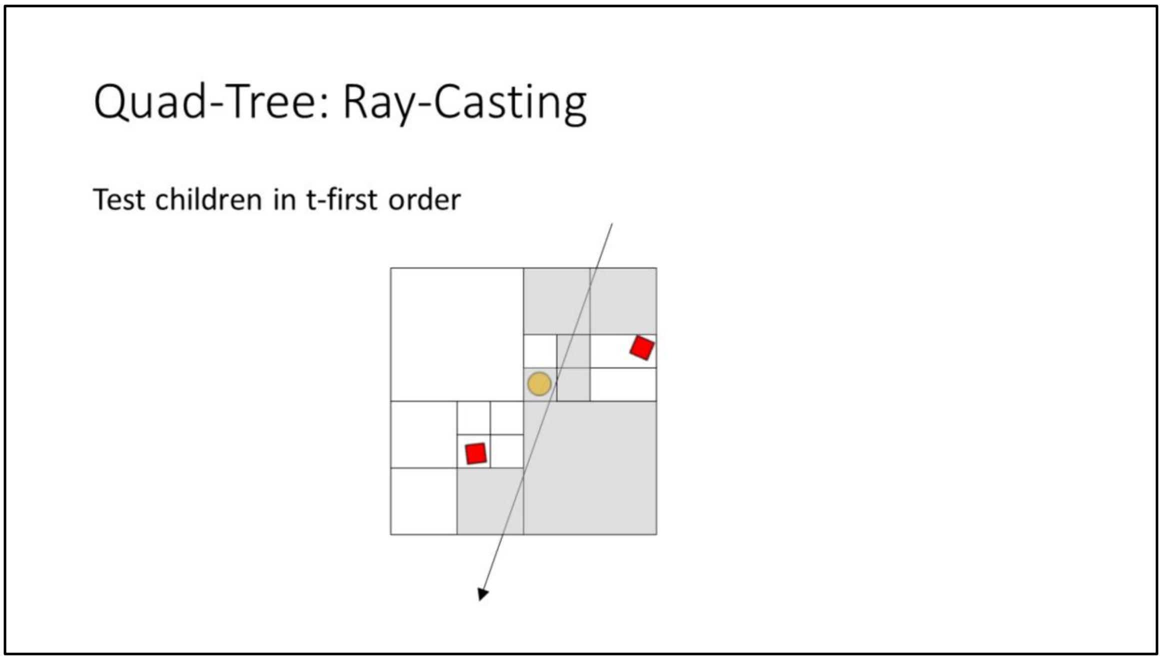
Quad-Tree: Object Test

Simply iterate over all children that intersect the shape



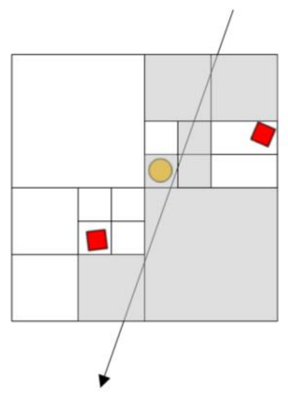
A generic object test is straightforward: simple iterate recursively over all child nodes that contain the cast shape.

Do note that we have to check all objects contained within the node, which wasn't easy to show in this picture.



