Hi, and thanks for coming to my talk. Today I want to talk about object discovery in virtual worlds. It'll take a bit before I get to what object discovery is and why we care about it. I want to start out with just describing virtual worlds and the type we're looking at.
metaverses

are user-generated worlds

-all sorts of applications – games, virtual galleries, music performances
metaverses

are user-generated worlds

– all sorts of applications – games, virtual galleries, music performances
Metaverses are user-generated worlds – all sorts of applications – games, virtual galleries, music performances.
Metaverses are user-generated worlds – all sorts of applications – games, virtual galleries, music performances
metaverses are user-generated worlds

—all sorts of applications — games, virtual galleries, music performances
metaverses

are user-generated worlds

—all sorts of applications – games, virtual galleries, music performances
Metaverses are user-generated worlds — all sorts of applications — games, virtual galleries, music performances.
There are many steps involved in getting the world rendered on screen, but the first happens when a client connects to the system. Object discovery is the service that tells you about other objects you might care about: it tells clients what they should render, and scripted objects about other objects they might want to interact with.
Object Discovery

which objects should I know about? display?

There are many steps involved in getting the world rendered on screen, but the first happens when a client connects to the system. Object discovery is the service that tells you about other objects you might care about: it tells clients what they should render, and scripted objects about other objects they might want to interact with.
Object Discovery

which objects should I know about?

display?

interact with?

There are many steps involved in getting the world rendered on screen, but the first happens when a client connects to the system. Object discovery is the service that tells you about other objects you might care about: it tells clients what they should render, and scripted objects about other objects they might want to interact with.
There are many steps involved in getting the world rendered on screen, but the first happens when a client connects to the system. Object discovery is the service that tells you about other objects you might care about: it tells clients what they should render, and scripted objects about other objects they might want to interact with.
Object Discovery

Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
Object Discovery

Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
Object Discovery

Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
Object Discovery

Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
Abstractly, this service is simple. When a client (or object) connects to a server in the system, it registers a query. This could be either explicit, where the query is requested with specific parameters, or implicit, where the server decides the parameters for the client and the client just sees the results. The output of the query should be a set of object identifiers. In practice, we also include some other information, including position and mesh, so the client can render everything.

Object discovery is a type of service that appears in many distributed object systems. In metaverses, and virtual worlds or simulation systems more generally, these queries are always geometric. They are also continuous, meaning that we receive an initial set of results followed by a stream of updates to keep the results up to date. For example, when an object is added to the world, we just receive one update rather than reevaluating the query and getting the entire set back with the new object included. And, of course, we care about large worlds where the system consists of many servers, so queries are distributed across all these servers.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance d. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance d. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance $d$. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance d. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance $d$. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance d. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
Unfortunately, metaverses today don't even come close to what's been described in fiction. Clients for metaverses can't display everything, so they have to choose some subset of objects. The problem isn't just display from one vantage point. If I take a few steps from this position...
... a lot of content pops in that wasn't there before. One of the first things we want to do -- join the world and look around for something interesting -- is crippled or impossible in these systems. And similar problems appear throughout the system, beyond just what to display; for example, we're also limited in how we can communicate with objects, making long distance communication effectively useless. The inability to find things is so bad that we have to break the immersive experience by switching to an external 2d map tool in order to find things to do.
Scalability/Distributed

User Generated

Less

More

One Server

Many Servers

Visual Fidelity

Low

High

Minecraft

The Sims

Half-Life/Counter-Strike

Neverwinter Nights

Multiplayer Game

WoW

EVE Online

The Sims, Half-Life/Counter-Strike, Neverwinter Nights, Multiplayer Game, EVE Online, WoW.
What we really need to do is render a complete view of the world. But this is fundamentally at odds with the desire to scale to large, sprawling, naturally growing worlds. No matter the resources we have available at our client, now or in 5 years, the worlds will grow in size, complexity, and detail: trying to process the entire world at the client is never going to be feasible.
what type of query provides an immersive experience?
only a global query can provide an immersive experience

