Hi, everyone. Welcome to this talk – Picking A Good Spot: Naughty Dog's Post System.

Ming-Lun "Allen" Chou / 周明倫

Senior Game Programmer @ Naughty Dog

AllenChou.net

@TheAllenChou

Uncharted 4
(2016)

Uncharted: The Lost Legacy
(2017)

The Last of Us Part II
(2020)

The Last of Us Part I – TLoU remake
(2022)

March 20-24, 2023 | San Francisco, CA #GDC23

GDC

My name is Allen Chou. I'm a senior game programmer at Naughty Dog.

I've been a longtime Naughty Dog fan, and I had the honor of working on the last 4 Naughty Dog games: Uncharted 4, Uncharted: The Lost Legacy, The Last of Us Part II, and The Last of Us Part I. Note that The Last of Us Part I is a remake of the original Last of Us, so it's ordered after Part II.

Acknowledgement

John Bellomy

Max Dyckhoff

Travis McIntosh

Kareem Omar

Eli Omernick

Hecate Robison

Here’s a shout-out to current and former dogs on who contributed to the post system discussed in this talk.

Problem:
Picking A Good Spot

Picking a good spot is a very common gameplay problem.

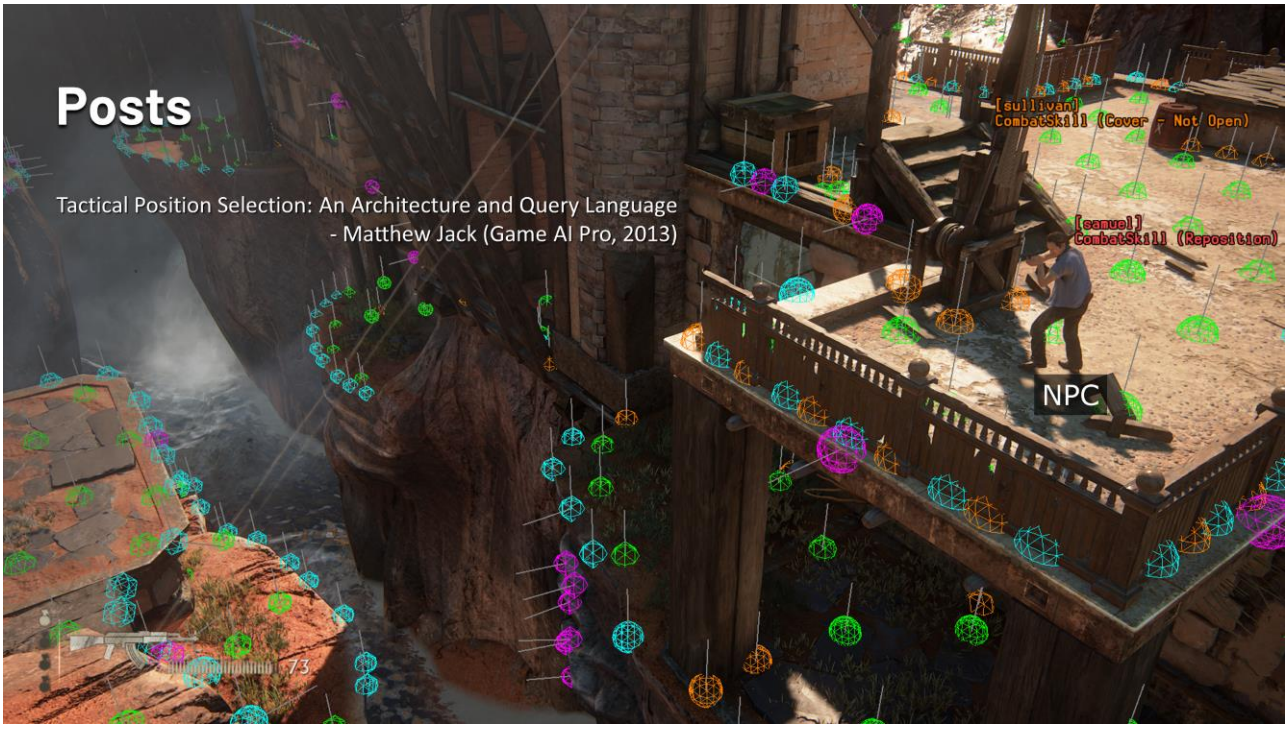
At Naughty Dog, we've encountered this problem the most frequently in AI, specifically where an NPC should go, so that will be the focus of this talk.
But the same principles can be applied to anywhere in gameplay.

In this talk, I will go over the evolution of our problem space and solutions in our recent games.

First, let’s look at a video of an NPC shooting at the player. When the player breaks line of sight, the NPC will try to reestablish line of sight in order to shoot at the player.

Now, let’s look at an NPC investigating a thrown bottle. They perform multiple corner checks around the broken bottle.

Before picking a good spot for an NPC to go to, we need to first determine where those spots are.



At Naughty Dog, we call them posts.

Posts can be tool-generated, manually placed in tools, or dynamically generated at load-time or run-time.

We try not to generate too many posts at run-time for performance reasons. We used to generate most posts at load-time and has since transitioned to generating them at tools-time.

We call our system the Post System.

If you've used Unreal, you might be familiar with Unreal's Environment Query System (a.k.a. EQS). The purposes of both systems are very similar.

A quick shout-out to Matthew Jack's article in Game AI Pro, Tactical Position Selection: An Architecture and Query Language. I inherited our post system and wasn't fully aware of its origins. I was made aware of Matthew's article by David

Rogers from PlayStation London Studio only just two days ago, and it appears that some of the core ideas and terminologies from our system most likely originated from the article. So I added this acknowledgement just in time, whew.

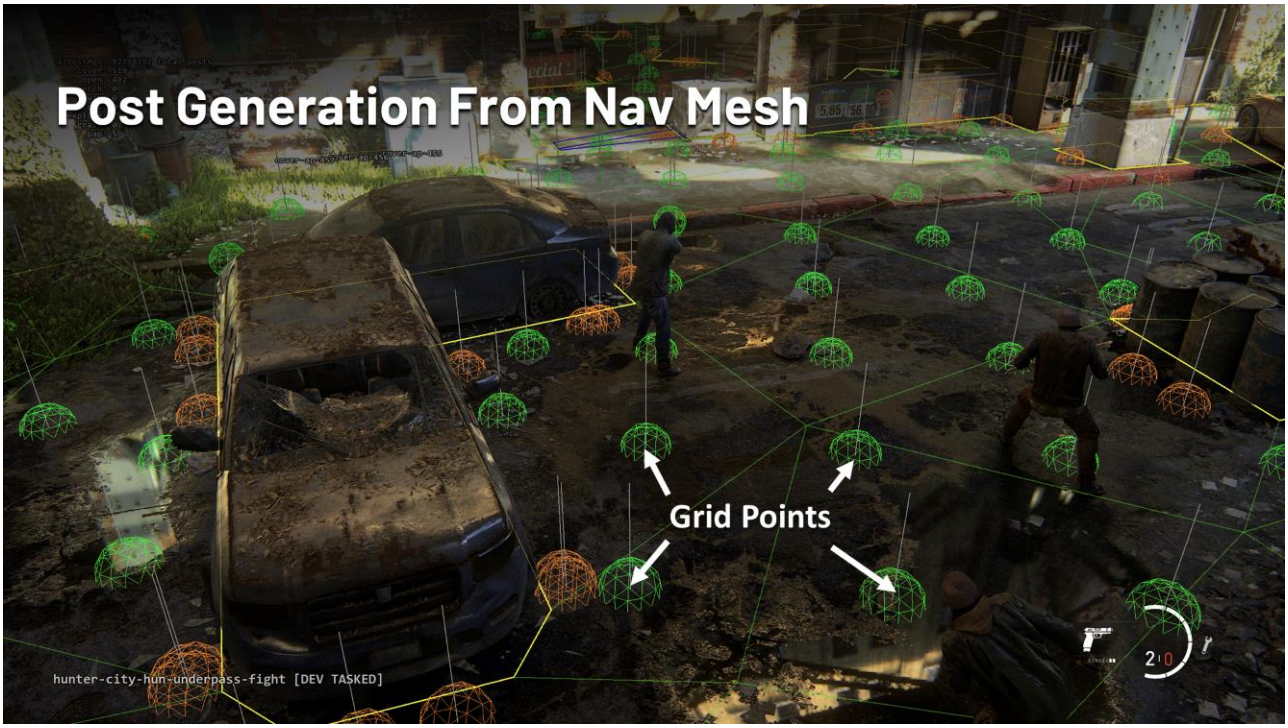
Post Generation

Post Type	Source
Open	Nav Mesh
Swim	Nav Mesh
Hang	Nav Ledge
Cover	Cover AP
Perch	Perch AP

Here are a few examples of types of posts generated from tools: open, cover, hang, and perch.

These posts are generated from nav data or hand-placed markups at tools-time.

Here AP stands for action packs. They are generated from level collision or markups placed by designers in tools.