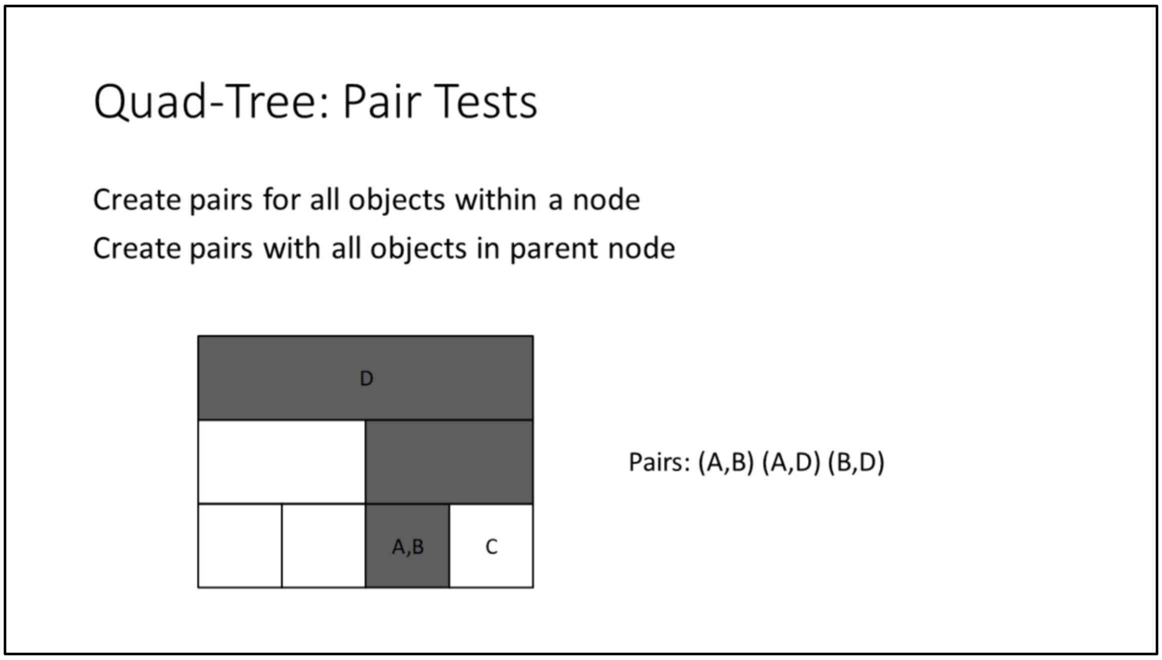
Quad-Tree: Ray-Casting

Test children in t-first order



A ray-cast through an adaptive quad-tree is easy. At each step check all children nodes and iterate through them in order of first t-value. Since the cells don't overlap in any way we can actually be guaranteed that we'll hit objects in t-first order (ignoring their ordering within the same cell) which can make efficient casts for just 1 object.

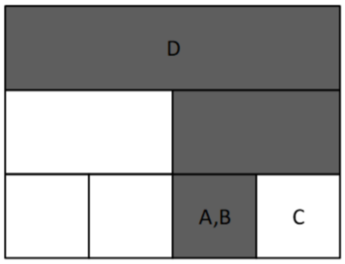
Do take extra care to handle rays starting inside a cell (the t-value is 0).



Quad-Tree: Pair Tests

Create pairs for all objects within a node

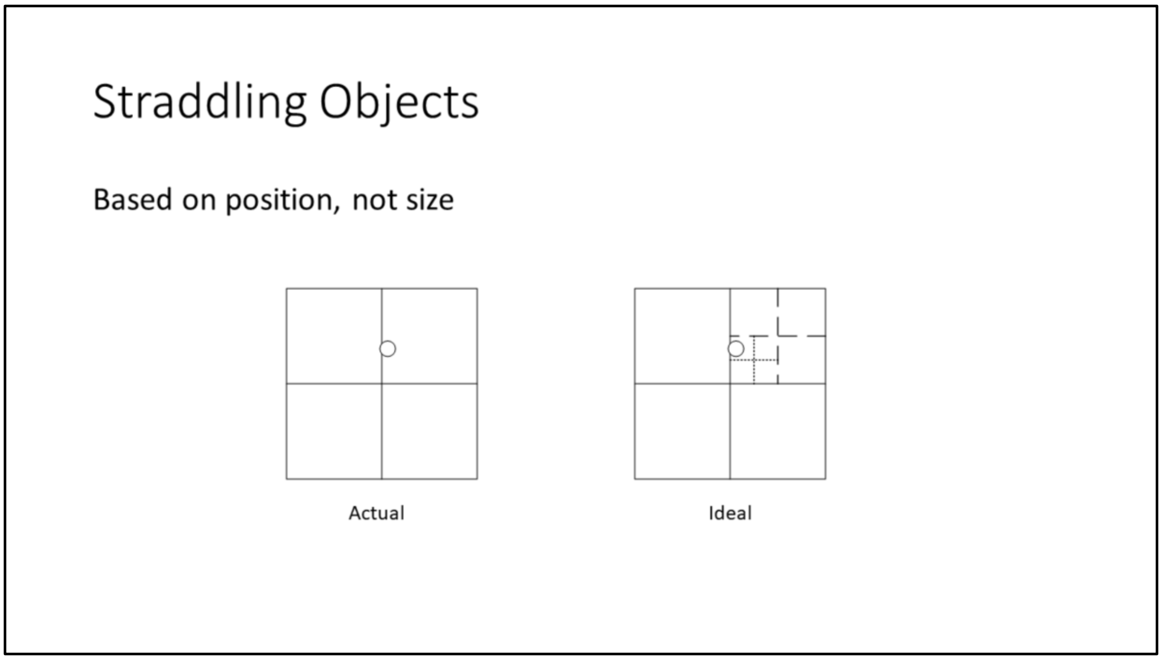
Create pairs with all objects in parent node