So what's a better choice of query? If we could implement any query, what type would give us a more immersive experience while still limiting the number of objects we have to render?
Our solution to this problem for clients is solid angle queries. Solid angle has a mathematical definition, just an extension of the idea of angle to 3 dimensions, but intuitively it represents how large an object will appear on screen when rendered. This encodes the rules we wanted for displaying large distant objects as well as smaller nearby objects. We’ll see later how we integrate the idea of aggregates with these queries.
Our solution to this problem for clients is solid angle queries. Solid angle has a mathematical definition, just an extension of the idea of angle to 3 dimensions, but intuitively it represents how large an object will appear on screen when rendered. This encodes the rules we wanted for displaying large distant objects as well as smaller nearby objects. We’ll see later how we integrate the idea of aggregates with these queries.
Our solution to this problem for clients is solid angle queries. Solid angle has a mathematical definition, just an extension of the idea of angle to 3 dimensions, but intuitively it represents how large an object will appear on screen when rendered. This encodes the rules we wanted for displaying large distant objects as well as smaller nearby objects. We’ll see later how we integrate the idea of aggregates with these queries.
Our solution to this problem for clients is solid angle queries. Solid angle has a mathematical definition, just an extension of the idea of angle to 3 dimensions, but intuitively it represents how large an object will appear on screen when rendered. This encodes the rules we wanted for displaying large distant objects as well as smaller nearby objects. We'll see later how we integrate the idea of aggregates with these queries.
Our solution to this problem for clients is solid angle queries. Solid angle has a mathematical definition, just an extension of the idea of angle to 3 dimensions, but intuitively it represents how large an object will appear on screen when rendered. This encodes the rules we wanted for displaying large distant objects as well as smaller nearby objects. We’ll see later how we integrate the idea of aggregates with these queries.
To demonstrate the effect of solid angle queries, here's the complete, ideal version of a scene we put together in Sirikata...
and here's what distance query returns, making it appear as if the world just drops off...
And this is what it looks like with solid angle queries, using the same number of objects as the previous slide. It's a big improvement because the distant mountains are now visible. However, we're still missing all those trees. That's because individually they are too small, so the object discovery service doesn't return them. To make sure we don't miss any of these objects, Sirikata also supports returning aggregates, collections of objects which have been combined and simplified.
This allows us to complete the scene. All the trees have been filled in, although at lower quality, by using aggregates instead of individual objects.
Flipping between the ideal and the version with aggregates that Sirikata uses, we can see some differences, but they are very close. This is just a simple scene we've used where we can render the ideal scene, but even at this scale, the need for these new techniques are clear.
Related Work

Builds on existing data structures & algorithms:

- R-Tree
- BVH
- BVH Refitting
- Async BVH
- LOD/Simplification
- Safe Periods/Lazy Updates
- Query Aggregation
- Approximate Queries

But has a unique set of requirements:

<table>
<thead>
<tr>
<th></th>
<th>Distributed</th>
<th>Global View</th>
<th>Aggregation</th>
<th>Dynamic &amp; User Generated</th>
<th>Continuous</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobiEyes</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Nam &amp; Sussman</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GigaVoxels</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLOD</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StarTrack</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Chaudhuri et al.</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

There is a huge amount of related work across multiple fields, including distributed systems, databases, and graphics. I've listed some relevant items here. I don't want to single out anything in particular. Rather I want to point out how we differ from this work. A lot of it is fundamental algorithms work which we actually build on because they provide basic techniques for efficiently evaluating queries. However, the vast majority of the work assumes a single host rather than a distributed systems. There are two categories that are more closely related. The graphics community has tackled interactive rendering of large scale worlds, but are usually concerned with mostly static worlds or worlds which fit on a single host. Distributed databases support distributed query processing, but are only occasionally applied to geometric data (GIS is one example) and don't handle the key trouble maker in our system, aggregation.