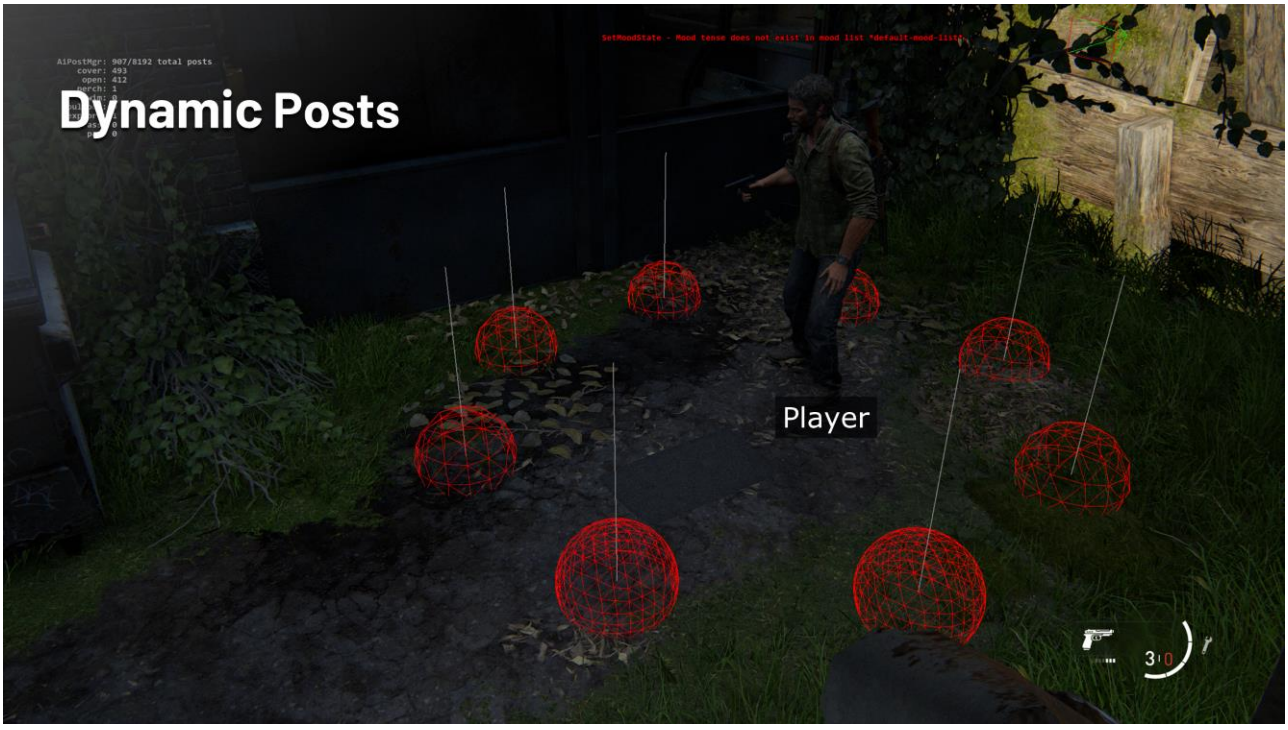


Nav meshes are our main source of post generation. Let’s take a deeper look.

Open posts are generated at grid points that lie on nav meshes.

Since this happens at tools-time, we didn’t use any fancy optimization and simply used a straightforward implementation.

For each nav mesh, iterate over all the grid points contained by its AABB. If a grid point is contained by a nav poly, then a post is generated there.



As mentioned before, posts can also be dynamically generated at run-time. Here we can see posts generated around the player for buddies to go to.

NOTE: the tiny debug draw text in screenshots are not meant to be read, so there’s no need to squint your eyes.

Here I just need you to pay attention to the position of the blue text. That is the post the NPC has chosen to go to.

But how do we pick a good post for NPCs to go to?

We use post ratings.

Each post is scored using a list of post criteria. Each criterion produces a normalized score between 0 and 1. Then each score is multiplied together to produce a final score.

If a post gets a zero score, then it is rejected. The post with the highest score is chosen.

Commonly-Used Criteria

- **Linear/Euclidean distance (to targets, allies, players, distractions, etc.)**
- **Path distance**
- **Visibility**
- **Time since last exposed**

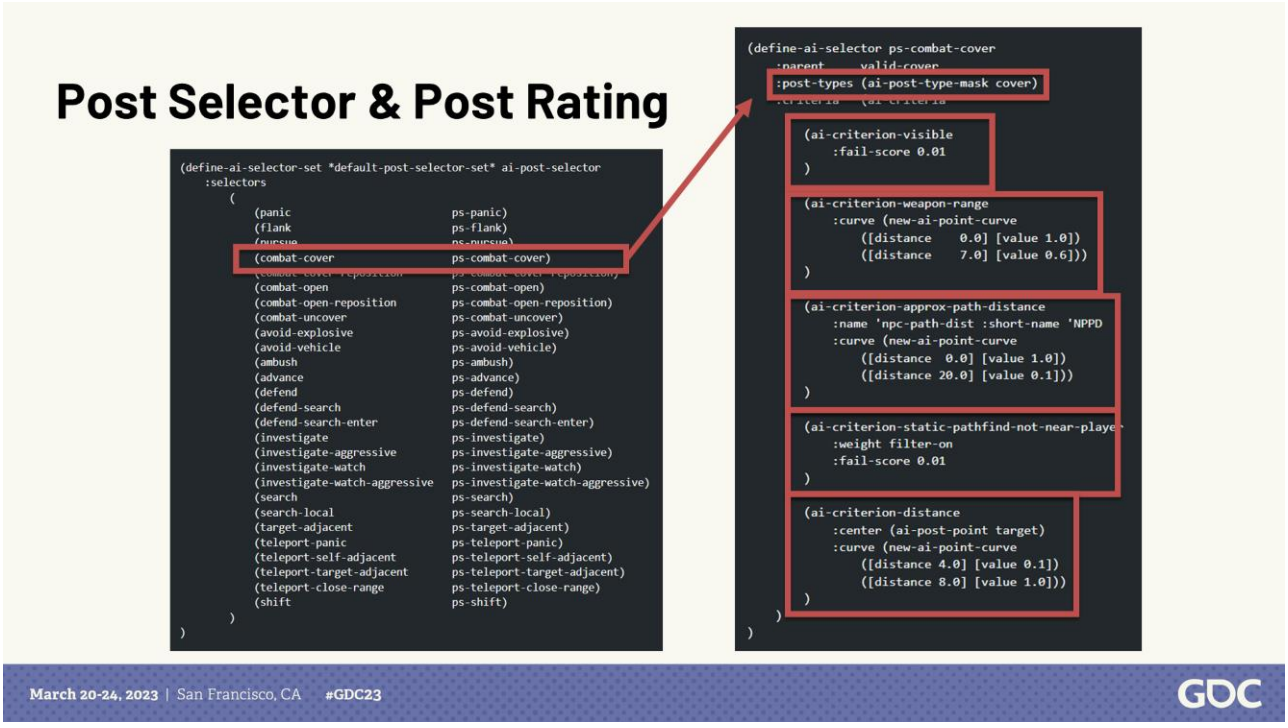
Here are some commonly-used criteria for scoring posts.

The most common one in our games is the linear distance. It can be the distance to targets, allies, players, distractions, or whatever points of interest.

Then we have path distance, or the path length, which is different from linear distance. For example, the target might be right on the other side of a fence an NPC with a gun can easily shoot at, but an NPC with a melee weapon might have to take a long path around to get to the target.

The visibility is useful for preferring posts that have line-of-sight to the target to shoot at the target.

Time since last exposed is useful for investigation or search, so NPCs prefer not to expose a recently-exposed area again.



We group different lists of criteria into post selectors.

Here is how we define a post selector mapping and each post selector's list of criteria in scripts.

Here we use the combat cover post selector as an example. It only considers cover posts and rates posts based on visibility, distance compared to weapon range, path distance from NPC, whether path leading to posts go anywhere near the player, and distance from the target. The curves for each criterion map values to normalized scores between 0 and 1.

Brute-Force Implementation

```
for (AiPost post : allPosts)
{
    for (AiCriterion criterion : allCriteria)
    {
        criterionScore = Evaluate(post, criterion);
        AddScore(post, criterionScore);
    }
}
```


Here's a very straightforward brute-force implementation of picking the best-rated post. We loop over all posts. For each post, we loop over all criteria and score the post with it.

I know here the code says AddScore, but what it really does it multiplying the scores.

If we don't have a lot of total posts or complicated criteria, then this approach would work.

Based on word of mouth in the company, this is how it was done in the very first Uncharted, which was before my time at the studio. This was good enough, because in Uncharted the NPCs only needed to score cover posts, and there weren't that many cover posts in each encounter; also, the scoring rules were very simple and weren't expensive to compute.

