gafferongames.com

Networked Physics in Virtual Reality | Gaffer On Games

Authority Scheme

24–31 minutes

Hello readers, I'm no longer posting new content on gafferongames.com

Please check out my new blog at mas-bandwidth.com!

Introduction

About a year ago, Oculus approached me and offered to sponsor my research. They asked me, effectively: "Hey Glenn, there's a lot of interest in networked physics in VR. You did a cool talk at GDC. Do you think could come up with a networked physics sample in VR that we could share with devs? Maybe you could use the touch controllers?"

I replied ~~"F*** yes!"~~ **cough** "Sure. This could be a lot of fun!". But to keep it real, I insisted on two conditions. One: the source code I developed would be published under a permissive open source licence (for example, BSD) so it would create the most good. Two: when I was finished, I would be able to write an article describing the steps I took to develop the sample.

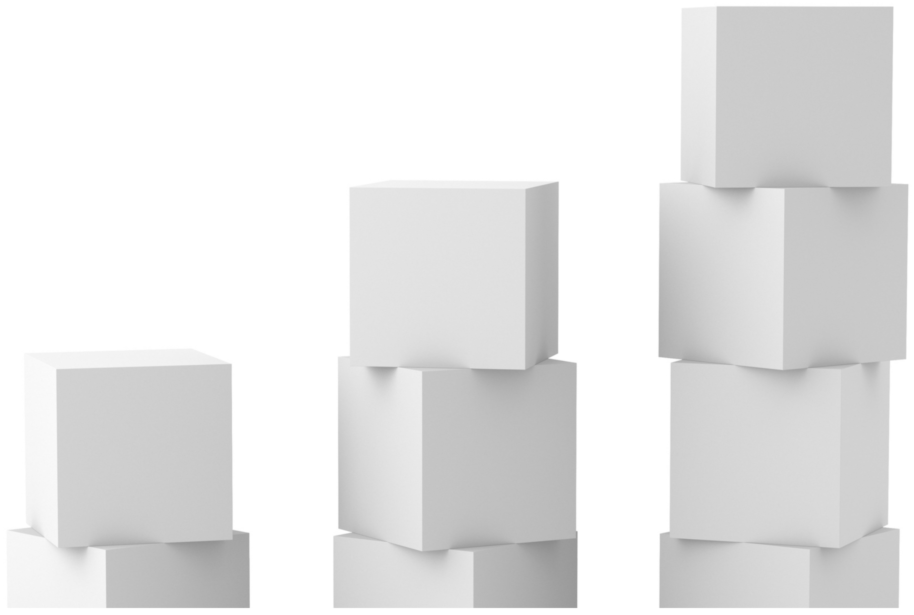
Oculus agreed. Welcome to that article! Also, the source for the networked physics sample is here, wherein the code that I wrote is released under a BSD licence. I hope the next generation of programmers can learn from my research into networked physics and create some really cool things. Good luck!

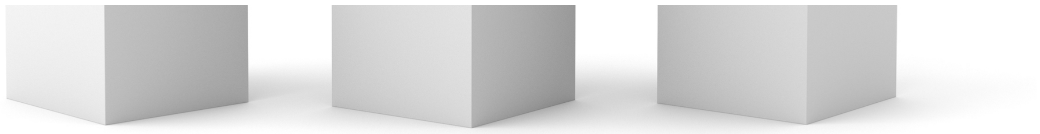
What are we building?

When I first started discussions with Oculus, we imagined creating something like a table where four players could sit around and interact with physically simulated cubes on the table. For example, throwing, catching and stacking cubes, maybe knocking each other's stacks over with a swipe of their hand.

But after a few days spent learning Unity and C#, I found myself actually *inside* the Rift. In VR, scale is *so important*. When the cubes were small, everything felt much less interesting, but when the cubes were scaled up to around a meter cubed, everything had this really cool sense of scale. You could make these *huge* stacks of cubes, up to 20 or 30 meters high. This felt really cool!

