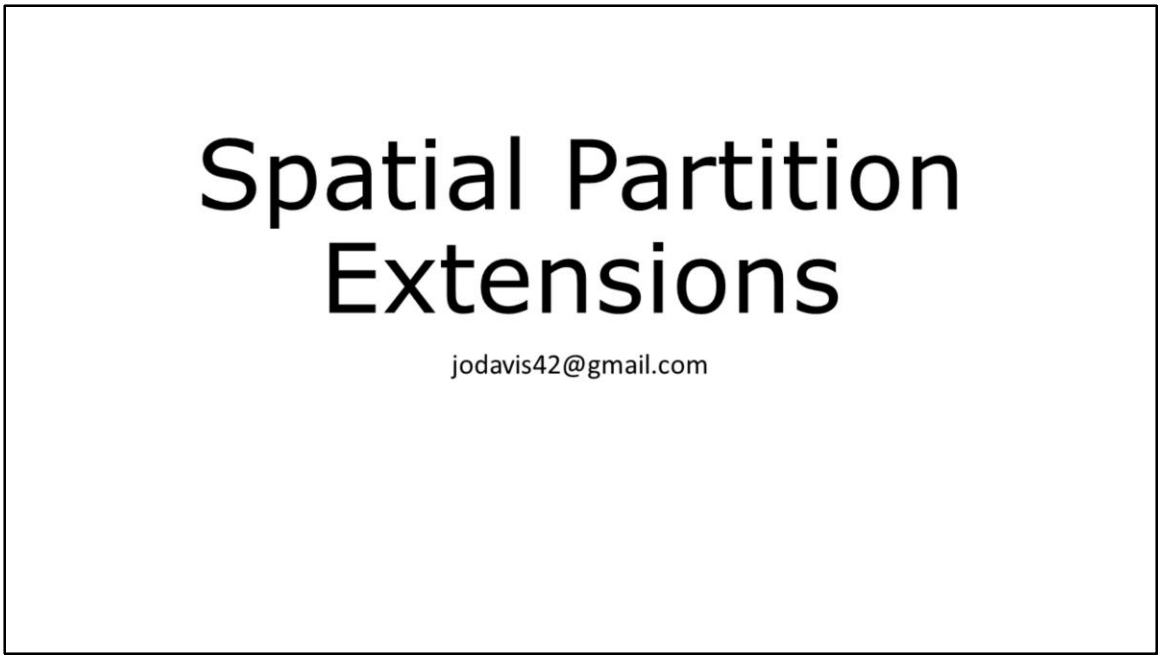
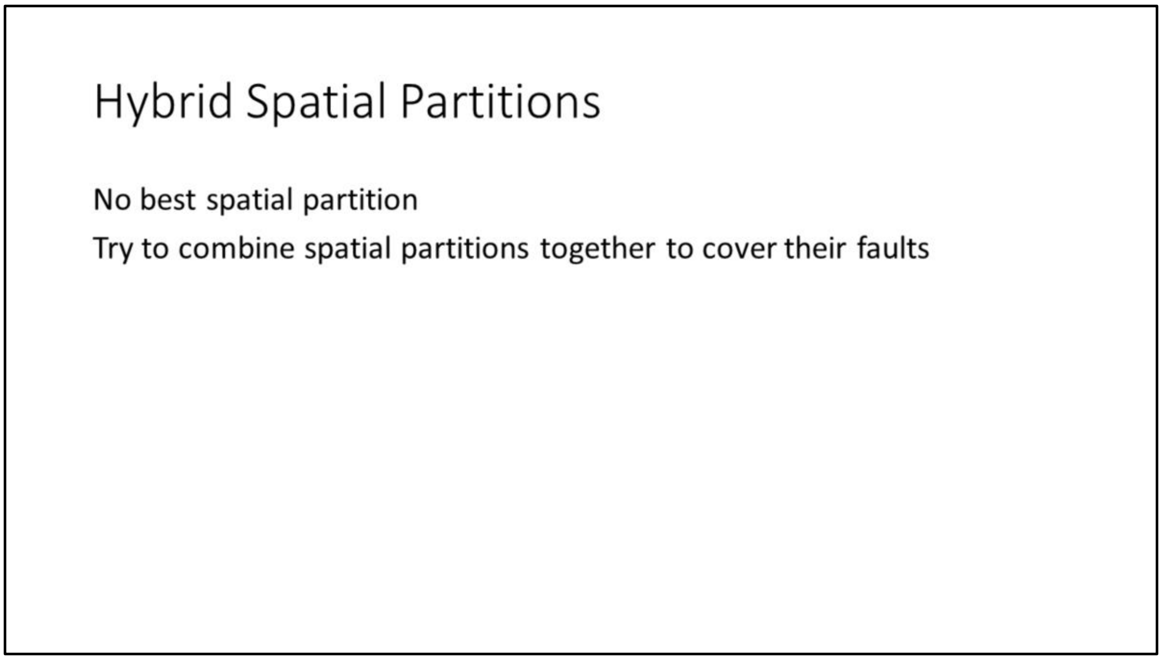


Spatial Partition Extensions

jodavis42@gmail.com

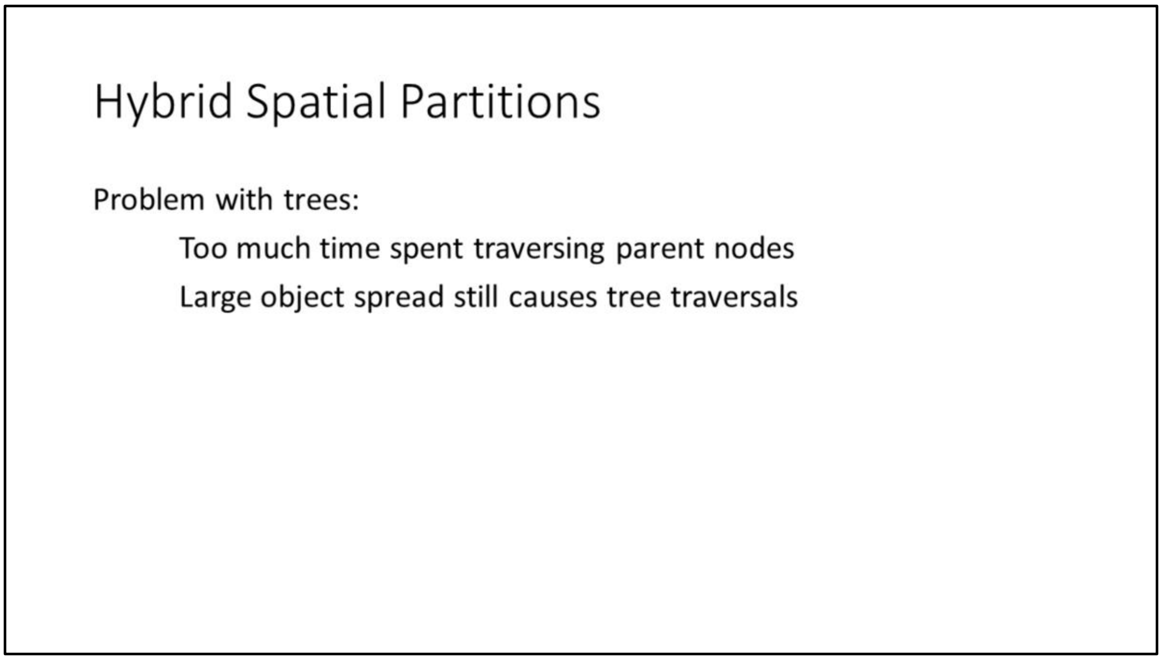
This lecture is about various ways to extend spatial partitions.


