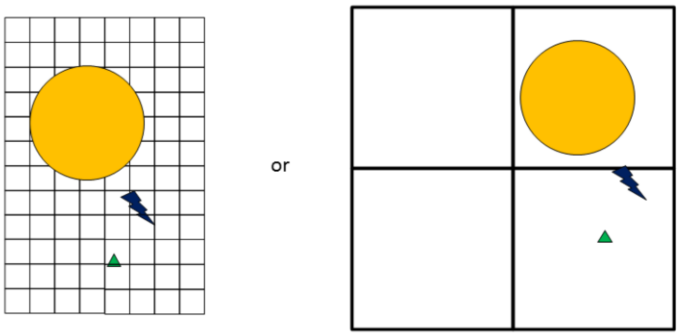
Hierarchical Grids

jodavis42@gmail.com

Uniform Grid

Uniform grids still suffer from varying object size

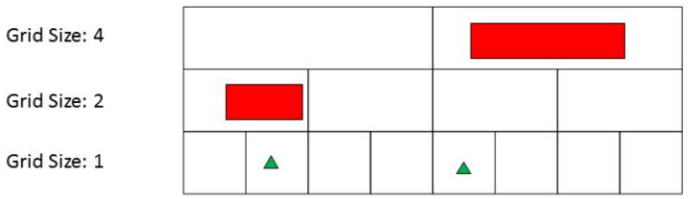


Despite all of the previous work done for uniform grids there is still one major problem that our uniform grid can't deal with elegantly: varying sized objects. Luckily there is an extension call H-Grids (hierarchical grids) that helps to fix this problem.

As a disclaimer up front, h-grids become very similar to various tree data structures that will be discussed later, in particular the loose oct/quad tree. For the most part the difference is in data representation, but concepts are incredibly similar.

H-Grid

Basically just add more grids



Insert objects into cells based upon position and size

As the name implies an h-grid is a hierarchical structure of grids. For example, an h-grid of only 2 levels where the finest level (our previous uniform grid) has size 1 would typically have a second level of size 2. With this we now have an implicit relation between grid levels.

An h-grid now fixes the problem of uniform grids by inserting an object into a different grid level depending on its size.

Grid Level

Grid level can be thought of as 3rd array index based on object size

Discretize sphere's radius to grid level:

$$i = ceil(\log_2(2 * radius))$$

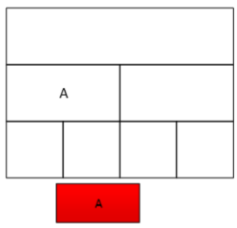
It's easiest to think of the h-grid level (which uniform grid level) as just a 3rd index that we can compute by discretizing the objects size. The problem with this discretization is that it's not linear, but exponential. If our grids increase by powers of 2 each time then we have the mapping:
Radius(.5) -> GridSize(1) -> Index(0)
Radius(1) -> GridSize(2) -> Index(1)
Radius(2) -> GridSize(4) -> Index(2)
Radius(4) -> GridSize(8) -> Index(3)
and so on.

So we can look at the largest radius that a grid cell can support as $r = \frac{1}{2}2^{index}$. To invert this problem to find the grid index we can just use log base 2: $index = \log_2(2r)$.
We do still need to worry about what happens with values that are not exactly on the grid size boundaries, things like a radius of 1.1.
Well $\log_2(1.1) = .13$ which means we'd compute a size index of 1.13. Remember that Index(1) can only support radius' up to 1 so we must use Index(2). Hence the final formula for computing a sphere's index is: $index = ceil(\log_2(r) + 1)$.

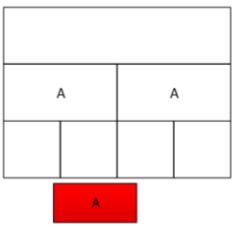
This was of course assuming that the grids at index 0 were of size 1 and that cells increased by powers of 2. Different based logs can easily be used to change our increasing power. To deal with different starting locations is left as an exercise to the reader (although the basic idea is to subtract off the starting power of 2).

H-Grid Insertion

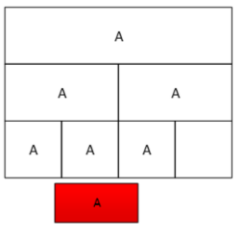
Similar problem as before, which cells to insert into?



Object center?



All overlapped on current grid level?



All overlapped?

Once again we have similar issues to discuss about where and how an object gets inserted, only this time there's an extra dimension to cover. This new dimension presents the first problem of do we insert into all grid levels or just the most appropriate one for the object. The short answer is to just insert into the most appropriate one.

Similarly we need to deal with how many cells on a grid level to insert into. If we were dealing with just pair tests for collision then we could get away with just the object center, however for the same reasons as before ray-casting would have issues.

H-Grid Removal

Same as with uniform grid

The only real difference here is we need to store in our Box struct the h-grid level that our object was inserted into.

H-Grid Update

Simple method: remove old values, insert new values

Can slightly optimize by not removing from a cell that you'll insert into

H-Grid update doesn't have much to talk about as the update is just a removal followed by an insertion. Minor optimizations can be made in an update by checking if we'll remove from a cell to just re-insert back into it. Assume each cell just stores an array of objects contained then we can avoid the overhead of finding and removing from that array if we're just going to re-insert into it.

Shape-Casting

Shape-casting is the same, just more complicated (check each h-grid level)

Shape-casting with an h-grid is the same as with a uniform grid. First figure out what cells at the top level (most coarse) are overlapped as you would a regular uniform grid. All of these cells correspond to cells at a lower level and they can recursively be iterated through to find all relevant objects.

Ray-Casting

Ray-casting is the same, just with more grids

Be careful to get t-first results

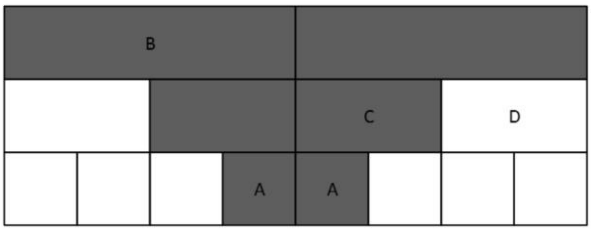
Ray-casting with an h-grid is almost identical to just casting against multiple uniform grids. The only major issue is sorting the results to be in t-first order. You can get tricky with casting and start at the root level and recursively descend into lower grid levels as you come across them, but writing this code is quite tricky. Also note that you could still have t-range differences across all levels of the grid as a lower grid can have an early t-value than a higher grid and vice versa.

Also be careful about duplicate intersections if you insert an object into multiple cells.

Pair Query

Check all cells above current object

Assumes object's below check against object



The simplest way to do pair checks in an h-grid is to check for all pairs at once. This is easiest by having each object traverse up the grid and registering pairs for each object it finds. By only traversing up though we won't find some pairs (for instance, object C will not find out it should be colliding with A). Since we're checking for all pairs at once though objects in lower cell levels will get the remaining check (A will catch C). Do note that we still have to be careful of duplicate pairs.

Hashing

H-Grids take a lot of memory

Hash cells (or chunks)

Almost any extra addition we talked about before with uniform grids can be added to h-grids. In particular, h-grids almost have to use some form of cell hashing otherwise they'd just take up too much memory. Similarly, chunks could be added.

Various other small optimizations can be made such as skipping marking if a level contains any objects and skipping it if it doesn't.