It's impossible to communicate visually what this feels like outside of VR, but it looks something like this…





… where you can select, grab and throw cubes using the touch controller, and any cubes you release from your hand interact with the other cubes in the simulation. You can throw a cube at a stack of cubes and knock them over. You can pick up a cube in each hand and juggle them. You can build a stack of cubes and see how high you can make it go.

Even though this was a lot of fun, it's not all rainbows and unicorns. Working with Oculus as a client, I had to define tasks and deliverables before I could actually start the work.

I suggested the following criteria we would use to define success:

1. Players should be able to pick up, throw and catch cubes without latency.
2. Players should be able to stack cubes, and these stacks should be stable (come to rest) and be without visible jitter.
3. When cubes thrown by any player interact with the simulation, wherever possible, these interactions should be without latency.

At the same time I created a set of tasks to work in order of greatest risk to least, since this was R&D, there was no guarantee we would actually succeed at what we were trying to do.

Network Models

First up, we had to pick a network model. A network model is basically a strategy, exactly *how* we are going to hide latency and keep the simulation in sync.

There are three main network models to choose from:

1. Deterministic lockstep
2. Client/server with client-side prediction
3. Distributed simulation with authority scheme

I was instantly confident of the correct network model: a distributed simulation model where players take over authority of cubes they interact with. But let me share with you my reasoning behind this.

First, I could trivially rule out a deterministic lockstep network model, since the physics engine inside Unity (PhysX) is not deterministic. Furthermore, even if PhysX was deterministic I could *still* rule it out because of the requirement that player interactions with the simulation be without latency.

The reason for this is that to hide latency with deterministic lockstep I needed to maintain two copies of the simulation and predict the authoritative simulation ahead with local inputs prior to render (GGPO style). At 90HZ simulation rate and with up to 250ms of latency to hide, this meant 25 physics simulation steps for each visual render frame. 25X cost is simply not realistic for a CPU intensive physics simulation.

This leaves two options: a client/server network model with client-side prediction (perhaps with dedicated server) and a less secure distributed simulation network model.

Since this was a non-competitive sample, there was little justification to incur the cost of running dedicated servers. Therefore, whether I implemented a client/server model with client-side prediction or distributed simulation model, the security would be effectively the same. The only difference would be if only one of the players in the game could theoretically cheat, or *all* of them could.

For this reason, a distributed simulation model made the most sense. It had effectively the same amount of security, and would not require any expensive rollback and resimulation, since players simply take authority over cubes they interact with and send the state for those cubes to other players.

While it makes intuitive sense that taking authority (acting like the server) for objects you interact can hide latency – since, well if you're the server, you don't experience any lag, right? – what's not immediately obvious is how to resolve conflicts.

What if two players interact with the same stack? What if two players, masked by latency, grab the same cube? In the case of conflict: who wins, who gets corrected, and how is this decided?

My intuition at this point was that because I would be exchanging state for objects rapidly (up to 60 times per-second), that it would be best to implement this as an encoding in the state exchanged between players over my network protocol, rather than as events.

I thought about this for a while and came up with two key concepts:

1. Authority
2. Ownership

Each cube would have authority, either set to default (white), or to whatever color of the player that last interacted with it. If another player interacted with an object, authority would switch and update to that player. I planned to use authority for interactions of thrown objects with the scene. I imagined that a cube thrown by player 2 could take authority over any objects it interacted with, and in turn any objects those objects interacted with, recursively.

Ownership was a bit different. Once a cube is owned by a player, no other player could take ownership until that player reliquished ownership.