But we do have a lot of posts and non-trivial criteria in later

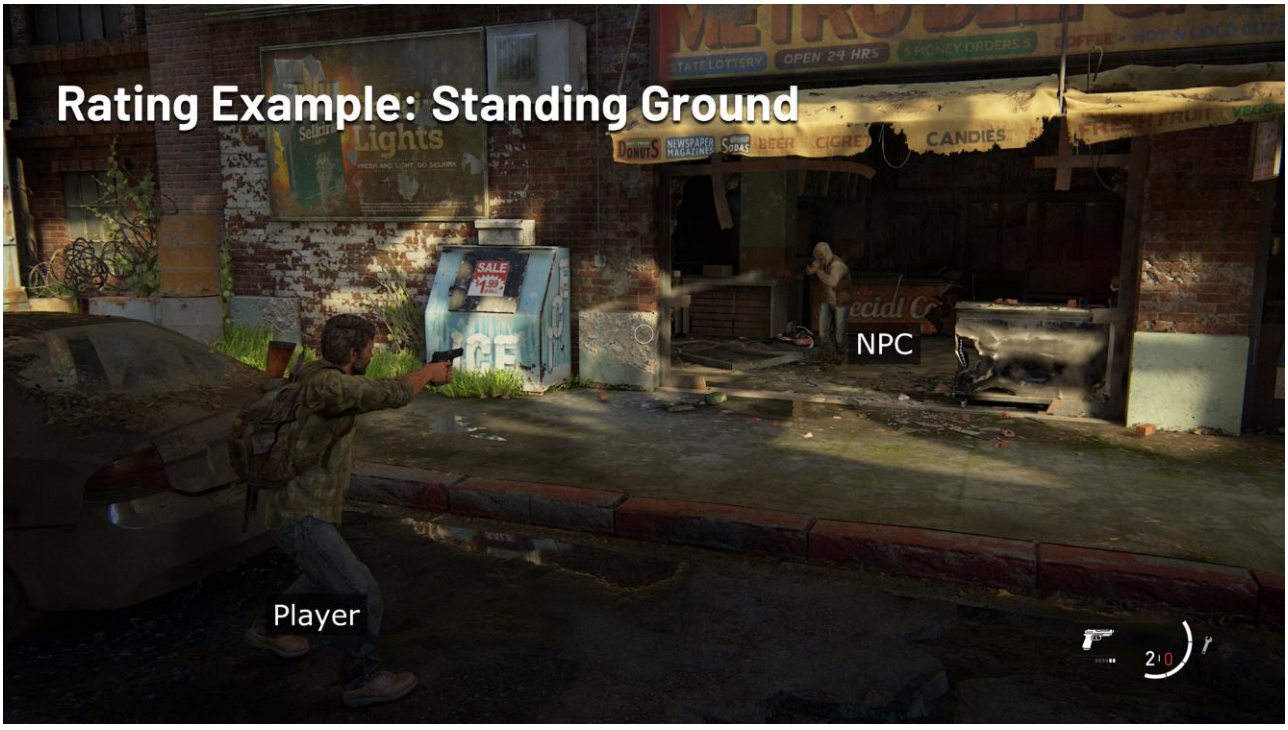
games, so optimization is needed.

Quick Optimization: Early-Out

```
for (AiPost post : allPosts)
{
    for (AiCriterion criterion : allCriteria)
    {
        criterionScore = Evaluate(post, criterion);
        if (criterionScore <= 0.0f)
        {
            RejectPost(post);
            continue;
        }

        AddScore(post, criterionScore);
    }
}
```

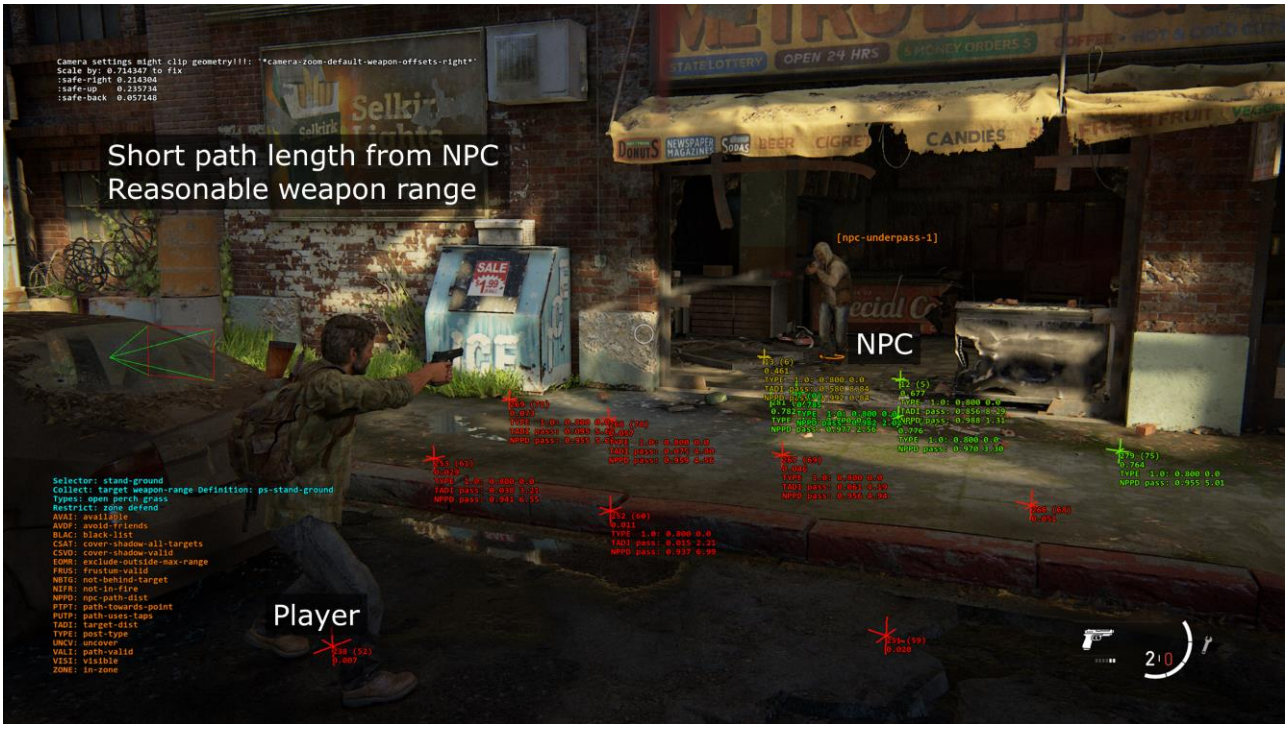

The first obvious and easy thing to do is early-out.

We order the criteria by how expensive they are. Then we early-out on the first criterion evaluation that gives the post a zero score and reject the post.

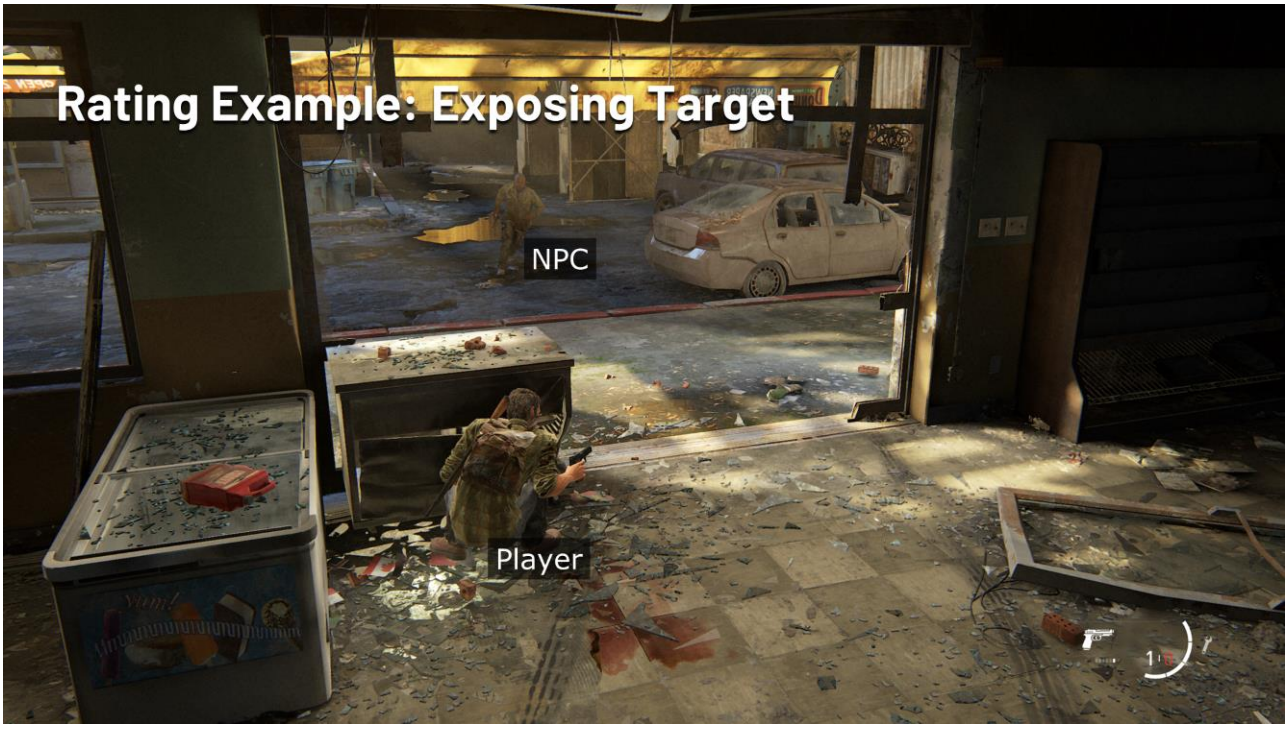


Before looking at more advanced optimization, let’s first look at some rating examples.