Hybrid Spatial Partitions

No best spatial partition

Try to combine spatial partitions together to cover their faults

One very useful concept is hybrid spatial partitions. One issue with spatial partitions is that they all have their problems. No spatial partition is the best. They all have various problems and some work better in some situations than others. One attempt to mitigate this is to combine spatial partitions together. By doing this we can hopefully gain the benefits of both spatial partitions.



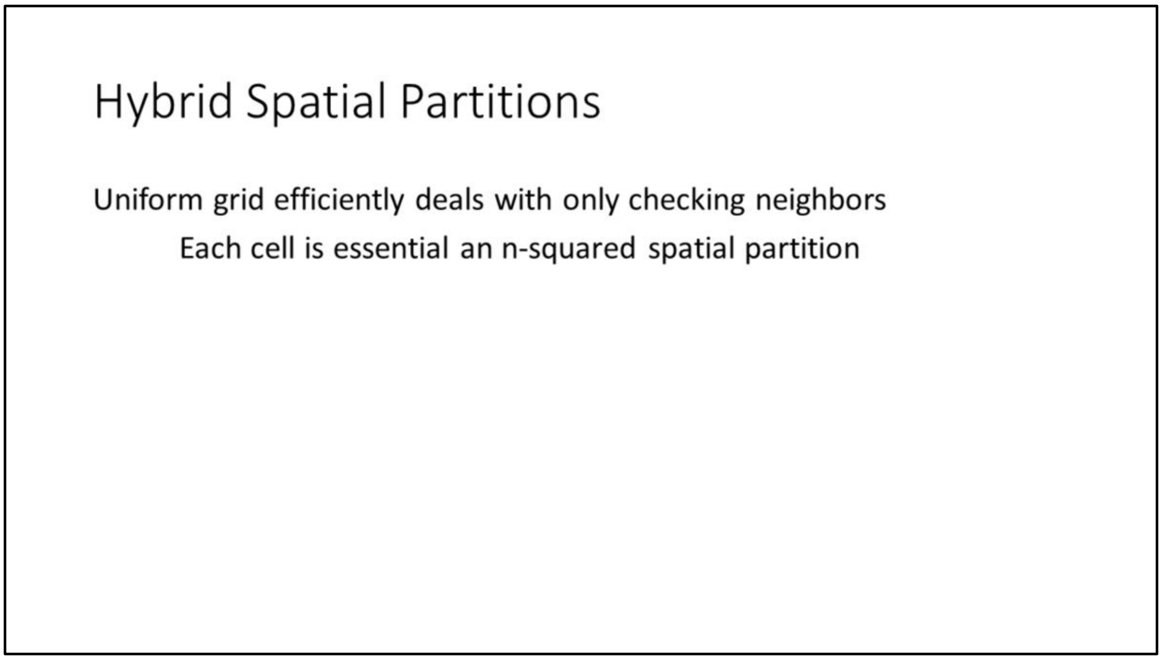
Hybrid Spatial Partitions

Problem with trees:

Too much time spent traversing parent nodes

Large object spread still causes tree traversals

One such idea is to look at a tree structure. Trees work really well (especially BVHs) but a lot of time can be spent in the root levels of a tree. That is, we have to spend a lot of time traversing nodes to get to children nodes. This is especially a problem with very large spread out scenes.

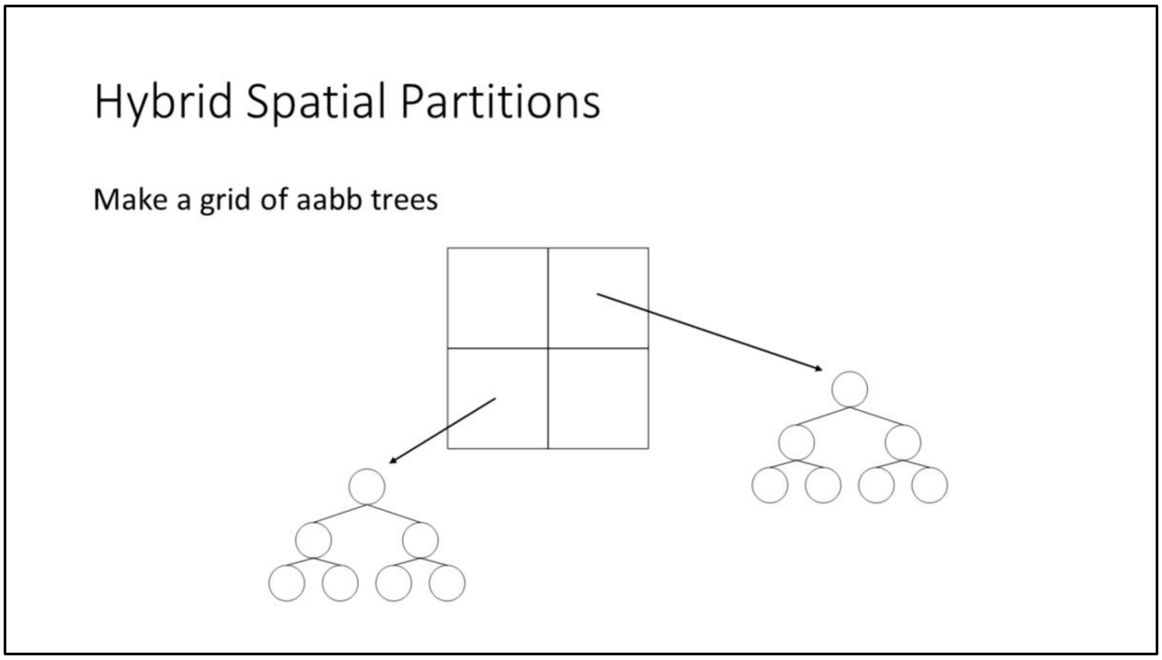


Hybrid Spatial Partitions

Uniform grid efficiently deals with only checking neighbors

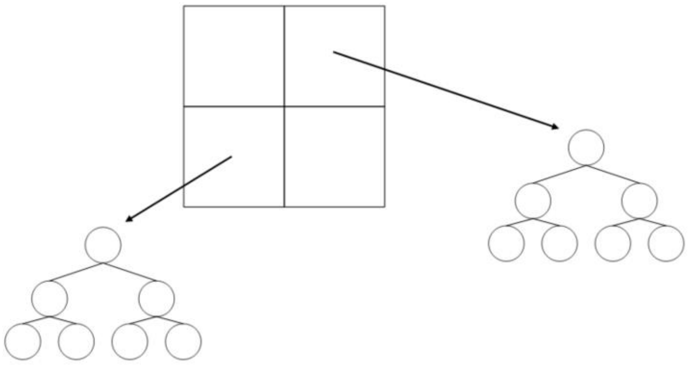
Each cell is essential an n-squared spatial partition