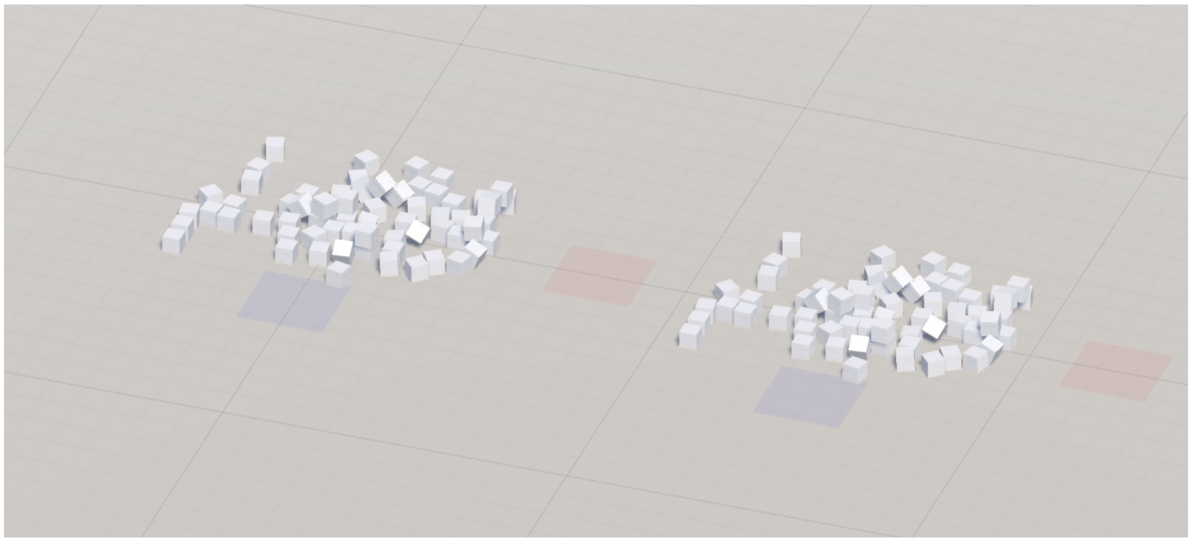
I planned to use ownership for players grabbing cubes, because I didn't want to make it possible for players to grab cubes out of other player's hands after they picked them up.

I had an intuition that I could represent and communicate authority and ownership as state by including two different sequence numbers per-cube as I sent them: an authority sequence, and an ownership sequence number. This intuition ultimately proved correct, but turned out to be much more complicated in implementation than I expected. More on this later.