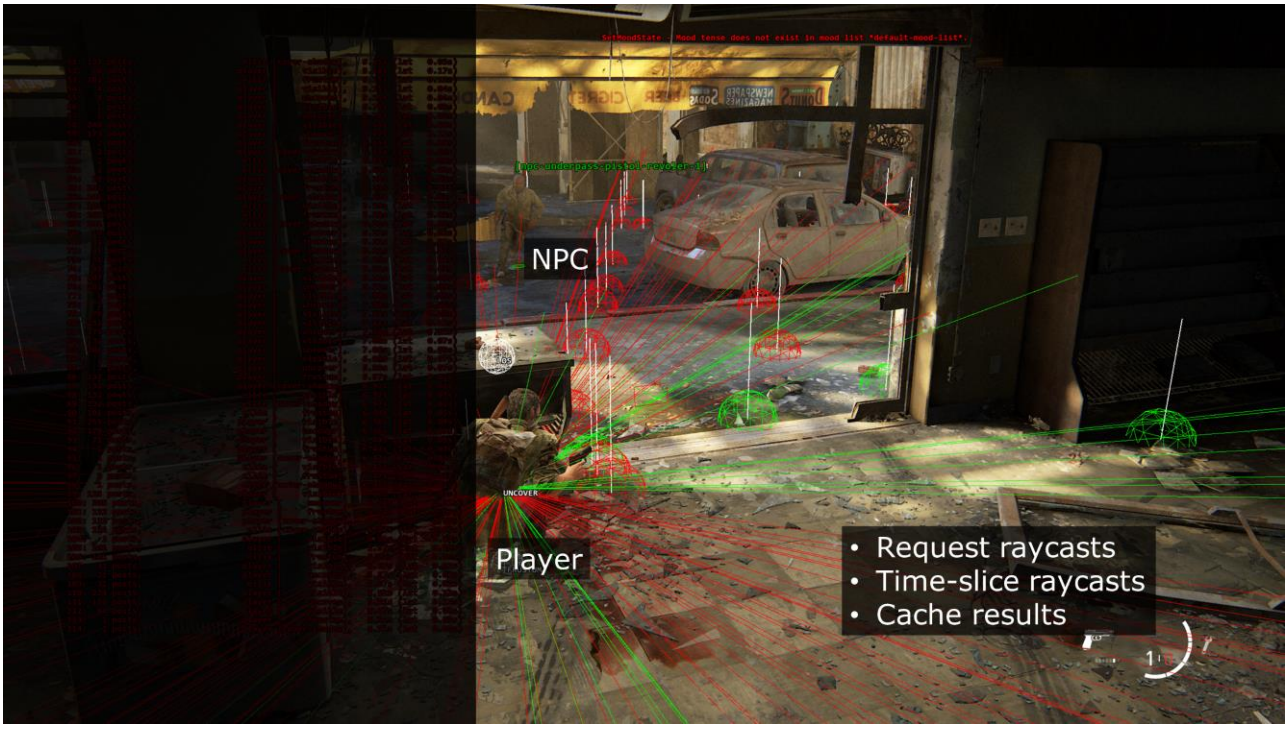
Here we want the NPC to find a good open post to stand his ground.



The criteria we used include preferring shorter path distances from the NPC to the post, as well as preferring posts within weapon range (there's a maximum distance and a minimum distance so the NPC doesn't end up at ridiculous distance unfit for their weapon). Here you can see the ratings are color-coded. Greener ratings mean higher scores, and redder ratings mean lower scores.



Next, here’s an example of an NPC picking a post to expose the player behind the cover to establish line of sight for shooting.



We are not showing ratings here but intermedia ray cast results used for rating posts based on visibility.

Green posts mean they provide line of sight to expose and shoot the player.

Since ray casts are not cheap, we don’t perform them synchronously during post rating. Instead, we first request the necessary raycasts for posts, time-slice the raycasts over multiple frames, cache results for a short while, and evaluate raycast-based criteria once results for all raycasts are cached. Multiple NPCs can share and reuse the post raycast results.

Raycast Result Caching

- **Post sets: bit arrays (8K max posts in TLoU Part 1; 64-bit ints x 128)**
- **Raycast result cache entry: Ray-casted posts + visibility results (2 bit arrays)**
- **Result query: Bitwise-OR of cached post bits**
  If a superset of requested post bits → cache hit
  Otherwise → cache miss
- **Request raycast: Final requested post bits = Needed posts bits – cached post bits**

Let's talk more about post raycast result caching.

We represent a set of posts as a bit array. If bit number 3 is set, post number 3 is in the set. We support up to 8K posts in The Last of Us; that is 128 64-bit integers in each bit array. Far from a performance bottleneck.

Every time a batch of raycast result returns, we cache a result entry consisted of 2 bit arrays: one representing the set of posts being requested for raycasts, and one representing the binary raycast result (hit or clear).

When querying a raycast result, we go through the cache entries that have not expired and bitwise-OR their post bits together, until either 1) the bitwise-OR becomes a superset of the requested post bits, meaning a cache hit, or 2) the cache entries have been exhausted, meaning a cache miss.

Upon a cache miss, we request raycasts for post bits equal to the needed post bits minus the bitwise-OR of post bits from the cached results that would still be valid later.

The third example is the video we saw earlier. This NPC investigates a thrown bottle.

The initial post needs to be a cover post near the throw bottle, and the cover must have line of sight to the bottle.

The subsequent post doesn’t have the line-of-sight requirement but is biased towards corners that haven’t been seen by any NPCs in a while, and it has to be a minimum distance away from the first post.

These two post selectors make the NPC perform two corner checks in quick succession, the first one uncovering the bottle, and the second one making them uncover additional corners nearby that haven’t been checked in a while.

Now that’s go back to optimization.

We still haven’t solved the problem with lots of posts.

Here is an NPC trying to see if there’s a better combat post than the current one.

Here are all the posts in the area. There are a lot of posts that are valid combat posts, but they are too far away from the NPC and player, so they can be reasonably ignored.

We optimize by cutting down the total numbers passed down to rating using collection volumes. Each post selector can specify one or more collection volumes. Only posts within at least one collection volume will be passed further down for rating. Here we collect posts within a sphere around the NPC and a smaller sphere around the player.

Optimization: Collection Volumes

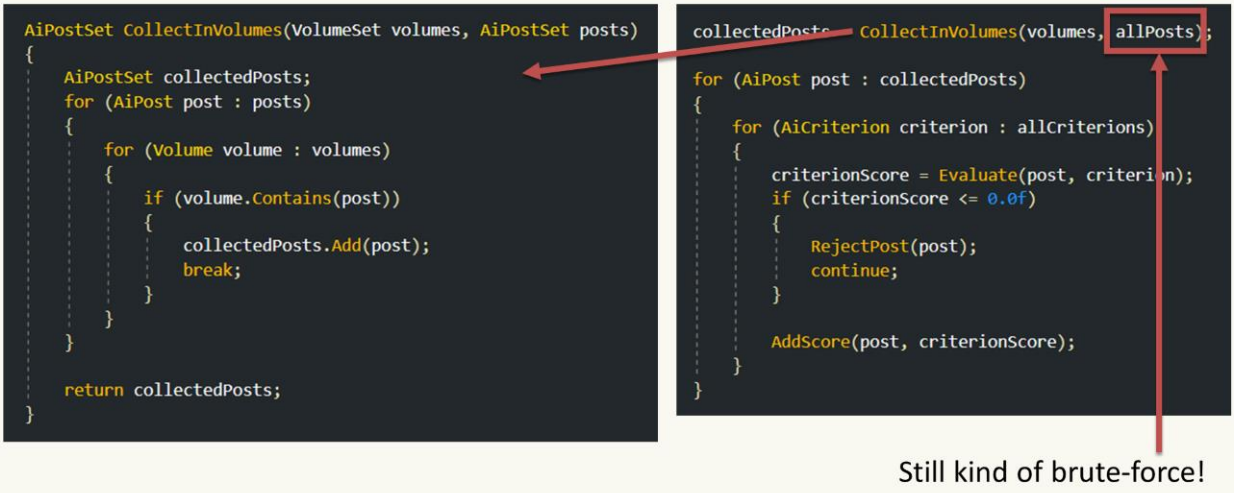
```
(define post-collect-weapon-range
    (new ai-post-collect-def
        :type (ai-post-collect-type weapon-range)
        :center (ai-post-point self)
    )
)

(define-ai-selector ps-combat-cover
    :parent     ps-valid-cover
    :post-types (ai-post-type-mask cover)
    :collect    (post-collect-weapon-range
                 post-collect-self-tight
                 post-collect-target-tight)
    :restrict   (ai-post-restrict-mask zone defend)
    :criteria   (ai-criteria

        (ai-criterion-visible
            :fail-score 0.01
```

```
collectedPosts = CollectInVolumes(volumes, allPosts);

for (AiPost post : collectedPosts)
{
    for (AiCriterion criterion : allCriterions)
    {
        criterionScore = Evaluate(post, criterion);
        if (criterionScore <= 0.0f)
        {
            RejectPost(post);
            continue;
        }

        AddScore(post, criterionScore);
    }
}
```


Here you can see part of an updated combat cover post selector that specifies 3 collection volumes, the weapon range volume is a torus corresponding to the minimum and maximum weapon range, the self-tight volume is a small sphere around the NPC themselves, and the target-tight volume is an even small sphere around the NPC's target.