Pairs: (A,B) (A,D) (B,D)

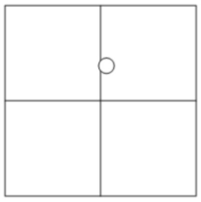
Performing pair-tests is trivial with the simple implementation. For each cell we perform an n^2 set of pair tests within the cell. We also have to recurse up through all parent cells and add a pairing for each object in our cell against the parent. We don't have to worry about children nodes as they'll test against us.

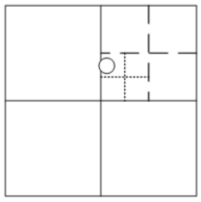
This test isn't the most efficient though as a child node can test against objects in the parent node that logically can't be anywhere near it. We can prune these tests if we prune objects as we go down the tree (instead of recursing up) but the extra cost of checking against the node's bounding volume at each level might counteract the savings. In theory we shouldn't have too many objects stuck at higher level nodes anyways.



Straddling Objects

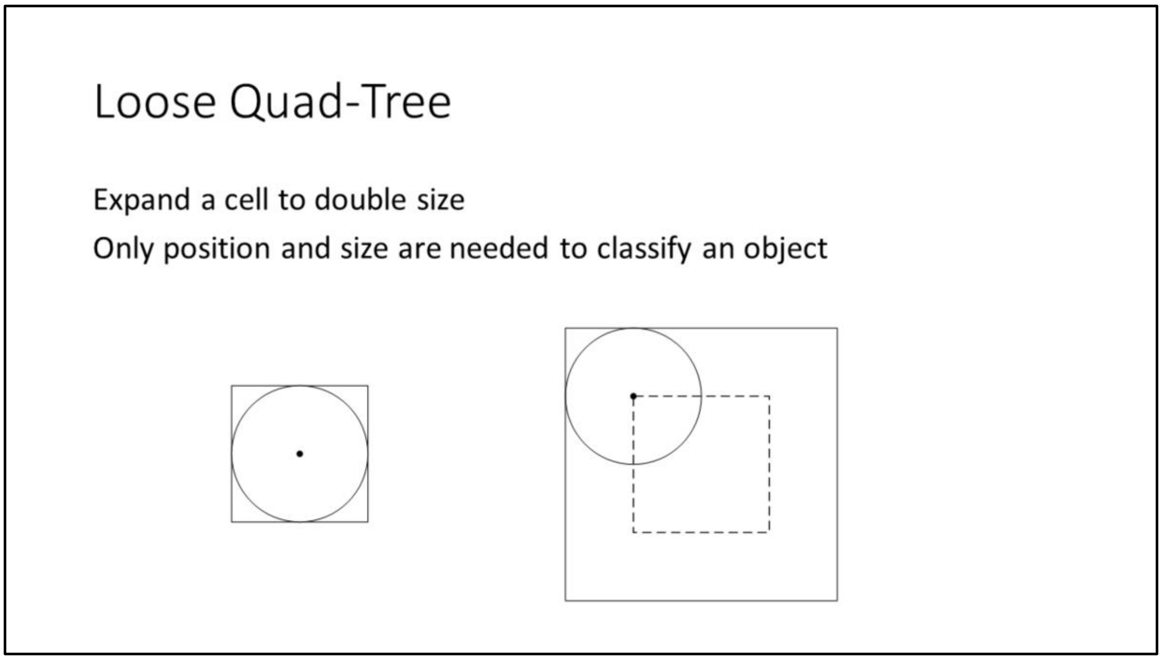
Based on position, not size


Actual


Ideal

When an object is straddling a cell’s boundaries we keep it at a higher level node. The idea is that big objects are more likely to straddle so we’ll keep them higher up in the tree (like an h-grid). Unfortunately, we are only performing this operation based upon the object’s position, it has nothing to do with its size. Because of this a lot of small objects can get stuck at a top level node just because they are unfortunately positioned.