None of these systems address all of our requirements: a global query like solid angles that give a complete view of the world/data, operating over systems large enough to require distribution, dynamic user-generated data where objects may enter, leave, and move around the world, and providing continuous queries for online exploration.
So I've described a new type of query. The first task is to figure out how to implement this efficiently on even a single server. I'm going to describe a data structure for efficiently evaluating solid angle queries, how we handle continuous queries since most queriers need to be kept up to date with the latest set of results, and how we handle aggregation to make sure we can display the entire world.
how are geometric queries evaluated efficiently?
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes [transition], shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add X [transition], whose bounding sphere surrounds A and B, Y [transition] to surround C and D, and then Z [transition] to surround both X and Y. Z is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes [transition], shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add $X$ [transition], whose bounding sphere surrounds $A$ and $B$, $Y$ [transition] to surround $C$ and $D$, and then $Z$ [transition] to surround both $X$ and $Y$. $Z$ is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes (transition), shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add $X$ (transition), whose bounding sphere surrounds $A$ and $B$, $Y$ (transition) to surround $C$ and $D$, and then $Z$ (transition) to surround both $X$ and $Y$. $Z$ is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes [transition], shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add X [transition], whose bounding sphere surrounds A and B, Y [transition] to surround C and D, and then Z [transition] to surround both X and Y. Z is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes [transition], shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add X [transition], whose bounding sphere surrounds A and B, Y [transition] to surround C and D, and then Z [transition] to surround both X and Y. Z is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
The core of our solution to this is a novel modification of an existing graphics data structure called the bounding volume hierarchy, or BVH. It’s called a bounding volume hierarchy because we start with a set of objects, shown geometrically on the left, and create leaf nodes [transition], shown on the right, where each node tracks the 3D bounding sphere that surrounds the object. We then build a hierarchy atop them, where each internal node bounds the objects below it: we add X [transition], whose bounding sphere surrounds A and B, Y [transition] to surround C and D, and then Z [transition] to surround both X and Y. Z is the root and bounds the entire scene. Each of these bounding volumes only has to track 2 values: the center and radius, as shown here for the root node.
Bounding Volume Hierarchy

culls many nodes for distance queries
To make query processing efficient, we augment each node with the largest object in the subtree, creating the largest object bounding volume hierarchy, or LBVH.
The final problem we need to address are the objects that wouldn't be included because they appear too small individually. However, they can be visually important when combined, as the figure shows. We use aggregates to be able to include them in the results. The basic idea isn't new, but existing systems rely heavily on precomputing this data, having meshes already available, and even on artists to manually make adjustments to ensure good optimization.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y has 2, simpler trees and the node Z, as the root, contains the entire scene, but simplified. Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
cuts ensure queries see the entire world (and make query updates faster too)

Overall, the LBVH reduces the number of nodes tested, and therefore the cost, of evaluating a query by 75-90% over the corresponding BVH. This modification is really what makes it possible to reasonably evaluate solid angle queries.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
We instead apply the same LBVH data structure to generate aggregates. Each leaf node has a mesh for a single object [transition]. Each internal node of the LBVH represents an aggregate of all leaf nodes below it. We generate an aggregate mesh for each internal node and simplify it so all internal nodes have approximately the same complexity, so the node Y [transition] above C and D contains 2, simpler trees and the node Z [transition], as the root, contains the entire scene, but simplified. [transition] Using the cut for each query we can return aggregates that the cut passes through, indicating they appear big enough to be displayed but their children do not. With aggregates, the entire scene is always displayed, but some is at lower quality.
To avoid redundant work when updating queries, we maintain a cut through the tree, indicating where we finished evaluating the query on the last iteration. To update the query, instead of starting at the root, we just walk and update the cut, moving it down or up the tree. For example if the querier moved, we might update this cut by walking horizontally, [transition] finding that node X does now satisfy the query...
To avoid redundant work when updating queries, we maintain a cut through the tree, indicating where we finished evaluating the query on the last iteration. To update the query, instead of starting at the root, we just walk and update the cut, moving it down or up the tree. For example if the querier moved, we might update this cut by walking horizontally, finding that node X does now satisfy the query...
To avoid redundant work when updating queries, we maintain a cut through the tree, indicating where we finished evaluating the query on the last iteration. To update the query, instead of starting at the root, we just walk and update the cut, moving it down or up the tree. For example if the querier moved, we might update this cut by walking horizontally, [transition] finding that node X does now satisfy the query...
Cuts avoid redundant work
20 - 56% faster query evaluation