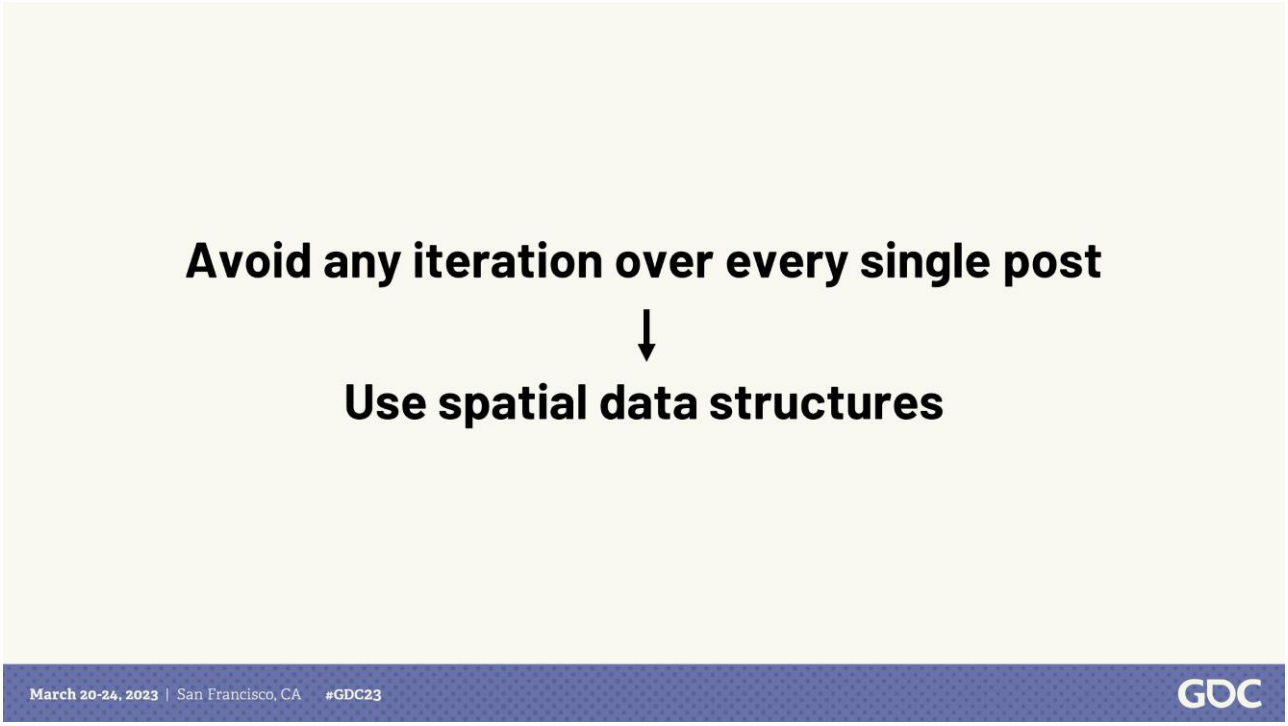
We now collect posts within collection volumes before rating them using the same logic as before.

Optimization: Collection Volumes

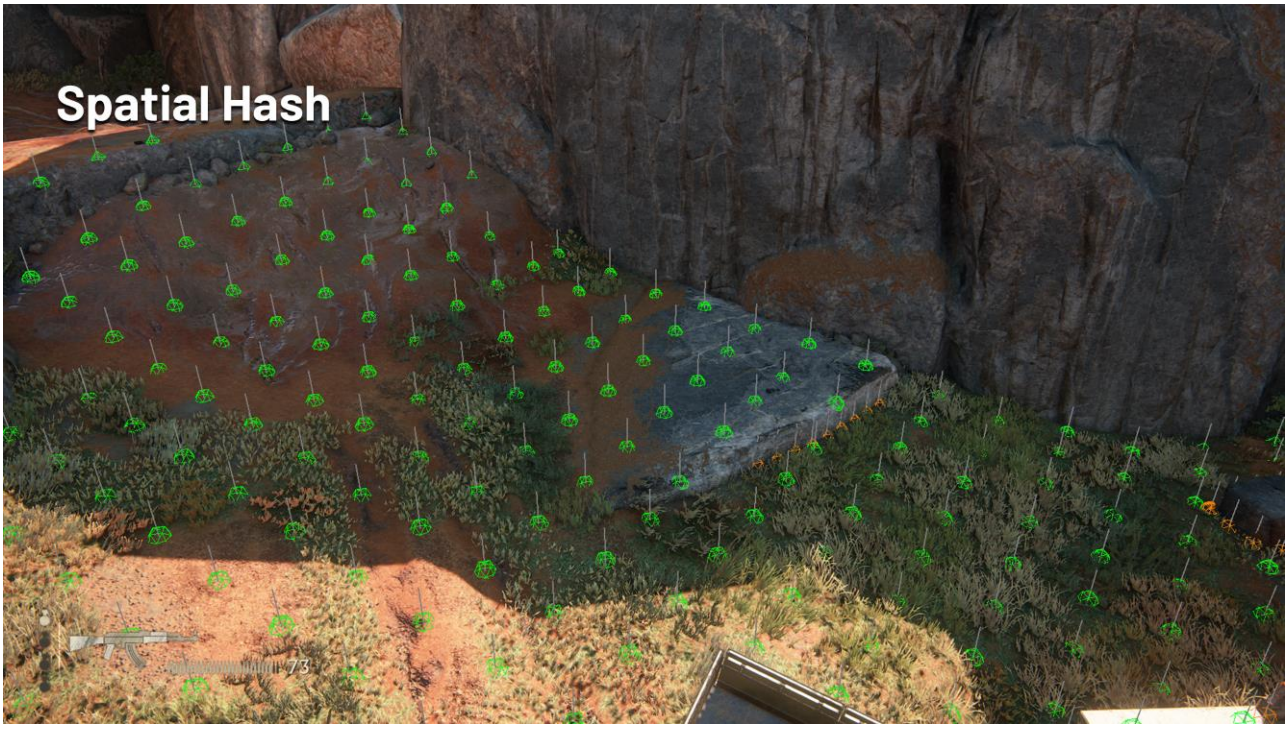


And here's the collection function. We loop through all posts within a post set and check if each post is contained by any collection volume.

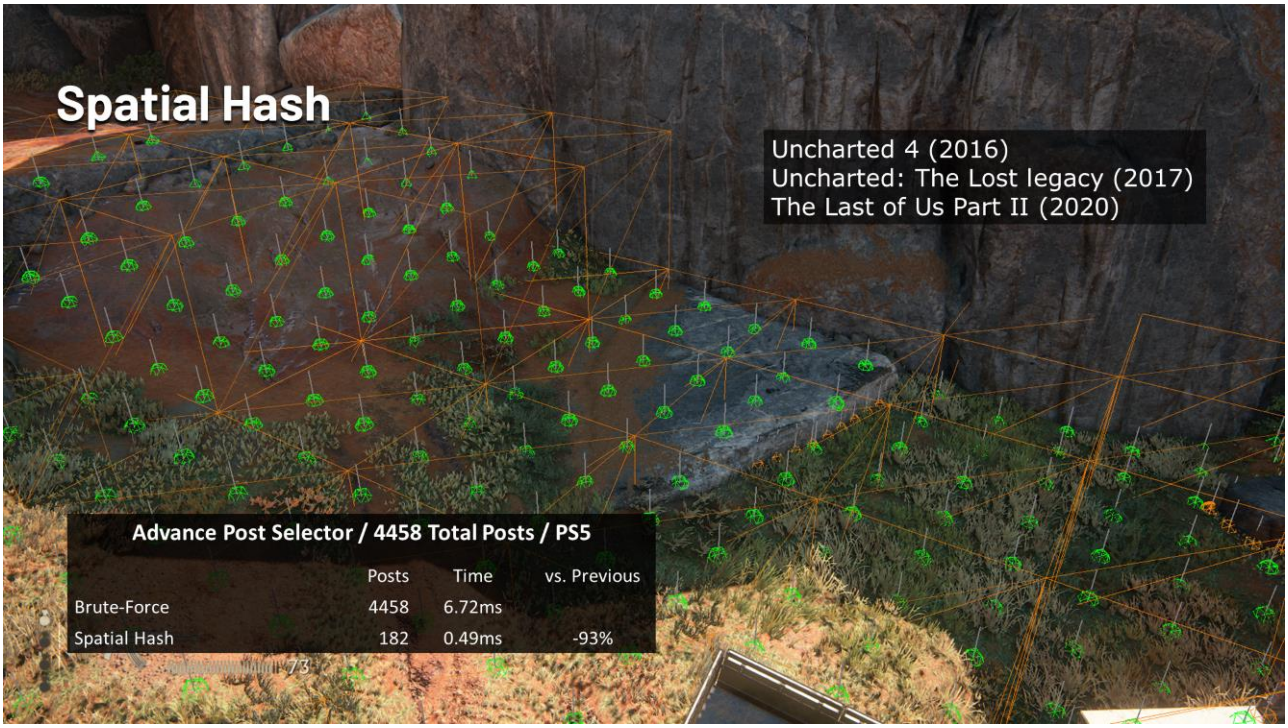
It is an improvement from before, but there is still something brute-force about this implementation. The post set being iterated over for the volume containment test is still all the posts.



We want to avoid any kind of iteration over every single post, so we decided to use spatial data structures to help eliminate that.



First, we tried spatial hash. Each post is hashed to a key that is its quantized coordinates, essentially hashing posts contained within the same grid cell to the same bucket.



This is a debug draw for the grid cells. You can see each cell contains 2 to 3 posts in each axis.