Uniform grids on the other hand have the exact opposite problem. It’s very quick to check only nearby objects by just checking neighboring cells. That being said, grids have the problem of having to pick a fixed grid size. If a lot of objects end up in the same grid cell then a lot of time can be wasted. We attempted to fix this before with the H-Grid, but maybe there’s another solution?



Hybrid Spatial Partitions

Make a grid of aabb trees



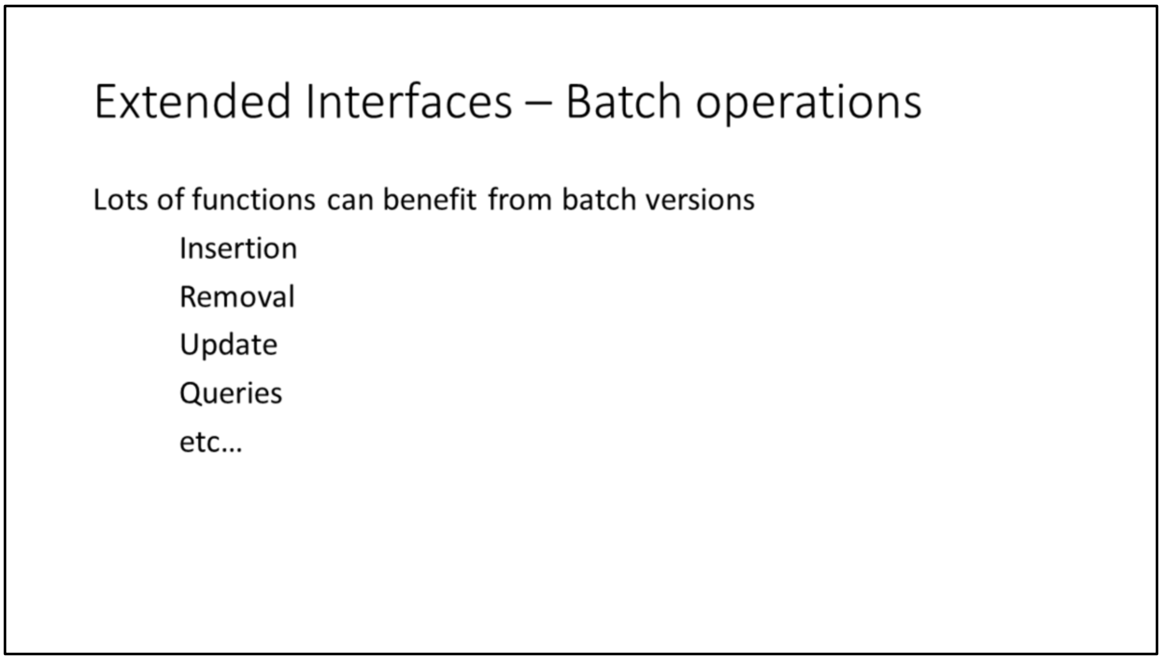
One such solution is to combine uniform grids with an aabb tree. Instead of storing a list in each cell, we can store an entire aabb tree. The best way to really think about this is we avoid a lot of high-level clutter in our tree by breaking it up into large areas. With a solution like this, it tends to be better to have very large coarse grids as the tree takes care of most of the heavy lifting.

Alternatively we can even do things like have cells in our grid point to certain tree internal nodes, effectively skipping portions of the tree.

Hybrid Spatial Partitions

Other Examples:

- Grid of Grid: Chunks
- Grid of SAP: Multi-SAP
- Grid of anything…
- etc…



Extended Interfaces – Batch operations

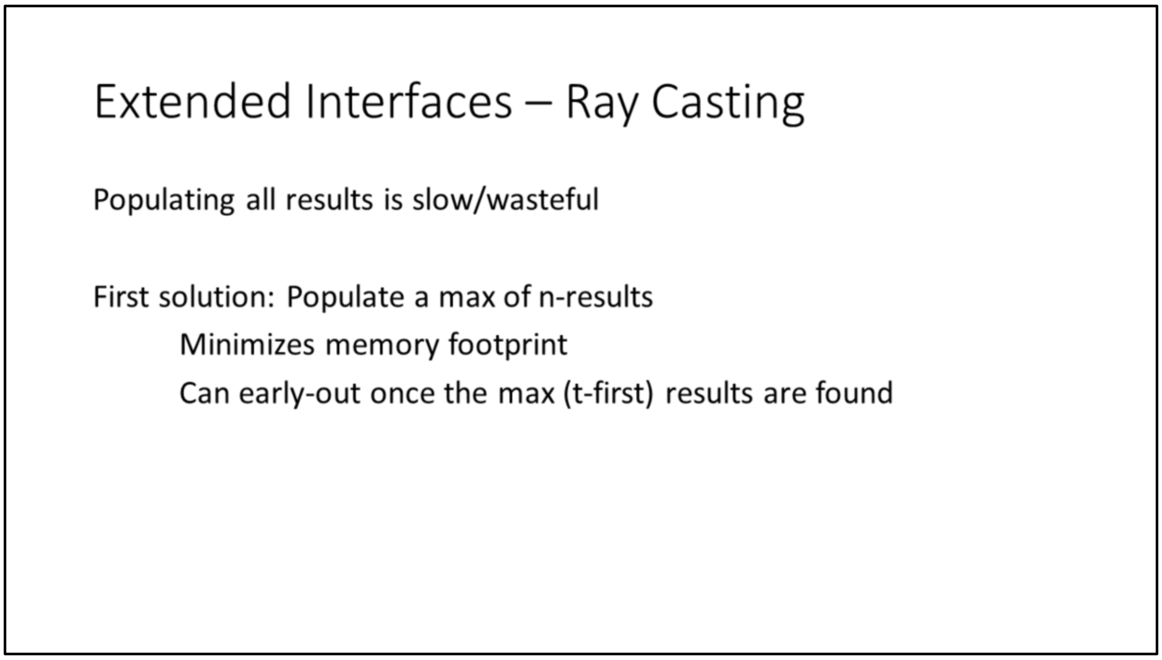
Lots of functions can benefit from batch versions

- Insertion
- Removal
- Update
- Queries
- etc…

Sometimes operations can be performed more efficiently if they're batched. One of the best examples of this was batch insertion/removal in SAP. Batch operations can extend to almost any spatial partition function though. In particular queries can gain a lot of benefits.