To avoid redundant work when updating queries, we maintain a cut through the tree, indicating where we finished evaluating the query on the last iteration. To update the query, instead of starting at the root, we just walk and update the cut, moving it down or up the tree. For example if the querier moved, we might update this cut by walking horizontally, finding that node X does now satisfy the query...
To avoid redundant work when updating queries, we maintain a cut through the tree, indicating where we finished evaluating the query on the last iteration. To update the query, instead of starting at the root, we just walk and update the cut, moving it down or up the tree. For example if the querier moved, we might update this cut by walking horizontally, finding that node X does now satisfy the query...
The LBVH culls subtrees for efficient solid angle queries, testing 75-90% fewer nodes than a BVH.

This example doesn't make it look like a lot of savings, but they can be substantial when you have deeper trees as you avoid retesting all the internal nodes. Exploiting standing queries with cuts improves query evaluation rate by 20% with an LBVH. Aggregates, which I'll talk about next, change the way cuts work a little, resulting in even more improvement, up to 56%.
LBVH culls subtrees for efficient solid angle queries
tests 75-90% fewer nodes than a BVH

cuts ensure a complete view
20-56% more efficient

This example doesn't make it look like a lot of savings, but they can be substantial when you have deeper trees as you avoid retesting all the internal nodes. Exploiting standing queries with cuts improves query evaluation rate by 20% with an LBVH. Aggregates, which I'll talk about next, change the way cuts work a little, resulting in even more improvement, up to 56%.
LBVH culls subtrees for efficient solid angle queries
tests 75-90% fewer nodes than a BVH

cuts ensure a complete view
20-56% more efficient

static/dynamic split handles moving objects efficiently
10-15% more efficient

This example doesn't make it look like a lot of savings, but they can be substantial when you have deeper trees as you avoid retesting all the internal nodes. Exploiting standing queries with cuts improves query evaluation rate by 20% with an LBVH. Aggregates, which I'll talk about next, change the way cuts work a little, resulting in even more improvement, up to 56%.
LBVH culls subtrees for efficient solid angle queries
tests 75-90% fewer nodes than a BVH

cuts ensure a complete view
20-56% more efficient

static/dynamic split handles moving objects efficiently
10-15% more efficient

aggregates provide simplified meshes to fill in the complete view

This example doesn't make it look like a lot of savings, but they can be substantial when you have deeper trees as you avoid retesting all the internal nodes. Exploiting standing queries with cuts improves query evaluation rate by 20% with an LBVH. Aggregates, which I'll talk about next, change the way cuts work a little, resulting in even more improvement, up to 56%.
LBVH culls subtrees for efficient solid angle queries
tests 75-90% fewer nodes than a BVH

cuts ensure a complete view
20-56% more efficient

static/dynamic split handles moving objects efficiently
10-15% more efficient

aggregates provide simplified meshes to fill in the complete view
on one server...

This example doesn't make it look like a lot of savings, but they can be substantial when you have deeper trees as you avoid retesting all the internal nodes. Exploiting standing queries with cuts improves query evaluation rate by 20% with an LBVH. Aggregates, which I'll talk about next, change the way cuts work a little, resulting in even more improvement, up to 56%.
With this local query processor, we could create a world on a single server and process queries for clients. This is an important first step because it shows that it's even feasible to evaluate this type of query, but we haven't talked at all about distribution yet. I already pointed out that queries are global, which seems like an obvious scalability challenge. However, the issues are deeper than that. I'm going to talk about some more traditional distributed query processor architectures from the literature and why they don't scale well for this application. Then I'll describe our solution, the Global Aggregation Tree, which is surprising because it's a globally shared data structure, which is often a scalability hurdle rather than a scalability solution.
If we look at current systems that do scale, we almost universally find distance queries. These queries just return object identifiers for nearby objects, within some distance $d$. These queries have been deeply studied and this approach scales well because they only contact neighboring servers. This means that only a small subset of data needs to be considered to respond to a query and the world can grow effectively indefinitely. However, it results in the problems we saw where the world just seems to drop off.
what do other distributed query processors do?
We distribute query processing when we have:

a) too much data
b) too many queries
c) both

A Taxonomy by Kossmann 2000
query shipping
(too much data)
query shipping
(too much data)
query shipping
(too much data)

Client → Query → D1 → Server → Query → D2 → Server → D3 → Server → D4 → Server
query shipping
(too much data)
query shipping
(too much data)

Client -> D1 (Server)

Results

Query

D1 -> D2 (Server)
D1 -> D3 (Server)
D1 -> D4 (Server)

D2 (Server) -> Results
D3 (Server) -> Results
D4 (Server) -> Results
query shipping
(too much data)

overloads servers with “popular” objects
data shipping
(too many queries)
data shipping
(too many queries)

Client → D1 → Server → D2 → Server → D3 → Server → D4
data shipping
(too many queries)
data shipping
(too many queries)
data shipping
(too many queries)
data shipping
(too many queries)

Query

Result

Client

D1
D2
D3
Server

Replicated Data

D2
Server

D3
Server

D4
Server

global queries require the entire dataset
hybrid shipping
(too much data, too many queries)
hybrid shipping
(too much data, too many queries)
hybrid shipping
(too much data, too many queries)
hybrid shipping
(too much data, too many queries)

Client

Queries

Server

Replicated Data Subsets

D1

D2

Aggregated Query

D3

D4

Server

Server

Server
hybrid shipping
(too much data, too many queries)

Client

Queries

Server

Aggregated Query

D1

D3

D4

Results

Replicated Data Subsets

D2

Server
hybrid shipping
(too much data, too many queries)

Server

Queries

Replicated Data Subsets

results

Aggregated Query

Client

D1

D2

D3

D4

Server

Server

Server

Server

Client

Client

looks like a good balance?
So why doesn't this scale well? That's not quite the right question. It **can** scale well. Rather, the question we need to ask is, why doesn't this scale well when we include **aggregates**?
why doesn't this hybrid approach scale well?

aggregates

So why doesn't this scale well? That's not quite the right question. It can scale well. Rather, the question we need to ask is, why doesn't this scale well when we include aggregates?
hybrid shipping (PIntO)
hybrid shipping (PIntO)
hybrid shipping (PIntO)

Queries
Client

Local Tree

Server

Aggregated Query

Client

Server

Server
hybrid shipping (PIntO)

Client

Queries

Local Tree

Server

Aggregated Query

Server

Server

Client

Client

Tree

Queries

Local

Server
hybrid shipping (PiIntO)

- Client
- Local Tree
- Queries
- Server
- Replicated Data
- Server
- Aggregated Query
- Server

Client

Queries

Local Tree

Server

Replicated Data

Server

Aggregated Query
hybrid shipping (PIntO)

- Queries
- Local Tree
- Combined Tree
- Server
- Replicated Data
- Aggregate Query
- Server
- Client
hybrid shipping (PIntO)
hybrid shipping (PIntO)

Client → Queries → Local Tree

Client → Results

Server → Combined Tree

New aggregates

Server → Replicated Data

Server → Aggregated Query
<table>
<thead>
<tr>
<th>Model</th>
<th>Mesh</th>
<th>Texture</th>
<th>Generation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>39 KB</td>
<td>1,474 KB</td>
<td></td>
</tr>
<tr>
<td>House</td>
<td>16 KB</td>
<td>521 KB</td>
<td></td>
</tr>
<tr>
<td>Level 2 Aggregate</td>
<td>384 KB</td>
<td>187 KB</td>
<td>721 ms</td>
</tr>
<tr>
<td>Root Aggregate</td>
<td>1,238 KB</td>
<td>1,549 KB</td>
<td>9,431 ms</td>
</tr>
</tbody>
</table>

aggregates are expensive
hybrid shipping (PIntO)