This is the solution we shipped in Uncharted 4, Uncharted: The Lost Legacy, and The Last of Us Part II.

Spatial Hash

```
AiPostSet CollectInVolumes(VolumeSet volumes, AiPostSet posts)
{
    AiPostSet collectedPosts;
    for (AiPost post : posts)
    {
        for (Volume volume : volumes)
        {
            if (volume.Contains(post))
            {
                collectedPosts.Add(post);
                break;
            }
        }
    }

    return collectedPosts;
}
```

(old)

```
AiPostSet CollectInVolumes(VolumeSet volumes)
{
    AiPostSet hashedPosts;
    for (Volume volume : volumes)
    {
        Aabb volumeAabb = volume.GetAabb();
        hashedPosts.Add(GetPostsFromSpatialHash(volumeAabb));
    }

    AiPostSet collectedPosts;
    for (AiPost post : hashedPosts)
    {
        for (Volume volume : volumes)
        {
            if (volume.Contains(post))
            {
                collectedPosts.Add(post);
                break;
            }
        }
    }

    return collectedPosts;
}
```

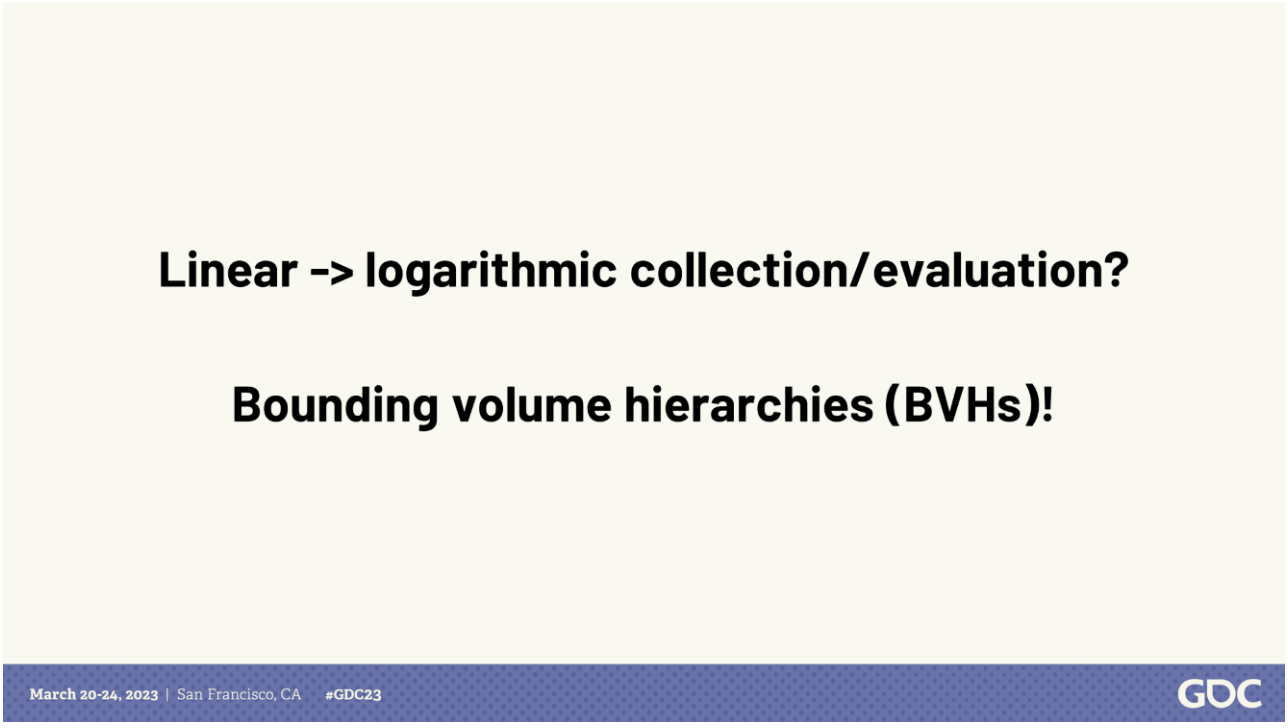

Here's our updated post collection using spatial hash. There is no longer a need to pass in a post set containing all the posts. It is now generated from the spatial hash.

We iterate over each volume, iterate over each grid cell touched by the volume's AABB, and the posts contained in the grid cells that intersect with the AABB of the volume to a post set. We then loop through this post set to perform the same volume containment test as before. You might get better performance by testing each cell with the volume as well, but I think that depends on the use case.

Now we no longer iterate through all posts. Just the ones that are within the grid cells near collection volumes.

This had been our implementation throughout The Last of Us, Uncharted 4, and The Last of Us Part II.

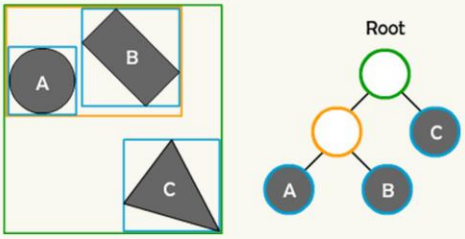
But can we do better? If the collection volumes are large, we can still collect hundreds, if not thousands, of posts, and we'd have to check their types against the selector's filter and rate every single one of them.



As programmers, when we are faced with linear complexity, the next natural thing to ask is if it's possible to make it logarithmic.
Can we reduce the complexity of post collection and evaluation from linear to logarithmic?

The answer is yes! By using bounding volume hierarchies.

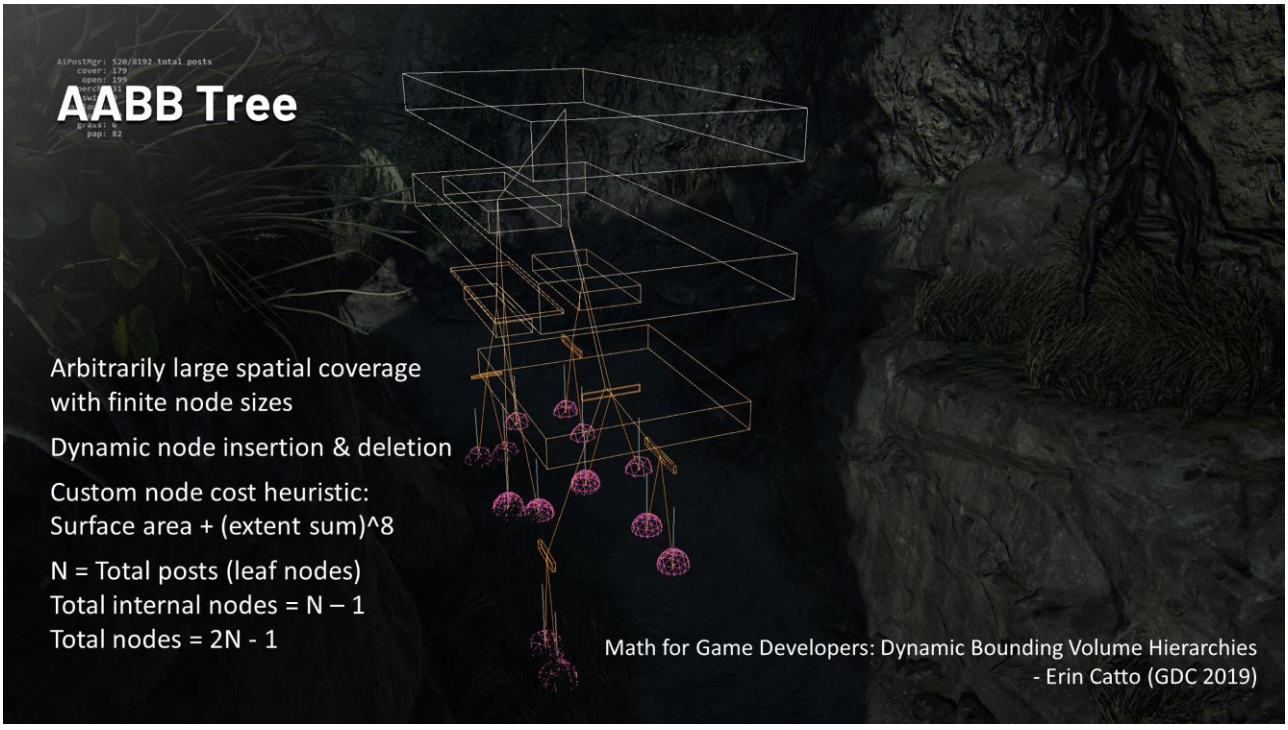
AABB Tree



The structure we chose is AABB tree.

Each node’s AABB is the union of its children’s AABBs.

Here you can see a nice little patch of posts.



And here is its corresponding AABB tree. Each node is given a thickness and offset vertically for visualization.

There are many other tree-based spatial data structures. We picked AABB tree because it lends itself well to arbitrarily large spatial coverage with finite node sizes, as well as dynamic node insertion and deletion. It's also easy to provide custom cost heuristics on how to split nodes.

Our AABB tree implementation is based on a talk by Erin Catto in GDC 2019 (https://box2d.org/files/ErinCatto_DynamicBVH_GDC2019.pdf). A cost heuristic is picked so that minimizing the cost of the entire tree during insertion and removal would result in good average query cost, much like the idea of balancing a binary search tree. But we are not balancing the tree height. We are balancing the average query cost. For example, in Erin Catto's talk, his primary use case was ray casting against the AABB tree for Overwatch, so he used the surface area of each node as the cost. This is good for ray casting, but not so great for

collection volumes, because extremely elongated AABBs can have the same surface area as a mostly cube AABB. So our cost heuristic is chosen, through trial & error, to be the AABB's surface area plus a bias term that is the extent sum to the power of 8. This seems to nicely discourage elongated nodes.

