Snapshot Interpolation

Interpolating between snapshots of visual state

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181424/https://gafferongames.com/about) on Sunday, November 30, 2014

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181424/https://gafferongames.com/about) and welcome to **Networked Physics (http://web.archive.org/web/20181107181424/https://gafferongames.com/categories/networked-physics/)**.

In the previous article (http://web.archive.org/web/20181107181424/https://gafferongames.com/post/deterministic_lockstep/) we networked a physics simulation using deterministic lockstep. Now, in this article we're going to network the same simulation with a completely different technique: **snapshot interpolation**.

Background

While deterministic lockstep is very efficient in terms of bandwidth, it's not always possible to make your simulation deterministic. Floating point determinism across platforms is hard (http://web.archive.org/web/20181107181424/https://gafferongames.com/post/floating_point_determinism/).

Also, as the player counts increase, deterministic lockstep becomes problematic: you can't simulate frame n until you receive input from *all* players for that frame, so players end up waiting for the most lagged player. Because of this, I recommend deterministic lockstep for 2-4 players at most.

So if your simulation is not deterministic or you want higher player counts then you need a different technique. Snapshot interpolation fits the bill nicely. It is in many ways the polar opposite of deterministic lockstep: instead of running two simulations, one on the left and one on the right, and using perfect determinism and synchronized inputs keep them in sync, snapshot interpolation doesn't run any simulation on the right side at all!

Snapshots

Instead, we capture a **snapshot** of all relevant state from the simulation on the left and transmit it to the right, then on the right side we use those snapshots to reconstruct a visual approximation of the simulation, all without running the simulation itself.

As a first pass, let's send across the state required to render each cube:

```
        vec3f position;
        quat4f orientation;
    };
```

I'm sure you've worked out by now that the cost of this technique is increased bandwidth usage. Greatly increased bandwidth usage. Hold on to your neckbeards, because a snapshot contains the visual state for the entire simulation. With a bit of math we can see that each cube serializes down to 225 bits or 28.1 bytes. Since there are 900 cubes in our simulation that means each snapshot is roughly 25 kilobytes. That's pretty big!

At this point I would like everybody to relax, take a deep breath, and imagine we live in a world where I can actually send a packet this large 60 times per-second over the internet and not have everything explode. Imagine I have FIOS (I *do*), or I'm sitting over a backbone link to another computer that is also on the backbone. Imagine I live in South Korea. Do whatever you need to do to suspend disbelief, but most of all, don't worry, because I'm going to spend the entire next article showing you how to optimize snapshot bandwidth.

When we send snapshot data in packets, we include at the top a 16 bit sequence number. This sequence number starts at zero and increases with each packet sent. We use this sequence number on receive to determine if the snapshot in a packet is newer or older than the most recent snapshot received. If it's older then it's thrown away.

Each frame we just render the most recent snapshot received on the right:

Look closely though, and even though we're sending the data as rapidly as possible (one packet per-frame) you can still see hitches on the right side. This is because the internet makes no guarantee that packets sent 60 times per-second arrive nicely spaced $^1/_{60}$ of a second apart. Packets are jittered. Some frames you receive two snapshot packets. Other frames you receive none.

Jitter and Hitches

because over the LAN those packets actually do tend to arrive at the same rate they were sent… and then you start trying to play your game over wireless or the internet and you start seeing hitches. Don't worry. There are ways to handle this!

First, let's look at how much bandwidth we're sending with this naive approach. Each packet is 25312.5 bytes plus 28 bytes for IP + UDP header and 2 bytes for sequence number. That's 25342.5 bytes per-packet and at 60 packets per-second this gives a total of 1520550 bytes per-second or 11.6 megabit/sec. Now there are certainly internet connections out there that can support that amount of traffic… but since, let's be honest, we're not really getting a lot of benefit blasting packets out 60 times per-second with all the jitter, let's pull it back a bit and send only 10 snapshots per-second:

You can see how this looks above. Not so great on the right side but at least we've reduced bandwidth by a factor of six to around 2 megabit/sec. We're definitely headed in the right direction.

Linear Interpolation

Now for the trick with snapshots. What we do is instead of immediately rendering snapshot data received is that we buffer snapshots for a short amount of time in an interpolation buffer. This interpolation buffer holds on to snapshots for a period of time such that you have not only the snapshot you want to render but also, statistically speaking, you are very likely to have the next snapshot as well. Then as the right side moves forward in time we interpolate between the position and orientation for the two slightly delayed snapshots providing the illusion of smooth movement. In effect, we've traded a small amount of added latency for smoothness.

You may be surprised at just how good it looks with linear interpolation @ 10pps:

Look closely though and you can see some artifacts on the right side. The first is a subtle position jitter when the player cube is hovering in the air. This is your brain detecting 1st order discontinuity at the sample points of position interpolation. The other artifact occurs when a bunch of cubes are in a katamari ball, you can see a sort of "pulsing" as the speed of rotation increases and decreases. This occurs because attached cubes interpolate linearly between two sample points rotating around the player cube, effectively interpolating *through* the player cube as they take the shortest linear path between two points on a circle.

Hermite Interpolation

I find these artifacts unacceptable but I don't want to increase the packet send rate to fix them. Let's see what we can do to make it look better at the same send rate instead. One thing we can try is upgrading to a more accurate interpolation scheme for position, one that interpolates between position samples while considering the linear velocity at each sample point.