- Client
  - Queries
  - Results
- Server
  - Local Tree
  - Combined Tree
- Aggregated Query
- Replicated Data

minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

Queries

Results

mineral query changes trigger aggregate (re)computation

Local Tree

Combined Tree

Server

Aggregated Query

Replicated Data

Client

Server

Client

Server
minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

Client

Queries

Local Tree

Combined Tree

Server

Replicated Data

Aggregated Query

Results

minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

Client

Queries

results

Local Tree

Combined Tree

Server

Aggregated Query

Replicated Data

minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

- Client
- Server

Queries
Results

Minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

Client ➔ Queries

Server ➔ Combined Tree

Local Tree ➔ Replicated Data

queries ➔ Aggregated Query

Results ➔ Client

minor query changes trigger aggregate (re)computation
hybrid shipping (PIntO)

each server generates its own combined tree based on different client queries
composing local query processors is very expensive because many aggregates must be created, stored, and downloaded

-unlike most distributed query processors, which want to optimize query performance alone, what's important to us is also the high cost of creating and using this dynamically generated data.
Global Aggregation Tree

one global tree constructed & shared by all servers

that minimizes creation of aggregates

–note that it sounds expensive to use one big shared data structure when we have a lot of nodes
Global Aggregation Tree

distributed bounding volume hierarchy

- each server has its own part of the tree from objects in its region
- one additional top-level server handles the higher-level aggregates + filtering queries to a subset of servers
Global Aggregation Tree

- each server has its own part of the tree from objects in its region
- one additional top-level server handles the higher-level aggregates + filtering queries to a subset of servers
Global Aggregation Tree

servers don’t answer queries

instead

clients answer queries, servers replicate data

–note that this sounds expensive since the whole point was to filter down what we send to the client because we have too much data already
multi-level replication

- replication process, through multiple levels. Triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. Triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
-replication process, through multiple levels. Triggers requests to get data from other servers
-Only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
-Note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- replication process, through multiple levels. Triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- replication process, through multiple levels. Triggers requests to get data from other servers.
- Only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- Note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- multi-level replication process, through multiple levels. Triggers requests to get data from other servers.
- Only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME: can we indicate this visually without it becoming too messy? Kind of critical to show the relationship of cuts across the servers...)
- Note that this means we're evaluating queries **on the client**. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries **on the client**. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. we could evaluate anything we can evaluate on the tree. better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. we could evaluate anything we can evaluate on the tree. better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries **on the client**. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- replication process, through multiple levels: triggers requests to get data from other servers.
- only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME: can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
- replication process, through multiple levels. Triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. As we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? Kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. We could evaluate anything we can evaluate on the tree. Better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
multi-level replication

- replication process, through multiple levels. triggers requests to get data from other servers
- only replicate the parts that are needed to evaluate a query. as we evaluate the query, the cut is pushed down and more data is requested (FIXME can we indicate this visually without it becoming too messy? kind of critical to show relationship of cuts across the servers...)
- note that this means we're evaluating queries on the client. we could evaluate anything we can evaluate on the tree. better yet, the centralized, expensive, hard to scale server infrastructure is now doing a lot less work, just replicating data and forwarding updates about the tree.
benefits
benefits

no new aggregates created when evaluating queries
benefits

no new aggregates created when evaluating queries
benefits

no new aggregates created when evaluating queries

simple, fine-grained replication of Global Aggregation Tree
replication protocol

commands are simple
(init, refine, coarsen, destroy)

but updates are tricky
(maintain consistent & connected tree structure, efficiently encode complex changes, e.g. move entire subtree)
evaluation:

does it scale?

how does it affect clients?
Workloads

We don’t actually have millions of users...
Workloads

We don’t actually have millions of users...