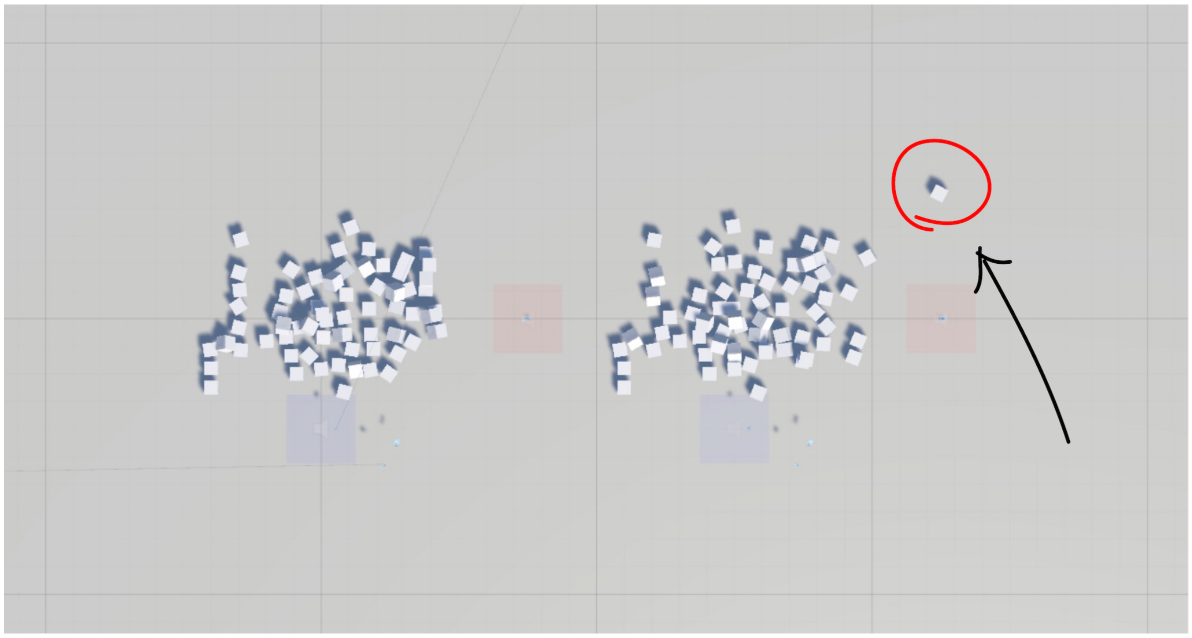
State Synchronization

Trusting I could implement the authority rules described above, my first task was to prove that synchronizing physics in one direction of flow could actually work with Unity and PhysX. In previous work I had networked simulations built with ODE, so really, I had no idea if it was really possible.

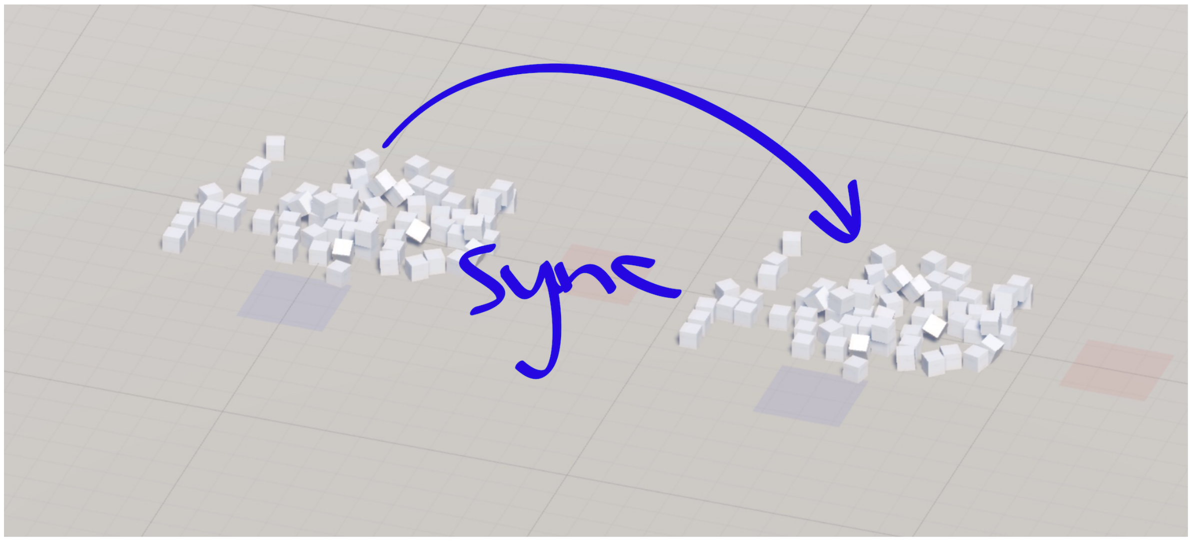
To find out, I setup a loopback scene in Unity where cubes fall into a pile in front of the player. There are two sets of cubes. The cubes on the left represent the authority side. The cubes on the right represent the non-authority side, which we want to be in sync with the cubes on the left.



At the start, without anything in place to keep the cubes in sync, even though both sets of cubes start from the same initial state, they give slightly different end results. You can see this most easily from top-down:



This happens because PhysX is non-deterministic. Rather than tilting at non-determinstic windmills, I *fight* non-determinism by grabbing state from the left side (authority) and applying it to the right side (non-authority) 10 times per-second:

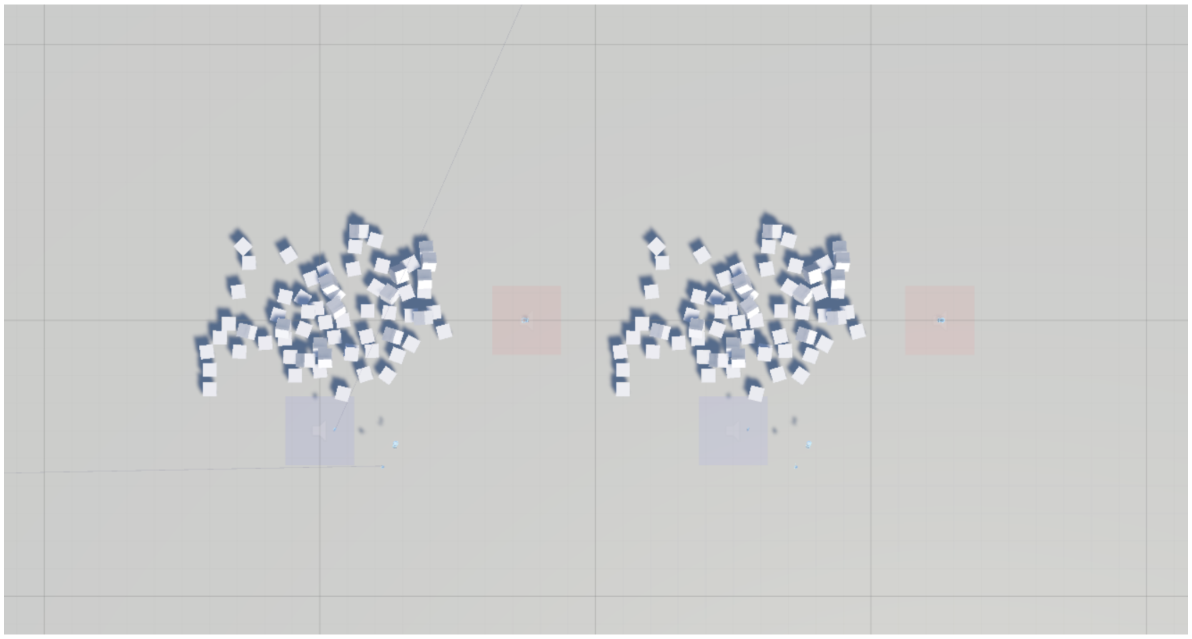


The state I grab from each cube looks like this:

```
struct CubeState
{
    Vector3 position;
    Quaternion rotation;
    Vector3 linear_velocity;
    Vector3 angular_velocity;
};
```

And when I apply this state to the simulation on the right side, I simply *snap* the position, rotation, linear and angular velocity of each cube to the state captured from the left side.

This simple change is enough to keep the left and right simulations in sync. PhysX doesn't even diverge enough in the 1/10th of a second between updates to show any noticeable pops.



This **proves** that a state synchronization based approach for networking can work with PhysX. *(Sigh of relief)*. The only problem of course, is that sending uncompressed physics state uses way too much bandwidth…

Bandwidth Optimization

To make sure the networked physics sample is playable over the internet, I needed to get bandwidth under control.

The easiest gain I found was to simply encode the state for at rest cubes more efficiently. For example, instead of repeatedly sending (0,0,0) for linear velocity and (0,0,0) for angular velocity for at rest cubes, I send just one bit:

```
[position] (vector3)
[rotation] (quaternion)
[at rest] (bool)
<if not at rest>
{
    [linear_velocity] (vector3)
    [angular_velocity] (vector3)
}
```

This is *lossless* technique because it doesn't change the state sent over the network in any way. It's also extremely effective, since statistically speaking, most of the time the majority of cubes are at rest.

To optimize bandwidth further we need to use *lossy techniques*. For example, we can reduce the precision of the physics state sent over the network by bounding position in some min/max range and quantizing it to a resolution of 1/1000th of a centimeter and sending that quantized position as an integer value in some known range. The same basic approach can be used for linear and angular velocity. For rotation I used the *smallest three representation* of a quaternion.

But while this saves bandwidth, it also adds risk. My concern was that if we are networking a stack of cubes (for example, 10 or 20 cubes placed on top of each other), maybe the quantization would create errors that add jitter to that stack. Perhaps it would even cause the stack to become *unstable*, but in a particularly annoying and hard to debug way, where the stack looks fine for you, and is only unstable in the remote view (eg. the non-authority simulation), where another player is watching what you do.