This can be done with an hermite spline (http://web.archive.org/web/20181107181424/https://en.wikipedia.org/wiki/Hermite_interpolation) (pronounced "air-mitt")

Unlike other splines with control points that affect the curve indirectly, the hermite spline is guaranteed to pass through the start and end points while matching the start and end velocities. This means that velocity is smooth across sample points and cubes in the katamari ball tend to rotate around the cube rather than interpolate through it at speed.

Above you can see hermite interpolation for position @ 10pps. Bandwidth has increased slightly because we need to include linear velocity with each cube in the snapshot, but we're able to significantly increase the quality at the same send rate. I can no longer see any artifacts. Go back and compare this with the raw, non-interpolated 10pps version. It really is amazing that we're able to reconstruct the simulation with this level of quality at such a low send rate.

As an aside, I found it was not necessary to perform higher order interpolation for orientation quaternions to get smooth interpolation. This is great because I did a lot of research into exactly interpolating between orientation quaternions with a specified angular velocity at sample points and it seemed difficult. All that was needed to achieve an acceptable result was to switch from linear interpolation + normalize (nlerp) to spherical linear interpolation (slerp) to ensure constant angular speed for orientation interpolation.

I believe this is because cubes in the simulation tend to have mostly constant angular velocity while in the air and large angular velocity changes occur only discontinuously when collisions occur. It could also be because orientation tends to change slowly while in the air vs. position which changes rapidly relative to the number of pixels affected on screen. Either way, it seems that slerp is good enough and that's great because it means we don't need to send angular velocity in the snapshot.

Handling Real World Conditions

Now we have to deal with packet loss. After the discussion of UDP vs. TCP in the previous article I'm sure you can see why we would never consider sending snapshots over TCP.

Snapshots are time critical but unlike inputs in deterministic lockstep snapshots don't need to be reliable. If a snapshot is lost we can just skip past it and interpolate towards a more recent snapshot in the interpolation buffer. We don't ever want to stop and wait for a lost snapshot packet to be resent. This is why you should always use UDP for sending snapshots.

I'll let you in on a secret. Not only were the linear and hermite interpolation videos above recorded at a send rate of 10 packets per-second, they were also recorded at 5% packet loss with +/- 2 frames of jitter @ 60fps. How I handled packet loss and jitter for those videos is by simply ensuring that snapshots are held in the interpolation buffer for an appropriate amount of time before interpolation.

works best at 2-5% packet loss is 3X the packet send rate. At 10 packets per-second this is 300ms. I also need some extra delay to handle jitter, which in my experience is typically only one or two frames @ 60fps, so the interpolation videos above were recorded with a delay of 350ms.

Adding 350 milliseconds delay seems like a lot. And it is. But, if you try to skimp you end up hitching for 1/10th of a second each time a packet is lost. One technique that people often use to hide the delay added by the interpolation buffer in other areas (such as FPS, flight simulator, racing games and so on) is to use extrapolation. But in my experience, extrapolation doesn't work very well for rigid bodies because their motion is non-linear and unpredictable. Here you can see an extrapolation of 200ms, reducing overall delay from 350 ms to just 150ms:

Problem is it's just not very good. The reason is that the extrapolation doesn't know anything about the physics simulation. Extrapolation doesn't know about collision with the floor so cubes extrapolate down through the floor and then spring back up to correct. Prediction doesn't know about the spring force holding the player cube up in the air so it the cube moves slower initially upwards than it should and has to snap to catch up. It also doesn't know anything about collision and how collision response works, so the cube rolling across the floor and other cubes are also mispredicted. Finally, if you watch the katamari ball you'll see that the extrapolation predicts the attached cubes as continuing to move along their tangent velocity when they should rotate with the player cube.

Conclusion

You could conceivably spend a great deal of time to improve the quality of this extrapolation and make it aware of various movement modes for the cubes. You could take each cube and make sure that at minimum the cube doesn't go through the floor. You could add some approximate collision detection or response using bounding spheres between cubes. You could even take the cubes in the katamari ball and make them predict motion to rotate around with the player cube.

But even if you do all this there will still be misprediction because you simply can't accurately match a physics simulation with an approximation. If your simulation is mostly linear motion, eg. fast moving planes, boats, space ships – you may find that a simple extrapolation works well for short time periods (50-250ms or so), but

How can we reduce the amount of delay added for interpolation? 350ms still seems unacceptable and we can't use extrapolation to reduce this delay without adding a lot of inaccuracy. The solution is simple: *increase the send rate*! If we send 30 snapshots per-second we can get the same amount of packet loss protection with a delay of 150ms. 60 packets per-second needs only 85ms.

In order to increase the send rate we're going to need some pretty good bandwidth optimizations. But don't worry, there's a *lot* we can do to optimize bandwidth. So much so that there was too much stuff to fit in this article and I had to insert an extra unplanned article just to cover all of it!

NEXT ARTICLE: Snapshot Compression (http://web.archive.org/web/20181107181424/https://gafferongames.com/post/snapshot_compression/)

● (http://web.archive.org/web/20181107181424/https://www.linkedin.com/in/glennfiedler/)

● (http://web.archive.org/web/20181107181424/https://twitter.com/gafferongames)

● (http://web.archive.org/web/20181107181424/https://github.com/gafferongames)