Second Life traces
(object layout, density x 256, avatar/object movement) +
Procedurally generated scenes +
Tiling
Comparison

PIntO
naïve hybrid shipping
first system design

Global Aggregation Tree
distributed query index
new system design
# of aggregates generated per server remains constant
Global Aggregation Tree creates less expensive aggregates
migration has minimal effect on GAT and client
Global Aggregation Tree

distributed query index
(single global data structure)
minimizes aggregates
Global Aggregation Tree

distributed query index
(single global data structure)
minimizes aggregates

weak consistency requirements
permit efficient pair-wise replication
Global Aggregation Tree

distributed query index
(single global data structure)
minimizes aggregates

weak consistency requirements
permit efficient pair-wise replication

multi-level replication
pushes query evaluation to clients
and allows flexible queries
Global Aggregation Tree

distributed query index
(single global data structure)
minimizes aggregates

weak consistency requirements
permit efficient pair-wise replication

multi-level replication
pushes query evaluation to clients
and allows flexible queries

prefetching policy
reduces query latency caused by chatty protocol
So finally, we come to some results! There is a bunch of detail in my dissertation about how this all gets integrated with the rest of the virtual world system, how the components act, and an overview of how the simplification works -- a new set of techniques that a colleague has developed to handle the scale and complexity of the while exploiting the inevitable reuse and regional similarity of objects in the world. Today, I'm just going to show some results, which demonstrate that we've got a full system that leverages the query processor to do rendering, handle object scripting queries. Finally I'll show that the system is actually quite a bit more general, and that we support applications far more diverse than just metaverses.
Object scripts are the other major user of querying in a metaverse system -- they need to find other objects to interact with, and then use the messaging infrastructure to carry out that interaction. It turns out that object scripts actually want the distance queries I said are awful at the beginning of the talk. This makes sense because often their interactions are (semi-)physical, requiring proximity. Because the LBVH is a superset of a BVH, this query is actually really easily implemented on the same infrastructure.

We also implemented a couple of other types of queries, just through the course of implementing the system, and interestingly, for debugging purposes. In the metaverse, these were all just variations of geometric queries. However, it turns out we can do quite a bit more with this query processing infrastructure than virtual worlds...
To demonstrate a bit of flexibility, we implemented a simple real-time Twitter visualization. We pipe in a day's worth of Tweets from the firehose, filtering to those in the US and that are geotagged. From there, we add each as an “object” to the system, build aggregates by computing trending topics and selecting a representative subset of the tweets, and build a Bloom filter of terms for querying. Then, we use a completely different client to query and render the results on a map. The user can filter by region, zooming in to see more detail, as well as add a keyword to filter the tweets with. This particular demo isn't exactly production ready, but it demonstrates the point: the global aggregation tree system is far more widely applicable than just metaverses.
Customize

record format
(mesh, tweet)

aggregate data and algorithm
(mesh, trending topic)

query format and algorithm
(solid angle, search term)

index data
(largest object, Bloom filter)

In generalizing the system we opened 4 key components for customization. First, the record format, where we're using record in the database sense. This allows completely different data sets to be integrated -- meshes and tweets in our examples. Meshes were actually URLs pointing at the mesh contents and we encoded tweets as simple JSON strings. To the system, aside from a few key geometric properties, the record is just an opaque blob of data.
In particular, the design will be good for large geometric data sets when you need to look at it at multiple resolutions, possibly with different parts at different resolutions at the same time. To get these different views you'll need to compute aggregates, and if this is expensive you want to minimize aggregate generation, just as I've described. Finally, since it deals with standing queries particularly well, it's generally best for continuous queries, which will be common in online exploration. Generally queries can be updated incrementally very efficiently.
contributions

**LBVH**
ensures complete view for a single server

**GAT**
scalable & flexible geometric queries
with aggregates

**Generalization**
multi-resolution online exploration of
large geometric datasets
THANKS