We got around this in h-grids by inserting into multiple cells. Is there some way we can achieve a similar effect without having to manage all of the extra overhead of multiple cells?

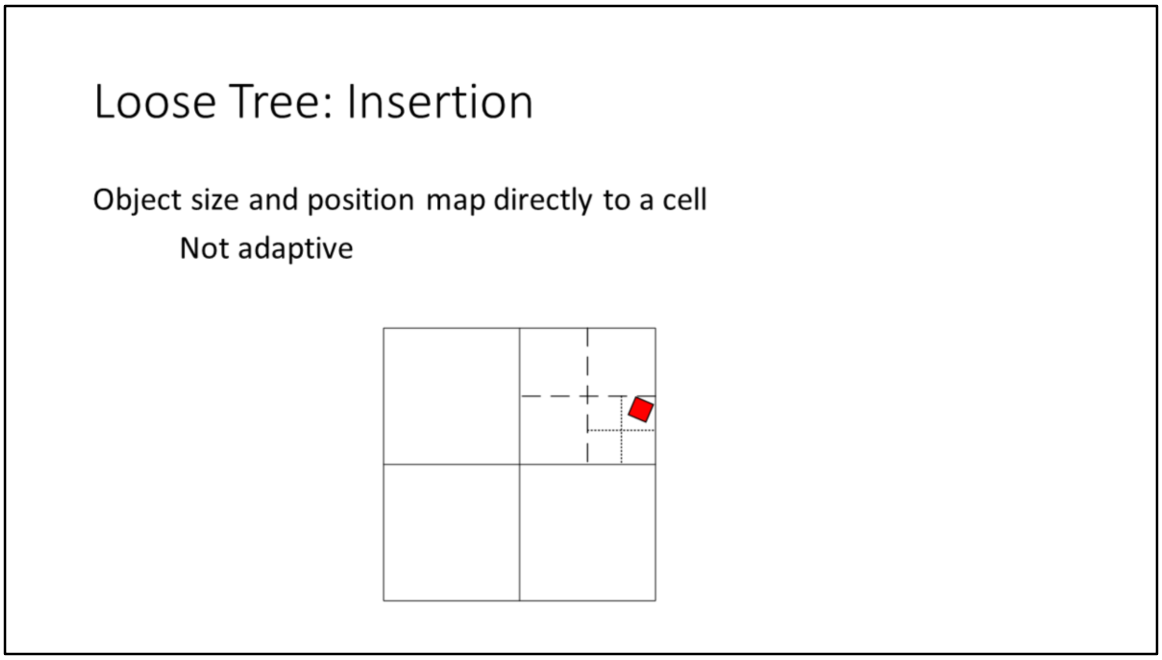


Loose Quad-Tree

Expand a cell to double size

Only position and size are needed to classify an object

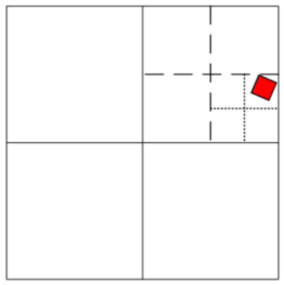
Ideally we'd also like a way to immediately map an object of a given position and size to a cell, just like with an h-grid. If we have a cell of size 8 then ideally we could insert any object of size 8 into it. However as the object moves around it would extend outside of the boundary of the cell and cause problems. Instead we can create a loose quad-tree by expanding each node's boundaries by 50%, effectively doubling it on each axis. When we do this the cell will contain any object of the correct size as long as the object's center is within the cell.