The best solution to this problem that I found was to quantize the state on *both sides*. This means that before each physics simulation step, I capture and quantize the physics state *exactly the same way* as when it's sent over the network, then I apply this quantized state back to the local simulation.

Now the extrapolation from quantized state on the non-authority side *exactly* matches the authority simulation, minimizing jitter in large stacks. At least, in theory.

Coming To ~~America~~ Rest

But quantizing the physics state created some *very interesting* side-effects!

1. PhysX doesn't really like you forcing the state of each rigid body at the start of every frame and makes sure you know by taking up a bunch of CPU.
2. Quantization adds error to position which PhysX tries very hard to correct, snapping cubes immediately out of penetration with huge pops!
3. Rotations can't be represented exactly either, again causing penetration. Interestingly in this case, cubes can get stuck in a feedback loop where they slide across the floor!
4. Although cubes in large stacks *seem* to be at rest, close inspection in the editor reveals that they are actually jittering by tiny amounts, as cubes are quantized just above surface and falling towards it.

There's not much I could do about the PhysX CPU usage, but the solution I found for the depenetration was to set *maxDepenetrationVelocity* on each rigid body, limiting the velocity that cubes are pushed apart with. I found that one meter per-second works very well.

Getting cubes to come to rest reliably was much harder. The solution I found was to disable the PhysX at rest calculation entirely and replace it with a ring-buffer of positions and rotations per-cube. If a cube has not moved or rotated significantly in the last 16 frames, I force it to rest. Boom. Perfectly stable stacks *with* quantization.

Now this might seem like a hack, but short of actually getting in the PhysX source code and rewriting the PhysX solver and at rest calculations, which I'm certainly not qualified to do, I didn't see any other option. I'm happy to be proven wrong though, so if you find a better way to do this, please let me know :)

Priority Accumulator

The next big bandwidth optimization I did was to send only a subset of cubes in each packet. This gave me fine control over the amount of bandwidth sent, by setting a maximum packet size and sending only the set of updates that fit in each packet.

Here's how it works in practice:

1. Each cube has a *priority factor* which is calculated each frame. Higher values are more likely to be sent. Negative values mean *"don't send this cube"*.
2. If the priority factor is positive, it's added to the *priority accumulator* value for that cube. This value persists between simulation updates such that the priority accumulator increases each frame, so cubes with higher priority rise faster than cubes with low priority.
3. Negative priority factors clear the priority accumulator to -1.0.

4. When a packet is sent, cubes are sorted in order of highest priority accumulator to lowest. The first n cubes become the set of cubes to potentially include in the packet. Objects with negative priority accumulator values are excluded.
5. The packet is written and cubes are serialized to the packet in order of importance. Not all state updates will necessarily fit in the packet, since cube updates have a variable encoding depending on their current state (at rest vs. not at rest and so on). Therefore, packet serialization returns a flag per-cube indicating whether it was included in the packet.
6. Priority accumulator values for cubes sent in the packet are cleared to 0.0, giving other cubes a fair chance to be included in the next packet.

For this demo I found some value in boosting priority for cubes recently involved in high energy collisions, since high energy collision was the largest source of divergence due to non-deterministic results. I also boosted priority for cubes recently thrown by players.

Somewhat counter-intuitively, reducing priority for at rest cubes gave bad results. My theory is that since the simulation runs on both sides, at rest cubes would get slightly out of sync and not be corrected quickly enough, causing divergence when other cubes collided with them.

Delta Compression

Even with all the techniques so far, it still wasn't optimized enough. With four players I really wanted to get the cost per-player down under 256kbps, so the entire simulation could fit into 1mbps for the host.

I had one last trick remaining: **delta compression**.

First person shooters often implement delta compression by compressing the entire state of the world relative to a previous state. In this technique, a previous complete world state or ‘snapshot’ acts as the *baseline*, and a set of differences, or *delta*, between the *baseline* and the *current* snapshot is generated and sent down to the client.