One example of batching is with raycasts, especially in a tree based structure. A large amount of time can be spent traversing top-level nodes. If rays can be grouped together then we can make only one pass through top level nodes.

One of the other cases where this can come up a lot is with object queries. Instead of performing 10 individual aabb queries against an aabb tree it could potentially be faster to construct a small aabb tree from the query objects. This tree can use the tree vs. tree query to efficiently find all collisions which minimizes top-level queries.



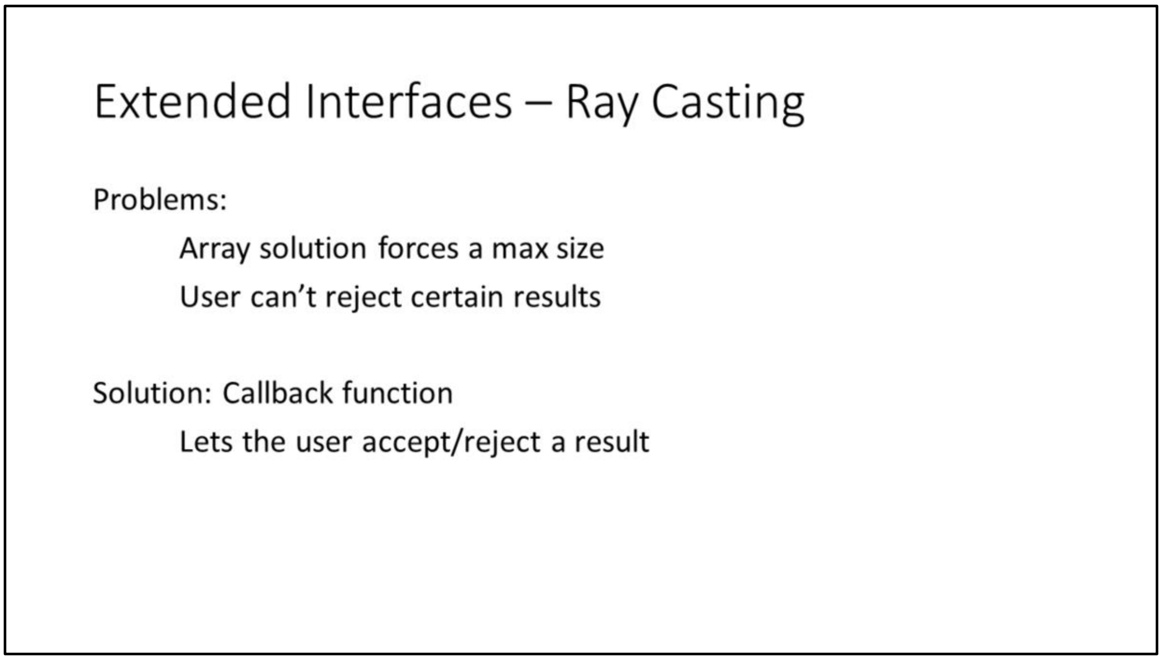
Extended Interfaces – Ray Casting

Populating all results is slow/wasteful

First solution: Populate a max of n-results
- Minimizes memory footprint
- Can early-out once the max (t-first) results are found

Another important extension is in casting. So far with a ray-cast we just populated all possible results into an array. This is wasteful as a user could be after only 1 result but we could end up giving them 100. This causes us to spend more time computing results. There are also a lot of wasted allocations.

One solution to this is to force the user to say how many results they want max. First of all, this allows us to allocate an array up-front of the max result size instead of potentially allocating multiple times. This also allows early-outs. If we find 5/5 results and we know that no result can possibly happen before our currently computed ones then we can stop iteration. This is most obvious in queries that are only after 1 result.



Extended Interfaces – Ray Casting

Problems:

Array solution forces a max size

User can't reject certain results

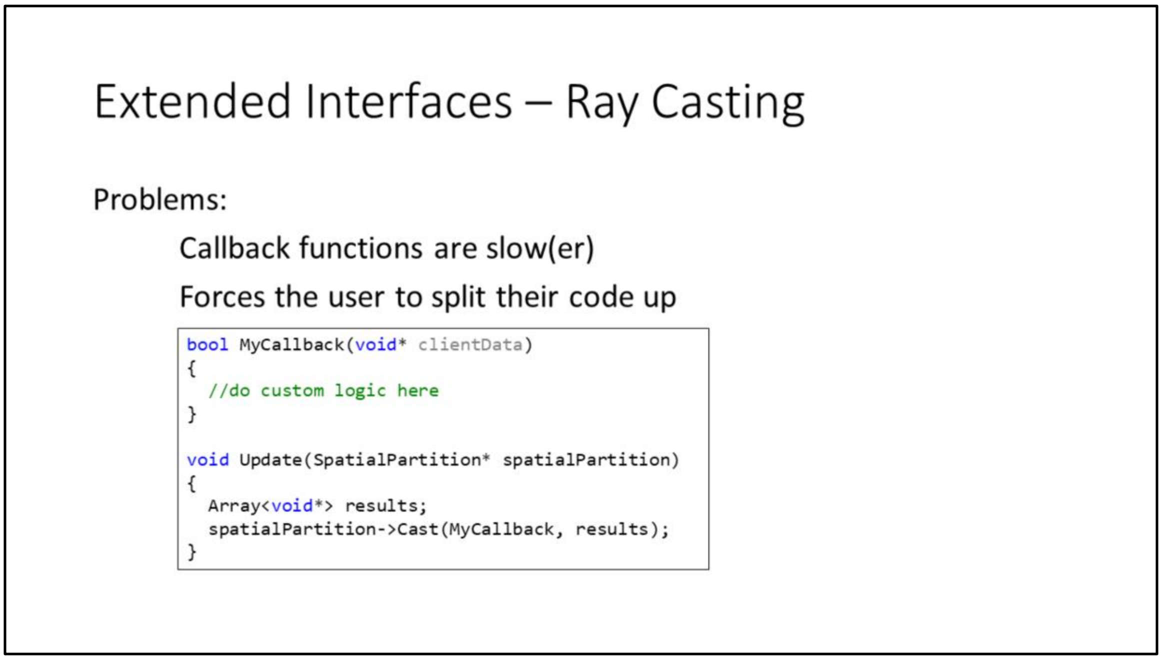
Solution: Callback function

Lets the user accept/reject a result

This method isn't without problems though. The first one is that we make the user specify a max size when they might not know what a good number is. If we aren't careful the user could just specify a really large number causing a very large allocation. The second (and bigger problem) is if the user is looking for a specific kind of object they can't filter things out.