In this implementation, the trees are always full, so the total number of internal nodes is total number of posts minus one. Posts are leaf nodes, so the total number of nodes is total number of posts times two minus one.

AABB Tree

```
AiPostSet CollectInVolumes(VolumeSet volumes, AiPostNode root)
{
    AiPostSet collectedPosts;
    for (Volume volume : volumes)
    {
        Visit(root, volume, collectedPosts);
    }

    return collectedPosts;
}
```

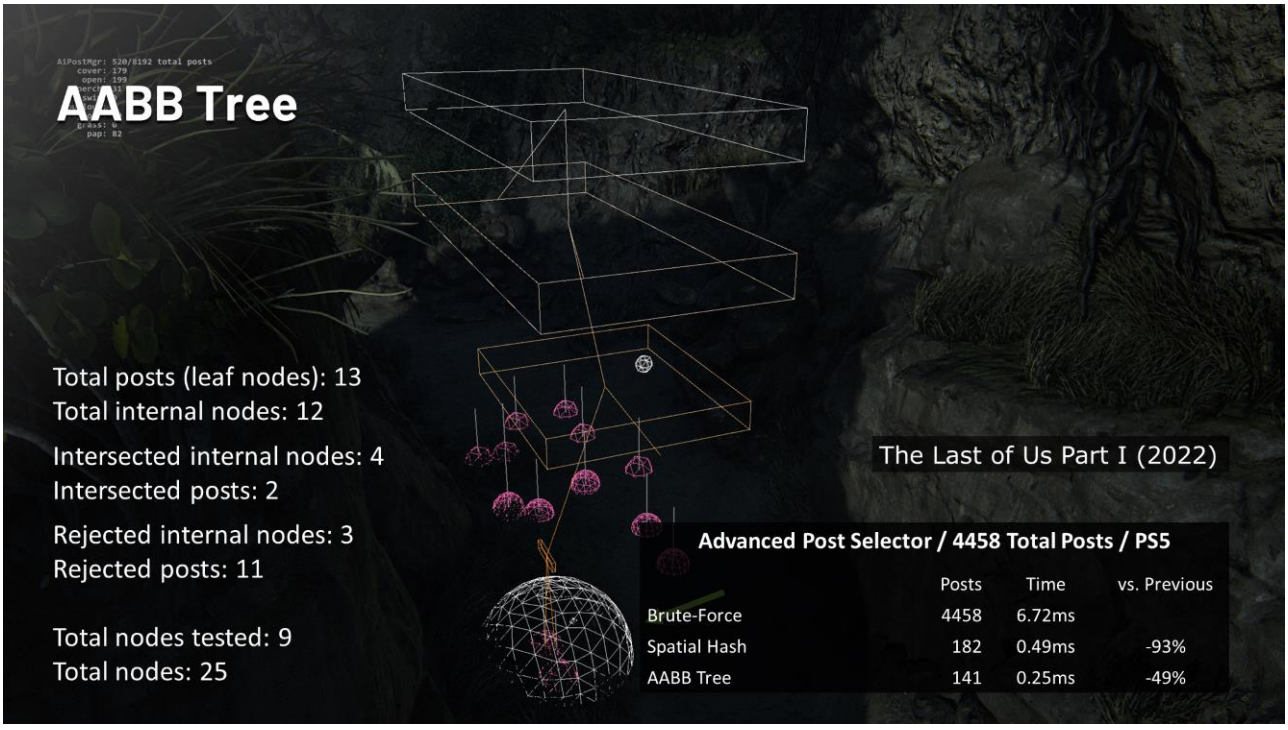
```
void Visit(AiPostNode node, Volume volume, AiPostSet& collectedPosts)
{
    if (!volume.Intersects(node.GetAabb()))
        return;

    if (node.IsInternal())
    {
        Visit(node.GetChildA(), volume, collectedPosts);
        Visit(node.GetChildB(), volume, collectedPosts);
    }
    else
    {
        collectedPosts.Add(node.GetPost());
    }
}
```


This is what our new collection function looks like. We first start with visiting the root node. And then we recursively visit each node's children if the node's AABB intersects with the collection volume.

Then we asked ourselves if we can make the pruning of tree traversal even more aggressive.

For example, if we propagate a bitwise-OR'd bitmask up the tree from the leaf nodes that indicates what types of posts an internal node contains, we can early-out on an internal node that doesn't contain any cover posts if we want to collect cover posts.

Also, nearby posts might have similar properties and are likely to be within the same ancestor node. We should be able to reject them as a whole if we can make sure that that they will all be rejected by a certain criterion.



Here we have a little collection sphere volume. Only the nodes whose AABBs intersect with the sphere are drawn. Remember that nodes are vertically offset for visualization.

This is the solution we shipped in The Last of Us Part I.

AABB Tree

```
AiPostSet CollectInVolumes(VolumeSet volumes, AiPostNode root)
{
    AiPostSet collectedPosts;
    for (Volume volume : volumes)
    {
        Visit(root, volume, collectedPosts);
    }

    return collectedPosts;
}
```

```
void Visit(AiPostNode node, Volume volume, AiPostSet& collectedPosts)
{
    if (!volume.Intersects(node.GetAabb()))
        return;

    if (node.IsInternal())
    {
        Visit(node.GetChildA(), volume, collectedPosts);
        Visit(node.GetChildB(), volume, collectedPosts);
    }
    else
    {
        collectedPosts.Add(node.GetPost());
    }
}
```

Can we make the pruning more aggressive?

Let’s look at the code again.

Here is where we prune visited nodes during collection.

Then we asked ourselves if we can make the pruning of tree traversal even more aggressive.

Broad Phase

```
AiPostSet CollectInVolumes(VolumeSet volumes, AiPostNode root)
{
    AiPostSet collectedPosts;
    for (Volume volume : volumes)
    {
        Visit(root, volume, collectedPosts);
    }

    return collectedPosts;
}
```

```
void Visit(AiPostNode node, Volume volume, AiPostSet& collectedPosts)
{
    if (!volume.Intersects(node.GetAabb()))
        return;

    if (node.IsInternal())
    {
        for (BroadPhase broadPhase : allBroadPhases)
        {
            if (!Evaluate(broadPhase, node))
                return;
        }

        Visit(node.GetChildA(), volume, collectedPosts);
        Visit(node.GetChildB(), volume, collectedPosts);
    }
    else
    {
        collectedPosts.Add(node.GetPost());
    }
}
```


As it turns out, it’s not too hard to come up corresponding algorithms for some criteria that work on internal AABB tree nodes.

We call them broad phases. This is a term borrowed from physics engines for logic that rejects groups of objects as a whole before processing individual objects.

We test internal nodes against broad phases and potentially reject them before visiting their children.

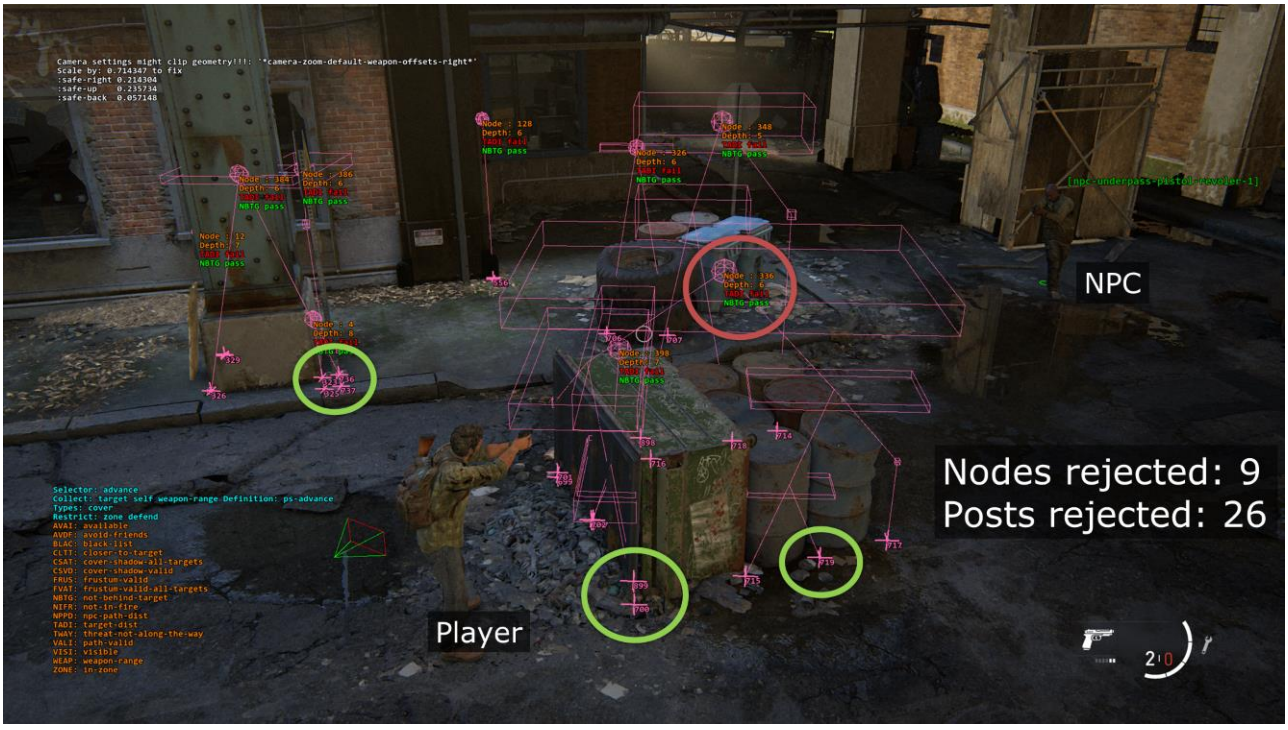
Broad Phase Examples

Post Criterion (Leaf Node)	Broad Phase (Internal Node)
Reject post if distance between target and post < threshold	Reject node if distance between NPC and **furthest** point in node's AABB < threshold
Reject post if post is not in stealth grass	Reject node if **none** of the underlying posts is in stealth grass
Reject post if post is behind target	Reject node if node's AABB is **entirely** behind target

Here are some examples of post criteria and their corresponding broad phases.