Loose Tree: Insertion

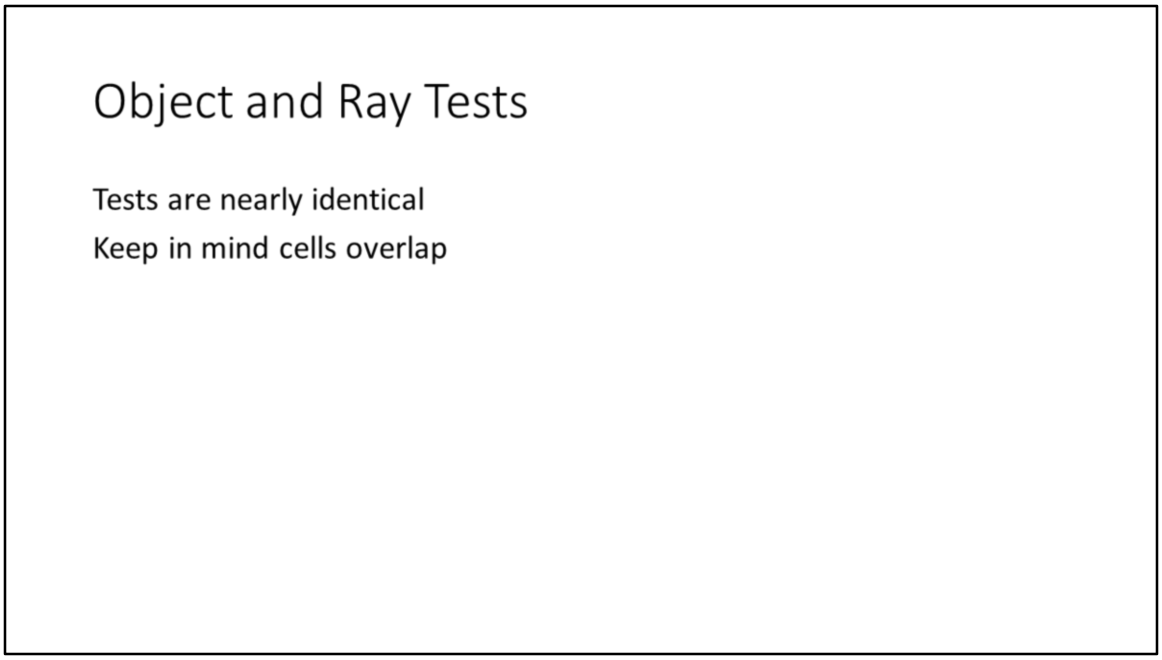
Object size and position map directly to a cell

Not adaptive



We can now augment the insertion test to just traverse down the tree to the correct depth based upon the object's size and position. As we have a direct mapping of size and position to a node we could also just insert node's into a hash-map. With this insertion becomes $O(1)$ with respect to finding the node. As this is a tree we still need to iterate up and create new parent nodes to the root. Do note that all parent-child relationships are implicit and can be computed from a given node's hash data.

Also note that this tree is no longer adaptive, an object goes directly into the cell appropriate for its size regardless of how many objects exist.



Object and Ray Tests

Tests are nearly identical

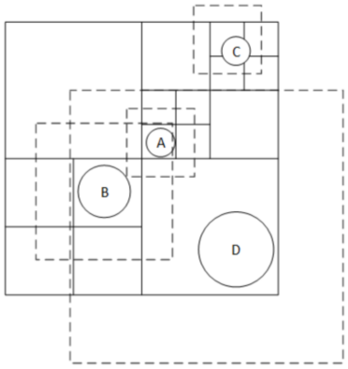
Keep in mind cells overlap

Object tests are the same as before. If an object overlaps the boundary of a cell (which is now larger) then we check all objects it contains and recurse into its children.

Ray tests are also the same, the only caveat is that we can no longer stop after the first node we hit since they overlap. If we want to terminate a ray-cast early we have to properly keep track of the first t value and only skip a node if its t-value is after our stored one.