Image that the user wants to find the first enemy that a ray hits. As there could be a lot of objects in the way (walls, bullets, etc…) they must provide a large number of possible results just to find what they're after.

One solution to this is to do a callback function instead of an array. A callback function per result lets the user accept/reject any result they want. In this case they could accept only 1 result (to find the closest) and reject anything that's not an enemy.



Extended Interfaces – Ray Casting

Problems:

Callback functions are slow(er)

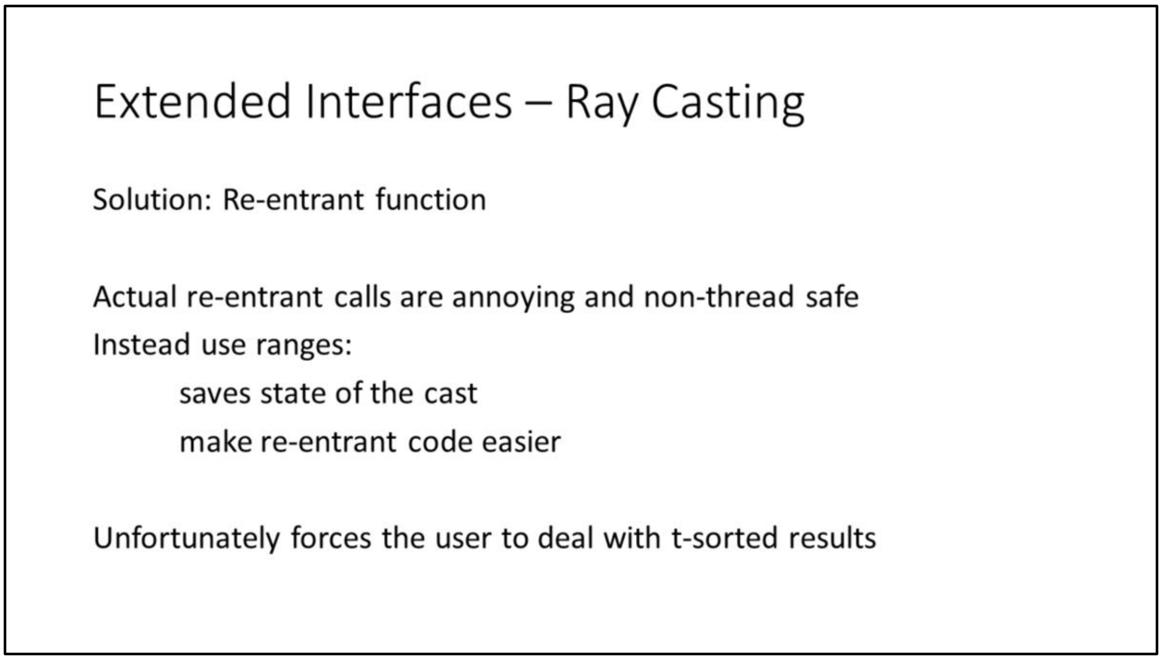
Forces the user to split their code up

```
bool MyCallback(void* clientData)
{
  //do custom logic here
}

void Update(SpatialPartition* spatialPartition)
{
  Array<void*> results;
  spatialPartition->Cast(MyCallback, results);
}
```

Callback functions are also a bit problematic. One (minor) problem is that a callback per object isn't the most efficient. Function pointer calls aren't the best for cache performance as the cpu can't know what code to next execute which can kill branch prediction.

The larger issue is that while this paradigm gives the user a lot more flexibility it is also really annoying from a user's perspective. The user has to split up their logic into 2 functions: the one that calls and uses the results and the one that filters results. Depending on the final interface, the callback function might not be able to do a lot (such as store results) although things like a policy class could be used. Either way, forcing the user to split their code is annoying.



Extended Interfaces – Ray Casting

Solution: Re-entrant function

Actual re-entrant calls are annoying and non-thread safe

Instead use ranges:

- saves state of the cast
- make re-entrant code easier

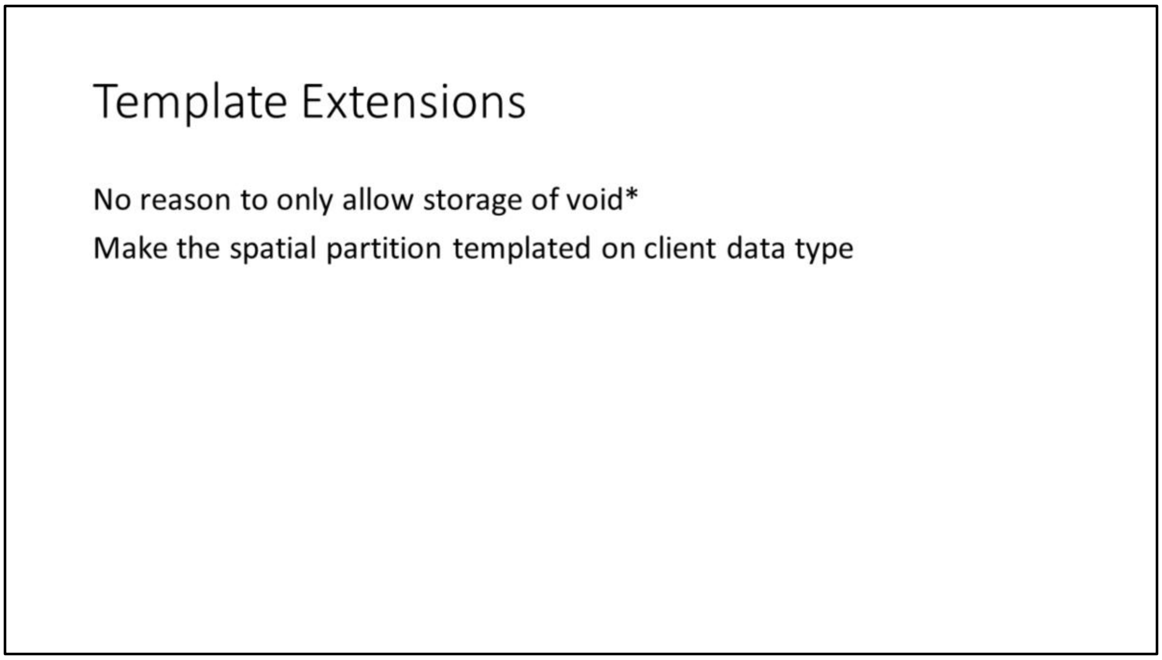
Unfortunately forces the user to deal with t-sorted results

The easiest solution to the code splitting problem is to allow the user to write all their code in one function. The main way to do this is some form of a re-entrant function call. Actually writing a re-entrant casting function isn't without problems though, most notably that it's non-thread safe. Instead one good solution is to make a range class.