This technique is (relatively) easy to implement because the state for all objects are included in each snapshot, thus all the server needs to do is track the most recent snapshot received by each client, and generate deltas from that snapshot to the current.

However, when a priority accumulator is used, packets don’t contain updates for all objects and delta encoding becomes more complicated. Now the server (or authority-side) can’t simply encode cubes relative to a previous snapshot number. Instead, the baseline must be specified *per-cube*, so the receiver knows which state each cube is encoded relative to.

The supporting systems and data structures are also much more complicated:

1. A reliability system is required that can report back to the sender which packets were received, not just the most recently received snapshot #.
2. The sender needs to track the states included in each packet sent, so it can map packet level acks to sent states and update the most recently acked state per-cube. The next time a cube is sent, its delta is encoded relative to this state as a baseline.
3. The receiver needs to store a ring-buffer of received states per-cube, so it can reconstruct the current cube state from a delta by looking up the baseline in this ring-buffer.

But ultimately, it’s worth the extra complexity, because this system combines the flexibility of being able to dynamically adjust bandwidth usage, with the orders of magnitude bandwidth improvement you get from delta encoding.

Delta Encoding

Now that I have the supporting structures in place, I actually have to encode the difference of a cube relative to a previous baseline state. How is this done?

The simplest way is to encode cubes that haven't changed from the baseline value as just one bit: *not changed*. This is also the easiest gain you'll ever see, because at any time most cubes are at rest, and therefore aren't changing state.

A more advanced strategy is to encode the *difference* between the current and baseline values, aiming to encode small differences with fewer bits. For example, delta position could be (-1,+2,+5) from baseline. I found this works well for linear values, but breaks down for deltas of the smallest three quaternion representation, as the largest component of a quaternion is often different between the baseline and current rotation.

Furthermore, while encoding the difference gives some gains, it didn't provide the order of magnitude improvement I was hoping for. In a desperate, last hope, I came up with a delta encoding strategy that included *prediction*. In this approach, I predict the current state from the baseline assuming the cube is moving ballistically under acceleration due to gravity.

Prediction was complicated by the fact that the predictor must be written in fixed point, because floating point calculations are not necessarily guaranteed to be deterministic. But after a few days of tweaking and experimentation, I was able to write a ballistic predictor for position, linear and angular velocity that matched the PhysX integrator within quantize resolution about 90% of the time.

These lucky cubes get encoded with another bit: *perfect prediction*, leading to another order of magnitude improvement. For cases where the prediction doesn't match exactly, I encoded small error offset relative to the prediction.

In the time I had to spend, I not able to get a good predictor for rotation. I blame this on the smallest three representation, which is highly numerically unstable, especially in fixed point. In the future, I would not use the smallest three representation for quantized rotations.

It was also painfully obvious while encoding differences and error offsets that using a bitpacker was not the best way to read and write these quantities. I'm certain that something like a range coder or arithmetic compressor that can represent fractional bits, and dynamically adjust its model to the differences would give much better results, but I was already within my bandwidth budget at this point and couldn't justify any further noodling :)

Synchronizing Avatars

After several months of work, I had made the following progress:

- Proof that state synchronization works with Unity and PhysX
- Stable stacks in the remote view while quantizing state on both sides
- Bandwidth reduced to the point where all four players can fit in 1mbps

The next thing I needed to implement was interaction with the simulation via the touch controllers. This part was a lot of fun, and was my favorite part of the project :)

I hope you enjoy these interactions. There was a lot of experimentation and tuning to make simple things like picking up, throwing, passing from hand to hand feel good, even crazy adjustments to ensure throwing worked great, while placing objects on top of high stacks could still be done with high accuracy.

But when it comes to networking, in this case the game code doesn't count. All the networking cares about is that avatars are represented by a head and two hands driven by the tracked headset and touch controller positions and orientations.

To synchronize this I captured the position and orientation of the avatar components in *FixedUpdate* along the rest of the physics state, and applied this state to the avatar components in the remote view.