Binary checks on internal nodes are done in the same way as post type checks shown earlier, by propagating bitwise-OR'd flags from leaf nodes.

Let’s look at a broad phase example where we want to collect posts that are not near the player.

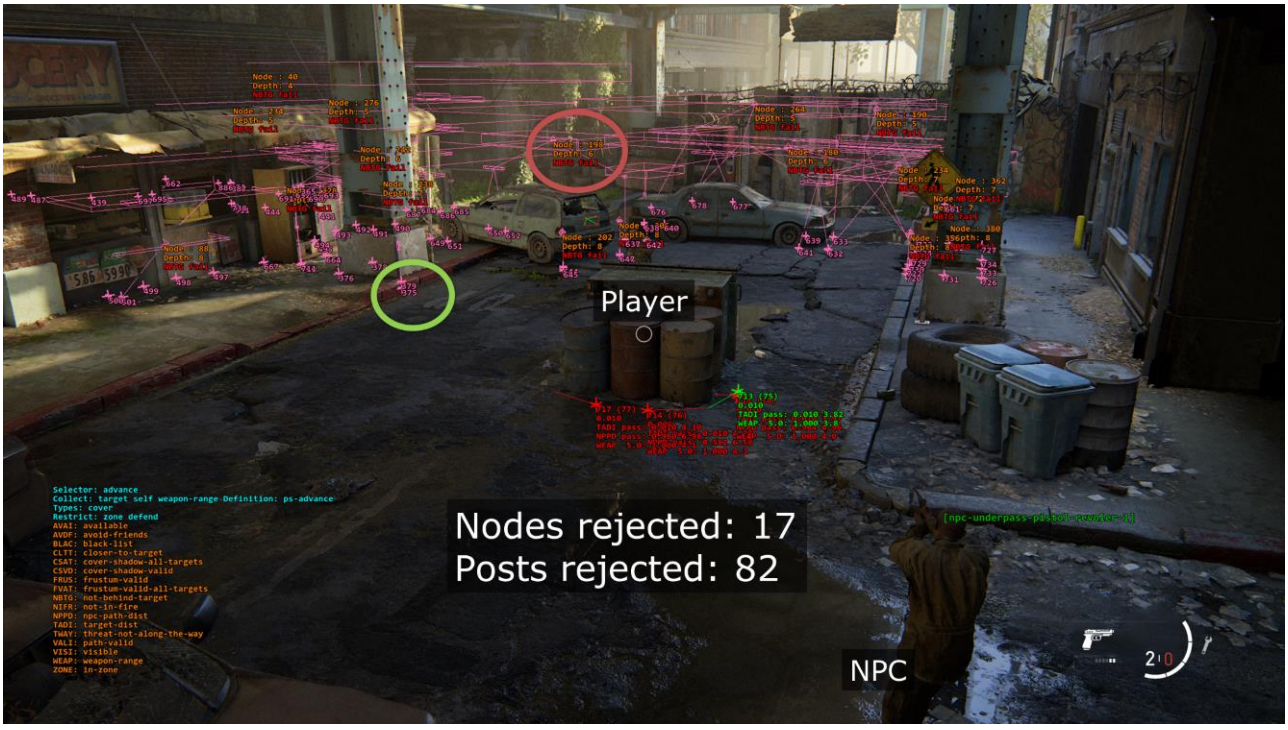


The pink numbered crosses are posts rejected due to the distance check broad phase. The red “TADI fail” texts mark internal nodes rejected by the broad phase.

We are rejecting posts within a certain distance of the player, so the corresponding broad phase is rejecting internal nodes whose furthest point within the node’s AABB is still within that distance of the player.

There are a total of 26 posts rejected, and that’s a result of rejecting just 9 internal nodes. We not only skipped the distance evaluation for these posts, but we also skipped all the cheaper criteria that would have been evaluated before the distance criterion if we had collected all these posts.

The second example is collecting posts that are not behind player, i.e., rejecting posts behind the player.



Again, the pink numbered crosses are rejected nodes, and the red “NBTG fail” texts mark internal nodes rejected by the broad phase.

We are rejecting posts behind the player, so the corresponding broad phase is rejecting internal nodes entirely behind the player.

- There are a total of 82 posts rejected, and that’s the result of rejecting just 17 internal nodes.
- Overall, switching from spatial hash over to AABB tree gave us a performance boost of cutting CPU time in half!

After looking at all the various methods of post collection, let's look at their performance stats.

We profiled the advance selector to advance NPCs towards the player, with a total of 4458 posts in the level. This is running on a PS5.

The brute-force approach iterated through and evaluated every single post; it took 6.72ms.

Spatial hash significantly cut down the number of posts down to 182; the evaluation took 0.49ms, about a 93% reduction from brute-force.

AABB tree without broad phase further reduced the collected posts to 141; the evaluation took 0.25ms, further reducing the CPU time by almost half.

Finally, AABB tree with broad phases (for player distance, not behind player, and not behind NPC) further cut down the number of posts by more than half, reducing the time by

another 20%.

There could be use cases where spatial hash is more efficient than AABB trees, but in our tests AABB trees proved to be performing better.

This presentation mostly focused on the AI applications of posts, since those were our initial use cases. But as mentioned at the beginning, and like Unreal's EQS, it doesn't have to be limited to AI applications. Any problem that requires picking of good spots can take advantage of posts. For example, in Unchrated 4 multiplayer, we used post selectors to choose where to spawn minions during boss fights. Also in a multiplayer setting, we can potentially use posts to pick a good spot around a player to respawn a teammate that is both away from threats and outside of any player's view.

Future Research

- **Much larger levels: 8K posts -> hundreds of thousands of posts (plus more NPCs)**
- **Sparse post sets: Maps of nav mesh IDs to local/smaller dense post sets**
- **Add/remove nav mesh tree nodes based on player proximity.**
- **More efficient raycast result caching.**

Finally, a little discussion of our future research on posts.

We have announced our next project to be a multiplayer game, and we are aiming for much larger levels.

We expanded our max number of posts from 8K to hundreds of thousands of posts, and the bit array approach quickly proved to not scale up well. Suddenly the bitwise operations became a performance bottleneck. This was also due to the fact that we have more NPCs. More NPCs mean more post evaluation and post raycast requests.

We have removed the number of max posts from being a performance bottleneck by implementing sparse bit arrays, which are maps of nav mesh IDs to local post bit arrays. There are no more global post sets that are huge bit arrays that account for all posts. The downside is that now each post set has a limit on the number of local post bit arrays it can contain, but that hasn’t become a problem for us yet. We can

always pick a safe number as our limit for each use case. For example, a post set capable of containing 32 local post bit arrays is more than enough for collecting posts in volumes that can hardly span 8 nav meshes.

We left out a tiny detail from our AABB tree-based post collection. There actually isn't a master post tree. Instead, each nav mesh has its own local post tree. Each nav mesh is inserted into a master nav mesh tree. We query this nav mesh tree first, and then query the post trees of the collected nav meshes. We are researching if actively adding removing nav mesh tree nodes can improve performance by reducing the query time on the nav mesh tree. We support more than 5K nav meshes, so if the players are only ever near less than a hundred nav meshes, the reduction in tree size might worth the extra overhead of monitoring player proximity to nav meshes and tree manipulation.

We have not looked into this yet, but we might be able to find a more efficient way to cache post raycast results now that the underlying post set representation has changed significantly.

Summary

- **Post generation: tools-time, load-time, run-time**
- **Post collection & evaluation:**
  - **Brute-force**
  - **Early-out**
  - **Collection volume**
  - **Spatial hash**
  - **BVH: AABB tree**
  - **Broad phase**
- **Raycast result caching with post sets**
- **Dense & sparse post sets**

Here's the summary of what we just went over:

- How we can generate posts at tools-time, load-time, and run-time.

- The evolution and optimizations of our post system from the brute-force approach to AABB trees with broad phases

- How we cache raycast results and how we use post sets to look for cache hits.

- Our post sets initially represented by dense sparse bit arrays and now sparse bit arrays.

This is the end of the talk. I hope you find it useful.

We are hiring. If you are interested, please visit our website.

The bottom left are my website and Twitter handle.

Thank you all very much for coming.

Q&A


AllenChou.net


@TheAllenChou