Quick side note. A range for a container (such as array) is basically a wrapper around iterators. The range knows if it's empty, can retrieve the current item, and can move to the next item.

Ranges for a spatial partition are a bit different though. A range returned from a spatial partition can keep track of the entire stack of state needed for a cast hence making everything thread safe. Unlike with containers though, a range in a spatial partition doesn't necessarily have an end though. Whenever you move to the "next item" in the range the cast starts up where it left off until it finds the next result. The range is only empty when there are no more results to process. In this way, the actually cast is contained in the range. This allows to user to walk through as many results as they want and even cancel the cast mid-way through by choosing to not move to the next item.

That being said this still isn’t without its flaws. Most notably, it’s not always possible to traverse t-first through a spatial partition (in a reasonable fashion). In this case the user is forced to sort all their results in t-first order. Luckily the n-based array solution can be combined with ranges to produce the first n-results from a cast.

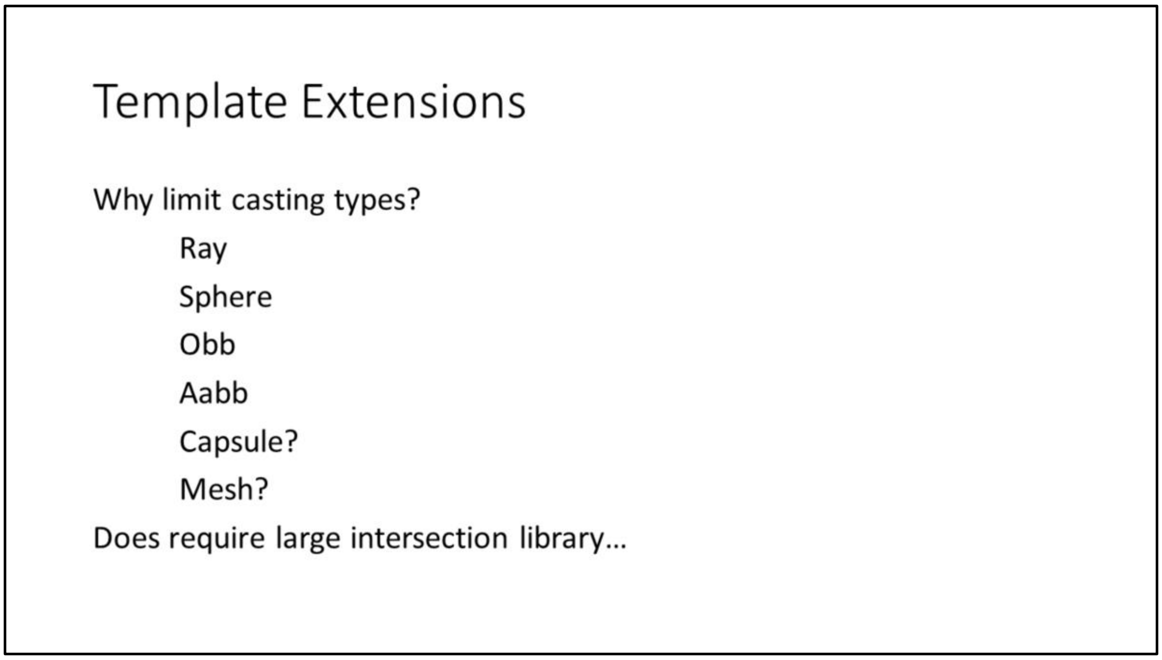


Template Extensions

No reason to only allow storage of void*

Make the spatial partition templated on client data type

Templates can be used to extend a spatial partition a bit further. The first of which is to allow the user to store whatever they want in the spatial partition, not just void*. The user might find it useful to store indices into an array of triangles or some other unknown thing.



Template Extensions

Why limit casting types?

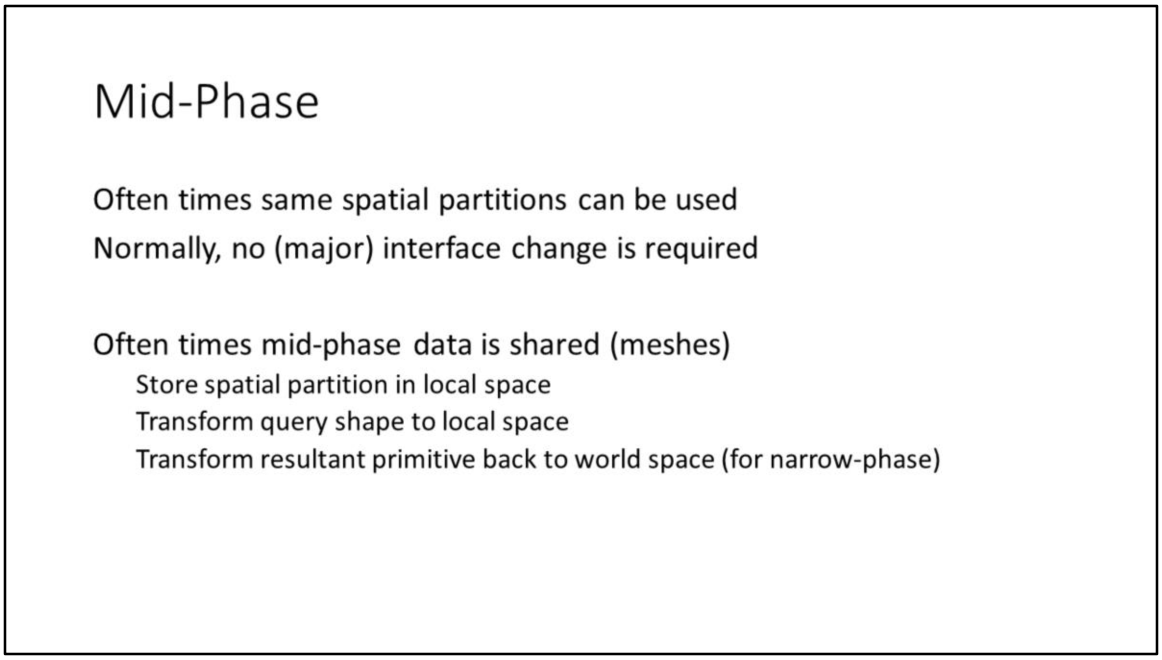
- Ray
- Sphere
- Obb
- Aabb
- Capsule?
- Mesh?

Does require large intersection library…

The second big template extension for spatial partitions is with casting. Currently we've describe limiting casting to a certain set of shapes such as rays and frustums. But what if the user wants to cast another shape? We could explicitly write every possible shape but this can become quite cumbersome. Most spatial partitions can be extended trivially to test more shapes. For example, checking an aabb tree with an aabb or a sphere is the same logic, just a different intersection test is performed at each level.

To properly pull this off does require a large intersection library though.