Loose Tree: Pair Test

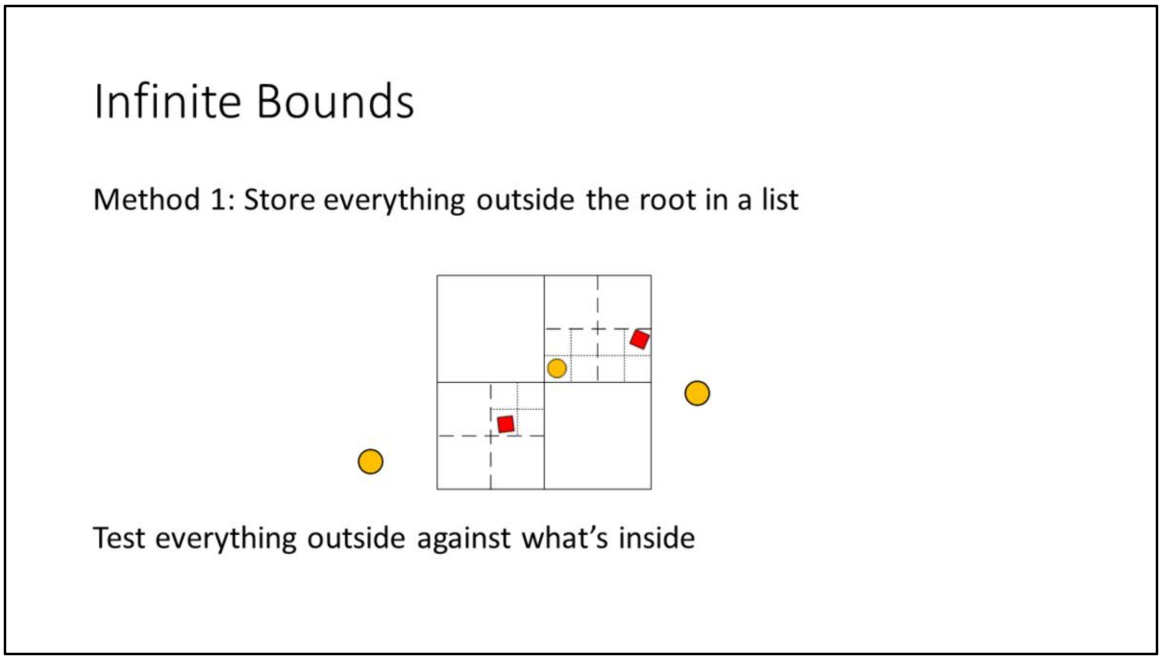
Have to test all "neighbor cells"



Pairs: (A,B) (A,D) (B,D)

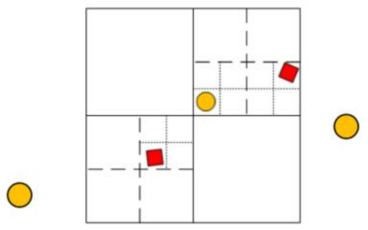
The biggest problem with the loose quad-tree is that pair tests become more difficult. Since a node's boundary now overlaps with its neighbors we have to check them as well. Unfortunately it's not just our direct neighbors we have to check. We have to check all neighboring 8 cells at each level (if they exist).

With all of that being said, loose quad-trees tend to work better for dynamic scenes than their adaptive counterparts. This is mostly because small objects won't get caught at a higher level tree node causing wasted calculation.



Infinite Bounds

Method 1: Store everything outside the root in a list



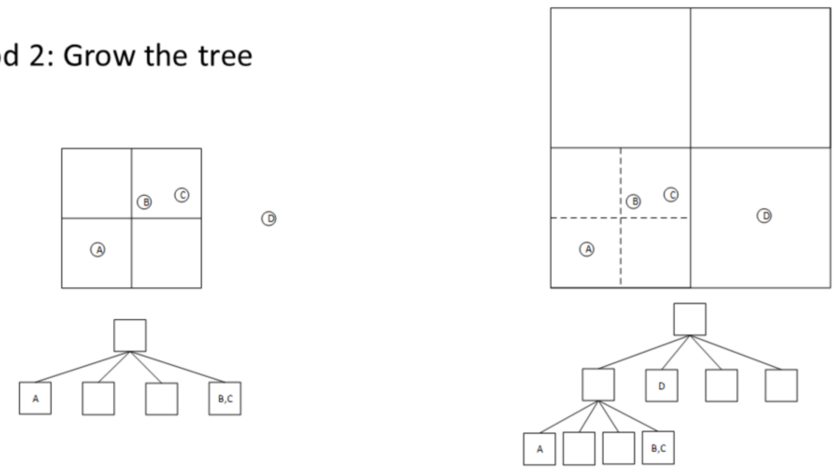
Test everything outside against what's inside

Currently the quad-tree we've discusses had finite bounds. This was primarily because we needed to create a root initially, but there are ways to allow our tree to "expand" infinitely.