But when I first tried this it looked *absolutely awful*. Why?

After a bunch of debugging I worked out that the avatar state was sampled from the touch hardware at render framerate in *Update*, and was applied on the other machine at *FixedUpdate*, causing jitter because the avatar sample time didn't line up with the current time in the remote view.

To fix this I stored the difference between physics and render time when sampling avatar state, and included this in the avatar state in each packet. Then I added a jitter buffer with 100ms delay to received packets, solving network jitter from time variance in packet delivery and enabling interpolation between avatar states to reconstruct a sample at the correct time.

To synchronize cubes held by avatars, while a cube is parented to an avatar's hand, I set the cube's *priority factor* to -1, stopping it from being sent with regular physics state updates. While a cube is attached to a hand, I include its id and relative position and rotation as part of the avatar state. In the remote view, cubes are attached to the avatar hand when the first avatar state arrives with that cube parented to it, and detached when regular physics state updates resume, corresponding to the cube being thrown or released.

Bidirectional Flow

Now that I had player interaction with the scene working with the touch controllers, it was time to start thinking about how the second player can interact with the scene as well.

To do this without going insane switching between two headsets all the time (!!!), I extended my Unity test scene to be able to switch between the context of player one (left) and player two (right).

I called the first player the “host” and the second player the “guest”. In this model, the host is the “real” simulation, and by default synchronizes all cubes to the guest player, but as the guest interacts with the world, it takes authority over these objects and sends state for them back to the host player.

To make this work without inducing obvious conflicts the host and guest both check the local state of cubes before taking authority and ownership. For example, the host won’t take ownership over a cube already under ownership of the guest, and vice versa, while authority is allowed to be taken, to let players throw cubes at somebody else’s stack and knock it over while it’s being built.

Generalizing further to four players, in the networked physics sample, all packets flow through the host player, making the host the *arbiter*. In effect, rather than being truly peer-to-peer, a topology is chosen that all guests in the game communicate only with the host player. This lets the host decide which updates to accept, and which updates to ignore and subsequently correct.

To apply these corrections I needed some way for the host to override guests and say, no, you don’t have authority/ownership over this cube, and you should accept this update. I also needed some way for the host to determine *ordering* for guest interactions with the world, so if one client experiences a burst of lag and delivers a bunch of packets late, these packets won't take precedence over more recent actions from other guests.

As per my hunch earlier, this was achieved with two sequence numbers per-cube:

1. Authority sequence
2. Ownership sequence

These sequence numbers are sent along with each state update and included in avatar state when cubes are held by players. They are used by the host to determine if it should accept an update from guests, and by guests to determine if the state update from the server is more recent and should be accepted, even when that guest thinks it has authority or ownership over a cube.

Authority sequence increments each time a player takes authority over a cube and when a cube under authority of a player comes to rest. When a cube has authority on a guest machine, it holds authority on that machine until it receives *confirmation* from the host before returning to default authority. This ensures that the final at rest state for cubes under guest authority are committed back to the host, even under significant packet loss.

Ownership sequence increments each time a player grabs a cube. Ownership is stronger than authority, such that an increase in ownership sequence wins over an increase in authority sequence number. For example, if a player interacts with a cube just before another player grabs it, the player who grabbed it wins.

In my experience working on this demo I found these rules to be sufficient to resolve conflicts, while letting host and guest players interact with the world lag free. Conflicts requiring corrections are rare in practice even under significant latency, and when they do occur, the simulation quickly converges to a consistent state.

Conclusion

High quality networked physics with stable stacks of cubes *is* possible with Unity and PhysX using a distributed simulation network model.

This approach is best used for *cooperative experiences only*, as it does not provide the security of a server-authoritative network model with dedicated servers and client-side prediction.

Thanks to Oculus for sponsoring my work and making this research possible!

The source code for the networked physics sample can be downloaded here.

Hello readers, I'm no longer posting new content on gafferongames.com

Please check out my new blog at mas-bandwidth.com!