Note: This is a large endeavor and not something I recommend to start with.



Mid-Phase

Often times same spatial partitions can be used

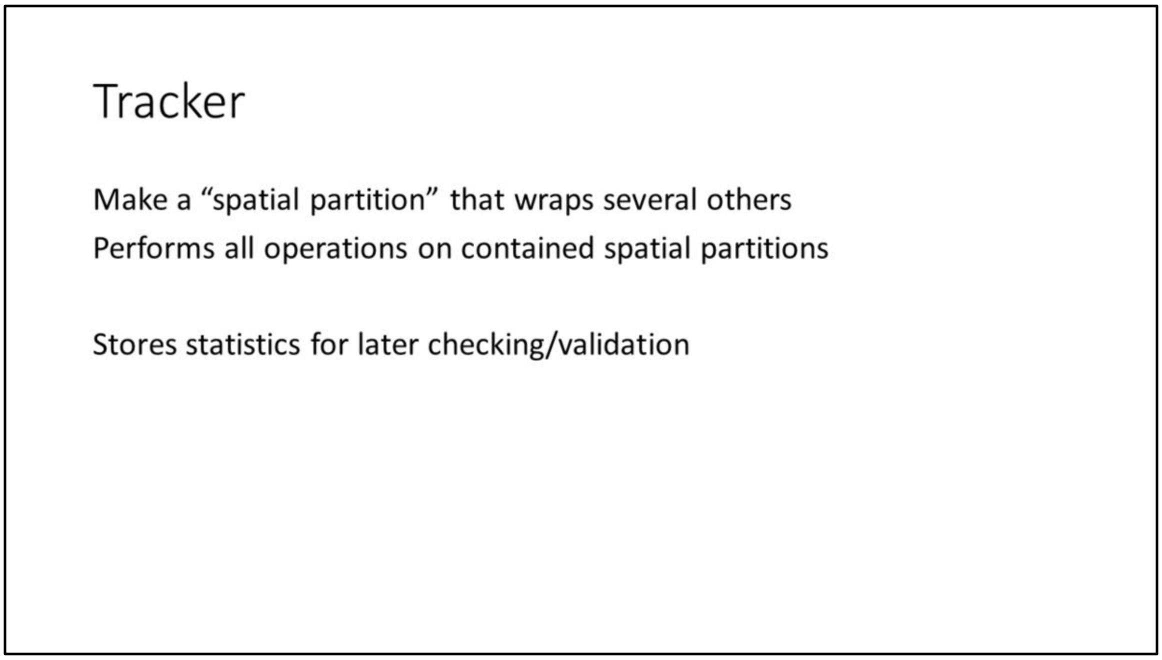
Normally, no (major) interface change is required

Often times mid-phase data is shared (meshes)

- Store spatial partition in local space
- Transform query shape to local space
- Transform resultant primitive back to world space (for narrow-phase)

While it was only mentioned briefly before, another key thing to consider when generalizing spatial partitions is re-usability. In particular, many mid-phases can use the exact same spatial partition that a broad-phase would use but just store different data in it. For instance, a triangle mesh might build a tree where all of the objects inserted are triangles of the mesh.

That being said, mid-phases are an interesting case. If a mid-phase is constructed in local space then it can be shared across all instances of a mesh. Typically no (major) interface changes are required for this but the usage of the spatial partition does change significantly. Most notable the objects are stored in local space in the spatial partition. When querying an object against the mid-phase it is then transformed back into local space. Afterwards any results that pass are transformed back into world space so narrow-phase can compute the actual results.


Tracker

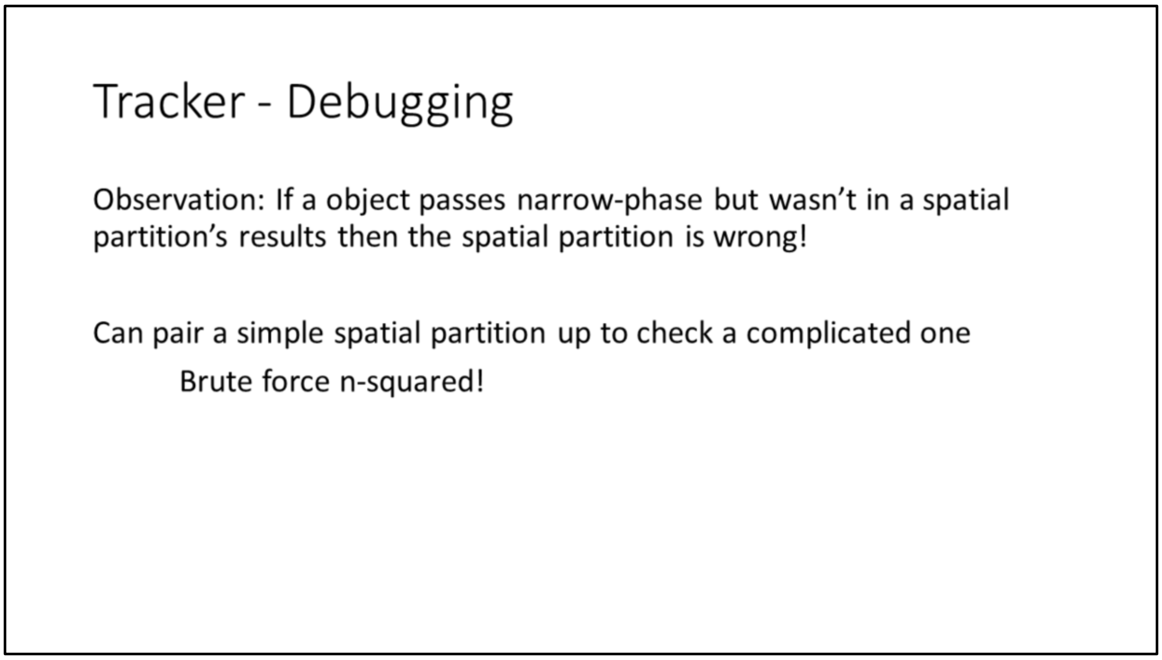
Make a “spatial partition” that wraps several others
Performs all operations on contained spatial partitions

Stores statistics for later checking/validation

The final set of spatial partition extensions I’ll talk about is a tracker. The core idea behind the tracker is to run multiple spatial partitions at the same time and collect statistics about each of them. You can think of the tracker as a fake spatial partition that performs all operations on all the spatial partitions it contains.

For instance, when inserting into the tracker it will insert into each spatial partition it contains and store the keys internally to re-map all subsequent operations.
More interestingly, during casts/queries the tracker will merge the results from all the spatial partitions it contains.

Do note, the tracker will likely not be performant, but it can give a lot of interesting data!



Tracker - Debugging

Observation: If a object passes narrow-phase but wasn't in a spatial partition's results then the spatial partition is wrong!