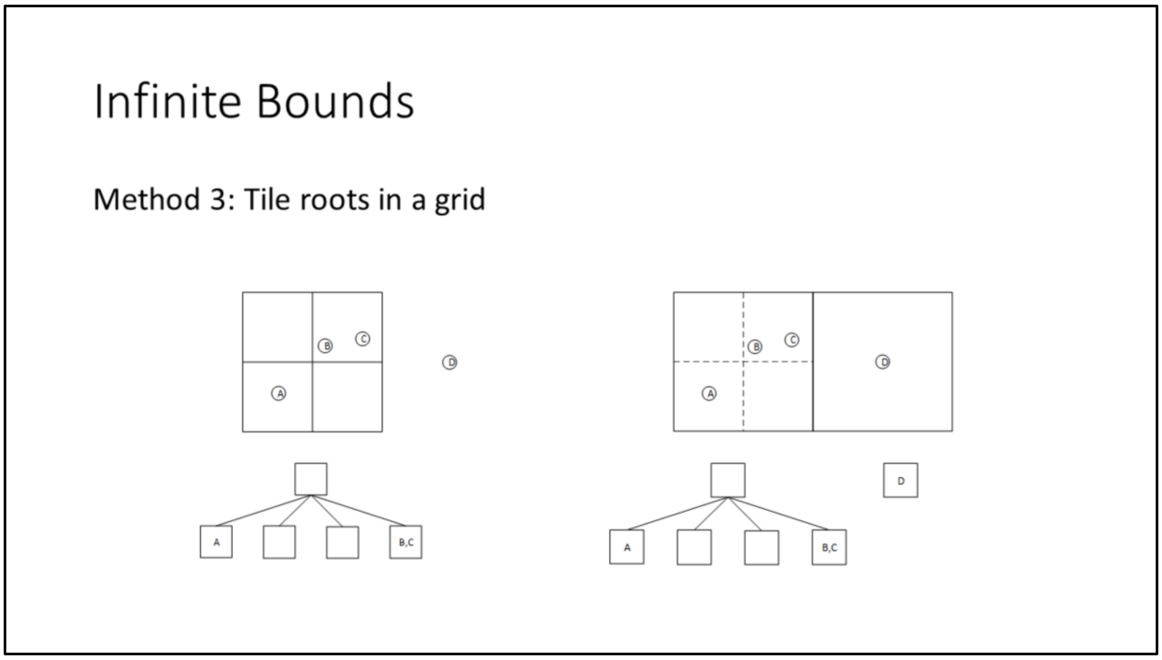
The first approach is to just consider everything outside of the root node as being in another list. Everything in this list would be assumed to be in one node. This node could even be thought of as an implicit parent of our root. When performing any sort of object or ray cast we'd have to just perform an n-squared iteration over all objects outside the grid. Likewise, during pair-tests we'd have to form a pair for every object outside as well as test them all against the tree.

Infinite Bounds

Method 2: Grow the tree



Another method is to expand the tree upwards. If we detect that we have to create a node at the same depth of the root then we could continue up the tree creating new parent nodes for both the new node and the parent until they meet. This final node would become the new root. Unfortunately this can cause the tree to grow quite a bit in height and waste lot of time checking empty nodes during any tests. However, this is the most “pure” method of growing the quad-tree.



Infinite Bounds

Method 3: Tile roots in a grid

Another solution is a hybrid scheme of sorts where we just turn the top level into an implicit uniform grid. That is we no longer have 1 root, but we have as many roots as needed. This method works quite well with our current hash-based node system.

With this we'd have to alter all tests and casts to test all roots, but we wouldn't be creating a large amount of dummy parents.

Questions?