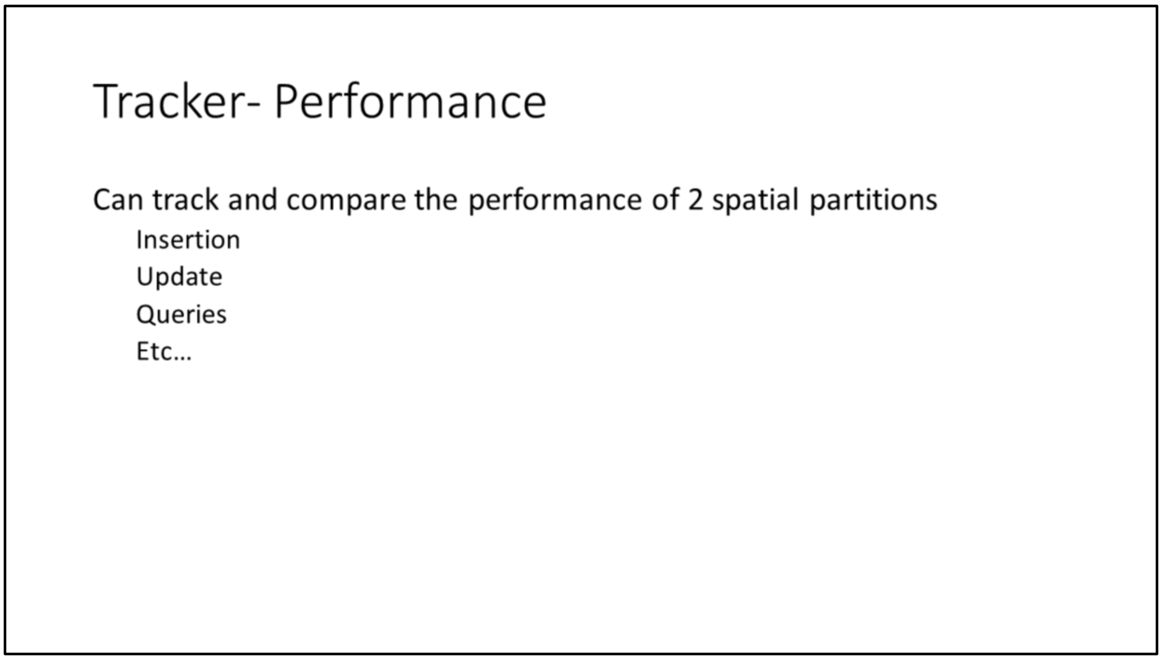
Can pair a simple spatial partition up to check a complicated one

- Brute force n-squared!

One of the best features of the tracker is it can be used to validate results of a spatial partition to help debug it. This does require the work of your narrow-phase to work, but it's well worth it.

The tracker can validate results with 1 key observation (example of ray-casting): if narrow-phase says a ray hits an object but a spatial partition doesn't return that object as a possible result then the spatial partition must be wrong.

So how do we use this observation? We can merge the results from any cast together (as it's fine for 1 spatial partition to return extra results) and then check what actually passed narrow-phase. If anything passed narrow-phase but didn't come back from one of the spatial partitions then there's an error. This is where the brute force n-squared spatial partition is helpful as every result will be tested. After validating a slightly more advanced spatial partition (such as n-squared aabb) then that can be used for validation and so on.

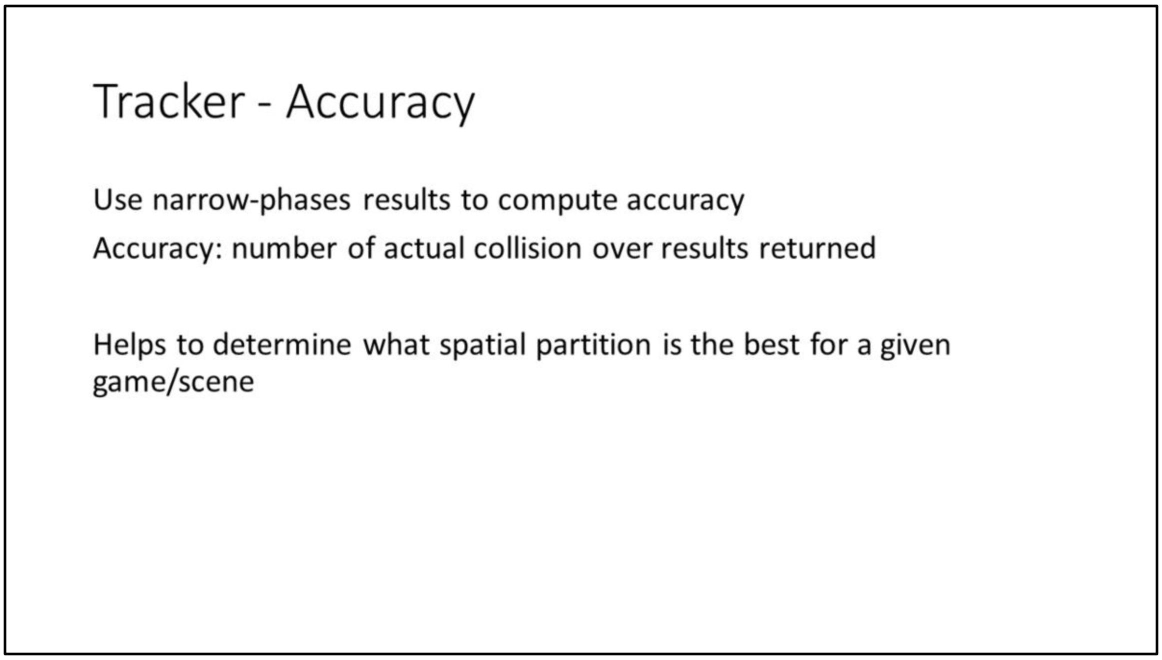


Tracker- Performance

Can track and compare the performance of 2 spatial partitions

- Insertion
- Update
- Queries
- Etc…

The tracker can be used for a lot more than debugging though. As mentioned before, there's no "best" spatial partition. One may be good at ray-casting while another may be really good at pair-queries. What the tracker can do is time how long each spatial partition takes to perform a given operation (insertion, removal, casting, etc…). As the exact same cast is performed on the same data just in different spatial partitions we can accurately compare 2 spatial partitions for how they perform against each other in a given scene.



Tracker - Accuracy

Use narrow-phases results to compute accuracy

Accuracy: number of actual collision over results returned

Helps to determine what spatial partition is the best for a given game/scene

Another thing the tracker can do (with the help of narrow-phase) is to track accuracy. Remember that one key consideration of a spatial partition is how tight-fitting it is, that is how many false-positives it returns. With the tracker we can keep track of all results from a spatial partition and then compute how many of those actually were intersecting. From this we can determine what's the “most accurate” spatial partition for a given scene.

Questions?